# The TypeScript Handbook

## Unions and Intersection Types

So far, the handbook has covered types which are atomic objects. However, as you model more types you find yourself looking for tools which let you compose or combine existing types instead of creating them from scratch.

Intersection and Union types are one of the ways in which you can compose types.

## Union Types

Occasionally, you’ll run into a library that expects a parameter to be either a number or a string. For instance, take the following function:

/\*\*

\* Takes a string and adds "padding" to the left.

\* If 'padding' is a string, then 'padding' is appended to the left side.

\* If 'padding' is a number, then that number of spaces is added to the left side.

\*/

function padLeft(value: string, padding: any) {

if (typeof padding === "number") {

return Array(padding + 1).join(" ") + value;

}

if (typeof padding === "string") {

return padding + value;

}

throw new Error(`Expected string or number, got '${typeof padding}'.`);

}

padLeft("Hello world", 4); // returns " Hello world"[Try](https://www.typescriptlang.org/play/#code/PQKhCgAIUgVBDA1gUwM6XpVAXATgSwDsBzDQgEw3PPQCIAHeao42ybAe3YAtlIAbZADNsAOigwAkkMgByRsxKzI+dJhwESAGh7JCcheRbLVGevT3lklTroHDsWfFfHRI0g0yNKVayIQBXAFsAI2RcHWxefSj4R0DQ8MgOGVRGAGM0XyorGy4ovkERJxcJYHAhAMJ07HwOfQUAGQcACgA3eH4A5AAuLDwWHUMWPvhCAE8ASkgAbygVGRbscYsUyGGSSABeHchaBLDcWmm5yDPIXGRsANx9AEFcXHhxlo3SAGpIAEZJ0QArDhEFpsY6QT4dLrIADc8wAvvN8ItlqsZG9trtaBoWKDTudLtdbusvCwwZAId0YWd4WcorgOAB3fzIRkAUUeHFwLQABiyAB4WGrWfqaUgc-zBQ46YgcRyyAAkM2RyDWb1hslEXMmMPh4CarVoAAlkPx+Fx6Rz+ORaDoACxayDAYAXK43Qh0c6QI0ms0Wq1AA)

The problem with padLeft in the above example is that its padding parameter is typed as any. That means that we can call it with an argument that’s neither a number nor a string, but TypeScript will be okay with it.

// passes at compile time, fails at runtime.

let indentedString = padLeft("Hello world", true);[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwAcpgAZERDACgDcoJkQAueAZwxi1QHMAaQ44Jy7MoqAJ4BKZmw7cA3ACgA9EvgBaDWGQYNa5aqIsWIFvCgZ4YHAFsCWCAmzWQfRFHunz8GGicgAdAoOFpygqBggwADK7ELwALz8pORUAEQAEiAQEDjwAO44MBDAqXzsDBJyQA)

In traditional object-oriented code, we might abstract over the two types by creating a hierarchy of types. While this is much more explicit, it’s also a little bit overkill. One of the nice things about the original version of padLeft was that we were able to just pass in primitives. That meant that usage was simple and concise. This new approach also wouldn’t help if we were just trying to use a function that already exists elsewhere.

Instead of any, we can use a union type for the padding parameter:

/\*\*

\* Takes a string and adds "padding" to the left.

\* If 'padding' is a string, then 'padding' is appended to the left side.

\* If 'padding' is a number, then that number of spaces is added to the left side.

\*/

function padLeft(value: string, padding: string | number) {

// ...

}

let indentedString = padLeft("Hello world", true);

Argument of type 'boolean' is not assignable to parameter of type 'string | number'.Argument of type 'boolean' is not assignable to parameter of type 'string | number'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAUMAFSF6iGgAqAhgNaSKhWiIAu0AlgHYDmjnAJo378GAIgAOVYV26jQLOPIAWkUABtIAMxYA6UuQCSm0AHJJ0nidDsGTVhx4AaZZE6nz-GVZuNx41-yQggou6loszOyBemSgRu5SnpbWtqCcAK4AtgBGMM4sKm4FVBEZOTCgcMaIkgDG9ClCgcGKBaoa2pHR+sB4mumctSzscG7mADLhABQAblRq6ZCo9jLOHjLLbDKgAD5pWbnQAJSgAN6koCCgOjd4AL54eBoRXIGcLEEAyls8oAC8oAm01EAAlIGo1IoAO4INT8UT5aCLI4AbiAA)

A union type describes a value that can be one of several types. We use the vertical bar (|) to separate each type, so number | string | boolean is the type of a value that can be a number, a string, or a boolean.

## Unions with Common Fields

If we have a value that is a union type, we can only access members that are common to all types in the union.

interface Bird {

fly(): void;

layEggs(): void;

}

interface Fish {

swim(): void;

layEggs(): void;

}

declare function getSmallPet(): Fish | Bird;

let pet = getSmallPet();

pet.layEggs();

// Only available in one of the two possible types

pet.swim();

Property 'swim' does not exist on type 'Bird | Fish'.

Property 'swim' does not exist on type 'Bird'.Property 'swim' does not exist on type 'Bird | Fish'.

Property 'swim' does not exist on type 'Bird'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4BQuBLAOwBcYAzAQwGNJQAhA6AE1AG9dRRyAbATwAUASlQA3OAWYBuTqB6U+AUQDmyxMLETpuAL75iZaFVqgAYgUQALdrMQB3AgFsNocZJld5S1epGutMnq4zJDU8tB05ACuRNQkBHBEoMqQJADKjpQ8PAAKqS7mVqAAPgxM2rg8qaAADtUAvMmpGVm5+UIydSQAdF4qasIyuCCgAPJE-KCUopQE8gBGVaDEoIl0cOSgJJZ0JHZwtXCIiASLu3x1iLhd3fZOg0A)

Union types can be a bit tricky here, but it just takes a bit of intuition to get used to. If a value has the type A | B, we only know for certain that it has members that both A and B have. In this example, Bird has a member named fly. We can’t be sure whether a variable typed as Bird | Fish has a fly method. If the variable is really a Fish at runtime, then calling pet.fly() will fail.

## Discriminating Unions

A common technique for working with unions is to have a single field which uses literal types which you can use to let TypeScript narrow down the possible current type. For example, we’re going to create a union of three types which have a single shared field.

type NetworkLoadingState = {

state: "loading";

};

type NetworkFailedState = {

state: "failed";

code: number;

};

type NetworkSuccessState = {

state: "success";

response: {

title: string;

duration: number;

summary: string;

};

};

// Create a type which represents only one of the above types

// but you aren't sure which it is yet.

type NetworkState =

| NetworkLoadingState

| NetworkFailedState

| NetworkSuccessState;

All of the above types have a field named state, and then they also have their own fields:

|  |  |  |
| --- | --- | --- |
| **NetworkLoadingState** | **NetworkFailedState** | **NetworkSuccessState** |
| state | state | state |
|  | code | response |

Given the state field is common in every type inside NetworkState - it is safe for your code to access without an existence check.

With state as a literal type, you can compare the value of state to the equivalent string and TypeScript will know which type is currently being used.

|  |  |  |
| --- | --- | --- |
| **NetworkLoadingState** | **NetworkFailedState** | **NetworkSuccessState** |
| "loading" | "failed" | "success" |

In this case, you can use a switch statement to narrow down which type is represented at runtime:

type NetworkState =

| NetworkLoadingState

| NetworkFailedState

| NetworkSuccessState;

function logger(state: NetworkState): string {

// Right now TypeScript does not know which of the three

// potential types state could be.

// Trying to access a property which isn't shared

// across all types will raise an error

state.code;

Property 'code' does not exist on type 'NetworkState'.

Property 'code' does not exist on type 'NetworkLoadingState'.Property 'code' does not exist on type 'NetworkState'.

Property 'code' does not exist on type 'NetworkLoadingState'.

// By switching on state, TypeScript can narrow the union

// down in code flow analysis

switch (state.state) {

case "loading":

return "Downloading...";

case "failed":

// The type must be NetworkFailedState here,

// so accessing the `code` field is safe

return `Error ${state.code} downloading`;

case "success":

return `Downloaded ${state.response.title} - ${state.response.summary}`;

}

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4BQAXATwAdJQA5SfAdwQGsAZOAQwBMBLAOwHMBlfZvjIBeUAG9coUIgFDUAIgA2LDj3kBuXAF9NBEmUo16AMWbtFkVv0Ejxk6bMgKAZmYusN9gMZxWT0JwArgC2AEYwmjq4eqQUVLTQdLyBXl6QiIjWQqCiElIyNgqIKWkZnlLQ6cRwnIj+eVKg+Oz4Fqgy0Fzcmo2grIHQguw1qEFhEfb5IcHM0ITt+J08PaBRUSCgALTbXoH425sxBvH0WSL2AD5xRolMbF1nl9cJdKbmlo9SV4YvyanpmUcumcgU4XmaNVAym43BgAAoCnJnqdHABKBZLbh2KQbABK7G4AAt8AE4NRQAAVfS8LydYgk1hwdKkkl0Thk0DUQnsLyE0BwZxNQlkfCEyqQewbapCTjNZiKJr6RAOGygHyBRSsUDhAB00RxYApcy6TTgoGY-wy5tAxHgpGgRE53N5oHYiE4AHISYhCbNLJKwBb4Fb5QqiKRldRzArBm6yMxOKAYPBoPZEZAdT4-LoDaAAEKEaRR-C8k2Q9MAGkp1Np7HpaoTAVm8HJorIoOGnADfTJia4at8ZGcynJCflhEQbrTxZdCMcOvTqOxjS8zDqoCUKi68mQk0alXwA0T8gAIr3lPceDrr+UV2uyPJXO8PLvermKcLFbFgoEZNrjjcrxuB8jigMKlQVnuuaIGaFqlJOPBCmQAAGWaQMhoDOOwkCaq6yqIMwzgSm+oAHkeoDIQAorACCgAAJGI6aZoOWg9tQnAXqo3DISsUiruu8jFJaiA7lBpFUORyFnuxnGWPRjHzpUiDVLUGbNK0kCsZs8lMUpKl1Au0yzIQWg8fYWjaEAA)

## Union Exhaustiveness checking

We would like the compiler to tell us when we don’t cover all variants of the discriminated union. For example, if we add NetworkFromCachedState to NetworkState, we need to update logger as well:

type NetworkFromCachedState = {

state: "from\_cache";

id: string;

response: NetworkSuccessState["response"];

};

type NetworkState =

| NetworkLoadingState

| NetworkFailedState

| NetworkSuccessState

| NetworkFromCachedState;

function logger(s: NetworkState) {

switch (s.state) {

case "loading":

return "loading request";

case "failed":

return `failed with code ${s.code}`;

case "success":

return "got response";

}

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYBsWBQAXATwAdJQA5SfAdwQGsAZOAQwBMBLAOwHMBlfZvjIBeUAG9QiAUNQAiADYsOPWaAC+AbgIkylGvQBizdvMit+gkeMnTIcgGbHTrWRtABjOKzuhOAVwBbACMYdS0iUgoqWmg6Xj93d0hERAshUFExXFAbSzlEBKSU12zQaGTiOE5EHyycnPx2fFNUKWgubi160FY-aEF2KtR-YJgu+oKAgOZoQlb8dp5xzVwVkFAAWi33P3wtje1IvRi6A3gAgGFmdwALMzSrOtyZUFl7c4B9d2u7kpz2VjzRadUrlRCVao+Y70eKJZKpWwAbVkYIhNVkAF0tCtDrpojDbBlSgAfKL6WJMNgdB4ksknIwme62WnQ2KwooIyws-GxM5wS4-JmWLS4ex+TjuRpVUCKbjcGAAChQdIJlgAlOJSohqE1bqAlQA6KTqzXdb41V6KKkqZClbrlfB9TiWpQdMqQACOfmS+D+9XNZDeTjMslt3RyDqdoAABo5GaxQDr8DcPF4yAASMSIA2ebxqaPjHIB14FOHFMPhyPQZ2ybhwfDu8FVdHLVZAA)

There are two ways to do this. The first is to turn on --strictNullChecks and specify a return type:

function logger(s: NetworkState): string {

Function lacks ending return statement and return type does not include 'undefined'.Function lacks ending return statement and return type does not include 'undefined'. switch (s.state) {

case "loading":

return "loading request";

case "failed":

return `failed with code ${s.code}`;

case "success":

return "got response";

}

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYBsWBQAXATwAdJQA5SfAdwQGsAZOAQwBMBLAOwHMBlfZvjIBeUAG9QiAUNQAiADYsOPWaAC+AbgIkylGvQBizdvMit+gkeMnTIcgGbHTrWRtABjOKzuhOAVwBbACMYdS0iUgoqWmg6Xj93d0hERAshUFEJKUs5RASklNVNbUi9GLoDeACAYWZ3AAszNKss2wcqgH13OsairRLdaPpmjNxQUAAfKP1YpjYuPlsxyenyoxMmpfGpsuH85NStld3Yyrgans3LfpBQAFoH9z98B7vcez9Od3x2OE5QRTcbgwAAUKFWw1sAEpUFJoAtxMtENR2PgGqAwQA6bJCKGI8bjbqIMgKJQLWTIZYE0DQKh+aD-UnzHg0yAARz8yXwripHmYxNAskcGxclOp41p+Hp-wABsLnKAUfh6h4vGQACRiRCYzzeNQyrTUokkvKJA4U3kSukMwXcOD4VmIYh-Yk88ZqXBqIA)

Because the switch is no longer exhaustive, TypeScript is aware that the function could sometimes return undefined. If you have an explicit return type string, then you will get an error that the return type is actually string | undefined. However, this method is quite subtle and, besides, [--strictNullChecks](https://www.typescriptlang.org/tsconfig#strictNullChecks) does not always work with old code.

The second method uses the never type that the compiler uses to check for exhaustiveness:

function assertNever(x: never): never {

throw new Error("Unexpected object: " + x);

}

function logger(s: NetworkState): string {

switch (s.state) {

case "loading":

return "loading request";

case "failed":

return `failed with code ${s.code}`;

case "success":

return "got response";

default:

return assertNever(s);

Argument of type 'NetworkFromCachedState' is not assignable to parameter of type 'never'.Argument of type 'NetworkFromCachedState' is not assignable to parameter of type 'never'. }

}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAUAC4CeADpKAHKQEDuCA1gDJwCGAJgJYB2A5gMoEWBcgF5QAb1CJBw1ACIANq0685oAL4BuQqXJVaDAGIsOCyGwFDREqTMjyAZibNs5m0AGM4be6C4BXAFsAIxgNbWIySmo6aHo+fw8PSERES2FQMUlpK3lEROTUtS0dKP1Y+kN4QIBhFg8AC3N062y7R2qAfQ96puLtUr0YhhbMvFBQAB9ogzjmdm5+O3GpmYrjU2blienykYKUtO3VvbiquFrerattEFAAWkePfwJH+7wHfy4PAg44LlALFSMAIVAAbjAABQAD1QXEgEOgAEo4QiwuIVgQGvAaH5ILiAKKwBCQuQAVXh0LIP3MoDgwQAVpAfvJQABqUDQpHadR4D5fH5-AFKHg8KEoNYjOwo2zQRYSFaIGgcAiNUCQxAAOhywiRComEx6iHIimUizkyBWBtA0Go-mgANNC14NsgAEd-CkCG4rZ4gSanJtXJbrRNbQR7QCAAaBlygZVYzzecgAEnEWq8PnUUe01qNJvySUOFt9YbtDtAch4cAIrsQJH+xp91p8Tn8CgIIdD4cjgOB0FBaOgGu5K156iAA)

Here, assertNever checks that s is of type never — the type that’s left after all other cases have been removed. If you forget a case, then s will have a real type and you will get a type error. This method requires you to define an extra function, but it’s much more obvious when you forget it because the error message includes the missing type name.

## Intersection Types

Intersection types are closely related to union types, but they are used very differently. An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. For example, Person & Serializable & Loggable is a type which is all of Person and Serializable and Loggable. That means an object of this type will have all members of all three types.

For example, if you had networking requests with consistent error handling then you could separate out the error handling into its own type which is merged with types which correspond to a single response type.

interface ErrorHandling {

success: boolean;

error?: { message: string };

}

interface ArtworksData {

artworks: { title: string }[];

}

interface ArtistsData {

artists: { name: string }[];

}

// These interfaces are composed to have

// consistent error handling, and their own data.

type ArtworksResponse = ArtworksData & ErrorHandling;

type ArtistsResponse = ArtistsData & ErrorHandling;

const handleArtistsResponse = (response: ArtistsResponse) => {

if (response.error) {

console.error(response.error.message);

return;

}

console.log(response.artists);

};