PETER WOLF
< Web Developer & Advisor />

React & TypeScript

Crash Course Documents

Source:
https://www.youtube.com/watch?v=TPACABQTHvM

Typing Variables

Declare a variable like

let url = „https://google.com“

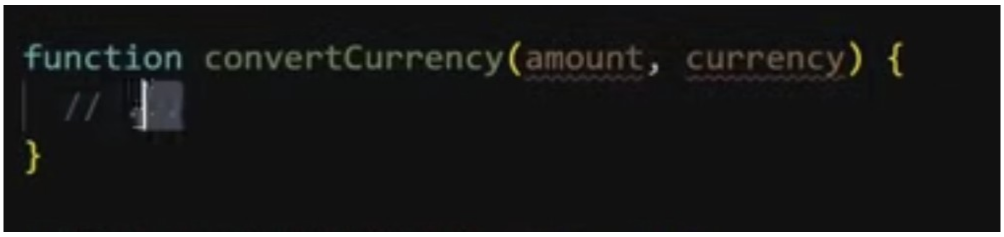
if you hover over the varibale, in the IDE you see typescript tells us of what type the varible is.

Change the string of URL to a number, you see the that typescript is messaging us an error.

url=5

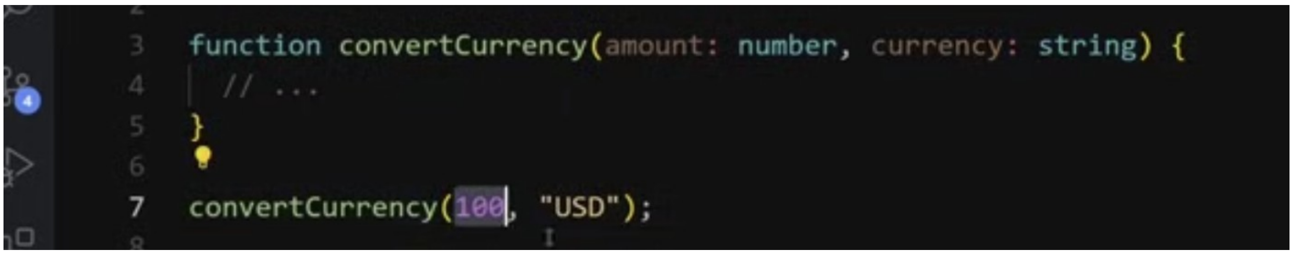
TS sets curlys below the varible name.

Typing functions



```
function convertCurrency(amount, currency) {
  //
}
```

we will not type the function itself...we want to type the parameters we pass to the function.

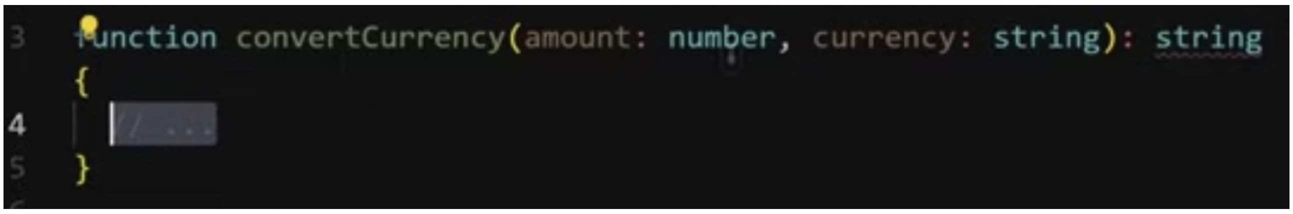


```
function convertCurrency(amount: number, currency: string) {
  // ...
}

convertCurrency(100, "USD");
```

Check if you change the value to not typing confirm.

Specify RETURN TYPE

In some cases you will define the return type – rarely used
the return type is placed after the ()

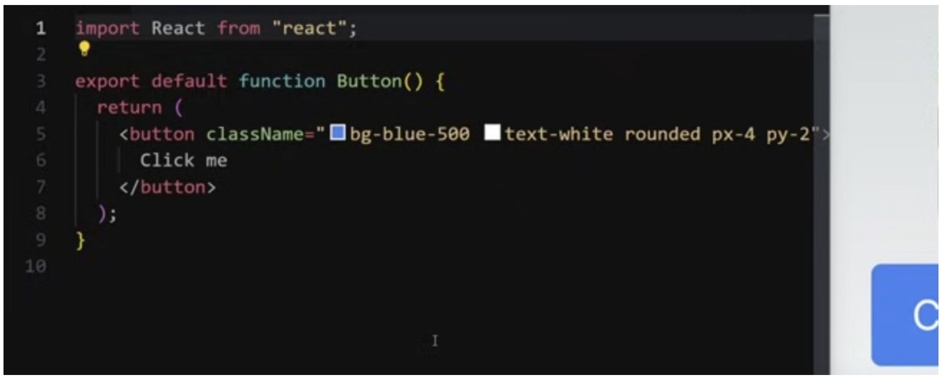


```
function convertCurrency(amount: number, currency: string): string
{
  // ...
}
```

PETER WOLF
< Web Developer & Advisor />

NOW USE IT IN REACT

In React all components are functions. If you write the name in our example BUTTON(){}
=> this is then a component.

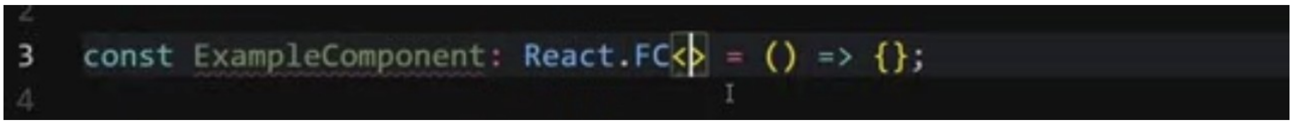


```
import React from "react";

export default function Button() {
  return (
    <button className="bg-blue-500 text-white rounded px-4 py-2">
      Click me
    </button>
  );
}
```

We only want to type the props – not the return value
In react people typically want to type the props only.

Excourse:

in the past versions of ReactJS, type decleration was done like this:



```
const ExampleComponent: React.FC<> = () => {};
```

FC stands for FunctionalComponent

Issue with this version:

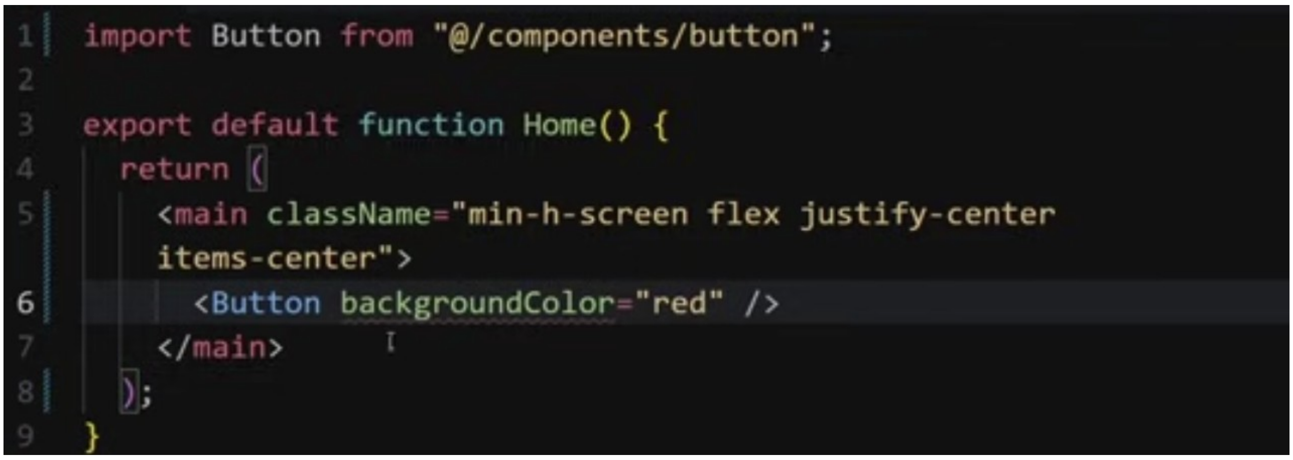
1. it is only used in arrow functions
2. it gets complicated in bigger applications

Let´s make an example:

I have two components

1. Homepage
2. Button

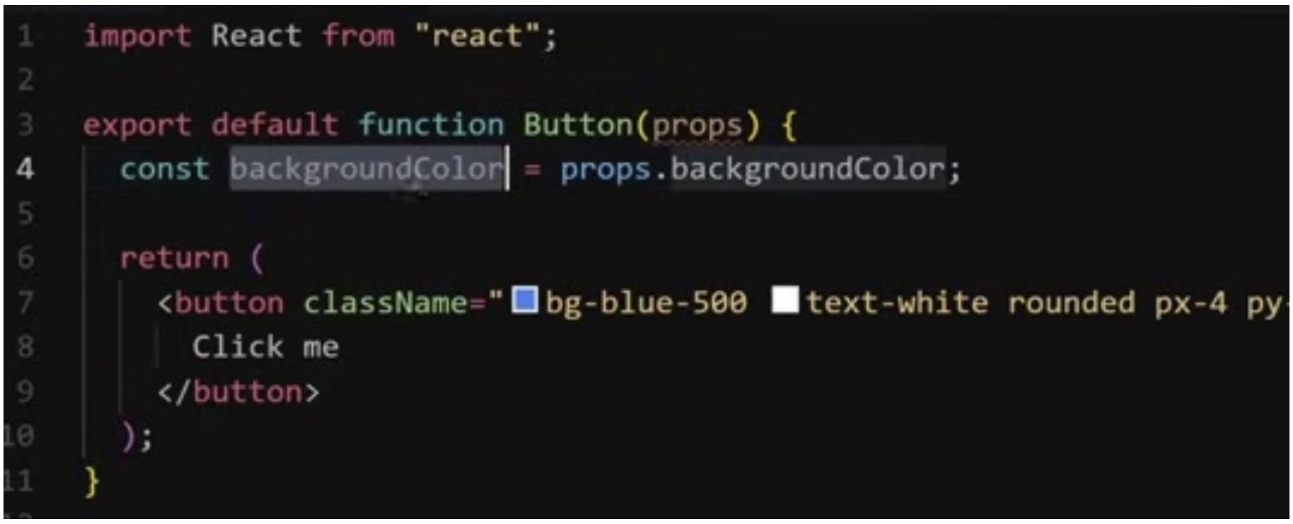
the BUTTON component will be imported to the homepage component.



```jsx
import Button from "@/components/button";

export default function Home() {
  return (
    <main className="min-h-screen flex justify-center
    items-center">
      <Button backgroundColor="red" />
    </main>
  );
}
```

With this, we will have an error message from typescript, that <Button/> cannot receive any props.

First we have to inform the <Button/> to accept props.
Props is an OBJECT! We can access the props like in line 4.



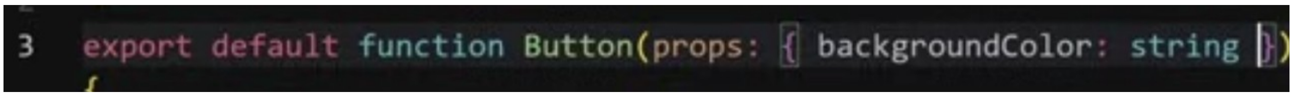
```jsx
1   import React from "react";
2
3   export default function Button(props) {
4     const backgroundColor = props.backgroundColor;
5
6     return (
7       <button className="bg-blue-500 text-white rounded px-4 py-
8         Click me
9       </button>
10    );
11  }
```

type the props

We cannot type the props like this:

(props : string){}

because props is an Object. We have to pass the types as an Object

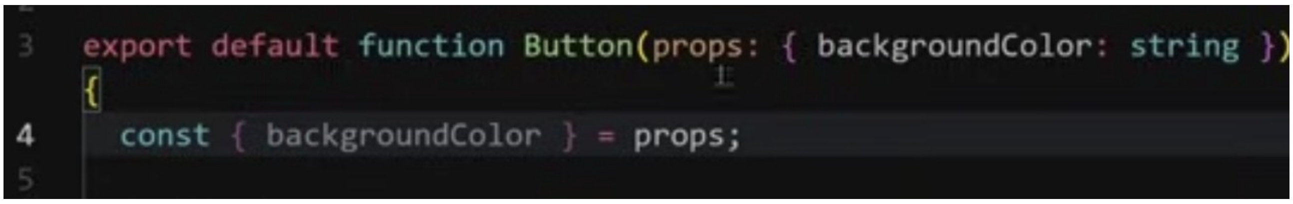


```
export default function Button(props: { backgroundColor: string })
```

destructure react object

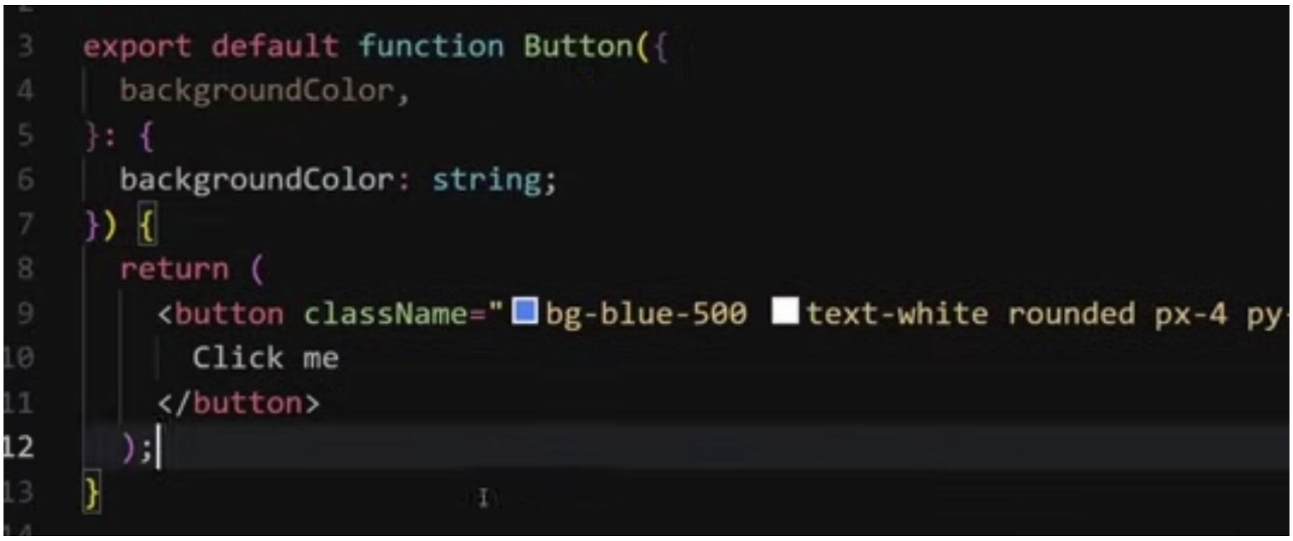
Typcially you don´t write your react component like above.

Destructure your component like this:



```
export default function Button(props: { backgroundColor: string })
{
  const { backgroundColor } = props;
```

but you can also desctructure immidiately in the props => **this is more realistic scenario**

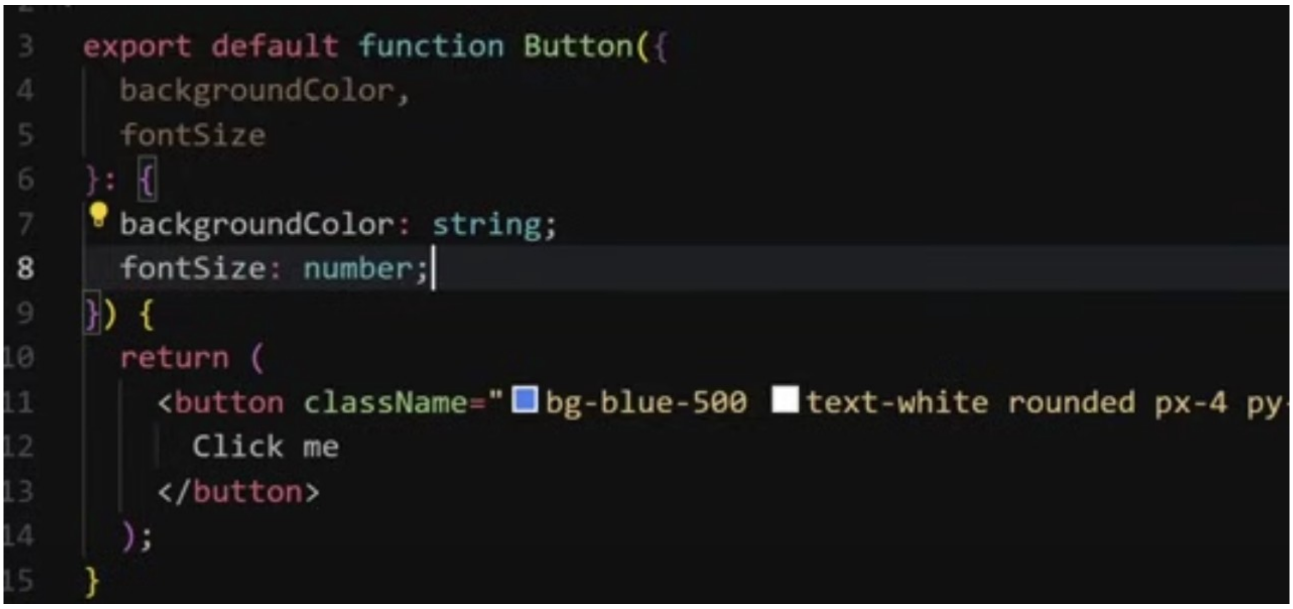


```
export default function Button({
  backgroundColor,
}: {
  backgroundColor: string;
}) {
  return (
    <button className="bg-blue-500 text-white rounded px-4 py
      Click me
    </button>
  );
}
```

extend the props

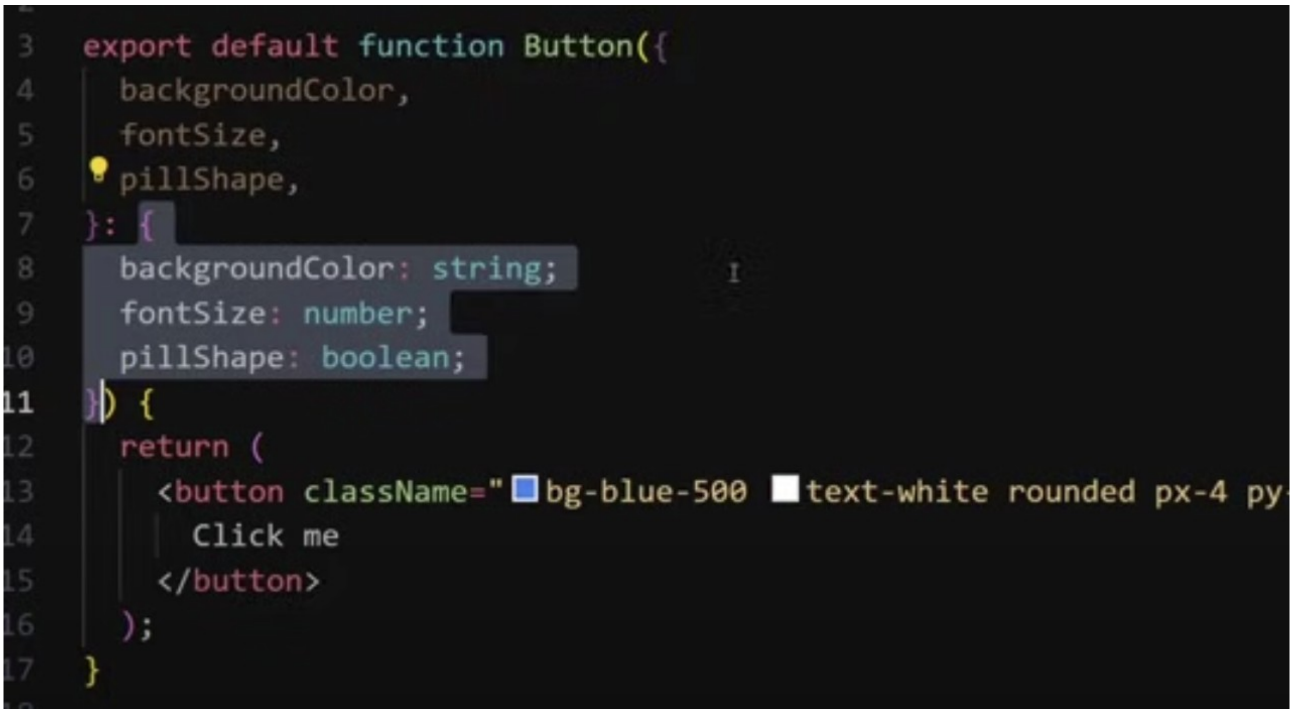
Lets pass another props fontSize to the <Button/>.

We have to declare another type as below.



```
export default function Button({
  backgroundColor,
  fontSize
}: {
  backgroundColor: string;
  fontSize: number;
}) {
  return (
    <button className="bg-blue-500 text-white rounded px-4 py
      Click me
    </button>
  );
}
```

Another prop

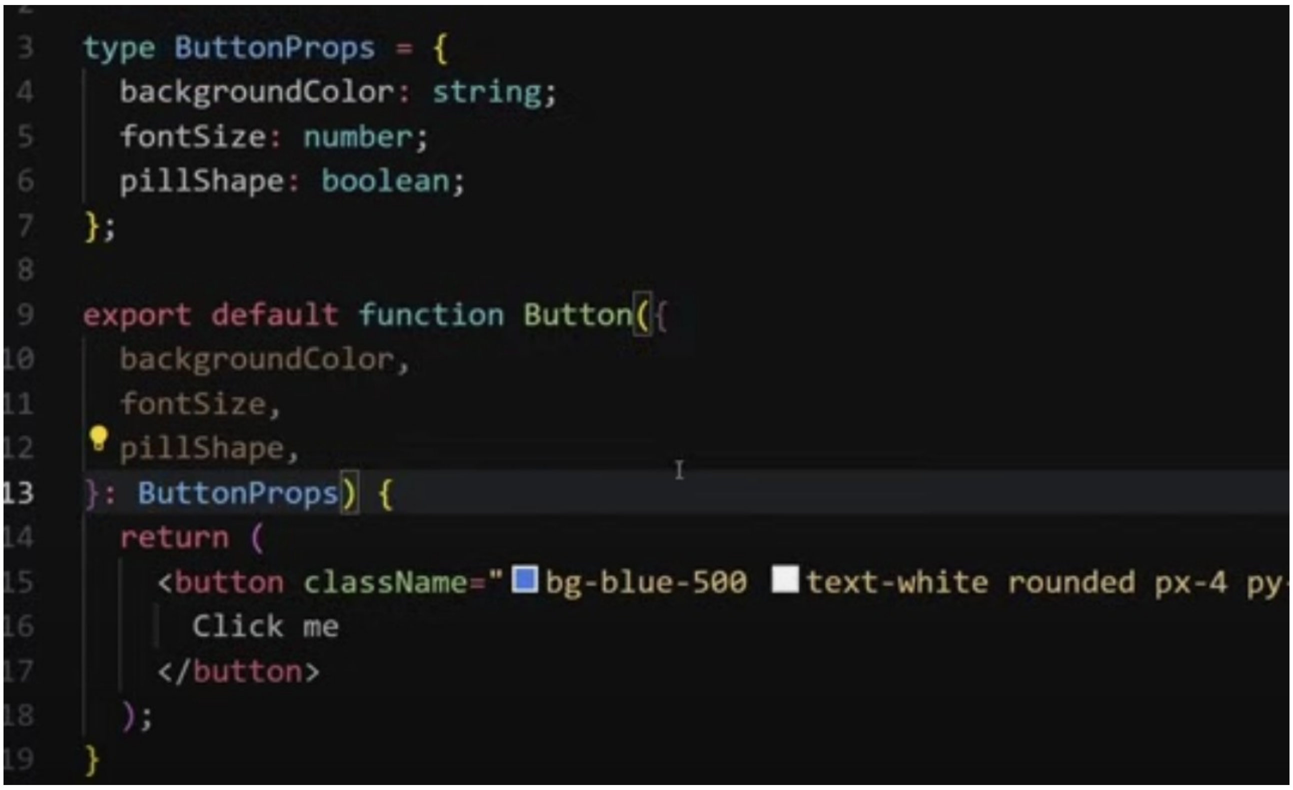


```
export default function Button({
  backgroundColor,
  fontSize,
  pillShape,
}: {
  backgroundColor: string;
  fontSize: number;
  pillShape: boolean;
}) {
  return (
    <button className="bg-blue-500 text-white rounded px-4 py
      Click me
    </button>
  );
}
```

What is now an issue?

If you have a look at the above example, you see that all props and types are inline to the component.
This is a repeating action and it blows up your component – CLEAN CODE !

We have to extract this into a seperate type.

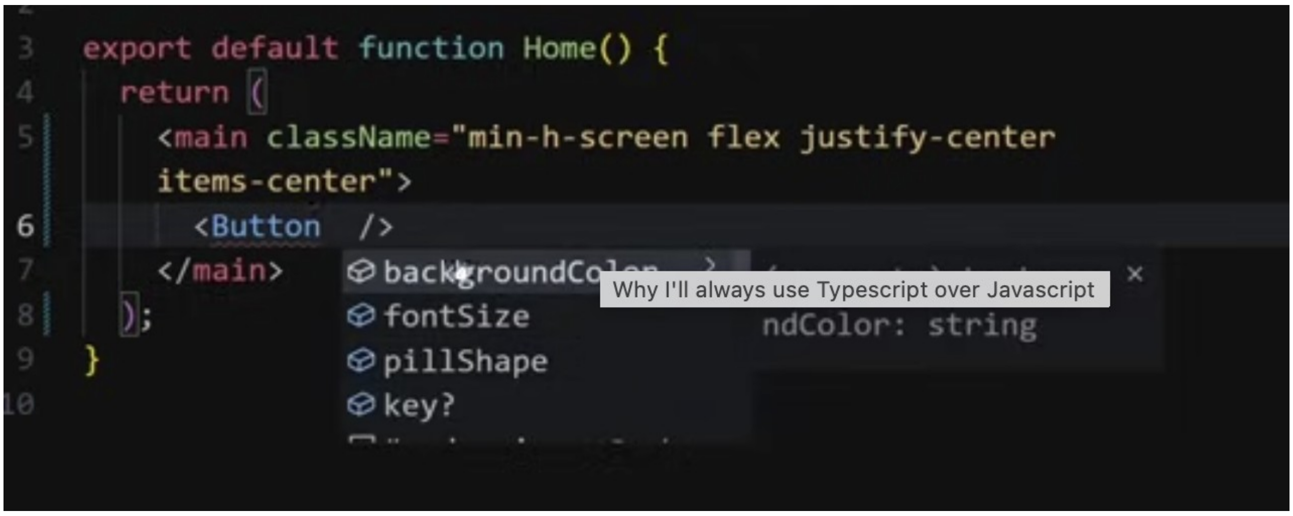


```
type ButtonProps = {
  backgroundColor: string;
  fontSize: number;
  pillShape: boolean;
};

export default function Button({
  backgroundColor,
  fontSize,
  pillShape,
}: ButtonProps) {
  return (
    <button className="bg-blue-500 text-white rounded px-4 py
      Click me
    </button>
  );
}
```

Benefits of this approach:
1. reduced component code
2. clean structured
3. maintanence improved

what types are available?

If you are not aware of the types which are available to an component, TS helps us to get the info.

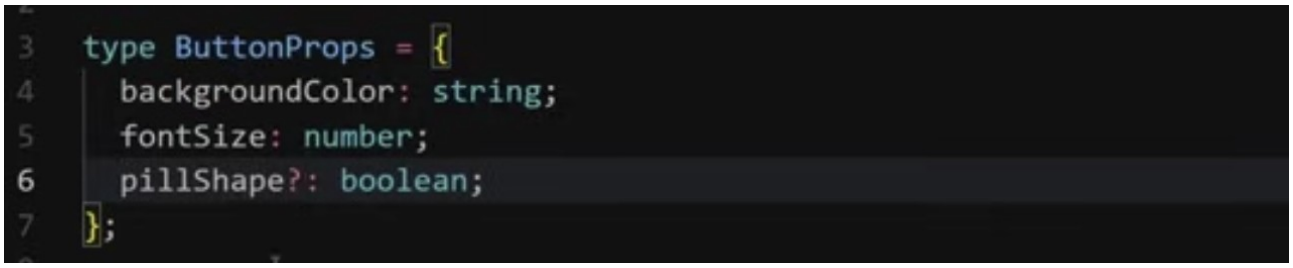


```
export default function Home() {
  return (
    <main className="min-h-screen flex justify-center
    items-center">
      <Button  />
    </main>
  );
}
```

Not passing props to component

In our example we have to pass all props to the <Button/> - because in default, all types are mandatory.

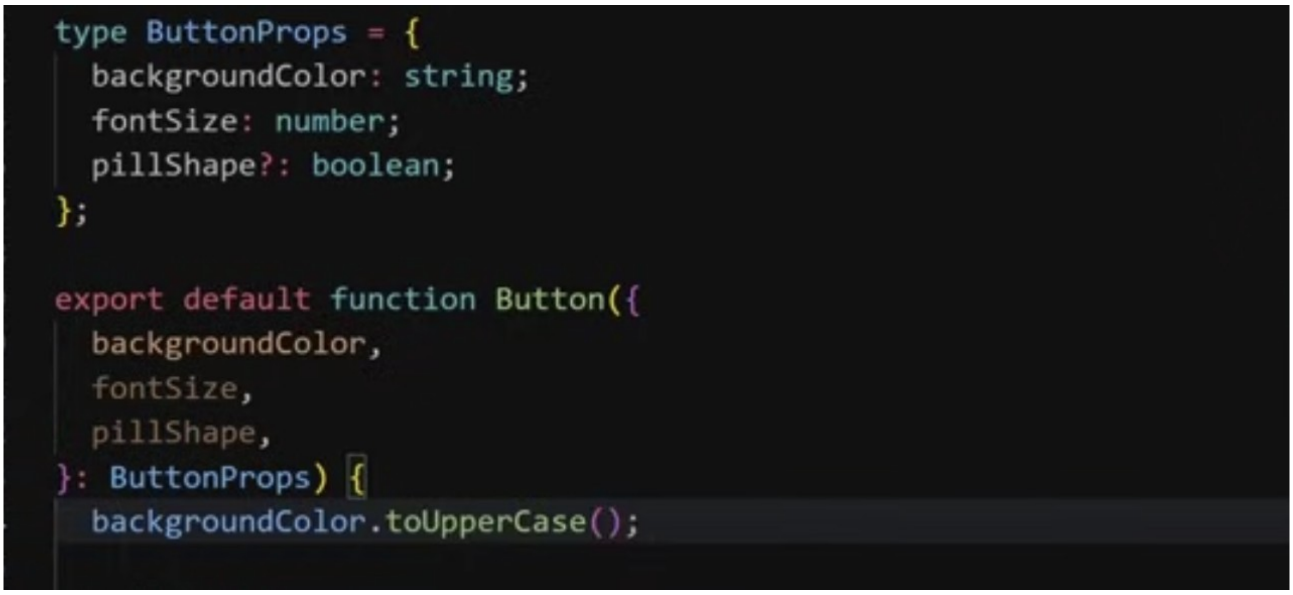
You can make props optional with „?“



```
type ButtonProps = {
  backgroundColor: string;
  fontSize: number;
  pillShape?: boolean;
};
```

Try something:

if I want to work with methods on an prop – e.g. backgroundColor.



```
type ButtonProps = {
  backgroundColor: string;
  fontSize: number;
  pillShape?: boolean;
};

export default function Button({
  backgroundColor,
  fontSize,
  pillShape,
}: ButtonProps) {
  backgroundColor.toUpperCase();
```

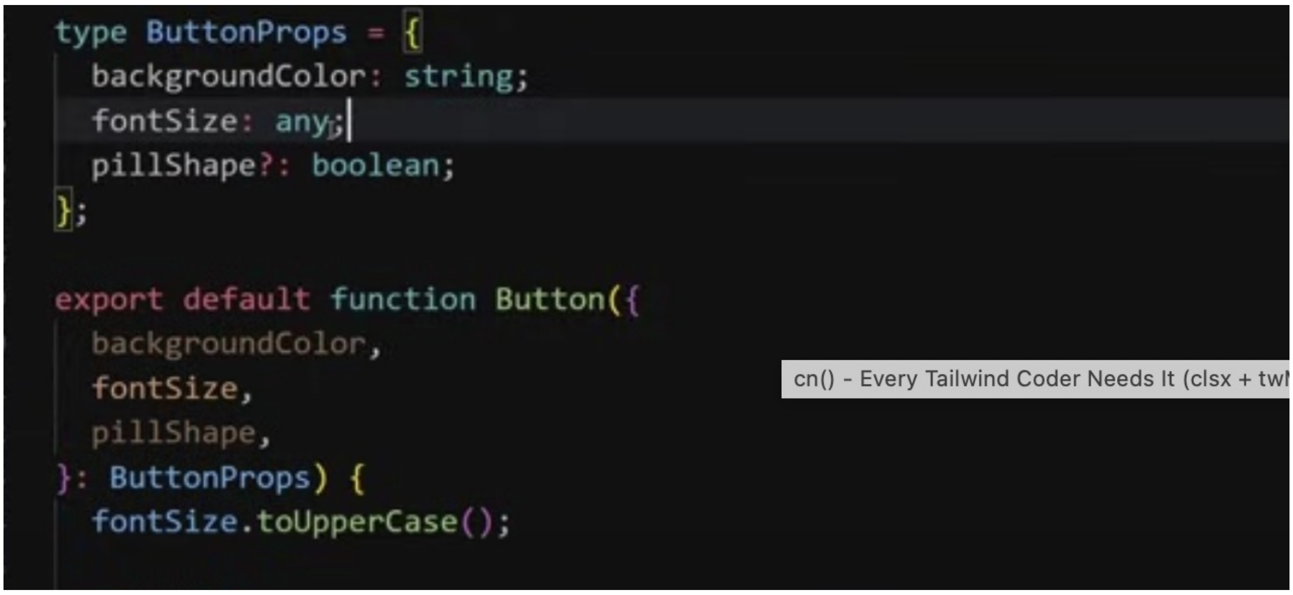
This .toUpperCase() works fine on backgroundColor because it is of type „string“.
On fontSize it will not work because it is of type „number“.

ANY – no type at all…

Working with the type ANY has it pros and cons…

pro-ish:
can pass every data type

con:
there is no data type check possible
application will have issues on the long run
in general it is like not-using-ts-at-all!



```
type ButtonProps = {
  backgroundColor: string;
  fontSize: any;
  pillShape?: boolean;
};

export default function Button({
  backgroundColor,
  fontSize,
  pillShape,
}: ButtonProps) {
  fontSize.toUpperCase();
```

With any you can use the .toUpperCase() method on fontSize without error message from TS.
But it will crash the application during rendering.

PETER WOLF
< Web Developer & Advisor />

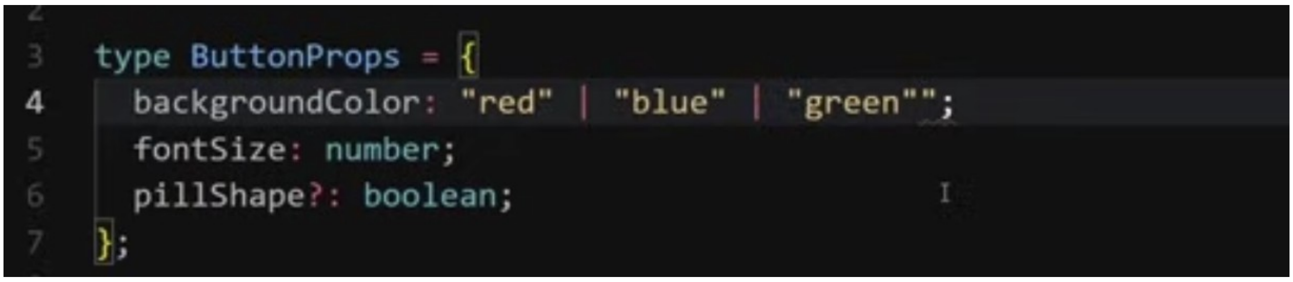
Return type in REACTJS

You can declare a return type in an REACT component by adding JSX.ELEMENT after the ():
But this is redundent, because TS infer (erschließt) that the return type of an component has to be an JSX element by nature of REACTJS.

Specify types concrete

We can declare types with „string“ or we can concrete say the types can be „red“ only.

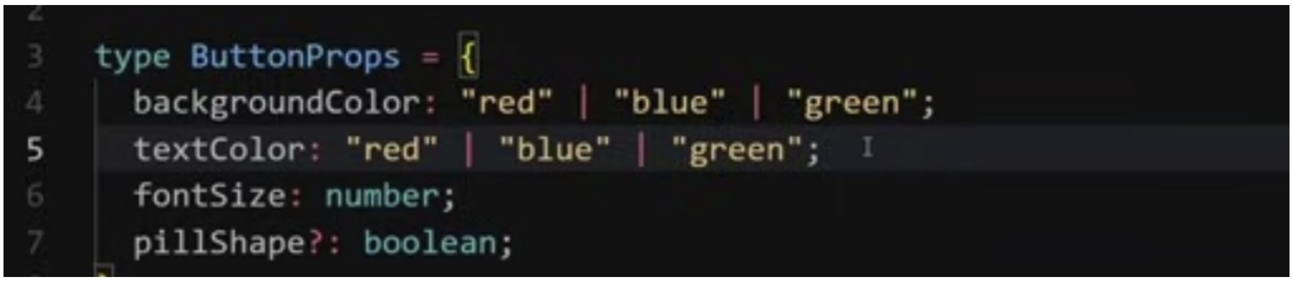
<u>Multiple options:</u>



```
3  type ButtonProps = {
4    backgroundColor: "red" | "blue" | "green"";
5    fontSize: number;
6    pillShape?: boolean;
7  };
```

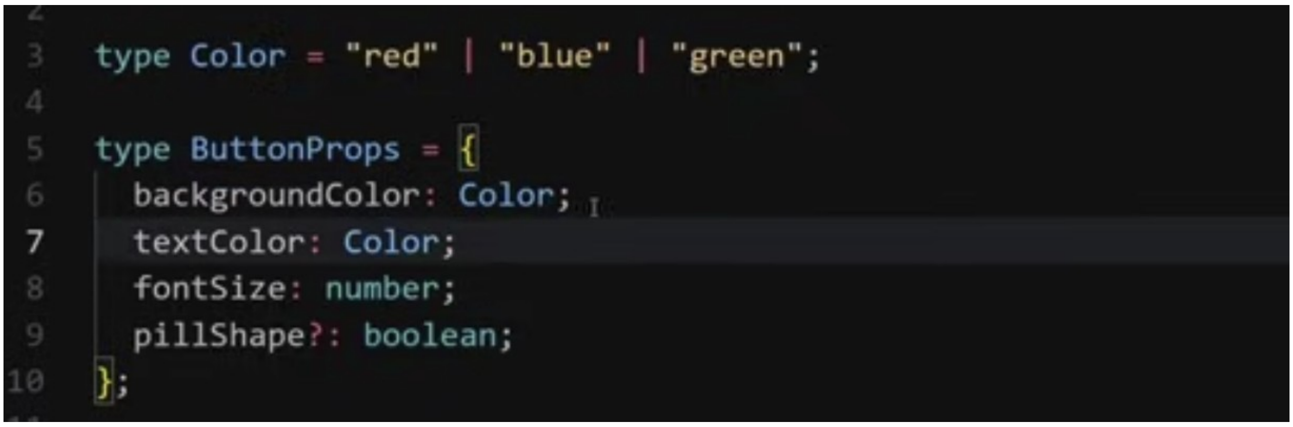
the line 4 is written in **UNION TYPE.**

Reuse the same type for different properties:



```
3  type ButtonProps = {
4    backgroundColor: "red" | "blue" | "green";
5    textColor: "red" | "blue" | "green";
6    fontSize: number;
7    pillShape?: boolean;
```

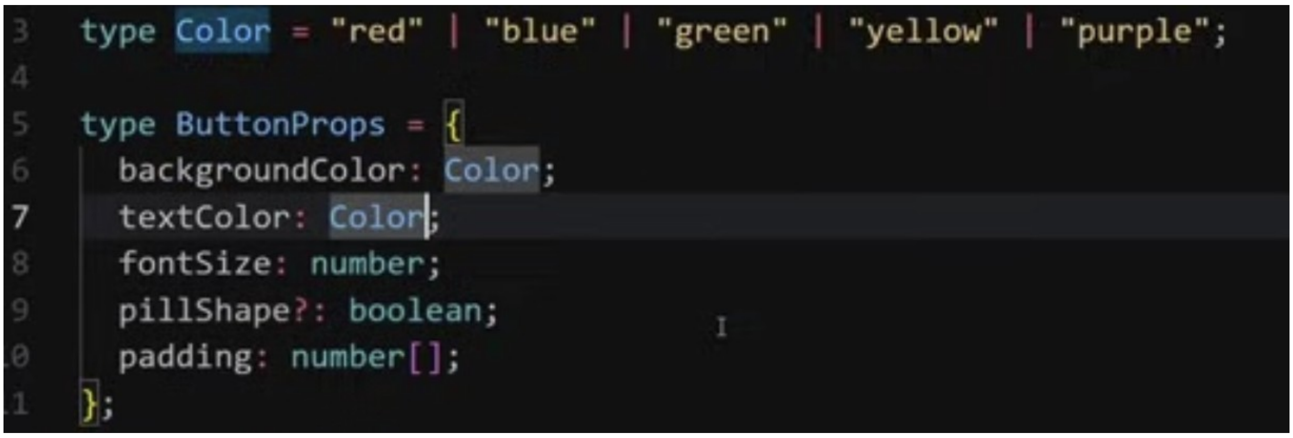
To follow the D-R-Y Principle, we can extract type like following:



```
type Color = "red" | "blue" | "green";

type ButtonProps = {
  backgroundColor: Color;
  textColor: Color;
  fontSize: number;
  pillShape?: boolean;
};
```

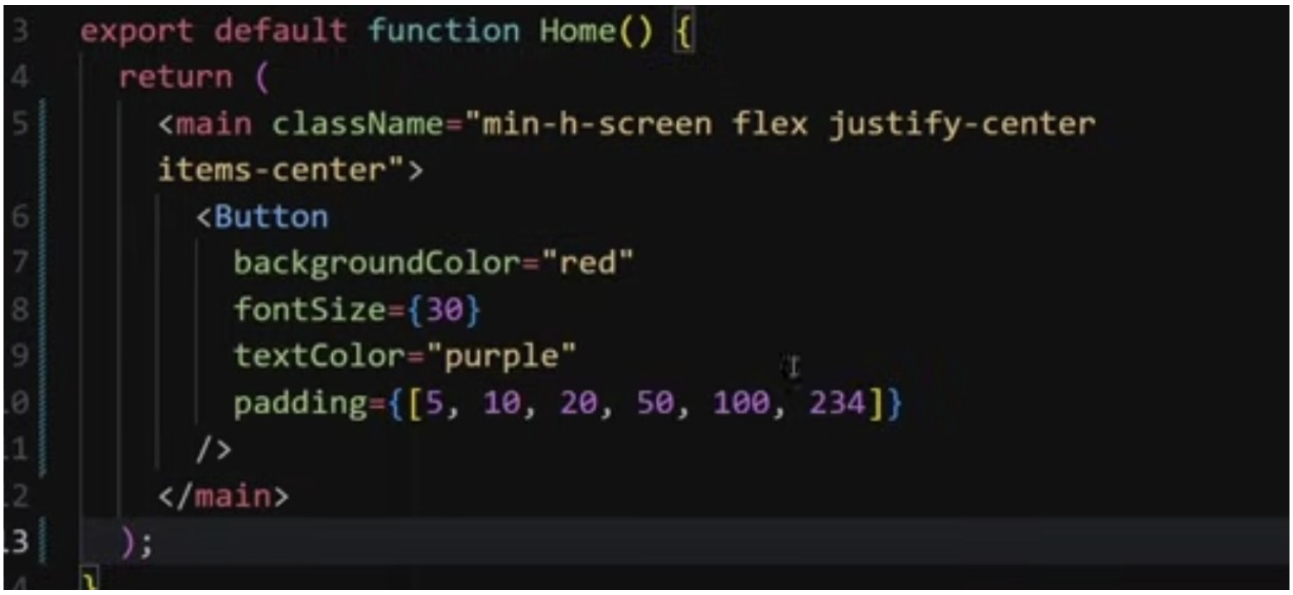
typing ARRAYS

Lets add the property PADDING to props. This could be an array [top, right, bottom, left]



```
type Color = "red" | "blue" | "green" | "yellow" | "purple";

type ButtonProps = {
  backgroundColor: Color;
  textColor: Color;
  fontSize: number;
  pillShape?: boolean;
  padding: number[];
};
```

lets pass the padding prop to the <Button/>



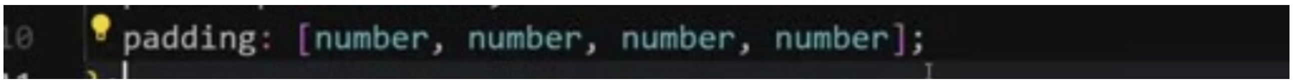
```
export default function Home() {
  return (
    <main className="min-h-screen flex justify-center
    items-center">
      <Button
        backgroundColor="red"
        fontSize={30}
        textColor="purple"
        padding={[5, 10, 20, 50, 100, 234]}
      />
    </main>
  );
}
```

here is an issue:

I can add more than the 4 entries of the array at the moment.

The TUPLE is the solution:

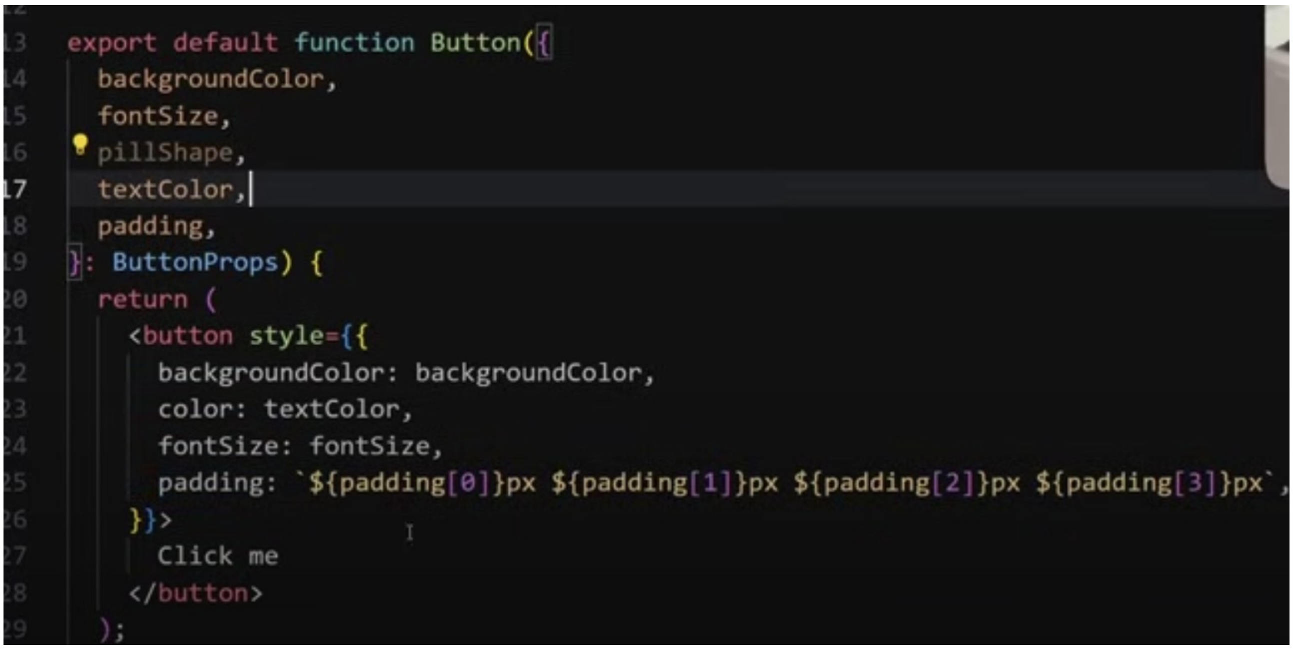
With an TUPLE you specify the array entries.



```
  padding: [number, number, number, number];
```

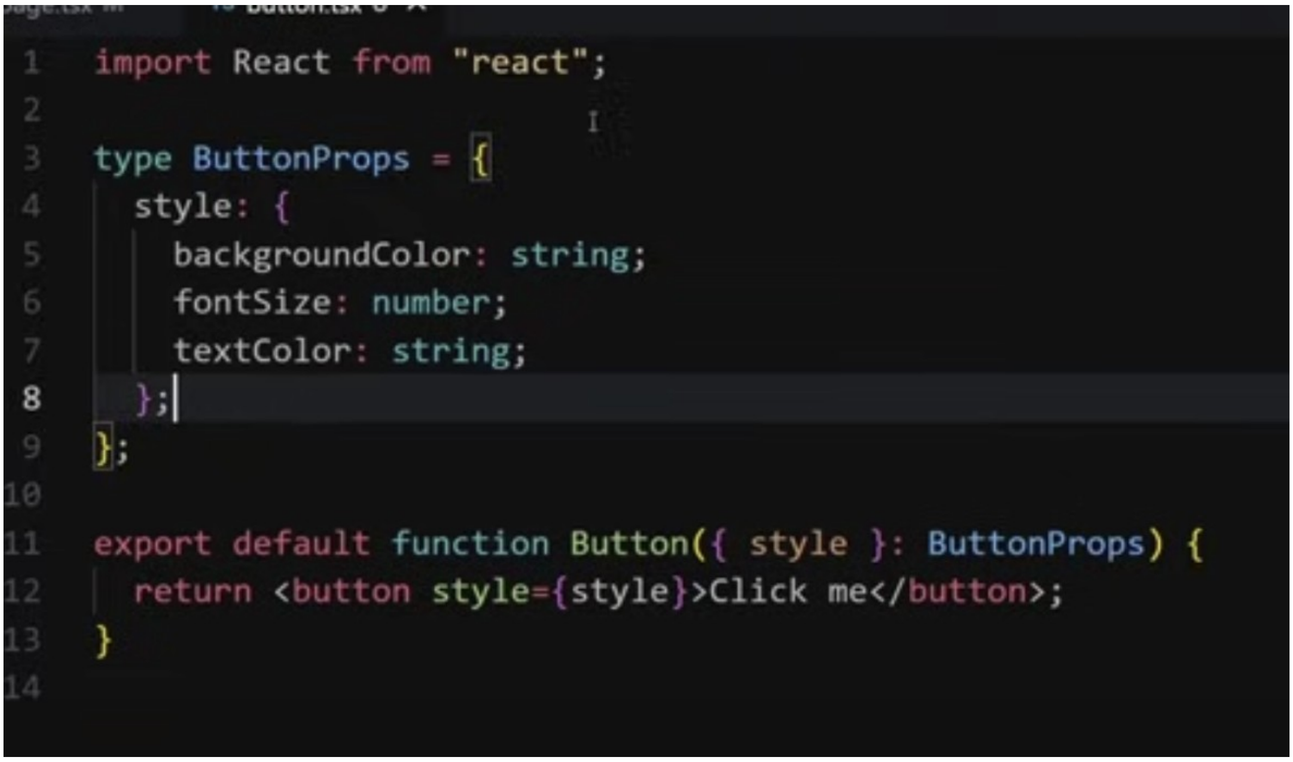
PETER WOLF
< Web Developer & Advisor />

NOW I can use the props in my <Button/>



```
export default function Button({
  backgroundColor,
  fontSize,
  pillShape,
  textColor,
  padding,
}: ButtonProps) {
  return (
    <button style={{
      backgroundColor: backgroundColor,
      color: textColor,
      fontSize: fontSize,
      padding: `${padding[0]}px ${padding[1]}px ${padding[2]}px ${padding[3]}px`,
    }}>
      Click me
    </button>
  );
```

Reduce code with passing only one prop.

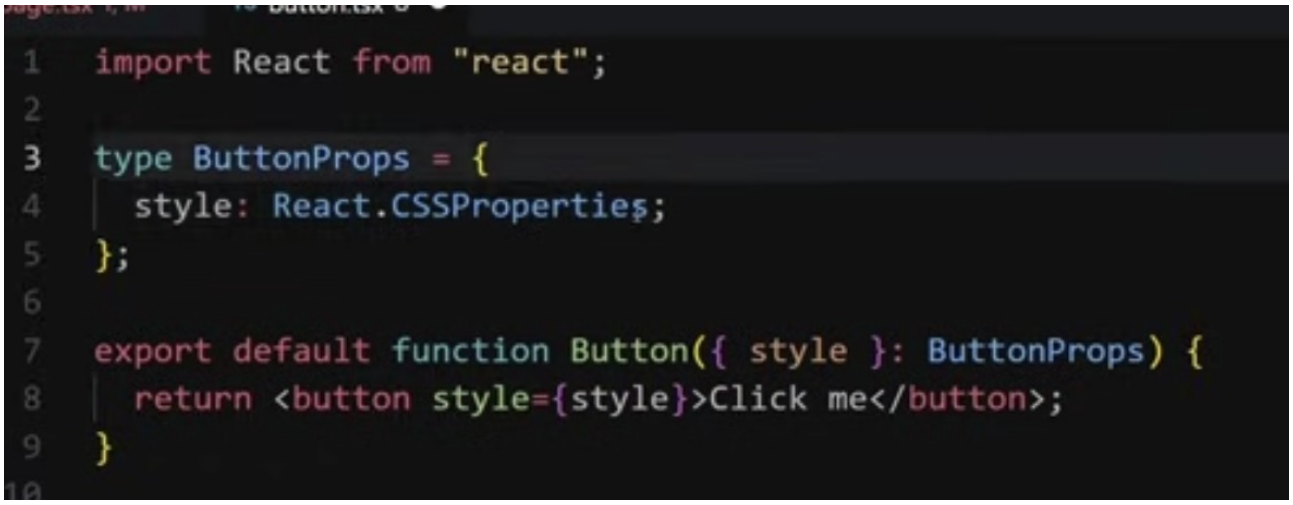


```
import React from "react";

type ButtonProps = {
  style: {
    backgroundColor: string;
    fontSize: number;
    textColor: string;
  };
};

export default function Button({ style }: ButtonProps) {
  return <button style={style}>Click me</button>;
}
```

The „style“ property is an object and it will get kind of combersome (unhandlich) for managing.

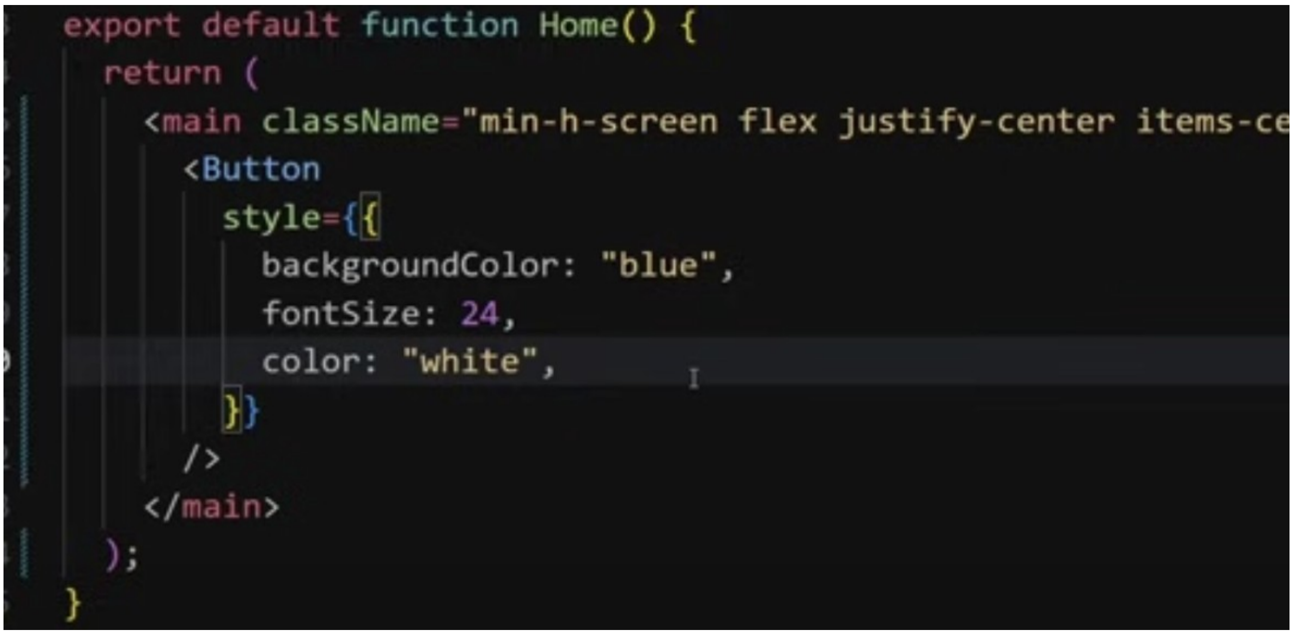
REACT helps us with CSS.PROPERTIES – it comes with the core of react.

Recreate your code like this:



```
import React from "react";

type ButtonProps = {
  style: React.CSSProperties;
};

export default function Button({ style }: ButtonProps) {
  return <button style={style}>Click me</button>;
}
```

Now we specify „style“ as an React.CSSProperties object and I can pass all css props to it.



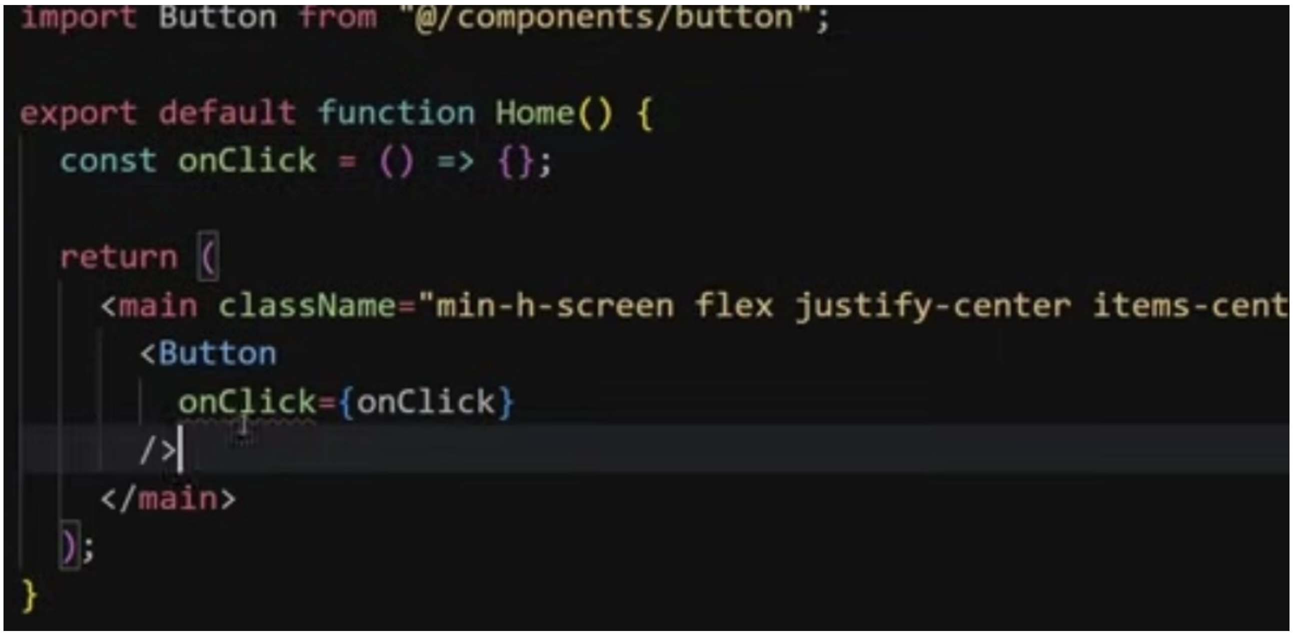
```
export default function Home() {
  return (
    <main className="min-h-screen flex justify-center items-ce
      <Button
        style={{
          backgroundColor: "blue",
          fontSize: 24,
          color: "white",
        }}
      />
    </main>
  );
}
```

BUT: keep in mind, that „textColor“ is not a css property – it has to be „color“.

Type a function as prop

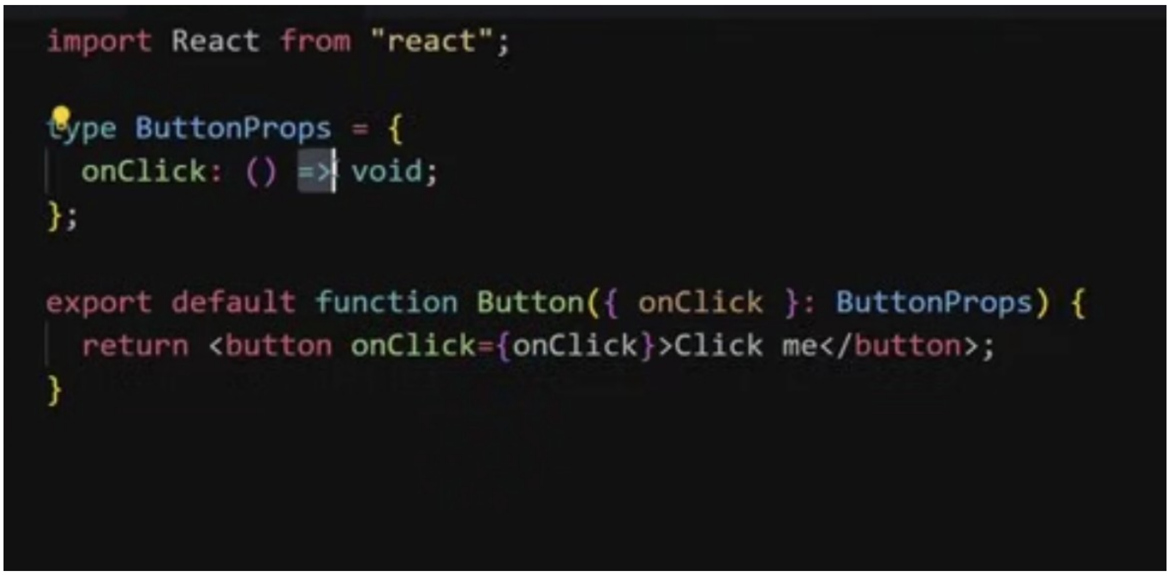
We want to pass a function as a prop => how do we type the function?

This is our example:



```tsx
import Button from "@/components/button";

export default function Home() {
  const onClick = () => {};

  return (
    <main className="min-h-screen flex justify-center items-cent
      <Button
        onClick={onClick}
      />
    </main>
  );
}
```

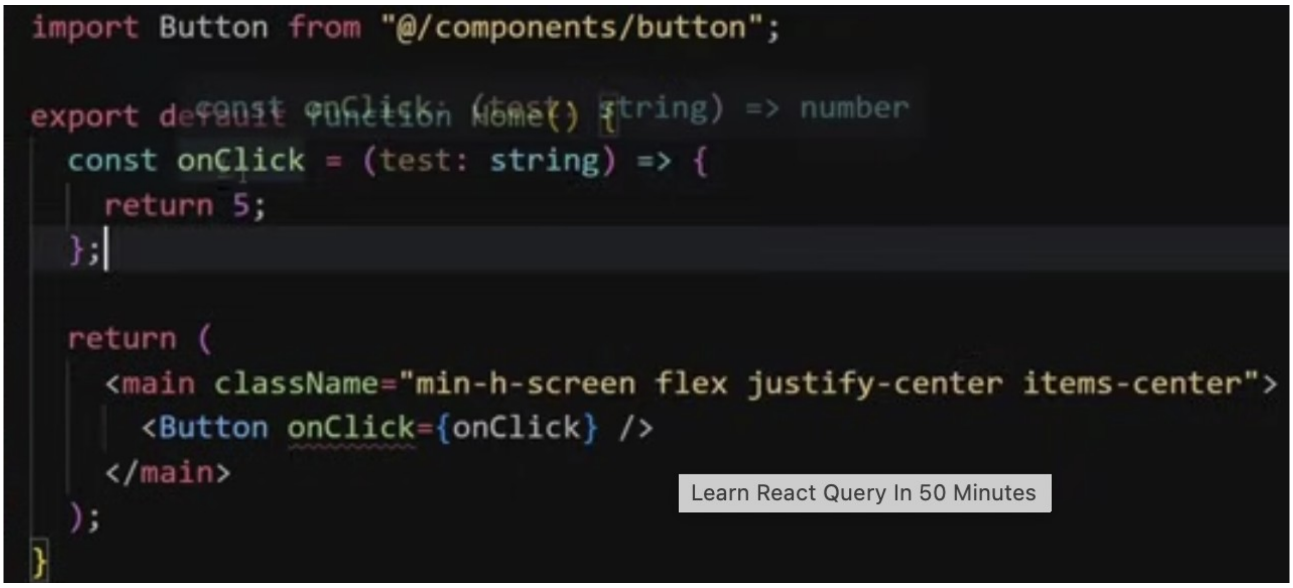
with the above example the function returns nothing…this is the typing of the function now:



```tsx
import React from "react";

type ButtonProps = {
  onClick: () => void;
};

export default function Button({ onClick }: ButtonProps) {
  return <button onClick={onClick}>Click me</button>;
}
```

Return value and parameter passing

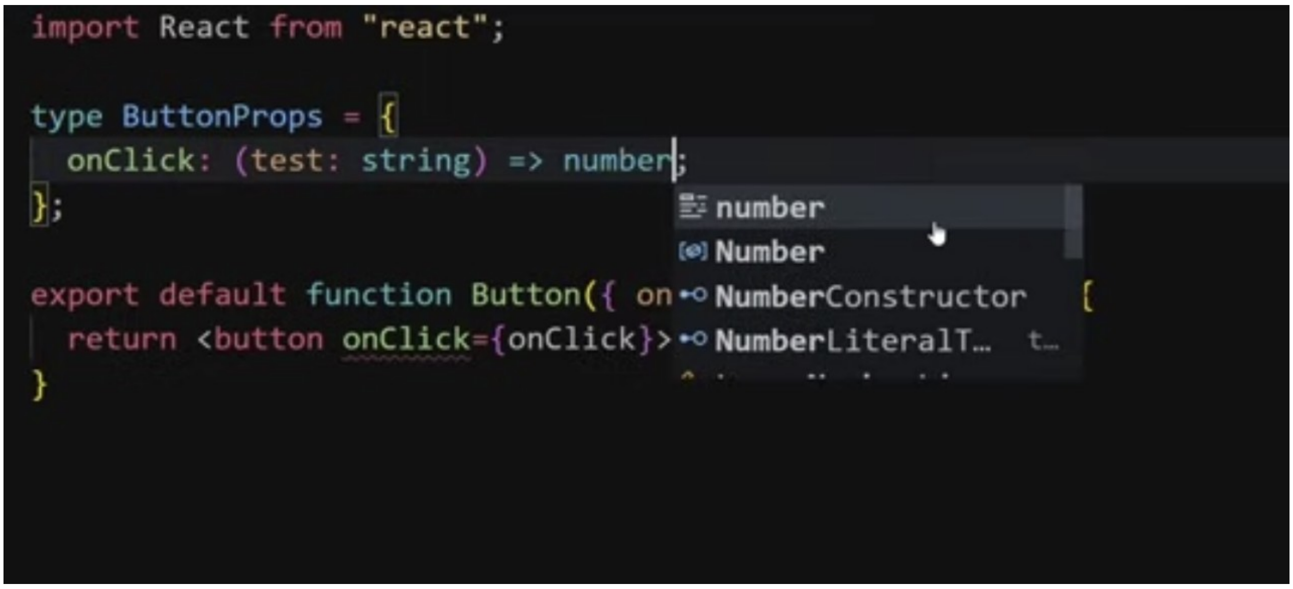
If the function returns something and it receives an attribute the typing looks like this:



```
import Button from "@/components/button";

export default function Home() {
  const onClick = (test: string) => {
    return 5;
  };

  return (
    <main className="min-h-screen flex justify-center items-center">
      <Button onClick={onClick} />
    </main>
  );
}
```

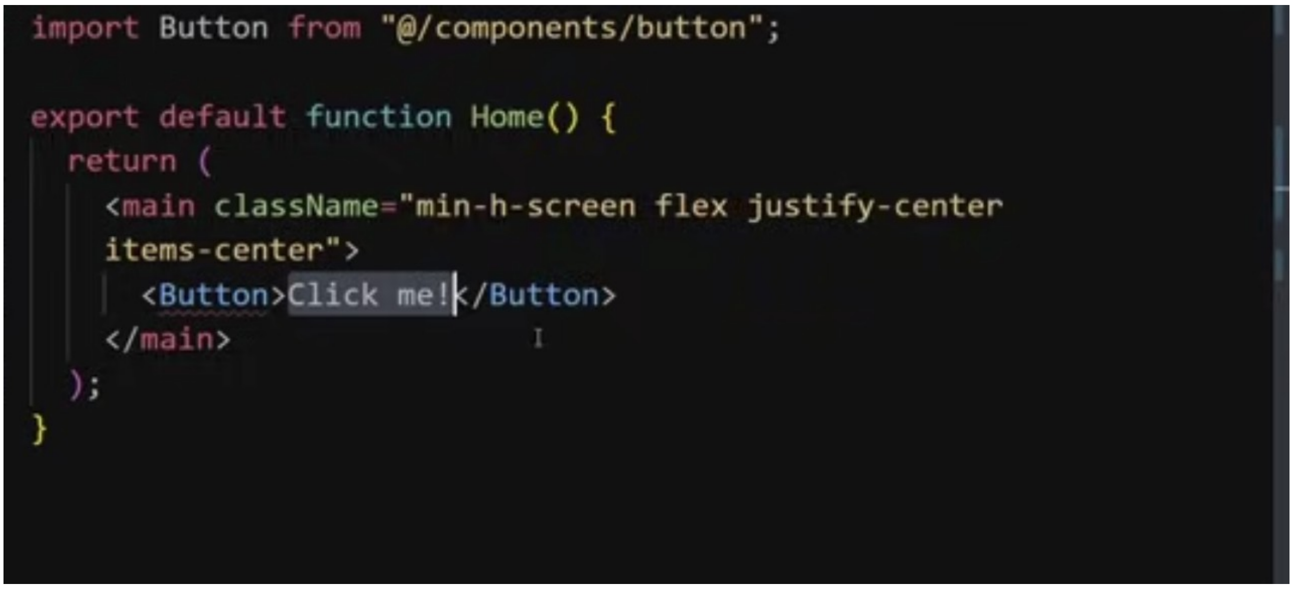
const onClick: (test: string) => number
Learn React Query In 50 Minutes

we define the attribute type as „string“. Within the <Button/> we do this following:



```
import React from "react";

type ButtonProps = {
  onClick: (test: string) => number;
};

export default function Button({ on
  return <button onClick={onClick}>
}
```

Type {children} in TS



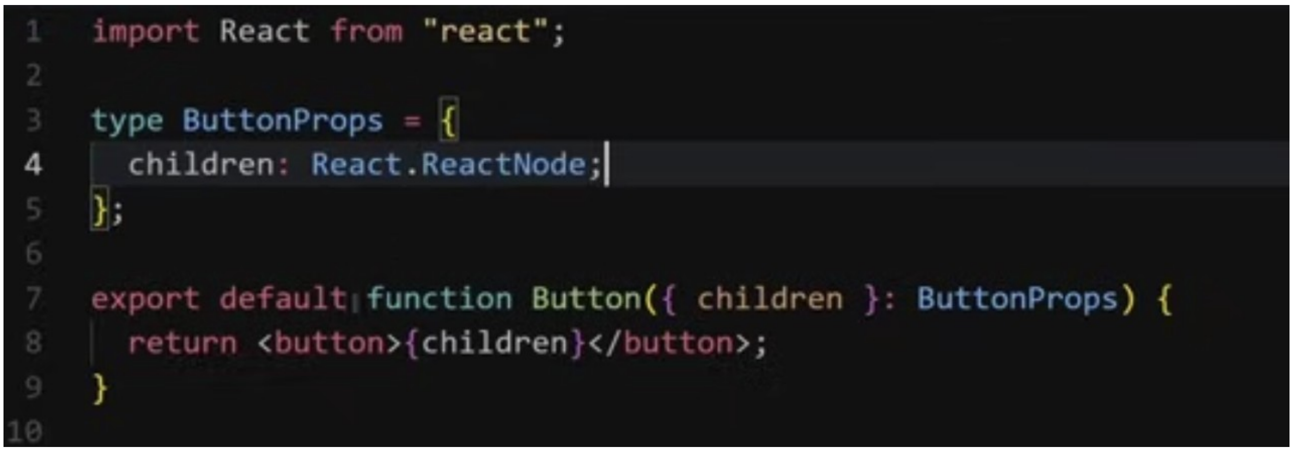
```
import Button from "@/components/button";

export default function Home() {
  return (
    <main className="min-h-screen flex justify-center
    items-center">
      <Button>Click me!</Button>
    </main>
  );
}
```

In the above example we didn´t use a self-closing component <Button/> - instead we write it like above.
<Button>Click Me!</Button>

„Click Me“ in the ReactWorld is a „children“

React.Node vs JSX.Element

To type the children in TS – do this



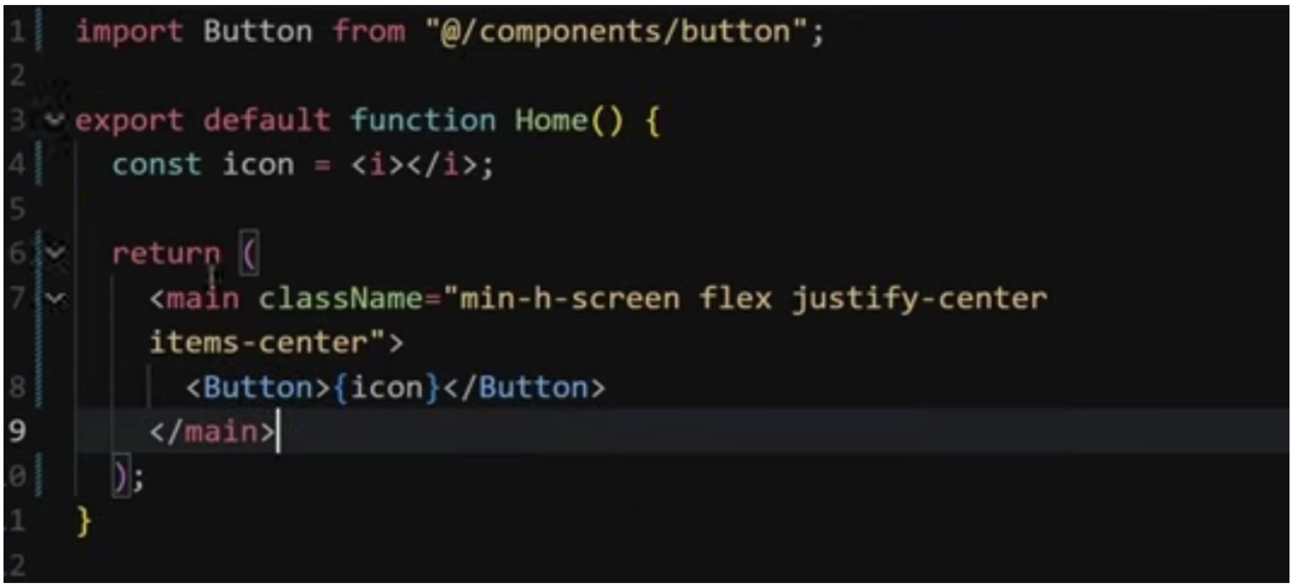
```
import React from "react";

type ButtonProps = {
  children: React.ReactNode;
};

export default function Button({ children }: ButtonProps) {
  return <button>{children}</button>;
}
```

React.ReactNode allows us to pass all types of data and html elements.

This approach is a quite GENERAL one – you can pass everything you want.

But sometimes you want to specify it more concrete:

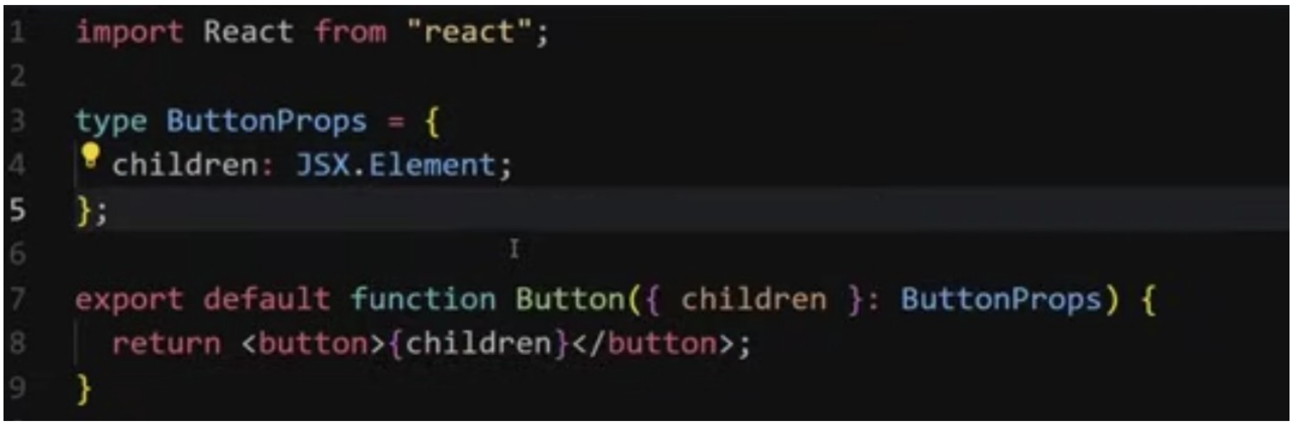
Let us define a icon element and pass it as children to the <Button></Button> component.



```
import Button from "@/components/button";

export default function Home() {
  const icon = <i></i>;

  return (
    <main className="min-h-screen flex justify-center
    items-center">
      <Button>{icon}</Button>
    </main>
  );
}
```

The icon is empty – only for demo!
With the React.ReactNode we can do that – because this is a general approach.

Now - we pass the icon and define the type more specific…

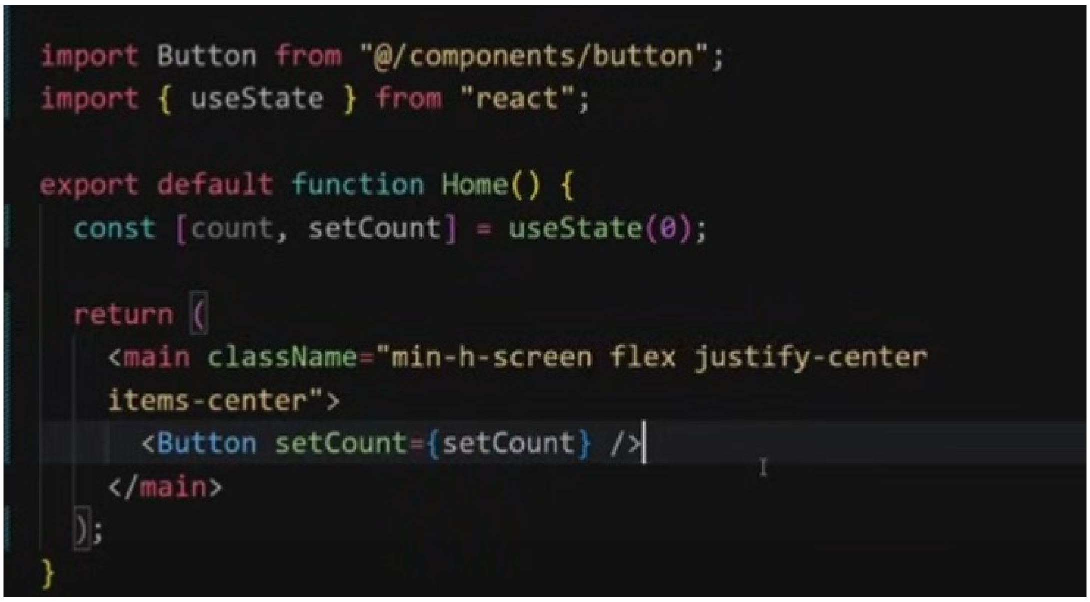


```
import React from "react";

type ButtonProps = {
  children: JSX.Element;
};

export default function Button({ children }: ButtonProps) {
  return <button>{children}</button>;
}
```

If I now try to pass a text instead of a jsx element, we get an error!

Typing useState setter function

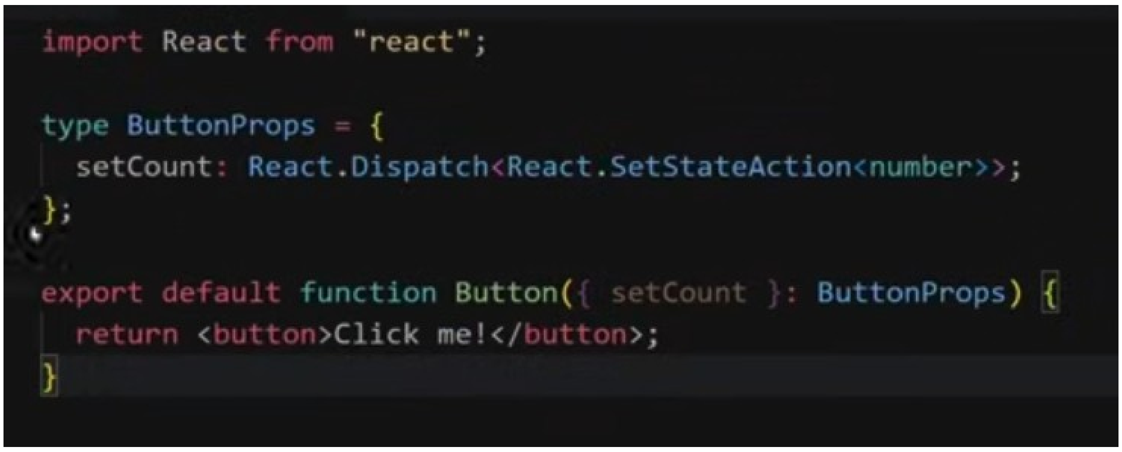
We want to pass a setter function of useState to a component.
First we have to adjust the code a bit!!!



```
import Button from "@/components/button";
import { useState } from "react";

export default function Home() {
  const [count, setCount] = useState(0);

  return (
    <main className="min-h-screen flex justify-center
    items-center">
      <Button setCount={setCount} />
    </main>
  );
}
```

We want now to type the setter function to accept it in our component
If we hover over the setCount setter function, we will get the type information.

The code below looks kind of crazy. The good thing is, you don´t have to memorize it.

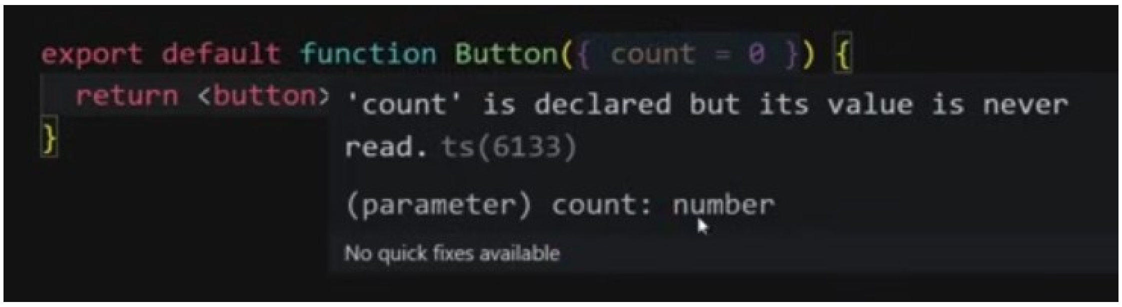


```
import React from "react";

type ButtonProps = {
  setCount: React.Dispatch<React.SetStateAction<number>>;
};

export default function Button({ setCount }: ButtonProps) {
  return <button>Click me!</button>;
}
```

PETER WOLF
< Web Developer & Advisor />

Default prop values

If you define a default prop value, you don´t have to type the props at all. With placing a default value, typescript set the type to the given default value type.



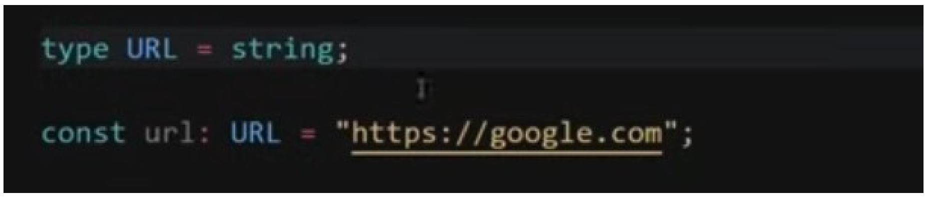
```
export default function Button({ count = 0 }) {
  return <button> 'count' is declared but its value is never
}               read. ts(6133)

                (parameter) count: number
                No quick fixes available
```

Type Alias VS Interface

With interface we can only describe an object. We cannot assign specific values to it.

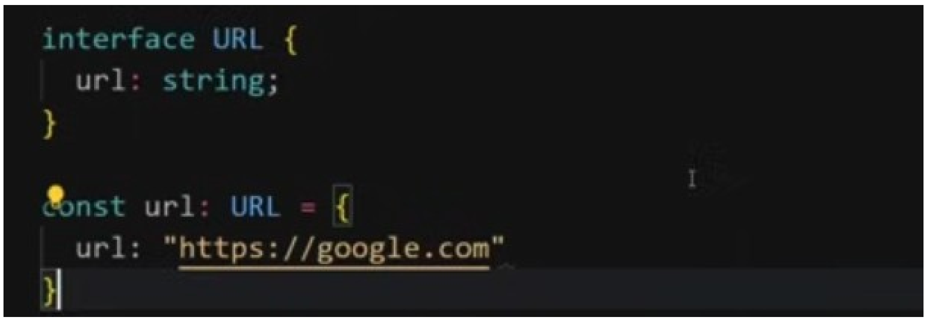
See this example:

With type I can do this:



```
type URL = string;

const url: URL = "https://google.com";
```

BUT I CANNOT do it with INTERFACE:
I have to modify my URL as an object to use the interface



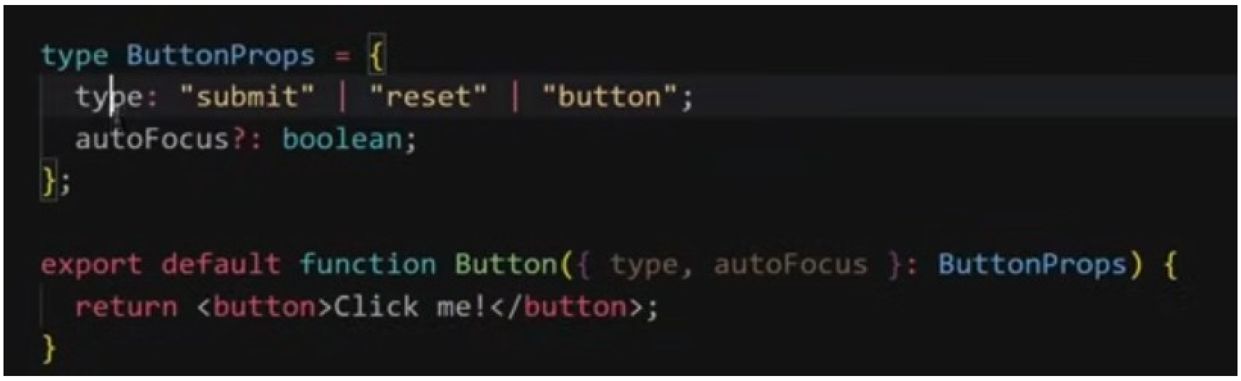
```
interface URL {
  url: string;
}

const url: URL = {
  url: "https://google.com"
}
```

In the real world you often use TYPE rather than INTERFACE.

ComponentPropsWithRef

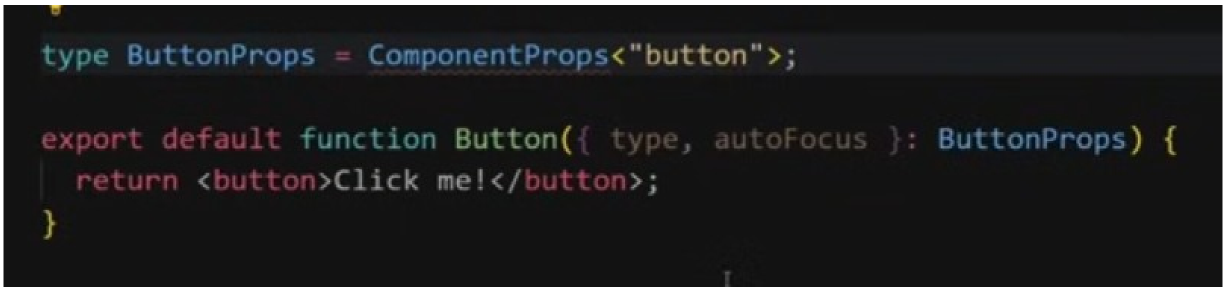
With our <Button/> we want to maybe use several attributes related to an <button></button>

e.g:



```
type ButtonProps = {
  type: "submit" | "reset" | "button";
  autoFocus?: boolean;
};

export default function Button({ type, autoFocus }: ButtonProps) {
  return <button>Click me!</button>;
}
```

But in the above example, the <Button/> accepts only two possible attributes.
What if we want to use more attributes for <button>?
We can ether specify them individually or we use **ComponentProps<>**

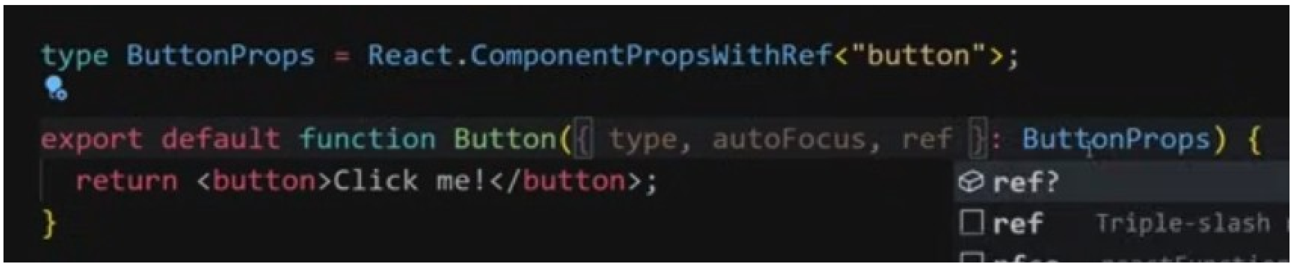


```
type ButtonProps = ComponentProps<"button">;

export default function Button({ type, autoFocus }: ButtonProps) {
  return <button>Click me!</button>;
}
```

Advance advice:

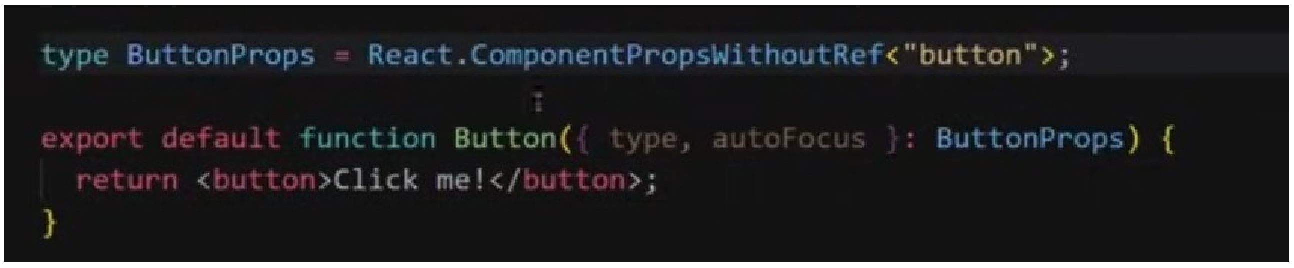
TS recommends to **specify as explicit** as possible.

If you pass besides the native attributes for example REF – then you have to declare it like this.



```
type ButtonProps = React.ComponentPropsWithRef<"button">;

export default function Button({ type, autoFocus, ref }: ButtonProps) {
  return <button>Click me!</button>;
}
```

OR the opposite withoutRef:



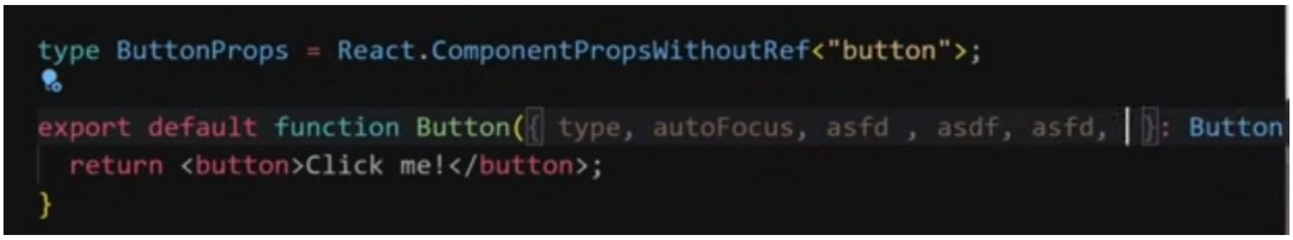
```
type ButtonProps = React.ComponentPropsWithoutRef<"button">;

export default function Button({ type, autoFocus }: ButtonProps) {
  return <button>Click me!</button>;
}
```

PETER WOLF
< Web Developer & Advisor />

The REST OPERATOR

With the above example you have a solution to check all possible native attributes by type. But you have to accept them in the component.

You can ether write the used attributes to the component…

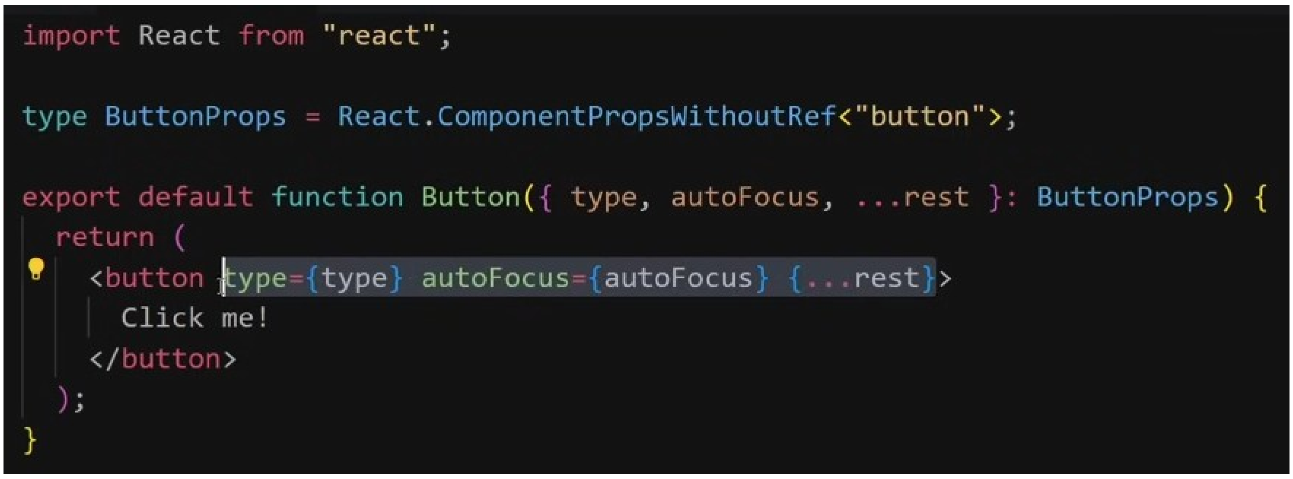


```
type ButtonProps = React.ComponentPropsWithoutRef<"button">;

export default function Button({ type, autoFocus, asfd , asdf, asfd, | }: Button
  return <button>Click me!</button>;
}
```

BUT there is a more elegant way to pass attributes:

in JavaScript there is the {...rest} operator. => ...rest can be named as well like ...props or else…

The rest operator provides an ARRAY with all passed attributes to the component.



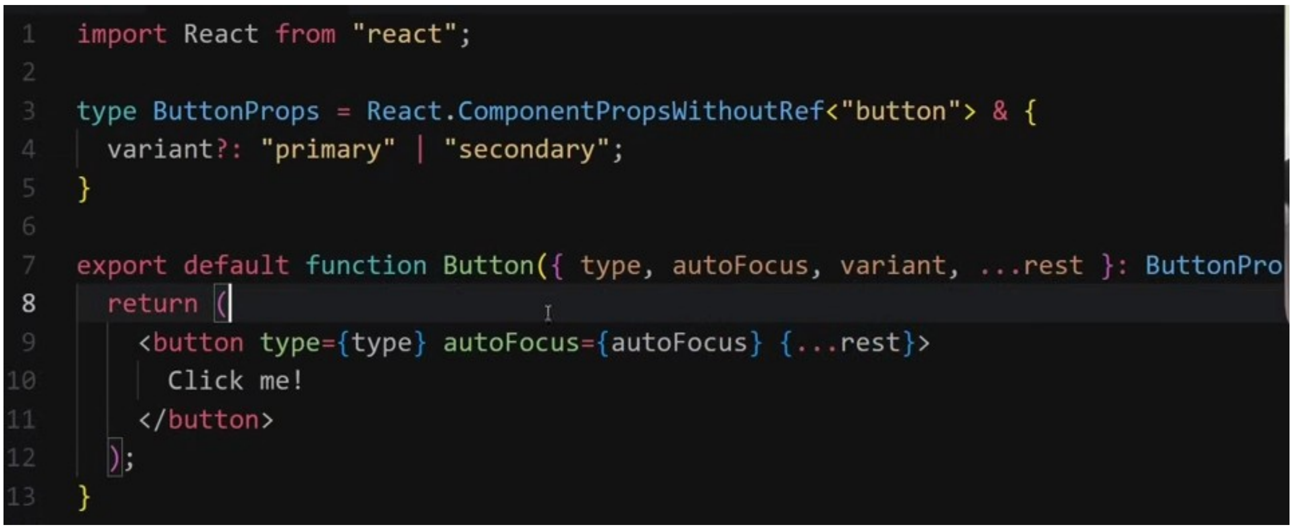
```
import React from "react";

type ButtonProps = React.ComponentPropsWithoutRef<"button">;

export default function Button({ type, autoFocus, ...rest }: ButtonProps) {
  return (
    <button type={type} autoFocus={autoFocus} {...rest}>
      Click me!
    </button>
  );
}
```

Now with this above we can accept all the native attributes a button can have.

Intersection (&)

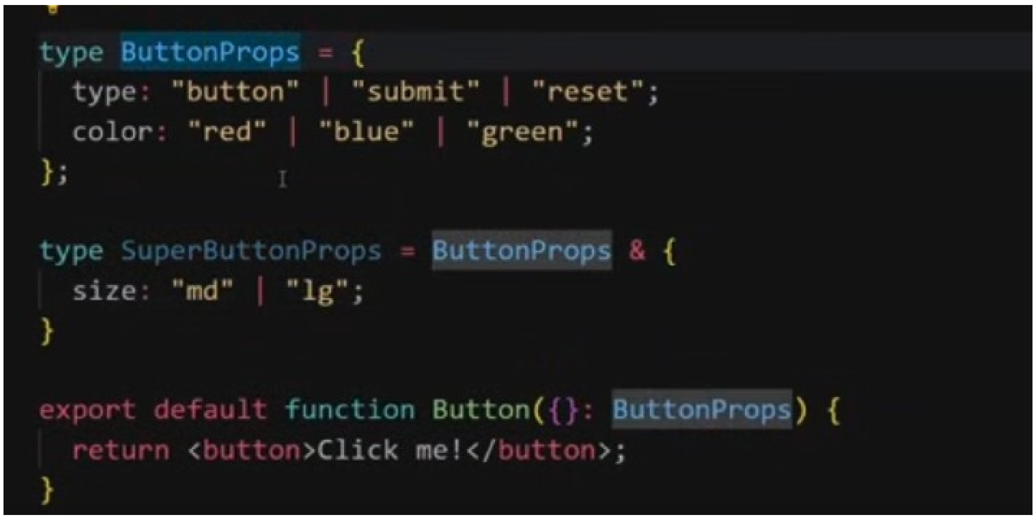
add additional attributes outside the native scope
What if, we want to pass additional props which are outside the scope of the native attributes – example <button>?

With the & we can pass additonal props to the component (in line 3)


```tsx
1  import React from "react";
2
3  type ButtonProps = React.ComponentPropsWithoutRef<"button"> & {
4    variant?: "primary" | "secondary";
5  }
6
7  export default function Button({ type, autoFocus, variant, ...rest }: ButtonPro
8    return (
9      <button type={type} autoFocus={autoFocus} {...rest}>
10       Click me!
11     </button>
12   );
13 }
```

<u>Pretty nice use case for intersection</u>

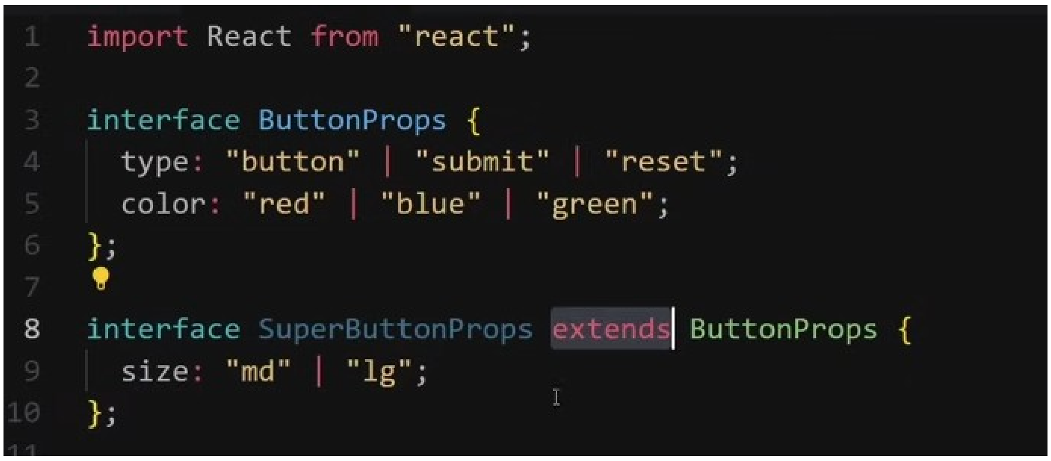
there are two buttons in the component. ButtonProps will be passed to the SuperButtonProps with & to be able to have the same attributes AND some more specific ones.


```tsx
type ButtonProps = {
  type: "button" | "submit" | "reset";
  color: "red" | "blue" | "green";
};

type SuperButtonProps = ButtonProps & {
  size: "md" | "lg";
}

export default function Button({}: ButtonProps) {
  return <button>Click me!</button>;
}
```

PETER WOLF
< Web Developer & Advisor />

Prop Typing with INTERFACE

We can use Interface typing as well to make the above example work.

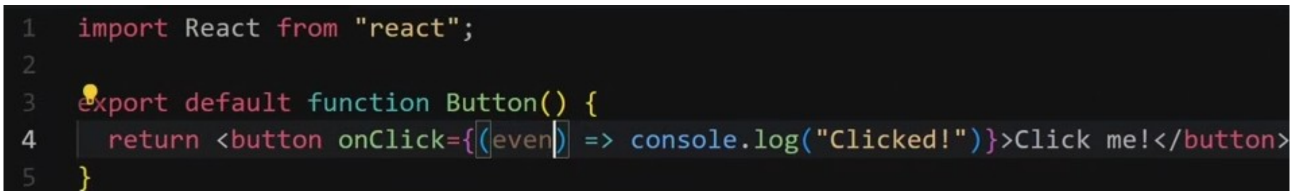


```
import React from "react";

interface ButtonProps {
  type: "button" | "submit" | "reset";
  color: "red" | "blue" | "green";
};

interface SuperButtonProps extends ButtonProps {
  size: "md" | "lg";
};
```

With ***&*** it called INTERSECTING
With ***extends*** it call EXTENDING

Typing Event Handler

Let us define an onClick event on the <button> example.

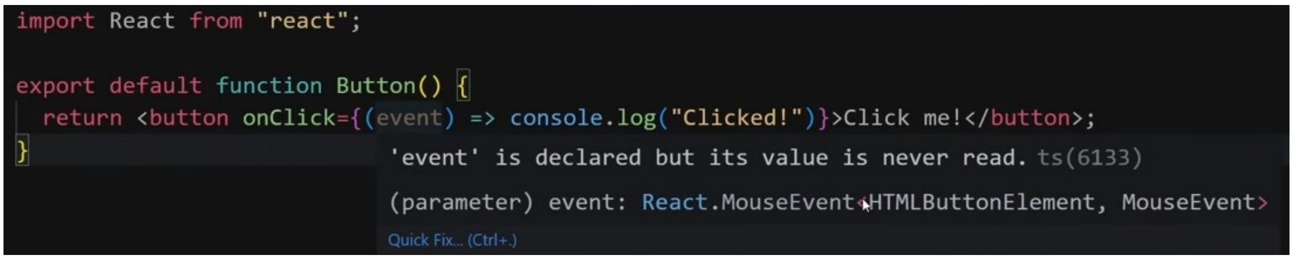


```
import React from "react";

export default function Button() {
  return <button onClick={(even) => console.log("Clicked!")}>Click me!</button>
}
```

This function is only available on this special onClick. TS sets the type for this event attribute out of the core. So we don´t need to type it extra.
TS has context here which declares the type for the function attributes automatically – and no error appears.

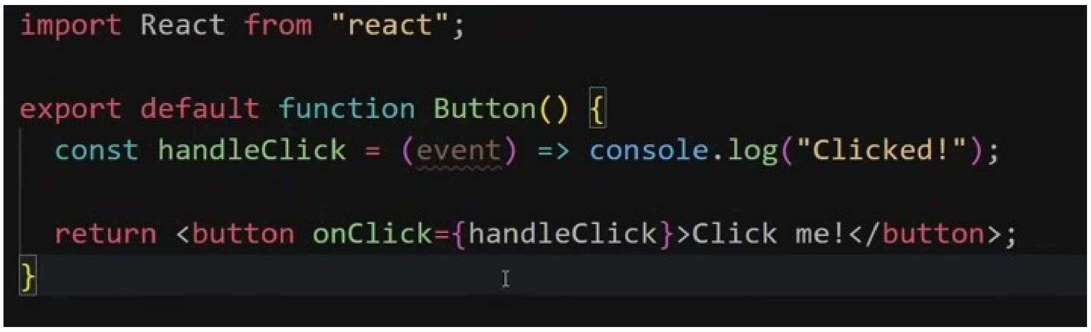
PETER WOLF
< Web Developer & Advisor />

On hover over the „event“ attribute, you see this:



```
import React from "react";

export default function Button() {
  return <button onClick={(event) => console.log("Clicked!")}>Click me!</button>;
}
                    'event' is declared but its value is never read. ts(6133)
                    (parameter) event: React.MouseEvent<HTMLButtonElement, MouseEvent>
                    Quick Fix... (Ctrl+.)
```

What if we extract the function and pass it to the button?

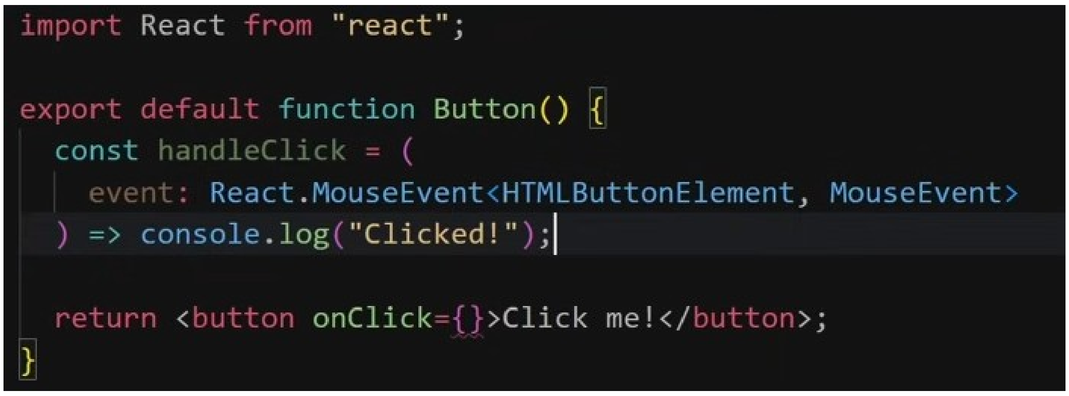


```
import React from "react";

export default function Button() {
  const handleClick = (event) => console.log("Clicked!");

  return <button onClick={handleClick}>Click me!</button>;
}
```

TS doesn´t know what the „event“ attribute is because we can use this extracted function anywhere else in the application.

<u>Now we need to type the function.</u>

Note: this typing looks kind of complicated. You can help yourself by checking the infobox from the inline function.

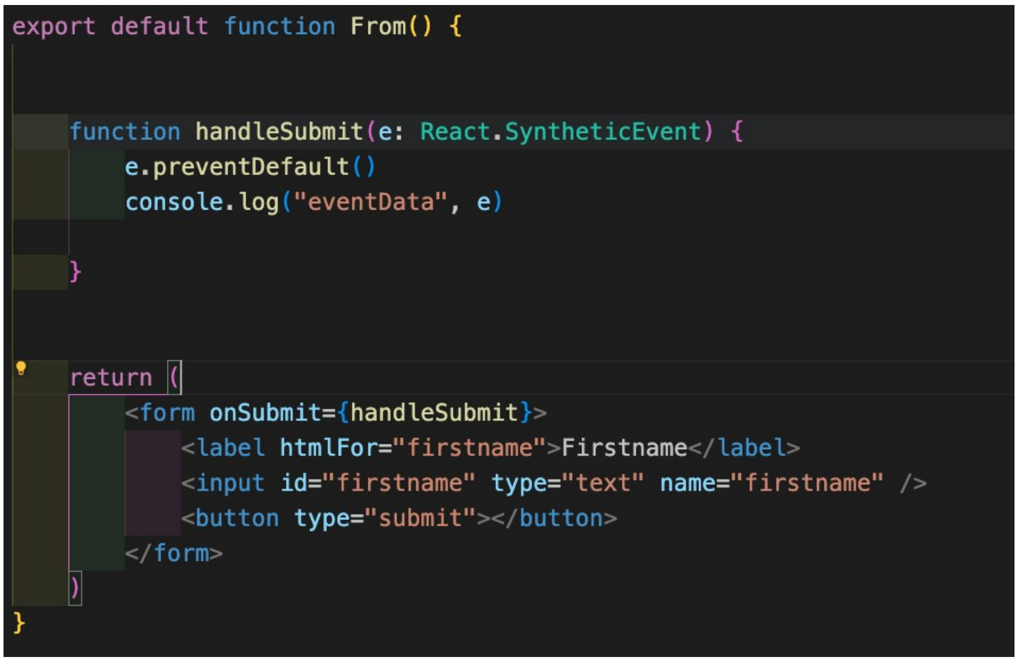


```
import React from "react";

export default function Button() {
  const handleClick = (
    event: React.MouseEvent<HTMLButtonElement, MouseEvent>
  ) => console.log("Clicked!");

  return <button onClick={}>Click me!</button>;
}
```

PETER WOLF
< Web Developer & Advisor />

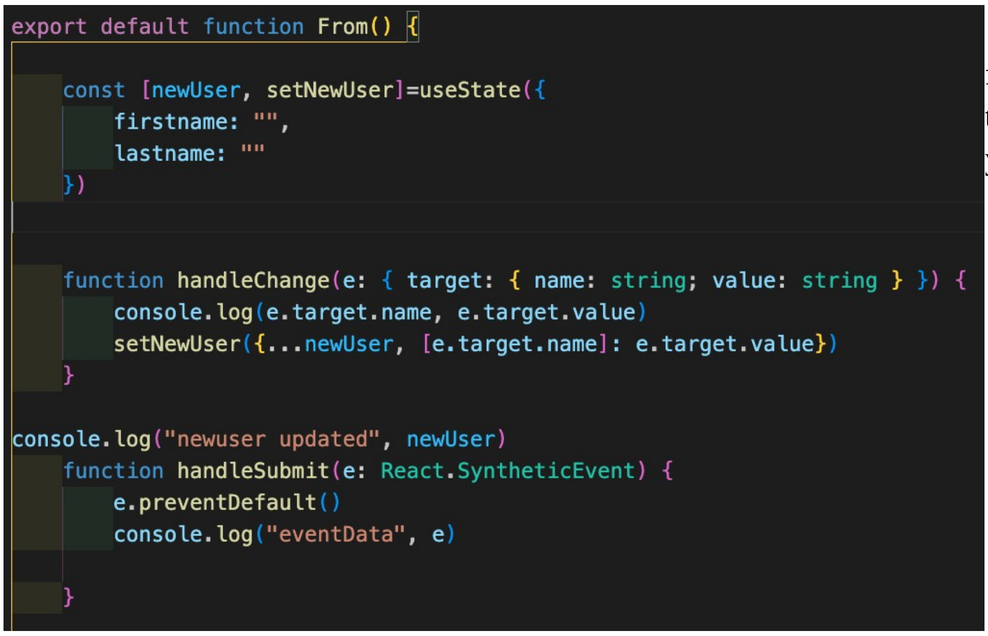
Form Submit TS handling

If you don't quite care about the type of the event, you can just use `React.SyntheticEvent`.



```
export default function From() {


  function handleSubmit(e: React.SyntheticEvent) {
    e.preventDefault()
    console.log("eventData", e)

  }


  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="firstname">Firstname</label>
      <input id="firstname" type="text" name="firstname" />
      <button type="submit"></button>
    </form>
  )
}
```

Form typing for onChange()



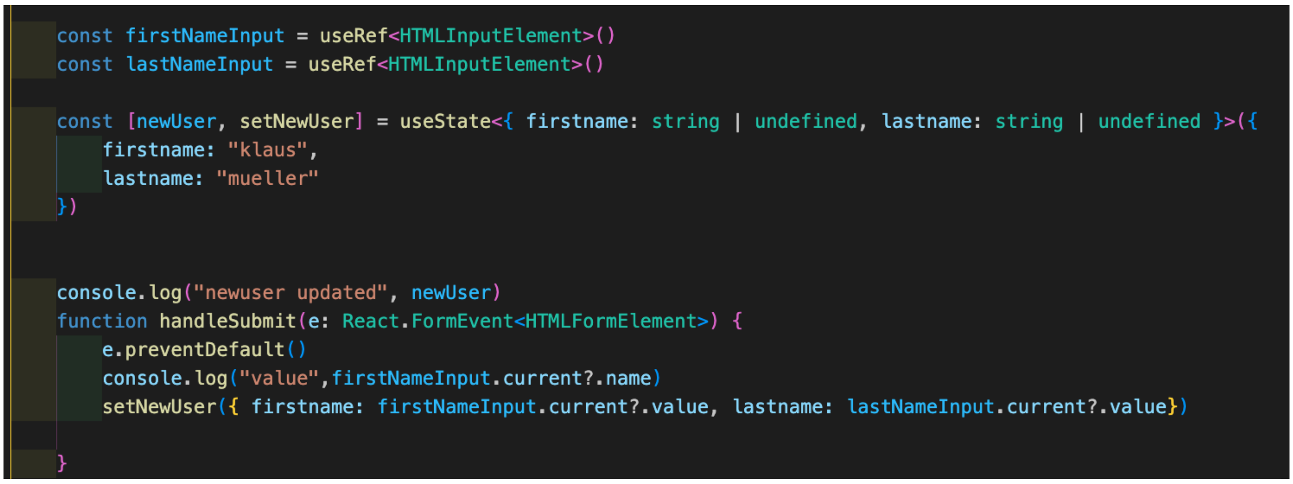
```
export default function From() {

  const [newUser, setNewUser]=useState({
    firstname: "",
    lastname: ""
  })


  function handleChange(e: { target: { name: string; value: string } }) {
    console.log(e.target.name, e.target.value)
    setNewUser({...newUser, [e.target.name]: e.target.value})
  }

console.log("newuser updated", newUser)
  function handleSubmit(e: React.SyntheticEvent) {
    e.preventDefault()
    console.log("eventData", e)

  }
```

The handleChange function event has to type the „target“ values you will work with.

PETER WOLF
< Web Developer & Advisor />

Typing useRef on Form

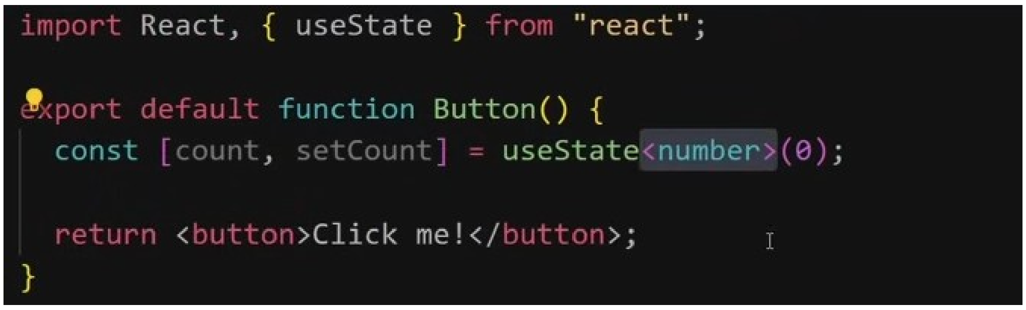
With useRef you have to type the useState and the onSubmit like this:



```
const firstNameInput = useRef<HTMLInputElement>()
const lastNameInput = useRef<HTMLInputElement>()

const [newUser, setNewUser] = useState<{ firstname: string | undefined, lastname: string | undefined }>({
    firstname: "klaus",
    lastname: "mueller"
})


console.log("newuser updated", newUser)
function handleSubmit(e: React.FormEvent<HTMLFormElement>) {
    e.preventDefault()
    console.log("value",firstNameInput.current?.name)
    setNewUser({ firstname: firstNameInput.current?.value, lastname: lastNameInput.current?.value})

}
```

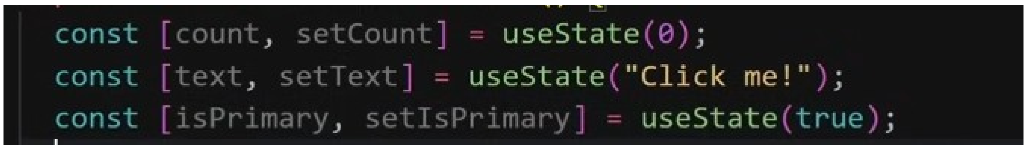
Typing USE STATE HOOK

It is rarely used for primitive types (string, number, boolean) – But you can type like this:



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

export default function Button() {
  const [count, setCount] = useState<number>(0);

  return <button>Click me!</button>;
}
```

The typing of the useState()- value is not neccesary, because TS can infer it from there automatically.

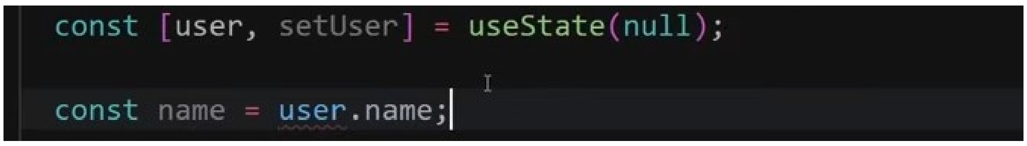


```
  const [count, setCount] = useState(0);
  const [text, setText] = useState("Click me!");
  const [isPrimary, setIsPrimary] = useState(true);
```

You will less see this in real life code.

For Objects it is different (non-primitives)

When you declare a user object with useState, you will often see this:

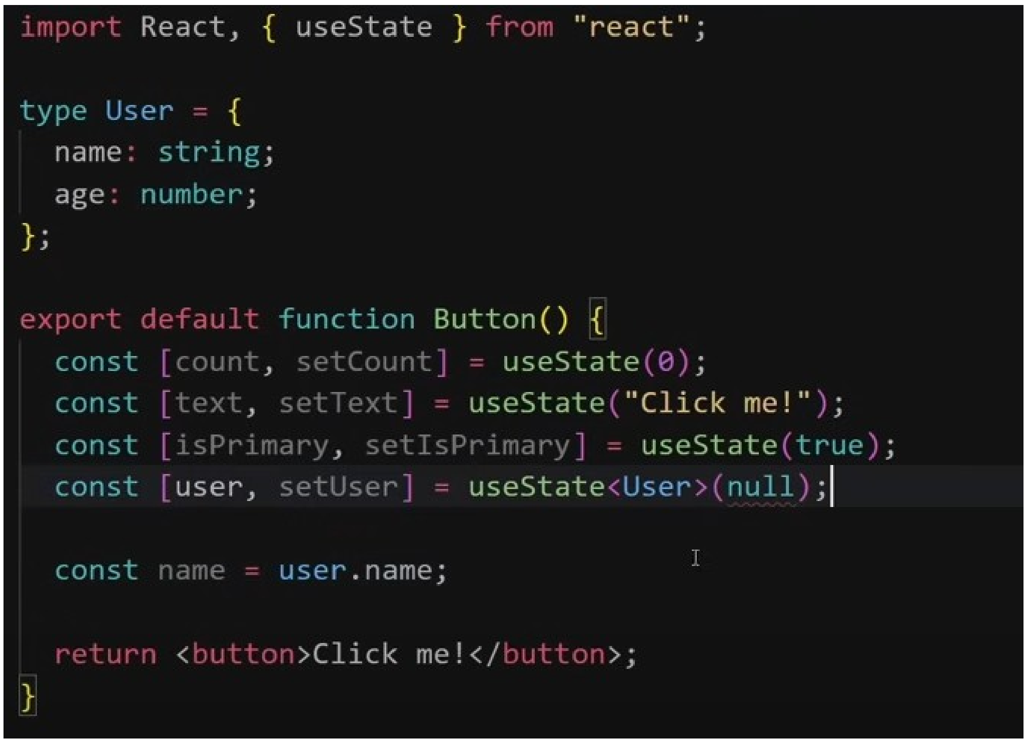


```
const [user, setUser] = useState(null);

const name = user.name;
```

the user is declared initial as null – which results to an error if you call the .name. This is because in null there is no .name or else.

Type UserType option

To make it work, we have to type User with it´s properties.

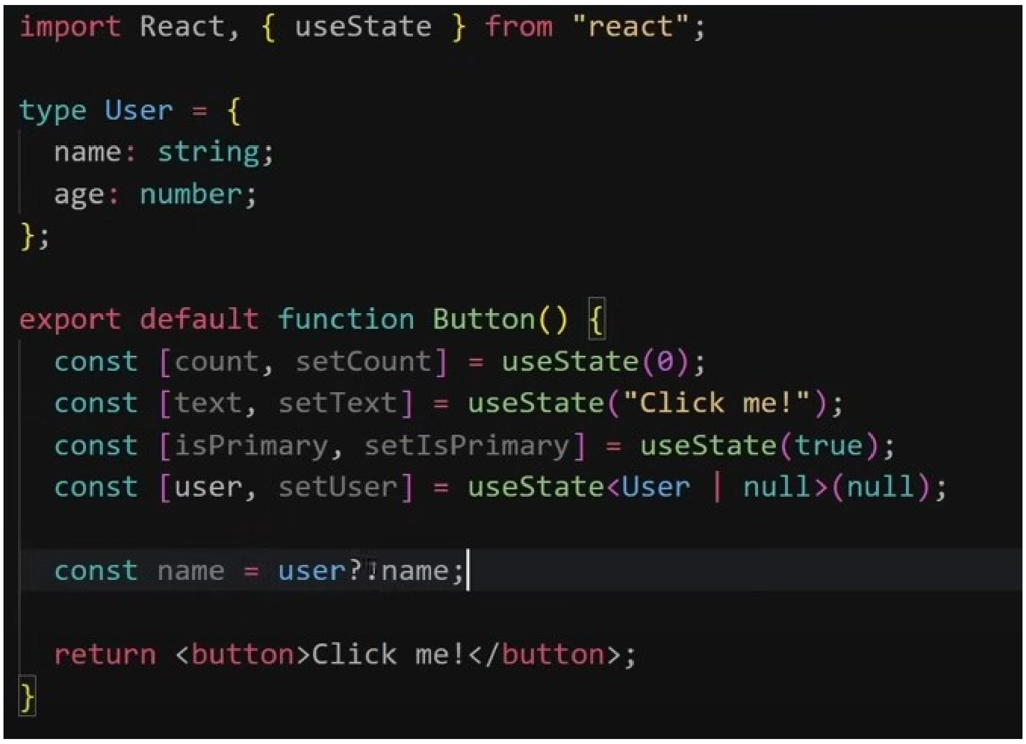


```
import React, { useState } from "react";

type User = {
  name: string;
  age: number;
};

export default function Button() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("Click me!");
  const [isPrimary, setIsPrimary] = useState(true);
  const [user, setUser] = useState<User>(null);

  const name = user.name;

  return <button>Click me!</button>;
}
```

Now the user object is declared and you can call .name.

But there is now an other issue with „null“ Init-value in the useState.
The null is used because we have to fetch the user first, and so Initially the user has „null“

To solve this error you can use



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

type User = {
  name: string;
  age: number;
};

export default function Button() {
  const [count, setCount] = useState(0);
  const [text, setText] = useState("Click me!");
  const [isPrimary, setIsPrimary] = useState(true);
  const [user, setUser] = useState<User | null>(null);

  const name = user?.name;

  return <button>Click me!</button>;
}
```

<u>There are modifications in the code above.</u>

1. useState has either User OR the null
2. If null is the case, the user.name throws an error – with optional chaining „?“ you can solve the error.

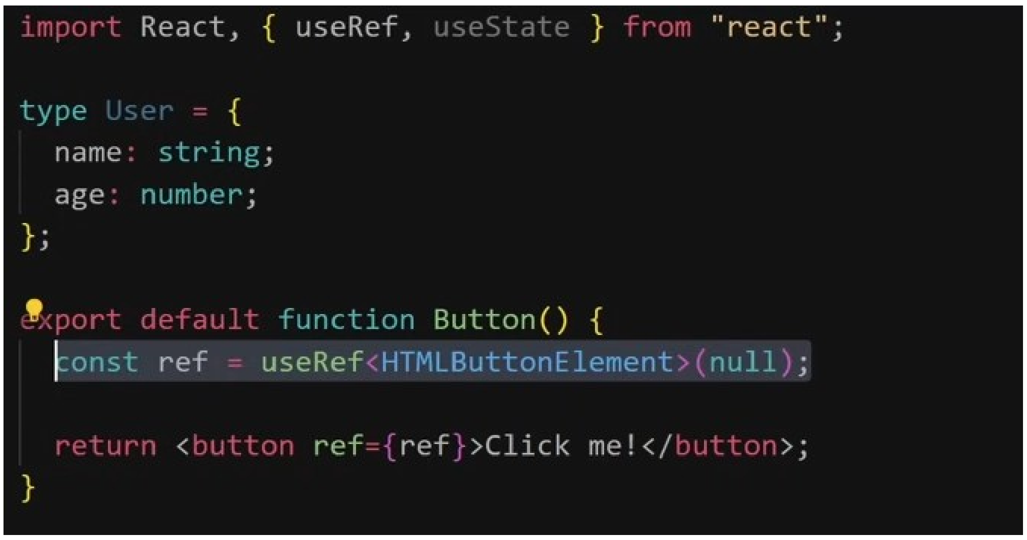
Typing UseRef Hook

With ref you can get a reference to an element - <button> in our example.

We now have to type for useRef what type it will be eventually passed.

You can use some helper types from react.

1. the general „Element“ type
2. Or the „HTMLElement“
3. Or super specific „HTMLButtonElement“

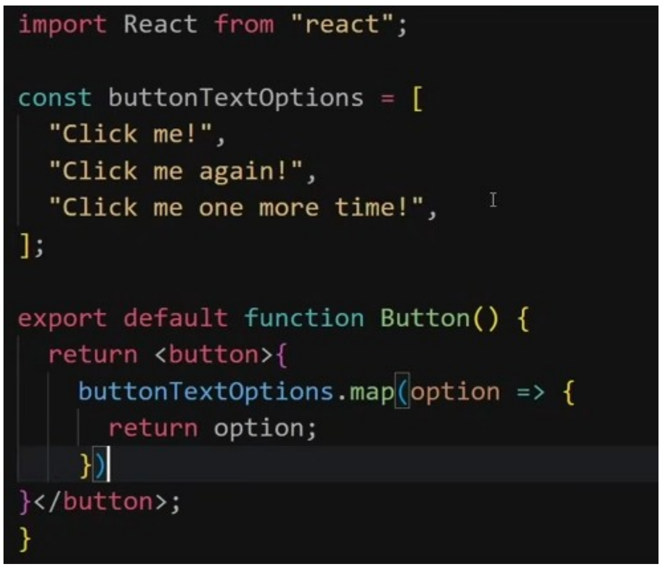


```
import React, { useRef, useState } from "react";

type User = {
  name: string;
  age: number;
};

export default function Button() {
  const ref = useRef<HTMLButtonElement>(null);

  return <button ref={ref}>Click me!</button>;
}
```

TS Tricks to improve your code

as Const = a typeScript feature

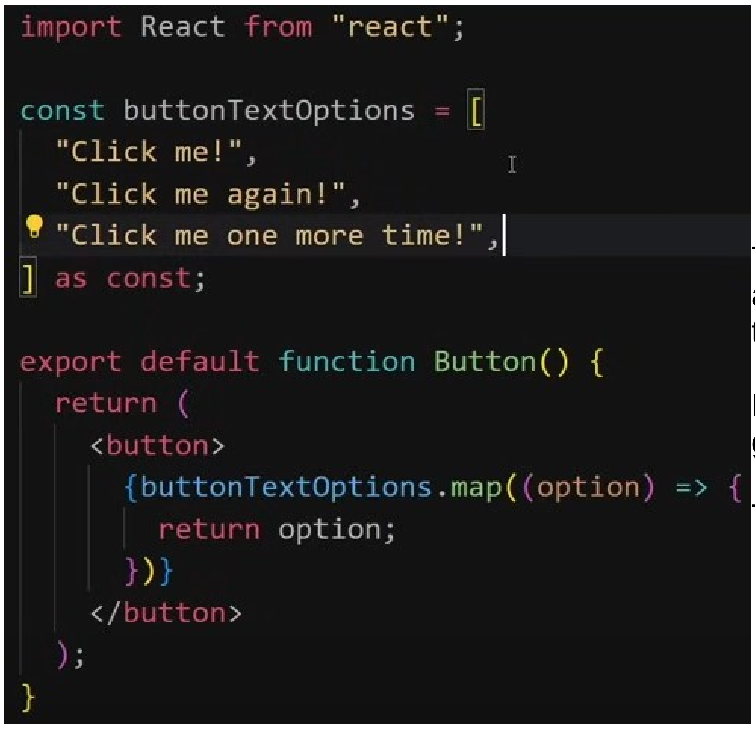
We have this example:



```
import React from "react";

const buttonTextOptions = [
  "Click me!",
  "Click me again!",
  "Click me one more time!",
];

export default function Button() {
  return <button>{
    buttonTextOptions.map(option => {
      return option;
    })
}</button>;
}
```

We have declared and array and we want to provide the strings to the <Button>

If you hover over the „option“ in the map, you see that the option is a string.
The example is not wrong – but we can more specific by adding „as const“-TS option.



```
import React from "react";

const buttonTextOptions = [
  "Click me!",
  "Click me again!",
  "Click me one more time!",
] as const;

export default function Button() {
  return (
    <button>
      {buttonTextOptions.map((option) => {
        return option;
      })}
    </button>
  );
}
```

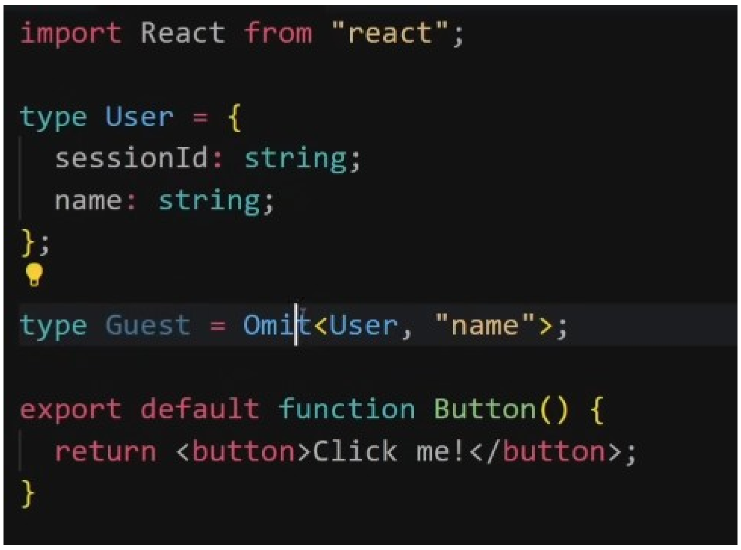
The „as const“ transforms the array in a readonly array. It will also specify the values of the array concrete.

Now when I hover over the option, I get the actual values.

This is a pure TS feature.

Omit utility

We have this example with User TypeScript.



```
import React from "react";

type User = {
  sessionId: string;
  name: string;
};

type Guest = Omit<User, "name">;

export default function Button() {
  return <button>Click me!</button>;
}
```

The User Type defines the properties for user. But for Guest Type we don´t have necessarly the name – but the sessionId.

So we can OMIT the UserType values.

Omit<User, „name“> => use the User Type without the „name“ property.

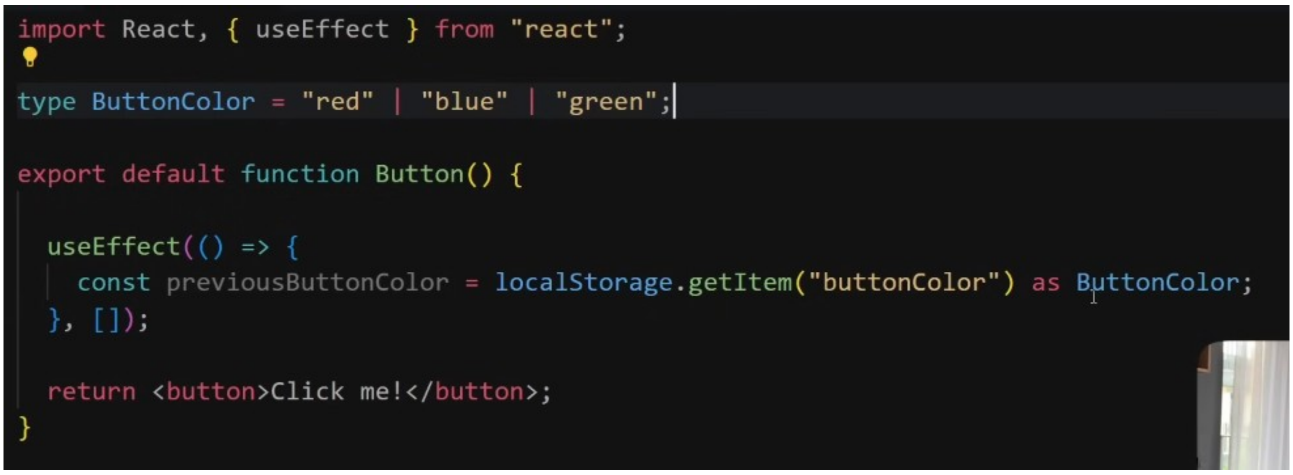
As Type Assortion

Let us say, we have stored buttonColor in the localStorage. With loading of the app, the useEffect checks if there is a „buttonColor“ already stored.

We have defined the Button Type.

Without the assortion, previousButtonColor is only string | null

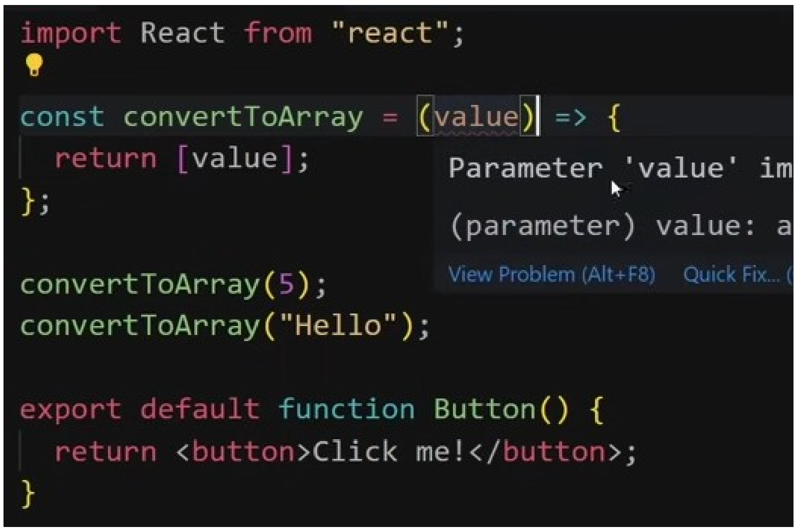
With assertion we know better than TS that the previousButtonColor can only be the typed ones.



```
import React, { useEffect } from "react";

type ButtonColor = "red" | "blue" | "green";

export default function Button() {

  useEffect(() => {
    const previousButtonColor = localStorage.getItem("buttonColor") as ButtonColor;
  }, []);

  return <button>Click me!</button>;
}
```

Generics – a concept of TS

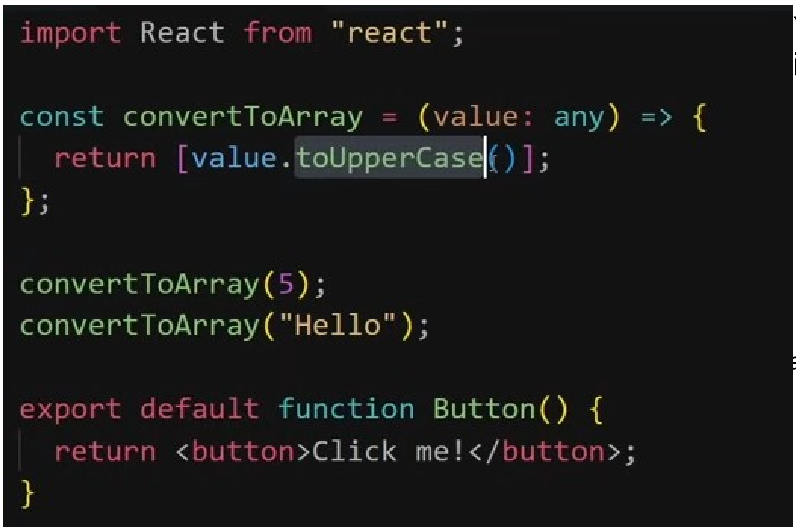
Let us say, we have a function which converts something…



```
import React from "react";

const convertToArray = (value) => {
  return [value];
};

convertToArray(5);
convertToArray("Hello");

export default function Button() {
  return <button>Click me!</button>;
}
```

```
Parameter 'value' im
(parameter) value: a
```

The value is not typed, so you can pass whatever you want.



```
import React from "react";

const convertToArray = (value: any) => {
  return [value.toUpperCase()];
};

convertToArray(5);
convertToArray("Hello");

export default function Button() {
  return <button>Click me!</button>;
}
```

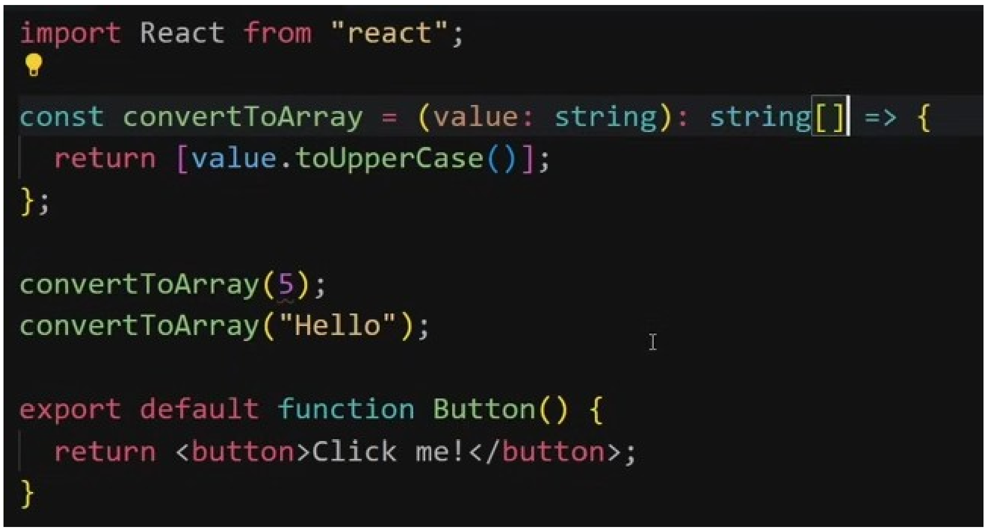
You can type with ANY...But there is some caviats..

advisor

the .toUpperCase()-method is only for strings available. You cannot use it on numbers. But the code doesn´t get an error – because any can be any value.

This function will break at some point.

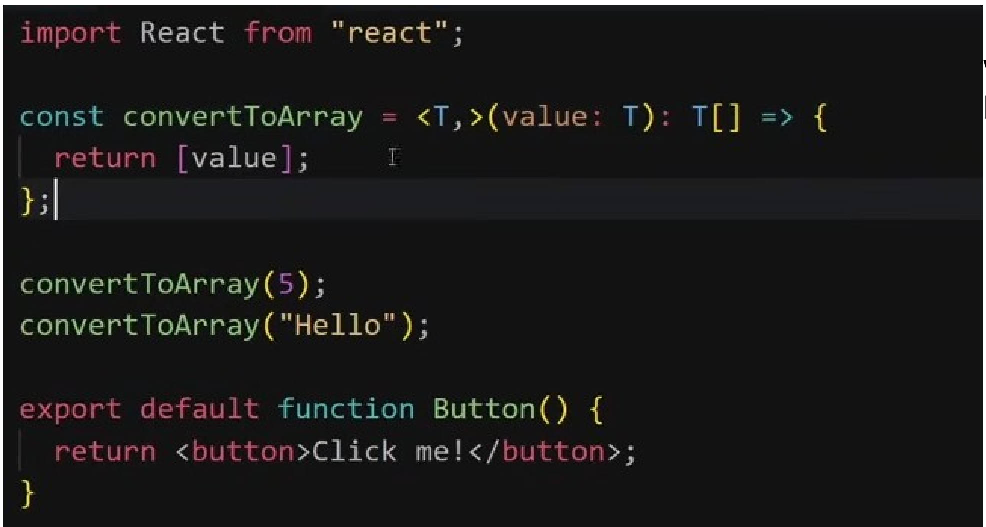
We can solve the issue by specifing the value and the return value.



```
import React from "react";

const convertToArray = (value: string): string[] => {
  return [value.toUpperCase()];
};

convertToArray(5);
convertToArray("Hello");

export default function Button() {
  return <button>Click me!</button>;
}
```

But then there is an error at the first convertToArray call with a number.

We want to pass whatever type we want to provide a new array with the passed values.

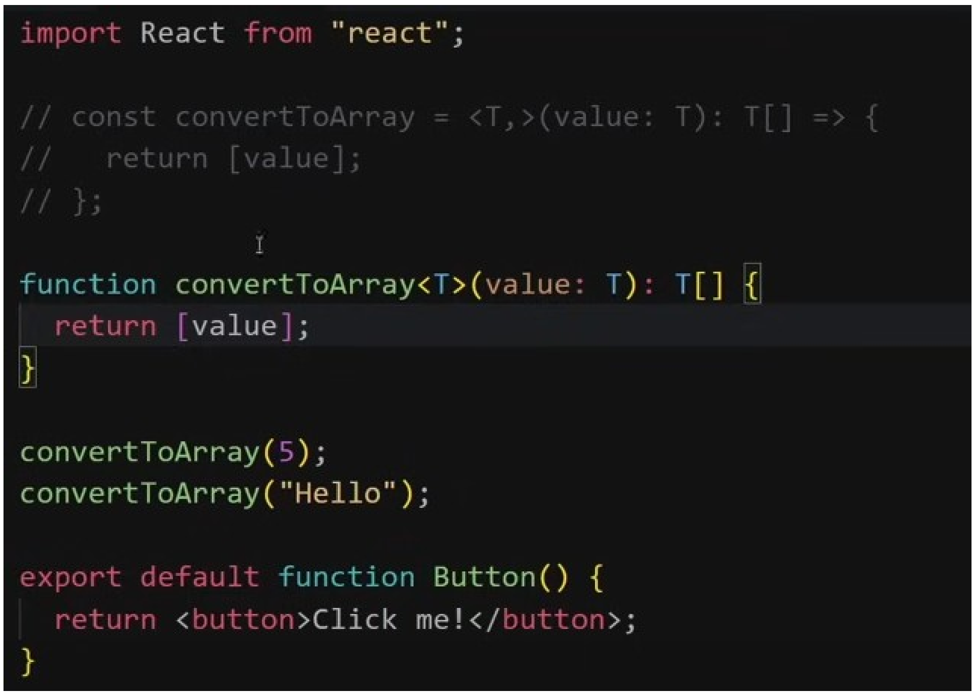


```
import React from "react";

const convertToArray = <T,>(value: T): T[] => {
  return [value];
};

convertToArray(5);
convertToArray("Hello");

export default function Button() {
  return <button>Click me!</button>;
}
```

We can make it happen like this...

With this code we provide a paring from the passed Value type to the return value type. With an array function we have to declare the <T> (naming convetion T is used) before. But writing in an array the <T> TS thinks that the passed value is an element. So we need to write it <T,>

alternativly we will use a function…

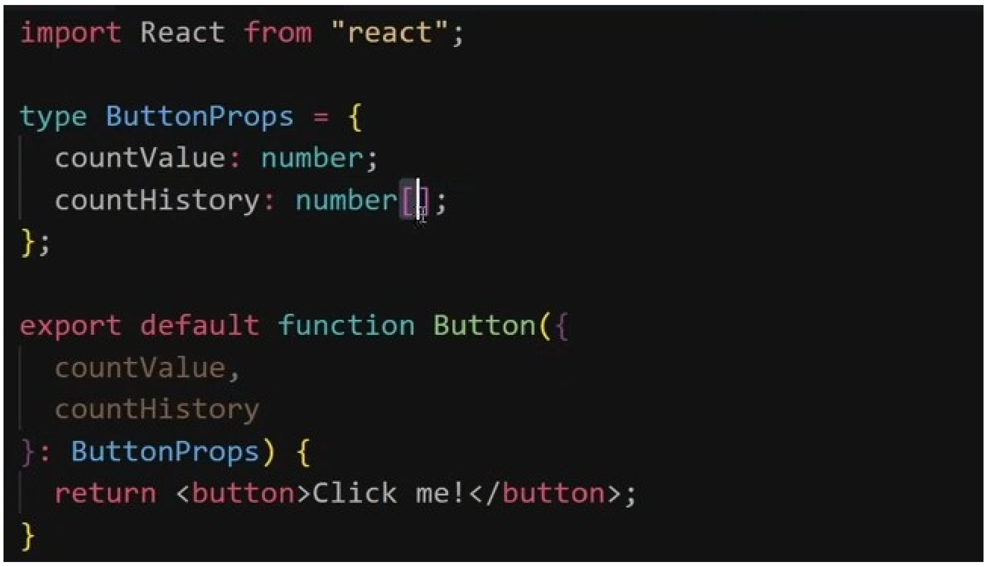


```
import React from "react";

// const convertToArray = <T,>(value: T): T[] => {
//   return [value];
// };

function convertToArray<T>(value: T): T[] {
  return [value];
}

convertToArray(5);
convertToArray("Hello");

export default function Button() {
  return <button>Click me!</button>;
}
```

Now the function can accept all type of values.

How can be a React Component scenario use Generics

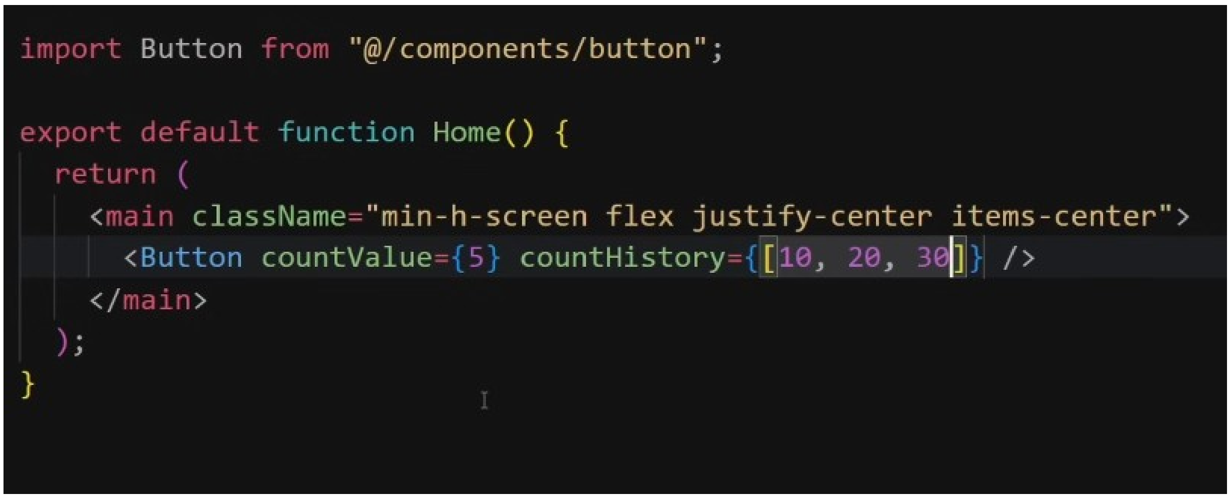
Example of Generics in React Component



```
import React from "react";

type ButtonProps = {
  countValue: number;
  countHistory: number[];
};

export default function Button({
  countValue,
  countHistory
}: ButtonProps) {
  return <button>Click me!</button>;
}
```

We have a <Button> which gets props of a countValue and countHistory.

We declare the type with hard coded value type number! Now we pass the props to <Button> from the parent – and the example below works as expected.



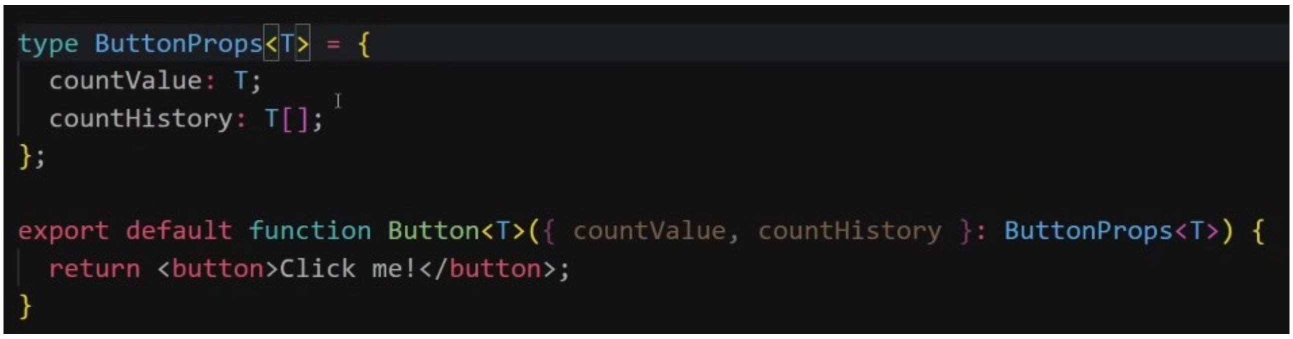
```
import Button from "@/components/button";

export default function Home() {
  return (
    <main className="min-h-screen flex justify-center items-center">
      <Button countValue={5} countHistory={[10, 20, 30]} />
    </main>
  );
}
```

But what if we want to allow other types – like strings or so?

If we change the number5 to string“5“ we get an error

This is because we have an relationship between the parameters.

Make your code flexible for all parameter types



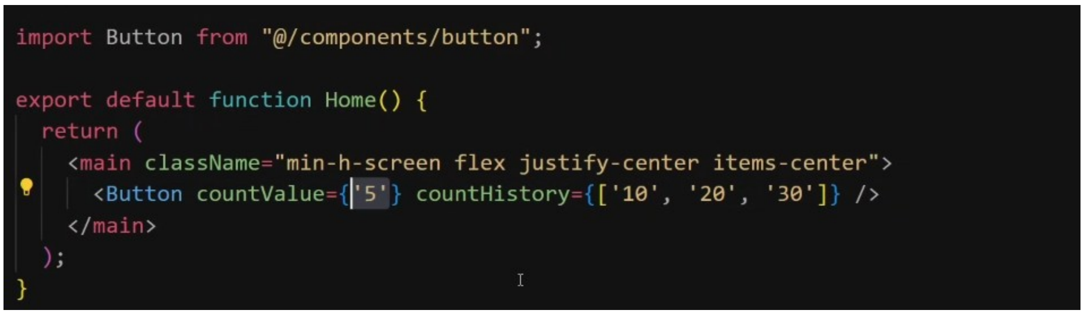
```
type ButtonProps<T> = {
  countValue: T;
  countHistory: T[];
};

export default function Button<T>({ countValue, countHistory }: ButtonProps<T>) {
  return <button>Click me!</button>;
}
```

Now we have restructured the code that the ButtonProps can work with the Generics.

1. at Button we have to pass <T> first – the button now accepts the Type Parameter<T>
2. ButtonProps has to accept Type Parameter as well
3. we need to pass the Type Paramter to the ButtonTypes

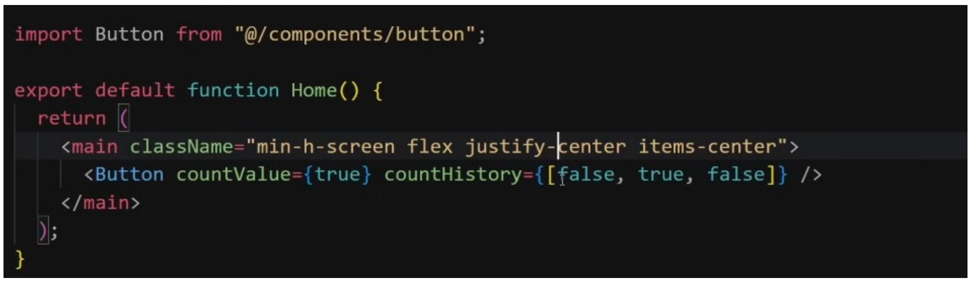
With this relational concept we can now use the countValue and countHistory flexible with any type of value.

Strings



```
import Button from "@/components/button";

export default function Home() {
  return (
    <main className="min-h-screen flex justify-center items-center">
      <Button countValue={'5'} countHistory={['10', '20', '30']} />
    </main>
  );
}
```

Booleans:

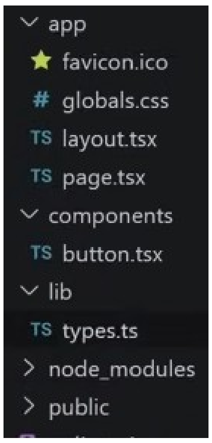


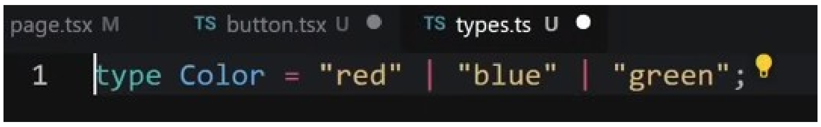
```
import Button from "@/components/button";

export default function Home() {
  return (
    <main className="min-h-screen flex justify-center items-center">
      <Button countValue={true} countHistory={[false, true, false]} />
    </main>
  );
}
```

This concept is the most complicated one in React. So If you are struggling with this, no worries…

File Architecture for reusable Types

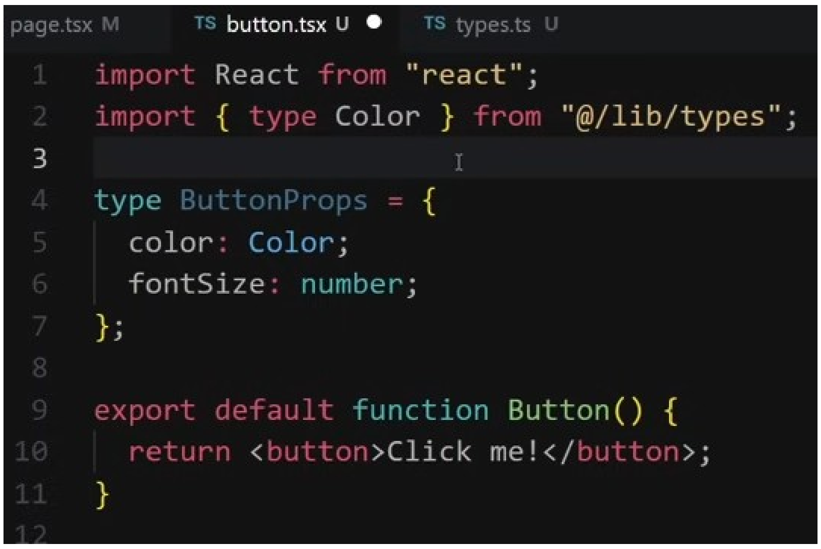
If you want to declare types for a global usage in your application, you need to add to your /lib/ folder for example the types.ts file





```
type Color = "red" | "blue" | "green";
```

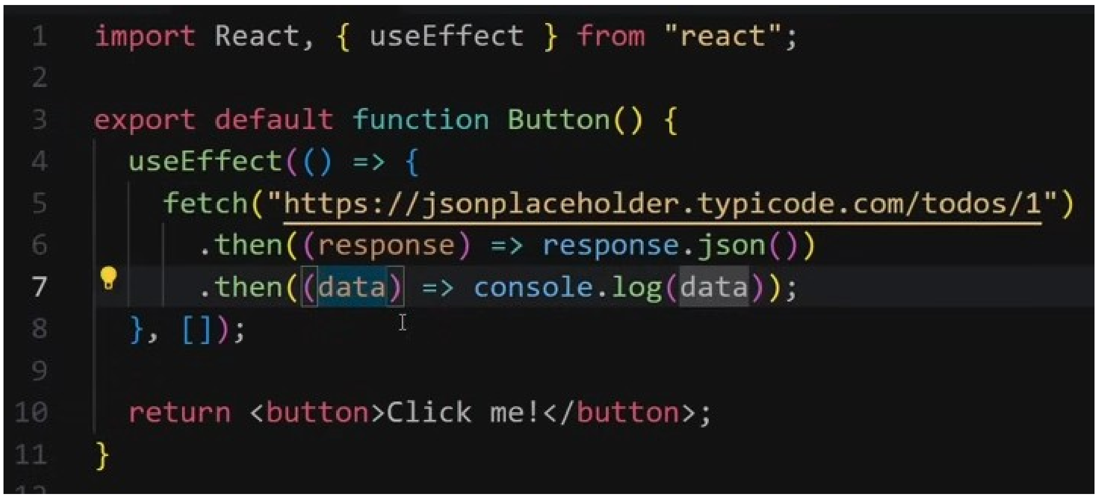
now you can import the Color in every component – with a specific tag like „type“ you declare the import for a type only. Otherwise it looks pretty much like a regular component.



```tsx
import React from "react";
import { type Color } from "@/lib/types";

type ButtonProps = {
  color: Color;
  fontSize: number;
};

export default function Button() {
  return <button>Click me!</button>;
}
```

The Unkown -type (OPTIONAL)

Our example fetches some data from a url.

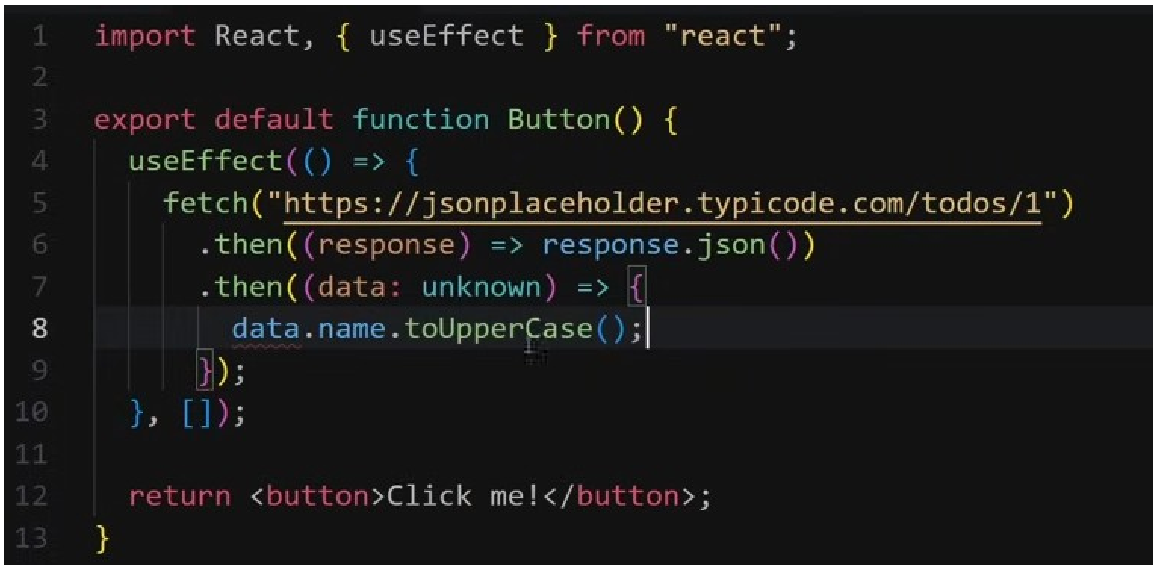


```tsx
import React, { useEffect } from "react";

export default function Button() {
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos/1")
      .then((response) => response.json())
      .then((data) => console.log(data));
  }, []);

  return <button>Click me!</button>;
}
```

By default, the data will be typed as ANY. We want to have as less any in our codebase. It can cause unexpected errors.

With the type of unknown we use an appropriate type.

Unknown means => „we don´t know anything about it“



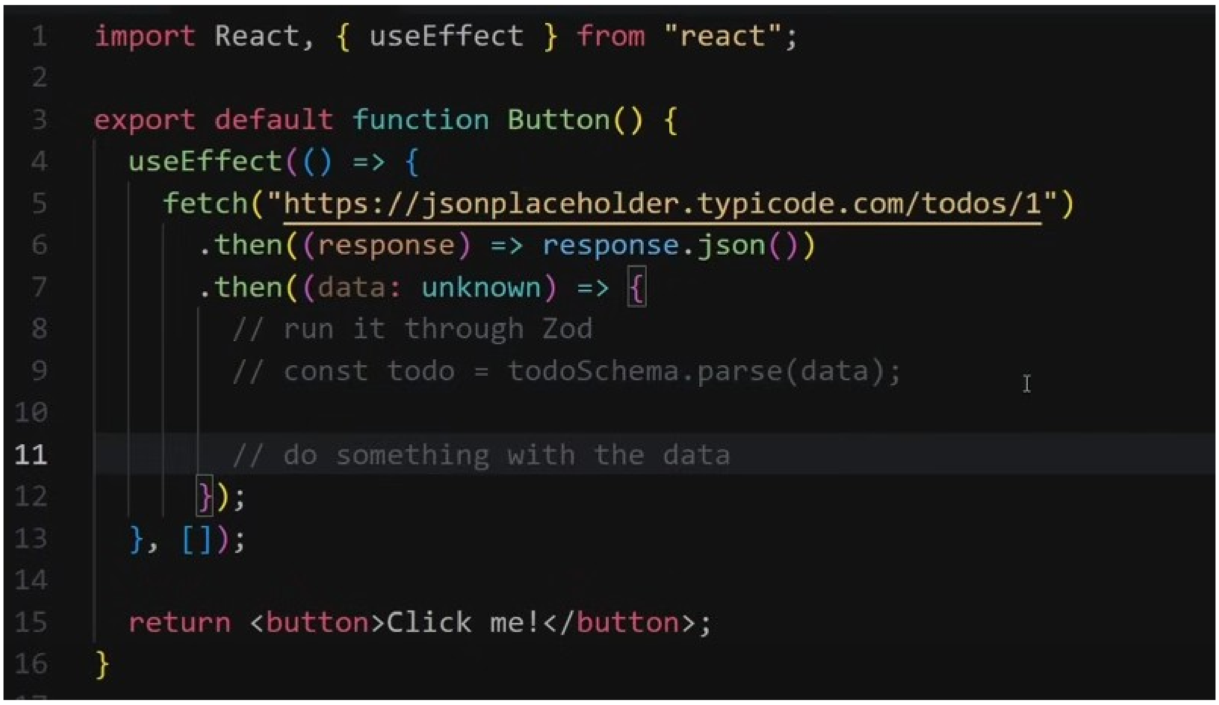
```
import React, { useEffect } from "react";

export default function Button() {
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos/1")
      .then((response) => response.json())
      .then((data: unknown) => {
        data.name.toUpperCase();
      });
  }, []);

  return <button>Click me!</button>;
}
```

Before we can use the data.name.toUpperCase() on an unknown type, we need to varify what we get back.

SCHEMAS can help with that.

ZOD – TS Schema validators

https://www.totaltypescript.com/tutorials/zod



```
import React, { useEffect } from "react";

export default function Button() {
  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos/1")
      .then((response) => response.json())
      .then((data: unknown) => {
        // run it through Zod
        // const todo = todoSchema.parse(data);

        // do something with the data
      });
  }, []);

  return <button>Click me!</button>;
}
```

This ZOD will be too much…
mybe I can create an example.