(i) We use optional cookies to improve your experience on our websites, such as through social media connections, and to display personalized advertising based on your online activity. If you reject optional cookies, only cookies necessary to provide you the services will be used. You may change your selection by clicking "Manage Cookies" at the bottom of the page. [Privacy Statement](#) [Third-Party Cookies](#)

Accept   Reject   Manage cookies

Do's and Don'ts

On this page

Is this page helpful?

Search Docs

Docs   Community   Tools

General Types

`Number` , `String` , `Boolean` , `Symbol` and `Object`

❌ **Don't** ever use the types `Number` , `String` , `Boolean` , `Symbol` , or `Object` These types refer to non-primitive boxed objects that are almost never used appropriately in JavaScript code.

```
/* WRONG */
function reverse(s: String): String;
```

✅ **Do** use the types `number` , `string` , `boolean` , and `symbol` .

```
/* OK */
function reverse(s: string): string;
```

Instead of `Object` , use the non-primitive `object` type ([added in TypeScript 2.2](#)).

Generics

❌ **Don't** ever have a generic type which doesn't use its type parameter. See more details in [TypeScript FAQ page](#).

any

❌ **Don't** use `any` as a type unless you are in the process of migrating a JavaScript project to TypeScript. The compiler *effectively* treats `any` as "please turn off type checking for this thing". It is similar to putting an `@ts-ignore` comment around every usage of the variable. This can be very helpful when you are first migrating a JavaScript project to TypeScript as you can set the type for stuff you haven't migrated yet as `any` , but in a full TypeScript project you are disabling type checking for any parts of your

Docs   Community   Tools

it, you can use **unknown**.

Callback Types

Return Types of Callbacks

❌ **Don't** use the return type any for callbacks whose value will be ignored:

```
/* WRONG */
function fn(x: () => any) {
  x();
}
```

✅ **Do** use the return type void for callbacks whose value will be ignored:

```
/* OK */
function fn(x: () => void) {
  x();
}
```

❓ **Why:** Using void is safer because it prevents you from accidentally using the return value of x in an unchecked way:

```
function fn(x: () => void) {
  var k = x(); // oops! meant to do something else
  k.doSomething(); // error, but would be OK if the return type ha
}
```

Optional Parameters in Callbacks

❌ **Don't** use optional parameters in callbacks unless you really mean it:

Docs   Community   Tools

```
}
```

Th   t
or might be invoked with 2 arguments. The author probably intended to say that the
callback might not care about the `elapsedTime` parameter, but there's no need to
make the parameter optional to accomplish this — it's always legal to provide a
callback that accepts fewer arguments.

✅ **Do** write callback parameters as non-optional:

```
/* OK */
interface Fetcher {
  getObject(done: (data: unknown, elapsedTime: number) => void):
}
```

Overloads and Callbacks

❌ **Don't** write separate overloads that differ only on callback arity:

```
/* WRONG */
declare function beforeAll(action: () => void, timeout?: number):
declare function beforeAll(
  action: (done: DoneFn) => void,
  timeout?: number
): void;
```

✅ **Do** write a single overload using the maximum arity:

```
/* OK */
declare function beforeAll(
```

Docs   Community   Tools

❓   **Why:** It's always legal for a callback to disregard a parameter, so there's no need for the shorter overload. Providing a shorter callback first allows incorrectly-typed functions to be passed in because they match the first overload.

Function Overloads

Ordering

❌ **Don't** put more general overloads before more specific overloads:

```
/* WRONG */
declare function fn(x: unknown): unknown;
declare function fn(x: HTMLElement): number;
declare function fn(x: HTMLDivElement): string;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: unknown, wat?
```

✅ **Do** sort overloads by putting the more general signatures after more specific signatures:

```
/* OK */
declare function fn(x: HTMLDivElement): string;
declare function fn(x: HTMLElement): number;
declare function fn(x: unknown): unknown;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: string, :)
```

❓   **Why:** TypeScript chooses the *first matching overload* when resolving function calls. When an earlier overload is "more general" than a later one, the later one is effectively hidden and cannot be called.

Use Optional Parameters

❌

Docs   Community   Tools

```
  diff(one: string): number;
  diff(one: string, two: string): number;
  diff(one: string, two: string, three: boolean): number;
}
```

✅ **Do** use optional parameters whenever possible:

```
/* OK */
interface Example {
  diff(one: string, two?: string, three?: boolean): number;
}
```

Note that this collapsing should only occur when all overloads have the same return type.

❔ **Why:** This is important for two reasons.

TypeScript resolves signature compatibility by seeing if any signature of the target can be invoked with the arguments of the source, *and extraneous arguments are allowed*. This code, for example, exposes a bug only when the signature is correctly written using optional parameters:

```
function fn(x: (a: string, b: number, c: number) => void) {}
var x: Example;
// When written with overloads, OK -- used first overload
// When written with optionals, correctly an error
fn(x.diff);
```

The second reason is when a consumer uses the "strict null checking" feature of TypeScript. Because unspecified parameters appear as `undefined` in JavaScript, it's usually fine to pass an explicit `undefined` to a function with optional arguments. This code, for example, should be OK under strict nulls:

```
var x: Example;
// When written with overloads, incorrectly an error because of n
```

Docs   Community   Tools

Use Union Types

❌ **Don't** write overloads that differ by type in only one argument position:

```
/* WRONG */
interface Moment {
  utcOffset(): number;
  utcOffset(b: number): Moment;
  utcOffset(b: string): Moment;
}
```

✅ **Do** use union types whenever possible:

```
/* OK */
interface Moment {
  utcOffset(): number;
  utcOffset(b: number | string): Moment;
}
```

Note that we didn't make `b` optional here because the return types of the signatures differ.

❓ **Why:** This is important for people who are "passing through" a value to your function:

```
function fn(x: string): Moment;
function fn(x: number): Moment;
function fn(x: number | string) {
  // When written with separate overloads, incorrectly an error
  // When written with union types, correctly OK
  return moment().utcOffset(x);
}
```

Docs   **Community**   **Tools**

The TypeScript docs
are an open source
project. Help us
improve these pages
by sending a Pull
Request ❤

Contributors to this
page:

MH 　 MZ 　 MF

J 　 15+

Last updated: Oct 27,
2025

This page loaded in
0.178 seconds.

Customize

Site Colours: | System |

Code Font: | Cascadia |

Popular Documentation Pages

Everyday Types

All of the common types in
TypeScript

Creating Types from Types

Techniques to make more
elegant types

More on Functions

How to provide types to
functions in JavaScript

More on Objects

How to provide a type shape to
JavaScript objects

Narrowing

How TypeScript infers types
based on runtime behavior

Variable Declarations

How to create and type
JavaScript variables

TypeScript in 5 minutes

An overview of building a
TypeScript web app

TSConfig Options

All the configuration options for
a project

Classes

How to provide types to
JavaScript ES6 classes

Community

Get Help

Blog

GitHub Repo

Docs 　 Community 　 Tools

Using TypeScript

Get Started

Download

Community

Playground

TSConfig Ref

Code Samples

Why TypeScript

Design

Made with ❤ in Redmond,
Boston, SF & Dublin

© 2012-2025 Microsoft

PrivacyTerms of Use

Docs

Community

Tools