

EXPERT INSIGHT

Early Access

# Full-Stack React, TypeScript, and Node

Build scalable and cloud-ready web applications  
using React 19, TypeScript, and Docker

**Second Edition**

David Choi

«packt»

# Full-Stack React, TypeScript, and Node

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Early Access Publication:** Full-Stack React, TypeScript, and Node

**Early Access Production Reference:** B18413

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

**ISBN:** 978-1-80323-577-6

[www.packt.com](http://www.packt.com)

# Table of Contents

1. [Full-Stack React, TypeScript, and Node: Build scalable and cloud-ready web applications using React 19, TypeScript, and Docker](#)
2. [1 Understanding TypeScript](#)
  - I. [Join our book community on Discord](#)
  - II. [Technical requirements](#)
  - III. [What is TypeScript?](#)
  - IV. [Why is TypeScript necessary?](#)
  - V. [Dynamic versus static typing](#)
  - VI. [Object-oriented programming](#)
    - i. [Encapsulation](#)
    - ii. [Abstraction](#)
    - iii. [Inheritance](#)
    - iv. [Polymorphism](#)
  - VII. [Summary](#)
3. [2 Exploring TypeScript](#)
  - I. [Join our book community on Discord](#)
  - II. [Technical requirements](#)
  - III. [What are types?](#)
    - i. [How do types work?](#)
  - IV. [Exploring TypeScript types](#)
    - i. [The any type](#)
    - ii. [The unknown type](#)
    - iii. [Intersection and union types](#)
    - iv. [Type literal](#)
    - v. [Type aliases](#)
    - vi. [Function return types](#)
    - vii. [Functions as types](#)
    - viii. [The never type](#)
  - V. [Understanding classes and interfaces](#)
    - i. [Classes](#)
    - ii. [Interfaces](#)
  - VI. [Understanding inheritance](#)
    - i. [Abstract classes](#)

- ii. [Interface inheritance](#)
- VII. [Understanding polymorphism](#)
- VIII. [Learning generics](#)
- IX. [Utility types](#)
  - i. [ReturnType<Type>](#)
  - ii. [Pick<Type, Keys>](#)
  - iii. [Omit<Type, Keys>](#)
- X. [Summary](#)

# Full-Stack React, TypeScript, and Node: Build scalable and cloud-ready web applications using React 19, TypeScript, and Docker

**Welcome to Packt Early Access.** We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: Understanding TypeScript
2. Chapter 2: Exploring TypeScript
3. Chapter 3: Building Better Apps with FS64- Features
4. Chapter 4: Learning Single page Application Concepts and How React Enables Them
5. Chapter 5: React Development with Components and Hooks
6. Chapter 6: Setting Up Our project Using Vitejs and Testing with Vitest
7. Chapter 7: Learning Redux, React Router, and React Query
8. Chapter 8: Learning Server-Side Development with Nodejs and Express
9. Chapter 9: Setting Up an Express project with TypeScript and Docker
10. Chapter 10: What We Will Build — Forum Application

# 1 Understanding TypeScript

Join our book community on Discord



<https://packt.link/EarlyAccessCommunity>. JavaScript is an enormously popular and powerful language. According to GitHub's State of the Octoverse 2022, it is the most popular language in the world. However, for large application development, its feature set is considered incomplete, and this is why TypeScript was created. In this chapter, we'll learn about the TypeScript language, why it was created, and what value it provides to JavaScript developers. We'll learn about the design philosophy Microsoft used in creating TypeScript and how these design decisions added important features for large application development. We'll also see how TypeScript enhances and improves JavaScript. We'll compare and contrast the JavaScript way of writing code with the way code is written in TypeScript. TypeScript has a wealth of cutting-edge features to benefit developers. Chief among them are static typing and **Object-Oriented Programming (OOP)** capabilities. These features can help make code that is higher quality and easier to



maintain. By the end of this chapter, you will understand how TypeScript provides additional features on top of JavaScript and makes writing large, complex applications easier and less prone to error. In this chapter, we're going to cover the following main topics:

- What is TypeScript?
- Why is TypeScript necessary?

## Technical requirements

To take full advantage of this chapter, you should be an intermediate developer experienced in coding with another type-safe language and platform. You'll also need to install Node and a JavaScript code editor, such as **Visual Studio Code** (we'll walk through the setup of our development environment together). You can find the GitHub repository for this chapter at <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node-2nd-Edition>. Use the code in the `Chap1` folder.

## What is TypeScript?

**TypeScript** is actually two distinct but related technologies – a language and a compiler:

- The language is a feature-rich, statically typed programming language that adds true object-oriented capabilities to JavaScript.
- The compiler converts TypeScript code into native JavaScript, but also provides the programmer with guidance in the form of errors during development.

TypeScript enables the developer to design software that's of higher quality. By using TypeScript, a developer can write code that is easier to understand and refactor and contains fewer bugs. Additionally, it adds discipline to the development workflow by forcing errors to be fixed while still in development. I will show you examples of these important TypeScript capabilities as we proceed with the chapter. TypeScript is a development-time technology. There is no runtime component, and no TypeScript code ever runs on any machine. Instead, the TypeScript compiler converts TypeScript into JavaScript, and that code is then deployed and run on browsers or servers. So then, if TypeScript has no runtime, how do developers get running code? TypeScript uses a process called transpilation. **Transpilation** is a method where code from one language is "compiled" or converted into another language. This is in contrast to standard software compilation, where code is converted into a binary representation that is intended to be run on the destination machine. The transpilation process does not result in a binary file, but an entirely different language. It is then this language that is ultimately run. Now that you know what TypeScript is. In the next section, we'll learn about why these features are necessary for building large, complex applications.

## Why is TypeScript necessary?

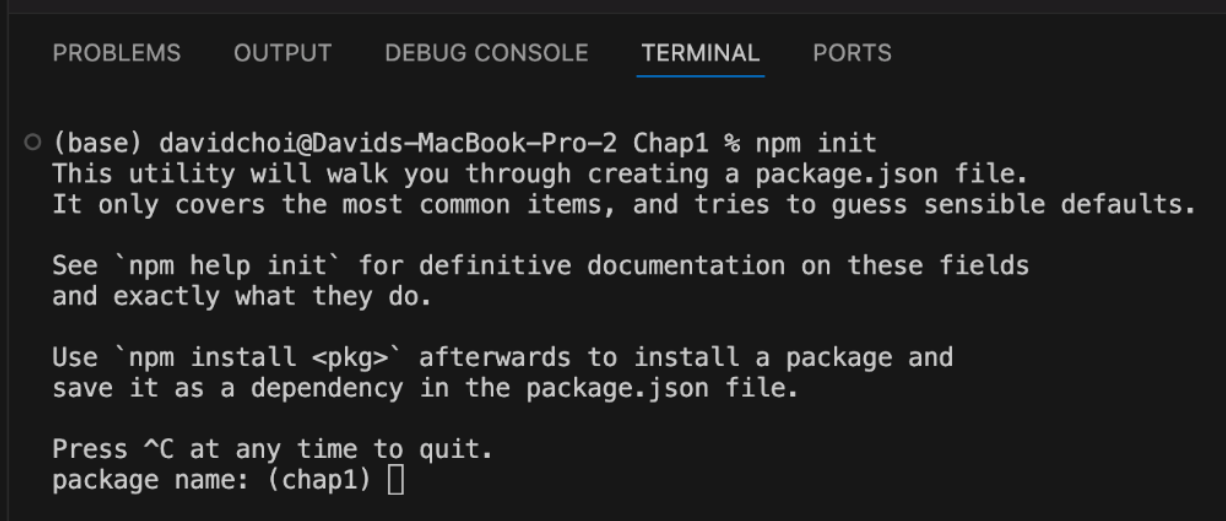
The JavaScript programming language was created by Brendan Eich and added to the Netscape browser in 1995. Since that time, JavaScript has enjoyed enormous success and is now used to build servers and desktop apps as well. However, this popularity and ubiquity have turned out to be a problem as well as a benefit. As larger and larger apps have been created, developers have started to notice the limitations of the language. Large application development requires more sophisticated language features than the browser development JavaScript was first created for. In general, almost all large application development languages, such as Java, C++, C#, and so on, provide static typing and OOP capabilities. In this section, we'll go over the advantages of static typing over JavaScript's dynamic typing. We'll also learn about OOP

and why JavaScript's method of doing OOP is too limited to use for large apps. But first, we'll need to install a few dependencies and programs to allow our examples to run. To do this, follow these instructions:

1. Let's install Node first. Node is a JavaScript runtime. It allows us to run JavaScript code outside of a browser. But Node also gives us **npm**, **Node Package Manager**, which is a JavaScript dependency manager that will allow us to install TypeScript as well as other dependencies. If you're familiar with Maven or Nuget it's effectively the same thing. We'll dive deep into Node in *Chapter 8, Learning Server-Side Development with Node.js and Express*. You can download Node from here: <https://nodejs.org/>. Please install version 323 or higher.
2. Install VSCode. It is a free code editor and its high-quality and rich features have quickly made it the standard development application for writing JavaScript code on any platform. You can use any code editor you like, but I will use VSCode extensively in this book.
3. Create a folder in your personal directory called `FullStackTypeScript`. We'll save all our project code in this folder.
4. Inside `FullStackTypeScript`, create another folder called `Chap1`.
5. Open VSCode and go to **File | Open**, and then open the `Chap1` folder you just created. Then, select **View | Terminal** and enable the terminal window within your VSCode window.
6. Type the following command into the terminal. This command will initialize your project, by creating a file called `package.json`, so that it can accept npm package dependencies. You'll need this since TypeScript is downloaded as a npm package:

```
npm init
```

`npm` is the main command, and whatever comes after can be thought of as parameters. You should see a screen like this:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

○ (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (chap1) 
```

Figure 1.1 – `npm init` screen

Accept the defaults for all the prompts, as we will only install TypeScript for now.

1. If you now open `package.json` you will see something like this:

```
{
  "name": "chap1",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
```

```
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
  }
}
```

Most of the fields are self-explanatory, but notice we have a section called `scripts`. This is where we would create scripts that we use to build, test, and execute our project. For this simple example, we will not use this section, but we will use it extensively in later chapters. Additionally, lower down, you can see the `dependencies` section, which refers to dependency packages our project needs.

1. Now, normally in a real project, I would install TypeScript with the following command:

```
npm install typescript
```

`npm install` is the command that installs packages locally to the project. However, in this case, we will install TypeScript globally to make things a bit easier to use for these simple examples. So then write this command and execute it:

```
npm install -g typescript
```

As you can see, we have added the `-g` parameter, which will cause TypeScript to be installed globally on the machine and make it available anywhere on the command line. Note that if you are running a Mac, you will probably need to enter `sudo` in order to install TypeScript globally.

### IMPORTANT NOTE

If you are already familiar with **npm** and have previously installed TypeScript globally, you must update your version to at least 5.5 or later.

After all the items have been installed, your VSCode screen should look like this:

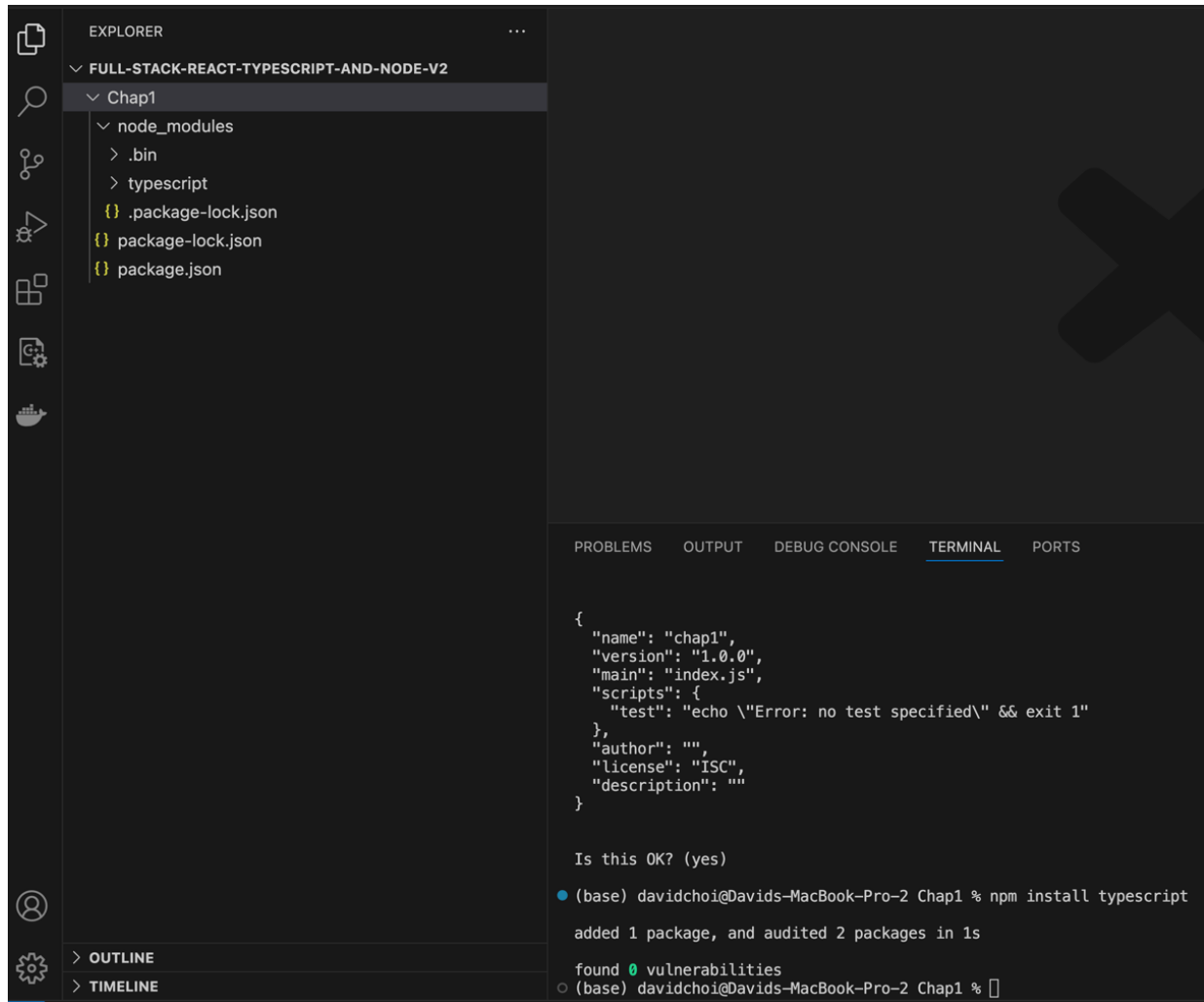


Figure 1.2 – VSCode after setup is complete

We've finished installing and setting up our environment. Now, we can take a look at some examples that will help us better understand the benefits of TypeScript.

## Dynamic versus static typing

Every programming language has and makes use of types. A **type** is simply a set of reusable rules that describe an object or variable. JavaScript is a dynamically typed language. In JavaScript, new variables do not need to declare their type, and even after they are set, they can be reset to a different type. This feature adds awesome flexibility to the language, but it is also the source of many bugs. TypeScript takes the alternative route and uses **static typing**. Static typing forces the developer to indicate the type of variable up front, when they create it. This removes ambiguity and eliminates many conversion errors between types. In the following examples, we'll take a look at some of the pitfalls of dynamic typing and how TypeScript's static typing can eliminate them:

### IMPORTANT NOTE

It is very important that you write out the code from scratch and try not to rely solely on the existing project files. Writing code yourself and attempting to run it will be the best way to learn.

1. On the root of the `Chap1` folder, let's create a file called `string-vs-number.ts`. The `.ts` file extension is a TypeScript-specific extension and allows the TypeScript compiler to recognize the file and transpile it into JavaScript.
2. Next, enter the following code into the file and save it:

```
let a = 5;
let b = '6';
console.log(a + b);
```

The code is very simple. We declared two variables `a` and `b`, and give them values. Then we print the sum of those values using `console.log`, which is the function that allows printing to the terminal or browser console.

1. Let's now transpile the code into JavaScript. In the terminal, type the following:

```
tsc string-vs-number.ts
```

`tsc` is the command to execute the TypeScript compiler, and the filename tells the compiler to check and transpile the file into JavaScript.

1. Once you run the `tsc` command, you should see a new file, `string-vs-number.js`, in the same folder. This is the transpiled JavaScript file. Let's run this file with the following command:

```
node string-vs-number.js
```

The `node` command acts as a runtime environment for the JavaScript file. In other words, it interprets the JavaScript inside the `.js` file and executes it. We'll learn much more about Node in *Chapter 8, Learning Server-Side Development with Node.js and Express*. Once you have run this script, you should see this:

```
56
```

Obviously, if we add two numbers together normally, we want a sum to happen, not a string concatenation. However, since the JavaScript runtime has no way of knowing this, it guesses the desired intent and converts the number variable into a string, and appends variable `b` to it.

1. Let's introduce TypeScript's static typing into this code and see what happens. Take a look at the updated code:

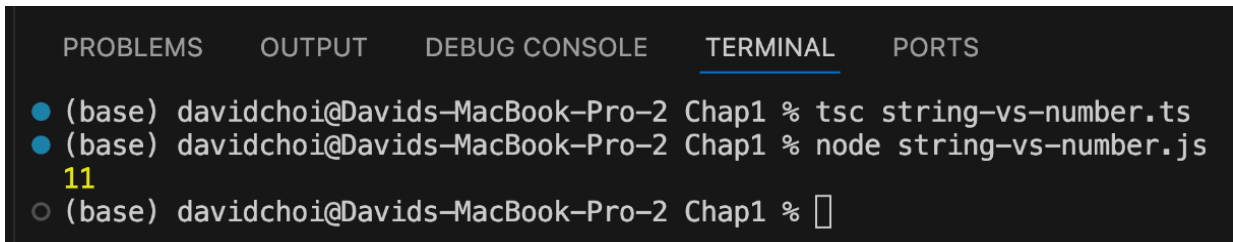
```
let a: number = 5;
let b: number = '6';
console.log(a + b);
```

As you can see, we've added `": number"` after each variable. This syntax is called **Type Annotation**. And we are explicitly using it to indicate to TypeScript what type these variables should be.

1. If you run the `tsc` compiler on this code, you will get the error  
Type 'string' is not assignable to type 'number'. This is exactly what we want. The compiler tells us that there is an error in our code and prevents the compilation from completing successfully.
2. Since we indicated that both variables are supposed to be numbers, the compiler checks for that and complains when it finds it not to be true. So, if we fix this code and set `b` to be a number, let's see what happens:

```
let a: number = 5;
let b: number = 6;
console.log(a + b);
```

1. Now, if you run the compiler, it will complete successfully, and running the JavaScript will result in the value 11 as shown here:



The screenshot shows a terminal window with a dark background. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is selected and underlined), and 'PORTS'. Below the tabs, there are three lines of terminal output: a blue prompt character followed by '(base) davidchoi@Davids-MacBook-Pro-2 Chap1 % tsc string-vs-number.ts', a blue prompt character followed by '(base) davidchoi@Davids-MacBook-Pro-2 Chap1 % node string-vs-number.js', and the output '11' in yellow. The third line shows a grey prompt character followed by '(base) davidchoi@Davids-MacBook-Pro-2 Chap1 %' and a cursor.

Figure 1.3 – Valid numbers addition

Great, we now know when we make these kinds of typing errors, TypeScript will catch our error and prevent it from being used at runtime. Let's look at another more complex example:

1. Create a new .ts file called test-age.ts and add the following code to it:

```
function canDrive(usr) {
  console.log("user is", usr.name);

  if (usr.age >= 16) {
    console.log("allow to drive");
  } else {
    console.log("do not allow to drive");
  }
}

const tom = {
  name: "tom"
}

canDrive (tom);
```

As you can see, the code has a function that checks the age of a user and determines, based on that age, whether they are allowed to drive. However, after this function definition, we see that a user is created but with no `age` property. This would become an issue when running the `canDrive` function, since Tom's `age` property would be undefined. An undefined `usr.age` conditional statement would resolve to false. This behavior is known as being falsy in JavaScript. Note, we'll learn about JavaScript's falsy feature as well as many other important JavaScript capabilities in *Chapter 3: Building Better Apps with ES6+ Features*. In the case of this example, if it turned out that the user `tom` was over 16 years old our `canDrive` function would run the wrong statement and cause a bug. There are ways in JavaScript to deal with this problem, at least partially. We could use a `for` loop to iterate through all of the property key names of the user object and check for an `age` name. Then, we could throw an exception or have some other error handler deal with this issue. However, if we had to do this on every function, it would become inefficient and onerous very quickly. Additionally, we would be doing these checks while the code is running. Obviously, for these errors, we would prefer to catch them before they reach users. TypeScript provides a simple solution to this issue and catches the error before the code even makes it into production. Take a look at the following updated code:

```
interface User {
  name: string;
  age: number;
}
```

```

function canDrive(usr: User) {
    console.log("user is", usr.name);

    if(usr.age >= 16) {
        console.log("allow to drive");
    } else {
        console.log("do not allow to drive");
    }
}

const tom = {
    name: "tom"
}
canDrive (tom);

```

Let's go through this updated code. At the top, we see something called an interface, and it is given the name of `User`. I'll detail interfaces and other types in later chapters, but for now, we can say an interface is a type without an implementation, otherwise known as a **contract**. The `User` interface has the two fields that we need: `name` and `age`. In other words, any object that is of type `User` must have those two fields. Now, below that, we see that our `canDrive` function's `usr` parameter has a type of `User`. And this, of course, tells the compiler only to allow parameters of the `User` type to be given to `canDrive`. Therefore, when I try and compile this code, the compiler complains that when `canDrive` is called, `age` is missing from the passed-in parameter `tom`— because it does not have that field:

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
⊙ (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % tsc test-age.ts
test-age.ts:20:10 - error TS2345: Argument of type '{ name: string; }' is not assignable to parameter of type 'User'.
  Property 'age' is missing in type '{ name: string; }' but required in type 'User'.

20 canDrive(tom);
   ~~~~~

test-age.ts:3:3
3   age: number;
  ~~~~~
'age' is declared here.

Found 1 error in test-age.ts:20
⊙ (base) davidchoi@Davids-MacBook-Pro-2 Chap1 %

```

Figure 1.4 – `canDrive` error

1. Once again, the compiler has caught our error. Let's try fixing this issue by giving `tom` a type, without adding the `age` property:

```

const tom: User = {
    name: "tom"
}

```

If we give `tom` a type of `User`, but do not add the required `age` property, we get the same error:

```
Property 'age' is missing in type '{ name: string; }' but required in type 'User'.ts(2741)
```

Clearly, we still have the same issue. So then, let's add the missing `age` property, and we should now see that the error goes away and our `canDrive` function works as it should. Here's the final working code:

```

interface User {
    name: string;
    age: number;
}

```

```

}

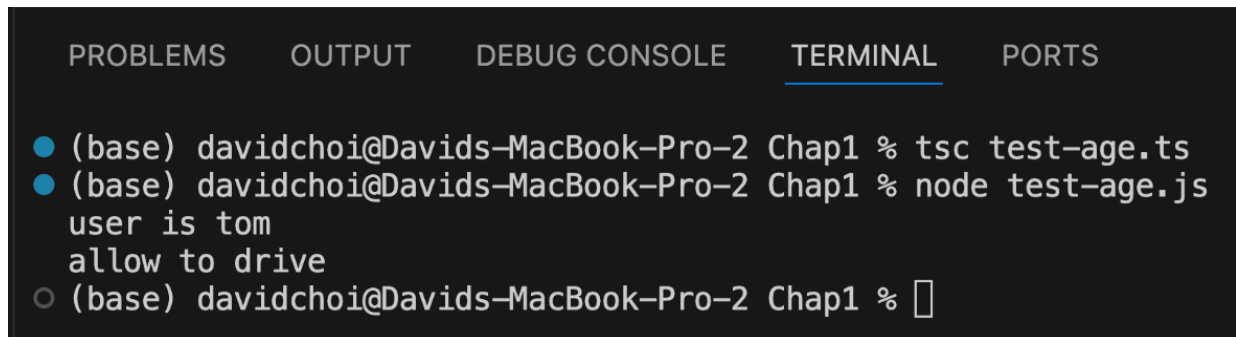
function canDrive(usr: User) {
    console.log("user is", usr.name);

    if(usr.age >= 16) {
        console.log("allow to drive");
    } else {
        console.log("do not allow to drive");
    }
}

const tom: User = {
    name: "tom",
    age: 25
}
canDrive (tom);

```

This code provides the required `age` property in the `tom` variable so that when `canDrive` is executed, the check for `usr.age` is done correctly and the appropriate code is run. Here's a screenshot of the output once this fix is made and the code is run again:



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % tsc test-age.ts
● (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % node test-age.js
  user is tom
  allow to drive
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % 

```

Figure 1.5 – `canDrive` successful result

In this section, we learned about some of the pitfalls of dynamic typing and how TypeScript's static typing can help protect against those issues. Static typing removes ambiguity from code, both to the compiler and other developers. This clarity can reduce errors and make for higher-quality code. In the next section, we will introduce object-oriented programming. This is a commonly used programming methodology that helps improve code quality.

## Object-oriented programming

JavaScript is known as an *Object-Oriented Programming language*. And it does have some of the capabilities of other OOP languages. However, JavaScript's implementation is limited when compared to TypeScript. In this section, we'll take a look at how JavaScript does OOP and how TypeScript improves upon JavaScript's capabilities. First, let's list the four pillars of Object Oriented Programming. There are four major principles of OOP:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Let's review each one.



## Encapsulation

An easier phrase for understanding encapsulation might be *information hiding*. In every program, you will have data and methods that allow you to do something with that data. When we use encapsulation, we are taking that data and putting it into a container of sorts. This container is known as a class in most programming languages, and basically, it protects that data so that nothing outside of the container can directly modify or view it. If you want to make use of the data, it must be done through methods that are controlled by the container object. This style of working with object data allows strict control of what happens to that data from a single place in code, instead of being dispersed through many locations across an application. The syntax for controlling information access, of course, varies across languages, but in TypeScript, this syntax is called **Accessors**. For example, the `private` accessor only allows members to be accessed by code within the class, and the `public` accessor allows any code outside or inside the class to modify the affected member. Until recently, there was no direct way of creating accessors in JavaScript. If a developer needed to indicate accessibility, they had to use TypeScript syntax to do so or write more involved JavaScript code structures to do so. However, it is now possible to hide members using plain JavaScript by using the `#` symbol, and we will prefer this JavaScript syntax in the book.

### IMPORTANT NOTE

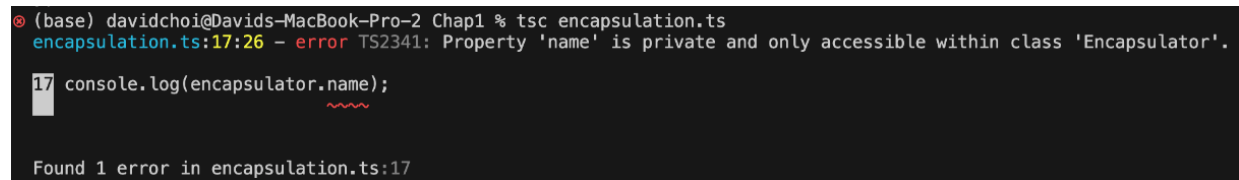
TypeScript attempts to be as close as possible to the current global standard for JavaScript, known as **ECMAScript**. Having said that, sometimes TypeScript syntax will be temporarily different from JavaScript until JavaScript “catches up”. Whenever a change to the JavaScript language overlaps with TypeScript’s features, in this book, we will prefer the standard JavaScript syntax.

Note that in the next section, we will provide some simple examples of OOP in TypeScript. However, in *Chapter 2, Exploring TypeScript*, we will see more robust examples of TypeScript capabilities, including OOP features. For now, let’s keep things simple so we can focus on the core concepts of OOP. Now let’s look at an example of using TypeScript’s accessors to enable encapsulation. Create a new `ts` file called `encapsulation.ts` and add the following code to it:

```
class Encapsulator {
  private name: string;
  get getName(): string {
    return this.name;
  }
  set setName(name: string) {
    this.name = name;
  }
  constructor(name: string) {
    this.name = name;
  }
}
const encapsulator = new Encapsulator("John");
console.log(encapsulator.name);
```

Now it’s a bit early to start digging too deeply into this code. However, let’s try and understand what is going on here at least at a basic level. First, when creating any kind of system to hide information, we would need some kind of container. And in this case, that is what our class type `Encapsulator`, is giving us. Next, we can see we have a single field called `name`. Now, a class does give us a container, but if we don’t expressly indicate any accessor type on our fields, we end up with public fields by default. Therefore, we use the `private` keyword to make our field `name` hidden from the outside world (we’ll use the `#` accessor in later code, but by using the word `private`, I am emphasizing its hidden nature). Next, we have decided to use getters and setters, `getName` and `setName`, in order to expose access to our `name` field outside of the class, but only indirectly. This means that we control in what manner our `name` field is

accessed and updated or even if it will be allowed to be updated. Now, in order to create a unique instance of our class, we use the `new` keyword with the class name like this: `new Encapsulator()`. If you compile and run this file, you should see something like this.



```
Ⓢ (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % tsc encapsulation.ts
encapsulation.ts:17:26 - error TS2341: Property 'name' is private and only accessible within class 'Encapsulator'.

17 console.log(encapsulator.name);
                        ~~~~~

Found 1 error in encapsulation.ts:17
```

*Figure 1.6 – Encapsulation example*

As you can see, since our `name` field is private, it is not accessible directly from outside of our class instance. Let's do a small update to fix this and view the name of our `Encapsulator` instance. At the end of our code, replace `encapsulator.name` with `encapsulator.getName`. Notice `getName` has no `()` appended to it. Now, if we compile and run our code, we should see the name **John** since that's what we passed to the class's constructor. We've now used a simple feature of TypeScript to perform Encapsulation and hide internal member information. Don't worry if not everything is crystal clear right now. As stated earlier, we'll be diving deep into TypeScript in *Chapter 2, Exploring TypeScript*. Let's continue with the next pillar of OOP, Abstraction.

## Abstraction

**Abstraction** is related to encapsulation. When using abstraction, you hide the internal implementation of how data is managed and provide a simplified interface to outside code. Primarily, this is done in order to create "loose coupling" of code. Loose coupling means that code responsible for one set of data is independent and separated from other code. In this way, it is possible to change the code in one part of the application without adversely affecting the code in another part. Abstraction for most OOP languages requires the use of a mechanism to provide access to an object, without revealing that object's internal workings. For most languages, this is called an interface or abstract class. Interfaces are like classes whose members have no actual working code. An abstract class is more flexible; you can have members both with and without an implementation. You can consider them to be a shell that only reveals the names and types of object members, but does not implement how they work. This capability is extremely important in producing the loose coupling mentioned previously and allowing code to be more easily modified and maintained. JavaScript does not support interfaces or abstract classes. TypeScript, however, supports both. Let's look at a simple example to better understand Abstraction. Create a new file called `abstraction.ts` and add this code to it:

```
interface User {
  name: string;
  age: number;
  canDrive();
}
class Person implements User {
  name: string;
  age: number;
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
  canDrive() {
    console.log("user is", this.name);
    if (this.age >= 16) {
      console.log("allow to drive");
    } else {
```

```

        console.log("do not allow to drive");
    }
}
const john: User = new Person("john", 15);
john.canDrive();

```

Now, in our previous section, we already introduced the concept of an interface with the `canDrive` example. So, in this code, we are extending that example by fleshing it out a bit more. First, we have our `User` interface; however, you can see we've added an empty function called `canDrive`. This function has the same signature as our original `canDrive` function, but is part of the `User` type. And as you can see, it takes no parameters and returns nothing. Next, we have our class `Person`. And you can see that when our `Person` class is declared, it indicates an additional keyword, `implements`. The keyword `implements` is telling TypeScript that our class `Person` is intending to define the implementation, the running code, of the members of `User`. So then our class `Person` near the bottom declares `canDrive` and gives it an implementation. To be clear, we could have given `canDrive` any implementation we wanted, but in this case, we've decided to give it the same implementation as our last example. And then finally we create an instance of `Person` and call the `canDrive` function. Now, if you compile and run this code, you will see this:

```

● (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % tsc abstraction.ts
● (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % node abstraction
user is john
do not allow to drive

```

*Figure 1.7 – Abstraction example*

The output is basically the same as our prior `canDrive` example. But by using interfaces, we have now decoupled our code's implementation from its signature. This is a powerful capability, because we no longer need to write our code for a specific implementation. Instead, we can now write code to just a shell and we can have any implementation we want depending on the situation.

## Inheritance

**Inheritance** is about code reuse. For example, if you needed to create objects for several types of vehicles —car, truck, and motorcycle—it would be inefficient to write distinct code for each vehicle type. It would be better to create a base type that has the core attributes of all vehicles, and then reuse that code in each specific vehicle type. This way, we write some of the needed code only once and share it across each vehicle type. The main feature used to create inheritance structures in both JavaScript and TypeScript is classes. Let's define a class, as we've been referring to classes but haven't yet defined them. A class is a kind of type that stores a related set of fields (data) and also may have functions called methods that can act on those fields. JavaScript supports inheritance by using a system called **Prototypical Inheritance**. Basically, what this means is that in JavaScript, every object instance of a specific type shares the same instance of a single core object. This core object is the prototype, and whatever fields or methods are created on the prototype are accessible across the various object instances. This is a good way of saving resources, such as memory, but it does not have the level of flexibility or sophistication of the inheritance model in TypeScript. In TypeScript, classes can inherit from other classes, but they can also inherit from interfaces and abstract classes. We've actually already seen the interface form of inheritance in the Abstraction example. We'll dive deep into these features in Chapter 2, *Exploring TypeScript*, but the point is that TypeScript has a more capable inheritance model than JavaScript. It allows for more kinds of inheritance and, therefore, more ways to reuse code. Let's look at a single example now.

1. Let's create a new `ts` file called `inheritance.ts` like this:

```

class Item {
  id: string;
  description: string;
  price: number;
  getId(): string {
    return this.id;
  }
}
class Bicycle extends Item {
  wheelCount: number;
  getWheelCount(): number {
    return this.wheelCount;
  }
}
const bicycle = new Bicycle();
bicycle.id = "123";
bicycle.description = "Mountain Bike";
bicycle.price = 299.99;
bicycle.wheelCount = 2;
console.log("id", bicycle.getId());
console.log("wheel count", bicycle.getWheelCount());

```

In this example, we are creating types for a company that sells many different types of products. First, we have a base class called `Item` that represents the most rudimentary information for all the products that our store sells. In other words, all the items sold by our store will always have this information: `id`, `description`, `price`, and `getId`. Since this is true, we don't need to recreate these fields and methods for each new type of item. We can simply reuse them. The definition of the `Bicycle` type shows us how to reuse the code in the `Item` type by using the `extends` keyword. By using this keyword, our `Bicycle` type does not need to define those members again and simply inherits them. However, since it is a specific item, it has its own members, `wheelCount` and `getWheelCount`, which we have also defined. After the definition of our two types, we can see that we create an instance of the `Bicycle` and set all of its fields, including those defined in the `Item` type. And we finish by logging the values of `id` and `wheelCount`.

1. If we run this code, you will see this output:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
● (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % tsc inheritance.ts
● (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % node inheritance
id 123
wheel count 2
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap1 % 

```

*Figure 1.8 – Inheritance example*

As you can see, our `id` and `wheelCount` values are displayed.

## Polymorphism

**Polymorphism** is related to inheritance. In polymorphism, it is possible to declare an object that is of some base type and then set that variable at runtime to a specific type that inherits from that base type. This technique is useful for scenarios where the specific type to use can only be known at runtime. In the case of JavaScript, there is no direct language support for polymorphism. It is possible to simulate it to some

degree, but there is no built-in way of enforcing the specific types to be used. TypeScript, however, can be used to implement Polymorphism. This topic is a bit complex, and we don't know enough about TypeScript just yet to explore this concept thoroughly. We'll learn more about TypeScript in Chapter 2, *Exploring TypeScript*, and then I'll provide examples for this topic there.

## Summary

In this chapter, we introduced TypeScript and learned why it was created. We learned why type safety and OOP capabilities are so important for building large apps. Then, we saw some examples comparing dynamic typing and static typing and saw why static typing is a better way of writing complex code. Finally, we compared the styles of OOP between the two languages and learned why TypeScript has a better and more capable system. The information in this chapter has given us a high-level conceptual understanding of the benefits of TypeScript. In the next chapter, we'll do a deeper dive into the TypeScript language. We'll learn more about types and investigate some of the most important features of TypeScript, such as classes, interfaces, and generics.

## 2 Exploring TypeScript

Join our book community on Discord



<https://packt.link/EarlyAccessCommunity> In this chapter, we'll dive deeper into the TypeScript language. We'll learn about TypeScript's explicit type declaration syntax, as well as the many built-in types in TypeScript and their purpose. We'll also learn how to create our own types. By the end of this chapter, you will have a strong understanding of the TypeScript language, which will allow you to read and

understand existing TypeScript code with ease. In this chapter, we're going to cover the following main topics:

- What are types?
- Exploring TypeScript types
- Understanding classes and interfaces
- Understanding inheritance
- Understanding polymorphism
- Learning generics
- Utility types

## Technical requirements

To take full advantage of this chapter, you should be an intermediate developer experienced in coding with another type-safe language and platform. You'll also need to install Node and a JavaScript code editor, such as **Visual Studio Code (VSCode)**. You can find the GitHub repository for this chapter at <https://github.com/PacktPublishing/Full-Stack-React-TypeScript-and-Node-2nd-Edition>. Use the code in the `Chap2` folder. Before continuing, we will set our base project similarly to how we did in *Chapter 1, Understanding TypeScript*:

1. Go to your `FullStackTypeScript` folder and create a new folder called `Chap2`.
2. Open VSCode and go to **File | Open Folder**, and then open the `Chap2` folder you just created. Then, select **View | Terminal** and enable the terminal window within your VSCode window.
3. Type the `npm init` command to initialize the project for `npm`, and accept all the defaults.
4. This time around, let's install TypeScript into our project instead of globally. Type this command:

```
npm install typescript
```

1. As you can see, this time, we left out the `-g` parameter, and if you look inside the `package.json` file, you should see in the `dependencies` section TypeScript 5.5 or higher. The `-g` parameter means to install the package on the machine and make it available to all projects, as opposed to installing it in the project.
2. Now, before we finish this time, we will also add a `tsconfig.json` file to the root of our `Chap2` folder. There will be a dedicated section later in this



chapter explaining the `tsconfig.json` file, which is responsible for configuring TypeScript's settings, but for now, just copy the `tsconfig.json` file from the `Chap2` source code folder.

Now we're ready to get started.

## What are types?

A **type** is a reusable set of rules. A type may include fields and methods. It can also be shared and reused repeatedly. When you reuse a type, you are creating an **instance** of it. This means that you are creating an *example* of your type that has specific values for fields and can act on those fields. In TypeScript, as the name implies, types are very important. They're the main reason why the language was created in the first place. Let's take a look at how types work in TypeScript.

## How do types work?

JavaScript does have types—number, string, Boolean, array, and so on are all types in JavaScript. However, those types are not explicitly set during the declaration; they are only inferred. In TypeScript, types are normally set during declaration. It is possible to allow the compiler to infer your type, but this feature exists as a convenience, and once a variable's type is set, it cannot be changed. In general, you're going to be explicitly setting your types in TypeScript. Also, in addition to the types supported by JavaScript, TypeScript has its own unique types and allows you to create your own types. Now, the first thing to realize about types in TypeScript is that they are handled by their shape (structure) and not by their name. This means the name of a type is not what is important to the TypeScript compiler. Instead, it's the members that the type has and their types that matter. Let's look at an example:

1. Create a file called `shape.ts` and add the following code:

```
class Person {
  name: string = "";
}
const jill: { name: string } = {
  name: "jill"
};
const person: Person = jill;
console.log(person);
```

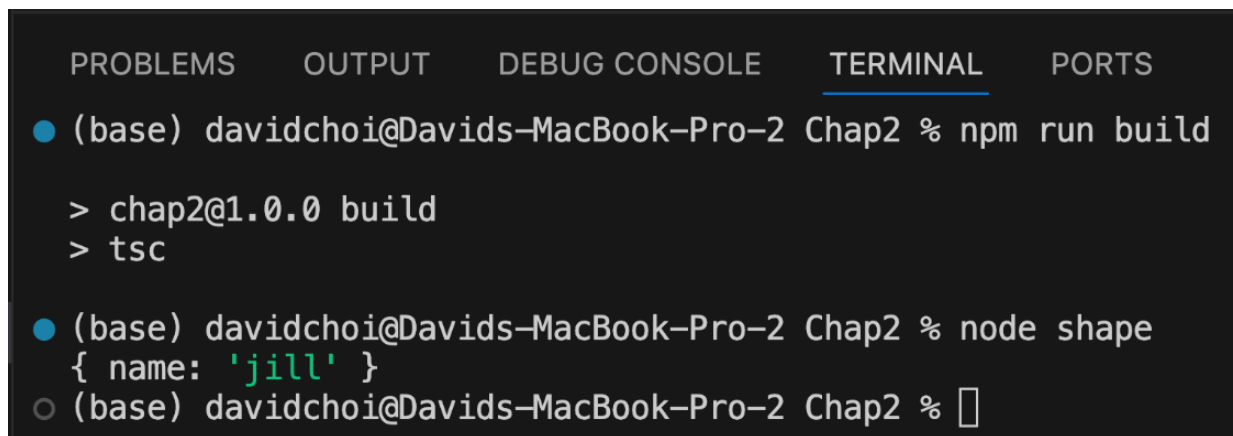
1. The first thing you should notice is that we have two ways of declaring the same type. We have a class called `Person` with a field called `name`. Below that, you see that we have a variable called `jill` that is of type `{ name: string }`. This second type is a little weird because this type declaration has no given name; it's more like a type definition. Nevertheless, these two types are effectively equivalent in TypeScript.
2. Now, below those lines, you can see that we have another variable called `person` of the type `Person`, and we set that variable to `jill`. Again, the compiler does not complain, and everything seems OK.
3. Even though both the `person` object and `jill` have the same `name` field, aren't they different? Something about this feels a little off, but let's continue and see whether we can figure this out.
4. Let's compile this code and run it to see what happens. However, in this case, let's do things a bit differently. As you saw earlier in the chapter, we installed TypeScript directly into the project itself. We are therefore also going to call it from within our project dependencies. Open the `Chap2` file called `package.json` and add this line to the `scripts` section just under "test" (make sure to add a comma after the "test" line):

```
"build": "tsc"
```

1. The `scripts` section allows us to add terminal commands to run various tasks. In this case, we use the standard name for creating a build, "build", and then that name is then *runnable* by `npm`. This command we just created tells TypeScript to transpile all files on the root of the project. Let's compile and run our code by using these two commands, run separately:

```
npm run build
node shape
```

Notice that we don't need to give a file extension. Once you've run the commands, you should see the following:

A screenshot of a VS Code terminal window. The terminal has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is active and underlined), and PORTS. The terminal shows a series of commands and their outputs. First, a blue prompt character indicates a new shell session: (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build. This is followed by two lines of output: > chap2@1.0.0 build and > tsc. Then, another blue prompt character shows: (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node shape. The output of this command is { name: 'jill' }. Finally, a grey prompt character shows: (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % followed by a cursor.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node shape
{ name: 'jill' }

○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.1 – The shape.ts output*

As you can see, the code compiles and runs without issue and we get the expected output. In TypeScript, unlike other languages, the compiler looks at the shape or structure of a type and is not concerned with its name at all. This is why the TypeScript compiler treats `person` as the same type as `jill`. You will see in later chapters, as we dig more deeply into TypeScript's type system, why it is so important to be aware of this behavior. TypeScript uses all the base JavaScript types and provides type declarations that can be used in annotations. Let's learn about the base TypeScript types now.

## Exploring TypeScript types

In this section, we'll look at some of the core types available in TypeScript. Using most of these types will give you error-checking and compiler warnings that can help improve your code. They will also provide information about your intent to other developers on your team. So, let's continue and see how these types work.

### The any type

**any** is a dynamic type that can be set to any other type. If you declare a variable to be of type `any`, this means that you can set it to anything and later reset it to anything. It is, in effect, *no type* because the compiler will not check the type on your behalf. There is a key fact to remember about the `any` compiler: the compiler will not intercede and warn you of issues at development time. Therefore, if possible, using the `any` type should be avoided. In a large application, it is not always possible for a developer to control the objects that are

used in their code. For example, if a developer is relying on a web service API call to get data, that data's type may be controlled by some other team or even a different company entirely. It is also possible that an API result schema may change frequently. Situations such as these require type flexibility and an escape hatch from the type system. The `any` type can provide that escape hatch. It is important not to abuse the `any` type. You should be careful to only use it when you have no other way of creating a type. There are, however, some alternatives to using the `any` type, and the `unknown` type is one of them. We'll cover that type next.

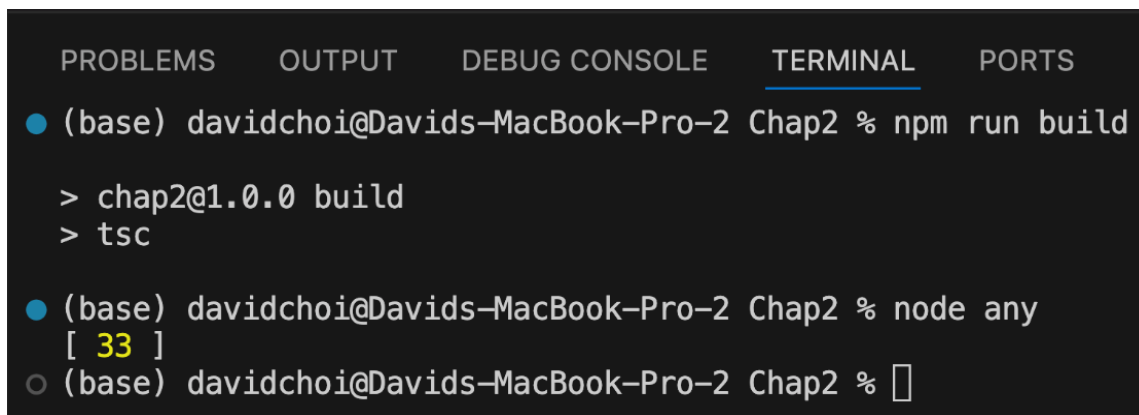
## The unknown type

**unknown** is a type released in TypeScript version 3. It is similar to `any` in that once a variable of this type is declared, a value of any type can be set to it. That value can subsequently be changed to any other type, just like the `any` type. However, you cannot call any of its members or set the variable as a value to another variable without first confirming what type it really is. The only time you can set an `unknown` variable to another variable, without first checking its type, is when you set an `unknown` type to another `unknown` or an `any` type. Let's take a look at an example of `any`, and then we'll see why the `unknown` type is preferable to using the `any` type (it is, in fact, recommended by the TypeScript team):

1. First, let's take a look at an example of the issue with using `any`. Go to VSCode and create a file called `any.ts`, and then type the following code:

```
let val: any = 22;
val = "string value";
val = new Array();
val.push(33);
console.log(val);
```

1. First, the `val` variable is declared, set to the `any` type, and given a value of `22`, a number. Then, that same variable is set to `string`. Then, it is reset into an empty `Array`. Finally, the `Array` has a `push` method called on it, which adds an element with a value of `33` to the end of the `Array`. If you run this code using the following commands, you will see this result:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build
> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node any
[ 33 ]
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.2 – any run result*

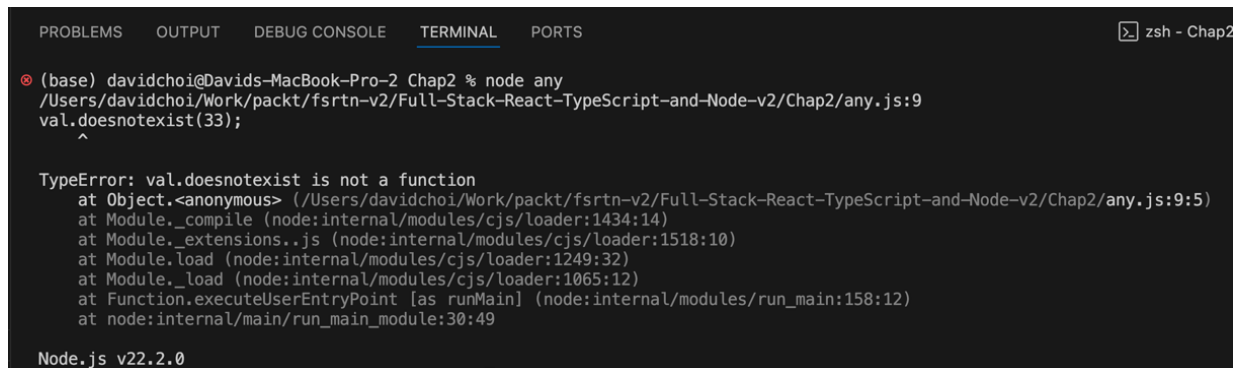
2. Since `val` is of the `any` type, we can set it to whatever we like. As you can see, we set the variable to multiple different types. But once it is set to an array, we call `push` on it, since `push` is a method of `Array`. However, this is obvious only because we, as developers, are aware that `Array` has a method called `push` on it. What if we accidentally called something that does not exist on `Array`? Let's replace the previous code with the following:

```
let val: any = 22;
val = "string value";
val = new Array();
val.doesNotExist(33);
console.log(val);
```

1. As you can see, the new code contains a call to a function called `doesNotExist` that is clearly not a valid `Array` function.
2. Let's try running the TypeScript compiler again:

```
npm run build
```

1. The compiler succeeds with no errors, unfortunately, since making something of the `any` type causes the compiler to no longer check the type. Additionally, we also lost IntelliSense, the VSCode development-time code highlighter and error checker. When you hover your mouse over the `doesNotExist` function, all you see is the `any` type. Only when we try and run the code do we get any indication that there is a problem, which is never what we want. Let's see what the exact error is when we run the code:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS zsh - Chap2
(base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node any
/Users/davidchoi/Work/packt/fsrtn-v2/Full-Stack-React-TypeScript-and-Node-v2/Chap2/any.js:9
val.doesnotexist(33);
  ^
TypeError: val.doesnotexist is not a function
    at Object.<anonymous> (/Users/davidchoi/Work/packt/fsrtn-v2/Full-Stack-React-TypeScript-and-Node-v2/Chap2/any.js:9:5)
    at Module._compile (node:internal/modules/cjs/loader:1434:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1518:10)
    at Module.load (node:internal/modules/cjs/loader:1249:32)
    at Module._load (node:internal/modules/cjs/loader:1065:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:158:12)
    at node:internal/main/run_main_module:30:49
Node.js v22.2.0
```

Figure 2.3 – any failing

In a complex application, it is an easy error to make, even if the mistake is simply mistyping something. Let's see a similar example using `unknown` :

1. First, comment out your code inside of `any.ts` and delete the `any.js` file (as we will use the same variable names, if you do not do this, it will cause conflict errors).

### IMPORTANT NOTE

We'll learn about something called namespaces later, which can eliminate these sorts of conflicts.

2. Now, create a new file called `unknown.ts` and add the following code to it:

```
let val: unknown = 22;
val = "string value";
val = new Array();
val.push(33);
console.log(val);
```

1. You will notice that VSCode gives you an error immediately, complaining about the `push` function. This is weird since obviously, `Array` has a method called `push` in it. This behavior shows how the `unknown` type works. You can consider the `unknown` type to be sort of like a label more than a type, and underneath that label is the actual type. However, the compiler cannot figure out the type on its own, so we need to explicitly *prove* the type to the compiler ourselves.
2. We use type guards to prove that `val` is of a certain type:

```
let val: unknown = 22;
val = "string value";
```

```

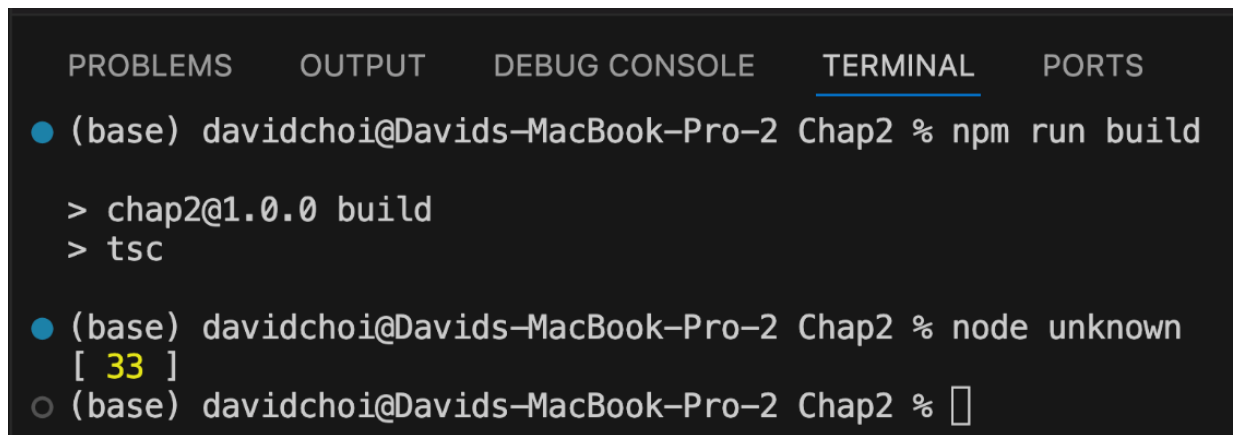
val = new Array();
if (val instanceof Array) {
    val.push(33);
}
console.log(val);

```

1. As you can see, we've wrapped our `push` call with a test to see whether `val` is an instance of `Array`.

Note that we'll learn more about `instanceof` in *Chapter 3, Building Better Apps with ES6+ Features*.

2. Once we've made this check, the call to `push` can proceed without error, as shown here:



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node unknown
[ 33 ]
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % 

```

Figure 2.4 – unknown

This mechanism is a bit cumbersome since we always have to test the type before calling members. However, it is still preferable over using the `any` type and a lot safer since it is checked by the compiler.

## Intersection and union types

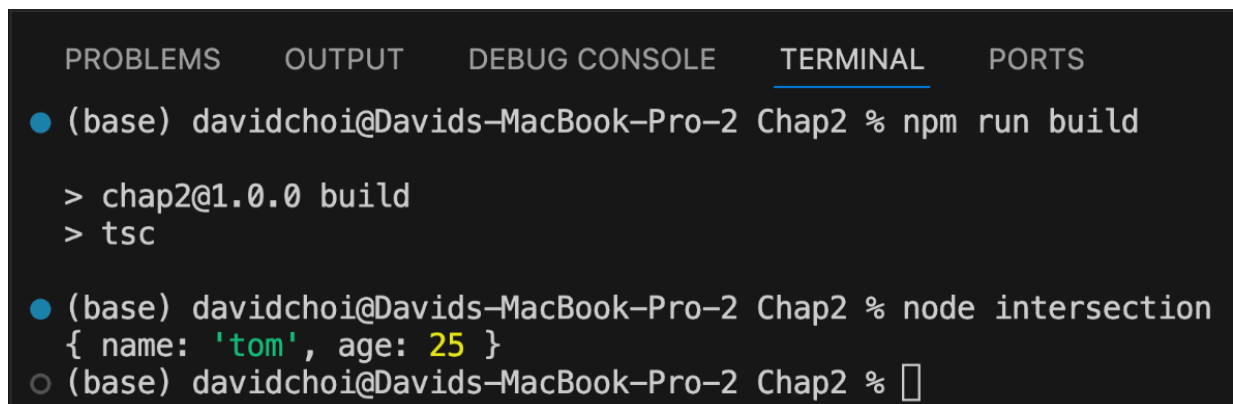
Remember when we started this section by saying that the TypeScript compiler focuses on type shape and not the name? This mechanism allows TypeScript to support what are called **intersection** types. This means that TypeScript allows the developer to create new types by merging multiple distinct types together. This is hard to imagine, so let me give you an example. If you look at the following code, you can see a variable called `obj` that has what looks like two types associated with it, with each type having only one field, `name` or `age`:

```
let obj: { name: string } & { age: number } = {  
  name: 'tom',  
  age: 25  
}
```

What we are doing in this code is merging two distinct types into a new single type by using the `&` symbol. This is why the `obj` variable can be set to the value that has both the `name` and `age` fields. Let's try running this code and displaying the result on the console. Create a new file called `intersection.ts` and add the following code to it:

```
let obj: { name: string } & { age: number } = {  
  name: 'tom',  
  age: 25  
}  
console.log(obj);
```

If you compile and run this code, you will see an object that contains both the `name` and `age` properties together:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build  
  
  > chap2@1.0.0 build  
  > tsc  
  
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node intersection  
  { name: 'tom', age: 25 }  
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.5 – Intersection result*

As you can see, both IntelliSense and the compiler accept the code, and the final object has both fields. This is an **intersection** type. Now there is another type that is somewhat similar in behavior to the intersection type, and that is the **union** type. In the case of unions, instead of merging types, we are using them in an “or” fashion, where it’s one type or another. Let’s look at an example. Create a new file called `union.ts` and add the following code to it:

```
let unionObj: null | { name: string } = null;  
unionObj = { name: 'jon'};  
console.log(unionObj);
```



The `unionObj` variable is declared to be of the `null` or `{ name: string }` type, by using the `|` symbol. If you compile and run this code, you'll see that it compiles and accepts both type values, just not at the same time. This means that the type value can be either `null` or an object instance of the `{ name: string }` type.

## Type literal

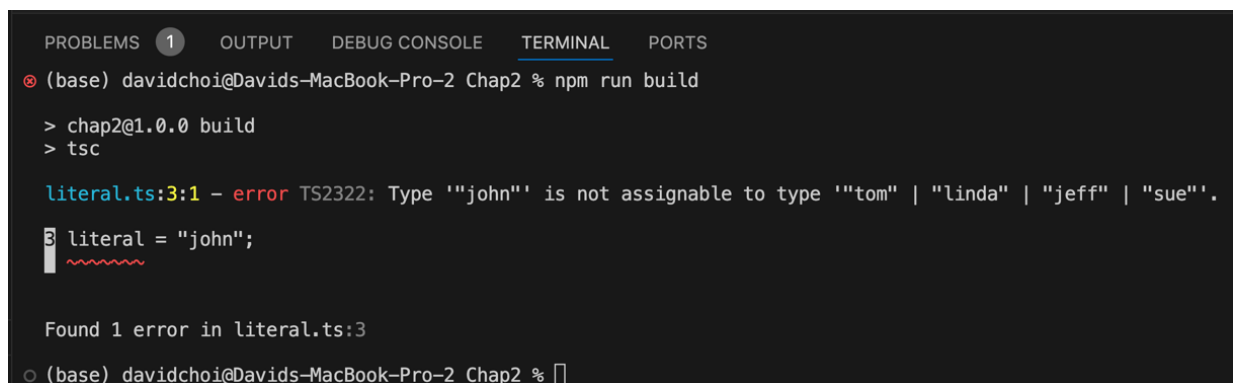
**Type literals** are similar to union types, but they use a set of hardcoded values. These values can be strings, numbers, arrays, and even objects. Let's create another file called `literal.ts` and add this simple example of string literals:

```
let literal: "tom" | "linda" | "jeff" | "sue" = "linda";
literal = "sue";
console.log(literal);
```

As you can see, we have a bunch of hardcoded strings as the type. This means that only values that are the same as any of these strings will be accepted for the `literal` variable. The compiler is happy to receive any of the values on the list, and even have it reset. However, it will not allow the setting of a value that is not on the list. Doing that will give a compile error. Let's see an example of this. Update the code as shown by resetting the `literal` variable to `john`:

```
let literal: "tom" | "linda" | "jeff" | "sue" = "linda";
literal = "sue";
literal = "john";
console.log(literal);
```

Here, we set the `literal` variable to `john`, and compiling gives the following error:



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
ⓧ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

literal.ts:3:1 - error TS2322: Type '"john"' is not assignable to type '"tom" | "linda" | "jeff" | "sue"'.
3  literal = "john";
   ~~~~~

Found 1 error in literal.ts:3

ⓧ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.6 – A literal error*

This type is great when you have multiple possible values for a variable, but you want to make them specific and limited.

## Type aliases

**Type aliases** are used very frequently in TypeScript. This is simply a method to give a different name to a type, and often it is used to provide a shorter, simpler name to some complex type. For example, here's one possible usage:

```
type Points = 20 | 30 | 40 | 50;
let score: Points = 20;
console.log(score);
```

In this code, we take a long numeric literal type and give it a shorter name of `Points`. Then, we declare `score` as the `Points` type and give it a value of `20`, which is one of the possible values for `Points`. And, of course, if we tried to set the score to, let's say, `99`, the compilation would fail. Another example of type aliases would be for object literal type declarations:

```
type ComplexPerson = {
  name: string,
  age: number,
  birthday: Date,
  married: boolean,
  address: string
}
```

Since the type declaration is very long and does not have a name, like, for example, a class would, we use an alias instead. Type aliasing can be used for just about any type in TypeScript, including things such as functions and generics, which we'll explore further later in the chapter.

## Function return types

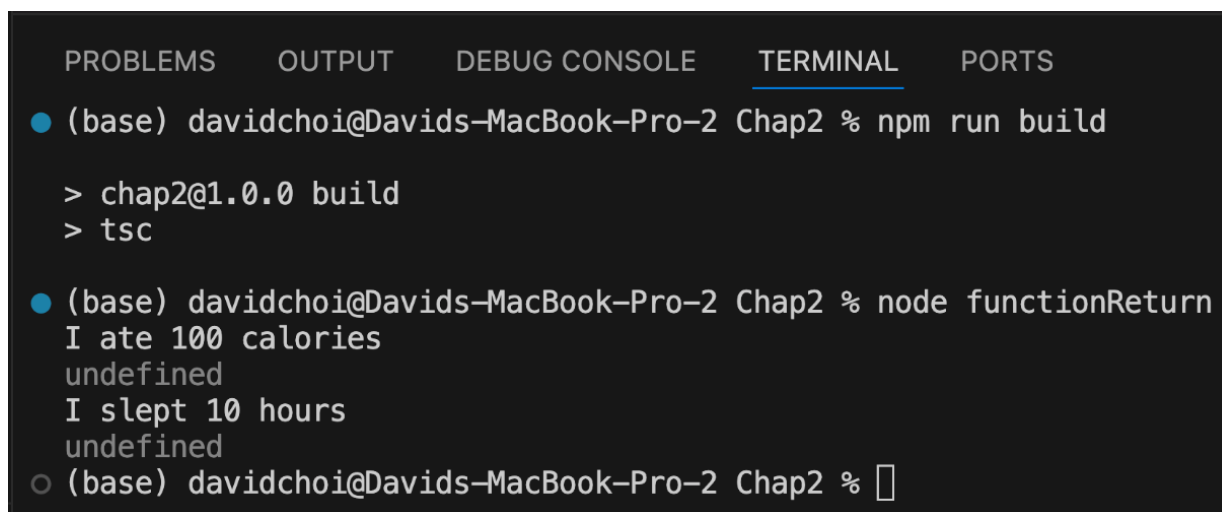
For completeness' sake, I wanted to show one example of a function return declaration. It's quite similar to a typical variable declaration. Create a new file called `functionReturn.ts` and add this code to it:

```
function runMore(distance: number): number {
  return distance + 10;
}
console.log(runMore(20));
```

The `runMore` function takes a parameter called `distance` of type `number` and returns a `number`. The parameter declaration is just like any variable declaration, but the function return comes after the parentheses and indicates what type is returned by the function. If you compile and run this function, it will, of course, display `30` in the terminal. If a function returns nothing, then you can either not declare any type for the return or you can declare `void` to be more explicit. Let's look at an example of returning `void`. Comment out the `runMore` function and console log, and then compile and run this code:

```
function eat(calories: number) {  
    console.log("I ate " + calories + " calories");  
}  
function sleepIn(hours: number): void {  
    console.log("I slept " + hours + " hours");  
}  
let ate = eat(100);  
console.log(ate);  
let slept = sleepIn(10);  
console.log(slept);
```

The two functions both return `void`, but only the `sleepIn` function is explicit about that. Here's the output:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build  
  
> chap2@1.0.0 build  
> tsc  
  
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node functionReturn  
I ate 100 calories  
undefined  
I slept 10 hours  
undefined  
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.7 – Function void results*

As you can see, their internal `console.log` statements do run and display messages. However, the two variables, `ate` and `slept`, which accept the function returns, are both **undefined**—since that is the value of something without a value in JavaScript. So, the function return type declaration is quite similar to variable

declarations. Now, if we use functions as type parameters, it looks a bit different. Let's take a look at that in the next section.

## Functions as types

It may seem a bit odd, but in TypeScript, a type can also be an entire function signature. In TypeScript, this signature can also act as a type for an object's fields or another function's parameters. Let's take a look at an example of this. Create a new file called `functionSignature.ts` and add the following code to it:

```
type Run = (miles: number) => boolean;
let runner: Run = function (miles: number): boolean {
    if (miles > 10) {
        return true;
    }
    return false;
}
console.log(runner(9));
```

The first line shows us a function type that we will be using in this code. The `Run` type alias is only there to make it easier to reuse the long function signature. The actual function type is `(miles: number) => boolean`. This looks odd, but it's just TypeScript's syntax for creating types from functions. So, the only things needed then are the parentheses to indicate parameters, the `=>` symbol, which indicates that this is a function, and then the return type. In the code after the function definition line, you have the declaration of the `runner` variable, which is of the `Run` type—our function type. This function simply checks whether the person has run more than 10 miles and returns `true` if they have and `false` if they have not. Now, this means that our variable `runner` is actually a function, and we can call it like any other function by using parentheses wrapped around a parameter value, like this: `--runner(9)`. This call is passed directly into `console.log`, which writes out the result of the function call. Compile and run this code and you should see this:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node functionSignature
false
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.8 – Function type result*

Calling `runner` with a parameter of `9` would make the function return `false`, which is correct.

## The never type

A type called **never** seems quite strange at first glance, so let's try and understand it. The `never` type is used as a return type for a function that never returns (i.e., does not complete), or a variable that is not set to anything, not even `null`. At first, this sounds like the `void` type. However, they are quite different. In `void`, a function does return, in the completed sense of the word; it just does not return any value (it returns `undefined`, which is effectively no value). In the case of `never`, the function does not finish at all. Now, this may seem totally useless, but it's actually quite powerful for indicating intent. Let's look at an example. Create a file called `never.ts` and add the following code:

```
function oldEnough(age: number): never | boolean {
  if (age > 59) {
    throw Error("Too old!");
  }
  if (age <= 18) {
    return false;
  }
  return true;
}
```

As you can see, this function returns a union type that is either `never` or `boolean`. Now, we could have only indicated `boolean`, and the code would still work. However, in this function, we are throwing an error if the person is over a certain age, indicating that this is an unexpected `age` value. So, since

encapsulation is a high-level principle for writing good-quality code, it is beneficial to indicate explicitly that a failure of the function to return could occur without needing the developer to know about the internals of how the function works. `never` provides that extra information. In this section, we learned about the many built-in types in TypeScript. We were able to see why using these types can improve our code quality and help us catch errors earlier in the coding cycle. In the next section, we'll learn how we can use TypeScript to create our own types and also follow **object-oriented programming (OOP)** principles.

## Understanding classes and interfaces

We've already briefly looked at classes and interfaces in previous sections. Let's take a deeper look and see why these types can help us write better code. Once we complete this section, we will be better prepared to write more readable, reusable code with fewer bugs.

### Classes

At a high level, **classes** in TypeScript look like classes in JavaScript. They are a container for a related set of fields and methods that can be instantiated and reused. However, classes in TypeScript support extra features for encapsulation that JavaScript does not. Let's take a look at a new example.

#### IMPORTANT NOTE

We will be creating many files in this project, and some of them will use the same name for some types. If you get errors about "duplicate identifiers," just go to the other file and comment out that code.

Create a new file called `classes.ts` and enter the following code:

```
class Person {
  constructor() {}
  msg: string = "";
  speak() {
    console.log(this.msg);
  }
}
const tom = new Person();
tom.msg = "hello";
tom.speak();
```

This example has a simple class called `Person` that is very similar to something you would see in JavaScript. Let's explain this code in bullet form to make it easier to follow:

- `Person` : Firstly, we have a name for the class so that it can be easily reused.
- `constructor` : Next, we have a method called `constructor` that can help build an instance of the class. Constructors are used to initialize any fields that the class might have and to do any other setup for the class instance, such as running functions (in this case, it does nothing).

`msg` : Then, we have a single field called `msg`, which is called using the `this` keyword.

- `this` : The `this` keyword represents the running instance of the class, in other words, an actual object instance of the `Person` class. This is a JavaScript feature and indicates that the field or method being called belongs to that object and that object alone. In addition, the `this` keyword can only be called inside the running object within one of its methods.
- `speak` : Next, there is a method called `speak` that writes the `msg` value to the console. We then create an instance of our class. Finally, we set the `msg` field to a value of `hello` and call the `speak` method.

Now, let's look at how classes differ between TypeScript and JavaScript.

## Access modifiers

We stated previously that one of the main principles of object-oriented development is encapsulation, or information hiding. Well, if we take a look at the code again, clearly, we are not hiding the `msg` variable as it is exposed and editable outside of the class. This is the default accessibility of all class members in TypeScript. If an explicit accessor is not set, that member is accessible and modifiable outside of the class. If you want this behavior explicitly, you can set the `public` accessor, but this shouldn't be necessary since again, it is the default. Let's see what TypeScript allows us to do with accessors. Let's update the code like this:

```
class Person {
    constructor(private msg: string) {}

    speak() {
        console.log(this.msg);
    }
}
```

```

}
const tom = new Person("hello");
// tom.msg = "hello";
tom.speak();

```

As you can see, we updated the constructor with a parameter that uses a keyword called `private`. This method of declaring a constructor parameter and also adding an access modifier is doing several things in one line. Firstly, it tells the compiler that the class has a field called `msg` of the `string` type that should be `private`. Secondly, by adding this field to the constructor, we are saying that whenever we call the constructor, e.g., `new Person("hello")`, we want the `msg` field to be set to whatever the parameter is set to. Now, what does setting something to `private` actually do? By setting the field to `private`, we make it inaccessible from outside the class. The result of this is that `tom.msg = "hello"` no longer works and causes an error. Try removing the comments, `//`, before `tom.msg = "hello"` and compiling. You should see this message:



```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
⊗ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

classes.ts:11:5 - error TS2341: Property 'msg' is private and only accessible within class 'Person'.

11  tom.msg = "hello";
    ~~~~~

Found 1 error in classes.ts:11

```

Figure 2.9 – Classes error

As you can see, it complains that a private member, `msg`, cannot be accessed from outside of the class. Also, please note that we only applied our modifier to a field, but access modifiers can be applied to any member field or method. Let's continue modifying this code. Update the `Person` type with this code:

```

class Person {
  private msg: string;
  constructor(msg: string) {
    this.msg = msg;
  }
  speak() {
    this.msg = "speak " + this.msg;
    console.log(this.msg);
  }
}

```



```
const tom = new Person("hello");  
// tom.msg = "hello";  
tom.speak();
```

As you can see, we've modified how the `msg` variable is being set and initialized. This code does the same thing as the code we just saw previously. However, it's a more verbose version. Now, thus far, we've only been using the `private` accessor. However, keep in mind that as we discussed in *Chapter 1, Understanding TypeScript*, there is the alternative `#` symbol, which is part of JavaScript, for making members private. Once we begin building our application, I will utilize that symbol most of the time unless there is some particular capability that only the TypeScript-style accessor can provide. Now, let's learn about the `readonly` modifier. This one is relatively straightforward; it causes a field to become read-only after it has been set one time in the constructor. Let's see what this looks like. Replace the `Person` code with this update by adding `readonly` to the declaration of the `msg` field, like this:

```
class Person {  
    constructor(private readonly msg: string) {}  
  
    speak () {  
        this.msg = "speak " + this.msg;  
        console.log(this.msg);  
    }  
}  
const tom = new Person("hello");  
// tom.msg = "hello";  
tom.speak();
```

Once you complete this update, VSCode IntelliSense complains because, in the `speak` function, we are attempting to change the value of `msg` even though it has already been set once through the constructor, which again is not allowed once you use `readonly` on a field. The `private` and `readonly` access modifiers are not the only modifiers available in TypeScript. There are several other types of access modifiers. However, they will make more sense if we explain them in the context of inheritance later. Now, as we continue this chapter, we will be introducing more features of newer versions of JavaScript, which forces us to modify the TypeScript compiler configuration. So, let's take a short detour and discuss TypeScript configuration.

## TypeScript configuration

TypeScript was first released in late 2012. Since then, multiple versions of ECMAScript, the official JavaScript standard, have been released. Therefore, to allow developers to target their desired JavaScript versions, and also to control various configuration settings around compilation and project setup, `tsconfig.json` was created. This configuration file is quite extensive and allows you to control many aspects of TypeScript configuration and transpilation, but for our purposes, we'll focus on a few of the most often-used settings. Let's learn about TypeScript configuration by adding a `tsconfig.json` file to the root of our Chap2 :

## IMPORTANT NOTE

As you'll recall from the beginning of this chapter, we copied over an existing `tsconfig.json` file. The settings we are about to create will be the same as that file, and so if you like, you can just follow along without having to recreate that file.

1. First, open your terminal and enter this command:

```
npx tsc --init
```

1. (Make sure you use two dashes before `init`.) This code triggers the TypeScript compiler, `tsc`, and causes it to create a file called `tsconfig.json` in the root of our Chap2 folder. You'll also notice that, unlike our installation of TypeScript, we are not using `npm`. Although both command-line tools are related, `npm` is intended for installing dependencies, and `npx` is for executing them. In addition, `npx` allows the execution of some commands without first having to install them globally. Let's continue.
2. Now that we've created our `tsconfig.json` file, let's take a look at some of the settings. Starting at the top, we can see a field called `compilerOptions`, and this is exactly what it seems. This section has various flags for setting compilation/transpilation and development-time IntelliSense. Here's a list of some of the more commonly used flags:
  - `target` : This flag is used to control which version of ECMAScript our code will be transpiled into. Remember, TypeScript is a development-time technology, and therefore, we need to select the desired version of ECMAScript that will actually run. After TypeScript 5.x, the default value is ES2016 (ES is short for ECMAScript). But in earlier TypeScript versions, it was ES3. For our purposes, we want the absolute latest version, so write `ESNext` in your file, like this:

`"target": "ESNext",`

- `lib`: This setting will configure the ECMAScript version and the various APIs that are available during development. In other words, it will provide IntelliSense, which makes available ECMAScript capabilities and methods that are only found in those versions of ECMAScript and API that we select. As you can see, it is an array and can take multiple values. For our development, we will use the following:

```
"lib": [  
  "ESNext",  
  "DOM",  
  "DOM.Iterable"  
]
```

- You already know what `ESNext` is. `DOM` means the API that allows us to interact with DOM nodes. Another way of saying DOM nodes is HTML elements. `DOM.Iterable` allows us to work with collections of DOM nodes, such as `NodeList`.
- `module`: This flag allows us to control what type of module format we will use. A module is simply an encapsulated set of code, sort of like a class, where we can be selective about what to expose to the outside world. A module is always a single JavaScript or TypeScript file.
- The modern form of creating modules uses ES6 syntax, which we will learn about soon. But there is an older form, called CommonJS. This older form is still supported and therefore TypeScript allows you to choose which format you would like. We will again use the latest format based on `ESNext`. Update the `module` flag like this:

`"module": "ESNext"`

- `strict`: This flag forces more type checks and stricter type rules enforcement. It is actually a flag that represents multiple related flags but exists as a single convenience flag, since many devs want the complete set of strictness rules. For example, the `strictNullChecks` sub-flag forces any variables that may get a value of null or undefined to be explicitly set to those types in their variable declaration. This is a pretty important flag since it prevents variables from unintentionally getting set as undefined and potentially causing exceptions at runtime. Another important sub-flag for OOP purposes is `strictPropertyInitialization`. This flag forces fields created within classes to be initialized either at declaration or in the constructor. We will set the `strict` flag to `true` for our project.

OK, we now have the ability to select our desired ECMAScript and API versions. So, let's continue learning about TypeScript class capabilities.

## Getters and setters

Another feature of classes is actually available in both TypeScript and JavaScript: **getters** and **setters**:

- **Getter:** A property that allows modification or validation of a related field before returning it
- **Setter:** A property that allows modification or computation of a value before setting it to a related field

In some other languages, these types of properties are known as computed properties. Let's look at an example. Create another file called `getSet.ts` and add the following code:

```
class Speaker {  
    #message: string = "";  
    constructor(private name: string) {}  
    // getter setter  
}
```

Our code is quickly getting more complicated. If you are new to JavaScript entirely, I provide a JavaScript refresher in *Chapter 3, Building Better Apps with ES6+ Features*. In this chapter, I will focus on TypeScript. The code is short, but there's a fair amount happening here, so let's go over it. First, near the very top, just after the `class Speaker` declaration starts, you can see that our `message` field is not set in the constructor but is set up as a private field, using the `#` symbol, and therefore it is not accessible directly from outside our class. In addition, that field is set immediately on declaration, because in TypeScript, with `strict` mode on, a field must have a value either on declaration or set in the constructor. The only initializer the constructor takes as a parameter is our `name` field. Now, let's add our actual getter and setter and overwrite the `// getter setter` comment with this:

```
get Message() {  
    if (!this.#message.includes(this.name)) {  
        throw Error("message is missing speaker's name");  
    }  
    return this.#message;  
}  
set Message(val: string) {
```

```

    let tmpMessage = val;
    if (!val.includes(this.name)) {
        tmpMessage = this.name + " " + val;
    }
    this.#message = tmpMessage;
}

```

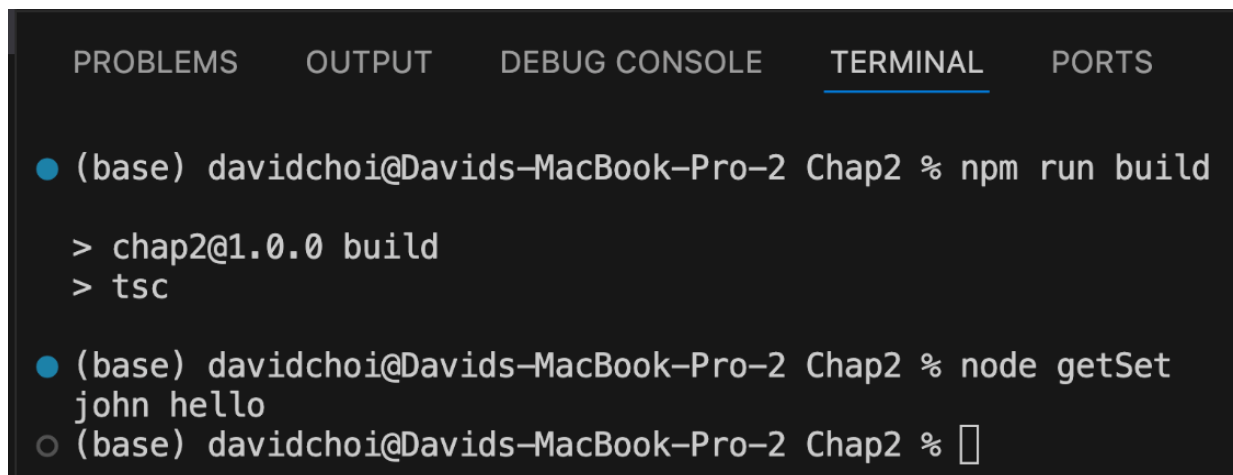
You can see that we start by declaring the `get Message()` property accessor. This is our getter. In the getter, we test to see whether our `message` field value has the speaker's name in it by using the JavaScript `includes` function. This function is like `contains` in other languages, and it searches for the given parameter as a substring of the original string. Now, if our `if` statement does not find the speaker's name (the `!` symbol is a negation symbol), we throw an exception to indicate an unwanted situation. Notice also that our `message` field is called `this.#message`. This is because when we use the `#` symbol when declaring a field, we must always use it subsequently when calling the associated field. The setter, also called `Message`, is indicated by the `set` keyword, and this property receives a string and adds the speaker's name if needed by checking whether it is missing from the `message` field. Note that although both getter and setter look like functions, they are not. When they are called later in code, they are called just like a field would be called *without* the parentheses. Now then, after the definition of our `Speaker` class is finished, we have the actual instantiation and usage of our class, as shown here. Add this code below the setter:

```

const speaker = new Speaker("john");
speaker.Message = "hello";
console.log(speaker.Message);

```

The `speaker` object is instantiated as a new speaker with the name `john` and its `Message` property is set to `hello`. This mechanism allows us to set the `message` field without actually exposing it to the outside world. Thereafter, the message is written to the console. Let's compile and run this code:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

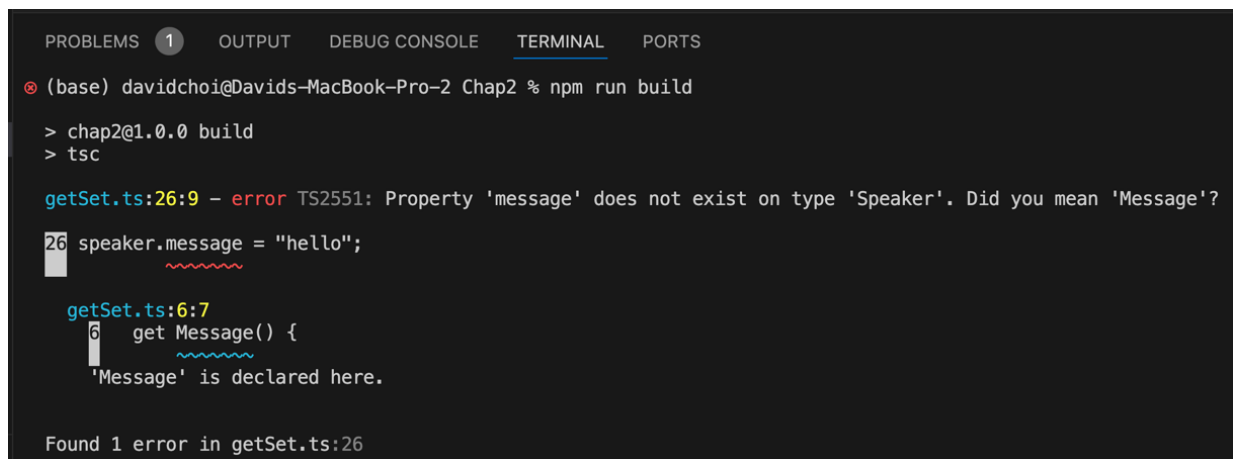
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node getSet
john hello
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.10 – getSet output*

To drive the point home further, let's try switching the `speaker.Message = "hello"` line to `speaker.message = "hello"`. If you compile, you should see this error:



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

⊗ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

getSet.ts:26:9 - error TS2551: Property 'message' does not exist on type 'Speaker'. Did you mean 'Message'?
26 speaker.message = "hello";
    ~~~~~

getSet.ts:6:7
6   get Message() {
    ~~~~~
'Message' is declared here.

Found 1 error in getSet.ts:26
```

*Figure 2.11 – Message field error*

This occurred because `message` is a private field and cannot be accessed from outside our class directly. You may be wondering why I mentioned getters and setters here when they are available in regular JavaScript, too. If you look at the example, you can see that the `message` field is private and the getter and setter properties are public. So, to allow good encapsulation, it is a best practice to hide our field and only expose it when needed via a getter and/or setter or some method that allows modification of the field. Also, remember that when deciding on an access level for your members, you want to start with the most restrictive capabilities first and then become less restrictive as

needed. Additionally, by allowing field access via getters and setters, we can do many different types of checks and modifications, as we've done in our example, so that we have ultimate control over what comes in and out of our class.

## Static properties and methods

Finally, let's discuss **static** properties and methods. When you mark something as `static` inside a class, you are saying that this member is a member of the class type and not of the class instance. Therefore, it can be accessed without needing to create an instance of a class, but instead by prefixing with the class name. Let's look at an example. Create a new file called `staticMember.ts` and add the following code:

```
class ClassA {
  static typeName: string;
  constructor(){}

  static getFullName() {
    return "ClassA " + ClassA.typeName;
  }
}
const a = new ClassA();
console.log(a.typeName);
```

If you attempt to compile this code, it will fail, stating that `typeName` is a static member of the `ClassA` type. Again, static members must be called using the class name. Here is the fixed version of the code:

```
class ClassA {
  static typeName: string;
  constructor(){}

  static getFullName() {
    return "ClassA " + ClassA.typeName;
  }
}
const a = new ClassA();
console.log(ClassA.typeName);
```

As you can see, we reference `typeName` with the class name. So then, the question is, why might you want to use a static member instead of an instance member? Under certain circumstances, it may be useful to share data across class instances. For example, you might want to do something like this:

```

class Runner {
    static lastRunTypeName: string;
    constructor(private typeName: string) {}

    run() {
        Runner.lastRunTypeName = this.typeName;
    }
}
const a = new Runner("a");
const b = new Runner("b");
b.run();
a.run();
console.log(Runner.lastRunTypeName);

```

In the case of this example, I am trying to determine the last class instance that has called the `run` function at any given time. If you compile and run this code, you will see that the displayed value in the terminal will be `a`, because `a`'s `run` method ran last. Another point to be aware of is that inside a class, static members can be called by both static and instance members. However, static members cannot call instance members. Now, we have learned about classes and their features in this section. This will help us design our code for encapsulation, which will enhance its quality. Next, we will learn about interfaces and contract-based coding.

## Interfaces

In OOP design, another important principle is abstraction. The goal of abstraction is to reduce complexity and the tight coupling of code by not exposing the internal implementation (we already covered abstraction in *Chapter 1, Understanding TypeScript*). One way of doing this is to use **interfaces** to show only the signature of a type, as opposed to its internal workings. An interface is also sometimes called a contract, since having specific types for parameters and return types enforces certain expectations between both the user and the creator of the interface. So, another way of thinking about interfaces is as strict rules about what can come out of and go into a type instance. Now, interfaces are just a set of rules. In order to have working code, we need an implementation of those rules to get anything done. So, let's show an example of an interface with implementation to get started. Create a new file called `interfaces.ts` and add the following interface definition:

```

interface Employee {
    name: string;
    id: number;
}

```



```

    isManager: boolean;
    getUniqueId: () => string;
}

```

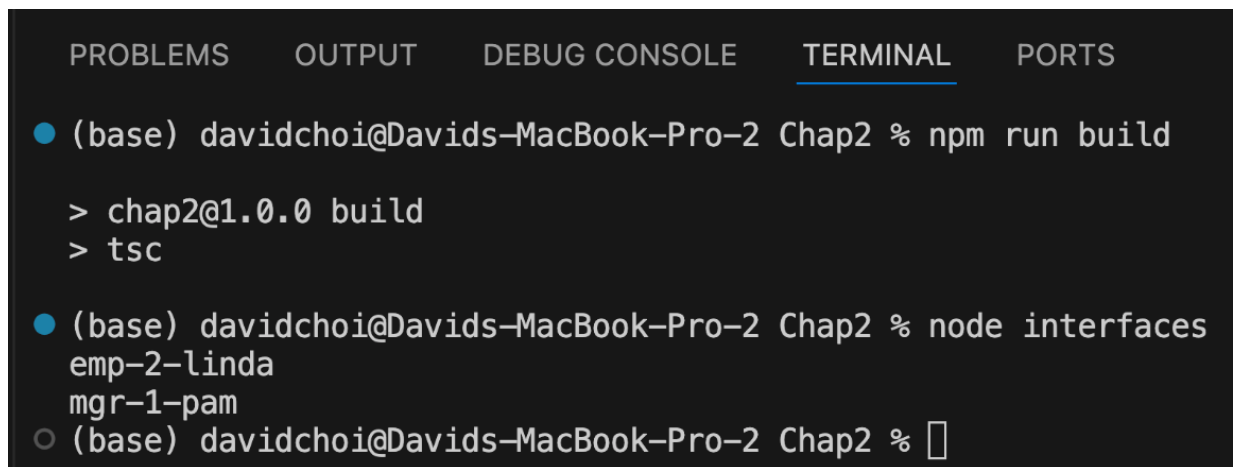
This interface defines an `Employee` type, which we will create instances of later. As you can see, there is no implementation of the `getUniqueId` function, just its signature. The implementation comes later when we define it. Now, add the implementation to the `interfaces.ts` file. Insert the following code, which creates two instances of the `Employee` interface:

```

const linda: Employee = {
  name: "linda",
  id: 2,
  isManager: false,
  getUniqueId: (): string => {
    let uniqueId = linda.id + "-" + linda.name;
    if(!linda.isManager) {
      return "emp-" + uniqueId;
    }
    return uniqueId;
  }
}
console.log(linda.getUniqueId());
const pam: Employee = {
  name: "pam",
  id: 1,
  isManager: true,
  getUniqueId: (): string => {
    let uniqueId = pam.id + "-" + pam.name;
    if(pam.isManager) {
      return "mgr-" + uniqueId;
    }
    return uniqueId;
  }
}
console.log(pam.getUniqueId());

```

So, we create an instance by instantiating an object literal called `linda`, setting the two field names— `name` and `id` —and then implementing the `getUniqueId` function. Later, we console log `linda.getUniqueId` call. After that, we create another object, called `pam`, based on the same interface. However, not only does it have different field values, but its implementation of `getUniqueId` is also different from the `linda` object. This is the main use of interfaces: to allow for a single structure across objects but to enable different implementations. In this way, we provide strict rules for the type structure, but also allow some flexibility in terms of how functions go about doing their work. Here's the output of our code:



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node interfaces
emp-2-linda
mgr-1-pam
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.12 – Employee interface results*

Another possible use of interfaces is when using third-party APIs. Sometimes, the type information is not well documented, and all you're getting back is untyped JSON or the object type is extremely large and has many fields you will never use. It is quite tempting, under these circumstances, to just use `any` as the type and be done with it. However, you should prefer providing a type declaration if at all possible. What you can do under these circumstances is to create an interface that has only the fields that you know and care about. Then, you can declare your data type to be of this type. At development time, TypeScript will not be able to check the type since for API network calls, data will be coming in at runtime.

Regardless, since TypeScript only cares about the shape of any given type, it will ignore the fields not mentioned in your type declaration, and as long as the data comes in with the fields you defined in your interface, the runtime will not complain and you will maintain development-time type safety. However, please do ensure you handle `null` and `undefined` fields appropriately, as they can cause exceptions during runtime. In this section, we learned about interfaces and the differences between interfaces and classes. We will be able to use interfaces to abstract away the implementation details of a class and, therefore, produce loose coupling between our code and, thus, better code quality. In the next section, we will learn about how classes and interfaces allow us to perform inheritance and, therefore, code reuse.

## Understanding inheritance

In this section, we'll learn about **inheritance**. Inheritance in OOP is a method for doing code reuse. This will shrink our application code size; generally, shorter

code tends to have fewer bugs. So, this will improve our app quality once we get started building. As stated, inheritance is primarily about allowing code reuse. Inheritance is also conceptually designed to be like real-life inheritance so that the logical flow of inheritance relationships can be intuitive and easier to understand. Let's look at an example of this now. Create a file called `classInheritance.ts` and add the following code:

```
class Vehicle {
  constructor(private wheelCount: number) {}
  showNumberOfWheels() {
    console.log(`wheels: ${this.wheelCount}`);
  }
}
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
  }
}
class Automobile extends Vehicle {
  constructor() {
    super(4);
  }
}
const motorCycle = new Motorcycle();
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();
```

A quick note if you've never seen backticks, ```, and `${}` before. It's called **string interpolation** and is simply a quick and easy way to insert variable values inside strings.

As you can see, there is a base class, also known as a parent, called `Vehicle`. This class acts as the main container for source code that is being reused later by whatever classes inherit from it, also known as children or subclasses. The child classes inherit from `Vehicle` by using the `extends` keyword. One thing to notice is that in the constructor for each child class, you see that the first line of code is the call to `super`. The first line of code in the constructor of a subclass must be the base class constructor, which is known as `super`. In this case, that would be the `Vehicle` class. Now, as you can see, each child is passing a different number of wheels to the parent's `wheelCount` variable via the parent's constructor. Then, at the end of the code, an instance of each child, `Motorcycle` and `Automobile`, is created and the `showNumberOfWheels` method is called. That method call to `showNumberOfWheels` might seem a bit strange since it's defined inside the parent

Vehicle class. However, that's the point of inheritance—that we can make use of code from base classes when we extend them. If we compile and run this code, we get the following:

```
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node abstractClass
wheels: 2
wheels: 4
```

*Figure 2.13 – The classInheritance result*

So, then, each child provides a different number of wheels to the parent `wheelCount` variable, although they cannot access the variable directly because it's private. Now, let's say that there was a reason why the child classes would want to access the `wheelCount` variable of the parent directly. For example, let's say that if a flat tire occurred, an updated wheel count would be necessary. What could we do? Well, let's try creating a function unique to each child class that tries to update `wheelCount`. Let's see what happens. Update the code by adding a new function, `updateWheelCount`, to the `Motorcycle` class:

```
class Vehicle {
  constructor(private wheelCount: number) {}
  showNumberOfWheels() {
    console.log(`wheels: ${this.wheelCount}`);
  }
}
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
  }
}
class Automobile extends Vehicle {
  constructor() {
    super(4);
  }
}
const motorCycle = new Motorcycle();
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();
```

As a test, we updated only the `Motorcycle` class and added an `updateWheelCount` method to it, and this gave us an error. Can you guess why?

It's because we are trying to access a private member of the parent class. Even when child classes inherit their members from a parent, they still do not have access to that parent's private members. This is the right behavior, again, to promote encapsulation. So, then, what do we do? Well, let's try editing the code again to allow this:

```
class Vehicle {
  constructor(protected wheelCount: number) {}
  showNumberOfWheels() {
    console.log(`wheels: ${this.wheelCount}`);
  }
}
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
  }
}
class Automobile extends Vehicle {
  constructor() {
    super(4);
  }
}
const motorCycle = new Motorcycle();
motorCycle.showNumberOfWheels();
const autoMobile = new Automobile();
autoMobile.showNumberOfWheels();
```

Do you see the small change we made? We changed the `wheelCount` parameter on the `Vehicle` parent class constructor to be of the `protected` accessor type. The `protected` type allows the class and any inheriting classes to have access to the member. Before we move on to the next topic, let's introduce the concept of **namespaces**. Namespaces are like containers that hide their contents from whatever is outside of the namespace. In that sense, it's sort of like a class, but it is capable of containing any number of classes, functions, variables, or any other types. Here's a simple example of using namespaces. Create a new file called `namespaces` and add the following code:

```
namespace A {
  class FirstClass {}
}
namespace B {
  class SecondClass {}
}
```

```

    const test = new FirstClass();
}

```

As you can see from this code, even before compiling, VSCode IntelliSense is already complaining that `FirstClass` cannot be found. This is because it is hidden from namespace `B`, since it is only defined in namespace `A`. This is the purpose of namespaces: to hide information within one scope, away from other scopes. In this section, we learned about inheriting from classes. Class inheritance is a very important tool for reusing code. In the next section, we'll look at using abstract classes, which is a more flexible way of doing inheritance.

## Abstract classes

As mentioned previously, interfaces can be useful for defining contracts, but they have no implementation of working code themselves. Classes have working implementations, but sometimes only a signature is required. It is possible, for certain situations, that we may want to have both classes and interfaces in one object type. For these types of scenarios, you would use an **abstract class** instead of either a class or an interface. Let's create a new file called `abstractClass.ts` and copy and paste our code from our `classInheritance.ts` file into it. If you do this, you might get some errors, since the two files both have the same class and variable names. Please feel free to comment out the code in the file you are not using whenever this occurs. So, in our new `abstractClass.ts` file, we are going to update it with namespaces and modify the `Vehicle` class to be abstract. Add the namespace and update the `Vehicle` class like this:

```

namespace AbstractNamespace {
    abstract class Vehicle {
        constructor(protected wheelCount: number) {}
        abstract updateWheelCount(newWheelCount: number): void;
        showNumberOfWheels() {
            console.log(`wheels: ${this.wheelCount}`);
        }
    }
    // the rest of our existing code
}

```

So, to start, we've wrapped all the code within a scope called namespace `AbstractNamespace`. Again, this is merely a container that allows us to control scoping so that the members of our `abstractClass.ts` file do not bleed out into the global scope. If you look at the new `Vehicle` code, we have a new keyword before the class called `abstract`. This is what indicates that the

class will be an abstract one. You can also see that we have a new function called `updateWheelCount`. This function has an `abstract` keyword in front of it, which indicates that it will have no implementation within the `Vehicle` class and needs to be implemented by an inheriting class. Now, after the `Vehicle` abstract class definition, we want our child classes to implement our `Vehicle` class. So, add the `Motorcycle` and `Automobile` classes below the `Vehicle` class:

```
class Motorcycle extends Vehicle {
  constructor() {
    super(2);
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
    console.log(`Motorcycle has ${this.wheelCount}`);
  }
}
class Automobile extends Vehicle {
  constructor() {
    super(4);
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
    console.log(`Automobile has ${this.wheelCount}`);
  }
  showNumberOfWheels() {
    console.log(`wheels: ${this.wheelCount}`);
  }
}
// more code here
```

After adding the classes, we instantiate them and call their respective `updateWheelCount` methods, as shown:

```
const motorCycle = new Motorcycle();
motorCycle.updateWheelCount(1);
const autoMobile = new Automobile();
autoMobile.updateWheelCount(3);
```

As you can see, the implementation of the `abstract` member `updateWheelCount` is in the child classes. This is the capability that an abstract class provides. An abstract class can act both like a regular class, providing member implementations, and like an interface, providing only the rules to be implemented by a sub-class.

Since an abstract class can have abstract members, you cannot instantiate an abstract class.

If you review the code of the `Automobile` class, you can see that it has its own implementation of `showNumberOfWheels`, even though this function is not abstract. This demonstrates something called **overriding**, which is the ability of a child to create a unique implementation of the parent's member and use that implementation instead. In this section, we learned about the different kinds of class-based inheritance. Learning about inheritance will allow us to reuse more of our code, reducing both code size and potential bugs. In the next section, we'll learn about doing inheritance with interfaces and how it's different from class-based inheritance.

## Interface inheritance

As explained earlier, **interfaces** are a way of setting rules for a type. This is also sometimes called a contract. Interfaces will allow us to separate implementation from definition and, therefore, provide abstraction. Let's learn how to use interfaces with inheritance. Create a new file called `interfaceInheritance.ts` and add the following code:

```
namespace InterfaceNamespace {
  interface Thing {
    name: string;
    getFullName: () => string;
  }
  interface Vehicle extends Thing {
    wheelCount: number;
    updateWheelCount: (newWheelCount: number) => void;
    showNumberOfWheels: () => void;
  }
  // more code coming here
}
```

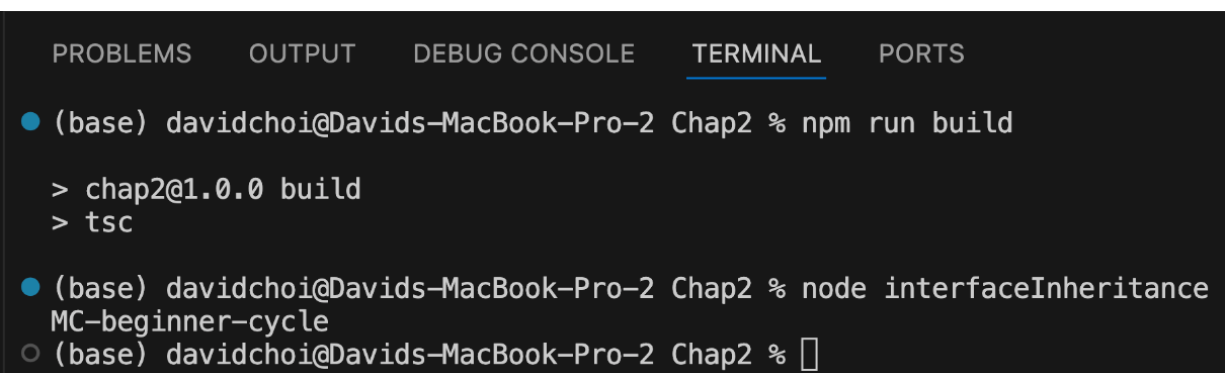
After the namespace, you can see that there is an interface called `Thing`, and after that, the `Vehicle` interface is defined, and it inherits from `Thing` using the `extends` keyword. I put this into the example to show that interfaces can also inherit from other interfaces. The `Thing` interface has two members: `name` and `getFullName` —and as you can see, although `Vehicle` extends `Thing`, there is no mention of those members anywhere inside of `Vehicle`. This is because `Vehicle` is an interface and therefore cannot have any implementation. Now, if you look at the following code, you will see that the `Motorcycle` class uses the



implements keyword to define implementations of the Vehicle interface's members. Replace the previous code's comment with this:

```
class Motorcycle implements Vehicle {
  name: string;
  wheelCount: number = 0;
  constructor(name: string) {
    // no super for interfaces
    this.name = name;
  }
  updateWheelCount(newWheelCount: number){
    this.wheelCount = newWheelCount;
    console.log(`Automobile has ${this.wheelCount}`);
  }
  showNumberOfWheels() {
    console.log(`moved Automobile ${this.wheelCount} miles`);
  }
  getFullName() {
    return "MC-" + this.name;
  }
}
const moto = new Motorcycle("beginner-cycle");
console.log(moto.getFullName());
}
```

Notice how by implementing the Vehicle interface, our Motorcycle class must implement both the members of Vehicle and Thing. This occurs because Vehicle extends Thing. So, if we compile and run this code, we get the following:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node interfaceInheritance
MC-beginner-cycle
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.14 – The interfaceInheritance result*

Interfaces do not provide a means to do code reuse directly, as they have no implementation. However, it is still advantageous for code reuse because the structure of interfaces provides type-safe expectations around what code will

receive and return. Hiding the implementation behind an interface is also beneficial in terms of doing encapsulation and abstraction, which are also important principles of OOP. In this section, we learned about inheritance and how it can be used for code reuse. We learned about how to do inheritance with the three major container types: classes, abstract classes, and interfaces. In the next section, we will cover generics.

## Understanding polymorphism

**Polymorphism** is a bit of an intimidating name as it's not immediately clear what it means. However, it's actually quite powerful because it allows us to maintain type safety in our code while still being able to use different code implementations as needed. Let's look at some code and see how this works. Create a file called `polymorphism.ts` and add the following code. This is going to be a fair bit of code, so let's go through it in pieces:

```
interface Animal {  
  name: string;  
  runMaxMiles(hours: number): number;  
}
```

First, we've created an interface that shows an `Animal` type. This type has a name and a method signature of `runMaxMiles`. Obviously, since this is an interface, our intention is to implement this interface with specific capabilities, so let's do that now:

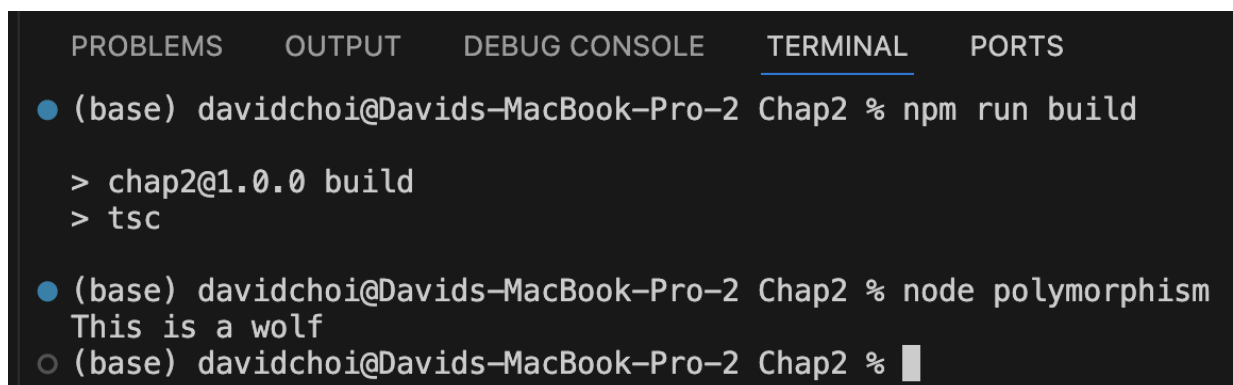
```
class Wolf implements Animal {  
  name: string = "";  
  runMaxMiles(hours: number): number {  
    return hours * 45;  
  }  
}  
  
class Cheetah implements Animal {  
  name: string = "";  
  runMaxMiles(hours: number): number {  
    return hours * 75;  
  }  
}
```

Here, we've created two distinct implementations: `Wolf` and `Cheetah`. Each one has a certain distance it can cover within a given time period—in other words,

miles per hour. Let's continue by adding the portion of our code that displays the polymorphism:

```
const hours = 0.5;
function pickTheBestAnimalToRun(hours: number): number {
  let animal: Animal | undefined;
  if (hours >= 0.5) {
    animal = new Wolf();
    animal.name = "wolfie";
  } else {
    animal = new Cheetah();
    animal.name = "cheetos";
  }
  if (animal instanceof Wolf) {
    console.log("This is a wolf");
  }
  if (animal instanceof Cheetah) {
    console.log("This is a cheetah");
  }
  return animal.runMaxMiles(hours);
}
pickTheBestAnimalToRun(hours);
```

So, what have we done in our `pickTheBestAnimalToRun` function? First, we created an animal variable of the `Animal` type. Notice that although an animal has a type, it is not initially set to anything. Next, we have a condition, and this is where the polymorphism is happening. This condition looks at the number of hours needed to be run and selects the appropriate animal to run for that duration. Since a wolf, although slower than a cheetah, is generally capable of running longer distances if the duration to be run is more than half an hour (0.5 hours), it will be selected and initialized over a cheetah. Let's build and run this code and see what happens:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build
> chap2@1.0.0 build
> tsc
● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node polymorphism
This is a wolf
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

*Figure 2.15 – The polymorphism result*

As you can see, our code runs and the `pickTheBestAnimalToRun` function selects the wolf since our `hours` value was `0.5`. Next, let's learn about **generics**.

## Learning generics

**Generics** allow a type definition to include an associated type that can be chosen by the user of the generic type, instead of being dictated by the type creator. In this way, there are structures and rules, but still some amount of flexibility. Generics will definitely come into play later when we code with React, so let's learn about them here. Generics can be used for functions, classes, and interfaces. Let's look at an example of generics with functions. Create a file called `functionGeneric.ts` and add the following code:

```
function getLength<T>(arg: T): number {
    if(arg.hasOwnProperty("length")) {
        return arg["length"];
    }
    return 0;
}
console.log(getLength<number>(22));
console.log(getLength("Hello world."));
```

Note that this code has errors, but we'll work through those together. If we start at the top, we see a function called `getLength<T>`. This function uses a generic, `<T>`, that tells the compiler that wherever it sees the `T` symbol, it can expect some associated type. Now, our function implementation checks to see whether the `arg` parameter has a field called `length`, by using the `hasOwnProperty` function. This function is built into JavaScript, and as you can see, it checks whether a certain property exists. If it does not have a `length` value, it just returns `0`. Finally, toward the bottom, you can see that the `getLength` function is called two times: once for a number and another time for a string. Additionally, you can see that for `number`, it explicitly has the `<number>` type indicator, whereas for `string`, it does not. These two examples are there only to show that you can be explicit about what the associated type is, but the compiler can usually figure out which type you meant based on the usage. Now, the first issue with this code is the errors. What's going on here is that TypeScript has received what is effectively an unknown type (i.e., `T`). Yes, we know it will be of some type, but we don't know which specific type it is and therefore we don't know if that type will even have a `hasOwnProperty` function. But in addition to this issue, we see there is another problem on the next line. In JavaScript, object properties can be accessed either with dot notation or with bracket notation. So, if `arg` was an

object, this syntax would work, but again, the compiler can't know that for sure right now. So, let's update this code to eliminate these errors. First, comment out the code we just wrote and add the following new code below it:

```
interface HasLength {
    length: number;
}
function getLength<T extends HasLength>(arg: T): number {
    return arg.length;
}
console.log(getLength<number>(22));
console.log(getLength("Hello world."));
```

This code is quite similar, except we use a `HasLength` interface to constrain what types are allowed. Constraining generic types is done with the `extends` keyword. By writing `T extends HasLength`, we are telling the compiler that whatever `T` is, it must inherit from or be of the `HasLength` type, which effectively means that it must have the `length` property. Therefore, when the two previous `getLength` calls are made, it fails for `number` types, since they don't have a `length` property, but it works for `string`. OK, now let's look at an example that uses interfaces and classes together. Let's create a file called `classGeneric.ts` and add the following code to it:

```
namespace GenericNamespace {
    interface Wheels {
        count: number;
        diameter: number;
    }
    interface Vehicle<T> {
        getName(): string;
        getWheelCount: () => T;
    }
    // more code coming here
}
```

So, we can see that we have an interface called `Wheels`, which provides wheel information such as `count` and `diameter`. We can also see that the `Vehicle` interface takes a generic of type `T`. You can probably guess what we're going to do, but let's continue by adding another type, `Automobile`, and replace the comment line with it:

```
class Automobile implements Vehicle<Wheels> {
    constructor(private name: string, private wheels: Wheels){}
    getName(): string {
        return this.name;
    }
}
```

```

    }
    getWheelCount(): Wheels {
        return this.wheels;
    }
}

```

We see that the `Automobile` class implements the `Vehicle` interface with the generic as the `Wheel` type, which gives an implementation to the `getName` and `getWheelCount` methods. Then, finally, let's add another class into the namespace just below our `Automobile` class called `Chevy`:

```

class Chevy extends Automobile {
    constructor() {
        super("Chevy", { count: 4, diameter: 18 });
    }
}

```

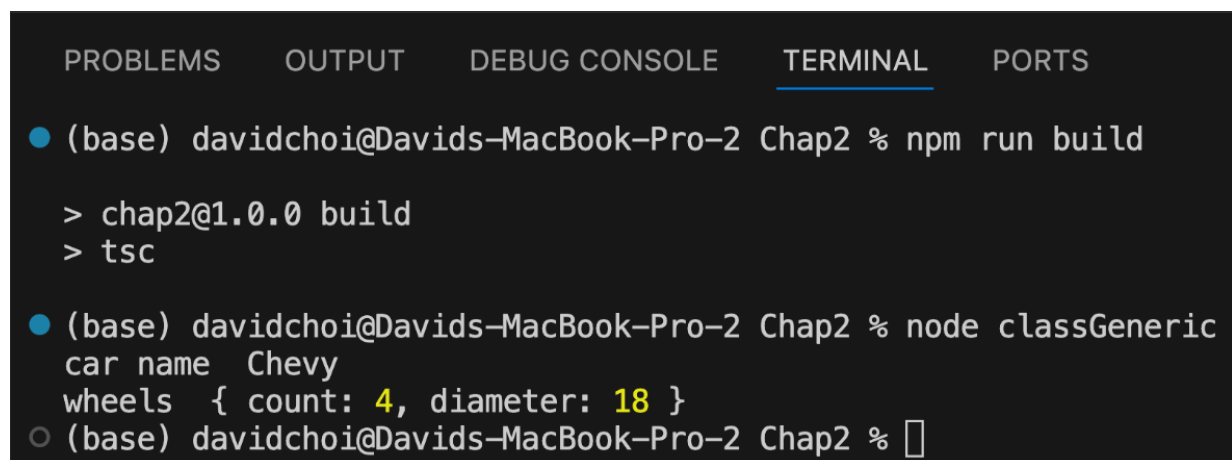
In the case of the `Chevy` class, we do not need to implement anything because we are directly inheriting from `Automobile` and, therefore, receiving its methods without having to also create them inside of `Chevy`. After all these types are defined, add this code after the `Chevy` class:

```

const chevy = new Chevy();
console.log("car name ", chevy.getName());
console.log("wheels ", chevy.getWheelCount());

```

The last three lines create an instance of the `Chevy` class, and then its methods are called inside the console logs. If you compile and run, you should see this:



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node classGeneric
car name  Chevy
wheels  { count: 4, diameter: 18 }
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % 

```

Figure 2.16 – The `classGeneric.ts` result

You can see that our inheritance hierarchy is several levels deep, but our code is able to successfully return a valid result. In this section, we learned about using generics on both functions and class types. Generics are important as they are commonly used in React development, as we'll see soon.

## Utility types

**Utility types** are created by the TypeScript team and provide pre-built solutions for common use cases in TypeScript. In this section, we'll learn about a few of the more commonly used utility types.

### ReturnType<Type>

`ReturnType` is probably one of the most frequently used utility types. As the name implies, `ReturnType<Type>` allows us to create a type from the type a function returns. Let's create an example. Create the `returnType.ts` file and add this code to it:

```
function getData() {
  return [
    {
      name: "jon",
      age: 24,
    },
    {
      name: "linda",
      age: 35,
    },
    {
      name: "tom",
      age: 21,
    },
  ];
}
```

As you can see, this function returns person data as an array. It does not have an explicit return type, so we want to type that explicitly. Let's add some more code to create that explicit type. Add this underneath the function:

```
type Result = ReturnType<typeof getData>;
```

Let's unpack this code. First, we start with a type alias called `Result`. Then, we make our `ReturnType` call and add a type in between the arrow brackets. The code `typeof getData` tells TypeScript to get the return type of the `getData` function and then associate it with `ReturnType`. The `ReturnType` is a generic. Note that you will learn more about `typeof` in *Chapter 3, Building Better Apps with ES6+ Features*. This line of code makes our `Result` type look like this:

```
{
  name: string;
  age: number;
}[]
```

Notice that it is an array type, as indicated by the square brackets. Next, let's add the code that will call our `getData` function and view its results:

```
const result: Result = getData();
console.log(result);
```

As you can see, we have a variable, `result`, of type `Result` that receives the `getData` function's return value, and then we log it on the console. If you compile and run this code, the result will look like this:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % npm run build

> chap2@1.0.0 build
> tsc

● (base) davidchoi@Davids-MacBook-Pro-2 Chap2 % node returnType
[
  { name: 'jon', age: 24 },
  { name: 'linda', age: 35 },
  { name: 'tom', age: 21 }
]
○ (base) davidchoi@Davids-MacBook-Pro-2 Chap2 %
```

Figure 2.17 – The `returnType.ts` result

## Pick<Type, Keys>

Sometimes, when dealing with things such as an API, we may find that the type has more information than we actually need. This may make dealing with that



type inefficient. Therefore, this utility type allows us to be selective about which properties we care about and only populate our type with those properties. Let's create an example. Create a new file, `pick.ts`, and add this code:

```
interface SuperComplexType {
  name: string;
  age: number;
  street: string;
  state: string;
  zip: string;
  employeeId: number;
  dateStarted: Date;
  favoriteFood: string;
  favoriteColor: string;
  favoriteSportsTeam: string;
  homeTown: string;
  corporateOfficeCity: string;
}
```

Now, let's pretend that the API we need to call is returning an object that has this type of information. Clearly, if we didn't need every single field here, it would be cumbersome to have to deal with such a type. So, let's create our own type with just the fields we need by using `Pick`. Add this code below `SuperComplexType`:

```
type SimpleType = Pick<
  SuperComplexType,
  "name" | "age" | "corporateOfficeCity"
>;
const adam: SimpleType = {
  name: "adam",
  age: 33,
  corporateOfficeCity: "New York",
};
```

Now, starting at the `SimpleType` line, you can see that we use the `Pick` utility type, pass our original `SuperComplexType`, and then select each field that we want in our new type. If you hover over this new `SimpleType`, you will see that it only includes the fields we selected. The `adam` variable was added just to give us confirmation that this new type works as expected.

## Omit<Type, Keys>

Now, if there's a type for including only the properties you want, you would think there's a type for removing the ones you don't. There is: `Omit<Type, Keys>`.

Let's revisit the `Pick` example we just used and revise it for `Omit`. Create a file called `omit.ts` and add this code:

```
// copy SuperComplexType here
type SimpleTypeOmit = Omit<
  SuperComplexType,
  "name" | "age" | "corporateOfficeCity"
>;
```

Make sure to copy the `SuperComplexType` interface to the top of your `omit.ts` file. If you hover over the `SimpleTypeOmit` type, it will show you a list of all the properties inside of `SuperComplexType` except the three properties you included in the `Omit` type. There are many utility types that you can choose from to provide helpers to improve and streamline your code. These utility types are created by the Microsoft TypeScript team. So, you know they're high quality. Therefore, before rolling your own helper, you should check the documentation and see whether there is a suitable type that already exists and use that.

## Summary

In this chapter, we learned about the TypeScript language. We learned about the many different types that exist in the language and also how to create our own types. We also learned how to use TypeScript to create object-oriented code. It was a large and complex chapter but will be useful knowledge for when we begin building our app. In the next chapter, we will review some of the most important features of JavaScript. We will also learn about some of the newer features in the later versions of the language. Since TypeScript is a true superset of JavaScript, it is important to have an up-to-date understanding of JavaScript in order to make maximal usage of TypeScript.