5 TypeScript: Next-level JavaScript

This chapter covers

- Using best practices with React and TypeScript
- Introducing static typing with TypeScript
- Reducing errors and increasing documentation with types

As we venture further into the realm of professional React development, we come across a pivotal tool in our arsenal: TypeScript. This chapter marks a significant step in enhancing your skills and understanding as a developer. TypeScript, often referred to as "JavaScript with superpowers," opens the door to a new world of possibilities in terms of robust, statically typed code and cleaner, more maintainable applications.

Over the past few years, TypeScript has gained widespread adoption in the JavaScript community. Its ability to catch bugs at compile time, provide improved code navigation and autocompletion, and enhance collaboration between developers has made it a go-to choice for building modern web applications. In this chapter, we will embark on a journey to understand TypeScript's fundamentals, its advanced features, and its application within the React ecosystem.

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch05**.

5.1 The importance of TypeScript

Trying to introduce TypeScript in a few chapters is like trying to explain all of calculus in a single math class. TypeScript is a huge topic, well deserving of complete books, and a lot of great literature about it is already available.

TypeScript is the bigger sibling of JavaScript. TypeScript can do everything JavaScript does, but it also does much more—namely, static typing and type safety checks. See figure 5.1 for an overview. TypeScript can be transpiled to JavaScript, losing all its additional functionality in the conversion, but as static typing and type safety checks can be validated before the code runs, you still get all the value that you can from it, while producing JavaScript in the end.

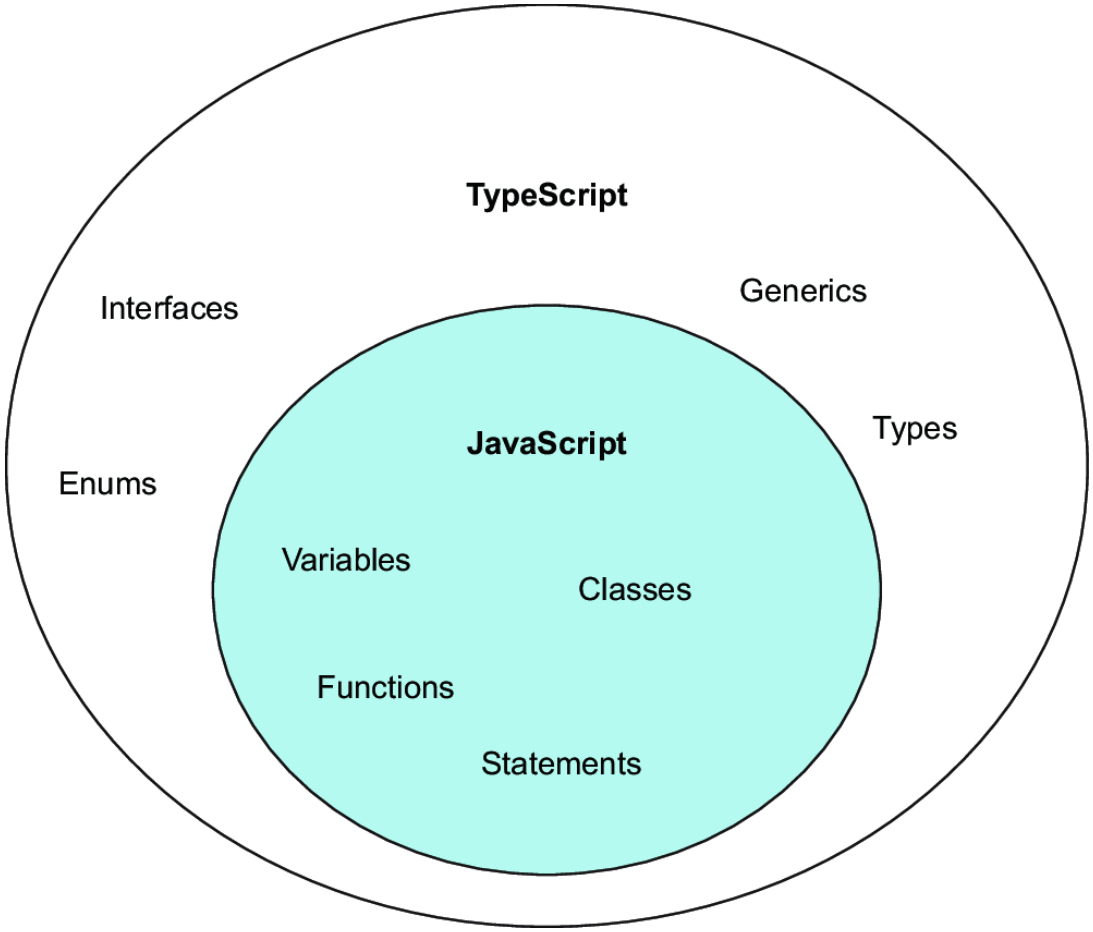

Figure 5.1 TypeScript contains all the features that JavaScript does, including variables, functions, classes, and statements. It also contains new features, including types, enums, interfaces, and generics.

Many React-based development teams have moved to exclusive use of TypeScript, which is the future of JavaScript in general and React in particular. That said, a ton of teams still aren't using it, and you can get by fine without knowing it, but your options will expand greatly if you add TypeScript to your skill set.

Before I proceed, it's important to note that our exploration of TypeScript and its integration with React is extensive, spanning two chapters. In this chapter, we'll delve into TypeScript's fundamental and advanced fea-

tures in the context of React applications, setting the stage for our discussion of integrating TypeScript with React in chapter 6. So get ready to bolster your skills as we embark on this journey across TypeScript's terrain. While we're focusing on React, it's important to note that all these principles apply outside the world of React, making this knowledge universally valuable.

Whether you're a seasoned React developer or just getting started, the knowledge you gain in this chapter and the next one will be invaluable. TypeScript empowers developers to write more reliable, scalable, and maintainable code. It reduces the likelihood of runtime errors and simplifies collaboration on larger codebases. I'll take you through TypeScript's core concepts, its integration with React, advanced techniques that use generics, and the practical application of TypeScript in real-world scenarios.

By the end of this chapter, you'll not only have the knowledge to work effectively with TypeScript but also understand its potential pitfalls and how to navigate them. So let's dive into the world of TypeScript, where the JavaScript of tomorrow meets the challenges of today's professional web development.

5.2 Introduction to TypeScript

TypeScript is your bridge to a world where code becomes more predictable and maintainable and less error prone. We'll dive deep into understanding the basics, starting with TypeScript files and their seamless integration with React, giving you the essential knowledge to bring static typing to your applications.

Next, we'll explore TypeScript's type system, an indispensable aspect of the language. I'll demystify static types and show you how to apply them effectively to your code. By the end of this section, you'll have a firm grasp of TypeScript's core concepts, enabling you to use its robust features to enhance the reliability and maintainability of your React applications. Let's begin your TypeScript journey and unlock the full potential of JavaScript in the professional development landscape.

5.2.1 TypeScript files and React

Before we get started with the details, let's briefly discuss file extensions. Although it's not a hard requirement, it's a common practice to save TypeScript in `*.ts` files rather than `*.js` files to signify that they contain source code in a (slightly) different language. Furthermore (and I haven't discussed this fact previously), some developers put React components (or files using JSX) in `*.jsx` files rather than plain `*.js` files, and those same people would put React TypeScript components in `*.tsx` files rather than `*.ts` files. Many TypeScript + React setups require you to do so and will fail if you put JSX in `*.ts` files. We will be using `*.ts` files for TypeScript without JSX and `*.tsx` files for TypeScript with JSX in this section for those very reasons.

5.2.2 Static types

The most basic feature of TypeScript is static typing. Let's start with an example, creating a single function intended to calculate the next number in the Fibonacci sequence by passing the two previous numbers to it:

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

If we implement this function in TypeScript, the type checker doesn't know the type of the two arguments, so it is unable to infer the return value of the response. Thus, we can call this function with whatever we want, and TypeScript won't help us:

```
const result = fibonacci(true, "Hi there");
```

But TypeScript will alert us that type information is missing, saying `Parameter 'a' implicitly has 'any' type`). For TypeScript to function, it needs a bit of information to get started. In particular, we need to specify the types of arguments passed to functions. We do that using a colon (`:`) followed by the type name. Primitive types' names can be lowercase (`number`) or capitalized (`Number`), but the names of complex

types are generally capitalized. In this example, we need to specify that `a` and `b` are numbers:

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

Now, if we try to use the function as we did before, TypeScript will come to our assistance:

```
const result = fibonacci(true, "Hi there");
```

TypeScript reports that an `Argument of type 'boolean' is not assignable to parameter of type 'number'.` If we change the first argument to a number, TypeScript will report an error for the second parameter. If we use the function correctly, however, TypeScript will correctly infer the types of all variables:

```
const f0 = 0;
const f1 = 1;
const f2 = fibonacci(f0, f1);
const f3 = fibonacci(f1, f2);
```

Note that we don't have to add type information for any of these variables directly. We could have written `const f0: number = 0`, but we didn't have to. TypeScript knows that `0` has the type `number`. Also, we don't have to write that `f2` has a type `number`, as TypeScript will correctly infer that the type returned from the Fibonacci function has the type `number`. TypeScript knows what all the built-in functions do, so it knows that the result of adding two numbers is itself a number. Similarly, it knows that the length of a string is a number and that the `number.toString()` method returns a string. For more complex functions, we can add type hints about what a function is supposed to return; TypeScript will validate that it does in fact return only that type.

If a variable can hold multiple types of values, you can specify that fact too. You can create your own type, which indicates exactly which values

are allowed. If you have a first-name variable that starts as null but can take a specific value later, you can specify that situation like so:

```
type Name = null | string;
let firstName: Name = null;
```

You can even specify that a value can be any type, but that approach is almost always a bad choice. Instead, use the literal type `any`. If you want to indicate that you don't know the type of a given value (maybe it's returned by a third-party API), you can use the type `unknown`. Also, you can specify specific values allowed for a type, such as a status variable that can be only `LOADING`, `SUCCESS`, or `ERROR`:

```
type Status = "LOADING" | "SUCCESS" | "ERROR";
let status: Status = "LOADING";
```

If we later try to assign the value `status = "COMPLETE"`, TypeScript will report this assignment as an error because that value is not valid:

```
Type '"COMPLETE"' is not assignable to type 'Status'.
```

When we're adding type information to a functional React component, we create a custom complex type, specifying the name and types of the properties passed to that function this way:

```
type PersonProps = {
  name: string;
  age: number;
};
function Person({ name, age }: PersonProps) {
  ...
}
```

We can also use an interface rather than a type, which is common practice for props definitions. A slight technical difference between interfaces and types exists, but we won't cover that difference here. Just know that you'll often see the previous example written this way instead:

```
interface PersonProps {
  name: string;
  age: number;
}
function Person({ name, age }: PersonProps) {
  ...
}
```

Note the slightly different notation for an interface in that it doesn't use an assignment (no equals sign). If you have optional properties, you indicate that fact by adding a question mark in the type or interface for the properties object. In the following example, both `name` and `age` are optional, and we assign a default value to the `name` when destructuring the properties but not to the `age`:

```
interface PersonProps {
  name?: string;
  age?: number;
}
function Person({ name = "Anonymous", age }: PersonProps) {
  ...
}
```

5.2.3 Employee display

Let's create an about page listing the employees of a fictional company. For that task, we need a simple component to display a single employee using TypeScript. We want to display this component in a card-like style, as a rounded rectangle with a bit of drop shadow. All these things come together in the file `EmployeeCard.tsx`, where we define the main employee-card component. You can see this file in the following listing.

Listing 5.1 An employee card using TypeScript

```
import "./employee.css";    #1
type Employee = {     #2
  name: string;
  title: string;
};
interface EmployeeCardProps {   #3
  item: Employee;    #4
}
function EmployeeCard({ item }: CardProps) {   #5
  return (
    <section className="employee">
      <h2 className="employee__name">
        {item.name}     #6
      </h2
      <h3 className="employee__title">
        {item.title}    #6
      </h3>
    </section>
  );
}
export default EmployeeCard;
```

#1 We import some styles to make the card look good.

#2 First, we define a type for an employee. (You might have that type somewhere central in your application.)

#3 Then we define the interface for this specific component, which takes an item of the employee type.

#4 Note that this property is required, as we don't include a question mark in the definition.

#5 We apply this interface to the object of properties received in the component.

#6 We can use values on the item object validated by TypeScript because of the type information we've added to the application.

We also have some supporting files to make this application work with `App.tsx`, `app.css`, and `employee.css`, as you can see in the next three listings.

Listing 5.2 The main application displaying a single employee card

```
import EmployeeCard from "./EmployeeCard";
import "./app.css";
function App() {
  return (
    <main>
      <EmployeeCard item={{    #1
        name: "Willy Wonka",   #1
        title: "Candy King"    #1
      }} />    #1
    </main>
  );
}
export default App;
```

#1 Here, we use our employee-card component. Note that we don't need to specify that this item object is of the correct type. TypeScript checks the object and validates that it has the required values and types—and only those values and types.

Listing 5.3 Base styles for our employee page

```
@import url("https://fonts.googleapis.com/css2?family=Open+Sans:wght@300 ");
* {
  box-sizing: border-box;
}
body {
  background-color: #eee;
  font-family: "Open Sans", sans-serif;
}
main {
  display: flex;
  align-items: center;
  justify-content: center;
  margin: 2em;
}
```

Listing 5.4 Styles specific to the employee card

```
.employee {
  background: white;
  flex: 0 0 300px;
  border-radius: 1em;
  padding: 1em 1.5em;
  box-shadow: 4px 4px 2px rgba(0 0 0 / 0.2);
}
.employee__name {
  margin: 0 0 0.5em;
  font-size: 24px;
}
.employee__title {
  margin: 0;
  text-transform: uppercase;
  font-size: 16px;
  color: #222;
}
```

What does this code look like in a browser? Well, as expected, it looks like a nicely styled employee card (figure 5.2).

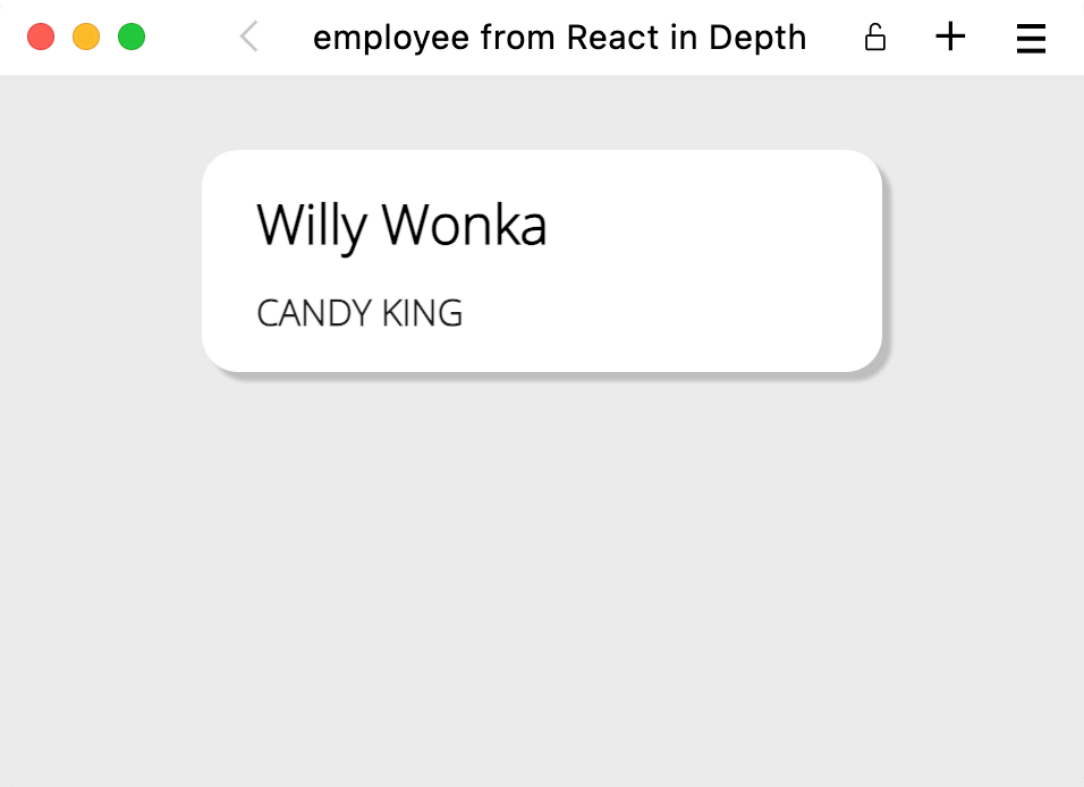

Figure 5.2 Our employee card in action. Nobody can tell from the screenshot that we used TypeScript behind the scenes, and they're not supposed to, but *we* know, so we appreciate the result a bit more. Also, Willy Wonka is king here.

EXAMPLE: EMPLOYEE

This example is in the `employee` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch05/employee
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch05-employee**.

5.2.4 Optional properties

Let's extend the example a bit by adding an optional website property to the employee type and then display it in the component if present. We could add that property as shown in the following listing.

Listing 5.5 An employee card with an optional website link

```
import "./employee.css";
export type Employee = {
  name: string;
  title: string;
  website?: string;   #1
};
interface EmployeeCardProps {
  item: Employee;
}
function EmployeeCard({ item }: CardProps) {
  return (
    <section className="employee">
      <h2 className="employee__name">{item.name}</h2>
      <h3 className="employee__title">{item.title}</h3>
      {item.website && (     #2
        <h4 className="employee__link">    #2
          Web: <a href={item.website}>{item.website}</a>   #2
        </h4>   #2
      )}    #2
    </section>
  );
}
export default EmployeeCard;
```

#1 Adds the optional property with a question mark

#2 Renders the optional property using a logical AND expression

We can use this new component to display two different employees—one with a website and one without a website—on the about page.

Listing 5.6 A new main application displaying two employees

```
import EmployeeCard, { type Employee } from "./EmployeeCard";
import "./app.css";
function App() {
  const employees: Employee[] = [
    {
      name: "Wayne Campbell",     #1
      title: "Host Extraordinaire",   #1
      website: "https://extremepartytime.com",    #1
    },
    {
      name: "Garth Algar",     #2
      title: "Tech Wizard",    #2
    },
  ];
  return (
    <main>
      {employees.map((employee) => (    #3
        <EmployeeCard    #3
          key={employee.name}    #3
          item={employee}   #3
        />    #3
      ))}    #3
    </main>
  );
}
export default App;
```

#1 Defines Wayne with his cool website. (Note: this site isn't Wayne's, but check it out anyway!)

#2 Defines Garth without a website

#3 Renders a list of employees using the map function

Rendering these two cool characters should look like figure 5.3.

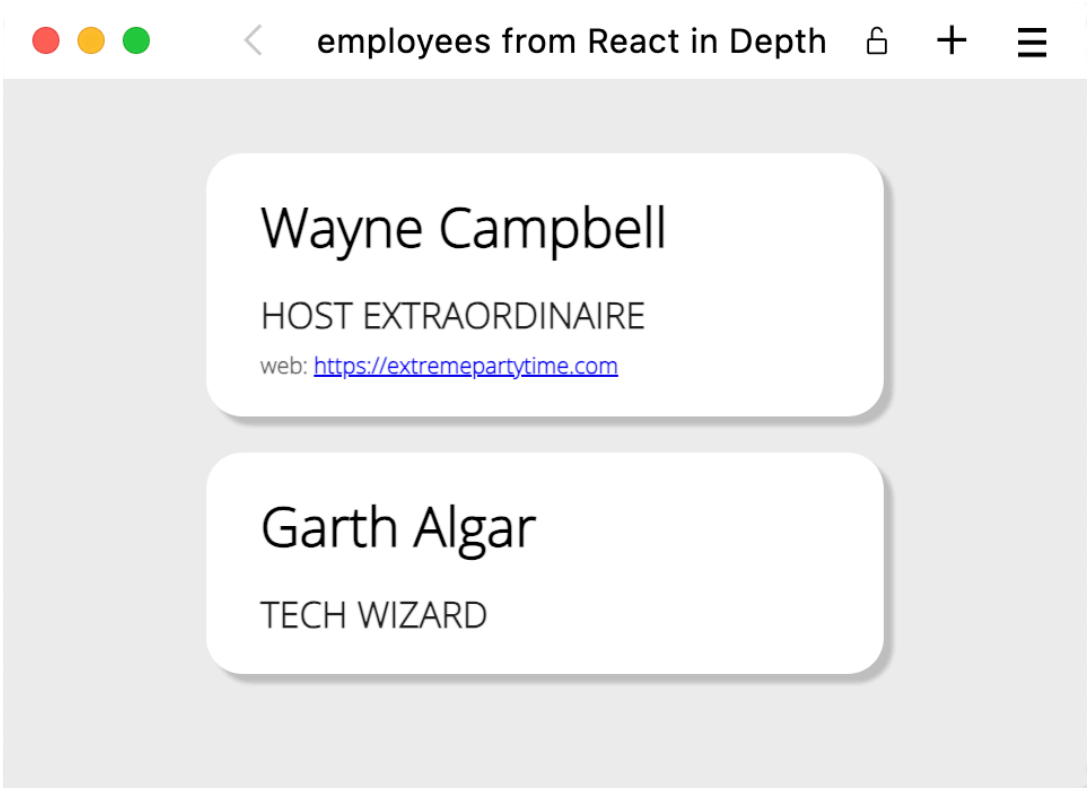

Figure 5.3 Rendering two infamous employees as card components. Only the first shows the optional website link. Go ahead and click it! I dare you!

EXAMPLE: EMPLOYEES

This example is in the `employees` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch05/employees
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch05-employees**.

5.3 Advanced TypeScript with generics

Generics is a well-known concept in the world of static typing, known from other, more complex, strongly typed languages such as Java and C#. A *generic type* takes another type as an argument. An array can be a generic type, for example; it's a list of something, but what it contains is unspecified until we *do* specify it. A number array is an instantiation of this generic type, with the array specifically said to contain numbers and only numbers.

Generics are used for all sorts of tricks in TypeScript and are especially important in React as well. In this section, we'll go over a bunch of important generic types that are built into TypeScript and React and that you'll often use in applications. Also, I'll go over some best practices that many React TypeScript teams use every day while creating large React applications:

- Typing `children`
- Extending interfaces
- Spreading props
- Optional and required properties
- Either/or properties
- Forwarding refs and memoizing components
- Typing hooks (covered in chapter 6)

Finally, we can make our own generics—not just for functions, but for complete React components. We can make a React component to take several properties that have to be in sync but aren't specified from the start. This process is fairly complex, so we'll save that task for section 5.4.

5.3.1 Understanding generic types

Let's create a function that returns the middle element of an array and should work with any array. As we saw in section 5.2.2, we can make it untyped, which is the same as typing it as `any`:

```
function getMiddle(list) {
  const mid = Math.floor(list.length / 2);
  return list[mid];
}
```

We can invoke this function on a list of numbers and store the result in a string, and it would compile:

```
const one: string = getMiddle([0, 1, 2]);
```

But we can see that this example is wrong; it's a list of numbers, so of course the middle item will also be a number, not a string. We can sig-

nify this fact by making the `getMiddle` function *generic*—that is, we can say that the function works with some type (which we may or may not put restrictions on), and then we can use this generic type to say that the function accepts an array of that type and returns a value of that type. The type can change with each invocation. To accomplish this task, we use angle-bracket notation (`<Type>`):

```
function getMiddle<Type>(list: Type[]): Type {
  const mid = Math.floor(list.length / 2);
  return list[mid];
}
```

Now, if we try the same thing as before, we get an error:

```
const one: string = getMiddle([0, 1, 2]);
```

TypeScript reports an error with the `one` variable:

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

If we assign the type to a properly typed variable, it will work. Then we can invoke the function with different types of arrays and get different types out:

```
const one: number = getMiddle([0, 1, 2]);
const apple: string = getMiddle(['pear', 'apple', 'banana']);
```

This code works as expected and is powerful, also in React. Note that in the invocations of `getMiddle`, the generic type argument is *inferred* by the value arguments to the function. We can also specify that argument explicitly, but we rarely have to. This code works as it did before:

```
const one: number = getMiddle<number>([0, 1, 2]);
```

If we specify a more specific set of types as the type argument, TypeScript loosens the definition of the return type. If we specify manu-

ally that we're invoking the function with an array of numbers or strings, we need to type the return value specifically as `number | string` to make the type system happy:

```
const one: number | string = getMiddle<number | string>([0, 1, 2]);
```

Note that we introduced this complexity ourselves. I use this example only to explain what would happen. We'll specify type arguments manually in later examples because the inferred type is not specific enough or because that approach makes typing easier.

5.3.2 Typing children

Let's create a heading component. We will use this component on a fictional website to display identical headings on all the pages, and we can easily update the headline styling in a single central place. Let's create the component without TypeScript first, using plain JavaScript:

```
function Heading({ children }) {
  return <h1 style={{ backgroundColor: "hotpink" }}>{children}</h1>;
}
```

That component is neat and simple; we know how to create it. We use the special property `children` to allow the component to be used as `<Heading>Title here</Heading>`. But how do we type this property? In this case, the heading of a page is probably a string, so let's type it as a string:

```
interface HeadingProps {
  children: string;
}
function Heading({ children }: HeadingProps) {
  return <h1 style={{ backgroundColor: "hotpink" }}>{children}</h1>;
}
```

That code seems fine, right? Well, yes. If we use this component like `<Heading>Hello</Heading>`, it works well. But what if we want to in-

clude dynamic values like `<Heading>Hello {name}</Heading>`? Now we get an error from TypeScript:

```
Type 'string[]' is not assignable to type 'string'.
```

That error does kind of make sense. Now we pass two string children to the component; the first is the string `"Hello "`, and the second is the value of the variable `name`. So we can modify the type to accept either a string or an array of strings like so:

```
interface HeadingProps {
  children: string | string[];
}
```

On one page, however, we need to use the heading to display a dynamic number, and at a different place. we might pass in a Boolean by using a logical `AND` expression, as in these two examples:

```
<Heading>You have {messages.length} new messages</Heading>
<Heading>There are {isValid && 'no'} errors</Heading>
```

Both examples are classic React patterns, showing how we pass things as child nodes to other components. But both examples fail with new errors:

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

Again, those errors make a lot of sense. Let's amend the type to accept strings, numbers, Booleans, and even arrays of those types:

```
interface HeadingProps {
  children:
    | string
    | number
    | boolean
    | (string | number | boolean)[];
}
```

We have everything covered now, right? Let's say that we want to include some special formatting, such as a bold word, as in `<Heading>Hello <strong>World</strong></Heading>`. Then we get a new error:

```
Type 'Element' is not assignable to type 'string | number | boolean'.
```

We need to add one more thing: `JSX.Element`. What if we don't pass in anything, as in `<Heading />`? We also need to allow the property to be missing by making it optional. Combining all those changes, we arrive at this code:

```
interface HeadingProps {
  children?:
    | string
    | number
    | boolean
    | JSX.Element
    | (string | number | boolean | JSX.Element)[];
}
```

This example is getting out of hand, and we haven't considered a few more edge cases yet. Luckily, React has an exported type that includes all these cases *and* the extra ones we haven't discussed. That type, called `ReactNode`, can be imported from the React package as follows:

```
import { type ReactNode } from 'react';
interface HeadingProps {
  children?: ReactNode;
}
function Heading({ children }: HeadingProps) {
  ...
}
```

That code is a lot cleaner. We still have to make the property optional, but we don't have to enumerate all the different possibilities.

An easier approach is to use the built-in interface `PropsWithChildren` in React, which defines an interface that takes the `children` property as defined in the preceding example and nothing else. Using this interface, we can define the same component this way:

```
import { type PropsWithChildren } from 'react';
function Heading({ children }: PropsWithChildren) {
  ...
}
```

NOTE There's no real difference between typing the `children` property directly with `ReactNode` or having React type it automatically by using `PropsWithChildren`. I prefer the latter approach for brevity's sake, so I will use it throughout this book.

CHILDREN AND MORE

Now let's start using this heading component for both small and large headings. We'll add a new property, `isLarge`, that defaults to `false`, so we can specify when a particular heading has to be extra-prominent this way:

```
<Heading isLarge>This is important!</Heading>.
```

How do we add this new property to the props interface now that we've removed that interface and replaced it with a built-in one from React? Well, I didn't mention one thing about `PropsWithChildren`: it takes an

optional generic type argument. Yes, even generic type arguments can be optional. In this case, the type argument is the interface for all the other, non-`children` props that the component accepts, and we can use it like so:

```
import { type PropsWithChildren } from 'react';     #1
interface HeadingProps {     #2
  isLarge?: boolean;    #2
}    #2
function Heading({
  isLarge = false, children   #3
}: PropsWithChildren<HeadingProps>) {    #3
  return (  #3
    <h1 style={{
      backgroundColor: "hotpink", fontSize: isLarge ? '48px' : '36px'
    }}>
      {children}
     </h1>
  );
}
```

#1 Imports the PropsWithChildren type from the React package
#2 Defines an interface defining nonchildren props for the component
#3 Combines the two interfaces by passing the props interface as a generic type argument to PropsWithChildren

The default value for the `PropsWithChildren` interface is an empty props interface, as though we invoked it with `PropsWithChildren<{}>`.

NOTE The default type argument is true only in React 18 and later. In React 17 and earlier, the interface does not have a default type argument, and if you want to use it for a component that accepts only the `children` property, you have to write it as `PropsWithChildren<{}>`. That code seems a bit clunky, but it works.

RESTRICTING CHILDREN TO CERTAIN TYPES IS NOT POSSIBLE

You may wonder whether we can use the type of the `children` property to restrict which types of elements are allowed. What if we want to create a `<TableRow>` component, and the only allowed child nodes are `<TableCell>` nodes? Unfortunately, React and TypeScript do not give you this opportunity. Typing `children` as `TableCell[]` may seem to be

trivial, but it doesn't work for various reasons—first and foremost because JSX is syntactic sugar and doesn't exist in the eyes of the compiler. Unfortunately, we cannot use TypeScript that way. There's no clever workaround for now, so make sure to design your components so that it's clear what kind of child nodes are valid.

5.3.3 Extending interfaces

Suppose that we have a generic button in our UI library that can render a string as the button label. We also have an icon variant of the button that allows the component to be rendered with both an icon and a label. Let's say that our button takes several properties, such as `color` and `size`:

```
interface ButtonProps {
  color: "primary" | "secondary";
  size: "large" | "medium" | "small";
  onClick?: () => void;
  disabled?: boolean;
  className?: string;
}
function Button({ ... }: PropsWithChildren<ButtonProps>) {
  return (...);
}
```

Note that this example is partial; we don't implement the button. Now let's look at how we would create the icon button. If we didn't use TypeScript, we would define it like this:

```
function IconButton({ icon, children, ...rest }) {
  return <Button {...rest}>{icon} {children}</Button>;
}
```

That is, we extract only the `icon` and `children` properties from the passed properties, using them combined as the children of the button component, and sending all other properties to the button as they are.

But if we want to type this feature, how do we specify that this icon button can take these six properties: the five properties that the button takes

plus an icon, which is a React node? The naive way would be to specify all six properties again, but that approach seems like a bad idea from the start. It would work, of course, but the two definitions would be unsynchronized. If we update the button properties later, the icon button properties won't update automatically.

Instead, we can extend one interface when defining another one. To do so, we need a reference to the interface for the props of the original component. Let's assume that we exported it so we can import it:

```
import { Button, type ButtonProps } from './Button';     #1
interface IconButtonProps extends ButtonProps {     #2
  icon: ReactNode,    #2
}    #2
function IconButton(
  { icon, children, ...rest }: PropsWithChildren<IconButtonProps>
) {
  return <Button {...rest}>{icon} {children}</Button>;
}
```

#1 Imports the component and the type for its properties from a different file
#2 Defines the properties for the icon button by extending the button props interface and adding only the new property, icon

After using this icon button for a while, we find out that it works only with the large button style, so we don't want developers to try to specify a different button style—only the large one. Now we want the icon button properties to extend every property of the button except one. How do we do it? We omit the extraneous property, using the `Omit` interface from TypeScript itself:

```
import { Button, type ButtonProps } from './Button';
interface IconButtonProps
  extends Omit<ButtonProps, "size"> {     #1
  icon: ReactNode,
}
function IconButton({ icon, children, ...rest }: IconButtonProps) {
  return <Button size="large" {...rest}>    #2
    {icon} {children}
  </Button>;
}
```

#1 Extends the button properties except the size property
#2 Hardcodes the size property to large while making sure that the rest object will never contain a different definition of size

We can even omit multiple properties by passing in a union of multiple strings:

```
Omit<ButtonProps, "size" | "color">
```

We can also do the opposite: pick just a few properties, which might be easier than omitting several. Suppose that we want to extend only the `color` property from the button component. We could do that by omitting the `size`, `onClick`, `disabled`, and `className` properties. But it's easier to say that we pick only the `color` property like this:

```
Pick<ButtonProps, "color">
```

`Pick` is also a built-in interface provided by TypeScript. In all these examples, we extend the properties of another component by using a direct reference to an interface describing these properties. But what do we do if we don't have a reference to the interface for the properties—only the component? Cue next section (via a small detour).

5.3.4 Spreading props in general

In the preceding example, we made a component that extended another of our own components. But often, we make components that extend built-in components or even third-party components. Let's start with a

component that has an HTML element inside it. We want to pass properties from outside the component to the HTML element.
We're going to create a `UserImage` component. This component takes some properties that are specific to this type of image, such as the `name` and `title` of the user, but it also takes properties that a regular image tag takes, such as `src`, `alt`, `width`, and `height`. We want to do something like this:

```
interface UserImageProps
  extends RegularImageProps {     #1
  name: string;
  title: string;
}
function UserImage({ name, title, ...rest }: UserImageProps {
  return (
    ...
      <img {...rest} />     #2
    ...
  );
}
```

#1 Tries to extend a regular image component, but RegularImageProps doesn't exist
#2 The goal is to use the rest value this way and have TypeScript validate that only valid props are passed.

In this example, we use a regular `<img />` tag, and we want TypeScript to validate that we can pass, say, `src` to our `UserImage` component but not a property such as `source` that doesn't exist for an image. We could type all the possible properties, but that list would be huge. In React, more than 200 possible properties are allowed for most HTML elements (mostly because of event handlers such as `onClick`, `onFocus`, and dozens more). As in section 5.3.3, we know that there must be a smarter way. But this time, we don't have a convenient interface directly describing the allowed properties. The `RegularImageProps` interface doesn't exist.

There is a better way: `ComponentProps`, and it's built into React. `ComponentProps` is a generic interface for which we must provide a type argument. This type argument is the type of component for which

we want to know the properties' interface. In our example, that component is `img`, so we pass it in as a string:

```
import { type ComponentProps } from 'react';
interface UserImageProps extends ComponentProps<"img"> { ... }
```

We have a slight problem, however: this interface also allows the property `ref` for all components, but references aren't received as regular props inside the component (at least not before React 19). As a result, it doesn't make sense to type the props received by a component to include this property. We'll get back to typing references in section 5.3.8. A slightly longer interface, `ComponentPropsWithoutRef`, does exactly what it says:

```
import { type ComponentPropsWithoutRef } from 'react';
interface UserImageProps extends ComponentPropsWithoutRef<"img"> { ... }
```

Now we have our component fully functional, accepting the right props. Remember that we can also `Pick<>` or `Omit<>` from this interface to allow or disallow specific properties. Maybe we don't want the developer to pass in the `alt` property because we're going to set it ourselves:

```
import { type ComponentPropsWithoutRef } from 'react';
interface UserImageProps extends
  Omit<ComponentPropsWithoutRef<"img">, "alt"> {     #1
  name: string;
  title: string;
}
function UserImage({ name, title, ...rest }: UserImageProps) {
  return (
    ...
      <img
        alt={`Profile image for ${name}`}
        {...rest}    #2
      />
    ...
  );
}
```

#1 Extends all the properties of an image element except the alt property
#2 By not allowing the alt property, we can be sure that our custom alt property will not be overridden.

But what if we're extending a custom component? Can we still use the `ComponentPropsWithoutRef`? Suppose that we have a `<Rating />` component from a third-party library that takes properties like those in this example (and might take even more properties):

```
<Rating icon="♥" max={6} value={4.3} label="4.3 hearts" />
```

We want to create a new `BookReview` component. We want to be able to pass in some specific things about the book, as well as some of the same properties that the `Rating` component takes, such as `value`, `label`, and `icon`. We would like to do this:

```
import { Rating, type RatingProps }    #1
  from 'cool-rating-library';    #1
interface BookReviewProps extends
  Pick<RatingProps, "value" | "label" | "icon"> {     #2
  title: string;
  reviewer: string;
  body: string;
}
function BookReview({ ... }: BookReviewProps) {
  ...
```

#1 Imports a component and its type from a third-party library

#2 Creates a new type from the type of the external component

In this example, we depend on the external library to provide a type of the props of the component. What if we don't have it? Can we use `ComponentPropsWithoutRef`? We can't use it directly, as `ComponentPropsWithoutRef<Rating>` doesn't make sense. Here, `Rating` is a real JavaScript variable, not a TypeScript type. But we can do the naive thing and use `typeof`:

```
import { type ComponentPropsWithoutRef } from 'react';
import { Rating } from 'cool-rating-library';
type RatingProps =    #1
  ComponentPropsWithoutRef<typeof Rating>;    #1
interface BookReviewProps extends
  Pick<RatingProps, "value" | "label" | "icon"> {     #2
  ...
```

#1 typeof does exactly what it says: returns the type of a given variable.

#2 Now we can use the type extracted from the component.

Let's turn this code into a full-blown example so we can play around with it. We'll create both components ourselves, but we won't expose the props type from the `Rating` component. We'll also get to play with CSS to make a fancy rating display. First, we have the `Rating` component.

Listing 5.7 `Rating` **component**

```
import { type PropsWithChildren } from "react";
import "./Rating.css";
interface StarsProps {
  count: number;
  faded?: boolean;
}
function Stars({
  count,
  faded = false,
  children,
}: PropsWithChildren<StarsProps>) {
  return (
    <span className={`rating__stars ${faded && "rating__stars--faded"}`}>
      {Array.from(Array(count).keys()).map((_, i) => (
        <span key={i} className="rating__star">
          {children}
        </span>
      ))}
    </span>
  );
}
interface RatingProps {     #1
  icon?: string;    #1
  value: number;    #1
  max?: number;    #1
  label?: string;   #1
}    #1
export function Rating({
  icon = "⭐",
  value,
  max = 5,
  label = "",
}: RatingProps) {
  const percentage = Math.round((value / max) * 100);
  return (
    <div className="rating" title={label}>
      <Stars faded count={max}>
        {icon}
      </Stars>
      <div className="rating__overlay" style={{ width: `${percentage}%` }}>
        <Stars count={max}>{icon}</Stars>
      </div>
```

```
      </div>
    );
  }
```

#1 Defines the interface for the allowed props of the Rating component. Note that this interface is not exported, so it's not directly accessible from outside this file.

Then we have the `BookReview` component, which contains the `Rating` component.

Listing 5.8 `BookReview` component using the `Rating` component

```
import { ComponentPropsWithoutRef } from "react";
import { Rating } from "./Rating";
import "./BookReview.css";
type RatingProps =    #1
  ComponentPropsWithoutRef<typeof Rating>;    #1
type PickedRatingProps = Pick<     #2
  RatingProps, "value" | "max" | "icon"    #2
>;    #2
interface BookReviewProps    #3
  extends PickedRatingProps {    #3
  title: string;     #4
  reviewer: string;    #4
  body: string;    #4
}
export function BookReview({
  title,
  reviewer,
  body,
  ...rest
}: BookReviewProps) {
  return (
    <section className="review">
      <Rating max={5} {...rest} />
      <h3 className="review__title">{title}</h3>
      <h4 className="review__byline">By {reviewer}</h4>
      <p className="review__body">{body}</p>
    </section>
  );
}
```

#1 Creates a type RatingProps by extracting props of the Rating component

#2 Creates a type PickedRatingProps by picking specific props from RatingProps

#3 Creates the BookReviewProps interface by extending PickedRatingProps

#4 Adds props specific to the BookReview component

Finally, let's include some instances of the book review in our main application.

Listing 5.9 The main app with a few reviews

```
import { BookReview } from "./BookReview";
function App() {
  return (
    <div className="reviews">
      <BookReview     #1
        title="Great book"    #1
        reviewer="Anonymous"    #1
        body="I loved the book"    #1
        value={4.8}    #1
      />    #1
      <BookReview     #2
        title="Mediocre Sci-fi"    #2
        reviewer="Sci-fi Lover"    #2
        body="The aliens are boring."  #2
        value={3.3}    #2
        icon="👽"    #2
      />    #2
      <BookReview    #3
        title="It's a classic!"   #3
        reviewer="Hiro Protagonist"    #3
        body="The perfect romance novel."   #3
        value={9.2}    #3
        max={10}    #3
        icon="💔"   #3
      />    #3
    </div>
  );
}
export default App;
```

#1 Creates the first review using only the required properties

#2 Creates a review with a custom icon

#3 Creates a review with both a custom icon, using 10 icons instead of the default 5

The result should look like figure 5.4.

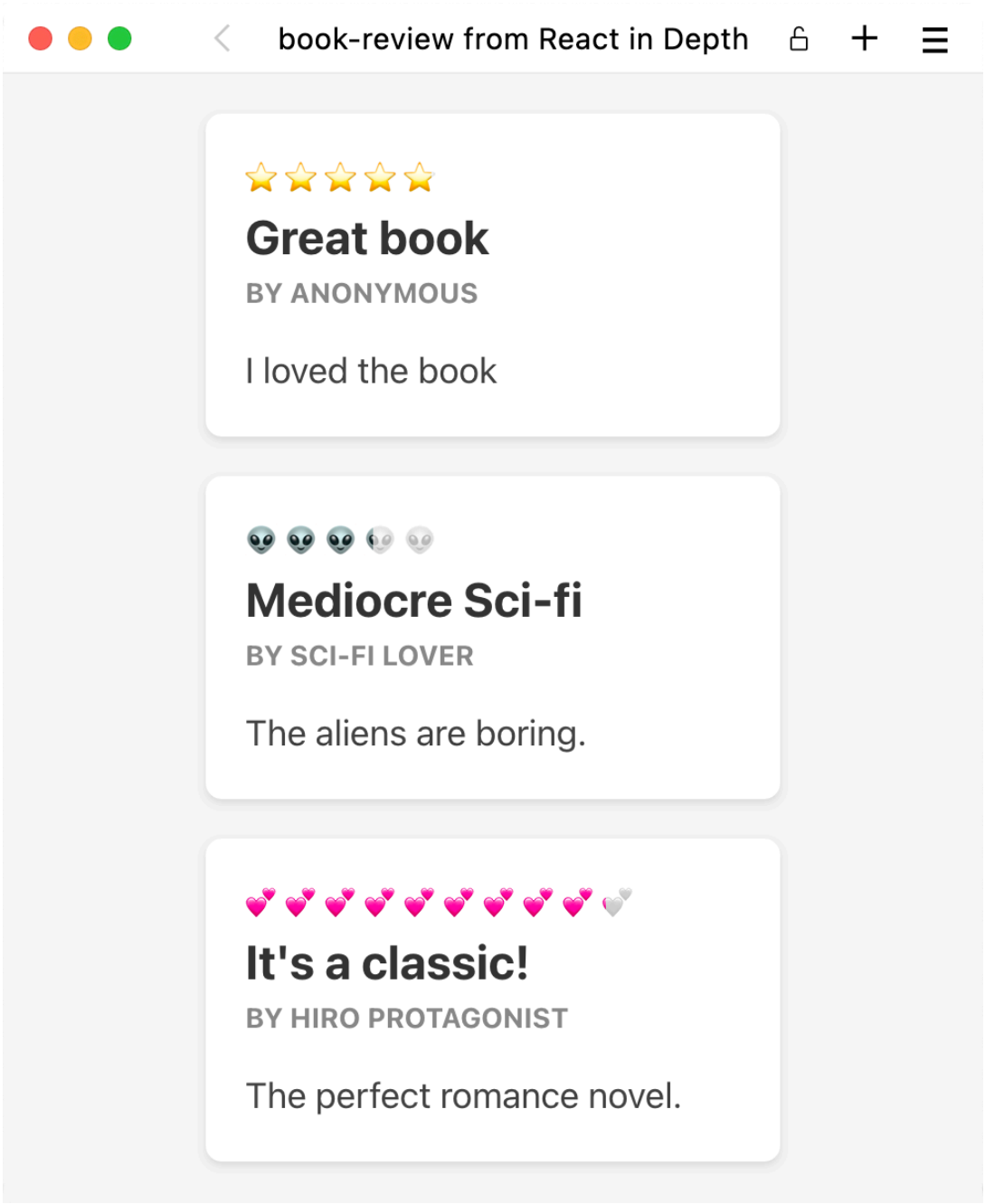

Figure 5.4 Three book reviews with slightly different rating displays. Varying the review icon with the genre looks cool, no? We can provide that function in a typesafe way even when the `BookReview` component exposes some properties of a third-party component.

EXAMPLE: BOOK-REVIEW

This example is in the `book-review` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch05/book-review
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch05-book-review**.

5.3.5 Restricting and loosening types

When we extend an interface, we can restrict the interface, but we cannot loosen it. If we have the interface

```
interface Style {
  width: number | string;
}
```

we can extend it while restricting it, perhaps for a `number`-only type:

```
interface NumberStyle extends Style {
  width: number;
}
```

But we cannot go the other way and loosen it for a broader definition:

```
interface AnyStyle extends Style {
  width: number | string | null;
}
```

The preceding code would result in the following TypeScript error message:

```
Interface 'AnyStyle' incorrectly extends interface 'Style'.
  Types of property 'width' are incompatible.
    Type 'string | number | null' is not assignable to type
    'string | number'.
      Type 'null' is not assignable to type 'string | number'.
```

So what do we do? In this case, we could not extend the interface because it has only one property. But suppose that the interface has more than one property. If so, we can omit the original and add it as a new one:

```
interface AnyStyle extends Omit<Style, "width"> {
  width: number | string | null;
}
```

This code works! We can use it in our components in the same way when we want to extend the properties of another component but restrict or loosen a type. The `value` property of an input, for example, is an optional property that can be a string, a `number`, or `undefined`. We can define our own `StringInput` component that allows the `value` to be a string only (no numbers are allowed):

```
interface StringInputProps extends ComponentPropsWithoutRef<"input"> {
  value?: string;
}
export function StringInput(props: StringInputProps) {
  return <input type="text" {...props} />;
}
```

If we want to do the opposite by creating an input in which the `value` could also be a Boolean, we'd have to omit the `value` before redefining it to be a looser type:

```
interface AnyInputProps
  extends Omit<ComponentPropsWithoutRef<"input">, "value"> {
  value?: string | boolean;
}
export function AnyInput({ value, ...rest }: AnyInputProps) {
  return <input type="text" value={String(value)} {...rest} />;
}
```

5.3.6 Using optional and required properties

In the preceding example, I talked about restricting or loosening a type and mentioned adding more or fewer alternatives to a union type. But we can toggle another aspect of a property, whether or not it's required. We could have created the string input in such a way that the `value` was not optional but required, by omitting the question mark:

```
interface StringInputProps extends ComponentPropsWithoutRef<"input"> {
  value: string;
}
```

But we can't go the other way. Suppose that we wanted to extend our previous `Rating` component but make the `value` of the `BookReview` component optional:

```
interface BookReviewProps extends PickedRatingProps {
  value?: number;
}
```

This code would result in a TypeScript error message similar to the one in section 5.3.5 because we're effectively expanding the type to be `number` or `undefined`, and that kind of loosening is not allowed, so we'd have to omit it before redefining it.

Sometimes, all we want to do is make a specific property required instead of optional, but we don't want to change anything else about the type. Suppose that we want to add a button and must define `onClick`. The built-in optional type for the `onClick` property of a button is fairly long:

```
// Built-in type for button onClick property
onClick?: React.MouseEventHandler<HTMLButtonElement>;
```

We don't want to repeat the entire definition—only remove the question mark. So we can refer directly to the old definition that we can get via `ComponentPropsWithoutRef`:

```
type ButtonProps = ComponentPropsWithoutRef<"button">;
interface ButtonWithClickProps extends ButtonProps {
  onClick: ButtonProps["onClick"];
}
```

This code, however, would mean doing a lot of typing if we want to do the same thing for multiple properties:

```
type ButtonProps = ComponentPropsWithoutRef<"button">;
interface HoverButtonProps extends ButtonProps {
  onMouseOver: ButtonProps["onMouseOver"];
  onMouseMove: ButtonProps["onMouseMove"];
  onMouseOut: ButtonProps["onMouseOut"];
}
```

There's a smarter way. TypeScript has a built-in interface, `Required<>`, that makes all properties of another type required. That behavior is not what we want here, but we can use it to create our own utility interface that makes some properties required:

```
type RequireSome<T, K extends keyof T> = T & Required<Pick<T, K>>
```

Then we can create our `HoverButtonProps` much more elegantly:

```
type ButtonProps = ComponentPropsWithoutRef<"button">;
type RequireSome<T, K extends keyof T> = T & Required<Pick<T, K>>
type HoverButtonProps = RequireSome<
  ButtonProps,
  "onMouseOver" | "onMouseMove" | "onMouseOut"
>;
```

You're probably beginning to see the power of TypeScript. You can do programming by using types alone!

5.3.7 Using either/or properties

Let's create a product-card component that can be used on an e-commerce website, but we want to use the same component to display prices for both regular items and on-sale items. We want the result to look like figure 5.5.

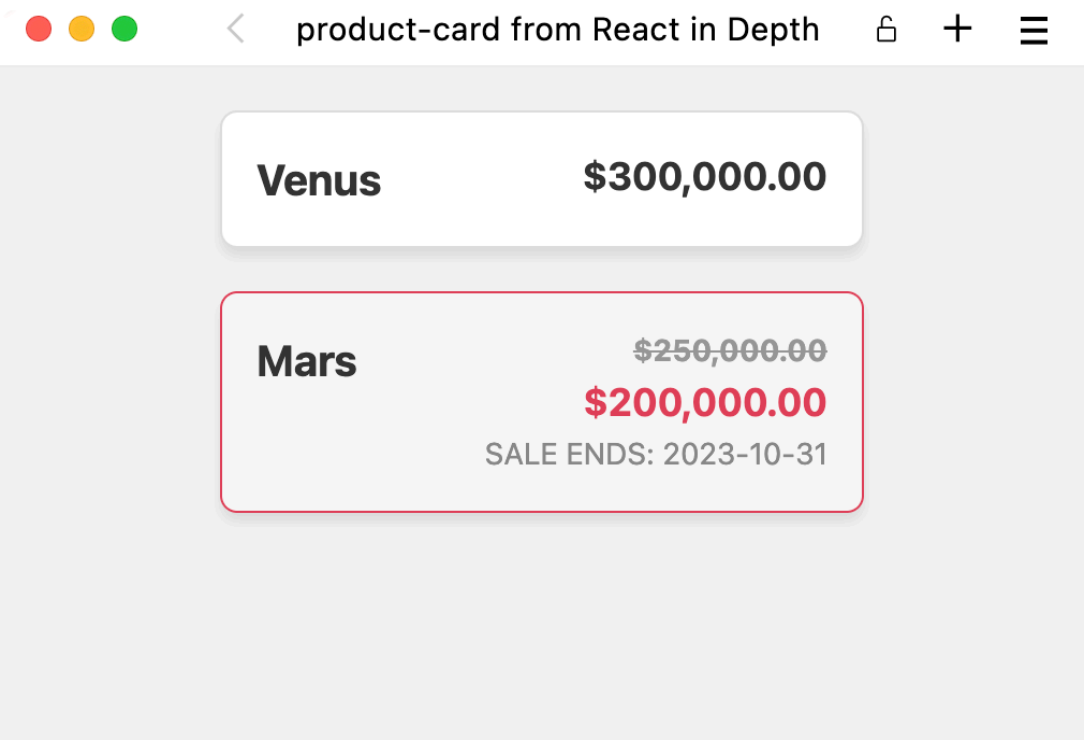

Figure 5.5 A product-card component that can be used for both regular items and on-sale items. Notice that the on-sale item has additional info, including the sale price and an expiry date.

We want to define a component that always requires some properties. Additionally, it will require some extra properties if another property has a certain value. We want the interface to match one interface or the other but not be some mix (figure 5.6).

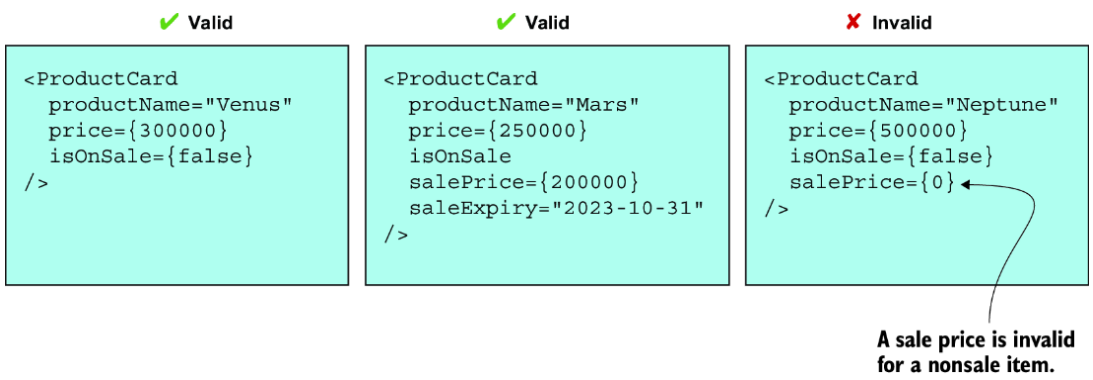

Figure 5.6 Valid and invalid property combinations for our product card

We know how to define these two interfaces separately:

```
interface ProductCardSaleProps {
  productName: string;
  price: number;
  isOnSale: true;
  salePrice: number;
  saleExpiry: string;
}
interface ProductCardNoSaleProps {
  productName: string;
  price: number;
  isOnSale: false;
}
```

To create the final props type for our component, we simply do the naive thing by making a union type of those two interfaces:

```
type ProductCardProps = ProductCardSaleProps | ProductCardNoSaleProps;
```

TypeScript is super clever. It knows that if the props received in the component has `isOnSale` set to `true`, the properties object must conform to the `ProductCardSaleProps` type, so it must have both `salePrice` and `saleExpiry` properties. Conversely, if `isOnSale` is `false`, it cannot have these properties. We can use an expression like this one:

```
{isOnSale && (
  <div className="sale-price">{props.salePrice}</div>
)}
```

Let's see how TypeScript knows about these types. Figure 5.7 shows a screenshot of a code snippet in Visual Studio Code with TypeScript support enabled; we're trying to access `props.salePrice` without checking `props.isOnSale`.

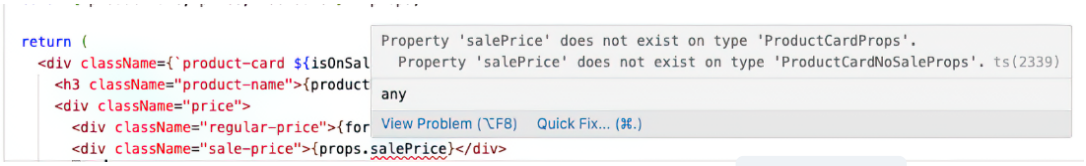

Figure 5.7 TypeScript throws an error because `salePrice` is not defined in `ProductCardNoSaleProps`, which is one of two possible interfaces for the properties.

Figure 5.8 shows how TypeScript knows that the props type must be the on-sale variant when we perform the check for `isOnSale` first.

Figure 5.8 When we check for `isOnSale` first, TypeScript is perfectly well aware that the original props object must be of type `ProductCardSaleProps`, so it must have the `salePrice`, and all is well here.

The following listing shows the full component for a product card.

Listing 5.10 Product card with a discriminated union

```
import "./ProductCard.css";
interface ProductCardSaleProps {    #1
  productName: string;
  price: number;
  isOnSale: true;
  salePrice: number;
  saleExpiry: string;
}
interface ProductCardNoSaleProps {    #2
  productName: string;
  price: number;
  isOnSale: false;
}
type ProductCardProps =    #3
  | ProductCardSaleProps   #3
  | ProductCardNoSaleProps;    #3
function formatPrice(price: number) {    #4
  return price.toLocaleString("en-US", {
    style: "currency",
    currency: "USD",
  });
}
export function ProductCard(     #5
  props: ProductCardProps   #5
) {   #5
  const { productName, price, isOnSale } = props;     #6
  return (
    <div className={`product-card ${isOnSale ? "on-sale" : ""}`}>
      <h3 className="product-name">{productName}</h3>
      <div className="price">
        <div className="regular-price">{formatPrice(price)}</div>
        {isOnSale && (     #7
          <>
            <div className="sale-price">
              {formatPrice(props.salePrice)}     #8
            </div>
            <div className="sale-expiry">
              Sale ends: {props.saleExpiry}  #8
            </div>
          </>
        )}
      </div>
```

```
    </div>
  );
}
```

#1 Creates the definition for a sale item

#2 Creates the definition for a nonsale item

#3 Combines the two definitions for our final component props type

#4 Defines a small utility function to help format numbers nicely

#5 Creates the component definition without destructuring the props; we'll get to why that matters later in this section.

#6 Destructs the props "manually" as a separate statement

#7 Renders conditional JSX depending on a property

#8 Accesses optional properties in a typesafe way

We can improve this code a bit. You may recognize the fact that we repeated a few properties between the two interfaces. We defined `productName` and `price` in both interfaces. Let's extract those properties to a base interface and update the resulting type. The result would be the new types shown in the following listing.

Listing 5.11 Product-card types with base props (partial)

```
interface BaseProps {     #1
  productName: string;   #1
  price: number;  #1
}    #1
interface ProductCardSaleProps {     #2
  isOnSale: true;    #2
  salePrice: number;    #2
  saleExpiry: string;    #2
}   #2
interface ProductCardNoSaleProps {    #2
  isOnSale: false;    #2
}   #2
type ProductCardProps = BaseProps &     #3
  (ProductCardSaleProps |    #3
   ProductCardNoSaleProps);  #3
```

#1 Creates an interface with the shared props

#2 Creates the two variants by specifying only the props that vary between them

#3 Creates the resulting component props type by merging the base props with the union of the two alternatives

EXAMPLE: PRODUCT-CARD

This example is in the `product-card` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch05/product-card
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch05-product-card**.

A few minor things might have caught you off guard. We've strayed from some things that we normally do.

First, as you can see in the function declaration in listing 5.10, we don't spread the props as we normally do. We almost always spread the props in the definition, but we don't here. There are two reasons why we don't spread the props directly in the definition:

- We can't spread conditional props.
- TypeScript doesn't keep track of the relationship of props after spreading.

The first reason is easier to understand. We cannot spread the props object into the values `salePrice` and `saleExpiry` because the props object doesn't always contain these values. If we did spread them, what would their values be if the item is not on sale? What would happen if we used them anyway? TypeScript doesn't work this way. You cannot read something from an object that doesn't exist under every circumstance.

To explain the second reason, let's take a small detour. Why can't we spread the conditional props into a `...rest` object, as we sometimes do, and then, if the item is on sale, read the values from that object? Suppose that we took the following approach:

```
  export function ProductCard({
    productName,
    price,
    isOnSale,
    ...rest
  }: ProductCardProps) {
    ...
          {isOnSale && (
            ...
                {formatPrice(rest.salePrice)}
```

This code is essentially the same thing as listing 5.10, right? We try to read the sale price only from the `rest` object (which is the leftover properties after the product name, price and on-sale flag have been removed), if the item is on sale. So all should be well, shouldn't it?

Unfortunately, no. TypeScript is clever but not *that* clever. After we extract the leftover properties from the props object, TypeScript forgets where the value came from and assigns it a type that's not connected to the other extracted types for that object.

Even when we find out that `isOnSale` is `true`, TypeScript doesn't update its knowledge about the `rest` object; it still has the same union type and has not been narrowed. Compare the type of the props object in figure 5.8 with the `rest` object in figure 5.9.

Figure 5.9 When we extract the `rest` properties from the passed props, TypeScript disconnects the value of that object from the other properties extracted. So even when `isOnSale` is `true`, the type of the `rest` variable doesn't change.

Unfortunately, we have to do things a bit differently when we use discriminated unions, but taking that approach isn't the most common thing to do. Know that when you need discriminated unions, you have to change your habits a bit.

5.3.8 Forwarding refs

This section gets technical, and if you don't use reference forwarding much, you can skip it for now. When you need to, please come back to this section to get the inside scoop.

Note In React 19, reference forwarding no longer matters. `forwardRef` is no more. In React 19, you can accept `ref` as a regular property in a functional component, and typing it is much easier because we no longer need to exclude the ref property from the properties object and type it separately.

Suppose that you want to create an input component, but to give an outside component access to the HTML element for the input field inside your component, you want to allow a reference to be forwarded to the input field.

In this example, we can say that the main component includes a login form and wants to place the user keyboard focus inside the username input field when the application component loads. The main component would look like this:

```
import { useEffect, useRef } from "react";
import "./App.css";
import { Input } from "./Input";
export default function App() {
  const username = useRef<HTMLInputElement>(null);
  useEffect(() => {
    username.current?.focus();
  }, []);
  return (
    <main>
      <h1>Login</h1>
      <form>
        <Input ref={username} label="Username" name="username" />
        <Input label="Password" name="password" type="password" />
      </form>
    </main>
  );
}
```

Our first attempt to create the input component looks like the following code snippet, which doesn't have the reference and is a basic input component:

```
import { ComponentPropsWithoutRef } from "react";
interface InputProps extends ComponentPropsWithoutRef<"input"> {
  label: string;
}
export function Input({ label, ...rest }: InputProps) {
  return (
    <label>
      <span>
        {label}
        <input {...rest} />
      </span>
    </label>
  );
}
```

To allow the main component to add a `ref` to the custom input component, two things have to happen:

- We need to accept such a reference inside our component.
- We need to pass that reference to the input element.

We achieve the first thing by using the React built-in function `forwardRef()`. The naive implementation is

```
export const Input = forwardRef(function Input(
  { label, ...rest }: InputProps,
  ref
) {
```

The second step is assigning this `ref` to the input element, which sounds easy enough:

```
<input ref={ref} {...rest} />
```

That code should be fine, right? But it doesn't work. TypeScript complains that

```
Type 'ForwardedRef<unknown>' is not assignable to type
'LegacyRef<HTMLInputElement> | undefined'.
  Type 'MutableRefObject<unknown>' is not assignable to type
  'LegacyRef<HTMLInputElement> | undefined'.
    Type 'MutableRefObject<unknown>' is not assignable to type
    'RefObject<HTMLInputElement>'.
      Types of property 'current' are incompatible.
        Type 'unknown' is not assignable to type 'HTMLInputElement | null'.
```

This error is a long way of saying that the `forwardRef` function told TypeScript that the `ref` belongs to an unknown element, and an input element is not unknown. To fix this problem, we need to tell TypeScript that the `ref` belongs to an input element and only to an input element. When we do, TypeScript allows us to assign that reference to the input element. We can take either of two approaches:

- Type the reference directly.
- Provide generic type arguments to the `forwardRef` function.

The simple solution is to type the `ref` argument inside the function. The type to use is `Ref<HTMLInputElement>`, where `Ref` is an interface from the React package:

```
import type { ComponentPropsWithoutRef, Ref } from "react";
import { forwardRef } from "react";
...
export const Input = forwardRef(function Input(
  { label, ...rest }: InputProps,
  ref: Ref<HTMLInputElement>
) {
```

That approach is simple, and it works. The other solution adds type arguments to `forwardRef` and takes two type arguments: the element type and the property type. We can provide those two arguments as type arguments and then remove them from the function definition:

```
export const Input = forwardRef<HTMLInputElement, InputProps>(
  function Input({ label, ...rest }, ref) {
```

The nice thing about this solution is that it doesn't require the extra `Ref` import. The not-so-nice thing is that the type arguments are in reverse order. First, we specify the element type, which is used in the type for the second argument of the component definition function. Second, we pass in the type for properties, which is the first argument. That process is a bit weird and hard to remember, so you might make mistakes (though TypeScript would scream at you if you reversed the order of the types by accident).

Pick whichever solution you prefer as long as you use it consistently. I prefer typing the arguments (the first solution), as I generally avoid providing type arguments whenever possible because it feels a bit forced. There's no substantial evidence to justify that choice—merely personal preference. In chapter 6, I'll address one extra caveat with regard to `forwardRef`.

Summary

- TypeScript offers a more predictable and maintainable approach to coding in React compared with coding in plain JavaScript.
- Different file extensions, such as `*.ts` and `*.tsx`, distinguish between regular TypeScript and React components with TypeScript.
- Static typing is a foundational feature of TypeScript, helping you catch type mismatches and enhancing code reliability.
- TypeScript allows you to create custom types, specify allowed values, and define optional properties.
- When designing React components with TypeScript, you can use both types and interfaces to define properties and their expected data types.
- Optional properties in TypeScript provide flexibility to data structures while ensuring consistent functionality.
- Generics in TypeScript allow for dynamic typing, making them crucial for advanced React applications.
- Using TypeScript with React, you can specify types for `children` properties by using the `ReactNode` type from React or the `PropsWithChildren` interface. This approach streamlines the process and covers all the valid child node types, including strings, numbers, Booleans, and JSX elements. `PropsWithChildren` not only types the `children` property but also wraps other potential props, making it a versatile choice.
- When extending interfaces in TypeScript, you can build on properties of an existing interface, ensuring that code isn't duplicated unnecessarily and remains easy to maintain.
- TypeScript enables you to gather props of built-in and custom components by using the `ComponentProps` and `ComponentPropsWithoutRef` utility types.
- Use TypeScript's discriminated unions for components with multiple configurations based on property values.
- TypeScript provides conditional type checking to ensure valid combinations of component properties.
- In TypeScript, using `forwardRef` in React requires correct typing of references, as the default type is considered to be unknown, necessitating explicit specification of the `ref` type for inner elements.