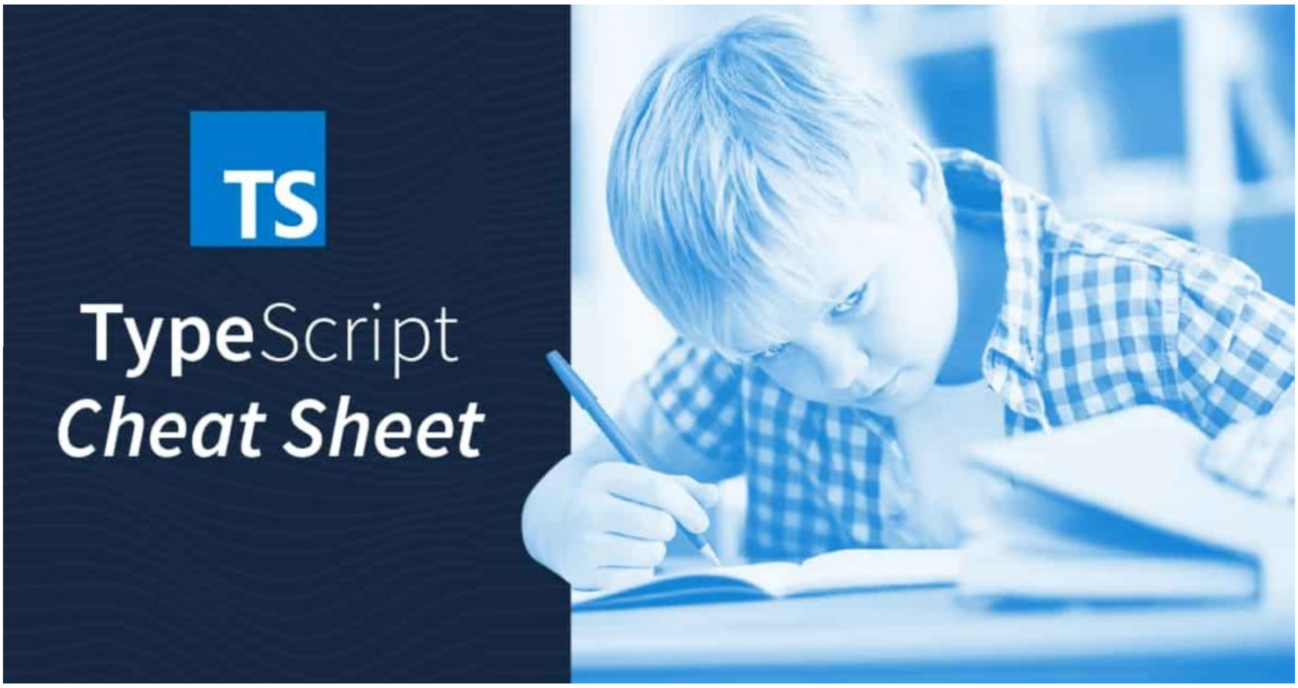
Search...

TypeScript 4.4 Cheat Sheet

SitePen Engineering | October 18, 2021



JAVASCRIPT TYPESCRIPT

This cheat sheet is an adjunct to our Definitive TypeScript Guide.

Originally published November 2018. **Updated October 2021 for TypeScript 4.4**.

Usage	
Install	`npm install TypeScript`
Run	`npx tsc`

Triple slash directives	
Reference built-in types	`/// <reference lib="es2016.array.include" />`

Reference other types	`/// <reference path="../my_types" />`
	`/// <reference types="jquery" />`

AMD	`/// <amd-module name="Name" />`
	`/// <amd-dependency path="app/foo" name="foo" />`

Compiler comments	
Don’t check this file	`// @ts-nocheck`
Check this file (JS)	`// @ts-check`
Ignore the next line	`// @ts-ignore`
Expect an error on the next line	`// @ts-expect-error`
Operators (TypeScript-specific and draft JavaScript)	

`??` (nullish coalescing)	`function getValue(val?: number): number | 'nil' {`
	`  // Will return 'nil' if `val` is falsey (including 0)`

<table>
<tr><td><code>?.</code> (optional chaining)</td><td><pre>
function countCaps(value?: string) {
  // The `value` expression be undefined if `value` is null or
  // undefined, or if the `match` call doesn't find anything.
  return value?.match(/[A-Z]/g)?.length ?? 0;
}
</pre></td></tr>
<tr><td><code>!</code> (null assertion)</td><td><pre>
let value: string | undefined;

// ... Code that we're sure will initialize `value` ...

// Assert that `value` is defined
console.log(`value is ${value!.length} characters long`);
</pre></td></tr>
<tr><td>&&=</td><td><pre>
let a;
let b = 1;

// assign a value only if current value is truthy

a &&= 'default'; // a is still undefined
b &&= 5; // b is now 5
</pre></td></tr>
<tr><td><code>||=</code></td><td><pre>
let a;
let b = 1;

// assign a value only if current value is falsy

a ||= 'default'; // a is 'default' now
b ||= 5; // b is still 1
</pre></td></tr>
<tr><td><code>??=</code></td><td><pre>
let a;
let b = 0;

// assign a value only if current value is null or undefined
</pre></td></tr>
</table>

Basic types

Untyped	`any`
A string	`string`
A number	`number`
A true / false value	`boolean`
A non-primitive value	`object`
Uninitialized value	`undefined`
Explicitly empty value	`null`
Null or undefined (usually only used for function returns)	`void`
A value that can never occur	`never`

Object types

Object

```
{
    requiredStringVal: string;
    optionalNum?: number;
    readonly readOnlyBool: bool;

}
```

Object with arbitrary string properties (like a hashmap or dictionary)

```
{ [key: string]: Type; }
{ [key: number]: Type; }
{ [key: symbol]: Type; }
{ [key: `data-${string}`]: Type; }
```

Literal types

String

```
let direction: 'left' | 'right';
```

Numeric

```
let roll: 1 | 2 | 3 | 4 | 5 | 6;
```

Arrays and tuples

Array of strings

```
string[]
```

or

```
Array<string>
```

Array of functions

```
(() => string)[]
```

	or `Array<() => string>`

Basic tuples	`let myTuple: [ string, number, boolean? ];`
	`myTuple = [ 'test', 42 ];`

Variadic tuples	`type Numbers = [number, number];`
	`type Strings = [string, string];`
	`type NumbersAndStrings = [...Numbers, ...Strings];`
	`// [number, number, string, string]`
	`type NumberAndRest = [number, ...string[]];`
	`// [number, varying number of string]`
	`type RestAndBoolean = [...any[], boolean];`
	`// [varying number of any, boolean]`

Named tuples	`type Vector2D = [x: number, y: number];`
	`function createVector2d(...args: Vector2D) {}`
	`// function createVector2d(x: number, y: number): void`

Functions	

Function type	`(arg1: Type, argN: Type) => Type;`
	or
	`{ (arg1: Type, argN: Type): Type; }`

	`{ new (): ConstructedType; }`
Function type with optional param	`(arg1: Type, optional?: Type) => ReturnType`
Function type with rest param	`(arg1: Type, ...allOtherArgs: Type[]) => ReturnType`
Function type with static property	`{ (): Type; staticProp: Type; }`
Default argument	`function fn(arg1 = 'default'): ReturnType {}`

Arrow function	`(arg1: Type): ReturnType => { ...; return value; }`
	or
	`(arg1: Type): ReturnType => value;`

`this` typing	`function fn(this: Foo, arg1: string) {}`

Overloads	`function conv(a: string): number;`
	`function conv(a: number): string;`
	`function conv(a: string | number): string | number {`
	`    ...`
	`}`

```
let myUnionVariable: number | string;
```

Intersection

```
let myIntersectionType: Foo & Bar;
```

Named types

Interface

```
interface Child extends Parent, SomeClass {
    property: Type;
    optionalProp?: Type;
    optionalMethod?(arg1: Type): ReturnType;
}
```

Class

```
class Child
extends Parent

implements Child, OtherChild {
    property: Type;
    defaultProperty = 'default value';
    private _privateProperty: Type;
    private readonly _privateReadonlyProperty: Type;
    static staticProperty: Type;

    static {
        try {
            Child.staticProperty = calcStaticProp();
        } catch {
            Child.staticProperty = defaultValue;
        }
    }

    constructor(arg1: Type) {
        super(arg1);
    }

    private _privateMethod(): Type {}

    methodProperty: (arg1: Type) => ReturnType;
    overloadedMethod(arg1: Type): ReturnType;
}
```

Enum

```
enum Options {
    FIRST,
    EXPLICIT = 1,
    BOOLEAN = Options.FIRST | Options.EXPLICIT,
    COMPUTED = getValue()
}

enum Colors {
    Red = "#FF0000",
    Green = "#00FF00",
    Blue = "#0000FF"
}
```

Type alias

```
    type Name = string;

    type Direction = 'left' | 'right';

    type ElementCreator = (type: string) => Element;

    type Point = { x: number, y: number };

    type Point3D = Point & { z: number };

    type PointProp = keyof Point; // 'x' | 'y'

    const point: Point = { x: 1, y: 2 };

    type PtValProp = keyof typeof point; // 'x' | 'y'
```

Generics

Function using type parameters

```
<T>(items: T[], callback: (item: T) => T): T[]
```

Interface

Constrained type parameter

```
<T extends ConstrainedType>(): T
```

Default type parameter

```
<T = DefaultType>(): T
```

Constrained and default type parameter

```
<T extends ConstrainedType = DefaultType>(): T
```

Generic tuples

```
type Arr = readonly any[];

function concat<U extends Arr, V extends Arr>(a: U, b: V):
[...U, ...V] { return [...a, ...b] }

const strictResult = concat([1, 2] as const, ['3', '4'] as const);
const relaxedResult = concat([1, 2], ['3', '4']);

// strictResult is of type [1, 2, '3', '4']
// relaxedResult is of type (string | number)[]
```

Index, mapped, and conditional types

Index type query (keyof)

```
type Point = { x: number, y: number };
let pointProp: keyof Point = 'x';

function getProp<T, K extends keyof T>(
    val: T,
    propName: K

): T[K] { ... }
```

Mapped types

```
type Stringify<T> = { [P in keyof T]: string; }
```

Conditional types

```
type Swapper = <T extends number | string>

    (value: T) => T extends number ? string : number;
```

is equivalent to

```
(value: number) => string
```

if T is number, or

```
(value: string) => number
```

if T is string

Conditional mapped types

```
interface Person {
    firstName: string;
    lastName: string;
    age: number;
}

type StringProps<T> = {
    [K in keyof T]: T[K] extends string ? K : never;
};

type PersonStrings = StringProps<Person>;

// PersonStrings is "firstName" | "lastName"
```

Utility types

Partial

```
Partial<{ x: number; y: number; z: number; }>
```

is equivalent to

```
{ x?: number; y?: number; z?: number; }
```

	`{`
	`    readonly x: number;`
	`    readonly y: number;`
	`    readonly z: number;`
	`}`

Pick	`Pick<{ x: number; y: number; z: number; }, 'x' | 'y'>`
	is equivalent to
	`{ x: number; y: number; }`

Record	`Record<'x' | 'y' | 'z', number>`
	is equivalent to
	`{ x: number; y: number; z: number; }`

Exclude	`type Excluded = Exclude<string | number, string>;`
	is equivalent to
	`number`

Extract	`type Extracted = Extract<string | number, string>;`
	is equivalent to
	`string`

is equivalent to

```
string | number
```

ReturnType

```
type ReturnValue = ReturnType<() => string>;
```

is equivalent to

```
string
```

InstanceType

```
class Renderer() {}
```

```
type Instance = InstanceType<typeof Renderer>;
```

is equivalent to

```
Renderer
```

Type guards

Type predicates

```
function isThing(val: unknown): val is Thing {
    // return true if val is a Thing
}

if (isThing(value)) {
    // value is of type Thing
}
```

`typeof`

```
declare value: string | number | boolean;
const isBoolean = typeof value === “boolean”;
```

```
}
```

instanceof

```
declare value: Date | Error | MyClass;

const isMyClass = value instanceof MyClass;

if (value instanceof Date) {
    // value is a Date
} else if (isMyClass) {
    // value is an instance of MyClass
} else {
    // value is an Error
}
```

in

```
interface Dog { woof(): void; }
interface Cat { meow(): void; }

function speak(pet: Dog | Cat) {
    if ('woof' in pet) {
        pet.woof()
    } else {
        pet.meow()
    }
}
```

Assertions

Type

```
let val = someValue as string;
```

or

```
let val = <string>someValue;
```

Const (immutable

```
let point = { x: 20, y: 30 } as const;
```

Ambient declarations	
Global	`declare const $: JQueryStatic;`

Module	`declare module "foo" {`
	`    export class Bar { ... }`
	`}`

Wildcard module	`declare module "text!*" {`
	`    const value: string;`
	`    export default value;`
	`}`

Is this cheat sheet missing anything? Let us know.

Is it time to asess if your team should be using TypeScript?

Connect with us for a **free technical assessment** and ask the experts whether TypeScript is the right choice for your application.