TypeScript for React Developers

Exercise 1

Take a look at exercises/01-template. This was generated using:

```
npx create-react-app 01-template --typescript
```

Note the following:

- The `tsconfig.json` file determines how TypeScript is configured. You always want `"strict": true`
- `.js` files became `.ts` or `.tsx` - must be `.tsx` if they contain JSX code
- In `package.json`, notice the `@types/` packages

Exercise 2 - Prop Types

Take a look at exercises/02-hello-world. Update `Hello.tsx` to look like:

```
import React from 'react';

export default function Hello(props: {name: string; language: string}) {
  return (
    <p>
      Hello {props.name}, welcome to this workshop on using React with {props.language}
    </p>
  );
}
```

In `App.tsx`, render:

```
return <Hello name="Your Name" language="TypeScript" />
```

The basic types in TypeScript are:

- `number`
- `string`
- `boolean`
- `null`
- `undefined`

You can also use literals as types:

- `'hello'`
- `'5'`
- `true`

React has a few very useful types to know:

- `React.ReactNode` - anything that would be valid to use as a child of a React component/return from a render function.

- `React.CSSProperties` - the type for a `style` object
- `React.ComponentType<{hello: string}>` - a component that takes the property `hello` of type `string`.

In addition to these basic types, there are a few more interesting types we will look at properly later:

- `any` - represents anything, and tells TypeScript not to bother type checking
- `unknown` - represents anything, but tells TypeScript not to let you access any properties/methods without first casting it to something else or testing it.
- `void` - the return type for a function that doesn't return anything
- `never` - a value that can't exist, meaning the code is unreachable e.g. `(() => {throw '';}())` is of type `never` because it never has an actual value.

Types can also be combined in objects, classes, interfaces, unions, intersections etc. We'll talk about most of these as we go through the remaining exercises.

Exercise 3 - State, Unions & Generics

Take a look at exercises/03-counter. If you run `yarn start` you should see a simple counter with a + and - button and the current value.

Try un-commenting the "Toggle Mode" button and the `import {toRoman, toArabic} from 'roman-numerals';` statement. You should see a type error on the line that says `setCount(toRoman(count));`. This is because `toRoman` returns a `string`, and TypeScript assumed our `counter` would always be a number, since we set it to a number when we started. To fix this, you'll need to replace:

```
const [count, setCount] = React.useState(0);
```

with:

```
const [count, setCount] = React.useState<string | number>(0);
```

Now TypeScript knows that `count` could be a `string`, or a `number`. Note that we use a single `|`, unlike the double `||` typically used in JavaScript/TypeScript to mean "or" at runtime.

At this point though, TypeScript complains when we attempt to do `count - 1` and `count + 1`. To fix this, try adding and then using the following helper:

```
function mapNumber(number: string | number, fn: (value: number) => number) {
  if (typeof number === 'string') return toRoman(fn(toArabic(number)));
  return fn(number);
}
```

You should now be able to run the example again, and toggle between counting using arabic numbers and roman numerals.

P.S. note the `@types/roman-numerals` package in package.json.

Another way to fix this example, which might work better in real world code, would be to separately store two different states:

1. The current value as a number.
2. A boolean indicating whether the number should be displayed in arabic notation.

You could also try adapting the example to work this way.

Exercise 3b - generics

One of the interesting thing about the `mapNumber` function is that we loose some information in the types. You might like to try out the following code:

```
const value: number = mapNumber(42, c => c + 1);
const romanValue: string = mapNumber('XLII', c => c + 1);

// ...
<p>{value} = {romanValue}</p>
```

TypeScript will complain that you cannot assign `mapNumber(42, c => c + 1)` to `value: number` because the type of `mapNumber` is `number | string`. It sees the `string` and thinks "what if it's a string, you can't assign a string to a number". We know better though, if you pass `mapNumber` a `number` it returns a `number`.

We can solve this with overloading:

```
function mapNumber(number: number, fn: (value: number) => number): number;
function mapNumber(number: string, fn: (value: number) => number): string;
function mapNumber(number: string | number, fn: (value: number) => number): string | number;
function mapNumber(number: string | number, fn: (value: number) => number) {
  if (typeof number === 'string') return toRoman(fn(toArabic(number)));
  return fn(number);
}
```

Here, we have 3 versions of the function:

1. If passed a number, we return a number
2. If passed a string, we return a string
3. If we are passed a value that could be a string or a number, we might return a string, or we might return a number.

> N.B. overloading is dangerous. TypeScript doesn't check that your overloads are actually valid. If you swap the return types of your overloads, TypeScript would be none the wiser. Only use overloading when absolutely necessary.

The other way to represent solve this would be using generics. This particular example doesn't lend itself very well to generics, so we end up with:

```
function mapNumber<T extends string | number>(
  value: T,
  fn: (value: number) => number,
): T extends number ? number : T extends string ? string : (number | string) {
  const n: string | number = value;
  if (typeof n === 'string') return toRoman(fn(toArabic(n))) as any;
  else return fn(n) as any;
}
```

This doesn't show TypeScript at its best, but does allow us to show of just how powerful it is, here we have:

- `<T extends string | number>` - this means there is a generic type `T` that must be asignable to `string | number`.
- `value: T` - the `value` parameter is of that type.
- `T extends number ? number : T extends string ? string : (number | string)` - if `T` is assignable to `number`, this returns `number`, if T is assignable to `string`, this returns `string`, otherwise it is assignable to `string | number`

- `as any` - this means, ignore the type of this expression, i.e. treat it as `any`.

Exercise 4 - Unions part 2

When declaring object types, you can write them inline, as we've seen above, you can also give them an alias, or declare them as an interface:

```
interface A {
  x: number;
}
type B = {x: number};
```

The advantage of the `interface` approach is that the name is associated with the type in error messages. If you use the type alias approach, the name isn't used in type aliases.

In TypeScript, you use unions like `string | number` and `string | null` to say that something is one of two types. This doesn't only work with simple types; it also works with objects, e.g. `{kind: 'a', a: number} | {kind: 'b', b: number}`. In addition to declaring types inline, you can declare and export interfaces and aliases:

```
export interface A {
  // the 'a' here is the type that only has one value, the literal "a"
  kind: 'a';
  a: number;
}

export interface B {
  kind: 'b';
  b: number;
}

type AorB = A | B;
export default AorB;
```

Take a look at exercises/04-wall. It creates two types of Post, `text` and `image`. Add the necessary types and everything should work great.

Exercise 5 - Optional Properties and Parameters

You can make a property or parameter optional with a `?` before the `:`. e.g.

```
interface Foo {
  x?: number;
  y?: number;
}
const a: Foo = {x: 10, y: 10};
const b: Foo = {x: 10};
const c: Foo = {y: 10};

function run(value?: number) {
}
run();
run(42);
```

This is almost equivalent to `| undefined`, but if you had:

```
interface Foo {
  x: number | undefined;
  y: number | undefined;
}
```

you'd have to do:

```
const b: Foo = {x: 10, y: undefined};
const c: Foo = {x: undefined, y: 10};
```

See if you can fix 05-optional's type errors.

Bonus, See if you can use destructuring to clean up how the defaults are specified. e.g.

```
function foo({a = 10, b = 10}: {a?: number; b?: number} = {}) {
  return a + b;
}
assert(foo() === 20);
```

If you do this with parameters, you don't even need to mark them as optional:

```
function increment(value: number, by: number = 1) {
  return value + by;
}
```

Exercise 6 - Generics

When you declare something with a function, class, interface or type alias, you can use a place holder instead of a concrete type for things that vary. For example:

```
function RenderTwice<T>(props: {value: T, render: (value: T) => React.ReactNode}) {
  return (
    <div>
      {render(value)}
      {render(value)}
    </div>
  );
}
```

Here we're saying you can give us a value of type `T` and a function that accepts a value of type `T` and that we will support any such value, but will always use the same value consistently.

Take a look at exercises/06-generics/src/List.tsx and see if you can remove all the `any` types. The `any` types are a proble because they mean TypeScript isn't really checking our work.

Covariance

One of the big differences between TypeScript and Flow is how they handle variance. The following is allowed by TypeScript, but rejected by Flow:

```
function logNumbers(values: Array<number | null>) {
  for (const n of numbers) {
    if (n) console.log(n);
  }
}
```

```
const numbers: Array<number> = [1, 2, 3];
logNumbers(numbers);
```

To see why Flow rejects this, consider code like:

```
function logNumbers(values: Array<number | null>) {
  values.push(null);
}
const numbers: Array<number> = [1, 2, 3];
logNumbers(numbers);
console.log(numbers);
```

My implementation of `logNumbers` may be a little bit nonsensical, but it does still match the original type signature. The issue is that subsequent use of `numbers` will assume that it is of type `Array<number>` but it actually contains a `null` value. TypeScript is intentionally being a bit loose here, because it's so common for a function to take an array and not mutate it's inputs. If we want to absolutely prevent this error though, the best thing we can do is be explicit about the fact that arrays are read only.

```
function logNumbers(values: ReadonlyArray<number | null>) {
  for (const n of numbers) {
    if (n) console.log(n);
  }
}
const numbers: Array<number> = [1, 2, 3];
logNumbers(numbers);
```

This code is much safer, and is accepted by both Flow and TypeScript. It's ok to pass an `Array<number>` to a function that expects a `ReadonlyArray<number | null>` because neither function will violate the expectations of the other.

Exercise 7 - classes

Classes in TypeScript work just like Classes in JavaScript. One difference is that the experimental property assignment syntax:

```
class Foo {
  x = 1;
}
```

is standard in TypeScript. You can give properties explicit types, just like you would a variable declaration. You can also declare properties that get assigned a value in the constructor. When extending `React.Component` in TypeScript you need to specify the `Props` and `State` type like: `React.Component<Props, State>`. If you don't have any Props or State, you can just use the empty `{}` in their place. Don't use `any` as it leads to a refactoring hazard later.

Try adding the `Props` and `State` types needed to the Counter component in exercises/07-class-counter

The other novel thing about TypeScript classes, is that you can add an access modifier `public`, `private`, `protected` and you can add `readonly`. The public/private modifier always preceeds the readonly modifier. e.g.

```
class Foo {
  private readonly x: number;
  constructor(value: number) {
    this.x = value;
  }
  print() {
    console.log(this.x);
  }
}
```

My recommendation is to always use `private` and `readonly` unless you actively plan to access the property from outside the class or modify the property. That way you'll get used to being that little bit more cautious around properties that are public.

You can mark the instance methods as `private readonly`. You can't do this with `state` or `render` as the visibility of these methods must match their visibility in the class you're extending.

> N.B. `private` and `protected` do not do anything at runtime. It's always possible to access a private property by doing `(myClass as any).privateProperty`.

Exercise 8 - Rest Params

If you have rest params, they can be typed as either an array:

```
function joinStrings(...strs: string[]) {
  return strs.join('');
}
```

or they can be a tuple, where the following two functions are approximately equivalent:

```
function add(...args: [number, number]) {
  return args[0] + args[1];
}
function add(a: number, b: number) {
  return a + b;
}
```

You can also use generic types for the Args, providing the generic is constrained to `extends any[]` which means that the TArgs would be assignable to `any[]`

```
function call<TArgs extends any[], TResult>(fn: (...args: TArgs) => TResult, ...args: TArgs): TR
  return fn(...args);
}
```

See if you can add the missing types to make the `debounce` function work in exercises/08-rest-params

Exercise 9 - Enums

It's common in JavaScript to have values used to distinguish between a set of states or objects of differing structures that appear in the same list (e.g. the posts from our earlier example).

You can represent this in TypeScript using literal types and unions:

```
type State = 'state1' | 'state2';
```

Sometimes though, you may want the states to be represented as integers, for performance, and to still give them frienly names in the code. This is where enums are most useful:

```
enum State {
  state1,
  state2,
}
```

Is equivalent to:

```
enum State {
  state1 = 0,
  state2 = 1,
}
```

You can also use string literals if you prefer:

```
enum State {
  state1 = 'state1',
  state2 = 'state2',
}
```

Defining an enum creates the type `State` , a type for each value in the enum (`State.state1` and `State.state2`), and a runtime object that maps the keys onto the values. This also allows you do enumerate the possible keys & values.

In exercises/09-enums, convert our Post types to use a `PostKind` enum. You can try either integers or strings as values.

> N.B. Although using numbers as values does offer a small performance improvement, especially if you're sending a lot of enum values over a network connection, they make debugging significantly more difficult. At runtime you'll just see the numbers, and it will be up to you to look up what that number means in that context.

Exercise 10 - Type Guards & never

When we render posts, we have to figure out which version of the union we are in, this is called a type guard. TypeScript can infer a lot of kinds of type guards automatically. e.g.

```
function maybeParse(value: number | string): number {
  if (typeof value === 'number') {
    // TypeScript knows that the value is of type `number` here
    return value;
  }
  // TypeScript knows the value is of type `string` here
  return parseInt(value, 10);
}
```

Sometimes we might have more complicated logic to figure out what type something is. In these cases, TypeScript can't always figure out the type for us. In these situations, there are two options.

You can do an unsafe cast:

```
// TypeScript doesn't know `isNumber` acts as a type guard
// because we just list it as returning `boolean`
function isNumber(value: unknown): boolean {
  return typeof value === 'number';
}
function maybeParse(value: number | string): number {
  if (isNumber(value)) {
    // TypeScript doesn't know `value` is a `number`
    return value as number;
  }
  // TypeScript doesn't know `value` is a `string`
  return parseInt(value as string, 10);
}
```

Alternatively, you can mark your function as a type guard

```
function isNumber(value: unknown): value is number {
  return typeof value === 'number';
}
function maybeParse(value: number | string): number {
  if (isNumber(value)) {
    // TypeScript knows that the value is of type `number` here
    return value;
  }
  // TypeScript knows the value is of type `string` here
  return parseInt(value, 10);
}
```

In this case, it's still unsafe because TypeScript doesn't actually check that your type guard is valid. It only checks that the function returns a boolean, not what value that boolean has.

In exercises/10-type-guards, try adding custom type guards for `isTextPost` and `isImagePost` .

One issue with how we've been writing our code for rendering a post is how we handle unexpected values. If you comment out the handling for ImagePosts, you won't see any error, they just won't render.

If you've handled both cases correctly, and you check the type of `post` at the end of the function, you'll see it is of type `never` (you can do this by hovering over the name in the appropriate part of the code). What `never` means here is that there is no possible post value that would reach this part of the code (assuming all the types are valid).

We could throw a runtime error here if the code becomes reachable, but we want to catch it using TypeScript. This is where the `assert-never` package comes in. You can pass it the `post` value, and it will throw a runtime exception explaining that the code should have been unreachable. The clever thing though, is the only value it accepts is of type `never` , so it also fails type checking if the code is unreachable.

Try using `assert-never` so that when you comment out one of the handlers, you get a clear error message.

> N.B. It's ok to return `never` from a function, regardless of which return type is expected, so you can return the result of `assertNever` to prevent any warnings/errors about missing returns in functions.

Exercise 11 - Intersection Types

Similar to how `A | B` is a type that is either `A` or `B` , `A & B` is a type that is both `A` and `B` at the same time. This doesn't make sense for things like `string` and `number` (a value cannot be both a string and a number), but for objects, it effectively merges the properties of both objects.

Take a look at 11-intersection-types. Find a way to define the type of the properties for the `Hello` component in terms of the properties of the `Welcome` component.

Exercise 12 - Runtime Type Validation

When you call external APIs, you don't generally have any guarantee about the type that's returned. You can always read the docs and write type definitions to match what you expect, but sometimes you may want to guard against unexpected changes in the API. If you validate types right away, issues will be easier to debug when the API changes.

In a new terminal, you can open and run the "backend" server. This will start an API server on http://localhost:8000 with a few REST endpoints and a GraphQL API.

In Exercise 12, I've left src/types.ts for you to fill in. You could just define types for `Account` and `Contact`. These would look something like:

```
export interface Account {
  id: number
  type: AccountType
  value: string
}

export interface Contact {
  id: number
  name: string
}
```

This won't catch errors if you make mistakes though. What if the `Account.value` was sometimes a `number` or the `Contact.id` was actually a `string`? You'd only find out if it broke code at runtime, and that could be hard to debug.

I've added two dependencies: `funtypes` and `funtypes-schemas`. These allow you to "parse" unknown inputs.

We can define a schema using code like:

```
import * as t from 'funtypes/readonly';
import * as s from 'funtypes-schemas';

const MyObjectSchema = t.Object({
  integerId: s.Integer({min: 1}),
  numericValue: t.Number,
  stringValue: t.String,
})
```

This schema can be used to parse an object with the desired shape via `MyObjectSchema.parse(someUnknownValue)`. TypeScript is able to correctly infer the type of the returned object.

To get the type that's returned by parsing using this schema, you can use `t.Static` like:

```
type MyObject = t.Static<typeof MyObjectSchema>
```

See if you can define schemas in `src/types.ts` and generate static types from these. Then see if you can update `src/api.ts` to parse the `unknown` values returned from the API into the appropriate types.

If (like in this example) both the frontend and backend are written in TypeScript, you might prefer to create a separate shared package for the types, and not do runtime validation on the client side, as it does have some performance impact.

P.S. You might find `src/useApiResponse.ts` interesting to review, it uses generic types and union types to create a small state machine to track the process of loading data from an API.

Exercise 13 - GraphQL types

GraphQL is a popular protocol for data fetching in GraphQL that can replace REST in many places. In exercise 13, The old `api.ts` is replaced with an `api` directory. The `src/api/operations.graphql` file defines our GraphQL operations. There's a script in `package.json` to generate types from these requests. You can run `yarn generate-graphql-types` to generate `src/api/operations.ts` which contains all the types.

Update `src/AccountDisplay.tsx` and `src/ContactDisplay.ts` x to reference the types generated by GraphQL.

Update `getContacts` in `src/ContactsContainer.tsx` to pass the `GetContacts` operation to the GraphQL request function in `src/api/request.ts` .

Here, we're not doing any runtime validation, but the GraphQL schema makes it easier to keep everything in sync.

Conditional Types

One of the most powerful features that TypeScript has (and flow doesn't) is conditional types. This lets you take a type (usually from a generic) and then make another type that depends on it. For example:

```
import {toRoman, toArabic} from 'roman-numerals';

function toggle<T extends string | number>(value: T): T extends string ? number : string {
  if (typeof value === 'string') return toArabic(value);
  else return toRoman(value);
}
```

You can also infer parts of the value. For example, the following makes `X` an alias for the `string` type.

```
type ReturnTypeOf<T> = T extends (...args: any[]) => infer R ? R : never;

function foo(): string {
  return 'Hello World';
}
type X = ReturnTypeOf<typeof foo>;
```

Escape Hatches

It's often not possible to make everything perfectly type safe, so you may need to use one of the escape hatches.

You can tell TypeScript to expect (and ignore) an error on the next line using a special comment:

```
// @ts-expect-error
const x: string = 42
```

You can also use `// @ts-ignore-error` but I always recommend you prefer `// @ts-expect-error` as that way TypeScript will force you to remove the escape hatch when it's no longer needed. Since this ignores any errors on that line, it's good practice to keep as little code as possible on the line where you've expected an error.

You can also perform an unsafe cast using `as` :

```
function unsafeAsNotNull<T>(value: T | null): T {
  return value as T
}
```

The downside here is that TypeScript will not automatically warn you when the `as` is not needed, but the upside is that it's more specific about exactly what unsafe thing you are doing, while `// @ts-expect-error` will ignore absolutely any errors on the following line.

Index Types and "keyof"

Sometimes you may want to use generics to manipulate objects, using things like `Object.keys`, `Object.fromEntries` and `Object.entries`. Although you will probably end up needing escape hatches within functions, you can still produce well defined types outside of the functions. For example, `Object.keys(obj)` always returns `string[]` but in most cases it would be fairly safe to say that it returns only the keys of `obj`. So you could write:

```
function objectKeys<T>(obj: T): (keyof T)[] {
  const result = Object.keys(obj)
  // @ts-expect-error
  return result
}

const x = objectKeys({a: 10, b: 20})
// x has type ('a' | 'b')[]
```

You can also create mapped types/indexed types, for example:

```
function mirrorKeys<T>(obj: T): {[TKey in keyof T]: TKey} {
  const result = Object.fromEntries(Object.keys(obj).map(k => [k, k]))
  // @ts-expect-error
  return result
}

const x = mirrorKeys({a: 10, b: 20})
// x has type: {a: 'a', b: 'b'}
```

Exercise 14 - Advanced Types

By combining escape hatches and index types, see if you can add types to `objectMap` in `exercises/14-advanced-types/src/objectMap.tsx` so that `mappedObject` has the correct type, `{one: string, two: string}`

Other Topics for Discussion

- Structural type compatibility - different from traditional OOP
- How does type inference work?