Typescript Cheatsheet

devhints.io

December 23, 2022

Abstract

Typescript cheatsheet adapted from devhints.io

Contents

Introduction

TypeScript is a programming language developed and maintained by Microsoft. It is a typed superset of JavaScript that compiles to plain JavaScript. It adds optional static typing to the language and helps to catch common mistakes in code by providing type information at compile time. TypeScript can also be used to develop JavaScript applications for both client-side and server-side execution. It is open-source and cross-platform, meaning it can be used to create applications for the web, desktop, and mobile devices. The language is designed for large-scale applications and transcompiles to JavaScript, making it easier to read and debug. It supports features like classes, modules, and interfaces, which helps to write object-oriented code. Additionally, TypeScript can be used to create definition files that can be used to describe existing JavaScript libraries, providing better intellisense and type safety when using them.

Typescript cheatsheet

Basic types

```
any
void

boolean
number
string

null
undefined

bigint
symbol

string[]          /* or Array<string> */
[string, number]  /* tuple */

string | null | undefined   /* union */

never  /* unreachable */
unknown

enum Color {
  Red,
  Green,
  Blue = 4
};

let c: Color = Color.Green
```

Declarations

```
let isDone: boolean
let isDone: boolean = false

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

```
// Return type is optional
function add (a: number, b: number) { ... }
```

Type assertions

Variables

```
let len: number = (input as string).length
let len: number = (<string> input).length  /* not allowed in JSX */
```

Functions

```
function object(this: {a: number, b: number}, a: number, b: number) {
  this.a = a;
  this.b = b;
  return this;
}

// this is used only for type declaration
let a = object(1,2);
// a has type {a: number, b: number}
```

Interfaces

Inline

```
function printLabel (options: { label: string }) {
  console.log(options.label)
}

// Note the semicolon
function getUser (): { name: string; age?: number } {
}
```

Explicit

```
interface LabelOptions {
  label: string
}

function printLabel(options: LabelOptions) { ... }
```

Optional properties

```
interface User {
  name: string;
  age?: number;
}
```

Read only

```
interface User {
  readonly name: string
}
```

Dynamic keys

```
{
  [key: string]: Object[]
}
```

Type aliases

```
type Name = string | string[]

Intersection

interface Colorful { ... }

interface Circle { ... }

type ColorfulCircle = Colorful & Circle;
```

Function types

```
interface User { ... }

function getUser(callback: (user: User) => any) { callback({...}) }

getUser(function (user: User) { ... })
```

Classes

```
class Point {
  x: number
  y: number
  static instances = 0
  constructor(x: number, y: number) {
    this.x = x
    this.y = y
  }
}
```

Inheritance

```
class Point {...}

class Point3D extends Point {...}

interface Colored {...}
```

```
class Pixel extends Point implements Colored {...}
```

Short fields initialisation

```
class Point {
  static instances = 0;
  constructor(
    public x: number,
    public y: number,
  ){}
}
```

Fields which do not require initialisation

```
class Point {
  public someUselessValue!: number;
  ...
}
```

Generics

```
class Greeter<T> {
  greeting: T
  constructor(message: T) {
    this.greeting = message
  }
}

let greeter = new Greeter<string>('Hello, world')
```

Modules

```
export interface User { ... }
```

Type extraction

```
interface Building {
  room: {
    door: string;
    walls: string[];
  };
}

type Walls = Building['room']['walls']; // string[]
```

Key of Type Operator

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

type P = keyof Point; // x | y
```

Conditinal Types

```
// SomeType extends OtherType ? TrueType : FalseType;

type ToArray<T> = T extends any ? T[] : never;

type StrArrOrNumArr = ToArray<string | number>; // string[] | number[]
```

Inferring

```
type GetReturnType<T> = T extends (...args: unknown[]) => infer R
  ? R
  : never;

type Num = GetReturnType<() => number>; // number

type First<T extends Array<any>> = T extends [infer F, ...infer Rest] ? F : never;

type Str = First<['hello', 1, false]>; // 'hello'
```

Literal Type

```
const point = { x: 4, y: 2 }; // { x: number, y: number }

const literalPoint = { x: 4, y: 2 } as const; // { readonly x: 4, readonly y: 2 };
```

Template Literal Types

```
type SpaceChar = ' ' | '\n' | '\t';

type TrimLeft<S extends string> = S extends `${SpaceChar}${infer Rest}` ? TrimLeft<Rest> : S;

type Str = TrimLeft<'    hello'>; // 'hello'
```