**Hello World Steps:**

1. Install Node.js from <https://nodejs.org/>
2. Install TypeScript 4.9+ using Node Package Manager command

**npm install -g typescript.**

Install VSCode from <https://code.visualstudio.com/Download>

You can test your install by checking the version.

**tsc --version**

Generate tsconfig.json by giving this command:

**tsc –init**

If you want to run TypeScript tsc command in Windows Powershell:

1. Open Powershell in Adminstrator mode
2. Run command:

**Set-ExecutionPolicy RemoteSigned**

Make it a Node.js project by giving the following command:

**npm init –y**

Install types for Node.js

**npm i @types/node –D**

Create .gitignore file

Create file app.ts

Note that the most basic types in TypeScript correspond to the seven basic kinds of primitives in JavaScript:

1. null

2. undefined

3. boolean // true or false

4. string // "", "Hello World"

5. number // 0, 5.1, -9

6. bigint // 0n, 5n, -10n

7. symbol // Symbol(), Symbol("hi")

Transpile JavaScript by running:

**Tsc**

Run transpiled javascript by running the following command:

**node app**

[TypeScript](https://www.typescriptlang.org/) is a typed superset of JavaScript that compiles to plain JavaScript. It offers classes, modules, and interfaces to help you build robust components.

Visual Studio Code includes TypeScript language support but does not include the TypeScript compiler, tsc. You will need to install the TypeScript compiler either globally or in your workspace to transpile TypeScript source code to JavaScript (tsc HelloWorld.ts).

Another option is to install the TypeScript compiler locally in your project

**npm install --save-dev typescript**

[H**ello World**](https://code.visualstudio.com/Docs/languages/typescript#_hello-world)

Let's start with a simple Hello World Node.js example. Create a new folder HelloWorld and launch VS Code.

**mkdir HelloWorld**

**cd HelloWorld**

**code .**

From the File Explorer, create a new file called helloworld.ts

Now add the following TypeScript code. You'll notice the TypeScript keyword let and the string type declaration.

**let message: string = 'Hello World';**

**console.log(message);**

To compile your TypeScript code, you can open the [Integrated Terminal](https://code.visualstudio.com/docs/terminal/basics) (Ctrl+`) and type tsc helloworld.ts. This will compile and create a new helloworld.js JavaScript file.

If you have Node.js installed, you can run node helloworld.js.

[**IntelliSense**](https://code.visualstudio.com/Docs/languages/typescript#_intellisense)

IntelliSense shows you intelligent code completion, hover information, and signature help so that you can write code more quickly and correctly.

VS Code provides IntelliSense for individual TypeScript files as well as TypeScript tsconfig.json projects

You can also show the hover information at the current cursor position with the Ctrl+K Ctrl+I keyboard shortcut

Signature help is shown automatically when you type a ( or , within a function call. Use Ctrl+Shift+Space to manually trigger signature help.

## [Snippets](https://code.visualstudio.com/Docs/languages/typescript#_snippets)

In addition to smart code completions, VS Code also includes basic TypeScript [snippets](https://code.visualstudio.com/docs/editor/userdefinedsnippets) that are suggested as you type.

You can install extensions to get additional snippets or define your own snippets for TypeScript. See [User Defined Snippets](https://code.visualstudio.com/docs/editor/userdefinedsnippets) for more information.

## [Errors and warnings](https://code.visualstudio.com/Docs/languages/typescript#_errors-and-warnings)

The TypeScript language service will analyze your program for coding problems and report errors and warnings:

* In the Status bar, there is a summary of all errors and warnings counts.
* You can click on the summary or press Ctrl+Shift+M to display the **PROBLEMS** panel with a list of all current errors.
* If you open a file that has errors or warnings, they will be rendered inline with the text and in the overview ruler.

## [Code navigation](https://code.visualstudio.com/Docs/languages/typescript#_code-navigation)

Code navigation lets you quickly navigate TypeScript projects.

* **Go to Definition** F12 - Go to the source code of a symbol definition.
* **Peek Definition** Alt+F12 - Bring up a Peek window that shows the definition of a symbol.
* **Go to References** Shift+F12 - Show all references to a symbol.
* **Go to Type Definition** - Go to the type that defines a symbol. For an instance of a class, this will reveal the class itself instead of where the instance is defined.
* **Go to Implementation** Ctrl+F12 - Go to the implementations of an interface or abstract method.

You can navigate via symbol search using the **Go to Symbol** commands from the **Command Palette** (Ctrl+Shift+P).

* **Go to Symbol in File** Ctrl+Shift+O
* **Go to Symbol in Workspace** Ctrl+T

# JSON ( Javascript Object Notation )

1. JSON stands for Javascript Object Notation.
2. JSON is a text-based data format that is used to store and transfer data.
3. // JSON syntax

{

"name": "Vipin",

"age": 21,

"gender": "male",

}

But wait, Is JSON is similar to javaScript objects?

The Answer is No.

1. JavaScript objects can contain functions but JSON not.
2. JavaScript objects can only be used in JavaScript but JSON can be created and used by other programming languages.

## JSON Data

1. JSON data consists of key/value pairs similar to JavaScript object properties.
2. The key and values are written in double quotes separated by a :.
3. Example :

// JSON data

"name": "Vipin"

1. JSON data requires double quotes for the key.

## JSON Object

1. The JSON object is written inside curly braces { }.
2. JSON objects can contain multiple key/value pairs.
3. Example :
4. // JSON object
5. { "name": "Vipin", "age": 21 }

## JSON Array

1. JSON array is written inside square brackets [ ].
2. Example :
3. // JSON array
4. [ "Vipin", "Ankit", "Raj"]

## Accessing JSON Data

1. We can access JSON data using the dot notation.
2. Example :
3. // JSON object
4. const detail = { "name": "Vipin", "age": 21 }
5. // accessing JSON object
6. console.log(detail. name); // Vipin
7. We can also use square bracket syntax [] to access JSON data.
8. Example :
9. // JSON object
10. const detail = {
11. "name": "Vipin",
12. "age": 21
13. }

// accessing JSON object

console.log(detail["age"]); // Vipin

## Use of JSON

1. JSON is the most commonly used format for transmitting data (data interchange) from a server to a client and vice-versa.
2. JSON data are very easy to parse and use.
3. JSON is language independent(We can create and use JSON in other programming languages too).

## Overview

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

JavaScript projects can use a jsconfig.json file instead, which acts almost the same but has some JavaScript-related compiler flags enabled by default.

A project is compiled in one of the following ways:

## Using tsconfig.json or jsconfig.json

* By invoking tsc with no input files, in which case the compiler searches for the tsconfig.json file starting in the current directory and continuing up the parent directory chain.
* By invoking tsc with no input files and a --project (or just -p) command line option that specifies the path of a directory containing a tsconfig.json file, or a path to a valid .json file containing the configurations.

When input files are specified on the command line, tsconfig.json files are ignored.

## Syntax Error

tsc app.ts

The Output:

app.ts:1:1 - error TS1435: Unknown keyword or identifier. Did you mean 'let'?

1 **lett** message = "Hello World";//syntax error

Found 1 error in app.ts:1

Note that .js file has been generated but it is not valid.

# Syntax Error

tsc app.ts

The Output:

app.ts:2:9 - error TS2551: Property 'loger' does not exist on type 'Console'. Did you mean 'log'?

2 console.**loger**(message);

~~~~~

../../../../../../usr/local/lib/node\_modules/typescript/lib/lib.dom.d.ts:17095:5

17095 log(...data: any[]): void;

~~~~~~~~~~~~~~~~~~~~~~~~~~

'log' is declared here.

Found 1 error in app.ts:2

Note that .js file has been generated but it is not valid.

# Syntax Error

tsc app.ts

The Output:

app.ts:2:1 - error TS2322: Type 'number' is not assignable to type 'string'.

2 message = **6**;

Found 1 error in app.ts:2

# Strong Typing

# **Everyday Types**

We’ll start by reviewing the most basic and common types you might encounter when writing JavaScript or TypeScript code. These will later form the core building blocks of more complex types.

## The primitives: string, number, and boolean

JavaScript has three very commonly used [primitives](https://developer.mozilla.org/en-US/docs/Glossary/Primitive): string, number, and boolean. Each has a corresponding type in TypeScript. As you might expect, these are the same names you’d see if you used the JavaScript typeof operator on a value of those types:

* string represents string values like "Hello, world"
* number is for numbers like 42. JavaScript does not have a special runtime value for integers, so there’s no equivalent to int or float - everything is simply number
* boolean is for the two values true and false

The type names String, Number, and Boolean (starting with capital letters) are legal, but refer to some special built-in types that will very rarely appear in your code. Always use string, number, or boolean for types.

## Arrays

To specify the type of an array like [1, 2, 3], you can use the syntax number[]; this syntax works for any type (e.g. string[] is an array of strings, and so on). You may also see this written as Array<number>, which means the same thing. We’ll learn more about the syntax T<U> when we cover generics.

Note that [number] is a different thing; refer to the section on [Tuples](https://www.typescriptlang.org/docs/handbook/2/objects.html#tuple-types).

### **Tuple Types**

A tuple type is another sort of Array type that knows exactly how many elements it contains, and exactly which types it contains at specific positions.

type StringNumberPair = [string, number];

### **any**

TypeScript also has a special type, any, that you can use whenever you don’t want a particular value to cause typechecking errors.

When a value is of type any, you can access any properties of it (which will in turn be of type any), call it like a function, assign it to (or from) a value of any type, or pretty much anything else that’s syntactically legal:

let obj: any = { x: 0 };

// None of the following lines of code will throw compiler errors.

// Using `any` disables all further type checking, and it is assumed

// you know the environment better than TypeScript.

obj.foo();

obj();

obj.bar = 100;

obj = "hello";

const n: number = obj;

[Try](https://www.typescriptlang.org/play/#code/DYUwLgBA9gRgVgLggQwHYE8IF4IG8IAeSADBAL4DcAsAFAD0dEAclKiNAGYRgAW7HUYMCgB3AJaoA5hGASQAZ04QAxlAAm7cUO48ATqJVQAtgAcxoXRBC79u+QDpaDCAFV5E6QAM06TxDVi8sgwoIrI2hwArrq81tzoJuzKfMoA1h4ANCioahBikIEo8vKRRiBqTozoUJEQqagGsVaoAG5i+qhlqJAw4GBxvGgQACoJIADKyrpiJmCONLB)

The any type is useful when you don’t want to write out a long type just to convince TypeScript that a particular line of code is okay.

### **noImplicitAny**

When you don’t specify a type, and TypeScript can’t infer it from context, the compiler will typically default to any.

You usually want to avoid this, though, because any isn’t type-checked. Use the compiler flag [noImplicitAny](https://www.typescriptlang.org/tsconfig#noImplicitAny) to flag any implicit any as an error.

### **No Implicit Any - noImplicitAny**

In some cases where no type annotations are present, TypeScript will fall back to a type of any for a variable when it cannot infer the type.

This can cause some errors to be missed, for example:

function fn(s) {

// No error?

console.log(s.subtr(3));

}

fn(42);

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDsHsEkFsAOAbAlgY1QFwIKQJ4BcoAZgIbIDOApgFAkCuk6Wq0kpkAFJQJSgBvWqFAhQAOWihqAJxnQZAfmGh07StGTUAdMmgBzHtsoMARlhlcAzL14BuWgF963ACwAme0A)

Turning on noImplicitAny however TypeScript will issue an error whenever it would have inferred any:

function fn(s) {

Parameter 's' implicitly has an 'any' type.Parameter 's' implicitly has an 'any' type.

console.log(s.subtr(3));

}

## Type Annotations on Variables

When you declare a variable using const, var, or let, you can optionally add a type annotation to explicitly specify the type of the variable:

let myName: string = "Alice";

[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAtgngcgQyiAXBAzmATgSwHYDmEAvBAEQCCwOAxiGQNwCwAUAPRsRfdcB6-A-hAAqMAA4gICPHgD2YBGByy8QA)

TypeScript doesn’t use “types on the left”-style declarations like int x = 0; Type annotations will always go after the thing being typed.

In most cases, though, this isn’t needed. Wherever possible, TypeScript tries to automatically infer the types in your code. For example, the type of a variable is inferred based on the type of its initializer:

// No type annotation needed -- 'myName' inferred as type 'string'

let myName = "Alice";

[Try](https://www.typescriptlang.org/play/#code/PTAEDkHtQFwTwA4FNQEMB27I1TAlpOqOkkgCbmgC0VoA5ALZzioNJ2h7oBmSATn0qoAzrEQo6wmHy4BzOgFgAUABskMUExZtQAXlAAiAIIq8AYyQGA3EA)

For the most part you don’t need to explicitly learn the rules of inference. If you’re starting out, try using fewer type annotations than you think - you might be surprised how few you need for TypeScript to fully understand what’s going on.

## Functions

Functions are the primary means of passing data around in JavaScript. TypeScript allows you to specify the types of both the input and output values of functions.

### **Parameter Type Annotations**

When you declare a function, you can add type annotations after each parameter to declare what types of parameters the function accepts. Parameter type annotations go after the parameter name:

// Parameter type annotation

function greet(name: string) {

console.log("Hello, " + name.toUpperCase() + "!!");

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAUEMCdIWwKYBcHVEgngBwaSA7fAeyUiQEsj8BYAKADMBXfAYwqtAHNoFkAKfPAQAuUAGck0cvk4BKUAG86oUCBXqNm9QD1de3ctAsqYogBsEAOjNFOfAEQAJBGZsAaUPdABqUIMSWSEQAqlg40ADCkGIIfPK+9gCEifayANx0AL5AA)

When a parameter has a type annotation, arguments to that function will be checked:

// Would be a runtime error if executed!

greet(42);

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

Even if you don’t have type annotations on your parameters, TypeScript will still check that you passed the right number of arguments.

### **Return Type Annotations**

You can also add return type annotations. Return type annotations appear after the parameter list:

function getFavoriteNumber(): number {

return 26;

}

Much like variable type annotations, you usually don’t need a return type annotation because TypeScript will infer the function’s return type based on its return statements. The type annotation in the above example doesn’t change anything. Some codebases will explicitly specify a return type for documentation purposes, to prevent accidental changes, or just for personal preference.

#### Functions Which Return Promises

If you want to annotate the return type of a function which returns a promise, you should use the Promise type:

async function getFavoriteNumber(): Promise<number> {

return 26;

}

[Try](https://www.typescriptlang.org/play/#code/IYZwngdgxgBAZgV2gFwJYHsIwOYFNkBiwAbugE6rK4ByCAtgEa5kAUAlAFwwAKZ6dqELgA8EekzIA+GAG8AsACgYMMvgRksAJgBsAbkUBfIA)

### **Anonymous Functions**

Anonymous functions are a little bit different from function declarations. When a function appears in a place where TypeScript can determine how it’s going to be called, the parameters of that function are automatically given types.

Here’s an example:

const names = ["Alice", "Bob", "Eve"];

// Contextual typing for function - parameter s inferred to have type string

names.forEach(function (s) {

console.log(s.toUpperCase());

});

// Contextual typing also applies to arrow functions

names.forEach((s) => {

console.log(s.toUpperCase());

});

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYMEYCwAoAYzgDtEAXUEgQwFtJFQBeUAbQCIBBAGwEtDI7ADSh2AITgAjYaICiAN0EBdANwECIUAGFS5SAA9yAV2rdQ5AJ4AHXiQDmoAGYInRkoXK9SoALSgr1NB0kHrQoIy2jjDQkAAm5nCgABbUiubWkOHk0LZ2BDT0iAB0ztCy1IRJABSObh5eJKBViACUoADeBKCgxGRw3JBF3HB2zUXkcACqVlYwWtSI)

Even though the parameter s didn’t have a type annotation, TypeScript used the types of the forEach function, along with the inferred type of the array, to determine the type s will have.

This process is called contextual typing because the context that the function occurred within informs what type it should have.

Similar to the inference rules, you don’t need to explicitly learn how this happens, but understanding that it does happen can help you notice when type annotations aren’t needed. Later, we’ll see more examples of how the context that a value occurs in can affect its type.

## Object Types

Apart from primitives, the most common sort of type you’ll encounter is an object type. This refers to any JavaScript value with properties, which is almost all of them! To define an object type, we simply list its properties and their types.

For example, here’s a function that takes a point-like object:

// The parameter's type annotation is an object type

function printCoord(pt: { x: number; y: number }) {

console.log("The coordinate's x value is " + pt.x);

console.log("The coordinate's y value is " + pt.y);

}

printCoord({ x: 3, y: 7 });

[Try](https://www.typescriptlang.org/play/#code/PTAEBUAsFNQBwIYCcEFtoBdpIOQGdQMBPOWBAO3IHsMEMBLK80egi0KgIwCtoBjDIRLQAsACgAZgFdyAxszhJ65DAGEqVJABMAFHAwAuUAG9QADyPkpqTtgDcoIpeu2koAL4BKE+NCgQfoFBwSF+AHoRkVHRMZG+oHxMeFQANtAAdClUAOY6AERQsImaWsp00PjmoABuCClSsKygeaAA1PAY6WaedvGJ5MlpmTn5hQka2mVYlUQ1dQ0sBC)

Here, we annotated the parameter with a type with two properties - x and y - which are both of type number. You can use , or ; to separate the properties, and the last separator is optional either way.

The type part of each property is also optional. If you don’t specify a type, it will be assumed to be any.

### **Optional Properties**

Object types can also specify that some or all of their properties are optional. To do this, add a ? after the property name:

function printName(obj: { first: string; last?: string }) {

// ...

}

// Both OK

printName({ first: "Bob" });

printName({ first: "Alice", last: "Alisson" });

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABABwE4zFAcgQwLYCmAFHAEYBWAXIgN6LAyoDOU1L6YA5gNyIA2OFgH42UDp0QBfAJS0AsAChEiAPQrEAOi2LJitYgBCcKAAtEAeQDSitBmz5idBs1aIAREdJup07jY72hEROjCzUbgCCfDAQBG4ANPyCrpHRTEwI3jLcQA)

In JavaScript, if you access a property that doesn’t exist, you’ll get the value undefined rather than a runtime error. Because of this, when you read from an optional property, you’ll have to check for undefined before using it.

function printName(obj: { first: string; last?: string }) {

// Error - might crash if 'obj.last' wasn't provided!

console.log(obj.last.toUpperCase());

'obj.last' is possibly 'undefined'.'obj.last' is possibly 'undefined'.

if (obj.last !== undefined) {

// OK

console.log(obj.last.toUpperCase());

}

// A safe alternative using modern JavaScript syntax:

console.log(obj.last?.toUpperCase());

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygIwA4AMAWDBYAKADMBXAOwGMAXASznNAAdpbzqA5AQwFtIAKOACMAVqgDeoYrSTVUiaq3IBzANygANlwUB+eYrbLQAXwCUocUVCgQoAKKwEoALSgetZQAtqoStG2eoLTEoADkwiIAdFoKoaAA7trkoT4scAButAAmkFkAhFa+DIhwGpDRcMqCotHa1JHUcACqTEwwAMLaAqamqoXBoNVRMT55ALxjoBQ50uS55pa)

## Union Types

TypeScript’s type system allows you to build new types out of existing ones using a large variety of operators. Now that we know how to write a few types, it’s time to start combining them in interesting ways.

### **Defining a Union Type**

The first way to combine types you might see is a union type. A union type is a type formed from two or more other types, representing values that may be any one of those types. We refer to each of these types as the union’s members.

Let’s write a function that can operate on strings or numbers:

function printId(id: number | string) {

console.log("Your ID is: " + id);

}

// OK

printId(101);

// OK

printId("202");

// Error

printId({ myID: 22342 });

Argument of type '{ myID: number; }' is not assignable to parameter of type 'string | number'.Argument of type '{ myID: number; }' is not assignable to parameter of type 'string | number'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFABmArgHYDGALgJZxmgAO0NZVAkgCYAUNnqZEgFsARjFAAfUIiosyAcwCUoAN6FQoCvURwANpAB0uuPO4AiAJpwS0UOwAioGilBnQAaiedFAbkIBfQhBQAHkAaUJmVg4eAEYABljfILBwyLkY8zR4tDNkgmCAUVgEdOiubhVQIQBPB1Q0TCw0UH9fIA)

### **Working with Union Types**

It’s easy to provide a value matching a union type - simply provide a type matching any of the union’s members. If you have a value of a union type, how do you work with it?

TypeScript will only allow an operation if it is valid for every member of the union. For example, if you have the union string | number, you can’t use methods that are only available on string:

function printId(id: number | string) {

console.log(id.toUpperCase());

Property 'toUpperCase' does not exist on type 'string | number'.

Property 'toUpperCase' does not exist on type 'number'.Property 'toUpperCase' does not exist on type 'string | number'.

Property 'toUpperCase' does not exist on type 'number'.

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4CwAoAMwFcA7AYwBcBLOU0AB2mtMoEkATACmo9VOIBbAEYxQAH1CJKzUgHMAlKADeBUKHJ1EcADaQAdDrhyeHfZTgBVBgxgBhAIaJIXBQoDcBAL5A)

The solution is to narrow the union with code, the same as you would in JavaScript without type annotations. Narrowing occurs when TypeScript can deduce a more specific type for a value based on the structure of the code.

For example, TypeScript knows that only a string value will have a typeof value "string":

function printId(id: number | string) {

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

// In this branch, id is of type 'string'

console.log(id.toUpperCase());

} else {

// Here, id is of type 'number'

console.log(id);

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABABwE4zFAkgEwBQw4BciYIAtgEYCmqiAPogM5TpgDmAlIgN4CwAKESIYwRHigBPZNThjCiALzLEAIhZt2q7vyHDEAegOIsSKAAsYTRJVQBDSOYA0InCOtzEUmYgDkGjHZfQX1ECAQmOAAbagA6KLh2AhxYqDgAVWQZVABhOyZqPE5OAG4QxABfRGoogt5y4SNEAAlaahcFK0RPb2o-MipaYL1hcLBImPjE5NLyisEKo)

Another example is to use a function like Array.isArray:

function welcomePeople(x: string[] | string) {

if (Array.isArray(x)) {

// Here: 'x' is 'string[]'

console.log("Hello, " + x.join(" and "));

} else {

// Here: 'x' is 'string'

console.log("Welcome lone traveler " + x);

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAdwKYBsJwLaoAqpwAO6qAFAB4BciAzlAE4xgDmA2gLqIA+djzLAJSIA3gFgAUIkQxgiMgEEGDAIYBPAHQxaS1WsqDh4qdMQB6M4gASqBqhoByCg5m1ED+k1acHk04iwwWjhSDXQ4FjIAIht0cIAaRCjEAGpECg0AKzhmaMQVMAATJMMAbj9EAF9EDFpUUQrpC2tbe3dnV3dPAV8TaUDg0PDIqIB1DCxcRHCwesYVAD)

Notice that in the else branch, we don’t need to do anything special - if x wasn’t a string[], then it must have been a string.

Sometimes you’ll have a union where all the members have something in common. For example, both arrays and strings have a slice method. If every member in a union has a property in common, you can use that property without narrowing:

// Return type is inferred as number[] | string

function getFirstThree(x: number[] | string) {

return x.slice(0, 3);

}

[Try](https://www.typescriptlang.org/play/#code/PTAECUFMBcFcCcB2poE8AOlQEsDOPEAzSeeSAE1AEN9FYBbAIxIG0BdUAH1F2nm0QBzALAAoQrEQBjaNgD2yQTABi2eLwAqACzKQAFAA8AXKDpNWHbr35CAlKADeY0KDJwkoAwDpcAG2xS+gAMADSgAMy2ANxiAL5AA)

It might be confusing that a union of types appears to have the intersection of those types’ properties. This is not an accident - the name union comes from type theory. The union number | string is composed by taking the union of the values from each type. Notice that given two sets with corresponding facts about each set, only the intersection of those facts applies to the union of the sets themselves. For example, if we had a room of tall people wearing hats, and another room of Spanish speakers wearing hats, after combining those rooms, the only thing we know about every person is that they must be wearing a hat.

## Type Aliases

We’ve been using object types and union types by writing them directly in type annotations. This is convenient, but it’s common to want to use the same type more than once and refer to it by a single name.

A type alias is exactly that - a name for any type. The syntax for a type alias is:

type Point = {

x: number;

y: number;

};

// Exactly the same as the earlier example

function printCoord(pt: Point) {

console.log("The coordinate's x value is " + pt.x);

console.log("The coordinate's y value is " + pt.y);

}

printCoord({ x: 100, y: 100 });

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBACg9gSwHbCgXigbwLACgpQAeAXFEgK4C2ARhAE4DceBIpFN9TuAvl3gPT8oAUUIBDAMbAANiCjAAFtADOYytDHL5SqBDF1pCervGUw0iHgBm5JFIRwkUMHWTAAwnDh0AJgAowYFJ4NwBKLGYoCUdlOAsAOmk4AHM-ACIAFR1o7x9kMWAIAHItQigANzFpcmgELTSoAGpnYHjCUK4CaKRYhKTUzOyvX3zCkqg5Sura+qaW+JAO)

You can actually use a type alias to give a name to any type at all, not just an object type. For example, a type alias can name a union type:

type ID = number | string;

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAkgIlAvFAdgVwLYCMICcoA+UAzsLgJYoDmA3EA)

Note that aliases are only aliases - you cannot use type aliases to create different/distinct “versions” of the same type. When you use the alias, it’s exactly as if you had written the aliased type. In other words, this code might look illegal, but is OK according to TypeScript because both types are aliases for the same type:

type UserInputSanitizedString = string;

function sanitizeInput(str: string): UserInputSanitizedString {

return sanitize(str);

}

// Create a sanitized input

let userInput = sanitizeInput(getInput());

// Can still be re-assigned with a string though

userInput = "new input";

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwHMQMBJVAB2QwAoBKALngGcMYtUCBuAWAChRIsBCnTY8zKKizYAXiGosYjRewINmrVT14B6HfAC0RsFSMG+GAJ7kEAVSYgYZShgDKk6VjnBXmjvABeDTYObT4RTFx8Jg9ZEGcqBVZlPzVGe0cEt1ivEB9U+ABvPnh4OAxkGGicuSSYWm0AXz4+PXgAYTgoDAQoCSk44Hh2Fz4IYnhkBycKKkD+zzk)

## Interfaces

An interface declaration is another way to name an object type:

interface Point {

x: number;

y: number;

}

function printCoord(pt: Point) {

console.log("The coordinate's x value is " + pt.x);

console.log("The coordinate's y value is " + pt.y);

}

printCoord({ x: 100, y: 100 });

[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgAoHtRmQbwLABQyyAHgFzIgCuAtgEbQDchxAnhdfU4QL6GEwqIBGGDoQyAA5QsAYXTooAEwAUksBQxYAlLhbIE4gM7oANhAB0p9AHMVAIgAqACxSHFS0HEgByI6WQANzhTKhRgf3tkAGopMAsSbWYiA2MzS2s7J1dUjy9ff1YgkLDkCOQo2PULViTefgJpOQVlFRxSCgBGAAZugBpkdmQe7uQeJKA)

Just like when we used a type alias above, the example works just as if we had used an anonymous object type. TypeScript is only concerned with the structure of the value we passed to printCoord - it only cares that it has the expected properties. Being concerned only with the structure and capabilities of types is why we call TypeScript a structurally typed type system.

### **Differences Between Type Aliases and Interfaces**

Type aliases and interfaces are very similar, and in many cases you can choose between them freely. Almost all features of an interface are available in type, the key distinction is that a type cannot be re-opened to add new properties vs an interface which is always extendable.

|  |  |
| --- | --- |
| **Interface** | **Type** |
| Extending an interface  interface Animal {  name: string;  }  interface Bear extends Animal {  honey: boolean;  }  const bear = getBear();  bear.name;  bear.honey; | Extending a type via intersections  type Animal = {  name: string;  }  type Bear = Animal & {  honey: boolean;  }  const bear = getBear();  bear.name;  bear.honey; |
| Adding new fields to an existing interface  interface Window {  title: string;  }  interface Window {  ts: TypeScriptAPI;  }  const src = 'const a = "Hello World"';  window.ts.transpileModule(src, {}); | A type cannot be changed after being created  type Window = {  title: string;  }  type Window = {  ts: TypeScriptAPI;  }  // Error: Duplicate identifier 'Window'. |

You’ll learn more about these concepts in later chapters, so don’t worry if you don’t understand all of these right away.

* Prior to TypeScript version 4.2, type alias names [may appear in error messages](https://www.typescriptlang.org/play?#code/PTAEGEHsFsAcEsA2BTATqNrLusgzngIYDm+oA7koqIYuYQJ56gCueyoAUCKAC4AWHAHaFcoSADMaQ0PCG80EwgGNkALk6c5C1EtWgAsqOi1QAb06groEbjWg8vVHOKcAvpokshy3vEgyyMr8kEbQJogAFND2YREAlOaW1soBeJAoAHSIkMTRmbbI8e6aPMiZxJmgACqCGKhY6ABGyDnkFFQ0dIzMbBwCwqIccabcYLyQoKjIEmh8kwN8DL), sometimes in place of the equivalent anonymous type (which may or may not be desirable). Interfaces will always be named in error messages.
* Type aliases may not participate [in declaration merging, but interfaces can](https://www.typescriptlang.org/play?#code/PTAEEEDtQS0gXApgJwGYEMDGjSfdAIx2UQFoB7AB0UkQBMAoEUfO0Wgd1ADd0AbAK6IAzizp16ALgYM4SNFhwBZdAFtV-UAG8GoPaADmNAcMmhh8ZHAMMAvjLkoM2UCvWad+0ARL0A-GYWVpA29gyY5JAWLJAwGnxmbvGgALzauvpGkCZmAEQAjABMAMwALLkANBl6zABi6DB8okR4Jjg+iPSgABboovDk3jjo5pbW1d6+dGb5djLwAJ7U).
* Interfaces may only be used to [declare the shapes of objects, not rename primitives](https://www.typescriptlang.org/play?#code/PTAEAkFMCdIcgM6gC4HcD2pIA8CGBbABwBtIl0AzUAKBFAFcEBLAOwHMUBPQs0XFgCahWyGBVwBjMrTDJMAshOhMARpD4tQ6FQCtIE5DWoixk9QEEWAeV37kARlABvaqDegAbrmL1IALlAEZGV2agBfampkbgtrWwMAJlAAXmdXdy8ff0Dg1jZwyLoAVWZ2Lh5QVHUJflAlSFxROsY5fFAWAmk6CnRoLGwmILzQQmV8JmQmDzI-SOiKgGV).
* Interface names will [always appear in their original form](https://www.typescriptlang.org/play?#code/PTAEGEHsFsAcEsA2BTATqNrLusgzngIYDm+oA7koqIYuYQJ56gCueyoAUCKAC4AWHAHaFcoSADMaQ0PCG80EwgGNkALk6c5C1EtWgAsqOi1QAb06groEbjWg8vVHOKcAvpokshy3vEgyyMr8kEbQJogAFND2YREAlOaW1soBeJAoAHSIkMTRmbbI8e6aPMiZxJmgACqCGKhY6ABGyDnkFFQ0dIzMbBwCwqIccabcYLyQoKjIEmh8kwN8DL) in error messages, but only when they are used by name.

For the most part, you can choose based on personal preference, and TypeScript will tell you if it needs something to be the other kind of declaration. If you would like a heuristic, use interface until you need to use features from type.

## Type Assertions

Sometimes you will have information about the type of a value that TypeScript can’t know about.

For example, if you’re using document.getElementById, TypeScript only knows that this will return some kind of HTMLElement, but you might know that your page will always have an HTMLCanvasElement with a given ID.

In this situation, you can use a type assertion to specify a more specific type:

const myCanvas = document.getElementById("main\_canvas") as HTMLCanvasElement;

[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBAtgTwMIEMwDcURgXhgExGAFc4BTMKAOgHMyoBRAGzPMoCEEBJfACgCI4KAJZgA+sDSYI-AJQwsMABIAVALIAZVBizNWFKAG4gA)

Like a type annotation, type assertions are removed by the compiler and won’t affect the runtime behavior of your code.

You can also use the angle-bracket syntax (except if the code is in a .tsx file), which is equivalent:

const myCanvas = <HTMLCanvasElement>document.getElementById("main\_canvas");

[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBAtgTwMIEMwDcURgXhgHgAkAVAWQBlUMsBRAGwFM4GwoA+AExGAFdnWAdAHMGUekxZQAQggCSHABQAiOCgCWYAPrA0mCEoCUAbiA)

Reminder: Because type assertions are removed at compile-time, there is no runtime checking associated with a type assertion. There won’t be an exception or null generated if the type assertion is wrong.

TypeScript only allows type assertions which convert to a more specific or less specific version of a type. This rule prevents “impossible” coercions like:

const x = "hello" as number;

Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.Conversion of type 'string' to type 'number' may be a mistake because neither type sufficiently overlaps with the other. If this was intentional, convert the expression to 'unknown' first.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYCsaCwAoAYzgDtEAXUAD1AF5QAiAC0gBtW4HQBDRUEgK4BbAEYwA3EA)

Sometimes this rule can be too conservative and will disallow more complex coercions that might be valid. If this happens, you can use two assertions, first to any (or unknown, which we’ll introduce later), then to the desired type:

const a = expr as any as T;

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEYD2A7AzgF3iAHgBxgC54oUBPAbgFgAoDMvBAFXgF54BvE4gRgvgBGxAEz8wxAMzwAvtRoB6efAC0qsAFcMq5bWTosUNtnwwSaEuTPwmFIA)

## Literal Types

In addition to the general types string and number, we can refer to specific strings and numbers in type positions.

One way to think about this is to consider how JavaScript comes with different ways to declare a variable. Both var and let allow for changing what is held inside the variable, and const does not. This is reflected in how TypeScript creates types for literals.

let changingString = "Hello World";

changingString = "Olá Mundo";

// Because `changingString` can represent any possible string, that

// is how TypeScript describes it in the type system

changingString;

let changingString: string

const constantString = "Hello World";

// Because `constantString` can only represent 1 possible string, it

// has a literal type representation

constantString;

const constantString: "Hello World"

[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAxgFghgOwOYEtkGUwCc1IgXggCIAJEYYAewgHVKtgATIgbgFgAoWRVDbXAsQDywAIcQAsgFcEjSq04B6RRABCIKHCkBnEBAAG3ZLkw5k+6IghYQABxu6EkRAE8Itytu0oARqAja-MgANBBg8GBKKijaEDCUAO4QACoutiDoUDi2kIwg2lm++RAokGhhMHpgaXraLoEgALacRrxIprjsHMoQAHoA-JwtlAiB0COBiGAdyIKk5FS0)

By themselves, literal types aren’t very valuable:

let x: "hello" = "hello";

// OK

x = "hello";

// ...

x = "howdy";

Type '"howdy"' is not assignable to type '"hello"'.Type '"howdy"' is not assignable to type '"hello"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oLACgAbSAF1AA9UAiAC0gILktAF5Qa6HKBufEUAeQDS+Miza16jHnj4A6eSLE04AdwAmAT25A)

It’s not much use to have a variable that can only have one value!

But by combining literals into unions, you can express a much more useful concept - for example, functions that only accept a certain set of known values:

function printText(s: string, alignment: "left" | "right" | "center") {

// ...

}

printText("Hello, world", "left");

printText("G'day, mate", "centre");

Argument of type '"centre"' is not assignable to parameter of type '"left" | "right" | "center"'.Argument of type '"centre"' is not assignable to parameter of type '"left" | "right" | "center"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFABmArgHYDGALgJZxmgAO0NZVAKpAB5UAUKoRFRZkA5gBpQAQwA2NUWQC2kNqgBEMyESprQAH1BqWogBY79hiiqow1ASlABvQqFAhQAOi+EAvoWasHNx8agASkDIycJIA7ggyACZqkhpaOnYA3P4iQTy8agDiAOQJUgCekopSNsmW1tCQ9hlAA)

Numeric literal types work the same way:

function compare(a: string, b: string): -1 | 0 | 1 {

return a === b ? 0 : a > b ? 1 : -1;

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABBOBbADgQwE4FMAUmAXIgM5TYxgDmANIgEYnmU0CUJAtAIyIA+iAAz9EvAN4BYAFCJEeKCGxJMiALzrGiAPxDEJFQD5NO3l24BuaQF8gA)

Of course, you can combine these with non-literal types:

interface Options {

width: number;

}

function configure(x: Options | "auto") {

// ...

}

configure({ width: 100 });

configure("auto");

configure("automatic");

Argument of type '"automatic"' is not assignable to parameter of type 'Options | "auto"'.Argument of type '"automatic"' is not assignable to parameter of type 'Options | "auto"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFACWAdgC4wBmAhgMaSgDyADmUXCYqAN6GigB3IgBMyAC1QkArgFsARjADchAL6FKUkrTYdQtDpSIBzKdEgAKAB6oWOzqAA+oAETUpZOM4CUPPqBCgAHTBqoT6JIYmZubcgiLiqACMAAzJoCpeygThkaYWru6emWEGxnnmBR4y1Gy03opAA)

There’s one more kind of literal type: boolean literals. There are only two boolean literal types, and as you might guess, they are the types true and false. The type boolean itself is actually just an alias for the union true | false.

### **Literal Inference**

When you initialize a variable with an object, TypeScript assumes that the properties of that object might change values later. For example, if you wrote code like this:

const obj = { counter: 0 };

if (someCondition) {

obj.counter = 1;

}

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEYD2A7AzgF3mpBbEAwqsAJYYmoBc8ARkkhCFCgNwCwAUAPRfwC0AsAFcMAvp2TosSGgCt4AXngBvREiEoMIGNQAM8AL7sOJAGbwAFDnxEUpcqgCUKzvHgzZAOmQatMRfAAjMYGQA)

TypeScript doesn’t assume the assignment of 1 to a field which previously had 0 is an error. Another way of saying this is that obj.counter must have the type number, not 0, because types are used to determine both reading and writing behavior.

The same applies to strings:

declare function handleRequest(url: string, method: "GET" | "POST"): void;

const req = { url: "https://example.com", method: "GET" };

handleRequest(req.url, req.method);

Argument of type 'string' is not assignable to parameter of type '"GET" | "POST"'.Argument of type 'string' is not assignable to parameter of type '"GET" | "POST"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFAAmkAxgDYCG0koAZgK4B2pALgJZxOgAWlTRcpABKkAI4NIiVgAoG0cqmnR2TAOYAaUAFtIrHnCKoARAHEAogBVjoAD6hjABQDyAZWsBKVADc47IgDchISkXNKgNGKgALygAN6g8ooOPKysAA4oIJAAHpTa6UIAdKHaxlq6+oYmFtagAL5BBHwCQqISUrKRRUla3ZUGRB4BQA)

In the above example req.method is inferred to be string, not "GET". Because code can be evaluated between the creation of req and the call of handleRequest which could assign a new string like "GUESS" to req.method, TypeScript considers this code to have an error.

There are two ways to work around this.

1. You can change the inference by adding a type assertion in either location:

// Change 1:

const req = { url: "https://example.com", method: "GET" as "GET" };

// Change 2

handleRequest(req.url, req.method as "GET");

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwAspVgIQAlEAR2RAGcMAKZGCALngZi1QHMAaeAFsQGAjmAcARAHEAogBUp8AD7wpABQDyAZSUBKDgDccWYAG4AsACgA9LfgBaZ2GQZnjm-fgBhInwQARjYbMDwGeDgqeABeeABveBZ2dQIMDAAHOjZ7EAAPKCEMsgA6MKEpQRExCWl5JXgoOnV65QBfKzsHP2JeBAAmG39SCmpaBkYokuTBKerxYE)

Change 1 means “I intend for req.method to always have the literal type "GET"”, preventing the possible assignment of "GUESS" to that field after. Change 2 means “I know for other reasons that req.method has the value "GET"“.

1. You can use as const to convert the entire object to be type literals:

const req = { url: "https://example.com", method: "GET" } as const;

handleRequest(req.url, req.method);

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwAspVgIQAlEAR2RAGcMAKZGCALngZi1QHMAaeAFsQGAjmAcARAHEAogBUp8AD7wpABQDyAZSUBKDgDccWYAG4AsACgA9LfgBaZ2GQZnjm2DwN4cKvAAvPAA3vAs7OoEGBgADnRs9iAAHlBCsWQAdN5CUoIiYhLS8krwAL7wUHTw3qgMVtZEJGSUNPRM-pkRgp0F4sD65kA)

The as const suffix acts like const but for the type system, ensuring that all properties are assigned the literal type instead of a more general version like string or number.

## null and undefined

JavaScript has two primitive values used to signal absent or uninitialized value: null and undefined.

TypeScript has two corresponding types by the same names. How these types behave depends on whether you have the [strictNullChecks](https://www.typescriptlang.org/tsconfig#strictNullChecks) option on.

### **strictNullChecks off**

With [strictNullChecks](https://www.typescriptlang.org/tsconfig#strictNullChecks) off, values that might be null or undefined can still be accessed normally, and the values null and undefined can be assigned to a property of any type. This is similar to how languages without null checks (e.g. C#, Java) behave. The lack of checking for these values tends to be a major source of bugs; we always recommend people turn [strictNullChecks](https://www.typescriptlang.org/tsconfig#strictNullChecks) on if it’s practical to do so in their codebase.

### **strictNullChecks on**

With [strictNullChecks](https://www.typescriptlang.org/tsconfig#strictNullChecks) on, when a value is null or undefined, you will need to test for those values before using methods or properties on that value. Just like checking for undefined before using an optional property, we can use narrowing to check for values that might be null:

function doSomething(x: string | null) {

if (x === null) {

// do nothing

} else {

console.log("Hello, " + x.toUpperCase());

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAEzgZTgWwKZQBYxgDmAFAB4BciAzlAE6FGIA+iYIANhwJSIDeAWABQiRDGCJyiALyy2nHv2GjRAelUo4bOPkbLEAX0TYO1bEpEqICanA7YAdBzikARAAkTzgDSJXiAGpEMgcoOABVAAdI7DoAYQBDMxJubgBufQNhAyA)

### **Non-null Assertion Operator (Postfix !)**

TypeScript also has a special syntax for removing null and undefined from a type without doing any explicit checking. Writing ! after any expression is effectively a type assertion that the value isn’t null or undefined:

function liveDangerously(x?: number | null) {

// No error

console.log(x!.toFixed());

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAGxgNwKYBECGYDmGATnCAM7ICeAFAB4D8AXImCALYBGxiAPiyMmQBKRAG8AsAChEiAPSzEAOTiJiJIlJkQEZOMgwA6ZHHx0AhAahwAYjFoYAJtSFCA3FIC+QA)

Just like other type assertions, this doesn’t change the runtime behavior of your code, so it’s important to only use ! when you know that the value can’t be null or undefined.

## Enums

Enums are a feature added to JavaScript by TypeScript which allows for describing a value which could be one of a set of possible named constants. Unlike most TypeScript features, this is not a type-level addition to JavaScript but something added to the language and runtime. Because of this, it’s a feature which you should know exists, but maybe hold off on using unless you are sure. You can read more about enums in the [Enum reference page](https://www.typescriptlang.org/docs/handbook/enums.html).

## Less Common Primitives

It’s worth mentioning the rest of the primitives in JavaScript which are represented in the type system. Though we will not go into depth here.

#### bigint

From ES2020 onwards, there is a primitive in JavaScript used for very large integers, BigInt:

// Creating a bigint via the BigInt function

const oneHundred: bigint = BigInt(100);

// Creating a BigInt via the literal syntax

const anotherHundred: bigint = 100n;

[Try](https://www.typescriptlang.org/play/#code/PTAEAEBcEMCcHMCmkBcpEGcBMAGXBYAKCJFAGFZFpIBLAO3lGlACMb57JQA3G5yABaJQAIXYBJOlwBmAVzoBjWgHs6RBaoxdViABLyAJpQNo2HKaAC8oiVIAUARhw4AlAG4iJMBSq0GTG3hJLl5+IVAAGxpIRFhoCNAMAE8paAAPdU0uaDplQVj9OiNEE1Z2TitQJxw6NyA)

You can learn more about BigInt in [the TypeScript 3.2 release notes](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-2.html#bigint).

## symbol

There is a primitive in JavaScript used to create a globally unique reference via the function Symbol():

const firstName = Symbol("name");

const secondName = Symbol("name");

if (firstName === secondName) {

This comparison appears to be unintentional because the types 'typeof firstName' and 'typeof secondName' have no overlap.This comparison appears to be unintentional because the types 'typeof firstName' and 'typeof secondName' have no overlap.

// Can't ever happen

}

//strongly typed syntax

let a : string = "Pakistan";

a = "USA";

let b : number = 9;

let c : boolean = true;

//type inference

let e = "USA";

let f = 10.9;

f = 22;

let g = false;

g = true;

# **Variable Declaration**

let and const are two relatively new concepts for variable declarations in JavaScript. [As we mentioned earlier](https://www.typescriptlang.org/docs/handbook/basic-types.html#a-note-about-let), let is similar to var in some respects, but allows users to avoid some of the common “gotchas” that users run into in JavaScript.

const is an augmentation of let in that it prevents re-assignment to a variable.

With TypeScript being an extension of JavaScript, the language naturally supports let and const. Here we’ll elaborate more on these new declarations and why they’re preferable to var.

If you’ve used JavaScript offhandedly, the next section might be a good way to refresh your memory. If you’re intimately familiar with all the quirks of var declarations in JavaScript, you might find it easier to skip ahead.

## var declarations

Declaring a variable in JavaScript has always traditionally been done with the var keyword.

var a = 10;

As you might’ve figured out, we just declared a variable named a with the value 10.

We can also declare a variable inside of a function:

function f() {

var message = "Hello, world!";

return message;

}

and we can also access those same variables within other functions:

function f() {

var a = 10;

return function g() {

var b = a + 1;

return b;

};

}

var g = f();

g(); // returns '11'

In this above example, g captured the variable a declared in f. At any point that g gets called, the value of a will be tied to the value of a in f. Even if g is called once f is done running, it will be able to access and modify a.

function f() {

var a = 1;

a = 2;

var b = g();

a = 3;

return b;

function g() {

return a;

}

}

f(); // returns '2'

### **Scoping rules**

var declarations have some odd scoping rules for those used to other languages. Take the following example:

function f(shouldInitialize: boolean) {

if (shouldInitialize) {

var x = 10;

}

return x;

}

f(true); // returns '10'

f(false); // returns 'undefined'

Some readers might do a double-take at this example. The variable x was declared within the *if* block, and yet we were able to access it from outside that block. That’s because var declarations are accessible anywhere within their containing function, module, namespace, or global scope - all which we’ll go over later on - regardless of the containing block. Some people call this *var*-scoping or function-scoping. Parameters are also function scoped.

These scoping rules can cause several types of mistakes. One problem they exacerbate is the fact that it is not an error to declare the same variable multiple times:

function sumMatrix(matrix: number[][]) {

var sum = 0;

for (var i = 0; i < matrix.length; i++) {

var currentRow = matrix[i];

for (var i = 0; i < currentRow.length; i++) {

sum += currentRow[i];

}

}

return sum;

}

Maybe it was easy to spot out for some experienced JavaScript developers, but the inner for-loop will accidentally overwrite the variable i because i refers to the same function-scoped variable. As experienced developers know by now, similar sorts of bugs slip through code reviews and can be an endless source of frustration.

### **Variable capturing quirks**

Take a quick second to guess what the output of the following snippet is:

for (var i = 0; i < 10; i++) {

setTimeout(function () {

console.log(i);

}, 100 \* i);

}

For those unfamiliar, setTimeout will try to execute a function after a certain number of milliseconds (though waiting for anything else to stop running).

Ready? Take a look:

10

10

10

10

10

10

10

10

10

10

Many JavaScript developers are intimately familiar with this behavior, but if you’re surprised, you’re certainly not alone. Most people expect the output to be

0

1

2

3

4

5

6

7

8

9

Remember what we mentioned earlier about variable capturing? Every function expression we pass to setTimeout actually refers to the same i from the same scope.

Let’s take a minute to consider what that means. setTimeout will run a function after some number of milliseconds, but only after the for loop has stopped executing; By the time the for loop has stopped executing, the value of i is 10. So each time the given function gets called, it will print out 10!

A common work around is to use an IIFE - an Immediately Invoked Function Expression - to capture i at each iteration:

for (var i = 0; i < 10; i++) {

// capture the current state of 'i'

// by invoking a function with its current value

(function (i) {

setTimeout(function () {

console.log(i);

}, 100 \* i);

})(i);

}

This odd-looking pattern is actually pretty common. The i in the parameter list actually shadows the i declared in the for loop, but since we named them the same, we didn’t have to modify the loop body too much.

## let declarations

By now you’ve figured out that var has some problems, which is precisely why let statements were introduced. Apart from the keyword used, let statements are written the same way var statements are.

let hello = "Hello!";

The key difference is not in the syntax, but in the semantics, which we’ll now dive into.

### **Block-scoping**

When a variable is declared using let, it uses what some call lexical-scoping or block-scoping. Unlike variables declared with var whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or for-loop.

function f(input: boolean) {

let a = 100;

if (input) {

// Still okay to reference 'a'

let b = a + 1;

return b;

}

// Error: 'b' doesn't exist here

return b;

}

Here, we have two local variables a and b. a’s scope is limited to the body of f while b’s scope is limited to the containing if statement’s block.

Variables declared in a catch clause also have similar scoping rules.

try {

throw "oh no!";

} catch (e) {

console.log("Oh well.");

}

// Error: 'e' doesn't exist here

console.log(e);

Another property of block-scoped variables is that they can’t be read or written to before they’re actually declared. While these variables are “present” throughout their scope, all points up until their declaration are part of their temporal dead zone. This is just a sophisticated way of saying you can’t access them before the let statement, and luckily TypeScript will let you know that.

a++; // illegal to use 'a' before it's declared;

let a;

Something to note is that you can still capture a block-scoped variable before it’s declared. The only catch is that it’s illegal to call that function before the declaration. If targeting ES2015, a modern runtime will throw an error; however, right now TypeScript is permissive and won’t report this as an error.

function foo() {

// okay to capture 'a'

return a;

}

// illegal call 'foo' before 'a' is declared

// runtimes should throw an error here

foo();

let a;

For more information on temporal dead zones, see relevant content on the [Mozilla Developer Network](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/let#Temporal_dead_zone_and_errors_with_let).

### **Re-declarations and Shadowing**

With var declarations, we mentioned that it didn’t matter how many times you declared your variables; you just got one.

function f(x) {

var x;

var x;

if (true) {

var x;

}

}

In the above example, all declarations of x actually refer to the same x, and this is perfectly valid. This often ends up being a source of bugs. Thankfully, let declarations are not as forgiving.

let x = 10;

let x = 20; // error: can't re-declare 'x' in the same scope

The variables don’t necessarily need to both be block-scoped for TypeScript to tell us that there’s a problem.

function f(x) {

let x = 100; // error: interferes with parameter declaration

}

function g() {

let x = 100;

var x = 100; // error: can't have both declarations of 'x'

}

That’s not to say that a block-scoped variable can never be declared with a function-scoped variable. The block-scoped variable just needs to be declared within a distinctly different block.

function f(condition, x) {

if (condition) {

let x = 100;

return x;

}

return x;

}

f(false, 0); // returns '0'

f(true, 0); // returns '100'

The act of introducing a new name in a more nested scope is called shadowing. It is a bit of a double-edged sword in that it can introduce certain bugs on its own in the event of accidental shadowing, while also preventing certain bugs. For instance, imagine we had written our earlier sumMatrix function using let variables.

function sumMatrix(matrix: number[][]) {

let sum = 0;

for (let i = 0; i < matrix.length; i++) {

var currentRow = matrix[i];

for (let i = 0; i < currentRow.length; i++) {

sum += currentRow[i];

}

}

return sum;

}

This version of the loop will actually perform the summation correctly because the inner loop’s i shadows i from the outer loop.

Shadowing should usually be avoided in the interest of writing clearer code. While there are some scenarios where it may be fitting to take advantage of it, you should use your best judgement.

### **Block-scoped variable capturing**

When we first touched on the idea of variable capturing with var declaration, we briefly went into how variables act once captured. To give a better intuition of this, each time a scope is run, it creates an “environment” of variables. That environment and its captured variables can exist even after everything within its scope has finished executing.

function theCityThatAlwaysSleeps() {

let getCity;

if (true) {

let city = "Seattle";

getCity = function () {

return city;

};

}

return getCity();

}

Because we’ve captured city from within its environment, we’re still able to access it despite the fact that the if block finished executing.

Recall that with our earlier setTimeout example, we ended up needing to use an IIFE to capture the state of a variable for every iteration of the for loop. In effect, what we were doing was creating a new variable environment for our captured variables. That was a bit of a pain, but luckily, you’ll never have to do that again in TypeScript.

let declarations have drastically different behavior when declared as part of a loop. Rather than just introducing a new environment to the loop itself, these declarations sort of create a new scope per iteration. Since this is what we were doing anyway with our IIFE, we can change our old setTimeout example to just use a let declaration.

for (let i = 0; i < 10; i++) {

setTimeout(function () {

console.log(i);

}, 100 \* i);

}

and as expected, this will print out

0

1

2

3

4

5

6

7

8

9

## const declarations

const declarations are another way of declaring variables.

const numLivesForCat = 9;

They are like let declarations but, as their name implies, their value cannot be changed once they are bound. In other words, they have the same scoping rules as let, but you can’t re-assign to them.

This should not be confused with the idea that the values they refer to are immutable.

const numLivesForCat = 9;

const kitty = {

name: "Aurora",

numLives: numLivesForCat,

};

// Error

kitty = {

name: "Danielle",

numLives: numLivesForCat,

};

// all "okay"

kitty.name = "Rory";

kitty.name = "Kitty";

kitty.name = "Cat";

kitty.numLives--;

Unless you take specific measures to avoid it, the internal state of a const variable is still modifiable. Fortunately, TypeScript allows you to specify that members of an object are readonly. The [chapter on Interfaces](https://www.typescriptlang.org/docs/handbook/interfaces.html) has the details.

## let vs. const

Given that we have two types of declarations with similar scoping semantics, it’s natural to find ourselves asking which one to use. Like most broad questions, the answer is: it depends.

Applying the [principle of least privilege](https://wikipedia.org/wiki/Principle_of_least_privilege), all declarations other than those you plan to modify should use const. The rationale is that if a variable didn’t need to get written to, others working on the same codebase shouldn’t automatically be able to write to the object, and will need to consider whether they really need to reassign to the variable. Using const also makes code more predictable when reasoning about flow of data.

Use your best judgement, and if applicable, consult the matter with the rest of your team.

The majority of this handbook uses let declarations.

## Destructuring

Another ECMAScript 2015 feature that TypeScript has is destructuring. For a complete reference, see [the article on the Mozilla Developer Network](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment). In this section, we’ll give a short overview.

### **Array destructuring**

The simplest form of destructuring is array destructuring assignment:

let input = [1, 2];

let [first, second] = input;

console.log(first); // outputs 1

console.log(second); // outputs 2

This creates two new variables named first and second. This is equivalent to using indexing, but is much more convenient:

first = input[0];

second = input[1];

Destructuring works with already-declared variables as well:

// swap variables

[first, second] = [second, first];

And with parameters to a function:

function f([first, second]: [number, number]) {

console.log(first);

console.log(second);

}

f([1, 2]);

You can create a variable for the remaining items in a list using the syntax ...:

let [first, ...rest] = [1, 2, 3, 4];

console.log(first); // outputs 1

console.log(rest); // outputs [ 2, 3, 4 ]

Of course, since this is JavaScript, you can just ignore trailing elements you don’t care about:

let [first] = [1, 2, 3, 4];

console.log(first); // outputs 1

Or other elements:

let [, second, , fourth] = [1, 2, 3, 4];

console.log(second); // outputs 2

console.log(fourth); // outputs 4

### **Tuple destructuring**

Tuples may be destructured like arrays; the destructuring variables get the types of the corresponding tuple elements:

let tuple: [number, string, boolean] = [7, "hello", true];

let [a, b, c] = tuple; // a: number, b: string, c: boolean

It’s an error to destructure a tuple beyond the range of its elements:

let [a, b, c, d] = tuple; // Error, no element at index 3

As with arrays, you can destructure the rest of the tuple with ..., to get a shorter tuple:

let [a, ...bc] = tuple; // bc: [string, boolean]

let [a, b, c, ...d] = tuple; // d: [], the empty tuple

Or ignore trailing elements, or other elements:

let [a] = tuple; // a: number

let [, b] = tuple; // b: string

### **Object destructuring**

You can also destructure objects:

let o = {

a: "foo",

b: 12,

c: "bar",

};

let { a, b } = o;

This creates new variables a and b from o.a and o.b. Notice that you can skip c if you don’t need it.

Like array destructuring, you can have assignment without declaration:

({ a, b } = { a: "baz", b: 101 });

Notice that we had to surround this statement with parentheses. JavaScript normally parses a { as the start of block.

You can create a variable for the remaining items in an object using the syntax ...:

let { a, ...passthrough } = o;

let total = passthrough.b + passthrough.c.length;

#### Property renaming

You can also give different names to properties:

let { a: newName1, b: newName2 } = o;

Here the syntax starts to get confusing. You can read a: newName1 as ”a as newName1”. The direction is left-to-right, as if you had written:

let newName1 = o.a;

let newName2 = o.b;

Confusingly, the colon here does not indicate the type. The type, if you specify it, still needs to be written after the entire destructuring:

let { a: newName1, b: newName2 }: { a: string; b: number } = o;

#### Default values

Default values let you specify a default value in case a property is undefined:

function keepWholeObject(wholeObject: { a: string; b?: number }) {

let { a, b = 1001 } = wholeObject;

}

In this example the b? indicates that b is optional, so it may be undefined. keepWholeObject now has a variable for wholeObject as well as the properties a and b, even if b is undefined.

## Function declarations

Destructuring also works in function declarations. For simple cases this is straightforward:

type C = { a: string; b?: number };

function f({ a, b }: C): void {

// ...

}

But specifying defaults is more common for parameters, and getting defaults right with destructuring can be tricky. First of all, you need to remember to put the pattern before the default value.

function f({ a = "", b = 0 } = {}): void {

// ...

}

f();

The snippet above is an example of type inference, explained earlier in the handbook.

Then, you need to remember to give a default for optional properties on the destructured property instead of the main initializer. Remember that C was defined with b optional:

function f({ a, b = 0 } = { a: "" }): void {

// ...

}

f({ a: "yes" }); // ok, default b = 0

f(); // ok, default to { a: "" }, which then defaults b = 0

f({}); // error, 'a' is required if you supply an argument

Use destructuring with care. As the previous example demonstrates, anything but the simplest destructuring expression is confusing. This is especially true with deeply nested destructuring, which gets really hard to understand even without piling on renaming, default values, and type annotations. Try to keep destructuring expressions small and simple. You can always write the assignments that destructuring would generate yourself.

## Spread

The spread operator is the opposite of destructuring. It allows you to spread an array into another array, or an object into another object. For example:

let first = [1, 2];

let second = [3, 4];

let bothPlus = [0, ...first, ...second, 5];

This gives bothPlus the value [0, 1, 2, 3, 4, 5]. Spreading creates a shallow copy of first and second. They are not changed by the spread.

You can also spread objects:

let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };

let search = { ...defaults, food: "rich" };

Now search is { food: "rich", price: "$$", ambiance: "noisy" }. Object spreading is more complex than array spreading. Like array spreading, it proceeds from left-to-right, but the result is still an object. This means that properties that come later in the spread object overwrite properties that come earlier. So if we modify the previous example to spread at the end:

let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };

let search = { food: "rich", ...defaults };

Then the food property in defaults overwrites food: "rich", which is not what we want in this case.

Object spread also has a couple of other surprising limits. First, it only includes an objects’ [own, enumerable properties](https://developer.mozilla.org/docs/Web/JavaScript/Enumerability_and_ownership_of_properties). Basically, that means you lose methods when you spread instances of an object:

class C {

p = 12;

m() {}

}

let c = new C();

let clone = { ...c };

clone.p; // ok

clone.m(); // error!

Second, the TypeScript compiler doesn’t allow spreads of type parameters from generic functions. That feature is expected in future versions of the language.

## using declarations

using declarations are an upcoming feature for JavaScript that are part of the [Stage 3 Explicit Resource Management](https://github.com/tc39/proposal-explicit-resource-management) proposal. A using declaration is much like a const declaration, except that it couples the lifetime of the value bound to the declaration with the scope of the variable.

When control exits the block containing a using declaration, the [Symbol.dispose]() method of the declared value is executed, which allows that value to perform cleanup:

function f() {

using x = new C();

doSomethingWith(x);

} // `x[Symbol.dispose]()` is called

At runtime, this has an effect roughly equivalent to the following:

function f() {

const x = new C();

try {

doSomethingWith(x);

}

finally {

x[Symbol.dispose]();

}

}

using declarations are extremely useful for avoiding memory leaks when working with JavaScript objects that hold on to native references like file handles

{

using file = await openFile();

file.write(text);

doSomethingThatMayThrow();

} // `file` is disposed, even if an error is thrown

or scoped operations like tracing

function f() {

using activity = new TraceActivity("f"); // traces entry into function

// ...

} // traces exit of function

Unlike var, let, and const, using declarations do not support destructuring.

### **null and undefined**

It’s important to note that the value can be null or undefined, in which case nothing is disposed at the end of the block:

{

using x = b ? new C() : null;

// ...

}

which is roughly equivalent to:

{

const x = b ? new C() : null;

try {

// ...

}

finally {

x?.[Symbol.dispose]();

}

}

This allows you to conditionally acquire resources when declaring a using declaration without the need for complex branching or repetition.

### **Defining a disposable resource**

You can indicate the classes or objects you produce are disposable by implementing the Disposable interface:

// from the default lib:

interface Disposable {

[Symbol.dispose](): void;

}

// usage:

class TraceActivity implements Disposable {

readonly name: string;

constructor(name: string) {

this.name = name;

console.log(`Entering: ${name}`);

}

[Symbol.dispose](): void {

console.log(`Exiting: ${name}`);

}

}

function f() {

using \_activity = new TraceActivity("f");

console.log("Hello world!");

}

f();

// prints:

// Entering: f

// Hello world!

// Exiting: f

## await using declarations

Some resources or operations may have cleanup that needs to be performed asynchronously. To accommodate this, the [Explicit Resource Management](https://github.com/tc39/proposal-explicit-resource-management) proposal also introduces the await using declaration:

async function f() {

await using x = new C();

} // `await x[Symbol.asyncDispose]()` is invoked

An await using declaration invokes, and awaits, its value’s [Symbol.asyncDispose]() method as control leaves the containing block. This allows for asynchronous cleanup, such as a database transaction performing a rollback or commit, or a file stream flushing any pending writes to storage before it is closed.

As with await, await using can only be used in an async function or method, or at the top level of a module.

### **Defining an asynchronously disposable resource**

Just as using relies on objects that are Disposable, an await using relies on objects that are AsyncDisposable:

// from the default lib:

interface AsyncDisposable {

[Symbol.asyncDispose]: PromiseLike<void>;

}

// usage:

class DatabaseTransaction implements AsyncDisposable {

public success = false;

private db: Database | undefined;

private constructor(db: Database) {

this.db = db;

}

static async create(db: Database) {

await db.execAsync("BEGIN TRANSACTION");

return new DatabaseTransaction(db);

}

async [Symbol.asyncDispose]() {

if (this.db) {

const db = this.db:

this.db = undefined;

if (this.success) {

await db.execAsync("COMMIT TRANSACTION");

}

else {

await db.execAsync("ROLLBACK TRANSACTION");

}

}

}

}

async function transfer(db: Database, account1: Account, account2: Account, amount: number) {

using tx = await DatabaseTransaction.create(db);

if (await debitAccount(db, account1, amount)) {

await creditAccount(db, account2, amount);

}

// if an exception is thrown before this line, the transaction will roll back

tx.success = true;

// now the transaction will commit

}

### **await using vs await**

The await keyword that is part of the await using declaration only indicates that the disposal of the resource is await-ed. It does not await the value itself:

{

await using x = getResourceSynchronously();

} // performs `await x[Symbol.asyncDispose]()`

{

await using y = await getResourceAsynchronously();

} // performs `await y[Symbol.asyncDispose]()`

### **await using and return**

It’s important to note that there is a small caveat with this behavior if you are using an await using declaration in an async function that returns a Promise without first await-ing it:

function g() {

return Promise.reject("error!");

}

async function f() {

await using x = new C();

return g(); // missing an `await`

}

Because the returned promise isn’t await-ed, it’s possible that the JavaScript runtime may report an unhandled rejection since execution pauses while await-ing the asynchronous disposal of x, without having subscribed to the returned promise. This is not a problem that is unique to await using, however, as this can also occur in an async function that uses try..finally:

async function f() {

try {

return g(); // also reports an unhandled rejection

}

finally {

await somethingElse();

}

}

To avoid this situation, it is recommended that you await your return value if it may be a Promise:

async function f() {

await using x = new C();

return await g();

}

## using and await using in for and for..of statements

Both using and await using can be used in a for statement:

for (using x = getReader(); !x.eof; x.next()) {

// ...

}

In this case, the lifetime of x is scoped to the entire for statement and is only disposed when control leaves the loop due to break, return, throw, or when the loop condition is false.

In addition to for statements, both declarations can also be used in for..of statements:

function \* g() {

yield createResource1();

yield createResource2();

}

for (using x of g()) {

// ...

}

Here, x is disposed at the end of each iteration of the loop, and is then reinitialized with the next value. This is especially useful when consuming resources produced one at a time by a generator.

## using and await using in older runtimes

using and await using declarations can be used when targeting older ECMAScript editions as long as you are using a compatible polyfill for Symbol.dispose/Symbol.asyncDispose, such as the one provided by default in recent editions of NodeJS.

## About Number, String, Boolean, Symbol and Object

It can be tempting to think that the types Number, String, Boolean, Symbol, or Object are the same as the lowercase versions recommended above. These types do not refer to the language primitives however, and almost never should be used as a type.

function reverse(s: String): String {

return s.split("").reverse().join("");

}

reverse("hello world");

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4BQAzAVwDsBjAFwEs5jRpIA3GRSAChVAGVzpLiBzAJSpuvAaADeuUHUjlC0WogB0iAA4AbSuVYAiXYOX0mSNoYBWcPnoMBuXAF9cuY8za6AFpA0a4oAO4IGgAmdkA)

Instead, use the types number, string, boolean, object and symbol.

function reverse(s: string): string {

return s.split("").reverse().join("");

}

reverse("hello world");

//use const where variable values do not change

const a = 5;

const b : number = 33;

const c ="best";

//I suggest use let instead of var everywhere,

//becuase let has blocked scope

if (true) {

let z = 4;

//use z

}

else {

let z = "string";

//use z

}

console.log("let: " + z);// Error: z is not defined in this scope

# **Modules**

JavaScript has a long history of different ways to handle modularizing code. Having been around since 2012, TypeScript has implemented support for a lot of these formats, but over time the community and the JavaScript specification has converged on a format called ES Modules (or ES6 modules). You might know it as the import/export syntax.

ES Modules was added to the JavaScript spec in 2015, and by 2020 had broad support in most web browsers and JavaScript runtimes.

For focus, the handbook will cover both ES Modules and its popular pre-cursor CommonJS module.exports = syntax, and you can find information about the other module patterns in the reference section under [Modules](https://www.typescriptlang.org/docs/handbook/modules.html).

## How JavaScript Modules are Defined

In TypeScript, just as in ECMAScript 2015, any file containing a top-level import or export is considered a module.

Conversely, a file without any top-level import or export declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Modules are executed within their own scope, not in the global scope. This means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the export forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the import forms.

## Non-modules

Before we start, it’s important to understand what TypeScript considers a module. The JavaScript specification declares that any JavaScript files without an import declaration, export, or top-level await should be considered a script and not a module.

Inside a script file variables and types are declared to be in the shared global scope, and it’s assumed that you’ll either use the [outFile](https://www.typescriptlang.org/tsconfig#outFile) compiler option to join multiple input files into one output file, or use multiple <script> tags in your HTML to load these files (in the correct order!).

If you have a file that doesn’t currently have any imports or exports, but you want to be treated as a module, add the line:

export {};

[Try](https://www.typescriptlang.org/play/#code/KYDwDg9gTgLgBAbwL4G4g)

which will change the file to be a module exporting nothing. This syntax works regardless of your module target.

## Modules in TypeScript

Additional Reading:  
[Impatient JS (Modules)](https://exploringjs.com/impatient-js/ch_modules.html#overview-syntax-of-ecmascript-modules)  
[MDN: JavaScript Modules](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)

There are three main things to consider when writing module-based code in TypeScript:

* **Syntax**: What syntax do I want to use to import and export things?
* **Module Resolution**: What is the relationship between module names (or paths) and files on disk?
* **Module Output Target**: What should my emitted JavaScript module look like?

### **ES Module Syntax**

A file can declare a main export via export default:

// @filename: hello.ts

export default function helloWorld() {

console.log("Hello, world!");

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKAFvWsB7AOgBcBnAWACh4APABwICcTQATeSZAV1lcm6IAxiWgFEOPIQDqzWGwAUASlABvaqFBDxZAgiKEA5goBEACSkEANKADuctgEITSgNzUAvkA)

This is then imported via:

import helloWorld from "./hello.js";

helloWorld();

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKAFvWsB7AOgBcBnAWACh4APABwICcTQATeSZAV1lcm6IAxiWgFEOPIQDqzWGwAUASlABvaqFBDxZAgiKEA5goBEACSkEANKADuctgEITSgNzUAvtRAQYCFOhY0IgctKSUVD4AtDFC3CQxUdTQqIwskvgEskzyoJBMBKigJkTAuJlEAFZkJu5U5TIOyq5AA)

In addition to the default export, you can have more than one export of variables and functions via the export by omitting default:

// @filename: maths.ts

export var pi = 3.14;

export let squareTwo = 1.41;

export const phi = 1.61;

export class RandomNumberGenerator {}

export function absolute(num: number) {

if (num < 0) return num \* -1;

return num;

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKVyAuALAZwDpdCBYAKHgA8AHAewCddQA3ZJ0O6UAXlABmYgEYALAG4qtRi1AJWhAI4BXTvAAqAdwb9QI4mJFTq9ZqwDGDRIVZ18vAQYBsxqtLNyLsZIUKgAJWREABMGVAA5FVQAI3gmAHEkeLxmUABvAF93U1lWSBVEC1xoa1BkGMIGWBVceAAKRGisJtj4gEoMqlBQaEhQRujQAB5QAAZOpnhcFSZEUFbQ)

These can be used in another file via the import syntax:

import { pi, phi, absolute } from "./maths.js";

console.log(pi);

const absPhi = absolute(phi);

const absPhi: number

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKVyAuALAZwDpdCBYAKHgA8AHAewCddQA3ZJ0O6UAXlABmYgEYALAG4qtRi1AJWhAI4BXTvAAqAdwb9QI4mJFTq9ZqwDGDRIVZ18vAQYBsx6WbkXYyQoVAAlZEQAEwZUADkVVAAjeCYAcSQ4vGZQAG8AX3dZVkgVRAtcaGtQZGjCBlgVXHgACkQorAaYuIBKdKpQUGhIUHqo0AAeUAAGdqZ4XBUmRFBm0AAq)

### **Additional Import Syntax**

An import can be renamed using a format like import {old as new}:

import { pi as π } from "./maths.js";

console.log(π);

(alias) var π: number

import π

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKVyAuALAZwDpdCBYAKHgA8AHAewCddQA3ZJ0O6UAXlABmYgEYALAG4qICDAQp0WZHTqkKlGQFptAYwCuubZqrRUjFqADe3XskKhAA8CgAvqEhMGqUACJiwHAQkAFaE3lKUVDoMiIQMCMSwDADmABQOAJThMqA5ubkAegD8QA)

You can mix and match the above syntax into a single import:

// @filename: maths.ts

export const pi = 3.14;

export default class RandomNumberGenerator {}

// @filename: app.ts

import RandomNumberGenerator, { pi as π } from "./maths.js";

RandomNumberGenerator;

(alias) class RandomNumberGenerator

import RandomNumberGenerator

console.log(π);

(alias) const π: 3.14

import π

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKVyAuALAZwDpdCBYAKHgA8AHAewCddQBjBxQ1u6UAXlABmYgEYALAG4qtRi1AATeJGQBXWKzaxkhQqABKyRAoaoAcqtQAjeEwDiSW3magA3gF8qVEBBgIU6FjIdHSkFJTQqHKshsamFta2DohOuMwANG6gvKA6oIADwKDuoJBMpqAARMTAOAQkAFaEFdKUVLEm5pY29o5MzkwtPgB6APxelBxcDAjEsAwA5)

You can take all of the exported objects and put them into a single namespace using \* as name:

// @filename: app.ts

import \* as math from "./maths.js";

console.log(math.pi);

const positivePhi = math.absolute(math.phi);

const positivePhi: number

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKVyAuALAZwDpdCBYAKHgA8AHAewCddQA3ZJ0O6UAXlABmYgEYALAG4qtRi1AJWhAI4BXTvAAqAdwb9QI4mJFTq9ZqwDGDRIVZ18vAQYBsxqtLNzIKxBdzRrUGQAI0IGWBVceAAKRBVULDjUYPgmAEpQAG8qUFBoSFBY+NAAHlAABgymeFwVJkRQJNAAKlAAWjdKXOra+sb4kwBfKhB2traLSPG2kbAoOCQ0)

You can import a file and not include any variables into your current module via import "./file":

// @filename: app.ts

import "./maths.js";

console.log("3.14");

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKVyAuALAZwDpdCBYAKHgA8AHAewCddQA3ZJ0O6UAXlABmYgEYALAG4qIUAFp5AYwCuuebOlgocJGkyhkdOqQqVoqRi1AAiYsBwESAK0JWplKgoaJCDBMVgMAOYAFFbC4lYAlBJAA)

In this case, the import does nothing. However, all of the code in maths.ts was evaluated, which could trigger side-effects which affect other objects.

#### TypeScript Specific ES Module Syntax

Types can be exported and imported using the same syntax as JavaScript values:

// @filename: animal.ts

export type Cat = { breed: string; yearOfBirth: number };

export interface Dog {

breeds: string[];

yearOfBirth: number;

}

// @filename: app.ts

import { Cat, Dog } from "./animal.js";

type Animals = Cat | Dog;

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKZjqubAHQAuAzgLABQ8AHgA4D2ATsaMQJ53ygDCyrAXlABvUACMm8eABMspYk2iIA5gG5Q7eMiYB5SACFoLABZZEAV1Rj4TUAF9VVKrUYtQS4jcjIAxtwAiDMoiVKDikjKkcgpKygDaALqOlGGa2nqGJmaW1kzJdk6UIBAwCCjoWMh0dCQUlHiurKJ8xAA0oIHBdqCQTAyooABEhMA4eASEAFakg8kcXKAA)

TypeScript has extended the import syntax with two concepts for declaring an import of a type:

###### import type

Which is an import statement which can only import types:

// @filename: animal.ts

export type Cat = { breed: string; yearOfBirth: number };

export type Dog = { breeds: string[]; yearOfBirth: number };

export const createCatName = () => "fluffy";

// @filename: valid.ts

import type { Cat, Dog } from "./animal.js";

export type Animals = Cat | Dog;

// @filename: app.ts

import type { createCatName } from "./animal.js";

const name = createCatName();

'createCatName' cannot be used as a value because it was imported using 'import type'.'createCatName' cannot be used as a value because it was imported using 'import type'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKZjqubAHQAuAzgLABQ8AHgA4D2ATsaMQJ53ygDCyrAXlABvUACMm8eABMspYk2iIA5gG5Q7eMiYB5SACFoLABZZEAV1Rj4TUAF9VVWoxZtO3ACINloIaIlS0qRyCkrKANoAuuqa2nqGJmaW1rYOTvTMrADGDIjyoFmS-PB8xAByaNxCABQAlL4AfKAARJCw5pCQ7M2OlFQgEDAIKOhYAG4E0NIkFJR4Lqwc)

###### Inline type imports

TypeScript 4.5 also allows for individual imports to be prefixed with type to indicate that the imported reference is a type:

// @filename: app.ts

import { createCatName, type Cat, type Dog } from "./animal.js";

export type Animals = Cat | Dog;

const name = createCatName();

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKZjqubAHQAuAzgLABQ8AHgA4D2ATsaMQJ53ygDCyrAXlABvUACMm8eABMspYk2iIA5gG5Q7eMiYB5SACFoLABZZEAV1Rj4TUAF9VVWoxZtO3ACINloIaIlS0qRyCkrKANoAuuqa2nqGJmaW1rYOTvTMrADGDIjyoFmS-PB8xAByaNxCABQAlL4AfKAARJCw5pCQ7M2OlCCgALRDWebEQwNU-VBwSJVYyHR0)

Together these allow a non-TypeScript transpiler like Babel, swc or esbuild to know what imports can be safely removed.

#### ES Module Syntax with CommonJS Behavior

TypeScript has ES Module syntax which directly correlates to a CommonJS and AMD require. Imports using ES Module are for most cases the same as the require from those environments, but this syntax ensures you have a 1 to 1 match in your TypeScript file with the CommonJS output:

import fs = require("fs");

const code = fs.readFileSync("hello.ts", "utf8");

[Try](https://www.typescriptlang.org/play/#code/PQgEB4CcFMDNpgOwMbVAFwJ4AdoGcBeAIkQHsATaI0YAPgFgAoMAAQFsKBXAG2gC5QyUmw6IAVniZgAtLOSd0s6UwCWbbKUjpQsPKAKgYAR04qYACiK6iASgDcTIYjzahlfTrwA6GAENyAGIqvADKmCiWABbQ3NykXuh4RAA0oEQKsAActnZAA)

You can learn more about this syntax in the [modules reference page](https://www.typescriptlang.org/docs/handbook/modules.html#export--and-import--require).

## CommonJS Syntax

CommonJS is the format which most modules on npm are delivered in. Even if you are writing using the ES Modules syntax above, having a brief understanding of how CommonJS syntax works will help you debug easier.

#### Exporting

Identifiers are exported via setting the exports property on a global called module.

function absolute(num: number) {

if (num < 0) return num \* -1;

return num;

}

module.exports = {

pi: 3.14,

squareTwo: 1.41,

phi: 1.61,

absolute,

};

[Try](https://www.typescriptlang.org/play/#code/PQgEB4CcFMDNpgOwMbVAFwJ4AdoGcBeAIkQHsATaI0YAPgFgAoMAWjeQFd02WnYOU6AJalEoAIYAjPKQA2XaAApEHALYAuUCtWSEASlABvJqFBDYoZWoigADAZjoOkMdtAAqUCwCMAbhOgjs6uav6MAL5MTKoUHLLQAHTQAB7YpJDoeKAERgHYQpoAzAneACwANAF4AI4c4jAAKgDupJreCaXelYym2AAWBaDtAGxdAVIy8ujQ3eG+QA)

Then these files can be imported via a require statement:

const maths = require("./maths");

maths.pi;

any

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFsHsBMFcA2BTAXKAxtSMB2ArAZwFgAoECAMwEsVcBDSNUSegFwAtCA6Nk8igB4ATskrJRuDMlBsAngAdkhALwAiXHGRrQwAHxlK8KW2rRcoegCNC0RPDbIAFLniR0ryFYkBKUAG8yUFBqSlAXN1BBUAAGP1E2eGELT1AAKlAAWgBGAG4g0ASklLd80gBfMjIYBBRuZAAPBWhhPlAVAIKFanQAZm5sgBYAGgLCAEd4elEAFQB3aHR)

Or you can simplify a bit using the destructuring feature in JavaScript:

const { squareTwo } = require("./maths");

squareTwo;

const squareTwo: any

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFsHsBMFcA2BTAXKAxtSMB2ArAZwFgAoECAMwEsVcBDSNUSegFwAtCA6Nk8igB4ATskrJRuDMlBsAngAdkhALwAiXHGRrQwAHxlK8KW2rRcoegCNC0RPDbIAFLniR0ryFYkBKUAG8yUFBqSlAXN1BBUAAGP1E2eGELT1AAKlAAWgBGAG4g0ASklLd80gBfMjIYBBRuZAAPBWhhPlAVAIKFanQAZm5sgBYAGgLCAEd4elEAFQB3aHR)

### **CommonJS and ES Modules interop**

There is a mis-match in features between CommonJS and ES Modules regarding the distinction between a default import and a module namespace object import. TypeScript has a compiler flag to reduce the friction between the two different sets of constraints with [esModuleInterop](https://www.typescriptlang.org/tsconfig#esModuleInterop).

## TypeScript’s Module Resolution Options

Module resolution is the process of taking a string from the import or require statement, and determining what file that string refers to.

TypeScript includes two resolution strategies: Classic and Node. Classic, the default when the compiler option [module](https://www.typescriptlang.org/tsconfig#module) is not commonjs, is included for backwards compatibility. The Node strategy replicates how Node.js works in CommonJS mode, with additional checks for .ts and .d.ts.

There are many TSConfig flags which influence the module strategy within TypeScript: [moduleResolution](https://www.typescriptlang.org/tsconfig#moduleResolution), [baseUrl](https://www.typescriptlang.org/tsconfig#baseUrl), [paths](https://www.typescriptlang.org/tsconfig#paths), [rootDirs](https://www.typescriptlang.org/tsconfig#rootDirs).

For the full details on how these strategies work, you can consult the [Module Resolution](https://www.typescriptlang.org/docs/handbook/modules/reference.html#the-moduleresolution-compiler-option) reference page.

## TypeScript’s Module Output Options

There are two options which affect the emitted JavaScript output:

* [target](https://www.typescriptlang.org/tsconfig#target) which determines which JS features are downleveled (converted to run in older JavaScript runtimes) and which are left intact
* [module](https://www.typescriptlang.org/tsconfig#module) which determines what code is used for modules to interact with each other

Which [target](https://www.typescriptlang.org/tsconfig#target) you use is determined by the features available in the JavaScript runtime you expect to run the TypeScript code in. That could be: the oldest web browser you support, the lowest version of Node.js you expect to run on or could come from unique constraints from your runtime - like Electron for example.

All communication between modules happens via a module loader, the compiler option [module](https://www.typescriptlang.org/tsconfig#module) determines which one is used. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it.

For example, here is a TypeScript file using ES Modules syntax, showcasing a few different options for [module](https://www.typescriptlang.org/tsconfig#module):

import { valueOfPi } from "./constants.js";

export const twoPi = valueOfPi \* 2;

[Try](https://www.typescriptlang.org/play/#code/PTAEAEDMEsBsFMB2BDAtvAXKAxge0QM4AuyiRBAdOQLABQ8AHgA64BORO+xoAbsrAFd4AeUgAFaKAC8oAMwUAjABYATAG46ICDAQp0WaIgAmjKgU1gAtNewCi1y3WioW7UAG9e-IaImgAvqCQrLiooABEFMB4hCRklABWBOEatHSMrhwx3EQA7rh+MnyCIuKSAFSg6kA)

#### ES2020

import { valueOfPi } from "./constants.js";

export const twoPi = valueOfPi \* 2;

[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcAsHsHcCiBbAlgFwLACgQWbACYCuANgKYBco5kATAAyM57gB2siATl7F5DlTIADn3SgA3qABuAQ1LFyAeQBmABVSgAvqBW9koAEQA6YAGNYbSOllt0kYwCtIhgNw4c5AB6iu4i1bi6PCwGqAAvDLyiqphAFSgdK5AA)

#### CommonJS

"use strict";

Object.defineProperty(exports, "\_\_esModule", { value: true });

exports.twoPi = void 0;

const constants\_js\_1 = require("./constants.js");

exports.twoPi = constants\_js\_1.valueOfPi \* 2;

[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcAsHsHcCiBbAlgFwLACgQWbACYCuANgKYBcoAxrMgQHYBWkOe4jsiATj7DzbZUyAA4D0oAN6gAbgENSxcgHkAZgAVUoAL6g1-ZKABEAOmB1GkdPMbpIp1sYDcOHOQAe4npMvXQ6PCwWqAAvHKKyuohAFSgAEzOQA)

#### UMD

(function (factory) {

if (typeof module === "object" && typeof module.exports === "object") {

var v = factory(require, exports);

if (v !== undefined) module.exports = v;

}

else if (typeof define === "function" && define.amd) {

define(["require", "exports", "./constants.js"], factory);

}

})(function (require, exports) {

"use strict";

Object.defineProperty(exports, "\_\_esModule", { value: true });

exports.twoPi = void 0;

const constants\_js\_1 = require("./constants.js");

exports.twoPi = constants\_js\_1.valueOfPi \* 2;

});

[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcAsHsHcCiBbAlgFwLACgQWbACYCuANgKYBcoxyhOe4AdrIgE5uxuQ6rIAOXdKADeoAG4BDUsXIB5AGYAFVKAC+oBZ2SgARADpgAY1hNI6SU3SR9AK0i6A3DhzkAHoLbCTZ4eniwKqAAvBLSsopBAFSgAEyOQA)

Note that ES2020 is effectively the same as the original index.ts.

You can see all of the available options and what their emitted JavaScript code looks like in the [TSConfig Reference for module](https://www.typescriptlang.org/tsconfig#module).

## TypeScript namespaces

TypeScript has its own module format called namespaces which pre-dates the ES Modules standard. This syntax has a lot of useful features for creating complex definition files, and still sees active use [in DefinitelyTyped](https://github.com/DefinitelyTyped/DefinitelyTyped). While not deprecated, the majority of the features in namespaces exist in ES Modules and we recommend you use that to align with JavaScript’s direction. You can learn more about namespaces in [the namespaces reference page](https://www.typescriptlang.org/docs/handbook/namespaces.html).

# Using Native ECMAScript Modules in Node.js

# Using Inquirer Package

The latest version (9+) of [Inquirer](https://github.com/SBoudrias/Inquirer.js/) has start using Native ECMA Script Packages. In most of our projects and assignment we will use this package.

Give the following command:

npm i inquirer

npm i --save-dev @types/inquirer

Add .gitignore file and Write your code in app.ts file.

Give the following commands:

tsc

node app.js

import inquirer from "inquirer";

let answers = await inquirer.prompt([{

name: "age",

type: "number",

message: "Enter your Age:"}

]);

console.log("Insha Allah, in " + (60 - answers.age) + " years you will be 60 years old.");

A collection of common interactive command line user interfaces.

Give it a try in your own terminal!

npx @inquirer/demo@latest

# Installation

npm install @inquirer/prompts

yarn add @inquirer/prompts

Inquirer recently underwent a rewrite from the ground up to reduce the package size and improve performance. The previous version of the package is still maintained (though not actively developed), and offered hundreds of community contributed prompts that might not have been migrated to the latest API. If this is what you're looking for, the [previous package is over here](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/inquirer).

# Usage

import { input } from '@inquirer/prompts';

const answer = await input({ message: 'Enter your name' });

# Prompts

## [Input](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/input)

import { input } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/input) for usage example and options documentation.

## [Select](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/select)

import { select } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/select) for usage example and options documentation.

## [Checkbox](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/checkbox)

import { checkbox } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/checkbox) for usage example and options documentation.

## [Confirm](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/confirm)

import { confirm } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/confirm) for usage example and options documentation.

## [Password](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/password)

import { password } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/password) for usage example and options documentation.

## [Expand](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/expand)

import { expand } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/expand) for usage example and options documentation.

## [Editor](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/editor)

Launches an instance of the users preferred editor on a temporary file. Once the user exits their editor, the content of the temporary file is read as the answer. The editor used is determined by reading the $VISUAL or $EDITOR environment variables. If neither of those are present, the OS default is used (notepad on Windows, vim on Mac or Linux.)

import { editor } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/editor) for usage example and options documentation.

## [Raw List](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/rawlist)

import { rawlist } from '@inquirer/prompts';

[See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/rawlist) for usage example and options documentation.

# Create your own prompts

The [API documentation is over here](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/core), and our [testing utilities here](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/testing).

# Advanced usage

All inquirer prompts are a function taking 2 arguments. The first argument is the prompt configuration (unique to each prompt). The second is providing contextual or runtime configuration.

The context options are:

| **Property** | **Type** | **Required** | **Description** |
| --- | --- | --- | --- |
| input | NodeJS.ReadableStream | no | The stdin stream (defaults to process.stdin) |
| output | NodeJS.WritableStream | no | The stdout stream (defaults to process.stdout) |
| clearPromptOnDone | boolean | no | If true, we'll clear the screen after the prompt is answered |

Example:

import { confirm } from '@inquirer/prompts';

const allowEmail = await confirm(

{ message: 'Do you allow us to send you email?' },

{

output: new Stream.Writable({

write(chunk, \_encoding, next) {

// Do something

next();

},

}),

clearPromptOnDone: true,

},

);

## Canceling prompt

All prompt functions are returning a cancelable promise. This special promise type has a cancel method that'll cancel and cleanup the prompt.

On calling cancel, the answer promise will become rejected.

import { confirm } from '@inquirer/prompts';

const answer = confirm(...); // note: for this you cannot use `await`

answer.cancel();

# Recipes

## Get answers in an object

When asking many questions, you might not want to keep one variable per answer everywhere. In which case, you can put the answer inside an object.

import { input, confirm } from '@inquirer/prompts';

const answers = {

firstName: await input({ message: "What's your first name?" }),

allowEmail: await confirm({ message: 'Do you allow us to send you email?' }),

};

console.log(answers.firstName);

## Ask a question conditionally

Maybe some questions depend on some other question's answer.

import { input, confirm } from '@inquirer/prompts';

const allowEmail = await confirm({ message: 'Do you allow us to send you email?' });

let email;

if (allowEmail) {

email = await input({ message: 'What is your email address' });

}

## Get default value after timeout

import { setTimeout } from 'node:timers/promises';

import { input } from '@inquirer/prompts';

const ac = new AbortController();

const prompt = input({

message: 'Enter a value (timing out in 5 seconds)',

});

prompt

.finally(() => {

ac.abort();

})

// Silencing the cancellation error.

.catch(() => {});

const defaultValue = setTimeout(5000, 'timeout', { signal: ac.signal }).then(() => {

prompt.cancel();

return 'Timed out!';

});

const answer = await Promise.race([defaultValue, prompt]);

## Using as pre-commit/git hooks, or scripts

By default scripts ran from tools like husky/lint-staged might not run inside an interactive shell. In non-interactive shell, Inquirer cannot run, and users cannot send keypress events to the process.

For it to work, you must make sure you start a tty (or "interactive" input stream.)

If those scripts are set within your package.json, you can define the stream like so:

"precommit": "my-script < /dev/tty"

Or if in a shell script file, you'll do it like so: (on Windows that's likely your only option)

#!/bin/sh

exec < /dev/tty

node my-script.js

## Wait for config

Maybe some question configuration require to await a value.

import { confirm } from '@inquirer/prompts';

const answer = await confirm({ message: await getMessage() });

# Community prompts

If you created a cool prompt, [send us a PR adding it](https://github.com/SBoudrias/Inquirer.js/edit/master/README.md) to the list below!

[**Interactive List Prompt**](https://github.com/pgibler/inquirer-interactive-list-prompt)  
Select a choice either with arrow keys + Enter or by pressing a key associated with a choice.

? Choose an option:

> Run command (D)

Quit (Q)

[**Action Select Prompt**](https://github.com/zenithlight/inquirer-action-select)  
Choose an item from a list and choose an action to take by pressing a key.

? Choose a file Open <O> Edit <E> Delete <X>

❯ image.png

audio.mp3

code.py

[**Table Multiple Prompt**](https://github.com/Bartheleway/inquirer-table-multiple)  
Select multiple answer from a table display.

Choose between choices? (Press <space> to select, <Up and Down> to move rows,

<Left and Right> to move columns)

┌──────────┬───────┬───────┐

│ 1-2 of 2 │ Yes? │ No? |

├──────────┼───────┼───────┤

│ Choice 1 │ [ ◯ ] │ ◯ |

├──────────┼───────┼───────┤

│ Choice 2 │ ◯ │ ◯ |

└──────────┴───────┴───────┘

[**Toggle Prompt**](https://github.com/skarahoda/inquirer-toggle)  
Confirm with a toggle. Select a choice with arrow keys + Enter.

? Do you want to continue? no / yes

[**Sortable Checkbox Prompt**](https://github.com/th0r/inquirer-sortable-checkbox)  
The same as built-in checkbox prompt, but also allowing to reorder choices using ctrl+up/down.

? Which PRs and in what order would you like to merge? (Press <space> to select, <a> to toggle all, <i> to invert selection, <ctrl+up> to move item up, <ctrl+down> to move item down, and <enter> to proceed)

❯ ◯ PR 1

◯ PR 2

◯ PR 3

[**Multi Select Prompt**](https://github.com/jeffwcx/inquirer-select-pro)

An inquirer select that supports multiple selections and filtering/searching.

? Choose your OS, IDE, PL, etc. (Press <tab> to select/deselect, <backspace> to remove selected

option, <enter> to select option)

>> vue

>[ ] vue

[ ] vuejs

[ ] fuelphp

[ ] venv

[ ] vercel

(Use arrow keys to reveal more options)

A collection of common interactive command line user interfaces.

## Table of Contents

1. [Documentation](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#documentation)
   1. [Installation](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#installation)
   2. [Examples](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#examples)
   3. [Methods](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#methods)
   4. [Objects](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#objects)
   5. [Question](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#question)
   6. [Answers](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#answers)
   7. [Separator](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#separator)
   8. [Prompt Types](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#prompt-types)
2. [User Interfaces and Layouts](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#user-interfaces-and-layouts)
   1. [Reactive Interface](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#reactive-interface)
3. [Support](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#support)
4. [Known issues](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#issues)
5. [News](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#news)
6. [Contributing](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#contributing)
7. [License](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#license)
8. [Plugins](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#plugins)

## Goal and Philosophy

**Inquirer.js** strives to be an easily embeddable and beautiful command line interface for [Node.js](https://nodejs.org/) (and perhaps the "CLI [Xanadu](https://en.wikipedia.org/wiki/Citizen_Kane)").

**Inquirer.js** should ease the process of

* providing error feedback
* asking questions
* parsing input
* validating answers
* managing hierarchical prompts

**Note:** **Inquirer.js** provides the user interface and the inquiry session flow. If you're searching for a full blown command line program utility, then check out [commander](https://github.com/visionmedia/commander.js), [vorpal](https://github.com/dthree/vorpal) or [args](https://github.com/leo/args).

## [Documentation](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#documentation)

### Installation

npm install --save inquirer

import inquirer from 'inquirer';

inquirer

.prompt([

/\* Pass your questions in here \*/

])

.then((answers) => {

// Use user feedback for... whatever!!

})

.catch((error) => {

if (error.isTtyError) {

// Prompt couldn't be rendered in the current environment

} else {

// Something else went wrong

}

});

Inquirer v9 and higher are native esm modules, this mean you cannot use the commonjs syntax require('inquirer') anymore. If you want to learn more about using native esm in Node, I'd recommend reading [the following guide](https://gist.github.com/sindresorhus/a39789f98801d908bbc7ff3ecc99d99c). Alternatively, you can rely on an older version until you're ready to upgrade your environment:

npm install --save inquirer@^8.0.0

This will then allow import inquirer with the commonjs require:

const inquirer = require('inquirer');

### Examples (Run it and see it)

Check out the [packages/inquirer/examples/](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/inquirer/examples) folder for code and interface examples.

node packages/inquirer/examples/pizza.js

node packages/inquirer/examples/checkbox.js

# etc...

### Methods

#### inquirer.prompt(questions, answers) -> promise

Launch the prompt interface (inquiry session)

* **questions** (Array) containing [Question Object](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#question) (using the [reactive interface](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#reactive-interface), you can also pass a Rx.Observable instance)
* **answers** (object) contains values of already answered questions. Inquirer will avoid asking answers already provided here. Defaults {}.
* returns a **Promise**

#### inquirer.registerPrompt(name, prompt)

Register prompt plugins under name.

* **name** (string) name of the this new prompt. (used for question type)
* **prompt** (object) the prompt object itself (the plugin)

#### inquirer.createPromptModule() -> prompt function

Create a self contained inquirer module. If you don't want to affect other libraries that also rely on inquirer when you overwrite or add new prompt types.

const prompt = inquirer.createPromptModule();

prompt(questions).then(/\* ... \*/);

### Objects

#### Question

A question object is a hash containing question related values:

* **type**: (String) Type of the prompt. Defaults: input - Possible values: input, number, confirm, list, rawlist, expand, checkbox, password, editor
* **name**: (String) The name to use when storing the answer in the answers hash. If the name contains periods, it will define a path in the answers hash.
* **message**: (String|Function) The question to print. If defined as a function, the first parameter will be the current inquirer session answers. Defaults to the value of name (followed by a colon).
* **default**: (String|Number|Boolean|Array|Function) Default value(s) to use if nothing is entered, or a function that returns the default value(s). If defined as a function, the first parameter will be the current inquirer session answers.
* **choices**: (Array|Function) Choices array or a function returning a choices array. If defined as a function, the first parameter will be the current inquirer session answers. Array values can be simple numbers, strings, or objects containing a name (to display in list), a value (to save in the answers hash), and a short (to display after selection) properties. The choices array can also contain [a Separator](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#separator).
* **validate**: (Function) Receive the user input and answers hash. Should return true if the value is valid, and an error message (String) otherwise. If false is returned, a default error message is provided.
* **filter**: (Function) Receive the user input and answers hash. Returns the filtered value to be used inside the program. The value returned will be added to the Answers hash.
* **transformer**: (Function) Receive the user input, answers hash and option flags, and return a transformed value to display to the user. The transformation only impacts what is shown while editing. It does not modify the answers hash.
* **when**: (Function, Boolean) Receive the current user answers hash and should return true or false depending on whether or not this question should be asked. The value can also be a simple boolean.
* **pageSize**: (Number) Change the number of lines that will be rendered when using list, rawList, expand or checkbox.
* **prefix**: (String) Change the default prefix message.
* **suffix**: (String) Change the default suffix message.
* **askAnswered**: (Boolean) Force to prompt the question if the answer already exists.
* **loop**: (Boolean) Enable list looping. Defaults: true
* **waitUserInput**: (Boolean) Flag to enable/disable wait for user input before opening system editor - Defaults: true

default, choices(if defined as functions), validate, filter and when functions can be called asynchronously. Either return a promise or use this.async() to get a callback you'll call with the final value.

{

/\* Preferred way: with promise \*/

filter() {

return new Promise(/\* etc... \*/);

},

/\* Legacy way: with this.async \*/

validate: function (input) {

// Declare function as asynchronous, and save the done callback

const done = this.async();

// Do async stuff

setTimeout(function() {

if (typeof input !== 'number') {

// Pass the return value in the done callback

done('You need to provide a number');

} else {

// Pass the return value in the done callback

done(null, true);

}

}, 3000);

}

}

### Answers

A key/value hash containing the client answers in each prompt.

* **Key** The name property of the question object
* **Value** (Depends on the prompt)
  + confirm: (Boolean)
  + input : User input (filtered if filter is defined) (String)
  + number: User input (filtered if filter is defined) (Number)
  + rawlist, list : Selected choice value (or name if no value specified) (String)

### Separator

A separator can be added to any choices array:

// In the question object

choices: [ "Choice A", new inquirer.Separator(), "choice B" ]

// Which'll be displayed this way

[?] What do you want to do?

> Order a pizza

Make a reservation

--------

Ask opening hours

Talk to the receptionist

The constructor takes a facultative String value that'll be use as the separator. If omitted, the separator will be --------.

Separator instances have a property type equal to separator. This should allow tools façading Inquirer interface from detecting separator types in lists.

### Prompt types

**Note:**: allowed options written inside square brackets (*[]*) are optional. Others are required.

#### List - {type: 'list'}

Take type, name, message, choices[, default, filter, loop] properties. (Note: default must be set to the index or value of one of the entries in choices)

#### Raw List - {type: 'rawlist'}

Take type, name, message, choices[, default, filter, loop] properties. (Note: default must be set to the index of one of the entries in choices)

#### Expand - {type: 'expand'}

Take type, name, message, choices[, default] properties. Note: default must be the index of the desired default selection of the array. If default key not provided, then help will be used as default choice

Note that the choices object will take an extra parameter called key for the expand prompt. This parameter must be a single (lowercased) character. The h option is added by the prompt and shouldn't be defined by the user.

See examples/expand.js for a running example.

#### Checkbox - {type: 'checkbox'}

Take type, name, message, choices[, filter, validate, default, loop] properties. default is expected to be an Array of the checked choices value.

Choices marked as {checked: true} will be checked by default.

Choices whose property disabled is truthy will be unselectable. If disabled is a string, then the string will be outputted next to the disabled choice, otherwise it'll default to "Disabled". The disabled property can also be a synchronous function receiving the current answers as argument and returning a boolean or a string.

#### Confirm - {type: 'confirm'}

Take type, name, message, [default, transformer] properties. default is expected to be a boolean if used.

#### Input - {type: 'input'}

Take type, name, message[, default, filter, validate, transformer] properties.

#### Input - {type: 'number'}

Take type, name, message[, default, filter, validate, transformer] properties.

#### Password - {type: 'password'}

Take type, name, message, mask,[, default, filter, validate] properties.

Note that mask is required to hide the actual user input.

#### Editor - {type: 'editor'}

Take type, name, message[, default, filter, validate, postfix, waitUserInput] properties

Launches an instance of the users preferred editor on a temporary file. Once the user exits their editor, the contents of the temporary file are read in as the result. The editor to use is determined by reading the $VISUAL or $EDITOR environment variables. If neither of those are present, notepad (on Windows) or vim (Linux or Mac) is used.

The postfix property is useful if you want to provide an extension.

### Use in Non-Interactive Environments

prompt() requires that it is run in an interactive environment. (I.e. [One where process.stdin.isTTY is true](https://nodejs.org/docs/latest-v12.x/api/process.html#process_a_note_on_process_i_o)). If prompt() is invoked outside of such an environment, then prompt() will return a rejected promise with an error. For convenience, the error will have a isTtyError property to programmatically indicate the cause.

## User Interfaces and layouts

Along with the prompts, Inquirer offers some basic text UI.

#### Bottom Bar - inquirer.ui.BottomBar

This UI present a fixed text at the bottom of a free text zone. This is useful to keep a message to the bottom of the screen while outputting command outputs on the higher section.

const ui = new inquirer.ui.BottomBar();

// pipe a Stream to the log zone

outputStream.pipe(ui.log);

// Or simply write output

ui.log.write('something just happened.');

ui.log.write('Almost over, standby!');

// During processing, update the bottom bar content to display a loader

// or output a progress bar, etc

ui.updateBottomBar('new bottom bar content');

## Reactive interface

Internally, Inquirer uses the [JS reactive extension](https://github.com/ReactiveX/rxjs) to handle events and async flows.

This mean you can take advantage of this feature to provide more advanced flows. For example, you can dynamically add questions to be asked:

const prompts = new Rx.Subject();

inquirer.prompt(prompts);

// At some point in the future, push new questions

prompts.next({

/\* question... \*/

});

prompts.next({

/\* question... \*/

});

// When you're done

prompts.complete();

And using the return value process property, you can access more fine grained callbacks:

inquirer.prompt(prompts).ui.process.subscribe(onEachAnswer, onError, onComplete);

## Support (OS Terminals)

You should expect mostly good support for the CLI below. This does not mean we won't look at issues found on other command line - feel free to report any!

* **Mac OS**:
  + Terminal.app
  + iTerm
* **Windows (**[**Known issues**](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/inquirer#issues)**)**:
  + [Windows Terminal](https://github.com/microsoft/terminal)
  + [ConEmu](https://conemu.github.io/)
  + cmd.exe
  + Powershell
  + Cygwin
* **Linux (Ubuntu, openSUSE, Arch Linux, etc)**:
  + gnome-terminal (Terminal GNOME)
  + konsole

## Known issues

* **nodemon** - Makes the arrow keys print gibrish on list prompts. Workaround: Add { stdin : false } in the configuration file or pass --no-stdin in the CLI. Please refer to [this issue](https://github.com/SBoudrias/Inquirer.js/issues/844#issuecomment-736675867)
* **grunt-exec** - Calling a node script that uses Inquirer from grunt-exec can cause the program to crash. To fix this, add to your grunt-exec config stdio: 'inherit'. Please refer to [this issue](https://github.com/jharding/grunt-exec/issues/85)
* **Windows network streams** - Running Inquirer together with network streams in Windows platform inside some terminals can result in process hang. Workaround: run inside another terminal. Please refer to [this issue](https://github.com/nodejs/node/issues/21771)

## News on the march (Release notes)

Please refer to the [GitHub releases section for the changelog](https://github.com/SBoudrias/Inquirer.js/releases)

## Contributing

**Unit test** Please add a unit test for every new feature or bug fix. npm test to run the test suite.

**Documentation** Add documentation for every API change. Feel free to send typo fixes and better docs!

We're looking to offer good support for multiple prompts and environments. If you want to help, we'd like to keep a list of testers for each terminal/OS so we can contact you and get feedback before release. Let us know if you want to be added to the list (just tweet to [@vaxilart](https://twitter.com/Vaxilart)) or just add your name to [the wiki](https://github.com/SBoudrias/Inquirer.js/wiki/Testers)

## License

Copyright (c) 2023 Simon Boudrias (twitter: [@vaxilart](https://twitter.com/Vaxilart))  
Licensed under the MIT license.

## Plugins

### Prompts

[**autocomplete**](https://github.com/mokkabonna/inquirer-autocomplete-prompt)  
Presents a list of options as the user types, compatible with other packages such as fuzzy (for search)  
  
[**checkbox-plus**](https://github.com/faressoft/inquirer-checkbox-plus-prompt)  
Checkbox list with autocomplete and other additions

[**inquirer-date-prompt**](https://github.com/haversnail/inquirer-date-prompt)  
Customizable date/time selector with localization support

[**datetime**](https://github.com/DerekTBrown/inquirer-datepicker-prompt)  
Customizable date/time selector using both number pad and arrow keys  
  
[Datetime Prompt](https://github.com/DerekTBrown/inquirer-datepicker-prompt/raw/master/example/datetime-prompt.png)

[**inquirer-select-line**](https://github.com/adam-golab/inquirer-select-line)  
Prompt for selecting index in array where add new element  
  
[**command**](https://github.com/sullof/inquirer-command-prompt)  
Simple prompt with command history and dynamic autocomplete

[**inquirer-fuzzy-path**](https://github.com/adelsz/inquirer-fuzzy-path)  
Prompt for fuzzy file/directory selection.  
  
[**inquirer-emoji**](https://github.com/tannerntannern/inquirer-emoji)  
Prompt for inputting emojis.

[**inquirer-chalk-pipe**](https://github.com/LitoMore/inquirer-chalk-pipe)  
Prompt for input chalk-pipe style strings

[**inquirer-search-checkbox**](https://github.com/clinyong/inquirer-search-checkbox)  
Searchable Inquirer checkbox

[**inquirer-search-list**](https://github.com/robin-rpr/inquirer-search-list)  
Searchable Inquirer list  
  
[**inquirer-prompt-suggest**](https://github.com/olistic/inquirer-prompt-suggest)  
Inquirer prompt for your less creative users.  
  
[**inquirer-s3**](https://github.com/HQarroum/inquirer-s3)  
An S3 object selector for Inquirer.  
  
[**inquirer-autosubmit-prompt**](https://github.com/yaodingyd/inquirer-autosubmit-prompt)  
Auto submit based on your current input, saving one extra enter

[**inquirer-file-tree-selection-prompt**](https://github.com/anc95/inquirer-file-tree-selection)  
Inquirer prompt for to select a file or directory in file tree  
  
[**inquirer-tree-prompt**](https://github.com/insightfuls/inquirer-tree-prompt)  
Inquirer prompt to select from a tree  
  
[**inquirer-table-prompt**](https://github.com/eduardoboucas/inquirer-table-prompt)  
A table-like prompt for Inquirer.

[**inquirer-table-input**](https://github.com/edelciomolina/inquirer-table-input)  
A table editing prompt for Inquirer.

[**inquirer-interrupted-prompt**](https://github.com/lnquy065/inquirer-interrupted-prompt)  
Turning any existing inquirer and its plugin prompts into prompts that can be interrupted with a custom key.  
  
[**inquirer-press-to-continue**](https://github.com/leonzalion/inquirer-press-to-continue)  
A "press any key to continue" prompt for Inquirer.js

Interactive free text input component for command line interfaces. Supports validation, filtering, transformation, etc.

# Installation

npm install @inquirer/input

yarn add @inquirer/input

# Usage

import input from '@inquirer/input';

const answer = await input({ message: 'Enter your name' });

## Options

| **Property** | **Type** | **Required** | **Description** |
| --- | --- | --- | --- |
| message | string | yes | The question to ask |
| default | string | no | Default value if no answer is provided (clear it by pressing backspace) |
| transformer | (string, { isFinal: boolean }) => string | no | Transform/Format the raw value entered by the user. Once the prompt is completed, isFinal will be true. This function is purely visual, modify the answer in your code if needed. |
| validate | string => boolean | string | Promise<string | boolean> | no | On submit, validate the filtered answered content. When returning a string, it'll be used as the error message displayed to the user. Note: returning a rejected promise, we'll assume a code error happened and crash. |
| theme | [See Theming](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/input#Theming) | no | Customize look of the prompt. |

## Theming

You can theme a prompt by passing a theme object option. The theme object only need to includes the keys you wish to modify, we'll fallback on the defaults for the rest.

type Theme = {

prefix: string;

spinner: {

interval: number;

frames: string[];

};

style: {

answer: (text: string) => string;

message: (text: string) => string;

error: (text: string) => string;

defaultAnswer: (text: string) => string;

};

};

# @inquirer/checkbox

Simple interactive command line prompt to display a list of checkboxes (multi select).

# Installation

npm install @inquirer/checkbox

yarn add @inquirer/checkbox

# Usage

import checkbox, { Separator } from '@inquirer/checkbox';

const answer = await checkbox({

message: 'Select a package manager',

choices: [

{ name: 'npm', value: 'npm' },

{ name: 'yarn', value: 'yarn' },

new Separator(),

{ name: 'pnpm', value: 'pnpm', disabled: true },

{

name: 'pnpm',

value: 'pnpm',

disabled: '(pnpm is not available)',

},

],

});

## Options

| **Property** | **Type** | **Required** | **Description** |
| --- | --- | --- | --- |
| message | string | yes | The question to ask |
| choices | Array<{ value: string, name?: string, disabled?: boolean | string, checked?: boolean } | Separator> | yes | List of the available choices. The value will be returned as the answer, and used as display if no name is defined. Choices who're disabled will be displayed, but not selectable. |
| pageSize | number | no | By default, lists of choice longer than 7 will be paginated. Use this option to control how many choices will appear on the screen at once. |
| loop | boolean | no | Defaults to true. When set to false, the cursor will be constrained to the top and bottom of the choice list without looping. |
| required | boolean | no | When set to true, ensures at least one choice must be selected. |
| validate | string\[\] => boolean | string | Promise<string | boolean> | no | On submit, validate the choices. When returning a string, it'll be used as the error message displayed to the user. Note: returning a rejected promise, we'll assume a code error happened and crash. |
| theme | [See Theming](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/checkbox#Theming) | no | Customize look of the prompt. |

The Separator object can be used to render non-selectable lines in the choice list. By default it'll render a line, but you can provide the text as argument (new Separator('-- Dependencies --')). This option is often used to add labels to groups within long list of options.

## Theming

You can theme a prompt by passing a theme object option. The theme object only need to includes the keys you wish to modify, we'll fallback on the defaults for the rest.

type Theme = {

prefix: string;

spinner: {

interval: number;

frames: string[];

};

style: {

answer: (text: string) => string;

message: (text: string) => string;

error: (text: string) => string;

defaultAnswer: (text: string) => string;

help: (text: string) => string;

highlight: (text: string) => string;

key: (text: string) => string;

disabledChoice: (text: string) => string;

renderSelectedChoices: <T>(

selectedChoices: ReadonlyArray<Choice<T>>,

allChoices: ReadonlyArray<Choice<T> | Separator>,

) => string;

};

icon: {

checked: string;

unchecked: string;

cursor: string;

};

helpMode: 'always' | 'never' | 'auto';

};

### theme.helpMode

* auto (default): Hide the help tips after an interaction occurs. The scroll tip will hide after any interactions, the selection tip will hide as soon as a first selection is done.
* always: The help tips will always show and never hide.
* never: The help tips will never show.
* Simple interactive command line prompt to gather boolean input from users.

# Installation

* npm install @inquirer/confirm
* yarn add @inquirer/confirm

# Usage

* import confirm from '@inquirer/confirm';
* const answer = await confirm({ message: 'Continue?' });

## Options

| **Property** | **Type** | **Required** | **Description** |
| --- | --- | --- | --- |
| message | string | yes | The question to ask |
| default | boolean | no | Default answer (true or false) |
| transformer | (boolean) => string | no | Transform the prompt printed message to a custom string |
| theme | [See Theming](https://github.com/SBoudrias/Inquirer.js/tree/main/packages/confirm#Theming) | no | Customize look of the prompt. |

## Theming

* You can theme a prompt by passing a theme object option. The theme object only need to includes the keys you wish to modify, we'll fallback on the defaults for the rest.
* type Theme = {
* prefix: string;
* spinner: {
* interval: number;
* frames: string[];
* };
* style: {
* answer: (text: string) => string;
* message: (text: string) => string;
* defaultAnswer: (text: string) => string;
* };
* };
* A collection of common interactive command line user interfaces.
* Give it a try in your own terminal!
* npx @inquirer/demo@latest

# Installation

* npm install @inquirer/prompts
* yarn add @inquirer/prompts
* Inquirer recently underwent a rewrite from the ground up to reduce the package size and improve performance. The previous version of the package is still maintained (though not actively developed), and offered hundreds of community contributed prompts that might not have been migrated to the latest API. If this is what you're looking for, the [previous package is over here](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/inquirer).

# Usage

* import { input } from '@inquirer/prompts';
* const answer = await input({ message: 'Enter your name' });

# Prompts

## [Input](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/input)

* import { input } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/input) for usage example and options documentation.

## [Select](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/select)

* import { select } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/select) for usage example and options documentation.

## [Checkbox](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/checkbox)

* import { checkbox } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/checkbox) for usage example and options documentation.

## [Confirm](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/confirm)

* import { confirm } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/confirm) for usage example and options documentation.

## [Password](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/password)

* import { password } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/password) for usage example and options documentation.

## [Expand](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/expand)

* import { expand } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/expand) for usage example and options documentation.

## [Editor](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/editor)

* Launches an instance of the users preferred editor on a temporary file. Once the user exits their editor, the content of the temporary file is read as the answer. The editor used is determined by reading the $VISUAL or $EDITOR environment variables. If neither of those are present, the OS default is used (notepad on Windows, vim on Mac or Linux.)
* import { editor } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/editor) for usage example and options documentation.

## [Raw List](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/rawlist)

* import { rawlist } from '@inquirer/prompts';
* [See documentation](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/rawlist) for usage example and options documentation.

# Create your own prompts

* The [API documentation is over here](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/core), and our [testing utilities here](https://github.com/SBoudrias/Inquirer.js/tree/master/packages/testing).

# Advanced usage

* All inquirer prompts are a function taking 2 arguments. The first argument is the prompt configuration (unique to each prompt). The second is providing contextual or runtime configuration.
* The context options are:

| **Property** | **Type** | **Required** | **Description** |
| --- | --- | --- | --- |
| input | NodeJS.ReadableStream | no | The stdin stream (defaults to process.stdin) |
| output | NodeJS.WritableStream | no | The stdout stream (defaults to process.stdout) |
| clearPromptOnDone | boolean | no | If true, we'll clear the screen after the prompt is answered |

* Example:
* import { confirm } from '@inquirer/prompts';
* const allowEmail = await confirm(
* { message: 'Do you allow us to send you email?' },
* {
* output: new Stream.Writable({
* write(chunk, \_encoding, next) {
* // Do something
* next();
* },
* }),
* clearPromptOnDone: true,
* },
* );

## Canceling prompt

* All prompt functions are returning a cancelable promise. This special promise type has a cancel method that'll cancel and cleanup the prompt.
* On calling cancel, the answer promise will become rejected.
* import { confirm } from '@inquirer/prompts';
* const answer = confirm(...); // note: for this you cannot use `await`
* answer.cancel();

# Recipes

## Get answers in an object

* When asking many questions, you might not want to keep one variable per answer everywhere. In which case, you can put the answer inside an object.
* import { input, confirm } from '@inquirer/prompts';
* const answers = {
* firstName: await input({ message: "What's your first name?" }),
* allowEmail: await confirm({ message: 'Do you allow us to send you email?' }),
* };
* console.log(answers.firstName);

## Ask a question conditionally

* Maybe some questions depend on some other question's answer.
* import { input, confirm } from '@inquirer/prompts';
* const allowEmail = await confirm({ message: 'Do you allow us to send you email?' });
* let email;
* if (allowEmail) {
* email = await input({ message: 'What is your email address' });
* }

## Get default value after timeout

* import { setTimeout } from 'node:timers/promises';
* import { input } from '@inquirer/prompts';
* const ac = new AbortController();
* const prompt = input({
* message: 'Enter a value (timing out in 5 seconds)',
* });
* prompt
* .finally(() => {
* ac.abort();
* })
* // Silencing the cancellation error.
* .catch(() => {});
* const defaultValue = setTimeout(5000, 'timeout', { signal: ac.signal }).then(() => {
* prompt.cancel();
* return 'Timed out!';
* });
* const answer = await Promise.race([defaultValue, prompt]);

## Using as pre-commit/git hooks, or scripts

* By default scripts ran from tools like husky/lint-staged might not run inside an interactive shell. In non-interactive shell, Inquirer cannot run, and users cannot send keypress events to the process.
* For it to work, you must make sure you start a tty (or "interactive" input stream.)
* If those scripts are set within your package.json, you can define the stream like so:
* "precommit": "my-script < /dev/tty"
* Or if in a shell script file, you'll do it like so: (on Windows that's likely your only option)
* #!/bin/sh
* exec < /dev/tty
* node my-script.js

## Wait for config

* Maybe some question configuration require to await a value.
* import { confirm } from '@inquirer/prompts';
* const answer = await confirm({ message: await getMessage() });

# Community prompts

* If you created a cool prompt, [send us a PR adding it](https://github.com/SBoudrias/Inquirer.js/edit/master/README.md) to the list below!
* [**Interactive List Prompt**](https://github.com/pgibler/inquirer-interactive-list-prompt)  
  Select a choice either with arrow keys + Enter or by pressing a key associated with a choice.
* ? Choose an option:
* > Run command (D)
* Quit (Q)
* [**Action Select Prompt**](https://github.com/zenithlight/inquirer-action-select)  
  Choose an item from a list and choose an action to take by pressing a key.
* ? Choose a file Open <O> Edit <E> Delete <X>
* ❯ image.png
* audio.mp3
* code.py
* [**Table Multiple Prompt**](https://github.com/Bartheleway/inquirer-table-multiple)  
  Select multiple answer from a table display.
* Choose between choices? (Press <space> to select, <Up and Down> to move rows,
* <Left and Right> to move columns)
* ┌──────────┬───────┬───────┐
* │ 1-2 of 2 │ Yes? │ No? |
* ├──────────┼───────┼───────┤
* │ Choice 1 │ [ ◯ ] │ ◯ |
* ├──────────┼───────┼───────┤
* │ Choice 2 │ ◯ │ ◯ |
* └──────────┴───────┴───────┘
* [**Toggle Prompt**](https://github.com/skarahoda/inquirer-toggle)  
  Confirm with a toggle. Select a choice with arrow keys + Enter.
* ? Do you want to continue? no / yes
* [**Sortable Checkbox Prompt**](https://github.com/th0r/inquirer-sortable-checkbox)  
  The same as built-in checkbox prompt, but also allowing to reorder choices using ctrl+up/down.
* ? Which PRs and in what order would you like to merge? (Press <space> to select, <a> to toggle all, <i> to invert selection, <ctrl+up> to move item up, <ctrl+down> to move item down, and <enter> to proceed)
* ❯ ◯ PR 1
* ◯ PR 2
* ◯ PR 3
* [**Multi Select Prompt**](https://github.com/jeffwcx/inquirer-select-pro)
* An inquirer select that supports multiple selections and filtering/searching.
* ? Choose your OS, IDE, PL, etc. (Press <tab> to select/deselect, <backspace> to remove selected
* option, <enter> to select option)
* >> vue
* >[ ] vue
* [ ] vuejs
* [ ] fuelphp
* [ ] venv
* [ ] vercel
* (Use arrow keys to reveal more options)
* The latest version of [Inquirer](https://github.com/SBoudrias/Inquirer.js/)(9+) and [Chalk](https://github.com/chalk/chalk)(5+) have started using Native ECMA Script Packages.
* In most of our projects and assignment we will use these packages.
* Give the following command:
* npm i inquirer
* npm i --save-dev @types/inquirer
* npm install chalk
* Add .gitignore file and Write your code in app.ts file.
* Give the following commands:
* tsc
* node app.js

## You can develop [CLI Games](https://www.youtube.com/watch?v=_oHByo8tiEY) using Inquirer, Chalk, etc. All your class projects will be CLI based and should be deployed as an [NPX command](https://blog.deepgram.com/npx-script/)

* [How to create a NPX tool](https://blog.shahednasser.com/how-to-create-a-npx-tool/)
* Run the following command to experience a CLI game:
* npx firequiz

## Highlights

* Expressive API
* Highly performant
* No dependencies
* Ability to nest styles
* [256/Truecolor color support](https://github.com/chalk/chalk#256-and-truecolor-color-support)
* Auto-detects color support
* Doesn't extend String.prototype
* Clean and focused
* Actively maintained
* [Used by ~86,000 packages](https://www.npmjs.com/browse/depended/chalk) as of October 4, 2022

## Install

npm install chalk

**IMPORTANT:** Chalk 5 is ESM. If you want to use Chalk with TypeScript or a build tool, you will probably want to use Chalk 4 for now. [Read more.](https://github.com/chalk/chalk/releases/tag/v5.0.0)

## Usage

import chalk from 'chalk';

console.log(chalk.blue('Hello world!'));

Chalk comes with an easy to use composable API where you just chain and nest the styles you want.

import chalk from 'chalk';

const log = console.log;

// Combine styled and normal strings

log(chalk.blue('Hello') + ' World' + chalk.red('!'));

// Compose multiple styles using the chainable API

log(chalk.blue.bgRed.bold('Hello world!'));

// Pass in multiple arguments

log(chalk.blue('Hello', 'World!', 'Foo', 'bar', 'biz', 'baz'));

// Nest styles

log(chalk.red('Hello', chalk.underline.bgBlue('world') + '!'));

// Nest styles of the same type even (color, underline, background)

log(chalk.green(

'I am a green line ' +

chalk.blue.underline.bold('with a blue substring') +

' that becomes green again!'

));

// ES2015 template literal

log(`

CPU: ${chalk.red('90%')}

RAM: ${chalk.green('40%')}

DISK: ${chalk.yellow('70%')}

`);

// Use RGB colors in terminal emulators that support it.

log(chalk.rgb(123, 45, 67).underline('Underlined reddish color'));

log(chalk.hex('#DEADED').bold('Bold gray!'));

Easily define your own themes:

import chalk from 'chalk';

const error = chalk.bold.red;

const warning = chalk.hex('#FFA500'); // Orange color

console.log(error('Error!'));

console.log(warning('Warning!'));

Take advantage of console.log [string substitution](https://nodejs.org/docs/latest/api/console.html#console_console_log_data_args):

import chalk from 'chalk';

const name = 'Sindre';

console.log(chalk.green('Hello %s'), name);

//=> 'Hello Sindre'

## API

### chalk.<style>[.<style>...](string, [string...])

Example: chalk.red.bold.underline('Hello', 'world');

Chain [styles](https://github.com/chalk/chalk#styles) and call the last one as a method with a string argument. Order doesn't matter, and later styles take precedent in case of a conflict. This simply means that chalk.red.yellow.green is equivalent to chalk.green.

Multiple arguments will be separated by space.

### chalk.level

Specifies the level of color support.

Color support is automatically detected, but you can override it by setting the level property. You should however only do this in your own code as it applies globally to all Chalk consumers.

If you need to change this in a reusable module, create a new instance:

import {Chalk} from 'chalk';

const customChalk = new Chalk({level: 0});

| **Level** | **Description** |
| --- | --- |
| 0 | All colors disabled |
| 1 | Basic color support (16 colors) |
| 2 | 256 color support |
| 3 | Truecolor support (16 million colors) |

### supportsColor

Detect whether the terminal [supports color](https://github.com/chalk/supports-color). Used internally and handled for you, but exposed for convenience.

Can be overridden by the user with the flags --color and --no-color. For situations where using --color is not possible, use the environment variable FORCE\_COLOR=1 (level 1), FORCE\_COLOR=2 (level 2), or FORCE\_COLOR=3 (level 3) to forcefully enable color, or FORCE\_COLOR=0 to forcefully disable. The use of FORCE\_COLOR overrides all other color support checks.

Explicit 256/Truecolor mode can be enabled using the --color=256 and --color=16m flags, respectively.

### chalkStderr and supportsColorStderr

chalkStderr contains a separate instance configured with color support detected for stderr stream instead of stdout. Override rules from supportsColor apply to this too. supportsColorStderr is exposed for convenience.

### modifierNames, foregroundColorNames, backgroundColorNames, and colorNames

All supported style strings are exposed as an array of strings for convenience. colorNames is the combination of foregroundColorNames and backgroundColorNames.

This can be useful if you wrap Chalk and need to validate input:

import {modifierNames, foregroundColorNames} from 'chalk';

console.log(modifierNames.includes('bold'));

//=> true

console.log(foregroundColorNames.includes('pink'));

//=> false

## Styles

### Modifiers

* reset - Reset the current style.
* bold - Make the text bold.
* dim - Make the text have lower opacity.
* italic - Make the text italic. (Not widely supported)
* underline - Put a horizontal line below the text. (Not widely supported)
* overline - Put a horizontal line above the text. (Not widely supported)
* inverse- Invert background and foreground colors.
* hidden - Print the text but make it invisible.
* strikethrough - Puts a horizontal line through the center of the text. (Not widely supported)
* visible- Print the text only when Chalk has a color level above zero. Can be useful for things that are purely cosmetic.

### Colors

* black
* red
* green
* yellow
* blue
* magenta
* cyan
* white
* blackBright (alias: gray, grey)
* redBright
* greenBright
* yellowBright
* blueBright
* magentaBright
* cyanBright
* whiteBright

### Background colors

* bgBlack
* bgRed
* bgGreen
* bgYellow
* bgBlue
* bgMagenta
* bgCyan
* bgWhite
* bgBlackBright (alias: bgGray, bgGrey)
* bgRedBright
* bgGreenBright
* bgYellowBright
* bgBlueBright
* bgMagentaBright
* bgCyanBright
* bgWhiteBright

## 256 and Truecolor color support

Chalk supports 256 colors and [Truecolor](https://github.com/termstandard/colors) (16 million colors) on supported terminal apps.

Colors are downsampled from 16 million RGB values to an ANSI color format that is supported by the terminal emulator (or by specifying {level: n} as a Chalk option). For example, Chalk configured to run at level 1 (basic color support) will downsample an RGB value of #FF0000 (red) to 31 (ANSI escape for red).

Examples:

* chalk.hex('#DEADED').underline('Hello, world!')
* chalk.rgb(15, 100, 204).inverse('Hello!')

Background versions of these models are prefixed with bg and the first level of the module capitalized (e.g. hex for foreground colors and bgHex for background colors).

* chalk.bgHex('#DEADED').underline('Hello, world!')
* chalk.bgRgb(15, 100, 204).inverse('Hello!')

The following color models can be used:

* [rgb](https://en.wikipedia.org/wiki/RGB_color_model) - Example: chalk.rgb(255, 136, 0).bold('Orange!')
* [hex](https://en.wikipedia.org/wiki/Web_colors#Hex_triplet) - Example: chalk.hex('#FF8800').bold('Orange!')
* [ansi256](https://en.wikipedia.org/wiki/ANSI_escape_code#8-bit) - Example: chalk.bgAnsi256(194)('Honeydew, more or less')

## Browser support

Since Chrome 69, ANSI escape codes are natively supported in the developer console.

## Windows

If you're on Windows, do yourself a favor and use [Windows Terminal](https://github.com/microsoft/terminal) instead of cmd.exe.

## Origin story

[colors.js](https://github.com/Marak/colors.js) used to be the most popular string styling module, but it has serious deficiencies like extending String.prototype which causes all kinds of [problems](https://github.com/yeoman/yo/issues/68) and the package is unmaintained. Although there are other packages, they either do too much or not enough. Chalk is a clean and focused alternative.

## Related

* [chalk-template](https://github.com/chalk/chalk-template) - [Tagged template literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#tagged_templates) support for this module
* [chalk-cli](https://github.com/chalk/chalk-cli) - CLI for this module
* [ansi-styles](https://github.com/chalk/ansi-styles) - ANSI escape codes for styling strings in the terminal
* [supports-color](https://github.com/chalk/supports-color) - Detect whether a terminal supports color
* [strip-ansi](https://github.com/chalk/strip-ansi) - Strip ANSI escape codes
* [strip-ansi-stream](https://github.com/chalk/strip-ansi-stream) - Strip ANSI escape codes from a stream
* [has-ansi](https://github.com/chalk/has-ansi) - Check if a string has ANSI escape codes
* [ansi-regex](https://github.com/chalk/ansi-regex) - Regular expression for matching ANSI escape codes
* [wrap-ansi](https://github.com/chalk/wrap-ansi) - Wordwrap a string with ANSI escape codes
* [slice-ansi](https://github.com/chalk/slice-ansi) - Slice a string with ANSI escape codes
* [color-convert](https://github.com/qix-/color-convert) - Converts colors between different models
* [chalk-animation](https://github.com/bokub/chalk-animation) - Animate strings in the terminal
* [gradient-string](https://github.com/bokub/gradient-string) - Apply color gradients to strings
* [chalk-pipe](https://github.com/LitoMore/chalk-pipe) - Create chalk style schemes with simpler style strings
* [terminal-link](https://github.com/sindresorhus/terminal-link) - Create clickable links in the terminal

## Maintainers

* [Sindre Sorhus](https://github.com/sindresorhus)
* [Josh Junon](https://github.com/qix-)

## Arrays

To specify the type of an array like [1, 2, 3], you can use the syntax number[]; this syntax works for any type (e.g. string[] is an array of strings, and so on). You may also see this written as Array<number>, which means the same thing. We’ll learn more about the syntax T<U> when we cover generics.

# 20 Array methods in Typescript you need to know with examples

## Pre-built functions have always been proven time-savers. Find the list of utility functions in typescript with demonstrations.

We, programmers, play with Arrays all the time, and yet most of the time we don’t know all helper functions provided by the TypeScript.

I have listed out 20 methods that you need to know, with **fun emojis**.

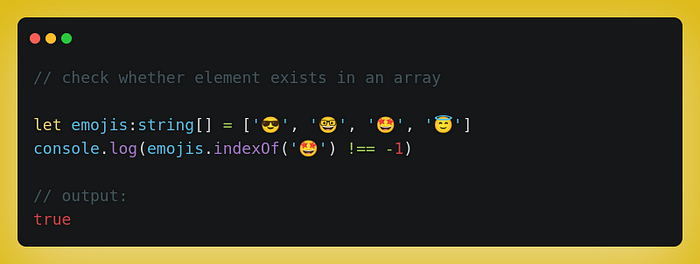
# 1. indexOf()

Every array element has an index. This method returns the index of an element in an array.

a. If specific element doesn’t exist within array, indexOf() returns -1.

Hence, this method can be used to check whether an element exists in an array or not.

**syntax:**  
array.indexOf(element)

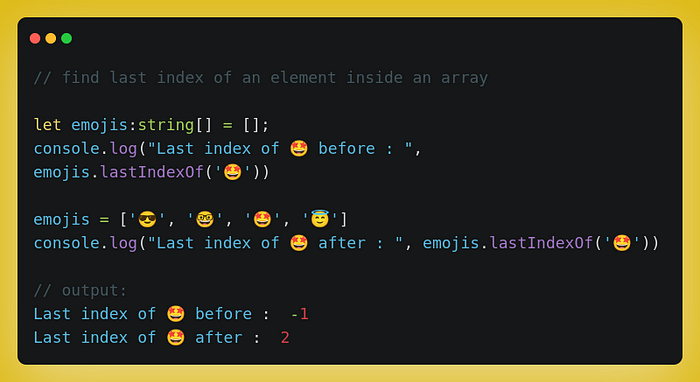


# 2. lastIndexOf()

This method returns an array’s last element’s index.

a. If an array is empty then, it returns -1 as of the indexOf() function.  
b. If an array has one more same element, then it returns the maximum index of duplicate items.

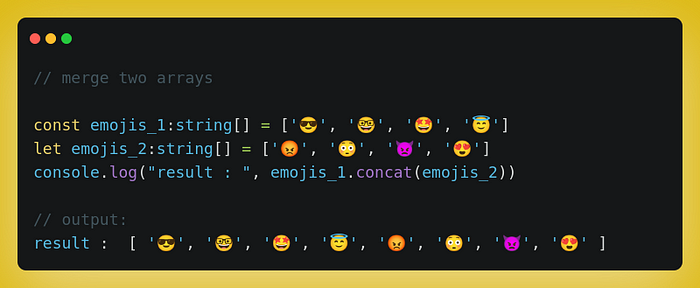
**syntax:**  
array.lastIndexOf(element)



# 3. concat()

As the name suggests, this method simply merges two arrays and returns a combined result.

**syntax:**  
array1.concat(array2)

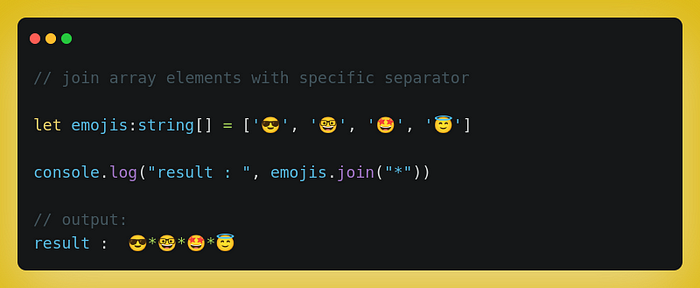


# 4. join()

According to the name, this method joins all elements of the array into a string with a given operator.

a. If an operator is not given, it joins elements with a comma(,).

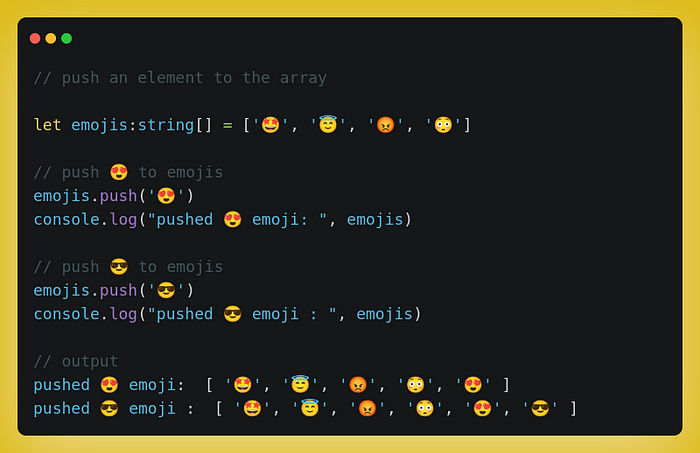
**syntax:**array.join(operator)



# 5. push()

This method pushes/adds one or more elements to the array at the last of an array.

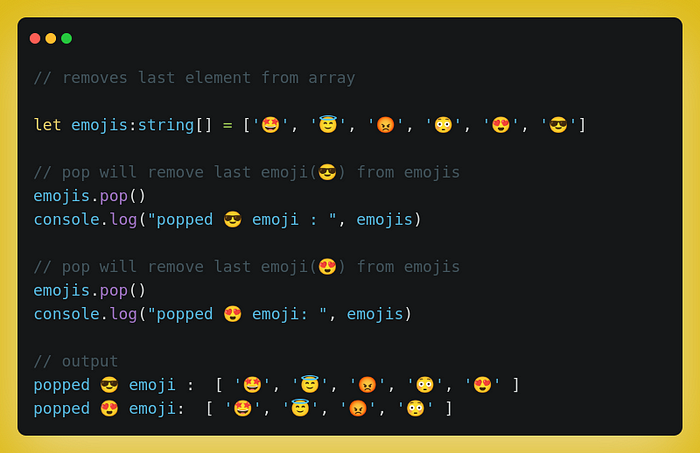
**syntax:**array.push(element)



# 6. pop()

This method pops/removes the last element from an array.

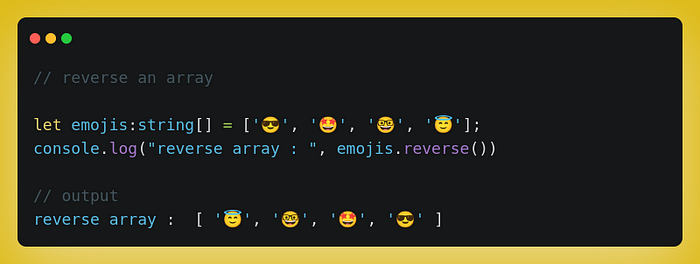
**syntax:**array.pop()



# 7. reverse()

As per the name, this method reverts the order of an array.

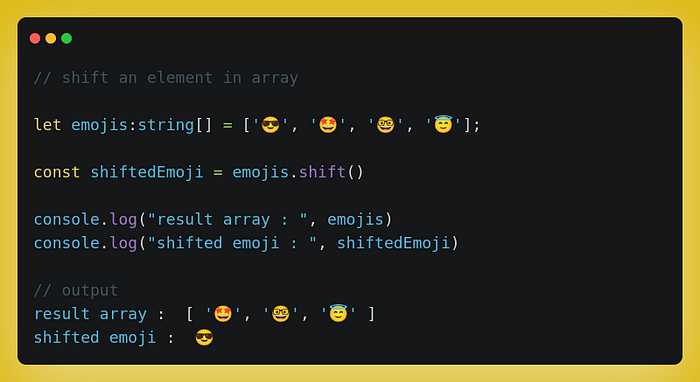
**syntax:**array.reverse()



# 8. shift()

This method removes starting(first) element from an array and **returns the removed element.**  
We can say that it’s the exact opposite of **pop()** method, which removes the last element and returns the result.

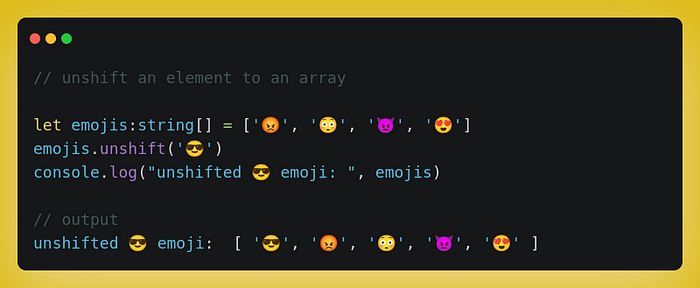
**syntax:**array.shift()



# 9. unshift()

It has the exact opposite behavior to the **shift()** method. It adds an element at starting of an array and returns a new array.

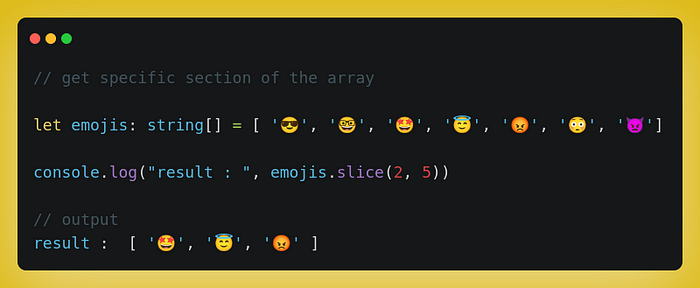
**syntax:**array.unshift(element)



# 10. slice()

This method cuts an array, in whichever manner we want and returns the trimmed array.  
a. It excludes the last index from an argument.

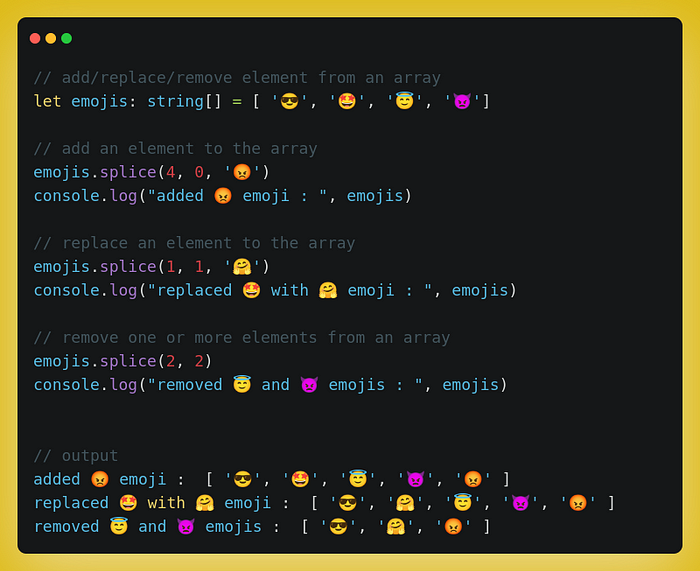
**syntax:**array.slice(start\_index, end\_index)



# 11. splice()

This method can be used for multiple purposes. For,  
1. Add an element to an array  
2. Replace specific elements within an array  
3. Remove specific elements from an array

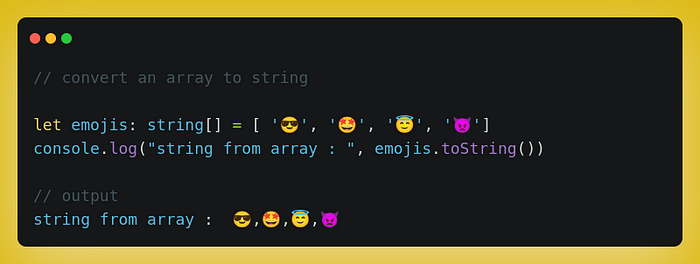
**syntax:**array.splice(index, number of elements to be removed, element1,..,elementN)



# 12. toString()

This method converts an array to a comma-separated string.

**syntax:**array.toString()

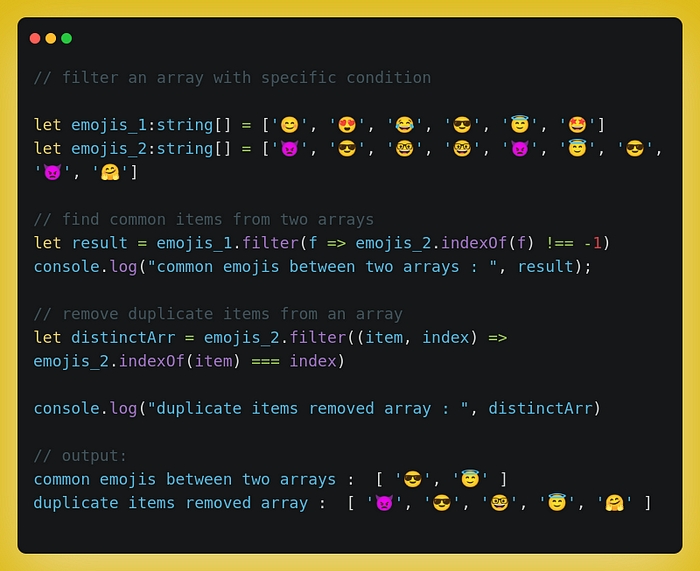


# 13. filter()

This method can also be used in multiple use cases. Like, such as finding even numbers from an array, finding common items from two arrays, or getting a distinct array.

Basically, it checks the conditions which are provided and returns a filtered array.

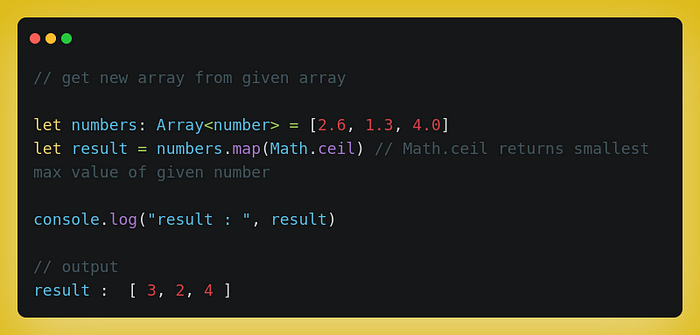
**syntax:**array.filter(callback)



# 14. map()

This method creates a new array with the results of calling a provided function on every element in this array.  
In the example, we’ve invoked map() with **Math.ceil**which returns the lowest maximum number.

**syntax:**array.map(callback)

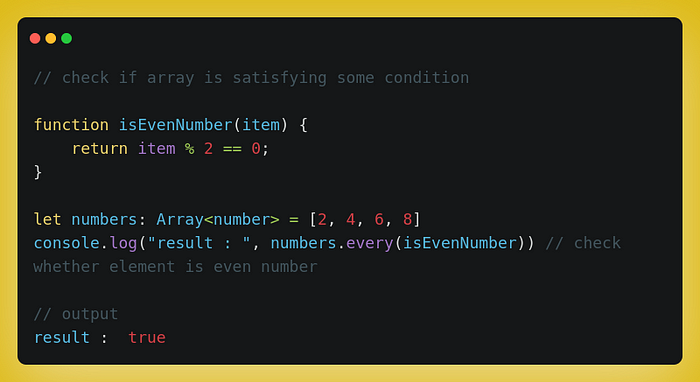


# 15. every()

This method tests whether all the elements in an array pass the test implemented by the provided function.

In the example, we have checked for even numbers.

**syntax:**array.every(callback)

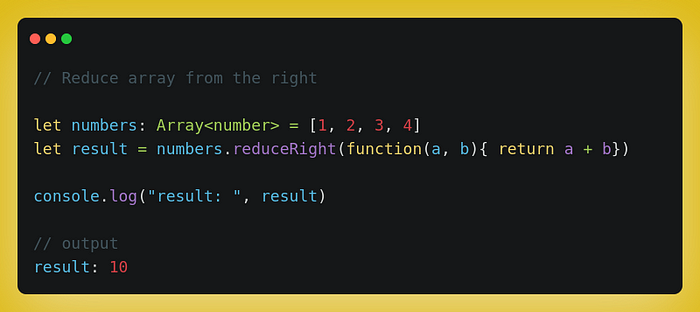


# 16. reduceRight()

This method applies a function simultaneously against two values of the array (from right to left) to reduce it to a single value.

In the example, the array is reduced with the addition of an element to the previous one(right to left).

**syntax:**array.reduceRight(callback)

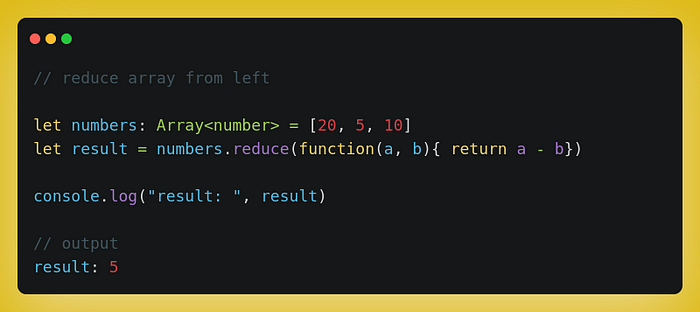


# 17. reduce()

This method behaves the exact opposite of the reduceRight() method.  
It applies a function simultaneously against two values of the array (from left to right) to reduce it to a single value.

In the example, an array is reduced with the subtraction of an element from the previous one(left to right).

**syntax:**array.reduce(callback)

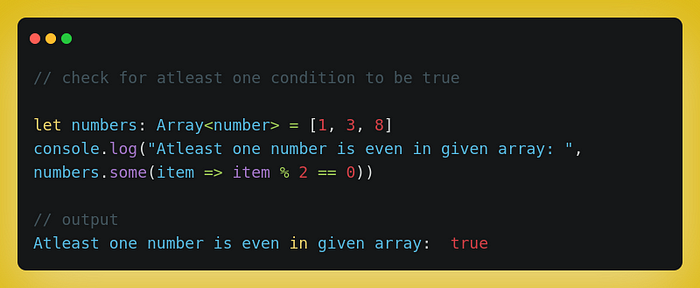


# 18. some()

This method is generally used for testing purposes.  
i.e. To know whether at least a single item from an array is fulfilling a given condition or not.

In the example, again we’ve checked for at least a single even number present in an array.

**syntax:**array.some(callback)

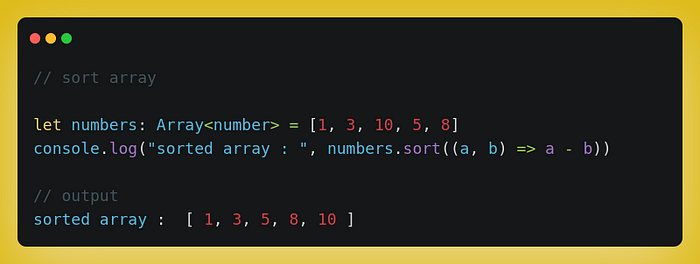


# 19. sort()

As the name suggests, this method arranges array elements in sorting orders.

In the example, we’ve sorted an array in ascending order. It will sort in descending order, with the condition b-a instead of a-b . similarly, we do in js.

**syntax:**array.sort(callback)

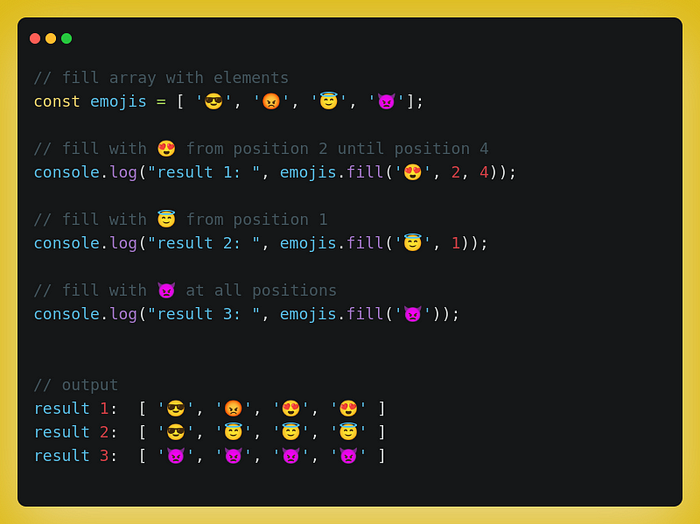


# 20. fill()

This method changes all elements in an array to a static value, from a start index (default 0) to an end index (default array.length) and returns the modified array.

a. It can add new elements to specific(multiple) positions

**syntax:**  
array.fill(value, start\_index, end\_index)



Hope you learned something today. Full code available at [typescript array methods](https://gist.github.com/nidhi-canopas/30b9a35533d0d2161e56f5d7ae642616).

Functions are the basic building block of any application, whether they’re local functions, imported from another module, or methods on a class. They’re also values, and just like other values, TypeScript has many ways to describe how functions can be called. Let’s learn about how to write types that describe functions.

## Function Type Expressions

The simplest way to describe a function is with a function type expression. These types are syntactically similar to arrow functions:

function greeter(fn: (a: string) => void) {

fn("Hello, World");

}

function printToConsole(s: string) {

console.log(s);

}

greeter(printToConsole);

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAcwE4FN1XagFMMALkVwENiBnKVGMZASkQF4A+RANzhgBNGBvALAAoRIgK4ARAAl0AG1lwANIgDqcVLO4T6AbmEBfYcNCRYCRAAcaYKABU4AYQQU4s9LgqVqtBokEjECGdXdAA6BWQPXQMjITRMbDwrWjtHYLddIA)

The syntax (a: string) => void means “a function with one parameter, named a, of type string, that doesn’t have a return value”. Just like with function declarations, if a parameter type isn’t specified, it’s implicitly any.

Note that the parameter name is **required**. The function type (string) => void means “a function with a parameter named string of type any“!

Of course, we can use a type alias to name a function type:

type GreetFunction = (a: string) => void;

function greeter(fn: GreetFunction) {

// ...

}

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBA4gThCwBiBXAdgY2ASwPbpQC8UAFAIYBcUAzsHDugOYCUxAfFAG544AmAbgCwAKABmGbPkJMESCHFJj01eIhSTcBNgG9RUKAHpDUAHTnRAXyA)

## Call Signatures

In JavaScript, functions can have properties in addition to being callable. However, the function type expression syntax doesn’t allow for declaring properties. If we want to describe something callable with properties, we can write a call signature in an object type:

type DescribableFunction = {

description: string;

(someArg: number): boolean;

};

function doSomething(fn: DescribableFunction) {

console.log(fn.description + " returned " + fn(6));

}

function myFunc(someArg: number) {

return someArg > 3;

}

myFunc.description = "default description";

doSomething(myFunc);

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAIhDOBjATgSwEYEN0BsIDEBXAO0WFQHtioBeKAbwFgAoKKAEwRVTHKoC4o8YGmIBzANws2ACngUAthACCyMYOKEF6CMgCUg9BQp5MxKcwC+FgGYkylauwoBlRRGAALVOJk3ignBIaFi4BPZ8xHoM0lCIVPJ4AHQ4FGJ+xEmcwTyRUADUUABEUMgehMjEEOzFBVD+MgBsenoWliwsdqR5CiBEpHLuqupQmtq60UyspeWVQkNqUAB8UADMbSy9-YhZXGi8jrTFnDaYhDjAHHu5jkUWLM5uSl4+6Vv2rUA)

Note that the syntax is slightly different compared to a function type expression - use : between the parameter list and the return type rather than =>.

## Construct Signatures

JavaScript functions can also be invoked with the new operator. TypeScript refers to these as constructors because they usually create a new object. You can write a construct signature by adding the new keyword in front of a call signature:

type SomeConstructor = {

new (s: string): SomeObject;

};

function fn(ctor: SomeConstructor) {

return new ctor("hello");

}

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAyg9gWwgeQEYCsIGNhQLxQCGAdiANwCwAUAPQ1QC0TWArsEw9aJLIhAMJxiAZ2AAnFjjhj8UAN7UoUYhADuUABTCAXFFFiAlsQDmASl3wkaTDkpUAvnYBmLYjgNCoT4hqliLfIIi4pLA0qbyilBiEMAsYsTKalB+GgBEABYQADbZcGmmdvZAA)

Some objects, like JavaScript’s Date object, can be called with or without new. You can combine call and construct signatures in the same type arbitrarily:

interface CallOrConstruct {

(n?: number): string;

new (s: string): Date;

}

[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgMJwDYYPJVQexAGcwoBXBMZAbwFgAoZZAChAH4AuZEMgWwCNoASi4kooAOYBuBkxAQA7iyKjSkkcgAicSDPoBfIA)

## Generic Functions

It’s common to write a function where the types of the input relate to the type of the output, or where the types of two inputs are related in some way. Let’s consider for a moment a function that returns the first element of an array:

function firstElement(arr: any[]) {

return arr[0];

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMGAnAzlAogGwKYC2eYUAFAIaqoBci5YAngNoC6AlIgN4CwAUIolR4oIVEkqomABhYBuPgF8gA)

This function does its job, but unfortunately has the return type any. It’d be better if the function returned the type of the array element.

In TypeScript, generics are used when we want to describe a correspondence between two values. We do this by declaring a type parameter in the function signature:

function firstElement<Type>(arr: Type[]): Type | undefined {

return arr[0];

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMGAnAzlAogGwKYC2eYUAPACoCeADngHwAUAhqqgFyJW0DaAugJQcueRAB9E4ACZ4UYPJMQBvALAAoRIlR4oIVEhapuABl4BuNQF8gA)

By adding a type parameter Type to this function and using it in two places, we’ve created a link between the input of the function (the array) and the output (the return value). Now when we call it, a more specific type comes out:

// s is of type 'string'

const s = firstElement(["a", "b", "c"]);

// n is of type 'number'

const n = firstElement([1, 2, 3]);

// u is of type undefined

const u = firstElement([]);

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtVKxgGcMBRCEAWxFQwB4AVATwAcQA+AClhgC55mbANoBdAJT9BCAD7w0oRFlQhgAbgCwAKAD02+AFpDYZBkP6tu+EXhZrORPAysEAchIwlAcxdaweElbwALwExGQU1LScQgBEUDEANPAxAEaJyWAx4ho6evi28PaOzvAuqMiUKSAwPpp+qAH4IYph5FQ0GNEAjEkATEkAzNkWesg2dg5ObHKoCkoqvv4YcsGhJG2RnaJiqkA)

### **Inference**

Note that we didn’t have to specify Type in this sample. The type was inferred - chosen automatically - by TypeScript.

We can use multiple type parameters as well. For example, a standalone version of map would look like this:

function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {

return arr.map(func);

}

// Parameter 'n' is of type 'string'

// 'parsed' is of type 'number[]'

const parsed = map(["1", "2", "3"], (n) => parseInt(n));

[Try](https://www.typescriptlang.org/play/#code/PTAEAcCcFMBdYJbUgWgQcwHYHsYFgAoAMwFdMBjRbTUAWwENwAeASU3BNgBpQB5TjrAB8ACnqRIALlBtBAbQC6PUhWljI6abM4BKUAF4hfAbun9Y8haADehUKBiwSkGuMgA6BuBEryOgNyEAL6EhCCgAAri9LRwyKAA5JgJoAgAzqDYRKCwAJ7g0IlpsJAImOgJYWAJ4OJp0AAmKemZ2XkFiZgktABGyIqVBOTUxRB1jQZ0jCJyAEQAjLM8swBMS6CzAMyzSqAimHqGY5D1bLD7OgFAA)

Note that in this example, TypeScript could infer both the type of the Input type parameter (from the given string array), as well as the Output type parameter based on the return value of the function expression (number).

### **Constraints**

We’ve written some generic functions that can work on any kind of value. Sometimes we want to relate two values, but can only operate on a certain subset of values. In this case, we can use a constraint to limit the kinds of types that a type parameter can accept.

Let’s write a function that returns the longer of two values. To do this, we need a length property that’s a number. We constrain the type parameter to that type by writing an extends clause:

function longest<Type extends { length: number }>(a: Type, b: Type) {

if (a.length >= b.length) {

return a;

} else {

return b;

}

}

// longerArray is of type 'number[]'

const longerArray = longest([1, 2], [1, 2, 3]);

// longerString is of type 'alice' | 'bob'

const longerString = longest("alice", "bob");

// Error! Numbers don't have a 'length' property

const notOK = longest(10, 100);

Argument of type 'number' is not assignable to parameter of type '{ length: number; }'.Argument of type 'number' is not assignable to parameter of type '{ length: number; }'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWdG00BYAKADMBXAOwGMAXASzitABtmBzSROgHgBUAngAdIoSAA86kKgBNEoAN5sZHOgAtUVCgFsARjFABfAHwAKAIaohogDSg91kZACUS0qFAMyoSwDpWVQ1QEwBeBwCg9TdFD09QaEg6CmgWCwBuOKNxVkQxWJJ4hKSUlj1MwuNSI1JSEDZOGABBWAtBLwU4HzpnUABybX0YAG0AXT7SGmYeBqouaBboNtBw9jnuOjNhgEZ7NFH7Hb37DFGXCvq1+YBlOmgGOY7QLtAe0X6LVgYaSD7QAB9+no4HoJiQplQZlcYLd7o9Vo0eGYAESfb6QZH2ZHAvTI851MAAUVgCAAhKAAHK6AxIUCyZh9OigdQWABuYgs-UCcw0f2E8FE0B6k2mTKocDoAHkANIrWZcJHbAAM9mVSvOQA)

There are a few interesting things to note in this example. We allowed TypeScript to infer the return type of longest. Return type inference also works on generic functions.

Because we constrained Type to { length: number }, we were allowed to access the .length property of the a and b parameters. Without the type constraint, we wouldn’t be able to access those properties because the values might have been some other type without a length property.

The types of longerArray and longerString were inferred based on the arguments. Remember, generics are all about relating two or more values with the same type!

Finally, just as we’d like, the call to longest(10, 100) is rejected because the number type doesn’t have a .length property.

### **Working with Constrained Values**

Here’s a common error when working with generic constraints:

function minimumLength<Type extends { length: number }>(

obj: Type,

minimum: number

): Type {

if (obj.length >= minimum) {

return obj;

} else {

return { length: minimum };

Type '{ length: number; }' is not assignable to type 'Type'.

'{ length: number; }' is assignable to the constraint of type 'Type', but 'Type' could be instantiated with a different subtype of constraint '{ length: number; }'.Type '{ length: number; }' is not assignable to type 'Type'.

'{ length: number; }' is assignable to the constraint of type 'Type', but 'Type' could be instantiated with a different subtype of constraint '{ length: number; }'.

}

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oLACgAzAVwDsBjAFwEs4TQBbKkq+o+gGUhIHMKALADwAVAJ4AHSKEgAPClwAmiUAG9QAGy68+qEmwBGMUAF8AfAAp8oUHD0ArVKIkAaSwyYs2O-THwBKB+KSyq5UBKBmNrYAdBo8-KAmALxuzKz0viquVtCQFETQdJEA3K5GUmqIQVmgOXkFKuqa-KiMqWzGJXhWRvhGQA)

It might look like this function is OK - Type is constrained to { length: number }, and the function either returns Type or a value matching that constraint. The problem is that the function promises to return the same kind of object as was passed in, not just some object matching the constraint. If this code were legal, you could write code that definitely wouldn’t work:

// 'arr' gets value { length: 6 }

const arr = minimumLength([1, 2, 3], 6);

// and crashes here because arrays have

// a 'slice' method, but not the returned object!

console.log(arr.slice(0));

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwFstUsDkCAZEVAcwwAsAeAFQE8AHBEADw2uADO8AN7wI1OvQBc8VOQBGIGPAC+APgAUAWABQ8eDnkArGW04AaXfqIkyBGXIKKYugJSmOIANy6A9L-gAWmCwZAxgwL8AgHJYGGj4GhAMIQA3KAhkBFFxWgYZADZVXTA8AQx4OPgAXkJiUnIqPPoNAG0ARnN4ACYugGYAXS6C1x8df0rUYHgwGCgBehAhRbh4RTAoZAEEOKhWZahUkCjK+GiBCCwwEASCZPocYC75MNkcCoYEOAxkGFQQaaGIzgDAAQhKZRw4gAdBAcDQNHFoRcriANAAGVyjIA)

### **Specifying Type Arguments**

TypeScript can usually infer the intended type arguments in a generic call, but not always. For example, let’s say you wrote a function to combine two arrays:

function combine<Type>(arr1: Type[], arr2: Type[]): Type[] {

return arr1.concat(arr2);

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABBOBbARjMBTAPAFQE8AHbAPgAoBDAJxoEYAuRI0gbQF0AaRWmgJmatsnAJRCSIjogDeAWABQiRDWxQQNJH3oA6FJCpRqdfqIDcigL5A)

Normally it would be an error to call this function with mismatched arrays:

const arr = combine([1, 2, 3], ["hello"]);

Type 'string' is not assignable to type 'number'.Type 'string' is not assignable to type 'number'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oLACgATSAYwBsBDaSUAMwFcA7YgFwEs4HRi4BbAI1YNIAHgAqATwAOkAHwAKStACMqCdIDaAXQA0oRWlVTIWgJSGNmgNz4QoALQPidZg7v5uDRMz2xQAXi5eASE5dSVdNF0MHVB1ACIAC0hSUjg4zRNLIA)

If you intended to do this, however, you could manually specify Type:

const arr = combine<string | number>([1, 2, 3], ["hello"]);

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXzBwFsAjLVEAHgBUBPABxAD4AKWGARgC547GBtALoAaeOwBMPPiCEBKKQxmCA3AFgAUAHpN8ALT6wyDPt0bCqAM4YxMGPAC8BYmQqUrMcgHN4AH3ipkUhAYVn4OUXFRAGYReH4AIgALEAgIHHjBWWUgA)

### **Guidelines for Writing Good Generic Functions**

Writing generic functions is fun, and it can be easy to get carried away with type parameters. Having too many type parameters or using constraints where they aren’t needed can make inference less successful, frustrating callers of your function.

#### Push Type Parameters Down

Here are two ways of writing a function that appear similar:

function firstElement1<Type>(arr: Type[]) {

return arr[0];

}

function firstElement2<Type extends any[]>(arr: Type) {

return arr[0];

}

// a: number (good)

const a = firstElement1([1, 2, 3]);

// b: any (bad)

const b = firstElement2([1, 2, 3]);

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMGAnAzlAogGwKYC2eYUAjADwAqAngA54B8AFAIaqoBciN9A2gLoBKRAG8AsAChEiVHighUSNql4AGfgG5JAX0mTQkWAmRpMuQsSgAmKnTyI8ADyjEAJukQsw1Ac2VcePGFxKRk5BSV2NU0dPQkAenjPLjAQAgAjPFREJgBzODhXQUkIBExPRABeEwxsfCISUiZeUgAaRCt2gGYhLQSk9K4vahz0liKSsqhEdKqas3rLK2a2ju7eoA)

These might seem identical at first glance, but firstElement1 is a much better way to write this function. Its inferred return type is Type, but firstElement2’s inferred return type is any because TypeScript has to resolve the arr[0] expression using the constraint type, rather than “waiting” to resolve the element during a call.

**Rule**: When possible, use the type parameter itself rather than constraining it

#### Use Fewer Type Parameters

Here’s another pair of similar functions:

function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {

return arr.filter(func);

}

function filter2<Type, Func extends (arg: Type) => boolean>(

arr: Type[],

func: Func

): Type[] {

return arr.filter(func);

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMGAbKBTATgRgDwAqAngA4YB8AFAIZZYBcixZA2gLoA0y4EjNWAc0bMMASkQBecogBGcOKgzUwo4aQztEAbwCwAKESIsGKCCxJaWAHQp02SqEiiA3PoC++-Y+jwktzFgATITqXABiPIgYAB6YYAAmAM6I-EJM6uJSsvKKylT6hpZqrJwF3JCMEZD6qukl2mXGpuaIljZoAQ48Lu5AA)

We’ve created a type parameter Func that doesn’t relate two values. That’s always a red flag, because it means callers wanting to specify type arguments have to manually specify an extra type argument for no reason. Func doesn’t do anything but make the function harder to read and reason about!

**Rule**: Always use as few type parameters as possible

#### Type Parameters Should Appear Twice

Sometimes we forget that a function might not need to be generic:

function greet<Str extends string>(s: Str) {

console.log("Hello, " + s);

}

greet("world");

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAcwE4FN1QDwGUqqLoAeU6YAJgM6JUExjIB8AFFQFyL6oCUiA3gFgAUIkQQEVOABt0AOmlxkLAEQAJdNMUAaRCsQBqWjwDcIgL4iRaTFFUB3OKmkUVpoA)

We could just as easily have written a simpler version:

function greet(s: string) {

console.log("Hello, " + s);

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAcwE4FN1QBQGcBciuUqMYyAlIgN4CwAUIohArnADboB07cy2AIgAS6drwA0iAYgDURCgG4GAXyA)

Remember, type parameters are for relating the types of multiple values. If a type parameter is only used once in the function signature, it’s not relating anything. This includes the inferred return type; for example, if Str was part of the inferred return type of greet, it would be relating the argument and return types, so would be used twice despite appearing only once in the written code.

**Rule**: If a type parameter only appears in one location, strongly reconsider if you actually need it

## Optional Parameters

Functions in JavaScript often take a variable number of arguments. For example, the toFixed method of number takes an optional digit count:

function f(n: number) {

console.log(n.toFixed()); // 0 arguments

console.log(n.toFixed(3)); // 1 argument

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMAFGAXIsIC2AjAUwCcBKRAbwFgAoRRCBAZzgBsCA6FuAczXajgAxGAA8CAExQkSAbkQB6eYgAMiAIZFuuAmCiMadBmGZtOPPgOFjJAZmlzFiAIzrN23TQC+QA)

We can model this in TypeScript by marking the parameter as optional with ?:

function f(x?: number) {

// ...

}

f(); // OK

f(10); // OK

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMAFADwPwC5FhAWwCMBTAJwEpEBvAWAChFEB6JxAOg-oF97VyBuZqwDyAaV4oAjAAYBQxGKA)

Although the parameter is specified as type number, the x parameter will actually have the type number | undefined because unspecified parameters in JavaScript get the value undefined.

You can also provide a parameter default:

function f(x = 10) {

// ...

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMAFAD0QXkQRgAwCUiA3gLABQiiA9NYgHSMUC+QA)

Now in the body of f, x will have type number because any undefined argument will be replaced with 10. Note that when a parameter is optional, callers can always pass undefined, as this simply simulates a “missing” argument:

// All OK

f();

f(10);

f(undefined);

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtVIAoAPAfgC55VkBbAIxBgEoKA3HLYAbgFgAoAen7wAtKLDIMo4X0HwAghAjwA8gGk+iAox69NARgAM2jQTShEWVCGDagA)

### **Optional Parameters in Callbacks**

Once you’ve learned about optional parameters and function type expressions, it’s very easy to make the following mistakes when writing functions that invoke callbacks:

function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {

for (let i = 0; i < arr.length; i++) {

callback(arr[i], i);

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAWwJ4DE4CcCiBDCACwAo8ssAuRPMVAbQF0AaRCPAG3YCMCBrK0lgDmVGqhYwwAEwCmADwD8VMCGRcZWAJSIAvAD5EANzgwp2gN4BYAFCJEwbImLsZURDF2IADAG53iAB5qcgA6FzAhKEI-GABqWIsbOzs2Th4IXkEsOhhmd00fJMQAXxtioA)

What people usually intend when writing index? as an optional parameter is that they want both of these calls to be legal:

myForEach([1, 2, 3], (a) => console.log(a));

myForEach([1, 2, 3], (a, i) => console.log(a, i));

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYDMbQEYAOABgBYCBYAKABNIBjAGwENpJQAzAVwDs6AXAJZxuoALYBPAGIIAokzoALABRVQoFtFRNu4gNoBdADSrQdJgwYAjeQGtUSlgHMtOw6AHdaADwD8qbpyiljAAlKAAvAB8oABucALUVCGocQkA3FQgoAC0uXScfLnZVBLS0HKKSrp4bmhuWEagDmFRpsKIcAyQAHQMcI7NIRmUpbLyytW19Y0ObgIt0XTtnT19A0xzIUNAA)

What this actually means is that *callback* might get invoked with one argument. In other words, the function definition says that the implementation might look like this:

function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {

for (let i = 0; i < arr.length; i++) {

// I don't feel like providing the index today

callback(arr[i]);

}

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYDMbQEYAOABgBYCBYAKADMBXAOwGMAXASzntAFsBPAMQQBRAIaMAFgAphsVMPo8A2gF0ANKEbCANpoBGogNaop0AOaz5a1vQAmkAB4B+VPVpcdMAJSgAvAD5QAG5wrNZeAN5UoKDUCKASmpDMoKw+oEQA3MmgADyg0tAAdAn0JsximawA1JXhkVGgIKAAkqDWHADkSdSQkJqgmqz6kKAADvABIVYmoGXDVrZ2M3DWwjx1URraeoz6xtAKrEoe6XUAvlSnQA)

In turn, TypeScript will enforce this meaning and issue errors that aren’t really possible:

myForEach([1, 2, 3], (a, i) => {

console.log(i.toFixed());

'i' is possibly 'undefined'.'i' is possibly 'undefined'.

});

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYDMbQEYAOABgBYCBYAKABNIBjAGwENpJQAzAVwDs6AXAJZxuoALYBPAGIIAokzoALABRVQoFtFRNu4gNoBdADSrQdJgwYAjeQGtUSlgHMtOw6AHdaADwD8qbpyiljAAlKAAvAB8oABucALUVCGocQkA3FQgoAC0uXScfLnZVBLS0HKKSrp4bmhuWEagDm4CYVGgAN4mdMKIcAyQAHQMcI5KAoN8cJICXpDUSiEhGZQAvstAA)

In JavaScript, if you call a function with more arguments than there are parameters, the extra arguments are simply ignored. TypeScript behaves the same way. Functions with fewer parameters (of the same types) can always take the place of functions with more parameters.

**Rule**: When writing a function type for a callback, never write an optional parameter unless you intend to call the function without passing that argument

## Function Overloads

Some JavaScript functions can be called in a variety of argument counts and types. For example, you might write a function to produce a Date that takes either a timestamp (one argument) or a month/day/year specification (three arguments).

In TypeScript, we can specify a function that can be called in different ways by writing overload signatures. To do this, write some number of function signatures (usually two or more), followed by the body of the function:

function makeDate(timestamp: number): Date;

function makeDate(m: number, d: number, y: number): Date;

function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {

if (d !== undefined && y !== undefined) {

return new Date(y, mOrTimestamp, d);

} else {

return new Date(mOrTimestamp);

}

}

const d1 = makeDate(12345678);

const d2 = makeDate(5, 5, 5);

const d3 = makeDate(1, 3);

No overload expects 2 arguments, but overloads do exist that expect either 1 or 3 arguments.No overload expects 2 arguments, but overloads do exist that expect either 1 or 3 arguments.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYDsGCwAoAMwFcA7AYwBcBLOU0AWwEMBrSAEScsgAoaHIiSkwYAHVKWIMARjACUqTtwDcBEhRp1GrDl14MJU2dAA0oACaGZMMwE8rxhaCWRVRMlVr1mbFzwYA8tAAKtQCQiLioJLWphYA-A42oLaJ0UbyinqgAN4EoKDUhKA85qAAhAC8laBk5pCE1KSQZQBkrSkV1bWk9Y3N5nK5+QWg0JCUxND0zQDuzno8tmaBIWGCwmJmg24FAL6gkAA2iJDD+KNjE1MzkPN+q6Hhm6Jyu6B7BJ-45HRCFgBGUA1Hy6bg8AFoADMABYMAA2LAADjeBF+pH+5jQwO0vkWGDMBNAGFRPz+lAsUJxoL8ALMULeQA)

In this example, we wrote two overloads: one accepting one argument, and another accepting three arguments. These first two signatures are called the overload signatures.

Then, we wrote a function implementation with a compatible signature. Functions have an implementation signature, but this signature can’t be called directly. Even though we wrote a function with two optional parameters after the required one, it can’t be called with two parameters!

### **Overload Signatures and the Implementation Signature**

This is a common source of confusion. Often people will write code like this and not understand why there is an error:

function fn(x: string): void;

function fn() {

// ...

}

// Expected to be able to call with zero arguments

fn();

Expected 1 arguments, but got 0.Expected 1 arguments, but got 0.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYMBYCwAoAMwFcA7AYwBcBLOU0Q0gCgA9VFLprSBzASlQA3ONQAmAbgIkKNOg2Z9QAbwKhQIUADptBAL4ENAURYAHSFUijQlOKABGkUAEM7AG0c3Q5J69egA7tSUABagAF4wtk7QPMQAtpCklIhSCuJAA)

Again, the signature used to write the function body can’t be “seen” from the outside.

The signature of the implementation is not visible from the outside. When writing an overloaded function, you should always have two or more signatures above the implementation of the function.

The implementation signature must also be compatible with the overload signatures. For example, these functions have errors because the implementation signature doesn’t match the overloads in a correct way:

function fn(x: boolean): void;

// Argument type isn't right

function fn(x: string): void;

This overload signature is not compatible with its implementation signature.This overload signature is not compatible with its implementation signature.

function fn(x: boolean) {}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYCcAWAsAFABmArgHYDGALgJZxmhFkAUAHqgEZxwA2kAhmQCUqAG5waAEwDchEKACC0AOYkAtpDJVQVAJ4AHSKBqIyAcm3QaygBZVCpSrXqMW7UIipWyykaHFSssTk1HQMTGyc3HyCQqAA3gC+QA)

function fn(x: string): string;

// Return type isn't right

function fn(x: number): boolean;

This overload signature is not compatible with its implementation signature.This overload signature is not compatible with its implementation signature.

function fn(x: string | number) {

return "oops";

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYCcAWAsAFABmArgHYDGALgJZxmhFkAUAHqoldDWQOYCUHLj14BuQiFAAlSFRLQGVAJ4AHSKBqIyAciqhuvABZVCpSrXqMW7UGRIBbAEYxBoR3DgAbSAEMy44nJqOgYmNiFuPlAAH1sHZ2h+UABvQlB9WXkGACIPFURsgIBfIA)

### **Writing Good Overloads**

Like generics, there are a few guidelines you should follow when using function overloads. Following these principles will make your function easier to call, easier to understand, and easier to implement.

Let’s consider a function that returns the length of a string or an array:

function len(s: string): number;

function len(arr: any[]): number;

function len(x: any) {

return x.length;

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAGwKZgBQGcBcitQBOMYA5gJR5ggC2ARqoQNwCwAUKJLAiuhgIaFCefmACeAbQC6lRNXqNWHcNHhI0mAB4jx5RAG92iRIVRQQhJJoB0G0lAAWSgL5A)

This function is fine; we can invoke it with strings or arrays. However, we can’t invoke it with a value that might be a string or an array, because TypeScript can only resolve a function call to a single overload:

len(""); // OK

len([0]); // OK

len(Math.random() > 0.5 ? "hello" : [0]);

No overload matches this call.

Overload 1 of 2, '(s: string): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'string'.

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

Overload 2 of 2, '(arr: any[]): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'any[]'.

Type 'string' is not assignable to type 'any[]'.No overload matches this call.

Overload 1 of 2, '(s: string): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'string'.

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

Overload 2 of 2, '(arr: any[]): number', gave the following error.

Argument of type 'number[] | "hello"' is not assignable to parameter of type 'any[]'.

Type 'string' is not assignable to type 'any[]'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwHYBsBOAsAFAAmkAxgDYCG0koAZgK4B2pALgJZxOjmRMAUKUIlbR2TAOYBKVEwYBbAEYwA3IRIVqtRiw5cefftWipKTAJ4BtALozQcpasIhQAWnekGrd68K8BAEQBUiqgLgDyANJ+hpYADLahEdEE-vwAspSsABYAdNBmRHDy-FKgAHygcbkArKAA-KAB2ZDk5HABoKjxiUA)

Because both overloads have the same argument count and same return type, we can instead write a non-overloaded version of the function:

function len(x: any[] | string) {

return x.length;

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAGwKZgBQA8BciCGYAngNoC6iAPogM5QBOMYA5gJSIDeAsAFCKL1UUEPSRYAdGhZQAFgG5eAXyA)

This is much better! Callers can invoke this with either sort of value, and as an added bonus, we don’t have to figure out a correct implementation signature.

Always prefer parameters with union types instead of overloads when possible

### **Declaring this in a Function**

TypeScript will infer what the this should be in a function via code flow analysis, for example in the following:

const user = {

id: 123,

admin: false,

becomeAdmin: function () {

this.admin = true;

},

};

[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBArhApgJxgXhgbwLACgYwCWAJgFwwCMATAMwA0eeBAhsQLaFjkBmzANkgb4YAI0Sg2iAILtOPOGGBRC4GAAoAlFiYEYUABaEIAOlYcw6PcjiIA3DoC+Qh7aA)

TypeScript understands that the function user.becomeAdmin has a corresponding this which is the outer object user. this, heh, can be enough for a lot of cases, but there are a lot of cases where you need more control over what object this represents. The JavaScript specification states that you cannot have a parameter called this, and so TypeScript uses that syntax space to let you declare the type for this in the function body.

interface DB {

filterUsers(filter: (this: User) => boolean): User[];

}

const db = getDB();

const admins = db.filterUsers(function (this: User) {

return this.admin;

});

[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgKoGdrIN4FgBQyywAJgFzIgCuAtgEbQDcBRcJNoFdA9twDYQ4IZvgC+BEhAR84UFAm4h0YZAHMIYACIAhCgAoAlMgC8APmQ6RAeivIAtA4RUwDuwVCRYiFDpwtkMMB8nhjQ6HqBwdD6YAAWwOgUoVBGZsg8-IIgBkmYUADaALoi4vgECkoqJHQmaho6hiIVyshsHEq11QB0kSF54TBUIAhgwIrIenEJudBGeITIcmBUUCDIU+hdbaAlBoxAA)

This pattern is common with callback-style APIs, where another object typically controls when your function is called. Note that you need to use function and not arrow functions to get this behavior:

interface DB {

filterUsers(filter: (this: User) => boolean): User[];

}

const db = getDB();

const admins = db.filterUsers(() => this.admin);

The containing arrow function captures the global value of 'this'.  
Element implicitly has an 'any' type because type 'typeof globalThis' has no index signature.The containing arrow function captures the global value of 'this'.  
Element implicitly has an 'any' type because type 'typeof globalThis' has no index signature.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygOwAYAsBGdGc0BYAKAEsA7AFxgDMBDAY0lAFVEZQBvU0UMgCaoKAVwC2AIxgBuXv0QBBAWMqoJcOABtI9CrJIBfUgMiNN9aC0ZwKiKqADmkKgBEAQqgAUASlABeAD5Qd30QUABaSMYRKkjw0koaaAZmYLduOVoyTST2GERPLJyYLyoACzIUNg5oX0DQdS0dCm9UPOgAbQBdfSMSUmtbewEJf0dndx99QbtQemVKRDGRgDoi3JqCn38g8sqV+ZUW6SA)

## Other Types to Know About

There are some additional types you’ll want to recognize that appear often when working with function types. Like all types, you can use them everywhere, but these are especially relevant in the context of functions.

### **void**

void represents the return value of functions which don’t return a value. It’s the inferred type any time a function doesn’t have any return statements, or doesn’t return any explicit value from those return statements:

// The inferred return type is void

function noop() {

return;

}

[Try](https://www.typescriptlang.org/play/#code/PTAEBUAsFNQSwHYDNoCdXQCagwFwK6oKi4CeADrHAM6gBuA9nJgLABQS+CAxrnA8QQMG5ABQBKUAG92oHNAJEA3OwC+QA)

In JavaScript, a function that doesn’t return any value will implicitly return the value undefined. However, void and undefined are not the same thing in TypeScript. There are further details at the end of this chapter.

void is not the same as undefined.

### **object**

The special type object refers to any value that isn’t a primitive (string, number, bigint, boolean, symbol, null, or undefined). This is different from the empty object type { }, and also different from the global type Object. It’s very likely you will never use Object.

object is not Object. **Always** use object!

Note that in JavaScript, function values are objects: They have properties, have Object.prototype in their prototype chain, are instanceof Object, you can call Object.keys on them, and so on. For this reason, function types are considered to be objects in TypeScript.

### **unknown**

The unknown type represents any value. This is similar to the any type, but is safer because it’s not legal to do anything with an unknown value:

function f1(a: any) {

a.b(); // OK

}

function f2(a: unknown) {

a.b();

'a' is of type 'unknown'.'a' is of type 'unknown'.

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYDsBGUOAOABgBYA2AWACgAzAVwDsBjAFwEs4HQacAKAQ1T8GATwCUoAN7VQofgDoARrzEBuUCFAB5ANLUAvtXrN2nbmgGpGAawZwA7gwnSqshcrUGgA)

This is useful when describing function types because you can describe functions that accept any value without having any values in your function body.

Conversely, you can describe a function that returns a value of unknown type:

function safeParse(s: string): unknown {

return JSON.parse(s);

}

// Need to be careful with 'obj'!

const obj = safeParse(someRandomString);

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEYD2A7AzgF3mpBbEASlCsHgMoYwCWKA5gFzaU20DcAsAFAD038AtILABXDIP5cAZsJRgMVVNiiSQABVhoQACjSNM1OgEpGMgNYokAdxTwA3l3jw4GYTBsApMgHkAcgDoABw1tNEMOTgBfLi5eeB8QEGB4DCR4ACMEMFgQaQh4SyoMAAt4AHIkNIArUoBCLmR0LArK+ABeJRV1GE0dPEJiUlwKA1owoA)

### **never**

Some functions never return a value:

function fail(msg: string): never {

throw new Error(msg);

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMAhjANgCgLYGcDmAXIrlAE4xj4CUxYApgG71mIDeAsAFCKJQAWZOAHdEDUQFEyQsjgLUA3NwC+QA)

The never type represents values which are never observed. In a return type, this means that the function throws an exception or terminates execution of the program.

never also appears when TypeScript determines there’s nothing left in a union.

function fn(x: string | number) {

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

// do something

} else if (typeof x === "number") {

// do something else

} else {

x; // has type 'never'!

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABMMAKAHgLkQZygJxjAHNEAfRMEAWwCMBTfASkQG8BYAKEURmEVRQAngAd6cfukQBeWYgBEeQiXksO3HogD0WxABM4uONXpQAFkWJceAX0T0ANjnq9+g0eMky58qnUaqbNaaOvqGOMamFiT2TvTBdo7OQRo86ADc2rpmAIY4iMJiiADkYPQAbozFAIQJXDZAA)

### **Function**

The global type Function describes properties like bind, call, apply, and others present on all function values in JavaScript. It also has the special property that values of type Function can always be called; these calls return any:

function doSomething(f: Function) {

return f(1, 2, 3);

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAEzgZTgWwKZQBYxgDmAFMAFyIBi408YAlIgN4CwAUIogE64jdJgJAIwAaRACZxAZgYBuDgF8gA)

This is an untyped function call and is generally best avoided because of the unsafe any return type.

If you need to accept an arbitrary function but don’t intend to call it, the type () => void is generally safer.

## Rest Parameters and Arguments

Background Reading:  
[Rest Parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters)  
[Spread Syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax)

### **Rest Parameters**

In addition to using optional parameters or overloads to make functions that can accept a variety of fixed argument counts, we can also define functions that take an unbounded number of arguments using rest parameters.

A rest parameter appears after all other parameters, and uses the ... syntax:

function multiply(n: number, ...m: number[]) {

return m.map((x) => n \* x);

}

// 'a' gets value [10, 20, 30, 40]

const a = multiply(10, 1, 2, 3, 4);

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAWxAG1gBzQTwBRgBciYIyARgKYBOANIgHRPLGkU0DaAugJSIDeAWABQiRNUpQQ1JMgbIAhpjx4AHnwC8APhKIAVInUBuEQF8RAeguIA5ApuIA5pIDOiAG4K0ISog4BGAAZ6ACZgxABmcIAWQK4RCAQXKEQFRA0UdCxcPCD6f1D6CPponiMgA)

In TypeScript, the type annotation on these parameters is implicitly any[] instead of any, and any type annotation given must be of the form Array<T> or T[], or a tuple type (which we’ll learn about later).

### **Rest Arguments**

Conversely, we can provide a variable number of arguments from an iterable object (for example, an array) using the spread syntax. For example, the push method of arrays takes any number of arguments:

const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

arr1.push(...arr2);

[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBAhgJwQRhgXhgbWQGhgJjwGYBdAbgFgAoUSWRBfdLAFjwFY8A2c6h5AHQAHAK4QAFgAoBMhvgCUZIA)

Note that in general, TypeScript does not assume that arrays are immutable. This can lead to some surprising behavior:

// Inferred type is number[] -- "an array with zero or more numbers",

// not specifically two numbers

const args = [8, 5];

const angle = Math.atan2(...args);

A spread argument must either have a tuple type or be passed to a rest parameter.A spread argument must either have a tuple type or be passed to a rest parameter.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYMDYCwAoEUASQDsAzGaSAE1ABcBPAB0lAEtFRSBXAWwBGMANoBdUAFoJoAEQBDUqDmw5jUAHd29ABagAXjDigEoPgja9BMRDIA0BIqTj1QiVgGN25du7kAbPzV6dSNLISQCdzhSRBdlAHMuAF5QYQAOW1AMUQBuSOjYpVJ4vzYUgFk5HQA6KoU0AApq5oTEAEocoA)

The best fix for this situation depends a bit on your code, but in general a const context is the most straightforward solution:

// Inferred as 2-length tuple

const args = [8, 5] as const;

// OK

const angle = Math.atan2(...args);

[Try](https://www.typescriptlang.org/play/#code/PTAEEkDsDMFMCd6wCagIYGdQCYC0AbWSAcwBcALUUgVwAdCBYAKAGMB7SDU9eYrAXlABtABwAaUAFYAuuiztOpANzMQoAPIBpZgq7oShUIICyaCgDozaSNgAU5h2l4YAlEqA)

Using rest arguments may require turning on [downlevelIteration](https://www.typescriptlang.org/tsconfig#downlevelIteration) when targeting older runtimes.

## Parameter Destructuring

Background Reading:  
[Destructuring Assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

You can use parameter destructuring to conveniently unpack objects provided as an argument into one or more local variables in the function body. In JavaScript, it looks like this:

function sum({ a, b, c }) {

console.log(a + b + c);

}

sum({ a: 10, b: 3, c: 9 });

The type annotation for the object goes after the destructuring syntax:

function sum({ a, b, c }: { a: number; b: number; c: number }) {

console.log(a + b + c);

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAZxAWwBQG9EEMA0iARoRIgL4BciOu1Y6RApgE4Dcx9jrHEXazFhQCUNALAAoRIggJkcADZMAdArgBzDLkQBqYrpnC2k8kA)

This can look a bit verbose, but you can use a named type here as well:

// Same as prior example

type ABC = { a: number; b: number; c: number };

function sum({ a, b, c }: ABC) {

console.log(a + b + c);

}

[Try](https://www.typescriptlang.org/play/#code/PTAEGUEMFsFNUgZ1ABwE4EsD2bSwB4woA2sAsAFAAuAnivAIIBCAwqALygDeCAXKADsArtABGsNAG5Qo-sLETpAYzkjxuAL6TKAMyEClVbANCIRACh6QANDNtLQG-sxYBKbpVCglWAYiykAHTEWADm5pCgANQy0d6u2hQaQA)

## Assignability of Functions

### **Return type void**

The void return type for functions can produce some unusual, but expected behavior.

Contextual typing with a return type of void does **not** force functions to **not** return something. Another way to say this is a contextual function type with a void return type (type voidFunc = () => void), when implemented, can return any other value, but it will be ignored.

Thus, the following implementations of the type () => void are valid:

type voidFunc = () => void;

const f1: voidFunc = () => {

return true;

};

const f2: voidFunc = () => true;

const f3: voidFunc = function () {

return true;

};

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAbg9gSwCYDECuA7AxlAvFACgEo8A+WRJAbgFgAoerODAZ2CgDMBGALguXTY8hErnIBvelCgAnCMDQyMUYDLQRadAL6bGzNpwBMfeAMw58xMirUb6e1uw4BmE5UEXO54AmYioknTScgpKNuqaOkA)

And when the return value of one of these functions is assigned to another variable, it will retain the type of void:

const v1 = f1();

const v2 = f2();

const v3 = f3();

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAbg9gSwCYDECuA7AxlAvFACgEo8A+WRJAbgFgAoerODAZ2CgDMBGALguXTY8hErnIBvelCgAnCMDQyMUYDLQRadAL6bGzNpwBMfeAMw58xMirUb6e1uw4BmE5UEXO54AmYioknTScgpKNuqaOvQA9NFQALSJWGjAifEOBjBcwtzEunRMjrCGOYZ59gX67DDOOc55QA)

This behavior exists so that the following code is valid even though Array.prototype.push returns a number and the Array.prototype.forEach method expects a function with a return type of void.

const src = [1, 2, 3];

const dst = [0];

src.forEach((el) => dst.push(el));

[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBBBOwYF4YG0CMAaGAmHAzALoDcAsAFCiSwAm0K6ADKZZQsAHQBmI8AogENgACwAUYgKYAbAJQoAfDHpROABwCuEcTNmySQA)

There is one other special case to be aware of, when a literal function definition has a void return type, that function must **not** return anything.

function f2(): void {

// @ts-expect-error

return true;

}

const f3 = function (): void {

// @ts-expect-error

return true;

};