# 4

# ESSENTIAL TYPES AND ANNOTATIONS

Now that we've covered most of the *why* of TypeScript, it's time to start with the *how*. In this chapter, we'll cover key concepts such as type annotations and type inference, as well as how to start writing type-safe functions.

It's important to build a solid foundation, as everything you'll learn later builds upon what you'll learn in this chapter.

## Type Annotations

One of the most common tasks you'll need to perform as a TypeScript developer is to annotate your code. Type annotations tell TypeScript what type something is supposed to be. Annotations will often use a colon (`:`), which is used to tell TypeScript that a variable or function parameter is of a certain type.

### The Basic Types

TypeScript has a number of basic types you can use to annotate your code. Here are a few of the most common ones:

```
let example1: string = "Hello World!";
let example2: number = 42;
let example3: boolean = true;
let example4: symbol = Symbol();
let example5: bigint = 123n;
let example6: null = null;
let example7: undefined = undefined;
```

Each of these types is used to tell TypeScript what type a variable or function parameter is supposed to be. The basic types like `string`, `boolean`, and `number` are covered. But TypeScript also has types for JavaScript features you may not have heard of, such as `bigint` and `symbol`. It also has separate types for `null` and `undefined`. You can express much more complex types in TypeScript, such as arrays, objects, functions, and more (we'll cover these in later chapters).

### Function Parameter Annotations

One of the most important annotations you'll use is for function parameters. For example, here is a `logAlbumInfo` function that takes in a `title string`, a `trackCount number`, and an `isReleased boolean`:

```
const logAlbumInfo = (
  title: string,

  trackCount: number,

  isReleased: boolean,
) => {
  // implementation
};
```

Each parameter's type annotation enables TypeScript to check that the arguments passed to the function are of the correct type. If the type doesn't match up, TypeScript will show a red squiggly line under any offending arguments:

```
logAlbumInfo("Black Gold", false, 15); // red squiggly line first under false, then under 15
```

In the previous example, you would first get an error under `false` because a `boolean` can't be passed in place of a number. TypeScript's errors describe this as not being "assignable" to `boolean`. If you fix that like so

```
logAlbumInfo("Black Gold", 20, 15); // red squiggly line under 15
```

you would still have an error under `15` because a number isn't assignable to a `boolean`.

### Variable Annotations

Just as with function parameters, you can also annotate variables. You use variable annotations to explicitly tell TypeScript what you expect the types of your variables to be. Here's an example of some variables with their associated types:

```
let albumTitle: string = "Midnights";
let isReleased: boolean = true;
let trackCount: number = 13;
```

Notice how each variable name is followed by a `:` and its primitive type before its value is set.

Once a variable has been declared with a specific type annotation, TypeScript ensures that the variable remains compatible with the type you specified. For example, this reassignment would work:

```
let albumTitle: string = "Midnights";

albumTitle = "1989"; // albumTitle has been reassigned without error
```

But this one would show an error:

```
let isReleased: boolean = true;

isReleased = "yes"; // red squiggly line under isReleased
```

TypeScript's static type checking is able to spot errors at compile time, which is happening behind the scenes as you write your code.

In the case of the previous `isReleased` example, the error message reads:

```
Type 'string' is not assignable to type 'boolean'.
```

In other words, TypeScript says that it expected `isReleased` to be a `boolean` but instead received a `string`. It's nice to be warned about these kinds of errors before you even run your code!

### Type Inference

TypeScript gives you the ability to annotate almost any value, variable, or function in your code. You might be thinking, "Wait, do I need to annotate everything? That's a lot of extra code." Don't worry. In many cases, TypeScript is smart enough to infer a type where none exists. This is called *type inference*.

### Variables Don't Always Need Annotations

Let's look again at the variable annotation example but drop the annotations:

```
let albumTitle = "Midnights";
let isReleased = true;
let trackCount = 13;
```

We didn't add the annotations, but TypeScript isn't complaining. What's going on?

Try hovering your cursor over each variable, and you'll see the following:

```
// hovering over each variable name shows:

let albumTitle: string;
let isReleased: boolean;
let trackCount: number;
```

Even though they aren't annotated, TypeScript is able to *infer* the type of each.

TypeScript still behaves as if variables have been annotated, warning you if you try to assign it a different type from what it was assigned originally. For example, if you assign `isReleased` the `boolean` `true` and then try to reassign it to a `string`, TypeScript will give you an error message:

```
let isReleased = true;
isReleased = "yes"; // red squiggly line under isReleased
// hovering over isReleased shows:
Type 'string' is not assignable to type 'boolean'.
```

When TypeScript infers the type of a variable, it also provides autocomplete on its associated methods. In this case, since `albumTitle` is inferred as a `string`, typing `albumTitle.toUpper` will allow autocompletion for `toUpperCase`.

This is an extremely powerful part of TypeScript. It means you can mostly *not* annotate variables and your IDE still will know what type things are.

### Function Parameters Always Need Annotations

Type inference can't work everywhere, however. Let's see what happens if you remove the type annotations from the `logAlbumInfo` function's parameters:

```
const logAlbumInfo = (
  title, // error: Parameter 'title' implicitly has an 'any' type.
  trackCount, // error: Parameter 'trackCount' implicitly has an 'any' type.
  isReleased, // error: Parameter 'isReleased' implicitly has an 'any' type.
) => {
  // rest of function body
};
```

Without type annotations for the function parameters, TypeScript now displays errors. This is because functions are very different from variables.

TypeScript can see what value is assigned to which variable, so it can make a good guess about the type, but TypeScript can't tell from a function parameter alone what type it's supposed to be.

It also can't detect the type from usage. If you had an `add` function that took two parameters, TypeScript wouldn't be able to tell that they were supposed to be numbers:

```
function add(a, b) {
  return a + b;
}
```

The `a` and `b` could be strings, Booleans, or anything else. TypeScript can't know from the function body what type they're supposed to be. So, while inference works in most situations, function parameters are an exception. You'll mostly need to annotate them in TypeScript.

**NOTE** *There are a couple of exceptions to this rule. Let's say you're mapping over an array of strings. You pass a function to `.map`, which receives the string you're currently mapping over as an argument:*

```
const strings = ['a', 'b'];
strings.map(str => str.toUppercase());
```

*In this context, the parameter `str` does not need to be annotated because TypeScript can infer it's a string. Very clever!*

### The any Type

The error you just encountered with the `logAlbumInfo` function's parameters was pretty scary:

```
Parameter 'title' implicitly has an 'any' type.
```

When TypeScript doesn't know what type something is, it assigns it the `any` type.

The `any` type disables TypeScript's type system. It turns off type safety on the thing it's assigned to, which means that anything can be assigned to it, any property on it can be accessed or assigned to, and it can be called like a function:

```
let anyVariable: any = "This can be anything!";

anyVariable(); // no error

anyVariable.deep.property.access; // no error
```

This code will error at runtime, but TypeScript isn't giving you a warning!

You can use the `any` type to turn off error messages in TypeScript, which can be a useful escape hatch for when a type is too complex to describe. It's also useful for migrating legacy JavaScript codebases to TypeScript.

Overusing any, however, defeats the purpose of using TypeScript, so it's best to avoid it whenever possible.

## Exercise 4-1: Basic Types with Function Parameters

Let's start with an add function that takes two Boolean parameters, a and b, and returns a + b:

```
export const add = (a: boolean, b: boolean) => {
  return a + b; // red squiggly line under a + b
};
```

Calling the add function creates the result variable, which is then checked to see if it is equal to a number:

```
const result = add(1, 2); // red squiggly line under 1

type test = Expect<Equal<typeof result,
  number>>; // red squiggly line under Equal through number
```

You're using a couple of type annotations, Expect and Equal, from the library @totaltypescript/helpers to act as tests for the exercises. In this case, you expect the result to be a number.

Currently, a few errors in the code are marked by red squiggly lines. The first is on the return line of the add function, where you have a + b:

```
// hovering over a + b shows:

Operator '+' cannot be applied to types 'boolean' and 'boolean'
```

There's also an error after the 1 argument in the add function call:

```
Argument of type 'number' is not assignable to parameter of type 'boolean'
```

Finally, you can see that the test result has an error because the result is currently typed as any, which is not equal to number.

Your challenge is to consider how you can change the types to make the errors go away and to ensure that result is a number. Hover over result to check it.

See *https://totalts.link/essentials-4-1/*.

### Solution

Common sense tells you that the Booleans in the add function should be replaced with some sort of number type. If you are coming from another language, you might be tempted to try using int or float, but TypeScript has only the number type:

```
function add(a: number, b: number) {
  return a + b;
}
```

Making this change resolves the errors and also gives you some other bonuses.

If you try calling the `add` function with a string instead of a number, you'll get an error that type `string` is not assignable to type `number`:

```
add("something", 2); // red squiggly line under something
```

Not only that, but the result of the function is now inferred for you:

```
const result = add(1, 2); // result is a number!
```

TypeScript can infer not only variables but also the return types of functions.

## Exercise 4-2: Annotating Empty Parameters

In this exercise, you have a `concatTwoStrings` function that is similar in shape to the `add` function. It takes two parameters, `a` and `b`, and returns a string:

```
const concatTwoStrings = (a, b) => {. // red squiggly line under a and b
  return [a, b].join(" ");
};
```

The `a` and `b` parameters have errors, which have not been annotated with types:

```
Parameter 'a' implicitly has an 'any' type.

Parameter 'b' implicitly has an 'any' type.
```

The `result` of calling `concatTwoStrings` with `"Hello"` and `"World"` and checking if it is a string does not show any errors:

```
const result = concatTwoStrings("Hello", "World");

type test = Expect<Equal<typeof result, string>>;
```

Your job is to add some function parameter annotations to the `concatTwoStrings` function to make the errors go away.

See *https://totalts.link/essentials-4-2/*.

### Solution

As we've mentioned, function parameters always need annotations in TypeScript. With this in mind, let's update the function declaration parameters so that `a` and `b` are both specified as `string`:

```
const concatTwoStrings = (a: string, b: string) => {
  return [a, b].join(" ");
};
```

This change fixes the errors.

For a bonus point, what type will the return type be inferred as?

```
const result = concatTwoStrings("Hello", "World"); // result is a string!
```

## Exercise 4-3: The Basic Types

As you've seen, TypeScript shows errors when types don't match. The following set of examples shows the basic types that TypeScript gives you to describe JavaScript:

```
export let example1: string = "Hello World!";
export let example2: string = 42; // red squiggly line under example2
export let example3: string = true; // red squiggly line under example3
export let example4: string = Symbol(); // red squiggly line under example4
export let example5: string = 123n; // red squiggly line under example5
```

You'll also notice there are several errors. Hovering over each of the underlined variables displays any associated error messages. For example, hovering over `example2` shows this:

```
Type 'number' is not assignable to type 'string'.
```

The type error for `example3` tells you this:

```
Type 'boolean' is not assignable to type 'string'.
```

Change the types of the annotations on each variable to make the errors go away.

See *https://totalts.link/essentials-4-3/*.

### Solution

Each of the following examples represents one of TypeScript's basic types and would be annotated as follows:

```
let example1: string = "Hello World!";
let example2: number = 42;
let example3: boolean = true;
let example4: symbol = Symbol();
let example5: bigint = 123n;
```

You've already seen `string`, `number`, and `boolean`. The `symbol` type is used for symbols, which are used to ensure that property keys are unique. The `bigint` type is used for JavaScript bigints.

**NOTE** *JavaScript has some limitations on how large numbers can be. Numbers larger than `Number.MAX_SAFE_INTEGER` can't reliably be added together. Bigints solve this problem by letting us represent large numbers in JavaScript.*

However, in practice, you typically won't annotate variables like this. If you remove the explicit type annotations, there won't be any errors at all:

```
let example1 = "Hello World!";
let example2 = 42;
let example3 = true;
let example4 = Symbol();
let example5 = 123n;
```

These basic types are useful to know, even if you don't always need them for your variable declarations.

## Exercise 4-4: The any Type

The following function, called `handleFormData`, accepts an `e` typed as `any`. The function prevents the default form submission behavior and then creates an object from the form data and returns it:

```
const handleFormData = (e: any) => {
  e.preventDefault();

  const data = new FormData(e.terget);

  const value = Object.fromEntries(data.entries());

  return value;
};
```

Here is a test for the function that creates a form, sets the `innerHTML` to add an input, and then manually submits the form. When it submits, you expect the value to equal the value that was in your form that you grafted in there:

```
it("Should handle a form submit", () => {
  const form = document.createElement("form");

  form.innerHTML = `

`;
```

```
  form.onsubmit = (e) => {
    const value = handleFormData(e);

    expect(value).toEqual({ name: "John Doe" });
  };

  form.requestSubmit();

  expect.assertions(1);
});
```

This isn't the normal way you would test a form, but it provides a way to test the example `handleFormData` function more extensively.

In the code's current state, no red squiggly lines are present. However, when running the test with Vitest, you get an error similar to the following:

```
This error originated in "any.problem.ts" test file.
It doesn't mean the error was thrown inside the file itself, but while it was running.
The latest test that might've caused the error is "Should handle a form submit".
It might mean one of the following:
- The error was thrown, while Vitest was running this test.
- This was the last recorded test before the error was thrown,
if error originated after test finished its execution.
```

Why is this error happening? Why isn't TypeScript giving you an error here?

We'll give you a clue. We've hidden a nasty typo in there. Can you fix it? See *https://totalts.link/essentials-4-4/*.

### Solution

In this case, using `any` did not help at all. In fact, the `any` annotations seem to actually turn off type checking! With the `any` type, you're free to do anything you want to the variable, and TypeScript will not prevent it. Using `any` also disables useful features like autocompletion, which can help you avoid typos. That's right—the error in the exercise code was caused by a typo of `e.terget` instead of `e.target` when creating the `FormData`:

```
const handleFormData = (e: any) => {
  e.preventDefault();

  const data = new FormData(e.terget); // e.terget! Whoops!

  const value = Object.fromEntries(data.entries());

  return value;
};
```

If `e` had been properly typed, TypeScript would have caught the error right away. We'll come back to this example in the future to see the proper typing.

Using `any` may seem like a quick fix when you have trouble figuring out how to properly type something, but it can come back to bite you later.

## Object Literal Types

Now that we've covered basic types, let's move on to object types. *Object types* are used to describe the shape of objects. Each property of an object can have its own type annotation. When defining an object type, you use curly brackets to contain the properties and their types:

```
const talkToAnimal = (animal: { name: string; type: string; age: number }) => {
  // rest of function body
};
```

This curly bracket syntax is called an *object literal type*.

## Optional Object Properties

You can use the `?` operator to mark the `age` property as optional:

```
const talkToAnimal = (animal: { name: string; type: string; age?: number }) => {
  // rest of function body
};
```

This means you don't need to provide the `age` property when calling the function.

One cool thing about type annotations with object literals is that they provide autocompletion for the property names while you're typing. For instance, when calling `talkToAnimal`, it will provide an autocomplete drop-down menu with suggestions for the `name`, `type`, and `age` properties. This can save you a lot of time, and it also helps you avoid typos in situations when you have several properties with similar names.

## Exercise 4-5: Object Literal Types

Here you have a `concatName` function that accepts a `user` object with `first` and `last` keys:

```
const concatName = (user) => { // red squiggly line under user
  return `${user.first} ${user.last}`;
};
```

The test expects that the full name should be returned. Since the full name *is* being returned, the test is passing:

```
it("should return the full name", () => {
  const result = concatName({
    first: "John",
    last: "Doe",
  });
```

```
  type test = Expect<Equal<typeof result, string>>;

  expect(result).toEqual("John Doe");
});
```

However, there is a familiar error on the `user` parameter in the `concatName` function:

```
Parameter 'user' implicitly has an 'any' type.
```

You can tell from the `concatName` function body that it expects `user.first` and `user.last` to be strings. How could you type the `user` parameter to ensure that it has these properties and that they are of the correct type?

See *https://totalts.link/essentials-4-5/*.

### Solution

To annotate the `user` parameter as an object, you can use the curly bracket syntax. Let's start by annotating the `user` parameter as an empty object:

```
const concatName = (user: {}) => {
  return `${user.first} ${user.last}`; // red squiggly line under .first and .last
};
```

The errors change, which is progress. The errors now show up under `.first` and `.last` in the function return:

```
Property 'first' does not exist on type '{}'.
Property 'last' does not exist on type '{}'.
```

To fix these errors, you need to add the `first` and `last` properties to the type annotation:

```
const concatName = (user: { first: string; last: string }) => {
  return `${user.first} ${user.last}`;
};
```

Now TypeScript knows that both the `first` and `last` properties of `user` are strings, and the test passes.

## Exercise 4-6: Optional Property Types

Here's a version of the `concatName` function that has been updated to return only the first name if a last name wasn't provided:

```
const concatName = (user: { first: string; last: string }) => {
  if (!user.last) {
    return user.first;
  }
```

```
  return `${user.first} ${user.last}`;
};
```

Like before, TypeScript gives you an error when testing that the function returns only the first name:

```
it("should return only the first name if last name not provided", () => {
  const result = concatName({
    first: "John",
  }); // red squiggly line under {first: "John"}

  type test = Expect<Equal<typeof result, string>>;

  expect(result).toEqual("John");
});
```

This time the entire `{first: "John"}` object is underlined in red, and the error message reads:

```
Argument of type '{ first: string; }' is not assignable to
  parameter of type '{ first: string; last: string; }'.
Property 'last' is missing in type '{ first: string; }' but
  required in type '{ first: string; last: string; }'.
```

The error tells you that you are missing a property, but the error is incorrect. You *do* want to support objects that include only a `first` property. In other words, `last` needs to be optional.

How would you update this function to fix the errors?

See *https://totalts.link/essentials-4-6/*.

### Solution

Similar to when you set a function parameter as optional, you can use the question mark (`?`) to specify that an object's property is optional:

```
function concatName(user: { first: string; last?: string }) {
  // implementation
}
```

Adding `?:` indicates to TypeScript that the property doesn't need to be present.

If you hover over the `last` property in the function body, you'll see that the `last` property is `string | undefined`:

```
// hovering over user.last shows:
(property) last?: string | undefined
```

This means it's `string` *or* `undefined`. It's a useful bit of TypeScript syntax that you'll see more of in the future.

## Type Aliases

So far, you've been declaring all of your types inline, which is fine for these simple examples, but in a real application, you're going to have types that repeat a lot across your app. They might be users, products, or other domain-specific types, and you won't want to have to repeat the same type definition in every file that needs it.

This is where the `type` keyword comes in. It allows you to define a type once and use it in multiple places:

```
type Animal = {
  name: string;
  type: string;
  age?: number;
};
```

It's a way to give a name to a type—in other words, a *type alias*—and then use that name wherever you need to use that type.

To create a new variable with the `Animal` type, you'll add it as a type annotation after the variable name:

```
let pet: Animal = {
  name: "Karma",
  type: "cat",
};
```

You can also use the `Animal` type alias in place of the object type annotation in a function

```
const getAnimalDescription = (animal: Animal) => {};
```

and call the function with your `pet` variable:

```
const desc = getAnimalDescription(pet);
```

Type aliases can be objects, but they can also use basic types:

```
type Id = string | number;
```

We'll cover this syntax in more detail later (it's called a *union type*), but it's basically saying that an `Id` can be either a `string` or a `number`.

Using a type alias is a great way to ensure that there's a single source of truth for a type definition, which makes it easier to make changes in the future.

## Sharing Types Across Modules

Type aliases can be created in their own *.ts* files and imported into the files where you need them, which is useful when sharing types in multiple places or when a type definition gets too large:

```
// In shared-types.ts

export type Animal = {
  width: number;
  height: number;
};

// In index.ts
import type { Animal } from "./shared-types";
```

As a convention, you can even create your own *.types.ts* files, which can help keep your type definitions separate from your other code.

## Exercise 4-7: The type Keyword

The following code uses the same type in multiple places:

```
const getRectangleArea = (rectangle: { width: number; height: number }) => {
  return rectangle.width * rectangle.height;
};

const getRectanglePerimeter = (rectangle: {
  width: number;
  height: number;
}) => {
  return 2 * (rectangle.width + rectangle.height);
};
```

The `getRectangleArea` and `getRectanglePerimeter` functions both take in a `rectangle` object with `width` and `height` properties.

Tests for each function pass as expected:

```
it("should return the area of a rectangle", () => {
  const result = getRectangleArea({
    width: 10,
    height: 20,
  });

  type test = Expect<Equal<typeof result, number>>;

  expect(result).toEqual(200);
});

it("should return the perimeter of a rectangle", () => {
  const result = getRectanglePerimeter({
    width: 10,
    height: 20,
  });

  type test = Expect<Equal<typeof result, number>>;
```

```
  expect(result).toEqual(60);
});
```

Even though everything is working, there's an opportunity for refactoring to clean things up. How could you use the `type` keyword to make this code more readable?

See *https://totalts.link/essentials-4-7/*.

#### Solution

You can use the `type` keyword to create a `Rectangle` type with `width` and `height` properties:

```
type Rectangle = {
  width: number;
  height: number;
};
```

With the `type` alias created, you can update the `getRectangleArea` and `getRectanglePerimeter` functions to use the `Rectangle` type:

```
const getRectangleArea = (rectangle: Rectangle) => {
  return rectangle.width * rectangle.height;
};

const getRectanglePerimeter = (rectangle: Rectangle) => {
  return 2 * (rectangle.width + rectangle.height);
};
```

This makes the code a lot more concise and gives you a single source of truth for the `Rectangle` type.

## Arrays

You can also describe the types of arrays in TypeScript. There are two different syntaxes for doing this. The first option is the square bracket syntax, which is similar to the type annotations you've made so far but with the addition of two square brackets (`[ ]`) at the end to indicate an array:

```
let albums: string[] = [
  "Rubber Soul",
  "Revolver",
  "Sgt. Pepper's Lonely Hearts Club Band",
];

let dates: number[] = [1965, 1966, 1967];
```

The second option is to explicitly use the `Array` type with angle brackets (`< >`) containing the type of data the array will hold:

```
typescript
let albums: Array<string> = [
  "Rubber Soul",
  "Revolver",
  "Sgt. Pepper's Lonely Hearts Club Band",
];
```

Both of these syntaxes are equivalent, but the square bracket syntax is a bit more concise when creating arrays. It's also the way that TypeScript presents error messages. Keep the angle bracket syntax in mind, though; you'll see more examples of it later in this chapter.

### *Arrays of Objects*

You have several choices when annotating an array of objects. You can specify the object inline:

```
type ArrayOfAlbums = { artist: string; title: string; year: number }[]
```

Or you can use a `type` alias and wrap it with `Array<>` or append square brackets:

```
type Album = { artist: string; title: string; year: number }
type ArrayOfAlbums = Array<Album>; // option 1
type ArrayOfAlbums = Album[]; // option 2
```

Finally, you can use the type like so:

```
// create an array of Albums
let selectedDiscography: ArrayOfAlbums = [
  {
    artist: "The Beatles",
    title: "Rubber Soul",
    year: 1965,
  },
  {
    artist: "The Beatles",
    title: "Revolver",
    year: 1966,
  },
];
```

If you try to update the array with an item that doesn't match the type, TypeScript will give you an error:

```
selectedDiscography.push({name: "Karma", type: "cat"};) // red squiggly line under name
// error message:
// Argument of type '{ name: string; type: string; }' is not
   assignable to parameter of type 'Album'.
```

### Tuples

Tuples let you specify an array with a fixed number of elements, where each element has its own type. Creating a tuple is similar to an array's square bracket syntax, except the square brackets contain the types instead of abutting the variable name:

```
// Tuple
let album: [string, number] = ["Rubber Soul", 1965];

// Array
let albums: string[] = [
  "Rubber Soul",
  "Revolver",
  "Sgt. Pepper's Lonely Hearts Club Band",
];
```

Tuples are useful for grouping related information together without having to create a new type. For example, if you wanted to group an album with its play count, you could do something like this:

```
let albumWithPlayCount: [Album, number] = [
  {
    artist: "The Beatles",
    title: "Revolver",
    year: 1965,
  },
  10000,
];
```

Now the `albumWithPlayCount` tuple can be indexed similarly to an array—at index `0` is the `Album`, and at index `1` is `1000`.

### Named Tuples

To add more clarity to the tuple, you can add names for each of the types in the square brackets:

```
type MyTuple = [album: Album, playCount: number];
```

This can be helpful when you have a tuple with a lot of elements or when you want to make the code more readable.

**NOTE** *The names of the tuples don't do anything at runtime; they're just descriptions to make it clear what should go in each element.*

## Exercise 4-8: Array Type

Consider the following shopping cart code:

```
type ShoppingCart = {
  userId: string;
};

const processCart = (cart: ShoppingCart) => {
  // Do something with the cart in here
};

processCart({
  userId: "user123",
  items: ["item1", "item2", "item3"], // red squiggly line under items
});
```

We have a `type` alias for `ShoppingCart` that currently has a `userId` property of type `string`. The `processCart` function takes in a `cart` parameter of type `ShoppingCart`. Its implementation doesn't matter at this point. What does matter is that when you call `processCart`, you are passing in an object with a `userId` and an `items` property that is an array of strings.

There is an error underneath `items` that reads:

```
Argument of type '{ userId: string; items: string[]; }' is not
  assignable to parameter of type 'ShoppingCart'.
Object literal may only specify known properties, and
  'items' does not exist in type 'ShoppingCart'.
```

As the error message points out, there is not currently a property called `items` on the `ShoppingCart` type. How would you change the types to fix this error?

See *https://totalts.link/essentials-4-8/*.

### Solution

For the `ShoppingCart` example, defining an array of `item` strings would look like the following when using the square bracket syntax:

```
type ShoppingCart = {
  userId: string;
  items: string[];
};
```

With this in place, you must pass in `items` as an array. A single string or other type would result in a type error.

The other syntax is to write `Array` explicitly and pass it a type in the angle brackets:

```
type ShoppingCart = {
  userId: string;
  items: Array<string>;
};
```

## Exercise 4-9: Arrays of Objects

Consider this `processRecipe` function that takes in a `Recipe` type:

```
type Recipe = {
  title: string;
  instructions: string;
};

const processRecipe = (recipe: Recipe) => {
  // Do something with the recipe in here
};

processRecipe({
  title: "Chocolate Chip Cookies",
  ingredients: [
    { name: "Flour", quantity: "2 cups" },
    { name: "Sugar", quantity: "1 cup" },
    // other ingredients here...
  ],
  instructions: "...",
});
```

The function is called with an object containing `title`, `instructions`, and `ingredients` properties, but there are errors because the `Recipe` type doesn't currently have an `ingredients` property:

```
Argument of type '{title: string; ingredients: {
  name: string;
  quantity: string; }[];
  instructions: string;
}' is not assignable to parameter of type 'Recipe'.

Object literal may only specify known properties, and 'ingredients'
does not exist in type 'Recipe'.
```

By combining what you've seen with typing object properties and working with arrays, how would you specify ingredients for the `Recipe` type?

See *https://totalts.link/essentials-4-9/*.

### Solution

There are a few different ways to express an array of objects. One approach is to create a new `Ingredient` type that you can use to represent the objects in the array:

```
type Ingredient = {
  name: string;
  quantity: string;
};
```

Then, the `Recipe` type can be updated to include an `ingredients` property of type `Ingredient[]`:

```
type Recipe = {
  title: string;
  instructions: string;
  ingredients: Ingredient[];
};
```

This solution reads nicely, fixes the errors, and helps create a mental map of your domain model.

As shown previously, using the `Array<Ingredient>` syntax would also work:

```
type Recipe = {
  title: string;
  instructions: string;
  ingredients: Array<Ingredient>;
};
```

It's also possible to specify the `ingredients` property as an inline object literal on the `Recipe` type using square brackets:

```
type Recipe = {
  title: string;
  instructions: string;
  ingredients: {
    name: string;
    quantity: string;
  }[];
};
```

Or using `Array<>` with the object literal inside:

```
type Recipe = {
  title: string;
  instructions: string;
  ingredients: Array<{
    name: string;
    quantity: string;
  }>;
};
```

The inline approaches are useful, but we prefer extracting them to a new type. This means that if another part of your application needs to use the `Ingredient` type, it can.

## Exercise 4-10: Tuples

Here you have a `setRange` function that takes in an array of numbers:

```
const setRange = (range: Array) => {
  const x = range[0];
  const y = range[1];

  // Do something with x and y in here
  // x and y should both be numbers!

  type tests = [
    Expect<Equal<typeof x, number>>, // red squiggly line under Equal<> statement
    Expect<Equal<typeof y, number>>, // red squiggly line under Equal<> statement
  ];
};
```

In the function, you grab the first element of the array and assign it to `x`, and you grab the second element of the array and assign it to `y`. Two tests in the `setRange` function are currently failing.

Using the `// @ts-expect-error` directive, you find a couple more errors that need fixing. Recall that this directive tells TypeScript you know there will be an error on the next line, and it should ignore it. However, if you say you expect an error but there isn't one, you will get the red squiggly line under the actual `// @ts-expect-error` line:

```
// both of these show a red squiggly line under the ts-expect-error directive

// @ts-expect-error too few arguments
setRange([0]);

// @ts-expect-error too many arguments
setRange([0, 10, 20]);
```

The code for the `setRange` function needs an updated type annotation to specify that it accepts only a tuple of two numbers.

See *https://totalts.link/essentials-4-10/*.

### Solution

In this case, you would update the `setRange` function to use the tuple syntax instead of the array syntax:

```
const setRange = (range: [number, number]) => {
  // rest of function body
};
```

If you want to add more clarity to the tuple, you can add names for each of the types:

```
const setRange = (range: [x: number, y: number]) => {
  // rest of function body
};
```

## Exercise 4-11: Optional Members of Tuples

This `goToLocation` function takes in an array of coordinates. Each coordinate has a `latitude` and `longitude`, which are both numbers, as well as an optional `elevation`, which is also a number:

```
const goToLocation = (coordinates: Array) => {
  const latitude = coordinates[0];
  const longitude = coordinates[1];
  const elevation = coordinates[2];

  // Do something with latitude, longitude, and elevation in here

  type tests = [
    Expect<Equal<typeof latitude, number>>, // red squiggly line under Equal<> statement
    Expect<Equal<typeof longitude, number>>, // red squiggly line under Equal<> statement
    Expect<Equal<typeof elevation, number | undefined>>,
  ];
};
```

Your challenge is to update the type annotation for the `coordinates` parameter to specify that it should be a tuple of three numbers, where the third number is optional.

See *https://totalts.link/essentials-4-11/*.

### Solution

The fix here is to provide an optional modifier (`?`) to the member of the tuple:

```
const goToLocation = (
  coordinates: [latitude: number, longitude: number, elevation?: number],
) => {};
```

The values are clear, and using the `?` modifier specifies that `elevation` is an optional number. It almost looks like an object, but it's still a tuple.

Alternatively, if you don't want to use named tuples, you can use the `?` modifier after the definition:

```
const goToLocation = (coordinates: [number, number, number?]) => {};
```

## Passing Types to Functions

Let's take a quick look back at the `Array` type discussed earlier:

```
Array<string>;
```

This type describes an array of strings. To make that happen, you're passing a type (`string`) as an argument to another type (`Array`).

There are lots of other types that can receive types as arguments, like `Promise<string>`, `Record<string, string>`, and more. In each of them, you use the angle brackets to pass a type to another type, but you can also use that syntax to pass types to functions.

## Passing Types to Set

A `Set` is a JavaScript feature that represents a collection of unique values. To create a `Set`, use the `new` keyword and call `Set`:

```
const formats = new Set();
```

If you hover over the `formats` variable, you can see that it is typed as `Set<unknown>`:

```
// hovering over formats shows:
const formats: Set<unknown>;
```

That's because the `Set` doesn't know what type it's supposed to be! You haven't passed it any values, so it defaults to an `unknown` type. You'll look at `unknown` in much more depth in the next chapter.

One way to make TypeScript know what type you want the `Set` to hold is to pass in some initial values:

```
const formats = new Set(["CD", "DVD"]);
```

In this case, since you specified two strings when creating the `Set`, TypeScript knows that `formats` is a `Set` of strings:

```
// hovering over formats shows:
const formats: Set<string>;
```

But it's not always the case that you know exactly what values you want to pass to a `Set` when you create it. You might want to create an empty `Set` that you know will hold strings later.

For this, you can pass a type to `Set` using the angle bracket syntax:

```
const formats = new Set<string>();
```

Now `formats` understands that it's a set of strings, and adding anything other than a string will fail:

```
formats.add("Digital"); // this works

formats.add(8); // red squiggly line under 8

// hovering over 8 shows:

Argument of type 'number' is not assignable to parameter of type 'string'
```

This is a really important thing to understand in TypeScript. You can pass types as well as values to functions.

## Not All Functions Can Receive Types

Most functions in TypeScript *can't* receive types. For example, let's look at `document.getElementById` that comes in from the DOM typing. A common scenario where you might want to pass a type is when calling `document.getElementById`. Here you're trying to get an audio element:

```
const audioElement = document.getElementById("player");
```

You know that `audioElement` is going to be an `HTMLAudioElement`, so it seems like you should be able to pass it to `document.getElementById`. However, you can't:

```
const audioElement = document.getElementById<HTMLAudioElement>("player");
// red squiggly line under HTMLAudioElement
```

When you hover over `HTMLAudioElement`, the error tells you that it expects zero type arguments, but you passed it one—in this case, `player`.

You can see whether a function can receive type arguments by hovering over it. Let's try hovering over `.getElementById`:

```
// hovering over .getElementById shows:
(method) Document.getElementById(elementId: string): HTMLElement | null
```

Notice that `.getElementById` contains no angle brackets in its hover, which is why you can't pass a type to it.

Let's contrast it with a function that *can* receive type arguments, like `document.querySelector`:

```
const audioElement = document.querySelector("#player");

// hovering over .querySelector shows:
(method) ParentNode.querySelector<Element>(selectors: string): Element | null
```

This type definition shows you that `.querySelector` has some angle brackets before the parentheses. In the brackets is the default value—in this case, `Element`. To fix the code, you could replace `.getElementById` with `.querySelector` and use the `#player` selector to find the audio element:

```
const audioElement = document.querySelector<HTMLAudioElement>("#player");
```

And everything works.

To tell whether a function can receive a type argument, hover over it and check whether it has any angle brackets.

## Exercise 4-12: Passing Types to Map

Here you are creating a `Map`, a JavaScript feature that represents a dictionary. In this case, you want to pass in a number for the key and an object for the value:

```
const userMap = new Map();

userMap.set(1, { name: "Max", age: 30 });

userMap.set(2, { name: "Manuel", age: 31 });

// @ts-expect-error // red squiggly line under @ts-expect-error
userMap.set("3", { name: "Anna", age: 29 });

// @ts-expect-error // red squiggly line under @ts-expect-error
userMap.set(3, "123");
```

There are red lines under the `@ts-expect-error` directives because currently any type of key and value is allowed in the `Map`:

```
// hovering over Map shows:
var Map: MapConstructor

new () => Map<any, any> (+3 overloads)
```

How would you type the `userMap` so that the key must be a number and the value is an object with `name` and `age` properties?

See *https://totalts.link/essentials-4-12/*.

### Solution

There are a few different ways to solve this problem, but we'll start with the most straightforward one. The first thing to do is to create a `User` type:

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

Following the patterns you've seen so far, you can pass `number` and `User` as the types for the `Map`:

```
const userMap = new Map<number, User>();
```

That's right, some functions can receive *multiple* type arguments. In this case, the `Map` constructor can receive two types: one for the key and one for the value.

With this change, the errors go away, and you can no longer pass incorrect types into the `userMap.set` function.

You can also express the `User` type inline:

```
const userMap = new Map<number, { name: string; age: number }>();
```

## Exercise 4-13: JSON.parse( ) Can't Receive Type Arguments

Consider the following code, which uses `JSON.parse()` to parse some JSON:

```
const parsedData = JSON.parse<{
  name: string; // red squiggly line under the full {} argument
  age: number;
}>('{"name": "Alice", "age": 30}');
```

There is currently an error under the type argument for `JSON.parse`:

```
Expected 0 type arguments, but got 1.
```

A test that checks the type of `parsedData` is currently failing, since it is typed as `any` instead of the expected type:

```
type test = Expect<
  Equal<
    // red squiggly line under the full Equal<>
    typeof parsedData,
    {
      name: string;
      age: number;
    }
  >
>;

it("Should be the correct shape", () => {
  expect(parsedData).toEqual({
    name: "Alice",
    age: 30,
  });
});
```

You've tried to pass a type argument to the `JSON.parse` function, but it doesn't appear to be working in this case. The test errors tell you that the type of `parsed` is not what you expect. The properties `name` and `age` are not being recognized.

Why is this happening? What would be a different way to correct these type errors?

See *https://totalts.link/essentials-4-13/*.

### Solution

Let's look a bit more closely at the error message you get when passing a type argument to `JSON.parse`:

```
Expected 0 type arguments, but got 1.
```

This message indicates that TypeScript is not expecting anything in the angle brackets when calling `JSON.parse`. To resolve this error, you can remove the angle brackets:

```
const parsedData = JSON.parse('{"name": "Alice", "age": 30}');
```

Now that `.parse` is receiving the correct number of type arguments, TypeScript is happy. However, you want your parsed data to have the correct type. Hovering over `JSON.parse`, you can see its type definition:

```
JSON.parse(text: string, reviver?: ((this:
  any, key: string, value: any) => any) undefined): any
```

It always returns `any`, which is a bit of a problem.

To get around this issue, you can give `parsedData` a variable type annotation with `name: string` and `age: number`:

```
const parsedData: {
  name: string;
  age: number;
} = JSON.parse('{"name": "Alice", "age": 30}');
```

Now you have `parsedData` typed as you want it to be.

The reason this works is because `any` disables type checking. So, you can assign it any type you want. You could assign it something that doesn't make sense, like `number`, and TypeScript wouldn't complain:

```
const parsedData: number = JSON.parse('{"name": "Alice", "age": 30}');
```

This is more "type faith" than "type safe." You are hoping that `parsedData` is the type you expect it to be. This relies on us keeping the type annotation up to date with the actual data.

## Typing Functions

Let's look at some additional ways to annotate function parameters.

### *Optional Parameters*

For cases where a function parameter is optional, you can add the `?` modifier before the `:`. Say you wanted to add an optional `releaseDate` parameter to the `logAlbumInfo` function. You could do so like this:

```
const logAlbumInfo = (
  title: string,
  trackCount: number,
  isReleased: boolean,
  releaseDate?: string,
) => {
  // rest of function body
};
```

Now you can call `logAlbumInfo` and include a release date string or leave it out:

```
logAlbumInfo("Midnights", 13, true, "2022-10-21");

logAlbumInfo("American Beauty", 10, true);
```

Hovering over the optional `releaseDate` parameter shows that it is now typed as `string | undefined`.

We'll discuss the `|` symbol later, but this means the parameter could be either a `string` or `undefined`. It would be acceptable to literally pass `undefined` as a second argument, or it can be omitted altogether.

### *Default Parameters*

In addition to marking parameters as optional, you can set default values for parameters by using the `=` operator.

For example, you could set the `format` to default to `"CD"` if no format is provided:

```
const logAlbumInfo = (
  title: string,
  trackCount: number,
  isReleased: boolean,
  format: string = "CD",
) => {
  // rest of function body
};
```

The annotation of `: string` can also be omitted since it can infer the type of the `format` parameter from the value provided, which is another nice example of type inference:

```
const logAlbumInfo = (
  title: string,
  trackCount: number,
  isReleased: boolean,
  format = "CD",
) => {
  // rest of function body
};
```

### Function Return Types

In addition to setting parameter types, you can also set the return type of a function. You can annotate the return type of a function by placing a `:` and the type after the closing parenthesis of the parameter list. For the `logAlbumInfo` function, you can specify that the function will return a string:

```
const logAlbumInfo = (
  title: string,
  trackCount: number,
  isReleased: boolean,
): string => {
  // rest of function body
};
```

If the value returned from a function doesn't match the type that was specified, TypeScript will show an error:

```
const logAlbumInfo = (
  title: string,
  trackCount: number,
  isReleased: boolean,
): string => {
  return 123; // red squiggly line under 123
};
```

Return types are useful for when you want to ensure that a function returns a specific type of value.

### Rest Parameters

Just like in JavaScript, TypeScript supports rest parameters by using the `...` syntax for the final parameter, which allows you to pass in any number of arguments to a function. For example, this `printAlbumFormats` is set up to accept an `album` and any number of `formats`:

```
function getAlbumFormats(album: Album, ...formats: string[]) {
  return `${album.title} is available in the following formats: ${formats.join(
    ", ",
  )}`;
}
```

Declaring the parameter with the `...formats` syntax combined with an array of strings lets us pass in any number of strings to the function

```
getAlbumFormats(
  { artist: "Radiohead", title: "OK Computer", year: 1997 },
  "CD",
  "LP",
  "Cassette",
);
```

or even spread in an array of strings:

```
const albumFormats = ["CD", "LP", "Cassette"];

getAlbumFormats(
  { artist: "Radiohead", title: "OK Computer", year: 1997 },
  ...albumFormats,
);
```

This means that `formats` will be passed as an array to the function.

As an alternative, you can use the `Array<>` syntax:

```
function getAlbumFormats(album: Album, ...formats: Array<string>) {
  // function body
}
```

### *Function Types*

We've shown how to use type annotations to specify the types of function parameters, but you can also use TypeScript to describe the types of the functions themselves. You can do this using the following syntax:

```
type Mapper = (item: string) => number;
```

This is a type alias for a function that takes in a `string` and returns a `number`.

You could then use this to describe a callback function passed to another function:

```
const mapOverItems = (items: string[], map: Mapper) => {
  return items.map(map);
};
```

Or, you could declare it inline:

```
const mapOverItems = (items: string[], map: (item: string) => number) => {
  return items.map(map);
};
```

This lets you pass a function to `mapOverItems` that changes the value of the items in the array:

```
const arrayOfNumbers = mapOverItems(["1", "2", "3"], (item) => {
  return parseInt(item) * 100;
});
```

Function types are as flexible as function definitions. You can declare multiple parameters, rest parameters, and optional parameters:

```
// Optional parameters
type WithOptional = (index?: number) => number;

// Rest parameters
type WithRest = (...rest: string[]) => number;

// Multiple parameters
type WithMultiple = (first: string, second: string) => number;
```

### *The void Type*

Some functions don't return anything. They perform some kind of action, but they don't produce a value. A great example is `console.log`:

```
const logResult = console.log("Hello!");
```

What type do you expect `logResult` to be? In JavaScript, the value is `undefined`. If you were to use `console.log(logResult)`, that's what you'd see in the console. But TypeScript has a special type for these situations when a function's return value should be deliberately ignored. It's called `void`. If you hover over `.log` in `console.log`, you'll see that it returns `void`:

```
(method) Console.log(...data: any[]): void
```

So, `logResult` is also `void`. This is TypeScript's way of saying "ignore the result of this function call."

### *Async Functions*

You've seen how to strongly type what a function returns via a return type:

```
const getUser = (id: string): User => {
  // function body
};
```

But what about when the function is asynchronous?

```
const getUser = async (id: string): User => { // red squiggly line under User
  // function body
};
// hovering over User shows:
The return type of an async function or method must be
the global Promise type. Did you mean to write 'Promise'?
```

Fortunately, TypeScript's error message is helpful here. It's telling you that the return type of an async function must be a `Promise`, so you can pass `User` to a `Promise`:

```
const getUser = async (id: string): Promise<User> => {
  const user = await db.users.get(id);

  return user;
};
```

Now your function must return a `Promise` that resolves to a `User`.

## Exercise 4-14: Optional Function Parameters

Here you have a `concatName` function whose implementation takes in two `string` parameters, `first` and `last`.

If there's no `last` name passed, the return would be just the `first` name. Otherwise, it would return `first` concatenated with `last`:

```
const concatName = (first: string, last: string) => {
  if (!last) {
    return first;
  }

  return `${first} ${last}`;
};
```

When calling `concatName` with a first and last name, the function works as expected without errors:

```
const result = concatName("John", "Doe");
```

However, when calling `concatName` with just a first name, you get an error:

```
const result2 = concatName("John"); // red squiggly line under concatName("John")
```

The error message reads:

```
Expected 2 arguments, but got 1.
```

Try to use an optional parameter annotation to fix the error. See *https://totalts.link/essentials-4-14/*.

### Solution

By adding a question mark to the end of a parameter, it will be marked as optional:

```
function concatName(first: string, last?: string) {
  // ...implementation
}
```

## Exercise 4-15: Default Function Parameters

Here you have the same `concatName` function as before, where the `last` name is optional:

```
const concatName = (first: string, last?: string) => {
  if (!last) {
    return first;
  }

  return `${first} ${last}`;
};
```

You also have a couple of tests. The first test checks that the function returns the full name when passed a first and last name:

```
it("should return the full name", () => {
  const result = concatName("John", "Doe");

  type test = Expect<Equal<typeof result, string>>;

  expect(result).toEqual("John Doe");
});
```

However, the second test expects that when `concatName` is called with just a first name as an argument, the function should use `Pocock` as the default last name:

```
it("should return the first name", () => {
  const result = concatName("John");

  type test = Expect<Equal<typeof result, string>>;

  expect(result).toEqual("John Pocock");
});
```

This test currently fails, with the output from Vitest indicating the error is on the `expect` line:

```
AssertionError: expected 'John' to deeply equal 'John Pocock'

- Expected

+ Received
- John Pocock

+ John

expect(result).toEqual("John Pocock");
```

Update the `concatName` function to use `Pocock` as the default last name if one is not provided.

See *https://totalts.link/essentials-4-15/*.

### Solution

To add a default parameter in TypeScript, you use the `=` syntax that is also used in JavaScript. In this case, you update `last` to default to `"Pocock"` if no value is provided:

```
export const concatName = (first: string, last?: string = "Pocock") => {
  return `${first} ${last}`;
};
```

While this passes your runtime tests, it actually fails in TypeScript:

```
Parameter cannot have question mark and initializer.
```

This is because TypeScript doesn't allow us to have both an optional parameter and a default value. The optionality is already implied by the default value.

To fix the error, remove the question mark from the `last` parameter:

```
export const concatName = (first: string, last = "Pocock") => {
  return `${first} ${last}`;
};
```

## Exercise 4-16: Rest Parameters

Here you have a `concatenate` function that takes in a variable number of strings:

```
export function concatenate(...strings) {
  // red squiggly line under `...strings`
  return strings.join("");
}
```

The code runs as expected, but there's an error on the `...strings` rest parameter:

```
// hovering over ...strings shows:
Rest parameter 'strings' implicitly has an 'any[]' type.
```

How would you update the rest parameter to specify that it should be an array of strings?

See *https://totalts.link/essentials-4-16/*.

### Solution

When using rest parameters, all of the arguments passed to the function will be collected into an array. This means that the `strings` parameter can be typed as an array of strings:

```
export function concatenate(...strings: string[]) {
  return strings.join("");
}
```

Or, of course, it can be typed using the `Array<>` syntax:

```
export function concatenate(...strings: Array<string>) {
  return strings.join("");
}
```

## Exercise 4-17: Function Types

Here you have a `modifyUser` function that takes in an array of `users`, an `id` of the user that you want to change, and a `makeChange` function that makes that change:

```
type User = {
  id: string;
  name: string;
};

const modifyUser = (user: User[], id: string, makeChange) => {
  // red squiggly line under `makeChange`

  return user.map((u) => {
    if (u.id === id) {
      return makeChange(u);
    }

    return u;
  });
};
```

Currently there is an error under `makeChange`:

```
Parameter `makeChange` implicitly has an `any` type.
```

Here's an example of how this function would be called:

```
const users: User[] = [
  { id: "1", name: "John" },
  { id: "2", name: "Jane" },
];

modifyUser(users, "1", (user) => {
  // red squiggly line under user
  return { ...user, name: "Waqas" };
});
```

In the previous example, the `user` parameter to the error function also has the "implicit `any`" error.

The `modifyUser` type annotation for the `makeChange` function needs to be updated so that it returns a modified user. For example, you should not be able to return a `name` of `123` because in the `User` type, `name` is a `string`:

```
modifyUser(
  users,
  "1",
  // @ts-expect-error
  (user) => {
    return { ...user, name: 123 };
  },
);
```

How would you type `makeChange` as a function so that it takes in a `User` and returns a `User`?

See *https://totalts.link/essentials-4-17/*.

### Solution

Let's start by annotating the `makeChange` parameter to be a function. For now, you'll specify that it returns `any`:

```
const modifyUser = (user: User[], id: string, makeChange: () => any) => {
  return user.map((u) => {
    if (u.id === id) {
      return makeChange(u); // red squiggly line under `u`
    }

    return u;
  });
};
```

With this first change in place, you get an error under `u` when calling `makeChange` since you said that `makeChange` takes in no arguments:

```
// In the user.map() function

return makeChange(u)

// hovering over u shows:

Expected 0 arguments, but got 1.
```

This says you need to add a parameter to the `makeChange` function type. In this case, you will specify that `user` is of type `User`:

```
const modifyUser = (
  user: User[],
  id: string,
  makeChange: (user: User) => any,
) => {
  // function body
};
```

This is pretty good, but you also need to make sure your `makeChange` function returns a `User`:

```
const modifyUser = (
  user: User[],
  id: string,
  makeChange: (user: User) => User,
) => {
  // function body
};
```

Now the errors are resolved, and you have autocompletion for the `User` properties when writing a `makeChange` function.

Optionally, you can clean up the code a bit by creating a type alias for the `makeChange` function type:

```
type MakeChangeFunc = (user: User) => User;

const modifyUser = (user: User[], id: string, makeChange: MakeChangeFunc) => {
  // function body
};
```

Both techniques behave the same, but if you need to reuse the `makeChange` function type, a type alias is the way to go.

## Exercise 4-18: Functions Returning void

Here you explore a classic web development example. You have an `addClickEventListener` function that takes in a listener function and adds it to the document:

```
const addClickEventListener = (listener) => {
  // red squiggly line under `listener`

  document.addEventListener("click", listener);
};

addClickEventListener(() => {
  console.log("Clicked!");
});
```

Currently there is an error under `listener` because it doesn't have a type signature. You're also *not* getting an error when you pass an incorrect value to `addClickEventListener`:

```
addClickEventListener(
  // @ts-expect-error // red squiggly line under @ts-expect-error
  "abc",
);
```

This is triggering your `@ts-expect-error` directive.

How should `addClickEventListener` be typed so that each error is resolved? See *https://totalts.link/essentials-4-18/*.

### Solution

Let's start by annotating the `listener` parameter to be a function. For now, you'll specify that it returns a string:

```
const addClickEventListener = (listener: () => string) => {
  document.addEventListener("click", listener);
};
```

The problem is that you now have an error when you call `addClickEventListener` with a function that returns nothing:

```
addClickEventListener(() => {
  // red squiggly line under `() => {`

  console.log("Clicked!");
});
```

When you hover over the error, you see the following message:

```
Argument of type '() => void' is not assignable to parameter of type '() => string'.
Type 'void' is not assignable to type 'string'.
```

The error message tells you that the `listener` function is returning `void`, which is not assignable to `string`. This suggests that instead of typing the `listener` parameter as a function that returns a string, you should type it as a function that returns `void`:

```
const addClickEventListener = (listener: () => void) => {
  document.addEventListener("click", listener);
};
```

This is a great way to tell TypeScript that you don't care about the return value of the `listener` function.

## Exercise 4-19: void vs. undefined

Let's say you've got a function that accepts a callback and calls it. The callback doesn't return anything, so you've typed it as `() => undefined`:

```
const acceptsCallback = (callback: () => undefined) => {
  callback();
};
```

But you're getting an error when you try to pass in `returnString`, a function that *does* return something:

```
const returnString = () => {
  return "Hello!";
};

// Argument of type '() => string' is not
// assignable to parameter of type '() => undefined'.
acceptsCallback(returnString); // red squiggly line under `returnString`
```

Why is this happening? Can you alter the type of `acceptsCallback` to fix this error?

See *https://totalts.link/essentials-4-19/*.

### Solution

The solution is to change the type of `callback` to `() => void`:

```
const acceptsCallback = (callback: () => void) => {
  callback();
};
```

Now you can pass in `returnString` without any issues. This is because `returnString` returns a `string`, and `void` tells TypeScript to ignore the return

value when comparing them. If you really don't care about the result of a function, type it as `() => void`.

## Exercise 4-20: Async Functions

This `fetchData` function awaits the `response` from a call to `fetch` and then gets the `data` by calling `response.json()`:

```
async function fetchData() {
  const response = await fetch("https://api.example.com/data");

  const data = await response.json();

  return data;
}
```

It's worth noting a couple of things here. Hovering over `response`, you can see that it has a type of `Response`, which is a globally available type:

```
// hovering over response shows:
const response: Response;
```

When hovering over `response.json()`, you can see that it returns a `Promise<any>`:

```
// hovering over response.json() shows:
const response.json(): Promise<any>
```

If you were to remove the `await` keyword from the call to `fetch`, the return type would also become `Promise<any>`:

```
const response = fetch("https://api.example.com/data");

// hovering over response shows:
const response: Promise<any>;
```

Consider this `example` and its test:

```
typescript
const example = async () => {
  const data = await fetchData();

  type test = Expect<Equal<typeof data, number>>;
};
```

The test is currently failing because `data` is typed as `any` instead of `number`.

How can you type `data` as a number without changing the calls to `fetch` or `response.json()`? There are two possible solutions.

See *https://totalts.link/essentials-4-20/*.

### Solutions

You might be tempted to try passing a type argument to `fetch`, similar to how you would with `Map` or `Set`. However, hovering over `fetch`, you can see that it doesn't accept type arguments:

```
// this won't work!

const response = fetch<number>("https://api.example.com/data");
// red squiggly line under number

// hovering over fetch shows:

function fetch(
  input: RequestInfo | URL,
  init?: RequestInit | undefined,
): Promise;
```

You also can't add a type annotation to `response.json()` because it doesn't accept type arguments either:

```
// this won't work!

const data: number = await response.json<number>();
// red squiggly line under number

// hovering over number shows:

Expected 0 type arguments, but got 1.
```

One thing that will work is to specify that `data` is a `number`:

```
const data: number = await response.json();
```

It works because `data` was `any` before, and `await response.json()` returns `any`. So, now you're putting `any` into a slot that requires a `number`.

However, the best way to solve this problem is to add a return type to the function. In this case, it should be a `number`:

```
// red squiggly line under number
async function fetchData(): number {
  // function body
}
```

Now `data` is typed as a `number`, except you have an error under your return type annotation:

```
The return type of an async function or method must be the global Promise type.
Did you mean to write 'Promise'?
```

Therefore, you should change the return type to `Promise<number>`:

```
async function fetchData(): Promise<number> {
  const response = await fetch("https://api.example.com/data");

  const data = await response.json();

  return data;
}
```

By wrapping the `number` in `Promise<>`, you make sure that the `data` is awaited before the type is figured out.

## Summary

This chapter introduced the foundational concepts of TypeScript's type system, focusing on type annotations and inference. You learned when to explicitly annotate your code and when TypeScript can figure out types automatically.

TypeScript provides basic types like `string`, `number`, `boolean`, `symbol`, `bigint`, `null`, and `undefined` that correspond to JavaScript's primitive values. These form the building blocks for more complex type annotations.

Function parameters always require type annotations since TypeScript can't infer what types they should accept. Variable annotations are often optional because TypeScript can infer types from assigned values, but function parameters need explicit typing.

The `any` type disables TypeScript's type checking entirely, allowing anything to be assigned or accessed without errors. While useful as an escape hatch, overusing `any` defeats the purpose of using TypeScript.

Object literal types use curly bracket syntax to describe object shapes, with optional properties marked using the `?` operator. Type aliases created with the `type` keyword let you define reusable types and share them across modules.

Arrays can be typed using square bracket syntax, like `string[]`, or using the generic `Array<string>` syntax. Tuples specify fixed-length arrays where each position has its own type, which is useful for grouping related data without creating new types.

Some functions, like `Set` and `Map`, can receive type arguments using angle brackets, while others, like `JSON.parse`, cannot. You can check by hovering over functions to see if they accept type parameters.

Function types describe the shape of functions themselves, including parameter and return types. Optional parameters use `?`, default parameters use `=`, and rest parameters use the spread syntax with array types.

The `void` type indicates functions that don't return meaningful values, while async functions must return `Promise<T>` where `T` is the resolved value type. These concepts form the essential foundation for working effectively with TypeScript's type system.