Tips

This lesson details six tips that will help you use TypeScript alongside React to build your apps.

WE'LL COVER THE FOLLOWING

Hover to discover the type of native browser elements and functions

TypeScript types do exist for native browser elements like `span`, `div`, and `input`. Types already exist for native browser functions like `fetch` as well. But what are the types of these elements and functions? Well, if we are using Visual Studio Code, then we can hover over where the item is referenced, and a tooltip will show us the type information!

```
<React.Fragment>
  <input
    ref={input}
    type="search"
    (property) JSX.IntrinsicElements.div: React.DetailedHTMLProps<Rea
    ct.HTMLAttributes<HTMLDivElement>, HTMLDivElement>
  <div>
    {items.map(item => (
      <div
        key={item}
        style={{ fontWeight: item === search ? "bold" : undefined }}
      >
        {item}
      </div>
    ))}
  </div>
</React.Fragment>
```

Use strict mode

We'll get the most benefit out of TypeScript if strict mode is on. It is particularly useful in helping us avoid the `any` type and `null` reference problems.

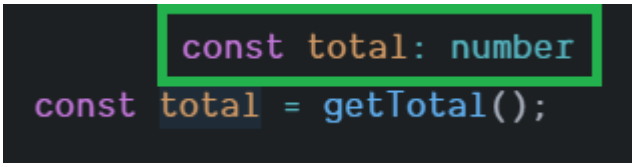
Strict mode is turned on in the following `tsconfig.json` setting:

```
{
  "compilerOptions": {
    "strict": true,
    ...
  },
  ...
}
```

We might think going straight to strict mode would be challenging when migrating an existing codebase from JavaScript to TypeScript. However, the `allowJs` setting allows a mixed JavaScript and TypeScript codebase, so the whole code doesn't need to be migrated before it compiles.

Let TypeScript infer types to avoid code bloat

A downside of TypeScript is the extra code we write for the types. Writing code takes time, and a lot of type annotations can make the code noisy and harder to read. However, we don't need to define type annotations for everything we declare because Typescript's powerful inference will work the types out for us in many cases. We can hover over declarations to discover the inferred type to make sure it is as required.



Avoid the `any` type

When starting with TypeScript, it is tempting to use the `any` type when we can't work out what the type should be. However, we need to remember that no type checking will take place on code that references the `any`, so the benefits of TypeScript are lost. Don't be tempted, take the time to work out what type you should use, it will pay off in the long run.

Generics are your friend when defining functions and classes that need to work with different types. Using the `unknown` type is preferable to `any` when using third party code or web APIs that don't have TypeScript types.

Use readonly to avoid mutation errors

Bugs, where the root cause is a mutation error, are often costly to find. Immutable data structures prevent these types of errors. Declaring immutable properties is simple with the `readonly` keyword. The `Readonly<T>` utility type is a convenient way to make all the properties in an object type readonly. It is important to remember that array and object properties that have the `readonly` before their property name are still mutable.

We can create immutable arrays by using the `readonly` modifier before the type:

```
function add(array: readonly number[]) {
  ...
}
```

Use unioned string literals rather than `string`

Use unioned string literals rather than `string` where you can

Often a string in a data structure can only be a defined set of values. Categories and statuses are common examples of these. In other statically typed languages, we would use the `string` type to represent these, but with TypeScript, we can use unioned string literals to form a much stronger type:

```
type Status = "Not Started" | "Busy" | "Complete";
let jobStatus: Status = "Not Started";
jobStatus = "Working";  // 💣 Type '"Working"' is not assignable to type
'Status'
```

Wrap up

I'm sure you will find these tips useful in your next React and TypeScript project.

Next, we'll remind ourselves of the benefits of using TypeScript with React and finish the course.