sitepoint

Unleashing the Power of TypeScript

Unleashing the Power of TypeScript

Ebook ISBN: 978-1-925836-58-5

- **Author:** Steve Kinney
- **Technical Editor:** Sebastian Springer
- **Product Manager:** Simon Mackie
- **English Editor:** Ralph Mason
- **Cover Designer:** Volodymyr Boyarinov

Notice of Rights

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Published by SitePoint Pty. Ltd.

71 Balmain Street, Cremorne, VIC, 3121
Australia
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

About the Author

Steve Kinney is the head of engineering for frontend and developer tools at Temporal and an instructor with Frontend Masters. Previously, Steve founded the front-end engineering program at the Turing School of Software and Design and was a New York City public school teacher for the better part of a decade.

Preface

Who Should Read This Book?

This book is for intermediate-level JavaScript developers who want to get to grips with the powerful features that TypeScript offers.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.
</p>
```

You’ll notice that we’ve used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- https://www.sitepoint.com/community/ are SitePoint’s forums, for help on any tricky problems.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

Chapter 1: Simplifying Reducers in React with TypeScript

Many of the example React applications we see tend to be small and easily understandable. But in the real world, React apps tend to grow to a scale that makes it impossible for us to keep all of the context in our head. This can often lead to unexpected bugs when a given value isn't what we thought it would be. That object you thought had a certain property on it? It turns out it ended up being `undefined` somewhere along the way. That function you passed in? Yeah, that's `undefined` too. That string you're trying to match against? It looks like you misspelled it one night when you were working after hours.

Using TypeScript in a React application enhances code quality by providing **static type checking**, which catches errors early in the development process. It also improves readability and maintainability through explicit type annotations, making it easier for teams to understand the code structure. Additionally, TypeScript features like interfaces and generics make it easier to build robust, scalable applications.

Large applications also tend to come with increasingly complicated state management. **Reducers** are a powerful pattern for managing state in client-side applications. Reducers became popular with Redux, and are now built into React with the `useReducer` hook, but they're framework and language agnostic. At the end of the day, a reducer is just a function. Redux and React's `useReducer` just add some additional functionality to trigger updates accordingly. We can use Redux with any frontend framework or without one all together. We could also write our own take on Redux pretty easily, if we were so inclined.

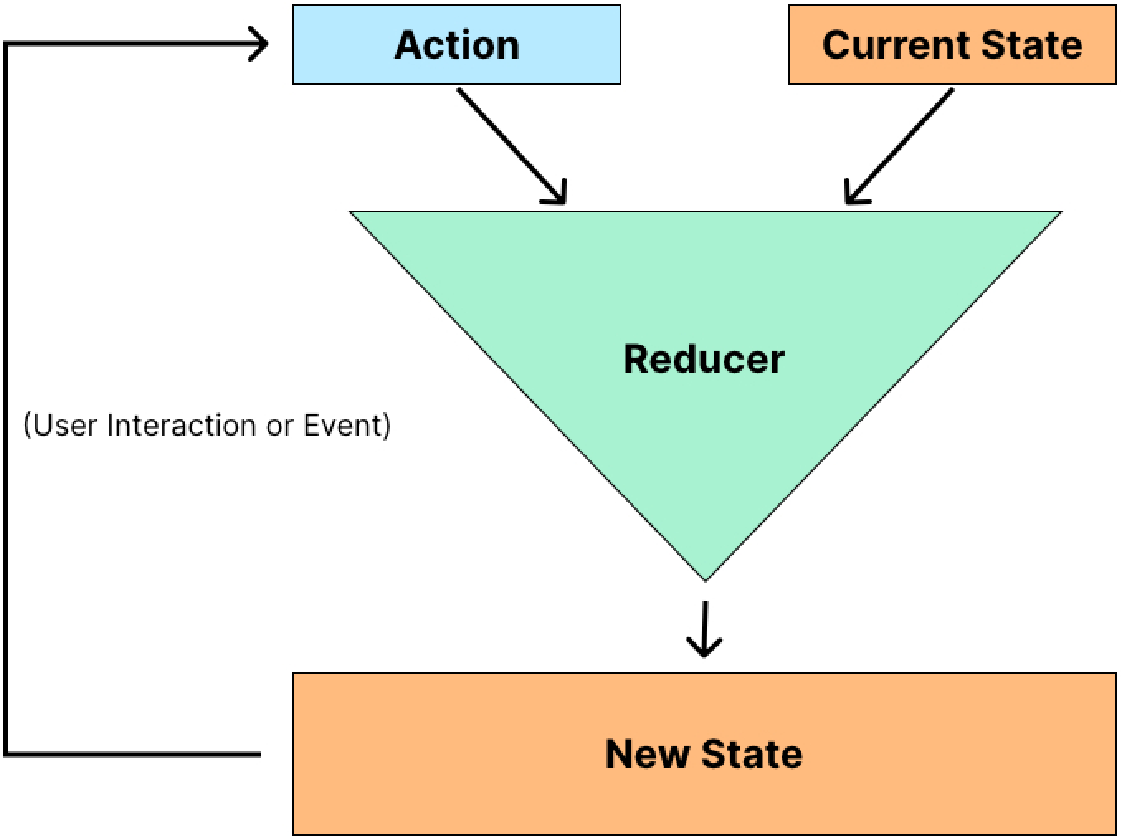
That said, Redux (and other implementations of the Flux architecture that it’s based on) often get criticized for requiring a lot of boilerplate code, and for being a bit cumbersome to use. At the risk of making an unintentional pun, we can leverage TypeScript not only to reduce the amount of boilerplate required, but also to make the overall experience of using reducers more pleasant.

Following Along with This Tutorial

You’re welcome to follow along with this tutorial in your own environment. I’ve also created a GitHub repository that you can clone, as well as a CodeSandbox demo that you can use to follow along.

Reducer Basics

A **reducer**, at its most fundamental level, is simply a function that takes two arguments: the current state, and an object that represents some kind of action that has occurred. The reducer returns the new state based on that action.



The following code is regular JavaScript, but it could easily be converted to TypeScript by adding `any` types to `state` and `action`:

```
export const incrementAction = { type: 'Increment' };
export const decrementAction = { type: 'Decrement' };

export const counterReducer = (state, action) => {
  if (action.type === 'Increment') {
    return { count: state.count + 1 };
  }

  if (action.type === 'Decrement') {
    return { count: state.count - 1 };
  }

  return state;
};
```

```
let state = { count: 1 };

state = counterReducer(state, incrementAction);
state = counterReducer(state, incrementAction);
state = counterReducer(state, decrementAction);

console.log(state); // Logs: { count: 1 }
```

Repo Code

You can find the code above (`01-basic-example`) in the GitHub repo for this tutorial.

Let’s review what’s going on in the code sample above:

- We pass in the current state and an action.
- If the action has a `type` property of `"Increment"`, we increment the `count` property on `state`.
- If the action has a `type` property of `"Decrement"`, we decrement the `count` property on `state`.
- If neither of the above two bullet points is true, we do nothing and return the original state.

Redux requires an action to be an object with a `type` property, as shown in the example above. React isn’t as strict. An action can be anything—even a primitive value like a number. For example, this is a valid reducer when using `useReducer`:

```
const counterReducer = (state = 0, value = 1) => state + value;
```

In fact, if you’ve used `useState` in React before, you’ve really used `useReducer`. `useState` is simply an abstraction of `useReducer`. For our purposes, we’ll stick to a model closer to Redux, where actions are objects with a `type` property, since that pattern will work almost universally.

The `useReducer` hook of Redux and React adds some extra functionality around emitting changes and telling React to update the state of a component accordingly—as opposed to repeatedly setting a variable, as shown above—but the basic principles are the same regardless of what library we're using.

Adding Types to Our Reducer

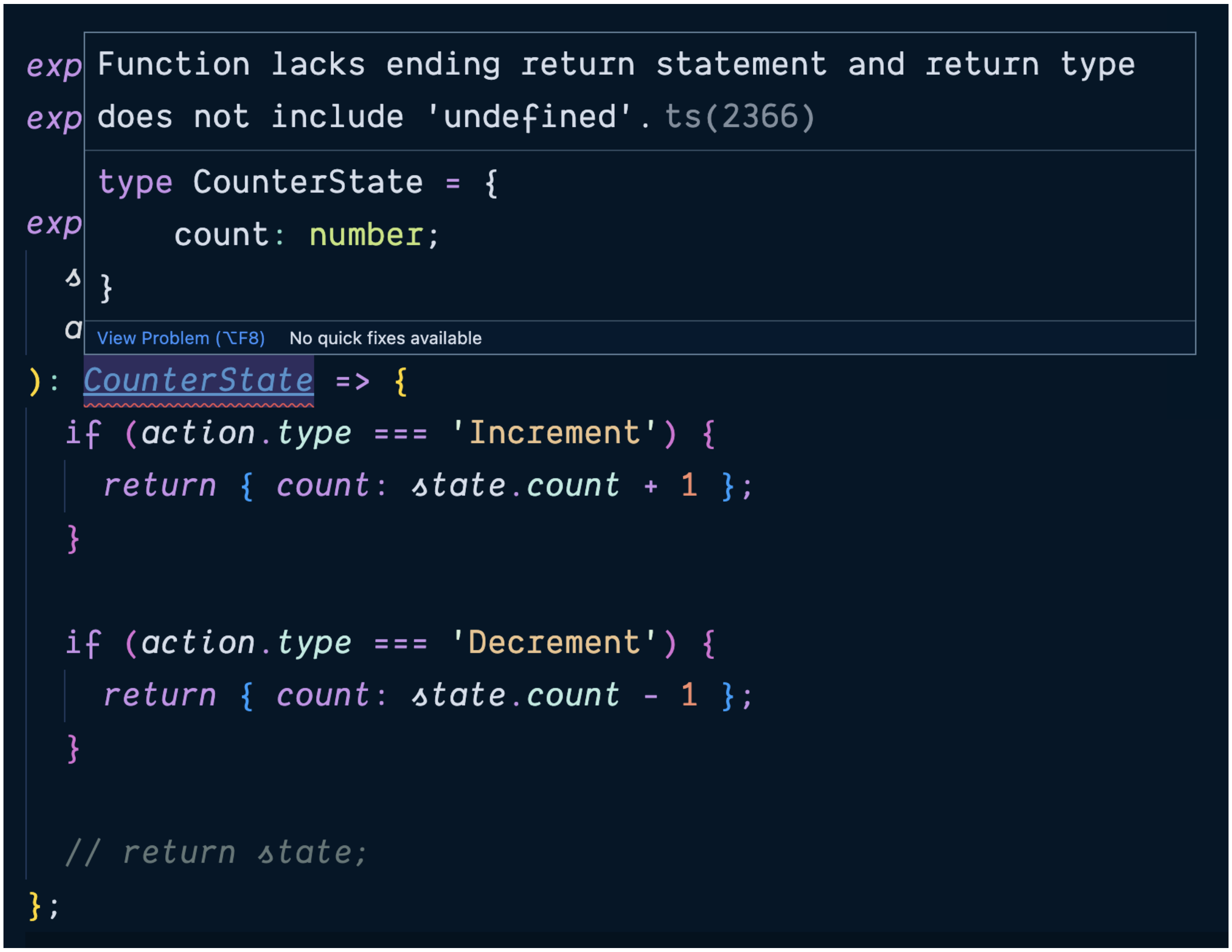
Adding TypeScript will protect us from some of the more obvious mistakes—by making sure that we both pass in the correct arguments and return the expected result. Both of these situations can be a bit tricky to triage in our applications, because they might not happen until *after* the user takes an action—like clicking a button in the UI. Let's quickly add some types to our reducer:

```
type CounterState = { count: number };
type CounterAction = { type: string };

export const incrementAction = { type: 'Increment' };
export const decrementAction = { type: 'Decrement' };

export const counterReducer = (
  state: CounterState,
  action: CounterAction,
): CounterState => {
  if (action.type === 'Increment') {
    return { count: state.count + 1 };
  }

  if (action.type === 'Decrement') {
    return { count: state.count - 1 };
  }

  return state;
};
```

Repo Code

You can find the code above (`02-reducer-with-types`) in the GitHub repo for this tutorial.

Let’s say that we forget to return `state` when none of the actions match any of our conditionals. TypeScript has analyzed our code and has been to told to expect that `counterReducer` will always return some kind of `CounterState`. If there’s even so much as a possibility that our code won’t behave as expected, it will refuse to compile. In this case, TypeScript has seen that there’s a mismatch between what we expect our code to do and what it actually does.



```
Function lacks ending return statement and return type does not include 'undefined'. ts(2366)
type CounterState = {
    count: number;
}
```

```
): CounterState => {
  if (action.type === 'Increment') {
    return { count: state.count + 1 };
  }

  if (action.type === 'Decrement') {
    return { count: state.count - 1 };
  }

  // return state;
};
```

We’ll also get the other protections we’ve come to expect from TypeScript, such as making sure we pass in the correct arguments and only access properties available on those objects.

But there’s a more insidious (and arguably more common) edge case that comes up when working with reducers. In fact, it’s one that I encountered when I was writing tests for the initial example at the beginning of this tutorial. What happens if we misspell the action type?

```
let state: CounterState = { count: 0 };

state = counterReducer(state, { type: 'increment' });
state = counterReducer(state, { type: 'INCREMENT' });
state = counterReducer(state, { type: 'Increement' });

console.log(state); // Logs: { count: 0 }
```

This the worst kind of bug, because it doesn’t cause an error. It just silently doesn’t do what we expect. One common solution is to store the action type names in constants:

```
export const INCREMENT = 'Increment';
export const DECREMENT = 'Decrement';
```

This is where the typical boilerplate begins. Since JavaScript can’t protect us from accidentally using a string that doesn’t match any of the conditionals in our reducer, we assign these strings to constants. Misspelling a constant or variable name will prevent our code from compiling and make it obvious that we have an issue.

Luckily, when we’re using TypeScript, we can avoid this kind of boilerplate altogether.

Adding Payloads to Actions

It’s common for actions to contain additional information about what happened. For example, a user might type a query into a search field and click **Submit**. We would want to know what they searched for in addition to knowing they clicked the **Submit** button.

There are no hard and fast rules for how to structure an action—other than Redux’s insistence that we include a `type` property. But it’s a good practice to follow some kind of standard like [Flux Standard Action](), which advises us put to any additional information needed in a `payload` property.

Following this convention, our `CounterAction` might look something like this:

```
type CounterAction = {
  type: string;
  payload: {
    amount: number;
  };
};

let state = counterReducer(
  { count: 1 },
  { type: 'Increment', payload: { amount: 1 } },
);
```

This is getting a bit complicated to type out. A common solution is to create a set of helper functions called *action creators*. **Action creators** are simply functions that format our actions for us. If we wanted to expand `counterReducer` to support the ability to increment or decrement by certain amounts, we might create the following action creators:

```
export const increment = (amount: number = 1): CounterAction =>
({
  type: INCREMENT,
  payload: { amount },
});

export const decrement = (amount: number = 1): CounterAction =>
({
  type: DECREMENT,
  payload: { amount },
});
```

Repo Code

You can find the code above (`03-action-creators`) in the GitHub repo for this tutorial.

We’re also going to need to update our reducer to support this new structure for our actions. This also feels like a good opportunity to look at the example holistically:

```
export const INCREMENT = 'Increment';
export const DECREMENT = 'Decrement';

type CounterState = { count: number };

type CounterAction = {
  type: string;
  payload: {
    amount: number;
  };
};

type CounterReducer = (
  state: CounterState,
  action: CounterAction,
) => CounterState;

export const increment = (amount: number = 1): CounterAction =>
({
  type: INCREMENT,
  payload: { amount },
});

export const decrement = (amount: number = 1): CounterAction =>
({
  type: DECREMENT,
  payload: { amount },
});

export const counterReducer: CounterReducer = (state, action) =>
{
  const { count } = state;

  if (action.type === 'Increment') {
    return { count: count + action.payload.amount };
  }

  if (action.type === 'Decrement') {
    return { count: count - action.payload.amount };
  }

  return state;
};

let state: CounterState = { count: 0 };
```

Using Unions

But wait, there’s more! We used some of the traditional patterns to get around the issue where our reducer ignores actions that it wasn’t explicitly told to look for, but what if we could use TypeScript to prevent that from happening in the first place?

Let’s assume that, in addition to being able to increment and decrement the counter, we can also reset it back to zero. This gives us three types of actions:

- `Increment`
- `Decrement`
- `Reset`

We’ll start with `Increment` and `Decrement` and address `Reset` later.

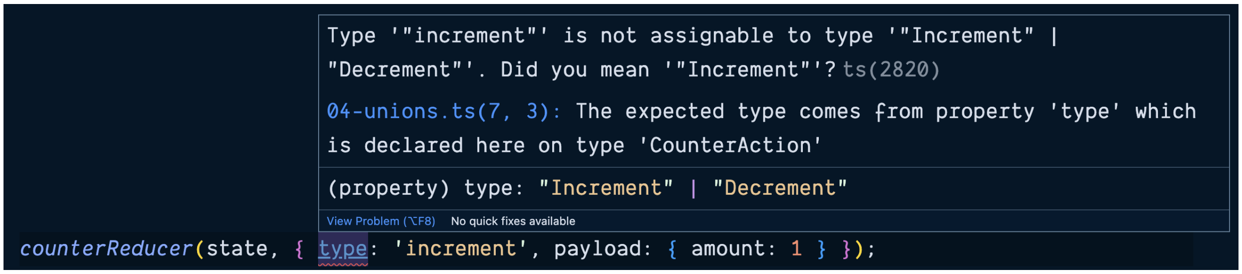
In the last section, we added the ability to increment or decrement by a certain amount. Reseting the counter will be a bit unusual — in that we can only ever reset it back to zero. (Sure, I could have just created a `Set` action that took a value, but I’m setting myself up to make a more important point in a bit.)

Let’s start with our two known quantities: `Increment` and `Decrement`. Instead of saying that the `type` property on an action can be any string, we can get a bit more specific.

In `04-unions` in our GitHub repo, I use the union of `'Increment' | 'Decrement'`. We’re now telling TypeScript that the `type` property on a `CounterAction` isn’t just *any* string, but rather that it’s one of exactly two strings:

```
type CounterAction = {
  type: 'Increment' | 'Decrement';
  payload: {
    amount: number;
  };
};
```

We get a number of benefits from this relatively simple change. The first and most obvious is that we no longer have to worry about misspelling or mistyping an action’s `type`.

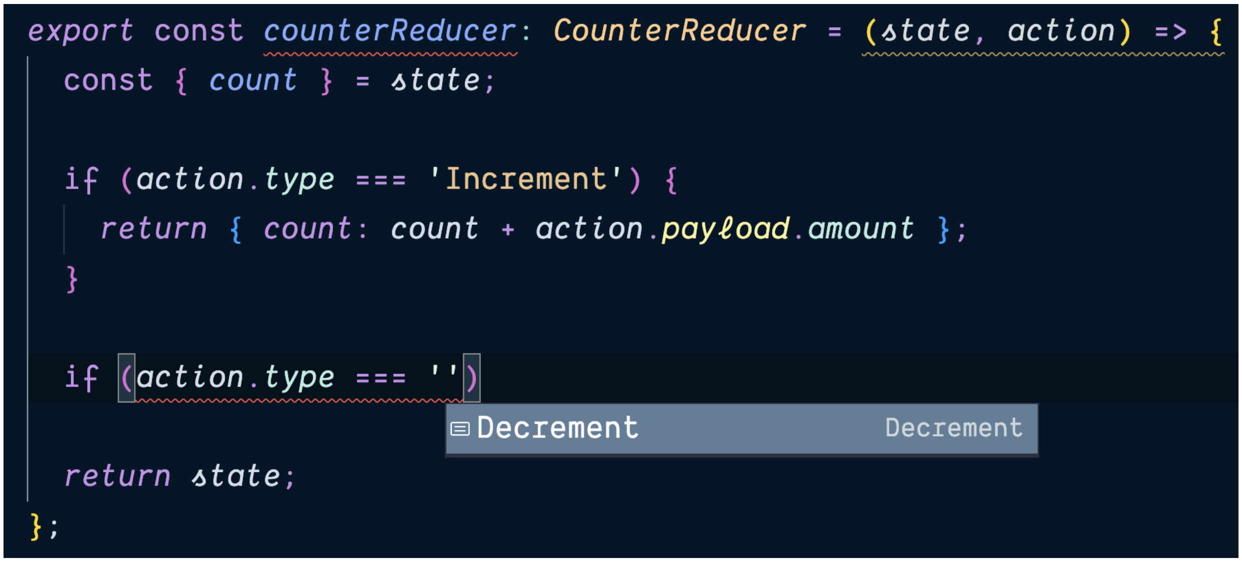


```
Type '"increment"' is not assignable to type '"Increment" |
"Decrement"'. Did you mean '"Increment"'? ts(2820)

04-unions.ts(7, 3): The expected type comes from property 'type' which
is declared here on type 'CounterAction'

(property) type: "Increment" | "Decrement"
```

```
counterReducer(state, { type: 'increment', payload: { amount: 1 } });
```

You’ll notice that TypeScript not only detects the error, but it’s even smart enough to provide a suggestion that can help us quickly address the issue.

We also get autocomplete for free whenever TypeScript has enough information to determine that we’re working with a `CounterAction`.



```
export const counterReducer: CounterReducer = (state, action) => {
  const { count } = state;

  if (action.type === 'Increment') {
    return { count: count + action.payload.amount };
  }

  if (action.type === '')

  return state;
};
```

If you look carefully, you'll notice that it's *only* recommending `Decrement`. That's because we've already created a conditional defining what we should do in the event that the `type` is `Increment`. TypeScript is able to deduce that, if there are only two properties and we've dealt with one of them, there's only one option left.

As promised, we can also now get rid of these two lines of code from `03-action-creators`:

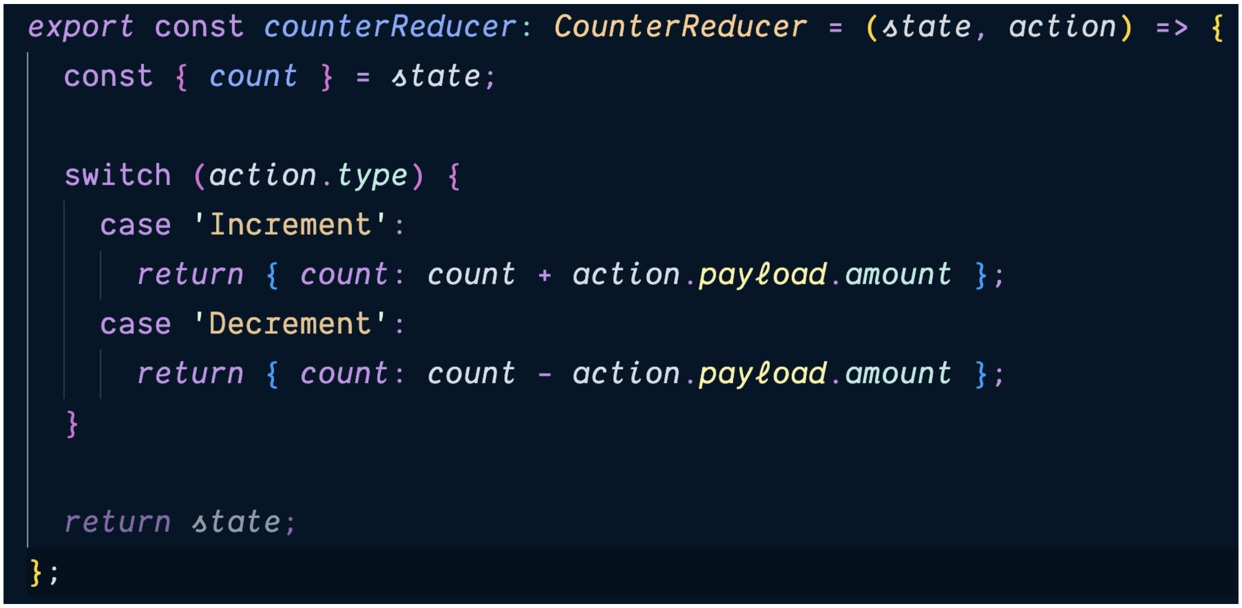
```
- export const INCREMENT = 'Increment';
- export const DECREMENT = 'Decrement';
```

With TypeScript's help, we can now go back to using regular strings and still enjoy all of the benefits of using constants. This might not seem like much in this simple example, but if you've ever used this pattern before, you know that it can get cumbersome to have to import these values in each and every file that uses them.

Removing the Default Case in the Reducer

Earlier on, TypeScript tried to help us by throwing an error when we omitted the line with `return state` at the end of the function that served as a fallback if none of the conditions above it were hit.

But now we’ve given TypeScript more information about what types of actions it can expect. I prefer to use conditionals (which is why I’ve done so throughout this tutorial), but if we use a `switch` statement instead, we’ll notice that something interesting happens.



```
export const counterReducer: CounterReducer = (state, action) => {
  const { count } = state;

  switch (action.type) {
    case 'Increment':
      return { count: count + action.payload.amount };
    case 'Decrement':
      return { count: count - action.payload.amount };
  }

  return state;
};
```

TypeScript has figured out that, since we’re returning from each case of the `switch` statement *and* we’ve covered all of the possible cases of `action.type`, there’s no need to return the original state in the event that our action slips through, because TypeScript can guarantee that will never happen.

There are a few things to take away from this:

- TypeScript will use the information we provide it to help us avoid common mistakes.
- TypeScript will also use this information to enable us to write less protective code.

Dealing with Different Kinds of Actions

Earlier, I hinted that we might add a third type of action: the ability to `Reset` the counter back to zero. This action is a lot like the actions we saw at the very beginning. It doesn't need a payload. However, it might be tempting to do something like this:

```
type CounterAction = {
  type: 'Increment' | 'Decrement' | 'Reset';
  payload: {
    amount: number;
  };
};
```

You'll notice that TypeScript is again upset that we risk returning `undefined` from our reducer. We'll handle that in a moment. But first, we should address the fact that `Reset` doesn't need a payload.

We don't want to have to write something like this:

```
let state = counterReducer({ count: 5 }, { type: 'Reset', {
payload: { amount: 0 } } });
```

We might be tempted to make the `payload` optional:

```
type CounterAction = {
  type: 'Increment' | 'Decrement' | 'Reset';
  payload?: {
    amount: number;
  };
};
```

But now, TypeScript will never be sure if `Increment` and `Decrement` are supposed to have a payload. This means that TypeScript will insist that we check to see if `payload` exists before we're allowed to access the `amount` property on it. At the same time, TypeScript will still allow us to needlessly put a payload on our `Reset` actions. I suppose this isn’t the worst thing in the world, but we can do better.

Using Unions for Action Types

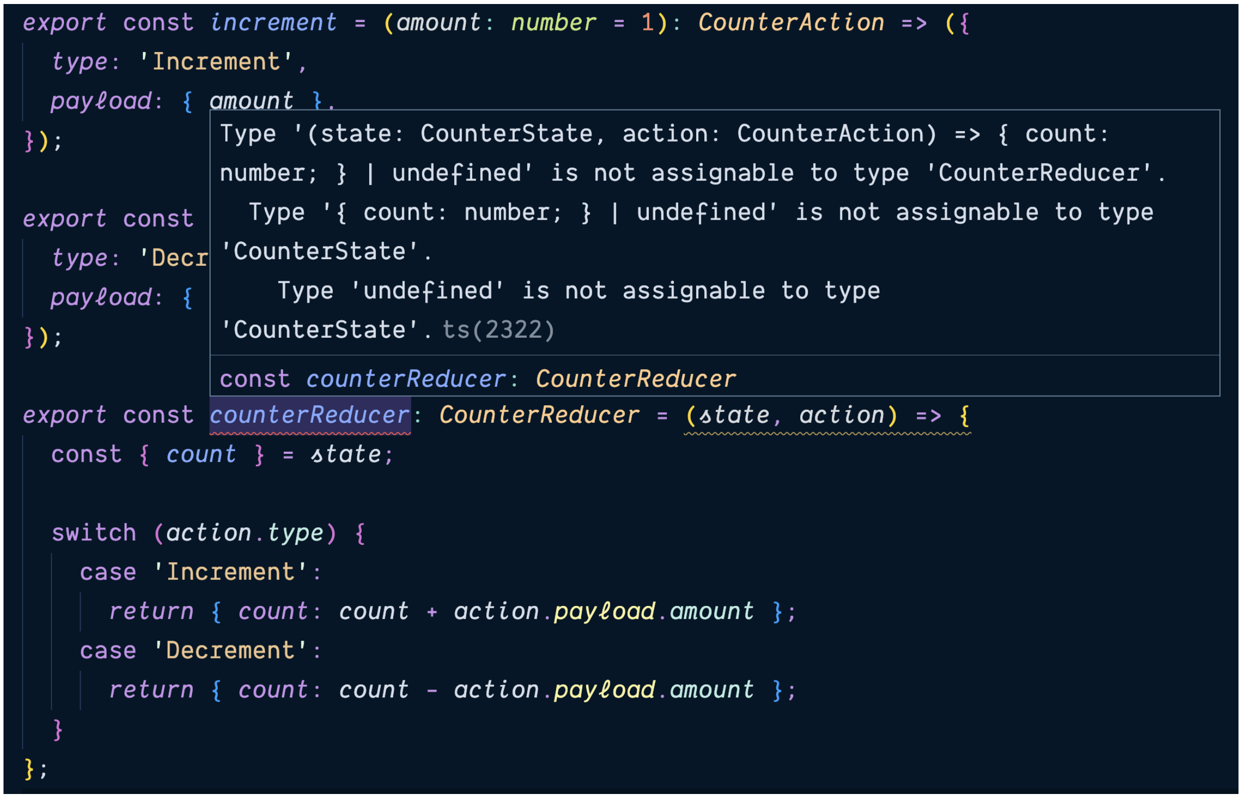
It turns out that we can use a similar solution to the one we used with the `type` property on `CounterAction`:

```
type CounterAdjustmentAction = {
  type: 'Increment' | 'Decrement';
  payload: {
    amount: number;
  };
};

type CounterResetAction = {
  type: 'Reset';
};

type CounterAction = CounterAdjustmentAction |
CounterResetAction;
```

TypeScript is smart enough to figure out the following:

- `CounterAction` is an object.
- `CounterAction` *always* has a `type` property.
- The `type` property is one of `Increment`, `Decrement`, or `Reset`.
- If the `type` property is `Increment` or `Decrement`, there’s a `payload` property that contains a number as the `amount`.
- If the `type` property is `Reset`, there’s no `payload` property.

By updating the type to include `CounterResetAction`, TypeScript has already figured out that we’re no longer providing an exhaustive list of cases to `counterReducer`.



```
export const increment = (amount: number = 1): CounterAction => ({
  type: 'Increment',
  payload: { amount },
});

export const
  type: 'Decr
  payload: {
});

export const counterReducer: CounterReducer = (state, action) => {
  const { count } = state;

  switch (action.type) {
    case 'Increment':
      return { count: count + action.payload.amount };
    case 'Decrement':
      return { count: count - action.payload.amount };
  }
};
```

```
Type '(state: CounterState, action: CounterAction) => { count:
number; } | undefined' is not assignable to type 'CounterReducer'.
  Type '{ count: number; } | undefined' is not assignable to type
'CounterState'.
    Type 'undefined' is not assignable to type
'CounterState'. ts(2322)

const counterReducer: CounterReducer
```

We can update the code:

```
export const counterReducer: CounterReducer = (state, action) =>
{
  const { count } = state;

  switch (action.type) {
    case 'Increment':
      return { count: count + action.payload.amount };
    case 'Decrement':
      return { count: count - action.payload.amount };
    case 'Reset':
      return { count: 0 };
  }
};
```

Repo Code

You can find the code above (`05-different-payloads`) in the GitHub repo for this tutorial.

If you’ve been following along, you might have noticed a few cool features (albeit unsurprising at this point). First, as we added that third case, TypeScript was able to infer that we were adding a case for `Reset` and suggested that as the only available autocompletion. Secondly, if we tried to reference `action` in the `return` statement, we would have noticed that it *only* let us access the `type` property because it’s well aware that `CounterResetActions` doesn’t have a `payload` property.

Other than clearly defining the type, we didn’t have to tell TypeScript much of anything in the code itself. It was able to use the information at hand to figure everything out on our behalf.

If you want to see this for yourself, you can create an action creator for resetting the counter:

```
export const reset = (): CounterAction => ({ type: 'Reset' });
```

I chose to use the broader `CounterAction` in this case. But you’ll notice that, even if you *try* to add a `payload` to it, TypeScript has already figured out that it’s not an option. But if you change the `type` to `"Increment"` for a moment, you’re suddenly permitted to add the property.

If we update the return type on the function to `CounterResetAction`, we’ll see that we only have one option for the type—`"Reset"`—and that payloads are forbidden:

```
export const reset = (): CounterResetAction => ({ type: 'Reset'
});
```

Using Our Reducer in a React Component

So far, we’ve talked a lot about the reducer pattern outside of any framework. Let’s pull our `counterReducer` into React and see how it works. We'll start with this simple component:

```
const Counter = () => {
  return (
    <main className="mx-auto w-96 flex flex-col gap-8 items-center">
      <h1>Counter</h1>
      <p className="text-7xl">0</p>
      <div className="flex place-content-between w-full">
        <button>Decrement</button>
        <button>Reset</button>
        <button>Increment</button>
      </div>
    </main>
  );
};

export default Counter;
```

Repo Code

You can find the code above (`06-react-component`) in the GitHub repo for this tutorial.

You might notice that there isn't much happening just yet. The `counterReducer` that we've been working on throughout this tutorial is ready for action. We just need to hook it up to the component, using the `useReducer` hook.

In the context of a reducer in React—or Redux, for that matter—`state` is the current snapshot of our component's data. `dispatch` is a function used to update that state based on actions. We can think of `dispatch` as the way to trigger changes, and the `state` as the resulting data after those changes.

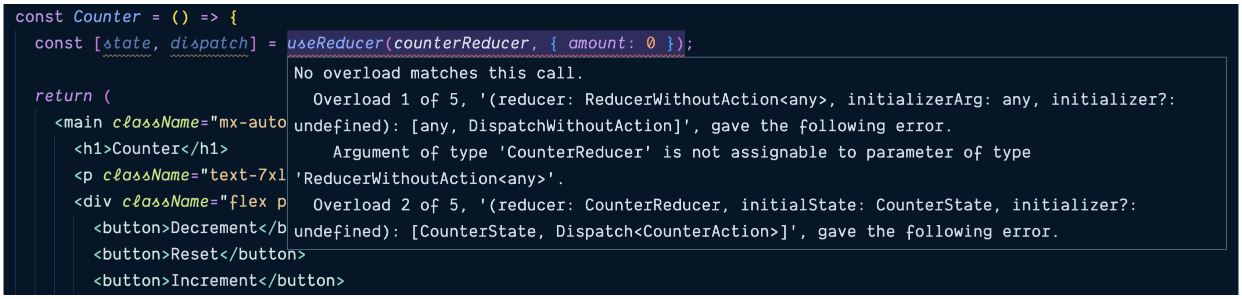
Let's add the following code inside the `Counter` component:

```
const [state, dispatch] = useReducer(counterReducer, { count: 0 });
```

Now, if we hover over `state` and `dispatch`, we'll see that TypeScript is able to automatically figure out what the correct types of each is:

```
const state: CounterState;
const dispatch: React.Dispatch<CounterAction>;
```

It's able to infer this from the type annotations on `counterReducer` itself. Similarly, it also will only allow us to pass in an initial state that matches `CounterState`—although the error isn't nearly as helpful. If we change the `count` property to `amount` in the initial state given to `useReducer`, we'll see a result similar to that pictured below.



```
const Counter = () => {
  const [state, dispatch] = useReducer(counterReducer, { amount: 0 });

  return (
    <main className="mx-auto
      <h1>Counter</h1>
      <p className="text-7xl
      <div className="flex p
        <button>Decrement</b
        <button>Reset</button>
        <button>Increment</button>
```

```
No overload matches this call.
  Overload 1 of 5, '(reducer: ReducerWithoutAction<any>, initializerArg: any, initializer?:
undefined): [any, DispatchWithoutAction]', gave the following error.
    Argument of type 'CounterReducer' is not assignable to parameter of type
'ReducerWithoutAction<any>'.
  Overload 2 of 5, '(reducer: CounterReducer, initialState: CounterState, initializer?:
undefined): [CounterState, Dispatch<CounterAction>]', gave the following error.
```

We can now use the `state` and `dispatch` from `useReducer` in our `Counter` component. TypeScript will know that `count` is a property on `state`, and it will only accept actions or the values returned from action creators that match the `CounterAction` type:

```
import { useReducer } from 'react';
import {
  counterReducer,
  increment,
  decrement,
  reset,
} from './05-different-payloads';

const Counter = () => {
  const [state, dispatch] = useReducer(counterReducer, { count:
0 });
```

```
  return (
    <main className="mx-auto w-96 flex flex-col gap-8 items-center">
      <h1>Counter</h1>
      <p className="text-7xl">{state.count}</p>
      <div className="flex place-content-between w-full">
        <button onClick={() => dispatch(decrement())}>Decrement</button>
        <button onClick={() => dispatch(reset())}>Reset</button>
        <button onClick={() => dispatch(increment())}>Increment</button>
      </div>
    </main>
  );
};

export default Counter;
```

Repo Code

You can find the code above (`07-react-component-complete`) in [the GitHub repo for this tutorial]().

I’d like to draw your attention to just how little TypeScript is in this component. In fact, if we were to change the file extension for `.tsx` to `.jsx`, it would still work. But behind the scenes, TypeScript is doing the important work of ensuring that our application will work as expected when we put it in the hands of our users.

Conclusion

The power of reducers is not in their inherent complexity but in their simplicity. It’s important to remember that reducers are just functions. In the past, they’ve received a bit of flack for requiring a fair amount of boilerplate.

TypeScript helps to reduce the amount of boilerplate, while also making the overall experience of using reducers a lot more pleasant. We’re protected from potential pitfalls in the form of incorrect arguments and unexpected results, which can be tricky to troubleshoot in our applications.

The combination of reducers and TypeScript can make it super easy to build resilient, error-free applications.

Don’t forget to refer to the GitHub repository and CodeSandbox demo to play around with any of the examples discussed above.

In the next part of this book, we’ll look at taking some of the mystery out of using generics in TypeScript. Generics allow us to write reusable code that works with multiple types, rather than a single one. It acts as a placeholder for the type, letting us write code that can adapt and enforce type consistency at compile time.

Chapter 2: A Gentle Introduction to Generics in TypeScript

If you’ve come from a language with a type system, you might already be familiar with generics. But if you’re like me and your experience consists predominately of dynamic languages like JavaScript, Python, or Ruby, generics might seem a bit bewildering at first. I’m certainly guilty of just copying and pasting examples without fully understanding what’s going on. But fear not! Generics are both powerful and also fairly simple once we understand them. Let’s start at the basics and build up from there, as we work towards a full understanding what generics are and how we can use them in our TypeScript code.

In short, a **generic** in TypeScript allows us to write reusable code that works with more than one type. Generics act as placeholders for types, letting us write code that can adapt and enforce type consistency at compile time. It’s not wrong to think of generics as variables, but just for our types.

Following Along with This Tutorial

All of the code that we’ll be working with can be found in this GitHub repository. I also encourage you to use the TypeScript Playground to try out ideas and play with the examples below.

What Are Generics?

Generics in TypeScript offer a means for creating reusable code components that work with a variety of types (rather than just one). As I mentioned, we can think of them as variables—but for our types. Generics provide a way to create flexible structures that allow us to define the type at a later time.

Let’s start with a simple linked list. A **linked list** node contains a value and then a link to the next node. We could start out with something link this:

```
type LinkedListNode = {
  value: any;
  next?: LinkedListNode;
};
```

However, I’m going to add an extra requirement. All of the nodes in the list should be of the same type. For example, if we wanted a linked list of strings, we could refine our type to make sure that value is always a string:

```
type LinkedListNode = {
  value: string;
  next?: LinkedListNode;
};

const firstNode: LinkedListNode = {
  value: 'first',
};

const secondNode: LinkedListNode = {
  value: 'second',
};

firstNode.next = secondNode;
```

Repo Code

You can find the code above (`01-without-generics.ts`) in the GitHub repo for this tutorial.

You might have already noticed the problem here. If we wanted to also support a linked list of numbers, we'd have to create another type, and so on for every single type of value we wanted our linked list to support. This is tedious, and it doesn't scale.

This is where generics come in. They let us define a placeholder that we can use in our types. If we wanted to have a linked list node, where the value could be any type, but that would ensure all of the later nodes were also of the *same* type, we could write something along these lines:

```
type LinkedListNode<T> = {
  value: T;
  next?: LinkedListNode<T>;
};

const firstNode: LinkedListNode<string> = {
  value: 'first',
};

const secondNode: LinkedListNode<string> = {
  value: 'second',
};

const thirdNode: LinkedListNode<number> = {
  value: 42,
};

const fourthNode: LinkedListNode<number> = {
  value: 17,
};

firstNode.next = secondNode;
thirdNode.next = fourthNode;
```

Repo Code

You can find the code above (`02-with-generics`) in the GitHub repo for this tutorial.

When we set `T` to either `string` or `number`, TypeScript uses that type everywhere that `T` is referenced.

An Experiment

If you want to run a little experiment, you can try to connect the `thirdNode` to `firstNode` in this playground. You'll see that TypeScript won't allow it.

A Word on Naming Generics

You might be wondering why I called my generic `T`. I mostly did this because it's convention. `T` stands for "type". If there's more than one generic in a type, you might sometimes see `U` used. Why? because U is the letter after T.

Generics don't need to be single letters. We could have easily used a longer name:

```
type LinkedListNode<NodeType> = {
  value: NodeType;
  next?: LinkedListNode<NodeType>;
};
```

Typically, though, we just see single letters used. For example, React's types will use `P` when referring to props.

If we need to use multiple generics, we can separate them with a comma, just like we would do for arguments in a function:

```
type Badge<L extends string, C extends number> {
  label: L,
  count: C
}
```

Extending Types in Generics

Right now, the T generic can be anything. In the example above, we used a string and a number, but we could have just as easily used a Boolean, an array, an object, or even a function.

Let’s say we wanted to limit the value of our linked list node to only support strings, numbers, and Booleans.

In TypeScript, constraints are expressed using the `extends` keyword. `T extends K` means it’s safe to presume that a value of type `T` is also of type `K`. An easy way of thinking about this is to just imagine we’re writing something with `extends` as a variable assignment.

Let’s take `19 extends number` as an example. We could think about it like this:

```
const nineteen: number = 19;
```

We can say that `T` has to extend another type:

```
type LinkedListNode<T extends string | number | boolean> = {
  value: T;
  next?: LinkedListNode<T>;
};
```

Repo Code

You can find the code above (`03-extending-types`) in the GitHub repo for this tutorial.

Now, our generic is limited to anything that meets the criteria of being a string, number, or Boolean. We can get a little bit silly with this. For example, we could say a node can only support a particular string or a specific number:

```
const firstNode: LinkedListNode<string> = {
  value: 'first',
};

const secondNode: LinkedListNode<'second'> = {
  value: 'second',
};

const thirdNode: LinkedListNode<number> = {
  value: 42,
};

const fourthNode: LinkedListNode<17> = {
  value: 17,
};

firstNode.next = secondNode;
thirdNode.next = fourthNode;
fourthNode.next = {
  value: 17,
};
```

The type `number` refers to any number. `17` is a literal type that only allows for that specific value. In this case, anything connected to `secondNode` would *have* to have a value of `"second"`, and anything connected to `fourthNode` would *have* to have a value of `17`. Each technically does extend `string` and `number` respectively. That said, I'm only doing this as a demonstration. I'm not sure why we'd want to do this in practice.

A more practical example would be if we wanted to only support a certain subset of strings or numbers. For example, we could limit the acceptable strings using a union, as seen in this demo:

```
type LinkedListNode<T extends 'loading' | 'success' | 'error'>
= {
  value: T;
  next?: LinkedListNode<T>;
};
```

Obviously, it’s getting a bit tedious to constantly have to define the type for each variable.

Using Generics in Functions

It’s a bit more common to see generics used with functions. Let’s consider an identity function that takes a value and returns it. I think we’re pretty clear at this point that we don’t want to do something like this:

```
const identity = (x: string): string => {
  return x;
};
```

Just as with the linked list, I don’t have time to create one of these for every type I come across. This sounds like a perfect use case for a generic:

```
export const identity = <T>(x: T): T => {
  return x;
};

export const foo = identity<string>('foo');
```

A Word on Using Generics with JSX

We might find that the syntax above won’t work if we’re using React and/or JSX. This is because the parser has a hard time differentiating between the syntax for a generic and a JSX component. We can see it failing in this playground, and we can see it compiling just fine with JSX turned off here.

To get around this, we have two options. We could simply add a comma after the generic:

```
const identity = <T>(x: T): T => {
  return x;
};
```

Or, we could exclusively use function declarations:

```
function identity<T>(x: T): T {
  return x;
}
```

Inferring Generics

TypeScript is always trying to get out of our way. If it can figure out what a generic ought to be, it will relieve us of the responsibility of having to tell it. Let’s take a look at the following code:

```
export const identity = <T>(x: T): T => {
  return x;
};

export const foo = identity<string>('foo');
export const two = identity(2);
```

Repo Code

You can find the code above (`04-inferring-generics`) in the GitHub repo for this tutorial.

In this code, we’re able to completely omit telling TypeScript what type `T` should be. It sees that we’re handing that function call a number, so it’s able to figure out that `T` is a number in this case.

We can also leverage this to simplify our linked list above:

```
type LinkedListNode<T> = {
  value: T;
  next?: LinkedListNode<T>;
};
```

```
export const identity = <T>(x: T): T => {
  return x;
};

export const foo = identity<string>('foo');
export const two = identity(2);

const createLinkedListNode = <T>(value: T): LinkedListNode<T>
=> {
  return { value };
};

const firstNode = createLinkedListNode('first');
const secondNode = createLinkedListNode('second');

firstNode.next = secondNode;
```

You'll notice that we no longer need to define what type of `LinkedListNode` we're expecting. As soon as TypeScript can narrow down the type for one reference to `T`, it can fill in all of the rest. `firstNode` and `secondNode` have the following types:

```
const firstNode: LinkedListNode<string>;
const secondNode: LinkedListNode<string>;
```

This gives us all the flexibility of a generic without needing to type any additional boilerplate.

Taking It a Step Further

Now that we have `createLinkedListNode`, we can create an additional utility function called `addNextNode`. The function should do the following:

- It should take an existing `LinkedListNode` as the first argument.
- Based on the type of the `value` of that `LinkedListNode`, the second argument should be a value of the same type.
- It should create a new `LinkedListNode` based on the second argument.

- It should set the `next` property on the first `LinkedListNode` to the new `LinkedListNode`.
- It should return the newly created node.

Basically, we want to make the following two tests pass:

```
it('should create a linked list node', () => {
  let node = createLinkedListNode('first');
  expect(node).toEqual({ value: 'first' });
});

it('should add a linked list node', () => {
  let firstNode = createLinkedListNode('first');
  let secondNode = addLinkedListNode(firstNode, 'second');

  expect(firstNode).toEqual({ value: 'first', next: { value:
'second' } });
  expect(secondNode).toEqual({ value: 'second' });
});
```

The key piece here is that the value of `T` should be the same throughout. There are a few ways we could write the function, and we won’t go over the merits of each. Here’s one possible implementation:

```
const createLinkedListNode = <T>(value: T): LinkedListNode<T>
=> {
  return { value };
};

const addLinkedListNode = <T>(
  node: LinkedListNode<T>,
  value: T,
): LinkedListNode<T> => {
  node.next = createLinkedListNode(value);
  return node.next;
};
```

Repo Code

You can find the code above (`05-linking-nodes`) in the GitHub repo for this tutorial.

Using Generics with Classes

We can also use generics with classes. The interesting feature in this case is that the value of `T` can be used throughout the class. Once it’s been set, it will enforce our type throughout all of its methods, and each individual instance of the class can have its own value for `T`. Here’s an example:

```
export class LinkedListNode<T> {
  value: T;
  next?: LinkedListNode<T>;

  constructor(value: T) {
    this.value = value;
  }

  add(value: T): LinkedListNode<T> {
    this.next = new LinkedListNode(value);
    return this.next;
  }
}
```

Repo Code

You can find the code above (`06-with-classes`) in the GitHub repo for this tutorial.

Now we can create multiple instances where `T` is correctly enforced for that particular instance:

```
const firstNode = new LinkedListNode('first');
const secondNode = firstNode.add('second');
const thirdNode = new LinkedListNode(42);
const fourthNode = thirdNode.add(17);
```

You can verify this in the unit tests found here.

Conclusion

I prefer to use types, but we can just as easily use generics with interfaces as well:

```
interface LinkedListNode<T> {
  value: T;
  next?: LinkedListNode<T>;
}
```

Generics are a powerful tool in TypeScript that allow us to create reusable code components that work with a variety of types, rather than just one. They provide a way to build flexible interfaces that enable us to define the type at a later time. Generics can be used with objects, functions, and classes. They can be extended or limited to specific types, and TypeScript can even infer the type for us in some cases.

While they may seem daunting at first, generics are fairly simple once you understand them. With a little practice, you’ll be able to use generics to write more efficient and effective code in no time flat.

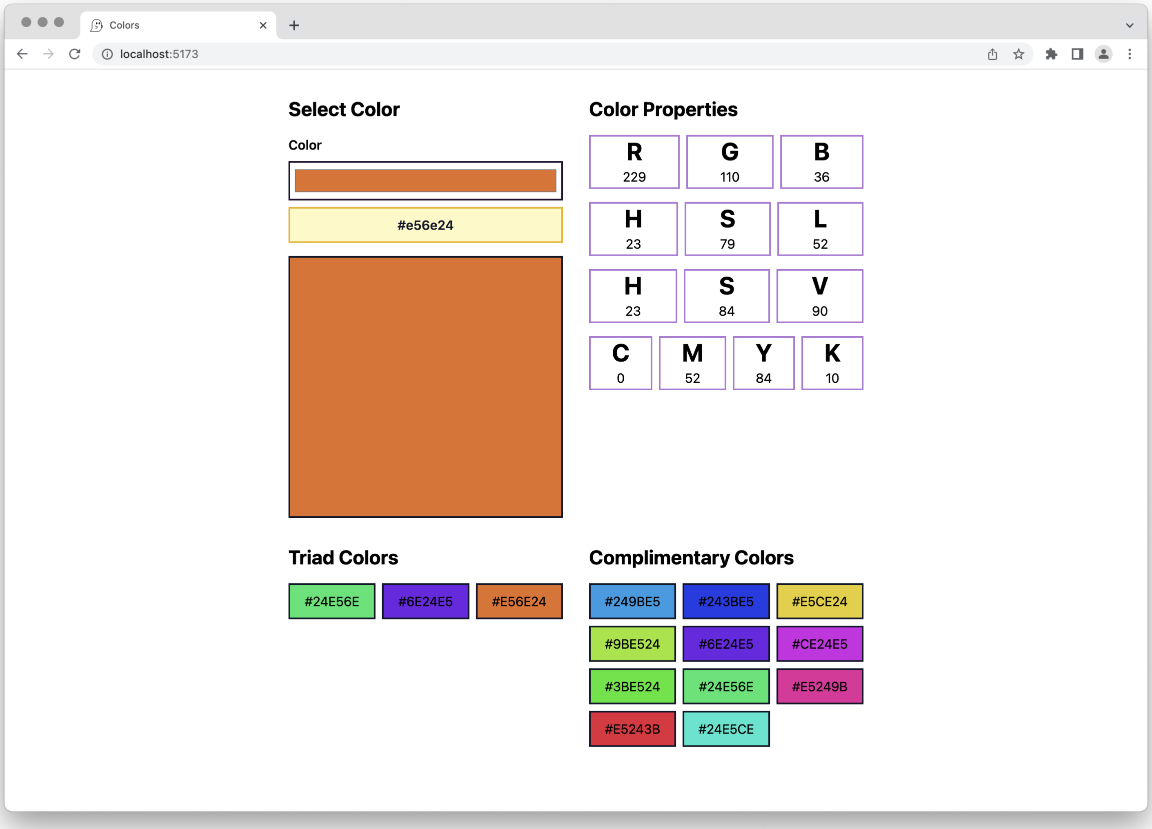
Chapter 3: Navigating the Perils and Pitfalls of Using React's Context API with TypeScript

Using TypeScript in any large React application provides a level of safety that's worth the additional effort over using JavaScript. TypeScript is able to comb through all of our components and confirm that we're passing in all the props each component requires, and it makes sure sure they're actually the types we assume they are and not something nefarious like `undefined`. But there are some cases where even that trade-off is thrown into question—particularly when it comes to using the Context API. Let's look at how to navigate some of the challenges of using `createContext` and `useContext` with TypeScript.

I'm going to assume you're familiar with the Context API, but let's do a quick summary anyway. The Context API in React is a built-in state management tool that allows for easy sharing of data throughout a component tree. It enables us to avoid **prop drilling**, the process of passing data from top to bottom through intermediate components.

With the Context API, we create a `Context` object that has two properties: a `Provider` component—which is used to wrap the part of the component tree of our application that needs access to our data—and a `Consumer` component for reading that data.

These days, we often don't use the `Consumer` directly, instead using the `useContext` hook to access the context's value, regardless of the level of each in the component hierarchy. The `useContext` hook is a much cleaner abstraction than trying to access the value from the `Consumer` in a render prop. This makes state and function sharing between components way more convenient and efficient, especially in larger applications. In fact, the React documentation goes as far as to mention that the `Consumer` is "an alternative and rarely used way to read the context value."



Select Color

Color

#e56e24

Color Properties

R	G	B
229	110	36

H	S	L
23	79	52

H	S	V
23	84	90

C	M	Y	K
0	52	84	10

Triad Colors

#24E56E	#6E24E5	#E56E24

Complimentary Colors

#249BE5	#243BE5	#E5CE24
#9BE524	#6E24E5	#CE24E5
#3BE524	#24E56E	#E5249B
#E5243B	#24E5CE	

We're going to work on a simple color picker application (pictured above) that uses the Context API to share the ability to get and set the color throughout the component hierarchy.

Following Along with This Tutorial

All of the code that we’ll be working with can be found in [this GitHub repository](). I’ll start on the `main` branch and then make additional branches at each milestone.

The Fundamental Problem

The core issue that’s going to be a thorn in our side is not necessarily a TypeScript problem. It’s a React problem:

- When we call `React.createContext`, TypeScript will want to know what kind of value we can expect.
- But we don’t have that value yet, because we have to wait for the `Provider` component to mount in order to use `useState`.
- TypeScript will take issue with the fact that we’re passing it something that isn’t what it expects.
- If we say that it can be a color *or* `undefined`, TypeScript will never be sure which one it’s working with.

We’ll look at a few approaches—such as using `any` or `undefined`—which will work, but which will have some trade-offs—before we land on a better (though more involved) solution.

A Rejected Solution: Use `any`

```
// We don't have access to `hexColor` outside of `ColorProvider`.
// So we'll use `null` for now.
export const ColorContext = createContext<any>(null);

export const ColorProvider = ({ children }: PropsWithChildren) => {
  const [hexColor, setHexColor] = useState('#e56e24');

  return (
    <ColorContext.Provider value={{ hexColor, setHexColor }}>
```

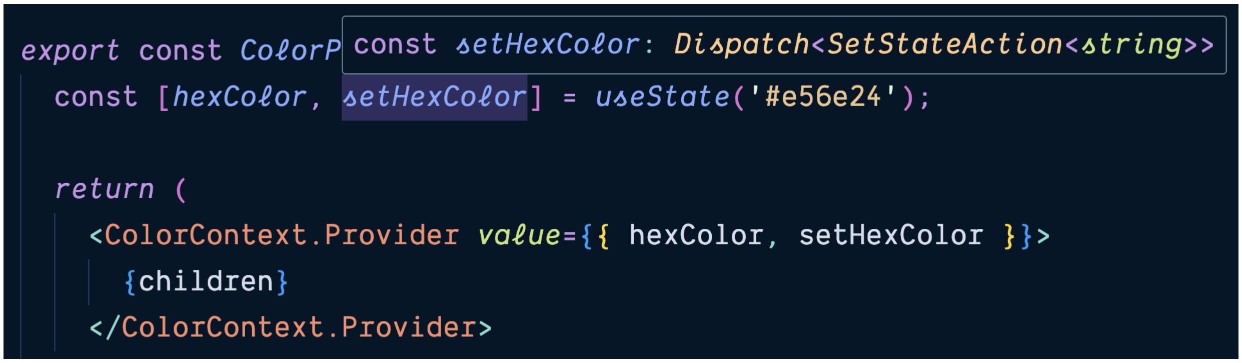
```
      {children}
    </ColorContext.Provider>
  );
};
```

In the code above, we get glimpse of the easier possible solution. We can just completely opt out of TypeScript by using `any`. Not only will we lose the type safety that comes with that, but we'll also lose all of the benefits of autocomplete in our code. This works, but we can do better.

A Feasible Solution: Use a Type Assertion

We don't have our initial hex color yet, but we know that we will eventually, right? In this case, we *could* lie to TypeScript and try to convince it that `null` is actually a hex color.

First, we'll need to figure out what the eventual type of the `value` prop is going to be. The easiest way to do this is to hover over `setHexColor` and look at the type.



```
export const ColorP const setHexColor: Dispatch<SetStateAction<string>>
  const [hexColor, setHexColor] = useState('#e56e24');

  return (
    <ColorContext.Provider value={{ hexColor, setHexColor }}>
      {children}
    </ColorContext.Provider>
```

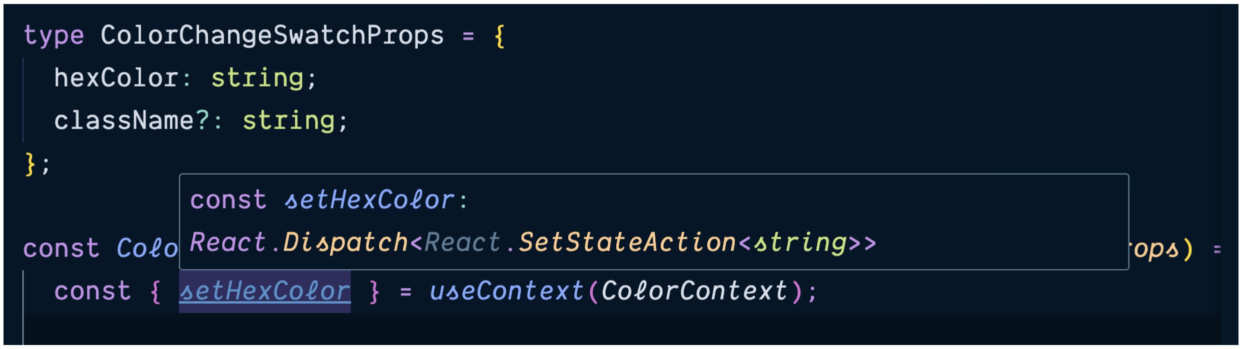
We know that we're passing `ColorContext.Provider` an object with `hexColor` and `setHexColor`. Let's break that out into its own type:

```
type ColorState = {
  hexColor: string;
  setHexColor: Dispatch<SetStateAction<string>>;
};
```

Now we can let TypeScript know what it can eventually expect to receive, even if we don't have the value just yet:

```
export const ColorContext = createContext<ColorState>(null as
any);
```

If we hover over `ColorContext`, we'll see the `const ColorContext: React.Context<ColorState>`. If we visit any of the components that use this context, we'll see that it has the correct type annotations, as pictured below.



```
type ColorChangeSwatchProps = {
  hexColor: string;
  className?: string;
};

const Colo                                                          ops) =
  const { setHexColor } = useContext(ColorContext);
```

```
const setHexColor:
React.Dispatch<React.SetStateAction<string>>
```

If we want to feel better about our deceit, we can at least tell it a lie that's exactly what it should come to expect eventually:

```
export const ColorContext = createContext<ColorState>(
  null as unknown as ColorState,
);
```

If we don't have a guilty conscience, we might also consider doing this so that ESLint doesn't get angry with us for explicitly using `any` in our code.

Repo Code

You can find the code above (`context.tsx`) in [the GitHub repo for this tutorial, in the `type-assertion` branch].

A Compromised Solution: Tell the Truth

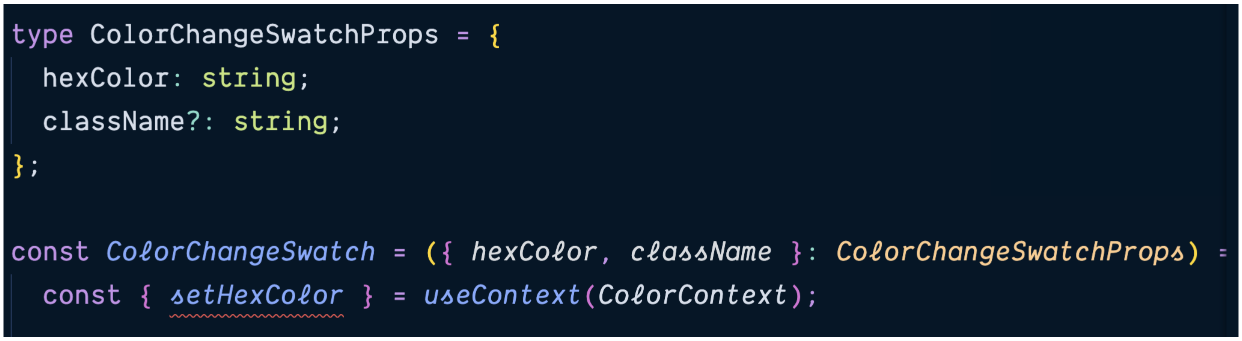
Another option is to tell the truth. The value of the context can either be `ColorState` or `undefined`:

```
export const ColorContext = createContext<ColorState | null>
(null);
```

This feels better, but it will cause a different set of problems:

```
Errors  Files
  1  src/components/color-change-swatch.tsx:13
  2  src/components/color-picker/color-select.tsx:11
  1  src/components/color-picker/color-swatch.tsx:6
  1  src/components/color-properties/to-cmyk.tsx:8
  1  src/components/color-properties/to-hsl.tsx:8
  1  src/components/color-properties/to-hsv.tsx:8
  1  src/components/color-properties/to-rgb.tsx:8
  1  src/components/related-colors/index.tsx:11
```

The issue is now that TypeScript isn't sure if we're working with `ColorState` or `undefined`. And now we need to check every time we use it.



```
type ColorChangeSwatchProps = {
  hexColor: string;
  className?: string;
};

const ColorChangeSwatch = ({ hexColor, className }: ColorChangeSwatchProps) =
  const { setHexColor } = useContext(ColorContext);
```

So it's now on us to check to see if it's defined or not every time we use it. Here's one example from the `ColorChangeSwatch` component:

```
const ColorChangeSwatch = ({ hexColor, className }:
ColorChangeSwatchProps) => {
  // 👀 Store the context in a variable.
  const context = useContext(ColorContext);

  return (
    <Button
      className={clsx(
        'border-2 border-slate-900 transition-shadow duration-
200 ease-in
        hover:shadow-xl',
        className,
      )}
      style={{ backgroundColor: hexColor }}
      onClick={() => {
        // 👀 Conditionally use any property on it.
        context?.setHexColor(hexColor);
      }}
    >
      {hexColor}
    </Button>
  );
};
```

Repo Code

You can find the code above (`color-change-swatch.tsx`) in the GitHub repo for this tutorial.

I’m not going to make you endure the chaos that this has caused in our application. But I will invite you to look at it here if you’re interested. It involves changing ten files, and this isn’t even that big an application. Needless to say, being honest with TypeScript probably isn’t worth it in this case.

A Reasonable Solution: Create an Abstraction

Let’s say we don’t want to lie to TypeScript and we also don’t want to endure what I just went through creating that example for you. It *is* possible for us to have our cake and eat it too. It's just going to take a little work.

Instead of checking whether or not the context is defined every single time we use it, we *could* create our own utilities that will do this for us and then never worry about it again. This is going to involve a little bit of upfront work, but—unlike in the previous solution—once we have our abstractions in place, we'll never have to think about them again.

I'm going to create a new branch off of `main` called create-context.

Creating Our Own `createContext`

Let's start by making a new file called `src/create-context.tsx`. We'll start with a simple function so that TypeScript doesn't get mad at us for having an empty file:

```
export const createContext = () => {};
```

Next, we're going to use React's `createContext` to create a `Context` object that we'll build on top of:

```
import React from 'react';

export const createContext = () => {
  const Context = React.createContext(null);
};
```

I'm using `React.createContext` as opposed to `import { createContext } from 'react';` so that we don't have any naming collisions. The name of our function isn't important, but I like consistency.

Now we'll do something similar to what we did in the previous example: we'll acknowledge that we might not know what the initial value is. We'll also use a generic so that this function is reusable for other contexts:

```
import React from 'react';

export const createContext = <T extends object>() => {
  const Context = React.createContext<T | null>(null);
};
```

I know that we're going to want to pass both `hexColor` and `setHexColor`, so I'm choosing to constrain `T` to some kind of object. If generics are bewildering to you, you can just choose to use the `ColorState` that we created earlier. That might look something like this:

```
// 🚨 Note: We won't be using this code. It's just an
alternative to code above.
export const createContext = () => {
  const Context = React.createContext<ColorState | null>(null);
};
```

Next, we're going to create our own take on the `useContext` hook. This will have access to the `Context` that we just created in its closure scope. The new `useContext` will check to make sure that context exists and that it's not `null`. If it *is* `null`, an error is thrown. TypeScript is smart enough to understand that, if our `Context` can only be either `T` or `null` and it's not `null`, it must be `T`. Going forward, we'll *only* be using our custom `useContext` hook:

```
import React from 'react';

export const createContext = <T extends object>() => {
  const Context = React.createContext<T | null>(null);

  const useContext = () => {
    const ctx = React.useContext(Context);

    if (ctx === null) {
      throw new Error('useContext must be inside a Provider with
a value.');
```

```
    }

    return ctx;
  };

  return [useContext, Context.Provider] as const;
};
```

To review:

- We know that the value of the context can either be `T` or `null`.
- So, pull the value out of `React.useContext` and take a look at it.
- Throw an error up if `ctx` is `null`. This means that we won't return anything in the case that `ctx` is `null`.
- If `ctx` can only either be `null` or `T` and it's not `null`, then it's `T`. This means we don't have to keep verifying that it's *not* `null` throughout our application like we did before.
- Return `Context.Provider` and our customized version of `useContext` as a tuple, like we might have seen with `useState` and `useReducer`.

This works because we're not using the React hook at the top level, which isn't allowed. Instead, we have a function that has access to the context at any point and can check to see if it's valid.

The `as const` at the end of the last line will tell TypeScript that we don't expect this array to change ever, and that it can be sure the first element will always be our `useContext` function and the second element will always be `Context.Provider`.

We end up with the following signature:

```
const createContext: <T extends object>() => readonly [
  () => T,
  React.Provider<T | null>,
];
```

Using Our New Hook

We've created our new hook. Now we just need to use it. Let's update `src/context.tsx` as follows:

```
import { Dispatch, PropsWithChildren, SetStateAction, useState }
from 'react';

import { createContext } from './create-context';

type ColorState = {
  hexColor: string;
  setHexColor: Dispatch<SetStateAction<string>>;
};

const [useContext, Provider] = createContext<ColorState>();

export const useColor = useContext;

export const ColorProvider = ({ children }: PropsWithChildren)
=> {
  const [hexColor, setHexColor] = useState('#e56e24');

  return <Provider value={{ hexColor, setHexColor }}>{children}
</Provider>;
};
```

You'll notice that our customized `createContext` doesn't require an argument anymore. We just need to tell it what type we're eventually expecting. This will return a hook and a provider. We'll export that hook. I gave it a useful name because it's bound to this specific context.

Finally, we use this `Provider` returned from our `createContext` function to wrap the `Application`.

We still need to update our code to use our new `useColor` hook instead of `useContext`. It will demonstrate this in src/components/color-picker/color-select.tsx, and you can check out the rest in the completed create-context branch:

```
import { useColor } from '../../context';
import HexColor from '../hex-color';

type ColorSelectProps = {
  label?: string;
};

const ColorSelect = ({ label = 'Color' }: ColorSelectProps) => {
  // 👀 Our new `useColor` hook in action!
  const { hexColor, setHexColor } = useColor();

  return (
    <div className="flex flex-col gap-2">
      <label htmlFor="color-input">{label}</label>
      <input
        id="color-input"
        className="h-80 w-full"
        type="color"
        value={hexColor}
        onChange={(e) => setHexColor(e.target.value)}
      />
      <HexColor hexColor={hexColor} />
    </div>
  );
};

export default ColorSelect;
```

Conclusion

Using the Context API with TypeScript can be a bit tricky, but there are solutions available. While using any can provide a quick fix, it’s not ideal for maintaining type safety, and it can lead to issues down the line. Using a type assertion will definitely work in most cases. But creating an abstraction around createContext and using a custom useContext hook will provide us with a much more solid solution. In fact, it hides the fact that we’re using a context at all and leaves us with a convenient hook—in this case, `useColor`. With this approach, we can ensure that the context’s value is not `null` and avoid the need for constant null-checking throughout our application.

In the next installment, we’ll continue to explore the use of TypeScript in our React applications by looking at extending built-in DOM types and creating polymorphic types for our component props.

Chapter 4: Extending DOM Elements and Creating Polymorphic Components

In most of the larger applications and projects I've worked on, I often find myself building a bunch of components that are really supersets or abstractions on top of the standard HTML elements. Some examples include custom button elements that might take a prop defining whether or not that button should be a primary or secondary button, or maybe one that indicates that it will invoke a dangerous action, such as deleting or removing a item from the database. I still want my button to have all the properties of a button in addition to the props I want to add to it.

Another common case is that I'll end up creating a component that allows me to define a label and an input field at once. I don't want to re-add all of the properties that an `<input />` element takes. I want my custom component to behave just like an input field, but *also* take a string for the label and automatically wire up the `htmlFor` prop on the `<label />` to correspond with the `id` on the `<input />`.

In JavaScript, I can just use `{...props}` to pass through any props to an underlying HTML element. This can be a bit trickier in TypeScript, where I need to explicitly define what props a component will accept. While it's nice to have fine-grained control over the exact types that my component accepts, it can be tedious to have to add in type information for every single prop manually.

In certain scenarios, I need a single adaptable component, like a `<div>`, that changes styles according to the current theme. For example, maybe I want to define what styles should be used depending on whether or not the user has manually enabled light or dark mode for the UI. I don't want to redefine this component for every single block element (such as `<section>`, `<article>`, `<aside>`, and so on). It should be capable of representing different semantic HTML elements, with TypeScript automatically adjusting to these changes.

There are a few strategies that we can employ:

- For components where we're creating an abstraction over just one kind of element, we can extend the properties of that element.
- For components where we want to define different elements, we can create polymorphic components. A **polymorphic component** is a component designed to render as different HTML elements or components while maintaining the same properties and behaviors. It allows us to specify a prop to determine its rendered element type. Polymorphic components offer flexibility and reusability without us having to reimplement the component. For a concrete example, you can look at Radix's implementation of a polymorphic component.

In this tutorial, we'll look at both of these strategies.

Following Along with This Tutorial

All of the code that we'll be working with can be found in this GitHub repository, starting with the main branch. I'll push up other branches along the way.

Mirroring and Extending the Properties of an HTML Element

Let’s start with that first example mentioned in the introduction. We want to create a button that comes baked in with the appropriate styling for use in our application. In JavaScript, we might be able to do something like this:

```
const Button = (props) => {
  return <button className="button" {...props} />;
};
```

In TypeScript, we could just add what we know we need. For example, we know that we need the `children` if we want our custom button to behave the same way an HTML button does:

```
const Button = ({ children }: React.PropsWithChildren) => {
  return <button className="button">{children}</button>;
};
```

You can imagine that adding properties one at a time could get a bit tedious. Instead, we can tell TypeScript that we want to match the same props that it would use for a `<button>` element in React:

```
const Button = (props: React.ComponentProps<'button'>) => {
  return <button className="button" {...props} />;
};
```

But we have a new problem. Or, rather, we *had* a problem that *also* existed in the JavaScript example and which we ignored. If someone using our new `Button` component passes in a `className` prop, it will override our `className`. We could (and we will) add some code to deal with this in a moment, but I don’t want to pass up the opportunity to show you how to use a utility type in TypeScript to say “I want to use *all* of the props from an HTML button *except* for one (or more)”:

```
type ButtonProps = Omit<React.ComponentProps<'button'>,
'className'>;

const Button = (props: ButtonProps) => {
  return <button className="button" {...props} />;
};
```

Now, TypeScript will stop us or anyone else from passing a `className` property into our `Button` component. If we just wanted to extend the class list with whatever is passed in, we could do that in a few different ways. We could just append it to the list:

```
type ButtonProps = React.ComponentProps<'button'>;

const Button = (props: ButtonProps) => {
  const className = 'button ' + props.className;

  return <button className={className.trim()} {...props} />;
};
```

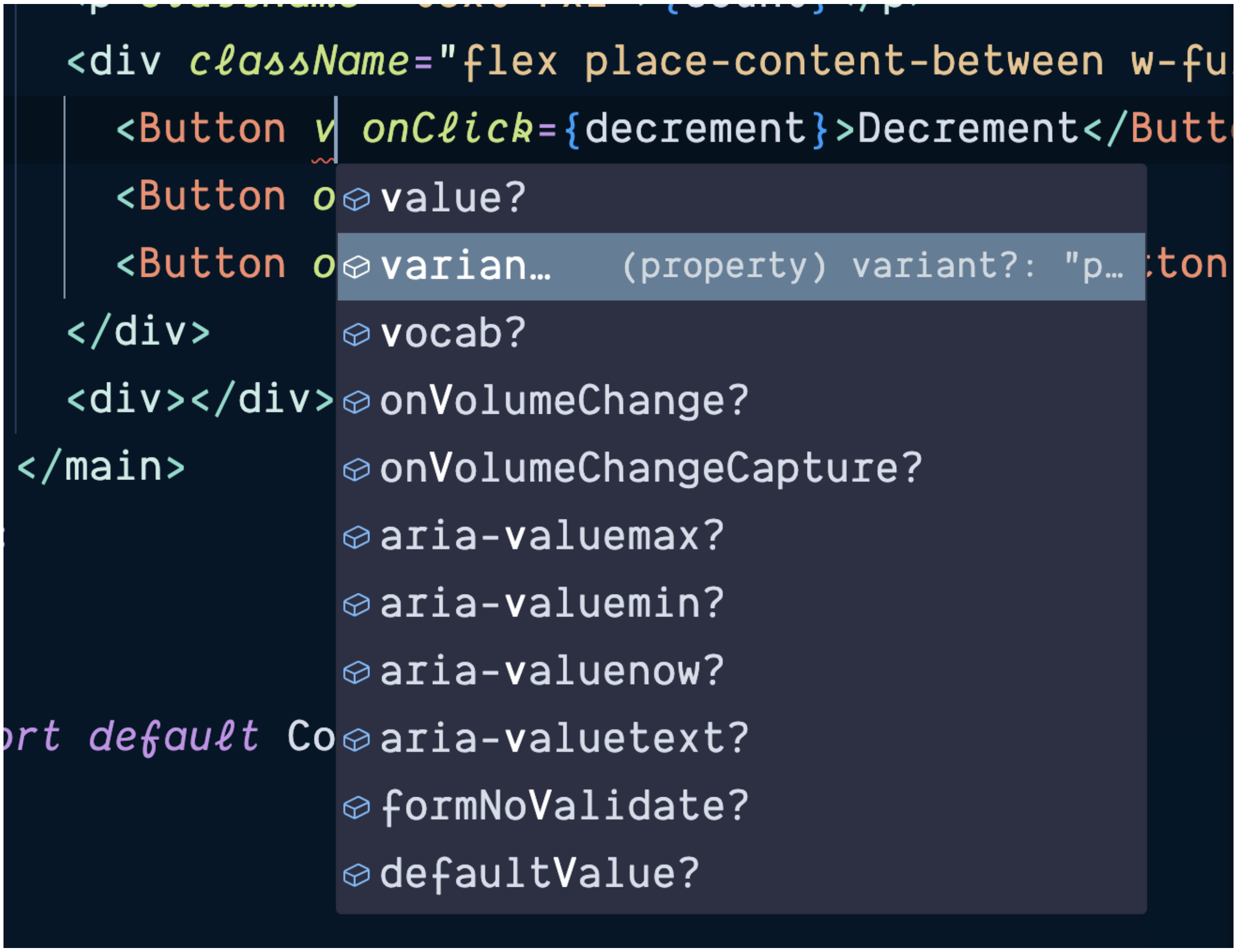
I like to use the clsx library when working with classes, as it takes care of most of these kinds of things on our behalf:

```
import React from 'react';
import clsx from 'clsx';

type ButtonProps = React.ComponentProps<'button'>;

const Button = ({ className, ...props }: ButtonProps) => {
  return <button className={clsx('button', className)}
{...props} />;
};

export default Button;
```

We learned how to limit the props that a component will accept. To extend the props, we can use an intersection:

```
type ButtonProps = React.ComponentProps<'button'> & {
  variant?: 'primary' | 'secondary';
};
```

We’re now saying that `Button` accepts all of the props that a `<button>` element accepts plus one more: `variant`. This prop will show up with all the other props we inherited from `HTMLButtonElement`.



```
  <div className="flex place-content-between w-fu
    <Button v onClick={decrement}>Decrement</Butt
    <Button o
    <Button o                                  ton
  </div>
  <div></div>
</main>

rt default Co
```

We can add support to our `Button` to add this class as well:

```
const Button = ({ variant, className, ...props }: ButtonProps)
=> {
  return (
    <button
      className={clsx(
        'button',
        variant === 'primary' && 'button-primary',
        variant === 'secondary' && 'button-secondary',
        className,
      )}
      {...props}
    />
  );
};
```

We can now update `src/application.tsx` to use our new button component:

```
diff --git a/src/application.tsx b/src/application.tsx
index 978a61d..fc8a416 100644
--- a/src/application.tsx
+++ b/src/application.tsx
@@ -1,3 +1,4 @@
+import Button from './components/button';
 import useCount from './use-count';

 const Counter = () => {
@@ -8,15 +9,11 @@ const Counter = () => {
       <h1>Counter</h1>
       <p className="text-7xl">{count}</p>
       <div className="flex place-content-between w-full">
-        <button className="button" onClick={decrement}>
+        <Button onClick={decrement}>
           Decrement
-        </button>
-        <button className="button" onClick={reset}>
-          Reset
-        </button>
-        <button className="button" onClick={increment}>
-          Increment
-        </button>
+        </Button>
+        <Button onClick={reset}>Reset</Button>
+        <Button onClick={increment}>Increment</Button>
       </div>
       <div>
         <form
@@ -32,9 +29,9 @@ const Counter = () => {
         >
           <label htmlFor="set-count">Set Count</label>
           <input type="number" id="set-count" name="set-count"
/>
-          <button className="button-primary" type="submit">
+          <Button variant="primary" type="submit">
             Set
```

```
-          </button>
+          </Button>
         </form>
       </div>
     </main>
```

Repo Code

You can find the changes above in the `button` branch of the GitHub repo for this tutorial.

Creating Composite Components

Another common component that I typically end up making for myself is a component that correctly wires up a label and input element with the correct `for` and `id` attributes respectively. I tend to grow weary typing this out over and over:

```
<label htmlFor="set-count">Set Count</label>
<input type="number" id="set-count" name="set-count" />
```

Without extending the props of an HTML element, I might end up slowly adding props as needed:

```
type LabeledInputProps = {
  id?: string;
  label: string;
  value: string | number;
  type?: string;
  className?: string;
  onChange?: ChangeEventHandler<HTMLInputElement>;
};
```

As we saw with the button, we can refactor it in a similar fashion:

```
type LabeledInputProps = React.ComponentProps<'input'> & {
  label: string;
};
```

Other than `label`, which we’re passing to the (uhh) label that we’ll often want grouped with our inputs, we’re manually passing props through one by one. Do we want to add `autofocus`? Better add another prop. It would be better to do something like this:

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

type LabeledInputProps = ComponentProps<'input'> & {
  label: string;
};

const LabeledInput = ({ id, label, ...props }:
LabeledInputProps) => {
  return (
    <>
      <label htmlFor={id}>{label}</label>
      <input {...props} id={id} readOnly={!props.onChange} />
    </>
  );
};

export default LabeledInput;
```

We can swap in our new component in `src/application.tsx`:

```
<LabeledInput
  id="set-count"
  label="Set Count"
  type="number"
  onChange={(e) => setValue(e.target.valueAsNumber)}
  value={value}
/>
```

We can pull out the things we need to work with and then just pass everything else on through to the `<input />` component, and then just pretend for the rest of our days that it’s a standard `HTMLInputElement`.

TypeScript doesn’t care, since `HTMLElement` is pretty flexible, as the DOM pre-dates TypeScript. It only complains if we toss something completely egregious in there.

Repo Code

You can see all of the changes above in the `input` branch of the GitHub repo for this tutorial.

Polymorphic Components

Earlier, I told you that, over the course of building out a large application, I tend to end up making a few wrappers around components. `Box` is a primitive wrapper around the basic block elements in HTML (such as `<div>`, `<aside>`, `<section>`, `<article>`, `<main>`, `<head>`, and so on). But just as we don’t want to lose all the semantic meaning we get from these tags, we also don’t need multiple variations of `Box` that are all basically the same. What we’d like to is use `Box` but also be able to specify what it ought to be under the hood.

Here’s an overly simplified take on a `Box` element inspired by Styled Components.

And here’s an example of a `Box` component from Paste, Twilio’s design system:

```
<Box as="article" backgroundColor="colorBackgroundBody"
padding="space60">
  Parent box on the hill side
  <Box
    backgroundColor="colorBackgroundSuccessWeakest"
    display="inline-block"
    padding="space40"
  >
    nested box 1 made out of ticky tacky
  </Box>
</Box>
```

Here’s a simple implementation that doesn’t have any pass through any of the props, like we did with `Button` and `LabelledInputProps` above:

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

type BoxProps = PropsWithChildren<{
  as: 'div' | 'section' | 'article' | 'p';
}>;

const Box = ({ as, children }: BoxProps) => {
  const TagName = as || 'div';
  return <TagName>{children}</TagName>;
};

export default Box;
```

We refine `as` to `TagName`, which is a valid component name in JSX. That works as far a React is concerned, but we also want to get TypeScript to adapt accordingly to the element we're defining in the `as` prop:

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

type BoxProps = ComponentProps<'div'> & {
  as: 'div' | 'section' | 'article' | 'p';
};

const Box = ({ as, children }: BoxProps) => {
  const TagName = as || 'div';
  return <TagName>{children}</TagName>;
};

export default Box;
```

I honestly don't even know if elements like `<section>` have any properties that a `<div>` doesn't. While I'm sure I could look it up, none of us feel good about this implementation.

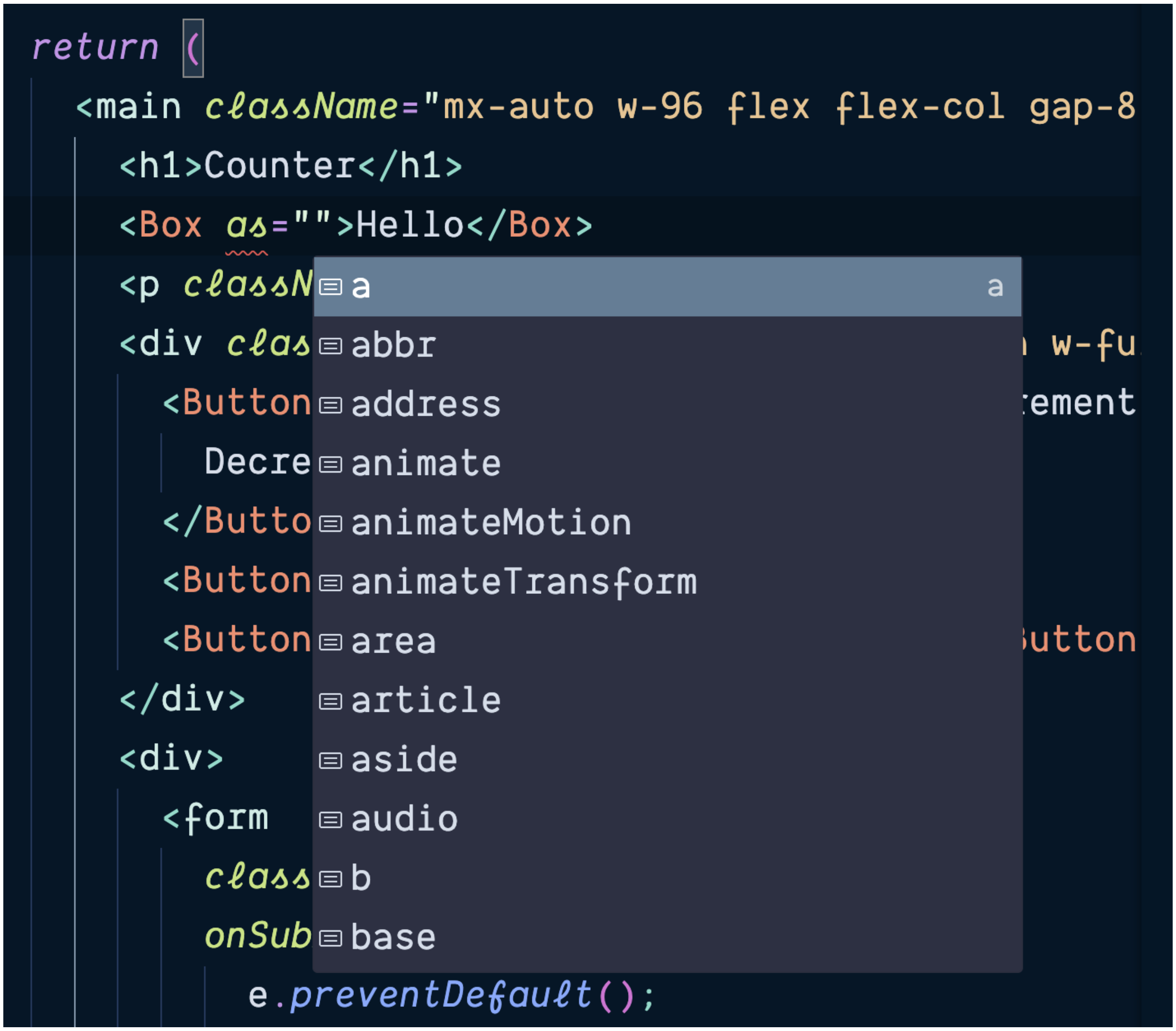
But what's that `'div'` being passed in there and how does it work? If we look at the type definition for `ComponentPropsWithRef`, we see the following:

```
type ComponentPropsWithRef<T extends ElementType> = T extends
new (
  props: infer P,
) => Component<any, any>
```

```
  ? PropsWithoutRef<P> & RefAttributes<InstanceType<T>>
  : PropsWithRef<ComponentProps<T>>;
```

We can ignore all of those ternaries. We’re interested in `ElementType` right now:

```
type BoxProps = ComponentPropsWithRef<'div'> & {
  as: ElementType;
};
```



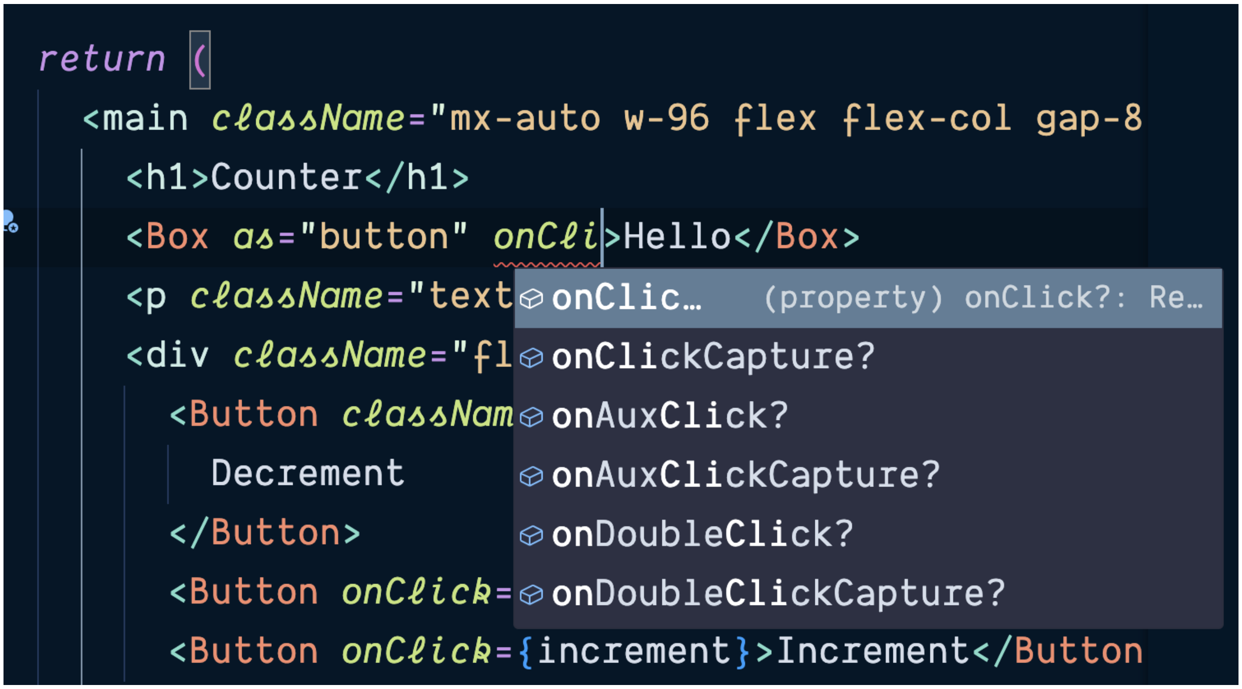
```
return (
  <main className="mx-auto w-96 flex flex-col gap-8
    <h1>Counter</h1>
    <Box as="">Hello</Box>
    <p classN
    <div clas                                    w-fu
      <Button                                    ement
        Decre
      </Butto
      <Button
      <Button                                    utton
    </div>
    <div>
      <form
        class
        onSub
          e.preventDefault();
```

Okay, that’s interesting, but what if we wanted the type argument we give to `ComponentProps` to be the same as … `as`?

We *could* try something like this:

```
import { ComponentProps, ElementType } from 'react';

type BoxProps<E extends ElementType> = Omit<ComponentProps<E>,
'as'> & {
  as?: E;
};

const Box = <E extends ElementType = 'div'>({ as, ...props }:
BoxProps<E>) => {
  const TagName = as || 'div';
  return <TagName {...props} />;
};

export default Box;
```

Now, a `Box` component will adapt to whatever element type we pass in with the `as` prop.



```
return (
  <main className="mx-auto w-96 flex flex-col gap-8
    <h1>Counter</h1>
    <Box as="button" onCli>Hello</Box>
    <p className="text
    <div className="fl
      <Button classNam
        Decrement
      </Button>
      <Button onClick=
      <Button onClick={increment}>Increment</Button
```

We can now use our `Box` component wherever we might otherwise use a `<div>`:

```
<Box as="section" className="flex place-content-between w-full">
  <Button className="button" onClick={decrement}>
    Decrement
  </Button>
  <Button onClick={reset}>Reset</Button>
  <Button onClick={increment}>Increment</Button>
</Box>
```

Repo Code

You can the final result on the the `polymorphic` branch of the GitHub repo for this tutorial.

Conclusion

We’ve just explored different strategies for creating polymorphic components with React and TypeScript. By extending the properties of an HTML element, it’s possible to create abstractions over that element with additional behavior. A polymorphic component is a single adaptable component that can represent different semantic HTML elements, with TypeScript automatically adjusting to these changes. The cool thing about this is that we can now universally add themes and other functionality to our application with this new primitive.

These approaches work in most versions of TypeScript. In the next chapter, we’re going to take a look at some of the specific improvements the latest release of TypeScript brings to the table, and how we can use them to further improve the reliability and maintainability of our code.

Chapter 5: A Guided Tour of the Three Biggest Features in TypeScript 5.0

TypeScript, JavaScript’s more dependable cousin, keeps getting better and better. TypeScript 5.0 and its subsequent minor versions, 5.1 and 5.2, are brimming with powerful new features that promise to make our code cleaner, more powerful, and a lot easier to work with. As you can tell from the release notes, TypeScript 5.0 shipped with a ton of changes—both large and small.

I’m going to focus on three of the features that have had the biggest impact in my day-to-day development:

- **Decorators**. TypeScript 5.0 supports decorators that align with the ECMAScript proposal, allowing the modification of class behavior in a reusable way. These have some important distinctions between the decorators previously supported by the `--experimentalDecorators` flag.
- **const type parameters**. The new `const` type parameters enhance function call inferences, and ensure declared constants remain unchanged.
- **Upgrades and changes to enums**. TypeScript 5.0 enhances enums, improving type safety and making them more flexible and user-friendly.

Decorators: the Next Generation

Decorators have *almost* been part of ECMAScript for as long as I can remember. These nifty tools let us modify classes and members in a reusable way. They’ve been on the scene for a while in TypeScript—albeit under an experimental flag. Although the Stage 2 iteration of decorators was always experimental, decorators have been widely used in libraries like MobX, Angular, Nest, and TypeORM. TypeScript 5.0’s decorators are fully in sync with the ECMAScript proposal, which is pretty much ready for prime time, sitting at Stage 3.

Decorators let us craft a function that tweaks the behavior of a class and its methods. Imagine needing to sneak in some debug statements into our methods. Before TypeScript 5.0, we’d have been stuck copying and pasting the debug statements manually in each method. With decorators, we just do the job once and the change will be supported through each method the decorator is attached to.

Let’s say we want to create a decorator for logging that a given method is deprecated:

```
class Card {
  constructor(public suit: Suit, public rank: Rank) {
    this.suit = suit;
    this.rank = rank;
  }

  get name(): CardName {
    return `${this.rank} of ${this.suit}`;
  }

  @deprecated // 👀 This is a decorator!
  getValue(): number {
    if (this.rank === 'Ace') return 14;
    if (this.rank === 'King') return 13;
    if (this.rank === 'Queen') return 12;
    if (this.rank === 'Jack') return 11;
    return this.rank;
  }
  // The new way to do it!
  get value(): number {
    if (this.rank === 'Ace') return 14;
    if (this.rank === 'King') return 13;
    if (this.rank === 'Queen') return 12;
    if (this.rank === 'Jack') return 11;
    return this.rank;
  }
}

const card = new Card('Spades', 'Queen');
card.getValue();
```

We want a warning message logged to the console whenever `card.getValue()` is called. We could implement the above decorator as follows:

```
const deprecated = <This, Arguments extends any[], ReturnValue>
(
  target: (this: This, ...args: Arguments) => ReturnValue,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Arguments) => ReturnValue
  >,
) => {
  const methodName = String(context.name);

  function replacementMethod(this: This, ...args: Arguments):
ReturnValue {
    console.warn(`Warning: '${methodName}' is deprecated.`);
    return target.call(this, ...args);
  }

  return replacementMethod;
};
```

This might look a little confusing at first, but let’s break it down:

- Our decorator function takes two arguments: `target` and `context`.
- `target` is the method itself that we’re decorating.

- `context` is metadata about the method.
- We return some method that has the same signature.
- In this case, we’re calling `console.warn` to log a deprecation notice and then we’re calling the method.

The `ClassMethodDecorator` type has the following properties on it:

- `kind`: the type of the decorated property. In the example above, this will be `method`, since we’re decorating a method on an instance of a `Card`.
- `name`: the name of property. In the example above, this is `getValue`.
- `static`: a value indicating whether the class element is a static (`true`) or instance (`false`) element.
- `private`: a value indicating whether the class element has a private name.
- `access`: an object that can be used to access the current value of the class element at runtime.
- `has`: determines whether an object has a property with the same name as the decorated element.
- `get`: invokes the setter on the provided object.

Kick the Tires

You can kick the tires of the code samples above in this playground.

Decorators provide convenient syntactic sugar for adding log messages—like we did in the example above—as well as a number of other common use cases. For example, we could create a decorator that automatically binds the method to the current instance or that modifies the property descriptor of the method or class.

Unleashing const Type Parameters

`const` type parameters allow us to declare values with `const`. TypeScript will now be able to infer the type, not just the literal values.

Put more simply, the `const` type parameters allow us to spell out our intentions clearly in our code. If a variable is supposed to be a constant that never changes, `const` type parameters give us a safety net, making sure it stays constant, come what may.

Previous versions of TypeScript have supported using `as const` to declare an object or array as `readonly`. Once TypeScript knows that an object or array isn’t going to change at runtime, it can be a lot more confident in the assumptions it makes about the type.

Consider the following two arrays:

```
const avengers = [
  'Black Widow',
  'Captain America',
  'Hawkeye',
  'Hulk',
  'Iron Man',
  'Thor',
];

const ninjaTurtles = [
  'Donatello',
  'Leonardo',
  'Michelangelo',
  'Raphael',
] as const;
```

If we hover over each in this playground, we’ll see that TypeScript has inferred two very different types:

```
const avengers: string[];
const ninjaTurtles: readonly [
  'Donatello',
  'Leonardo',
  'Michelangelo',
  'Raphael',
];
```

Things get a little trickier when working with `readonly` data structures with functions. Consider the following:

```
type WithNames = { names: readonly string[] };

const turtles = getNames({
  names: ['Donatello', 'Leonardo', 'Michelangelo', 'Raphael'],
});
```

In this example, `turtles` would be `string[]`, which is not what we might expect if we look closely at `WithNames`, which asserts that the `names` property is `readonly`. Previously, we were able to get something closer to what we might expect by adding `as const` to the array as we called the function:

```
const turtles = getNames({
  names: ['Donatello', 'Leonardo', 'Michelangelo', 'Raphael']
as const,
});
```

`turtles` would now correctly be the following type:

```
const turtles: readonly ['Donatello', 'Leonardo',
'Michelangelo', 'Raphael'];
```

But this can be cumbersome, and if we forget to add `as const` each time we call the function, we might end up with something other than what we expect—and that’s never good.

In TypeScript 5.0, we can add `const` to the type of the parameter itself, and then we no longer have to worry about adding `as const` every single time we invoke the function:

```
//                  ⬇
const getNames = <const T extends WithNames>(arg: T):
T['names'] => {
  return arg.names;
};

const turtles = getNames({
  names: ['Donatello', 'Leonardo', 'Michelangelo', 'Raphael'],
});

const avengers = getNames({
  names: [
    'Black Widow',
    'Captain America',
    'Hawkeye',
    'Hulk',
    'Iron Man',
    'Thor',
  ],
});
```

The types are now as follows:

```
const turtles: readonly ['Donatello', 'Leonardo',
'Michelangelo', 'Raphael'];
const avengers: readonly [
  'Black Widow',
  'Captain America',
  'Hawkeye',
  'Hulk',
  'Iron Man',
  'Thor',
];
```

Have a Play

You can play around with the code above in this TypeScript playground.

As with anything, there are some trade-offs. Using `as const` sets the object as read-only at the time of declaration. `const` parameters give us another option: the ability to set the object as read-only at the time of invocation. Having the ability to choose the behavior that best serves our use case is a welcome addition to TypeScript.

Upgraded and Enhanced Enums

Enums—which provide a handy (if somewhat under appreciated) way of defining a set of named constants—have received a major boost in TypeScript 5.0.

In the past, we could have accidentally passed an incorrect number to a function expecting an enum and gotten away with it. TypeScript 5.0 is a lot stricter. It will throw an error if there's a mismatch. It now treats all enums as union enums, making them safer and easier to work with.

Up until TypeScript 5.0, an enum was really just a set of numeric constraints. Let's say we had an enum that tracked the state of an HTTP request:

```
enum RequestState {
  NotStarted,
  Loading,
  Success,
  Error,
}
```

TypeScript will compile that down to the following object:

```
{
  "0": "NotStarted",
  "1": "Loading",
  "2": "Success",
  "3": "Error",
  "NotStarted": 0,
  "Loading": 1,
  "Success": 2,
  "Error": 3
}
```

Enums could be treated as a bound set of numbers. In the case of `RequestState`, this would be 0, 1, 2, and 3. But in previous versions of TypeScript, we could get around the predefined values in an enum with any number. For example, the following is valid in TypeScript 4.9.5:

```
const logRequestState = (request: RequestState) => {
  console.log('Request:', request, RequestState[request]);
};

logRequestState(RequestState.Loading); // ✅ Logs: "Request:",
1,  "Loading"
logRequestState(0); // ✅ Logs: "Request:",  0,  "NotStarted"
logRequestState(55); // ✅ Logs:  "Request:",  55,  undefined
🤨
```

`55` should not be a valid argument for `logRequestState`. In TypeScript 5.0, all enums are now treated as union enums, which provide better type safety. Effectively, `RequestState` is `0 | 1 | 2 | 3`, which means `55` is no longer a valid argument. This is closer to what we might have expected, as seen in this example:

```
logRequestState(RequestState.Loading); // ✅ Logs: "Request:",
1,  "Loading"
logRequestState(0); // ✅ Logs: "Request:",  0,  "NotStarted"
logRequestState(55); // ❌ Argument of type '55' is not
assignable to parameter
➥of type 'RequestState'.
```

TypeScript is now better at enforcing types set in enums. The following will compile in TypeScript 4.9.5, but it *won't* in TypeScript 5.0 and later. One word of caution: if we try to dynamically compute the number, TypeScript is not able to be as helpful. For example, `logRequestState(3 + 2)` will *not* generate an error:

```
enum Letters {
  A = 'Aye!',
  B = 'Bee!',
  C = 'Sea!',
}

enum Numbers {
  one = 1,
  two = 2,
  three = Letters.C,
}

const n: number = Numbers.three;
```

As you can see, `Numbers.three` is technically a letter and should be allowed to be assigned to a number. TypeScript 5.0 makes sure this behaves as expected. That said, it's a breaking change and may require us to revisit our code.

Previously, I avoided enums in favor of unions. The new behavior should work well with existing code and also provide more safety than the previous implementations. Both enums and unions are valid means of defining a set of values, and the choice now comes down to our preference.

Conclusion

TypeScript 5.0 is a major advancement to the user-friendliness and flexibility of the language. It nudges us toward better coding practices, promotes cleaner code, and catches potential errors before they become a headache. The TypeScript team is continuing to add important features and quality-of-life improvements to the language. It's obviously a smaller set of changes this time around, but TypeScript 5.1 has added additional features such as easier implicit returns for functions that return `undefined`.

For me, TypeScript has been critical to my confidence in making changes to and maintaining large code bases. I often find that the compiler catches edge cases I would otherwise have missed and that might have caused my pager to go off at an undesirable hour. I’m personally excited to see how the language evolves over the next few years—especially if the current pace of updates continues.