2
Introducing TypeScript

In this chapter, we will start by understanding what TypeScript is and how it provides a much richer type system on top of JavaScript. We will learn about the basic types in TypeScript, such as numbers and strings, and then learn how to create our own types to represent objects and arrays using different TypeScript features. Finally, we will finish the chapter by understanding the TypeScript compiler and its key options in a React app.

By the end of the chapter, you'll be ready to learn how to use TypeScript to build frontends with React.

In this chapter, we'll cover the following topics:

- Understanding the benefits of TypeScript
- Understanding JavaScript types
- Using basic TypeScript types
- Creating TypeScript types
- Using the TypeScript compiler

Technical requirements

We will use the following technologies in this chapter:

- **Browser**: A modern browser such as Google Chrome.
- **TypeScript Playground**: This is a website at `https://www.typescriptlang.org/play/` that allows you to play around with and understand the features of TypeScript without installing it.
- **CodeSandbox**: We'll briefly use this online tool to explore JavaScript's type system. This can be found at `https://codesandbox.io/`.

- **Visual Studio Code**: We'll need an editor to experience TypeScript's benefits and explore the TypeScript compiler. This one can be installed from `https://code.visualstudio.com/`. Other editors that could be used can be found at `https://github.com/Microsoft/TypeScript/wiki/TypeScript-Editor-Support`.
- **Node.js** and **npm**: TypeScript is dependent on these pieces of software. You can install them from `https://nodejs.org/en/download/`.

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter2`.

Understanding the benefits of TypeScript

In this section, we will start by understanding what TypeScript is, how it relates to JavaScript, and how TypeScript enables teams to be more productive.

Understanding TypeScript

TypeScript was first released in 2012 and is still being developed, with new releases happening every few months. But what is TypeScript, and what are its benefits?

TypeScript is often referred to as a superset or extension of JavaScript because any feature in JavaScript is available in TypeScript. Unlike JavaScript, TypeScript can't be executed directly in a browser – it must be transpiled into JavaScript first.

> **Note**
>
> It is worth being aware that a proposal is being considered that *would* allow TypeScript to be executed directly in a browser without transpilation. See the following link for more information: `https://github.com/tc39/proposal-type-annotations`.

TypeScript adds a rich type system to JavaScript. It is generally used with frontend frameworks such as Angular, Vue, and React. TypeScript can also be used to build a backend with Node.js. This demonstrates how flexible TypeScript's type system is.

When a JavaScript codebase grows, it can become hard to read and maintain. TypeScript's type system solves this problem. TypeScript uses the type system to allow code editors to catch type errors as developers write problematic code. Code editors also use the type system to provide productivity features such as robust code navigation and code refactoring.

Next, we will step through an example of how TypeScript catches an error that JavaScript can't.

Catching type errors early

The type information helps the TypeScript compiler catch type errors. In code editors such as Visual Studio Code, a type error is underlined in red immediately after the developer has made a type mistake. Carry out the following steps to experience an example of TypeScript catching a type error:

1. Open Visual Studio Code in a folder of your choice.
2. Create a new file called `calculateTotalPrice.js` by choosing the **New File** option in the **EXPLORER** panel.

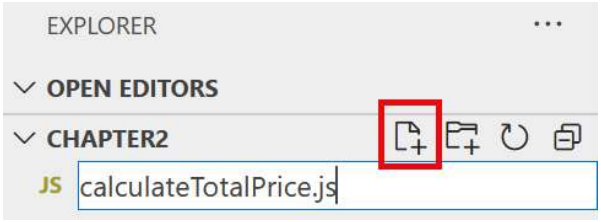

Figure 2.1 – Creating a new file in Visual Studio Code

3. Enter the following code into the file:

```
function calculateTotalPriceJS(product, quantity,
discount) {
  const priceWithoutDiscount = product.price * quantity;
  const discountAmount = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}
```

Remember that the code snippets are available online to copy. The link to the previous snippet can be found at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter2/Section1-Understanding-TypeScript/calculateTotalPrice.js`.

There is a bug in the code that might be difficult to spot, and the error won't be highlighted by Visual Studio Code.

4. Now create a copy of the file but with a `.ts` extension instead of `.js`. A file can be copied by right-clicking on the file in the **EXPLORER** panel and selecting the **Copy** option. Then right-click the **EXPLORER** panel again and select the **Paste** option to create the copied file.

> **Note**
>
> A `.ts` file extension denotes a TypeScript file. This means a TypeScript compiler will perform type checking on this file.

5. In the `calculateTotalPrice.ts` file, remove the `JS` from the end of the function name and make the following highlighted updates to the code:

```
function calculateTotalPrice(
  product: { name: string; unitPrice: number },
  quantity: number,
  discount: number
) {
  const priceWithoutDiscount = product.price * quantity;
  const discountAmount = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}
```

Here, we have added TypeScript **type annotations** to the `function` parameters. We will learn about type annotations in detail in the next section.

The key point is that the type error is now highlighted by a red squiggly underline:

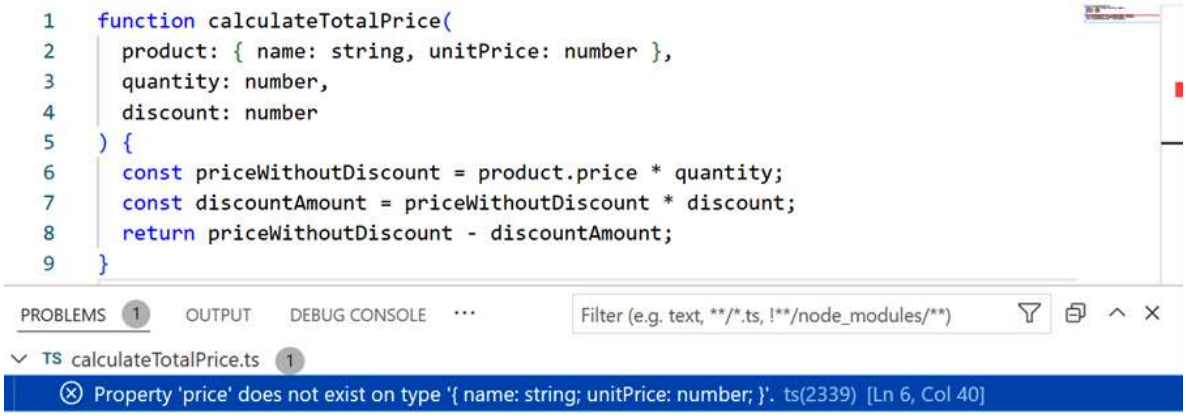

Figure 2.2 – Highlighted type error

The bug is that the function references a `price` property in the product object that doesn't exist. The property that should be referenced is `unitPrice`.

Catching these problems early in the development process increases the team's throughput and is one less thing for quality assurance to catch. It could be worse – the bug could have gotten into the live app and given users a bad experience.

Keep these files open in Visual Studio Code because we will run through an example of TypeScript improving the developer experience next.

Improving developer experience and productivity with IntelliSense

IntelliSense is a feature in code editors that gives useful information about elements of code and allows code to be quickly completed. For example, IntelliSense can provide the list of properties available in an object.

Carry out the following steps to experience how TypeScript works better with IntelliSense than JavaScript and how this positively impacts productivity. As part of this exercise, we will fix the price bug from the previous section:

1. Open `calculateTotalPrice.js` and on line 2, where `product.price` is referenced, remove `price`. Then, with the cursor after the dot (.), click *Ctrl* + spacebar. This opens Visual Studio Code's IntelliSense:

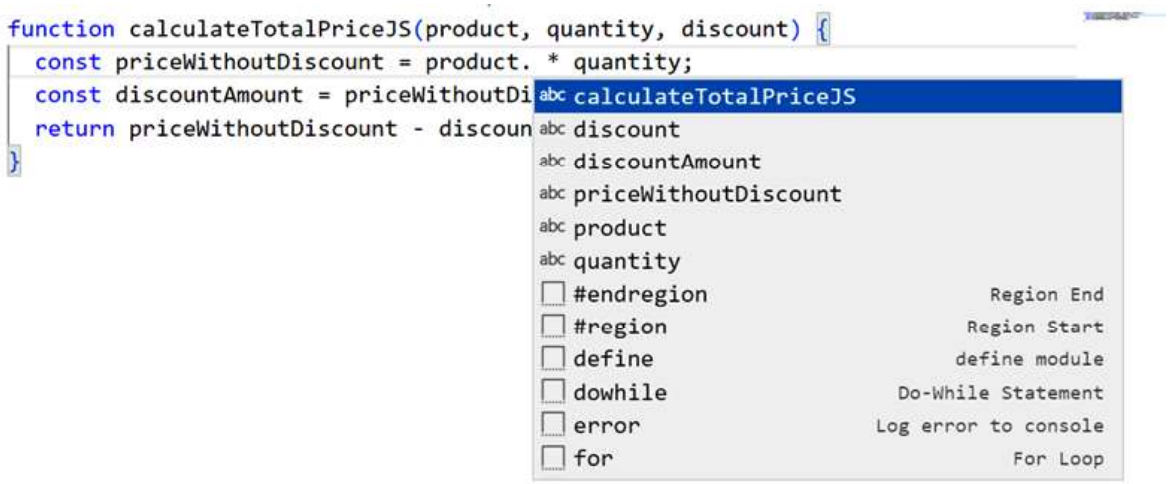

Figure 2.3 – IntelliSense in a JavaScript file

Visual Studio Code can only guess the potential property name, so it lists variable names and function names it has seen in the file. Unfortunately, IntelliSense doesn't help in this case because the correct property name, `unitPrice`, is not listed.

2. Now open `calculateTotalPrice.ts`, remove `price` from `product.price`, and press *Ctrl* + spacebar to open IntelliSense again:

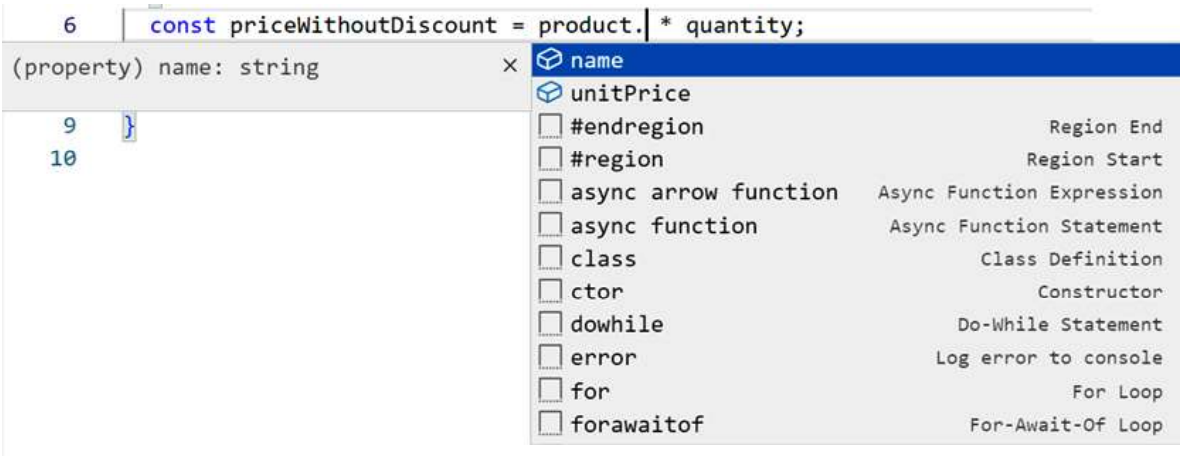

Figure 2.4 – IntelliSense in a TypeScript file

   This time, Visual Studio Code lists the correct properties.

3. Select **unitPrice** from IntelliSense to resolve the type error.

IntelliSense is just one tool that TypeScript provides. It can also provide robust refactoring features, such as renaming React components, and helps with accurate code navigation, such as going to a function definition.

To recap what we learned in this section:

- TypeScript's type-checking feature helps catch problems earlier in the development process
- TypeScript enables code editors to offer productivity features such as IntelliSense
- These advantages provide significant benefits when working in larger codebases

Next, we will learn about the type system in JavaScript. This will further underline the need for TypeScript in a large codebase.

Understanding JavaScript types

Before understanding the type system in TypeScript, let's briefly explore the type system in JavaScript. To do this, open the CodeSandbox at `https://codesandbox.io/` and carry out the following steps:

1. Create a new plain JavaScript project by choosing the **Vanilla** option.

2. Open `index.js`, remove its content, and replace it with the following code:

```
let firstName = "Fred"
console.log("firstName", firstName, typeof firstName);
let score = 9
console.log("score", score, typeof score);
let date = new Date(2022, 10, 1);
console.log("date", date, typeof date);
```

The code assigns three variables to various values. The code also outputs the variable values to the console, along with their JavaScript type.

Here's the console output:

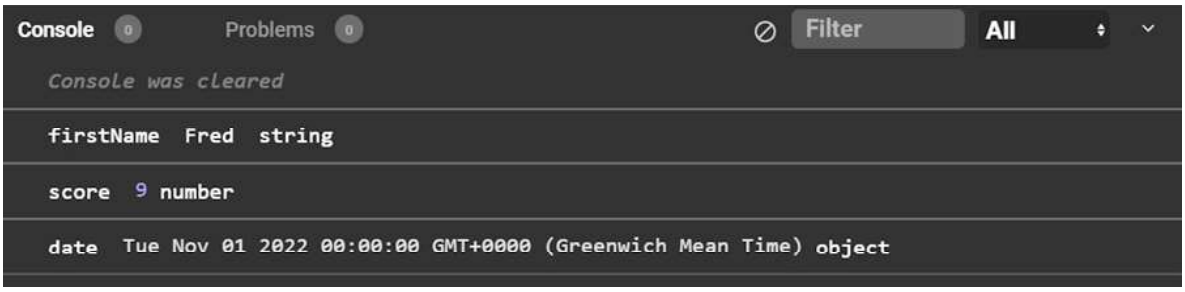

Figure 2.5 – Some JavaScript types

It isn't surprising that `firstName` is a string and `score` is a number. However, it is a little surprising that `date` is an object rather than something more specific such as a date.

3. Let's add another couple of lines of code after the existing code:

```
score = "ten"
console.log("score", score, typeof score);
```

Again, the console output is a little surprising:

Figure 2.6 – Variable changing type

The `score` variable has changed from a `number` type to a `string` type! This is because JavaScript is loosely typed.

A key point is that JavaScript only has a minimal set of types, such as `string`, `number`, and `boolean`. It is worth noting that all of the JavaScript types are available in TypeScript because Typescript is a superset of Javascript.

Also, JavaScript allows a variable to change its type – meaning that the JavaScript engine won't throw an error if a variable is changed to a completely different type. This loose typing makes it impossible for code editors to catch type errors.

> **Note**
>
> For more information on JavaScript types, see `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures`.

Now that we understand the limitations of the type system in JavaScript, we will learn about TypeScript's type system, starting with basic types.

Using basic TypeScript types

In this section, we'll start by understanding how TypeScript types can be declared and how they are inferred from assigned values. We will then learn the basic types commonly used in TypeScript that aren't available in JavaScript and understand helpful use cases.

Using type annotations

TypeScript type annotations enable variables to be declared with specific types. These allow the TypeScript compiler to check that the code adheres to these types. In short, type annotations allow TypeScript to catch bugs where our code uses the wrong type much earlier than we would if we were writing our code in JavaScript.

Open the TypeScript Playground at `https://www.typescriptlang.org/play` and carry out the following steps to explore type annotations:

1. Remove any existing code in the left-hand pane and enter the following variable declaration:

   ```
   let unitPrice: number;
   ```

   The type annotation comes after the variable declaration. It starts with a colon followed by the type we want to assign to the variable. In this case, `unitPrice` is going to be a `number` type. Remember that `number` is a type in JavaScript, which means that it is available for us to use in TypeScript too.

   The transpiled JavaScript appears on the right-hand side as follows:

   ```
   let unitPrice;
   ```

However, notice that the type annotation has disappeared. This is because type annotations don't exist in JavaScript.

> **Note**
>
> You may also see `"use strict";` at the top of the transpiled JavaScript. This means that the JavaScript will be executed in JavaScript strict mode, which will pick up more coding mistakes. For more information on JavaScript strict mode, see `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode`.

2. Add a second line to the program:

```
unitPrice = "Table";
```

Notice that a red line appears under `unitPrice` on this line. If you hover over the underlined `unitPrice`, a type error is described:

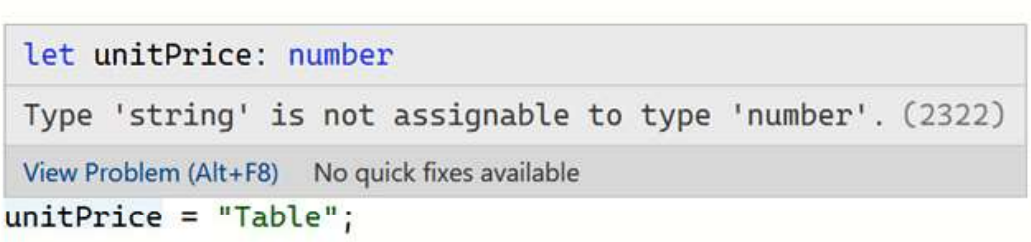

Figure 2.7 – A type error being caught

3. You can also add type annotations to function parameters and a function's return value using the same syntax as annotating a variable. As an example, enter the following function in the TypeScript Playground:

```
function getTotal(
  unitPrice: number,
  quantity: number,
  discount: number
): number {
  const priceWithoutDiscount = unitPrice * quantity;
  const discountAmount = priceWithoutDiscount * discount;
  return priceWithoutDiscount - discountAmount;
}
```

We've declared the `unitPrice`, `quantity`, and `discount` parameters, all with a `number` type. The return type annotation comes after the function's parentheses, which is also a `number` type in the preceding example.

> **Note**
>
> We have used both `const` and `let` to declare variables in different examples. `let` allows the variable to change the value after the declaration, whereas `const` variables can't change. In the preceding function, `priceWithoutDiscount` and `discountAmount` never change the value after the initial assignment, so we have used `const`.

4. Add another line of code to call `getTotal` with an incorrect type for `quantity`. Assign the result of the call to `getTotal` to a variable with an incorrect type:

```
let total: string = getTotal(500, "one", 0.1);
```

Both errors are immediately detected and highlighted:

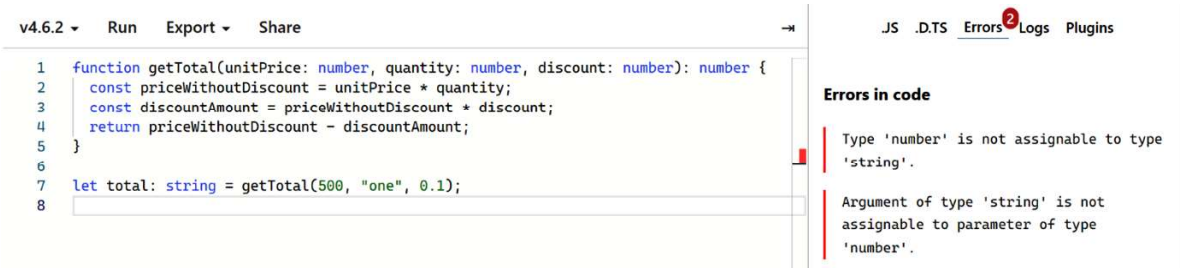

Figure 2.8 – Both type errors being caught

This strong type checking is something that we don't get in JavaScript, and it is very useful in large codebases because it helps us immediately detect type errors.

Next, we will learn how TypeScript doesn't always need type annotations in order to type-check code.

Using type inference

Type annotations are really valuable, but they require additional code to be written. This extra code takes time to write. Luckily, TypeScript's powerful **type inference** system means type annotations don't need to be specified all the time. TypeScript infers the type of a variable when it is assigned a value from that value.

Explore type inference by carrying out the following steps in the TypeScript Playground:

1. First, remove any previous code and then add the following line:

```
let flag = false;
```

2. Hover over the `flag` variable. A tooltip will appear showing the type that `flag` has been inferred to:

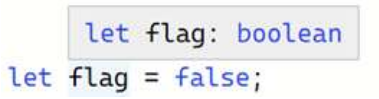

Figure 2.9 – Hovering over a variable reveals its type

3. Add another line beneath this to incorrectly set `flag` to an invalid value:

```
flag = "table";
```

A type error is immediately caught, just like when we used a type annotation to assign a type to a variable.

Type inference is an excellent feature of TypeScript and prevents code bloat that lots of type annotations would bring. Therefore, it is common practice to use type inference and only revert to using type annotations where inference isn't possible.

Next, we will look at the `Date` type in TypeScript.

Using the Date type

We are already aware that a `Date` type doesn't exist in JavaScript, but luckily, a `Date` type does exist in TypeScript. The TypeScript `Date` type is a representation of the JavaScript `Date` object.

> **Note**
> See the following link for more information on the JavaScript `Date` object: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date`.

To explore the TypeScript `Date` type, carry out the following steps in the TypeScript Playground:

1. First, remove any previous code and then add the following lines:

```
let today: Date;
today = new Date();
```

A `today` variable is declared that is assigned a `Date` type and set to today's date.

2. Refactor these two lines into the following single line that uses type inference rather than a type annotation:

```
let today = new Date();
```

3. Check that `today` has been assigned the `Date` type by hovering over it and checking the tooltip:

Figure 2.10 – Confirmation that today has inferred the Date type

4. Now, check IntelliSense is working by adding `today.` on a new line:

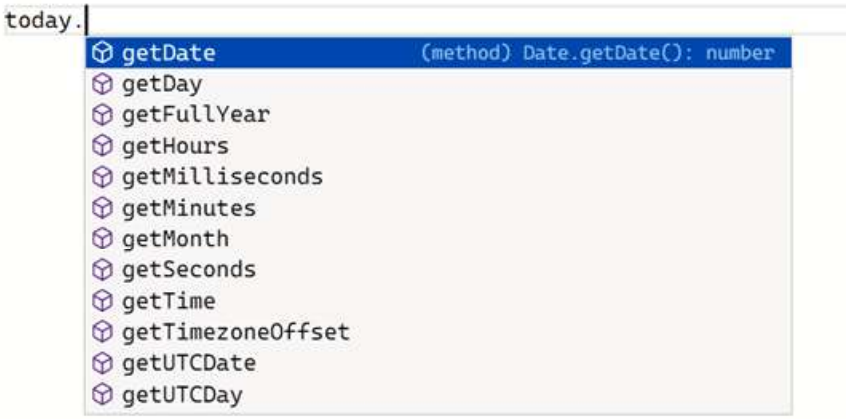

Figure 2.11 – IntelliSense working nicely on a date

5. Remove this line and add a slightly different line of code:

```
today.addMonths(2);
```

An `addMonths` function doesn't exist in the `Date` object, so a type error is raised:

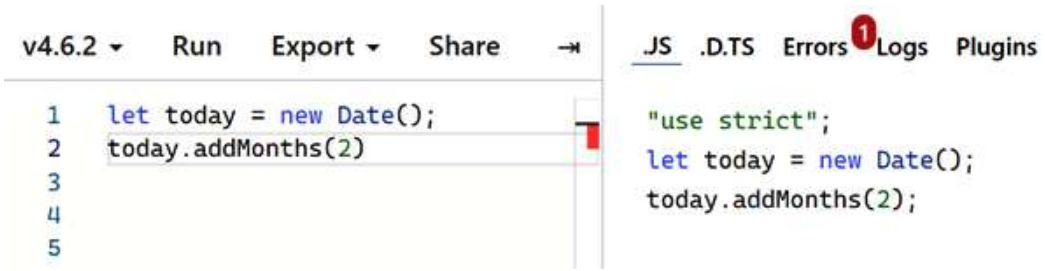

Figure 2.12 – Type error caught on a date

In summary, the `Date` type has all the features we expect – inference, IntelliSense, and type checking – which are really useful when working with dates.

Next, we will learn about an escape hatch in TypeScript's type system.

Using the any type

What if we declare a variable with no type annotation and no value? What will TypeScript infer as the type? Let's find out by entering the following code in the TypeScript Playground:

```
let flag;
```

Now, hover the mouse over `flag`:

Figure 2.13 – Variable given the any type

So, TypeScript gives a variable with no type annotation and no immediately assigned value the `any` type. It is a way of opting out of performing type checking on a particular variable and is commonly used for dynamic content or values from third-party libraries. However, TypeScript's increasingly powerful type system means that we need to use `any` less often these days.

Instead, there is a better alternative: the `unknown` type.

Using the unknown type

`unknown` is a type we can use when we are unsure of the type but want to interact with it in a strongly-typed manner. Carry out the following steps to explore how this is a better alternative to the `any` type:

1. In the TypeScript Playground, remove any previous code, and enter the following:

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data) => {
    console.log("firstName", data.firstName);
  });
```

   The code fetches a Star Wars character from a web API. No type errors are raised, so the code appears okay.

2. Now click on the **Run** option to execute the code:

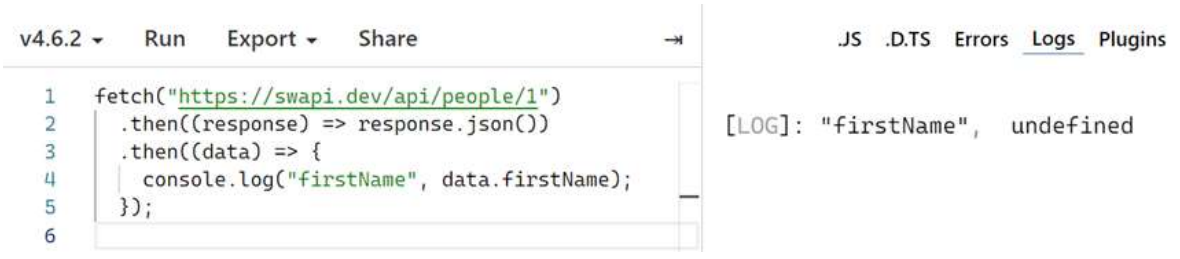

Figure 2.14 – firstName property has an undefined value

The `firstName` property doesn't appear to be in the fetched data because it is `undefined` when it is output to the console.

Why wasn't a type error raised on line 4 where `firstName` was referenced? Well, `data` is of the type `any`, which means no type checking will occur on it. You can hover over `data` to confirm that it has been given the `any` type.

3. Give `data` the `unknown` type annotation:

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data: unknown) => {
    console.log("firstName", data.firstName);
  });
```

A type error is now raised where `firstName` is referenced:

Figure 2.15 – Type error on unknown data parameter

The `unknown` type is the opposite of the `any` type, as it contains nothing within its type. A type that doesn't contain anything may seem useless. However, a variable's type can be widened if cheks are made to allow TypeScript to widen it.

4. Before we give TypeScript information to widen `data`, change the property referenced within it from `firstName` to `name`:

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
  .then((data: unknown) => {
    console.log("name", data.name);
  });
```

`name` is a valid property, but a type error is still occurring. This is because `data` is still `unknown`.

5. Now make the highlighted changes to the code to widen the `data` type:

```
fetch("https://swapi.dev/api/people/1")
  .then((response) => response.json())
```

```
  .then((data: unknown) => {
    if (isCharacter(data)) {
      console.log("name", data.name);
    }
  });

function isCharacter(
  character: any
): character is { name: string } {
  return "name" in character;
}
```

The code snippet can be copied from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter2/Section2-Basic-types/Using-the-unknown-type/code.ts`.

The `if` statement uses a function called `isCharacter` to verify that a `name` property is contained within the object. The result of this call is `true` in this example, so the logic will flow into the `if` branch.

Notice the return type of `isCharacter`, which is:

```
character is { name: string }
```

This is a **type predicate**. TypeScript will narrow or widen the type of `character` to `{ name: string }` if the function returns `true`. The type predicate is `true` in this example, so `character` is widened to an object with a `name` string property.

6. Hover over the `data` variable on each line where it is referenced. `data` starts off with the `unknown` type where it is assigned with a type annotation. Then, it is widened to `{name: string}` inside the `if` branch:

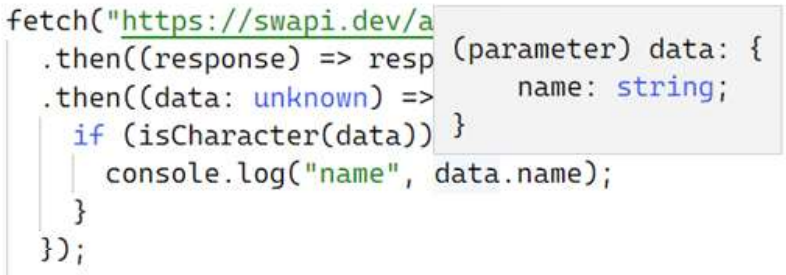

Figure 2.16 – Widened type given to data

Notice that the type error has also disappeared. Nice!

7. Next, run the code. You will see `Luke Skywalker` output to the console.

In summary, the `unknown` type is an excellent choice for data whose type you are unsure about. However, you can't interact with `unknown` variables – the variable must be widened to a different type before any interaction.

Next, we will learn about a type used for a function not returning a value.

Using the void type

The `void` type is used to represent a function's return type where the function doesn't return a value.

As an example, enter the following function in the TypeScript Playground:

```
function logText(text: string) {
  console.log(text);
}
```

Hovering over the function name confirms that the function return type is given a `void` type.

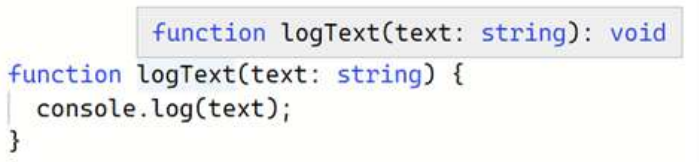

Figure 2.17 – Return type confirmed as void

You may think that you could use `undefined` as the return type for the preceding example:

```
function logText(text: string): undefined {
  console.log(text);
}
```

However, this raises a type error because a return type of `undefined` means that the function is expected to return a value (of type `undefined`). The example function doesn't return any value, so the return type is `void`.

In summary, `void` is a special type for a function's return type where the function doesn't have a return statement.

Next, we will learn about the `never` type.

Using the never type

The `never` type represents something that will never occur and is typically used to specify unreachable code areas. Let's explore an example in the TypeScript Playground:

1. Remove any existing code and enter the following code:

```
function foreverTask(taskName: string): never {
  while (true) {
    console.log(`Doing ${taskName} over and over again
      ...`);
  }
}
```

   The function invokes an infinite loop, meaning the function is never exited. So, we have given the function a return type annotation of `never` because we don't expect the function to be exited. This is different from `void` because `void` means it *will* exit, but with no value.

> **Note**
>
> We used a JavaScript template literal to construct the string to output to the console in the preceding example. Template literals are enclosed by backticks (`` ` ` ``) and can include a JavaScript expression in curly braces prefixed with a dollar sign (`${expression}`). Template literals are great when we need to merge static text with variables. See this link for more information on template literals: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals`.

2. Change the `foreverTask` function to break out of the loop:

```
function foreverTask(taskName: string): never {
  while (true) {
    console.log(`Doing ${taskName} over and over again
      ...`);
    break;
  }
}
```

TypeScript quite rightly complains:

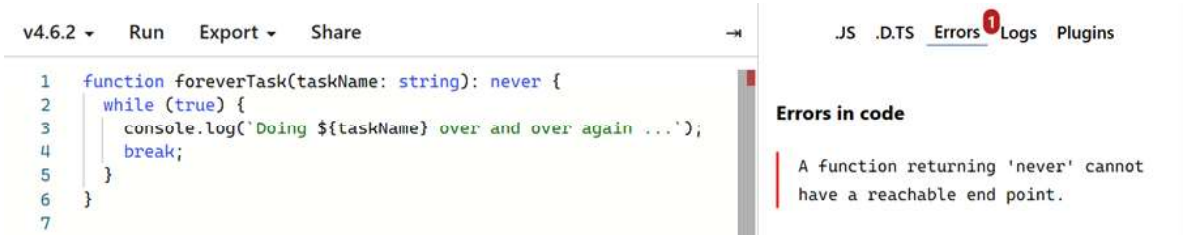

Figure 2.18 – Type error on the never return type

3. Remove the `break` statement and remove the `never` return type annotation:

```
function foreverTask(taskName: string) {
  while (true) {
    console.log(`Doing ${taskName} over and over again
...`);
  }
}
```

4. Hover over the `foreverTask` function name with your mouse. We can see that TypeScript has inferred the return type as `void`:

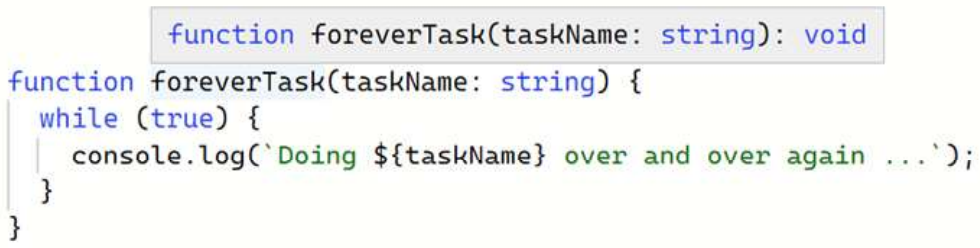

Figure 2.19 – Return type inferred as void

So, TypeScript is unable to infer the `never` type in this case. Instead, it infers the return type as `void`, which means the function will exit with no value, which isn't the case in this example. This is a reminder to always check the inferred type and resort to using a type annotation where appropriate.

In summary, the `never` type is used in places where code never reaches.

Next up, let's cover arrays.

Using arrays

Arrays are structures that TypeScript inherits from JavaScript. We add type annotations to arrays as usual, but with square brackets `[]` at the end to denote that this is an array type.

Let's explore an example in the TypeScript Playground:

1. Remove any existing code, and enter the following:

```
const numbers: number[] = [];
```

   Alternatively, the `Array` generic type syntax can be used:

```
const numbers: Array<number> = [];
```

   We will learn about generics in TypeScript in *Chapter 11*, *Reusable Components*.

2. Add `1` to the array by using the array's `push` function:

```
numbers.push(1);
```

3. Now add a string to the array:

```
numbers.push("two");
```

   A type error is raised as we would expect:

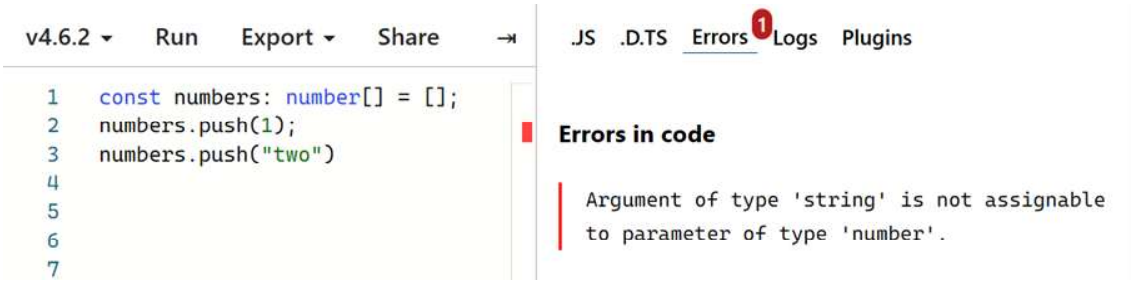

Figure 2.20 – Type error when adding a string type to a number array

4. Now replace all the code with the following:

```
const numbers = [1, 2, 3];
```

5. Hover over `numbers` to verify that TypeScript has inferred its type to be `number[]`.

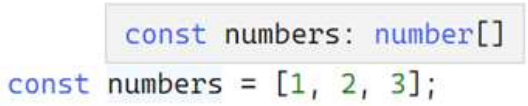

Figure 2.21 – Array type inference

   Excellent – we can see that TypeScript's type inference works with arrays!

Arrays are one of the most common types used to structure data. In the preceding examples, we've only used an array with elements having a `number` type, but any type can be used for elements, including objects, which have their own properties.

Here's a recap of all the basic types we have learned in this section:

- TypeScript adds many useful types to JavaScripts types, such as `Date`, and is capable of representing arrays.
- TypeScript can infer a variable's type from its assigned value. A type annotation can be used where type inference doesn't give the desired type.
- No type checking occurs on variables with the `any` type, so this type should be avoided.
- The `unknown` type is a strongly-typed alternative to `any`, but `unknown` variables must be widened to be interacted with.
- `void` is a return type for a function that doesn't return a value.
- The `never` type can be used to mark unreachable areas of code.
- Array types can be defined using square brackets after the array item type.

In the next section, we will learn how to create our own types.

Creating TypeScript types

The last section showed that TypeScript has a great set of standard types. In this section, we will learn how to create our own types. We will start by learning three different methods for creating object types. We will then learn about strongly-typing JavaScript classes. Lastly, we will learn two different methods for creating types for variables that hold a range of values.

Using object types

Objects are very common in JavaScript programs, so learning how to represent them in TypeScript is really important. In fact, we have already used an object type earlier in this chapter for the `product` parameter in the `calculateTotalPrice` function. Here is a reminder of the `product` parameter's type annotation:

```
function calculateTotalPrice(
  product: { name: string; unitPrice: number },
  ...
) {
  ...
}
```

An object type in TypeScript is represented a bit like a JavaScript object literal. However, instead of property values, property types are specified instead. Properties in the object definitions can be separated by semicolons or commas, but using a semicolon is common practice.

Clear any existing code in the TypeScript Playground and follow this example to explore object types:

1. Enter the following variable assignment to an object:

```
let table = {name: "Table", unitPrice: 450};
```

   If you hover over the `table` variable, you'll see it is inferred to be the following type:

```
{
  name: string;
  unitPrice: number;
}
```

   So, type inference works nicely for objects.

2. Now, on the next line, try to set a `discount` property to `10`:

```
table.discount = 10;
```

   A `discount` property doesn't exist in the type, though – only the `name` and `unitPrice` properties exist. So, a type error occurs.

3. Let's say we want to represent a `product` object containing the `name` and `unitPrice` properties, but we want `unitPrice` to be optional. Remove the existing code and replace it with the following:

```
const table: { name: string; unitPrice: number } = {
  name: "Table",
};
```

4. This raises a type error because `unitPrice` is a required property in the type annotation. We can use a `?` symbol as follows to make this optional rather than required:

```
const table: { name: string; unitPrice?: number } = {
  name: "Table",
};
```

   The type error disappears.

> **Note**
>
> The `?` symbol can be used in functions for optional parameters. For example, `myFunction(requiredParam: string, optionalParam?: string)`.

Now, let's learn a way to streamline object type definitions.

Creating type aliases

The type annotation we used in the last example was quite lengthy and would be longer for more complex object structures. Also, having to write the same object structure to assign to different variables is a little frustrating:

```
const table: { name: string; unitPrice?: number } = ...;
const chair: { name: string; unitPrice?: number } = ...;
```

Type aliases solve these problems. As the name suggests, a type alias refers to another type, and the syntax is as follows:

```
type YourTypeAliasName = AnExistingType;
```

Open the TypeScript Playground and follow along to explore type aliases:

1. Start by creating a type alias for the product object structure we used in the last example:

   ```
   type Product = { name: string; unitPrice?: number };
   ```

2. Now assign two variables to this `Product` type:

   ```
   let table: Product = { name: "Table" };
   let chair: Product = { name: "Chair", unitPrice: 40 };
   ```

   That's much cleaner!

3. A type alias can extend another object using the `&` symbol. Create a second type for a discounted product by adding the following type alias:

   ```
   type DiscountedProduct = Product & { discount: number };
   ```

   `DiscountedProduct` represents an object containing `name`, `unitPrice` (optional), and `discount` properties.

> **Note**
> A type that extends another using the `&` symbol is referred to as an **intersection type**.

4. Add the following variable with the `DiscountedProduct` type as follows:

   ```
   let chairOnSale: DiscountedProduct = {
     name: "Chair on Sale",
     unitPrice: 30,
   ```

```
  discount: 5,
};
```

5. A type alias can also be used to represent a function. Add the following type alias to represent a function:

```
type Purchase = (quantity: number) => void;
```

The preceding type represents a function containing a `number` parameter and doesn't return anything.

6. Use the `Purchase` type to create a `purchase` function property in the `Product` type as follows:

```
type Purchase = (quantity: number) => void;
type Product = {
  name: string;
  unitPrice?: number;
  purchase: Purchase;
};
```

Type errors will be raised on the `table`, `chair`, and `chairOnSale` variable declarations because the `purchase` function property is required.

7. Add a `purchase` function property to the `table` variable declarations as follows:

```
let table: Product = {
  name: "Table",
  purchase: (quantity) =>
    console.log(`Purchased ${quantity} tables`),
};
table.purchase(4);
```

The type error is resolved on the `table` variable declaration.

8. A purchase property could be added in a similar way to the `chair` and `chairOnSale` variable declarations to resolve their type errors. However, ignore these type errors for this exploration and move on to the next step.

9. Click the **Run** option to run the code that purchases four tables. **"Purchased 4 tables"** is output to the console.

In summary, type aliases allow existing types to be composed together and improve the readability and reusability of types. We will use type aliases extensively in this book.

Next, we will explore an alternative method of creating types. Leave the TypeScript Playground open with the code intact – we'll use this in the next section.

Creating interfaces

As we created in the last example with type aliases, object types can be created using TypeScript's **interface** syntax. An interface is created with the `interface` keyword, followed by its name, followed by the bits that make up the `interface` in curly brackets:

```
interface Product {
  ...
}
```

Go to the TypeScript Playground that contains the code from the type alias exploration, and follow along to explore interfaces:

1. Start by replacing the `Product` type alias with a `Product` interface as follows:

   ```
   interface Product {
     name: string;
     unitPrice?: number;
   }
   ```

   The `table` variable assignment has a type error because the `purchase` property doesn't exist yet – we'll add this in *step 4*. However, the `chair` variable assignment compiles without error.

2. An interface can extend another interface using the `extends` keyword. Replace the `DiscountedProduct` type alias with the following interface:

   ```
   interface DiscountedProduct extends Product {
     discount: number;
   }
   ```

   Notice that the `chairOnSale` variable assignment compiles without error.

3. An interface can also be used to represent a function. Add the following interface to represent a function, replacing the type alias version:

   ```
   interface Purchase {(quantity: number): void}
   ```

   The interface syntax for creating functions isn't as intuitive as using a type alias.

4. Add the `Purchase` interface to the `Product` interface as follows:

   ```
   interface Product {
     name: string;
   ```

```
  unitPrice?: number;
  purchase: Purchase;
}
```

The type error on the `table` variable declarations is resolved now, but type errors are raised on the `chair` and `chairOnSale` variable declarations.

5. Click the **Run** option to run the code that purchases four tables. **"Purchased 4 tables"** is output to the console.

In the preceding steps, we carried out the same tasks using an interface as we did using a type alias. So, the obvious question is, *when should I use a type alias instead of an interface and vice versa?* The capabilities of type aliases and interfaces for creating object types are very similar – so the simple answer is that it is down to preference for object types. Type aliases can create types that interfaces can't, though, such as union types, which we shall cover later in this chapter.

> **Note**
>
> See the following link for more information on the differences between type aliases and interfaces: `https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#differences-between-type-aliases-and-interfaces`.
>
> The rest of this book uses type aliases rather than interfaces to define types.

Next, we will learn how to use TypeScript with classes.

Creating classes

A **class** is a standard JavaScript feature that acts as a template for creating an object. Properties and methods defined in the class are automatically included in objects created from the class.

Open the TypeScript Playground, remove any existing code, and carry out the following steps to explore classes in TypeScript:

1. Add the following code to create a class to represent a product with properties for the name and unit price:

```
class Product {
  name;
  unitPrice;
}
```

If you hover over the `name` and `unitPrice` properties, you'll see that they have the `any` type. As we know, that means no type checking will occur on them.

2. Add the following type annotations to the properties:

```
class Product {
  name: string;
  unitPrice: number;
}
```

Unfortunately, TypeScript raises the following error:

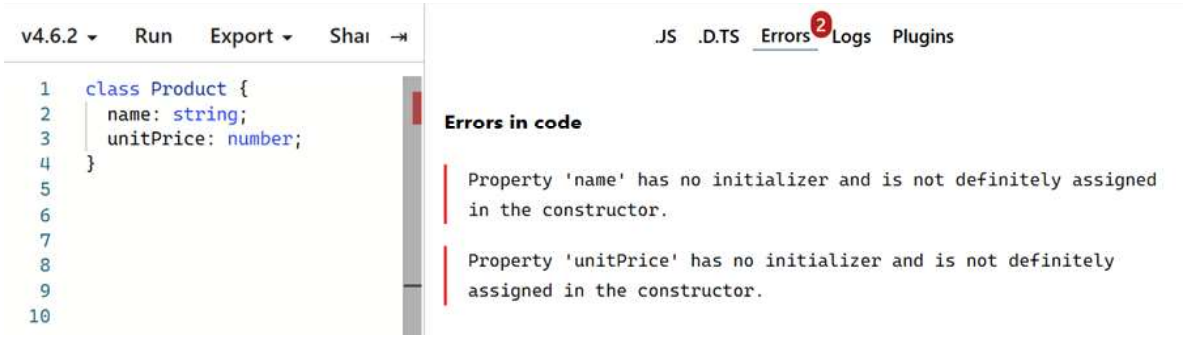

Figure 2.22 – Type errors on the class properties

The error is because when an instance of the class is created, those property values would be `undefined`, which isn't within the `string` or `number` types.

3. A solution would be to make the properties optional so that they can accept `undefined` as a value. Try this solution by adding a `?` symbol at the start of the type annotations:

```
class Product {
  name?: string;
  unitPrice?: number;
}
```

4. If we don't want the values to initially be `undefined`, we can assign initial values like so:

```
class Product {
  name = "";
  unitPrice = 0;
}
```

If you hover over the properties now, you will see that `name` has been inferred to be a `string` type, and `unitPrice` has been inferred to be a `number` type.

5. Another method of adding types to class properties is in a constructor. Remove the values assigned to the properties and add a constructor to the class as follows:

```
class Product {
  name;
  unitPrice;
  constructor(name: string, unitPrice: number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }
}
```

If you hover over the properties, you'll see that the correct types have been inferred.

6. In fact, the properties don't need to be defined if the constructor parameters are marked as `public`.

```
class Product {
  constructor(public name: string, public unitPrice:
    number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }
}
```

TypeScript will automatically create properties for constructor parameters that are marked as `public`.

7. Type annotations can be added to method parameters and return values, just like we have previously done for functions:

```
class Product {
  constructor(public name: string, public unitPrice:
    number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }
  getDiscountedPrice(discount: number): number {
    return this.unitPrice - discount;
  }
}
```

8. Now create an instance of the class and output its discounted price to the console:

```
const table = new Product("Table", 45);
console.log(table.getDiscountedPrice(5));
```

If you run the code, **40** is output to the console.

In summary, class properties can be given a type in a constructor or by assigning a default value. Class methods can be strongly-typed just like regular JavaScript functions.

> **Note**
>
> For more information on classes, see the following link: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`.

Next, we will learn how to create a type to represent a range of values.

Creating enumerations

Enumerations allow us to declare a meaningful set of friendly names that a variable can be set to. We use the `enum` keyword, followed by the name we want to give to it, and then its possible values in curly braces.

Let's explore an example in the TypeScript Playground:

1. Start by creating the enumeration for `Level` containing `Low`, `Medium`, and `High` values:

```
enum Level {
  Low,
  Medium,
  High
}
```

2. Now, create a `level` variable and assign it to the values `Low` and then `High` from the `Level` enumeration. Also, output the `level` values to the console:

```
let level = Level.Low;
console.log(level);
level = Level.High
console.log(level);
```

Notice that you get IntelliSense as you reference the enumeration.

3. Click the **Run** option to execute the code and observe the enumeration values:

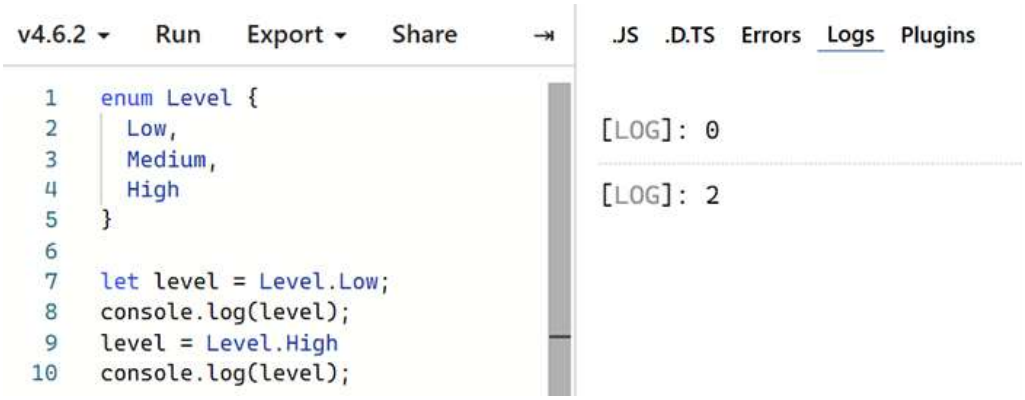

Figure 2.23 – Output of enumeration values

By default, enumerations are zero-based numbers (this means that the first enumeration value is 0, the next is 1, the next is 2, and so on). In the preceding example, `Level.Low` is `0`, `Level.Medium` is `1`, and `Level.High` is `2`.

4. Instead of the default values, custom values can be explicitly defined against each enumeration item after the equals (=) symbol. Explicitly set the values to be between `1` and `3`:

```
enum Level {
  Low = 1,
  Medium = 2,
  High = 3
}
```

You can rerun the code to verify this works.

5. Now, let's do something interesting. Assign `level` to a number greater than 3:

```
level = 10;
```

Notice that a type error *doesn't* occur. This is a little surprising – number-based enumerations aren't as type-safe as we would like.

6. Instead of using number enumeration values, let's try strings. Replace all of the current code with the following:

```
enum Level {
  Low = "L",
  Medium = "M",
  High = "H"
}
```

```
let level = Level.Low;
console.log(level);
level = Level.High
console.log(level);
```

If this code is run, we see **L** and **H** output to the console as expected.

7. Add another line that assigns `level` to the following strings:

```
level = "VH";
level = "M"
```

We immediately see type errors raised on these assignments:

Figure 2.24 – Confirmation that string enumerations are type-safe

In summary, enumerations are a way of representing a range of values with user-friendly names. They are zero-based numbers by default and not as type-safe as we would like. However, we can make enumerations string-based, which is more type-safe.

Next, we will learn about union types in TypeScript.

Creating union types

A **union type** is the mathematical union of multiple other types to create a new type. Like enumerations, union types can represent a range of values. As mentioned earlier, type aliases can be used to create union types.

An example of a union type is as follows:

```
type Level = "H" | "M" | "L";
```

This `Level` type is similar to the enumeration version of the `Level` type we created earlier. The difference is that the union type only contains values (`"H"`, `"M"`, `"L"`) rather than a name (`"High"`, `"Medium"`, `"Large"`) and a value.

Clear any existing code in the TypeScript Playground, and let's have a play with union types:

1. Start by creating a type to represent `"red"`, `"green"`, or `"blue"`:

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

   Note that this type is a union of strings, but a union type can consist of any types – even mixed types!

2. Create a variable with the `RGB` type and assign a valid value:

   ```
   let color: RGB = "red";
   ```

3. Now try assigning a value outside the type:

   ```
   color = "yellow";
   ```

   A type error occurs, as expected:

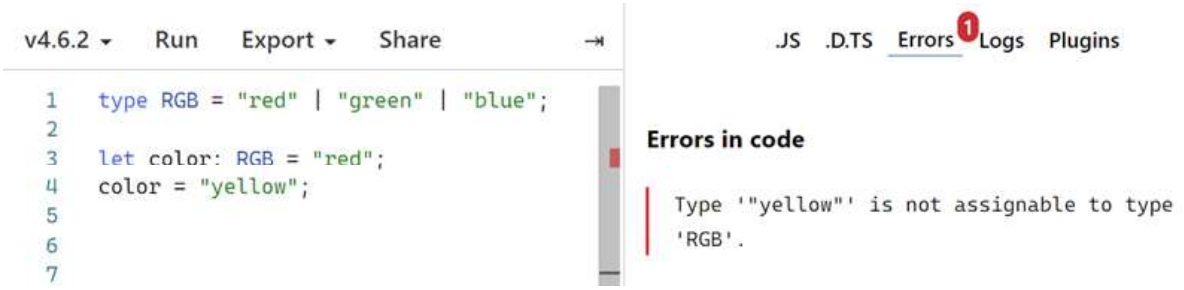

Figure 2.25 – Type error on the union type

Union types consisting of strings are great when a type can only hold a specific set of strings, as in the preceding example.

Here's a recap of what we have learned about creating types:

- Objects and functions can be represented using type aliases or interfaces. They have very similar capabilities, but the type alias syntax is a little more intuitive for representing functions.
- The `?` symbol can specify that an object property or function parameter is optional.
- Type annotations can be added to class properties and constructor and method parameters to make them type-safe.

- Like string-based union types, string-based enumerations are great for a specific set of strings. A string union type is the simplest approach if the strings are meaningful. If the strings aren't meaningful, then a string enumeration can be used to make them readable.

Now that we have covered types, next, we will learn about the TypeScript compiler.

Using the TypeScript compiler

In this section, we will learn how to use the TypeScript compiler to type check code and transpile it into JavaScript. First, we will use Visual Studio Code and create a simple TypeScript project containing code we have written in a previous section. We will then use the terminal within Visual Studio Code to interact with the TypeScript compiler.

Open Visual Studio Code in a blank folder of your choice, and carry out the following steps:

1. In the **EXPLORER** panel in Visual Studio Code, create a file called `package.json` containing the following content:

```
{
  "name": "tsc-play",
  "dependencies": {
    "typescript": "^4.6.4"
  },
  "scripts": {
    "build": "tsc src/product.ts"
  }
}
```

   The file defines a project name of `tsc-play` and sets TypeScript as the only dependency. The file also defines an npm script called `build` that will invoke the TypeScript compiler (`tsc`), passing it to the `product.ts` file in the `src` folder. Don't worry that `product.ts` doesn't exist – we will create it in *step 3*.

2. Now open the Visual Studio Code terminal by selecting **New Terminal** from the **Terminal** menu, then enter the following command:

```
npm install
```

   This will install all the libraries listed in the `dependencies` section of `package.json`. So, this will install TypeScript.

3. Create a folder called `src` and then create a file called `product.ts` within it.

4. Open `product.ts` and add the following content:

```
class Product {
  constructor(public name: string, public unitPrice:
    number) {
    this.name = name;
    this.unitPrice = unitPrice;
  }
  getDiscountedPrice(discount: number): number {
    return this.unitPrice - discount;
  }
}

const table = new Product("Table", 45);
console.log(table.getDiscountedPrice(5));
```

This code will be familiar from the section on using classes. The code can be copied from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter2/Section4-Using-the-compiler/src/product.ts`.

5. Enter the following command in the terminal:

```
npm run build
```

This will run the npm `build` script we defined in the first step.

After the command finishes, notice that a `product.js` file appears next to `product.ts` in the `src` folder.

6. Open the transpiled `product.js` file and read the content. It will look as follows:

```
var Product = /** @class */ (function () {
  function Product(name, unitPrice) {
    this.name = name;
    this.unitPrice = unitPrice;
    this.name = name;
    this.unitPrice = unitPrice;
  }
  Product.prototype.getDiscountedPrice = function
    (discount) {
    return this.unitPrice - discount;
```

```
  };
  return Product;
})();
var table = new Product("Table", 45);
console.log(table.getDiscountedPrice(5));
```

Notice that the type annotations have been removed because they aren't valid JavaScript. Notice also that it has been transpiled to JavaScript, capable of running in very old browsers.

The default configuration that the TypeScript compiler uses isn't ideal. For example, we probably want the transpiled JavaScript in a completely separate folder, and are likely to want to target newer browsers.

7. The TypeScript compiler can be configured using a file called `tsconfig.json`. Add a `tsconfig.json` file at the root of the project, containing the following code:

```
{
  "compilerOptions": {
    "outDir": "build",
    "target": "esnext",
    "module": "esnext",
    "lib": ["DOM", "esnext"],
    "strict": true,
    "jsx": "react",
    "moduleResolution": "node",
    "noEmitOnError": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "build"]
}
```

This code can be copied from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter2/Section4-Using-the-compiler/tsconfig.json`.

Here's an explanation of each setting in the `compilerOptions` field:

- `outDir`: This is the folder that the transpiled JavaScript is placed in.
- `target`: This is the version of JavaScript we want to transpile to. The `esnext` target means the next version.
- `Module`: This is the type of module used within the code. The `esnext` module means standard JavaScript modules.

- `Lib`: Gives the standard library types included in the type checking process. `DOM` gives the browser DOM API types, and `esnext` are types for APIs in the next version of JavaScript.
- `Strict`: When set to `true`, means the strictest level of type checking.
- `Jsx`: When set to `React`, allows the compiler to transpile React's JSX.
- `moduleResolution`: This is how dependencies are found. We want TypeScript to look in the `node_modules` folder, so we have chosen `node`.
- `noEmitOnError`: When set to `true`, means the transpilation won't happen if a type error is found.

The `include` field specifies the TypeScript files to compile, and the `exclude` field specifies the files to exclude.

> **Note**
>
> For more information on the TypeScript compiler options, see the following link: `https://www.typescriptlang.org/tsconfig`.

8. The TypeScript compiler configuration now specifies all files in the `src` folder to be compiled. So, remove the file path on the `build` script in `package.json`:

```
{
  ...,
  "scripts": {
    "build": "tsc"
  }
}
```

9. Delete the previous transpiled `product.js` in the `src` folder.
10. Rerun the `build` command in the terminal:

```
npm run build
```

This time the transpiled file is placed in a `build` folder. You will also notice that the transpiled JavaScript now uses classes that are supported in modern browsers.

11. The final thing we are going to try is a type error. Open `product.ts` and update the constructor to reference an incorrect property name:

```
class Product {
  constructor(public name: string, public unitPrice:
    number) {
    this.name = name;
```

```
    this.price = unitPrice;
  }
  ...
}
```

12. Delete the `build` folder to remove the previously transpiled JavaScript file.
13. Rerun the `build` command in the terminal:

```
npm run build
```

The type error is reported in the terminal. Notice that the JavaScript file is not transpiled.

In summary, TypeScript has a compiler, called `tsc`, that we can use to carry out type checking and transpilation as part of a continuous integration process. The compiler is very flexible and can be configured using a file called `tsconfig.json`. It is worth noting that Babel is often used to transpile TypeScript (as well as React), leaving TypeScript to focus on type checking.

Next, we will recap what we have learned in this chapter.

Summary

TypeScript complements JavaScript with a rich type system, and in this chapter, we experienced catching errors early using TypeScript's type checking.

We also learned that JavaScript types, such as `number` and `string`, can be used in TypeScript, as well as types that only exist in TypeScript, such as `Date` and `unknown`.

We explored union types and learned that these are great for representing a specific set of strings. We now understand that string enumerations are an alternative to string union types if the string values aren't very meaningful.

New types can be created using type aliases. We learned that type aliases could be based on objects, functions, or even union types. We now know that the `?` symbol in a type annotation makes an object property or function parameter optional.

We also learned a fair bit about the TypeScript compiler and how it can work well in different use cases because it is very configurable. This will be important when we start to use TypeScript with React in the next chapter. There, we will learn different ways of setting up React and TypeScript projects before learning to strongly-type React props and state.

Questions

Answer the following questions to check what you have learned about TypeScript:

1. What would the inferred type be for the `flag` variable in the following code?

```
let flag = false;
```

2. What is the return type in the following function?

```
function log(message: string) {
  return console.log(message);
}
```

3. What is the type annotation for an array of dates?
4. Will a type error occur in the following code?

```
type Point = {x: number; y: number; z?: number};
const point: Point = { x: 24, y: 65 };
```

5. Use a type alias to create a number that can only hold integer values between and including 1 and 3.
6. What TypeScript compiler option can be used to prevent the transpilation process when a type error is found?
7. The following code raises a type error because `lastSale` can't accept `null` values:

```
type Product = {
  name: string;
  lastSale: Date;
}
const table: Product = {name: "Table", lastSale: null}
```

   How can the `Product` type be changed to allow `lastSale` to accept `null` values?

Answers

1. The `flag` variable would be inferred to be a `boolean` type.
2. The return type in the function is `void`.
3. An array of dates can be represented as `Date[]` or `Array<Date>`.
4. A type error will not be raised on the `point` variable. It doesn't need to include the `z` property because it is optional.

5. A type for numbers 1-3 can be created as follows:

```
type OneToThree = 1 | 2 | 3;
```

6. The `noEmitOnError` compiler option (set to `true`) can be used to prevent the transpilation process when a type error is found.
7. A union type can be used for the `lastSale` property to allow it to accept `null` values:

```
type Product = {
  name: string;
  lastSale: Date | null;
}
const table: Product = {name: "Table", lastSale: null}
```