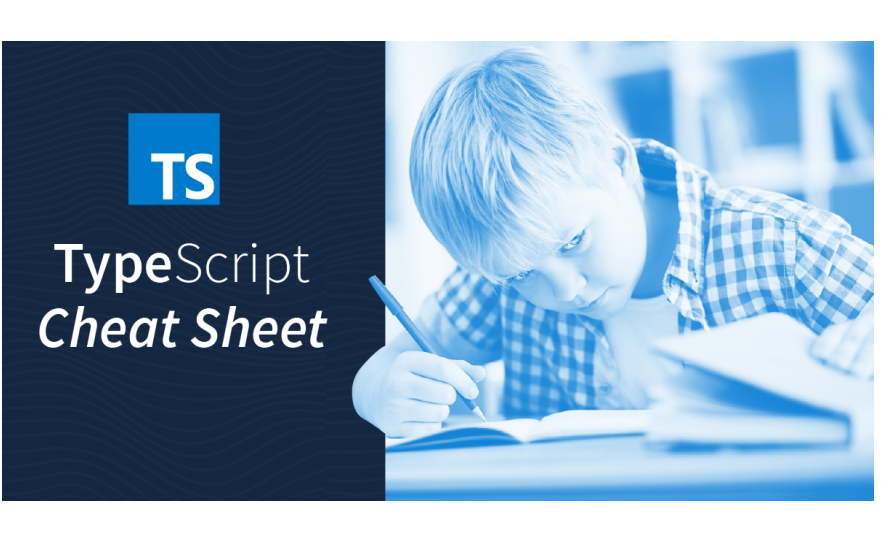
TypeScript Cheat Sheet

By Nick Nisi on November 6, 2018 6:45 am

JavaScript TypeScript



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

Last updated for: TypeScript 3.1

Usage

```
npm install typescript
tsc
```

Primitive types

Any type (explicitly untyped)
```
any
```

void type (undefined or null, use for function returns only)
```
void
```

Undefined type
```
undefined
```

Null type
```
null
```

never type
```
never
```

unknown type
```
unknown
```

String (including ES6 multi-line string templates)
```
string
```

Number
```
number
```

Boolean
```
boolean
```

object (may be an Object or non-primitive)
```
object
```

Named types (interface, class, enum)

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: Type = 'default value';
  private _privateProperty: Type;
  private readonly _privateReadonlyProperty: Type;
  static staticProperty: Type;
  constructor(arg1: Type) {
    super(arg1);
  }
  private _privateMethod(): Type {}
  methodProperty: (arg1: Type) => ReturnType;
  overloadedMethod(arg1: Type): ReturnType;
  overloadedMethod(arg1: OtherType): ReturnType;
  overloadedMethod(arg1: CommonT): CommonReturnT {}
  static staticMethod(): ReturnType {}
  subclassedMethod(arg1: Type): ReturnType {
    super.subclassedMethod(arg1);
  }
}
```

Enum
```
enum Options {
  FIRST,
  EXPLICIT = 1,
  BOOLEAN = Options.FIRST | Options.EXPLICIT
}
enum Colors {
  Red = "#FF0000",
  Green = "#00FF00",
  Blue = "#0000FF"
}
```

Object type literals

Object with implicit Any properties
```
{ foo; bar; }
```

Object with optional property
```
{ required: Type; optional?: Type; }
```

Hash map
```
{ [key: string]: Type; }
```

Union and intersection types

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

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

Arrays and tuples

Array of strings

`string[]` or `Array<string>`

Array of functions that return strings

`{ (): string; }[]` or `Array<() => string>`

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

Functions

Function

`{ (arg1: Type, argN: Type): Type; }` or

`(arg1: Type, argN: Type) => Type;`

Constructor

`{ new (): ConstructedType; }` or

`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: Type = 'default'): ReturnType {}
```

Arrow function

`(arg1: Type): ReturnType => {}` or

`(arg1: Type): ReturnType => Expression`

this typing
```
function fn(this: Foo)
```

Generics

Function using type parameters
```
<T>(items: T[], callback: (item: T) => T): T[]
```

Interface with multiple types
```
interface Pair<T1, T2> {
  first: T1;
  second: T2;
}
```

Constrained type parameter
```
<T extends ConstrainedType>(): T
```

Default type parameter
```
<T = ConstrainedType>(): T
```

Constrained and default type parameter
```
<T extends ConstrainedType = ConstrainedType>(): T
```


Partial type

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

is equivalent to

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

Readonly type

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

is equivalent to
```
{
  readonly x: number;
  readonly y: number;
  readonly z: number;
}
```

Pick type

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

is equivalent to

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

Record type

`Record<'x' | 'y' | 'z', number>`

is equivalent to

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

Conditional types

Conditional types
```
declare function createLabel<T extends number | string>(idOrName: T): T extends number ? Id : Name;
```

Exclude

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

is equivalant to

`number`

Extract

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

is equivalant to

`string`

NonNullable

`type NonNull = NonNullable<string | number | void>;`

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`

Other

Type of a variable
```
typeof varName
```

Is this cheat sheet missing anything? Let us know.

Other posts in the series

1. The Definitive TypeScript Guide
2. TypeScript Cheat Sheet
3. Advanced TypeScript Concepts: Classes and Types