TypeScript Cookbook

Other Ressources

- Release Notes - Overview
- Compiler Options
- tsconfig.json

Initialize TS

```
tsc --init
```

Show (Effective) Configuration

```
tsc --showConfig
```

Disable Lint & Type Checking for a file

```
// tslint:disable
// @ts-nocheck
```

Using tslint and prettier without conflicts

```
{
  "extends": [
    "tslint:latest",
    "tslint-plugin-prettier",
    "tslint-config-prettier"
  ],
  "rules": {
    "prettier": [true, ".prettierrc"]
  }
}
```

Compiler Options

Some Checks

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "strictNullChecks": true
  }
}
```

Use same name for Class and Interface

You don't have to repeat prop-names within the class:

```
interface SomeClass {
  propOne: string;
  propTwo: number;
}

class SomeClass {
  constructor(one: string, two: number) {
    this.propOne = one;
    this.propTwo = two;
  }
}
```

Mixin Classes

> See here: https://github.com/microsoft/TypeScript/pull/13743

Type Definitions

Built-In Basic Types and Interfaces

```
interface ArrayLike<T> {
  readonly length: number;
  readonly [n: number]: T;
}

/**
 * Make all properties in T optional
 */
type Partial<T> = {
  [P in keyof T]?: T[P];
};

/**
 * Make all properties in T required
 */
type Required<T> = {
  [P in keyof T]-?: T[P];
};

/**
 * Make all properties in T readonly
 */
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

/**
 * From T, pick a set of properties whose keys are in the union K
 */
type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
};

/**
 * Construct a type with a set of properties K of type T
 */
type Record<K extends keyof any, T> = {
  [P in K]: T;
};

/**
 * Exclude from T those types that are assignable to U
 */
type Exclude<T, U> = T extends U ? never : T;

/**
 * Extract from T those types that are assignable to U
 */
type Extract<T, U> = T extends U ? T : never;

/**
 * Construct a type with the properties of T except for those in type K.
 */
```

```
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;

/**
 * Exclude null and undefined from T
 */
type NonNullable<T> = T extends null | undefined ? never : T;

/**
 * Obtain the parameters of a function type in a tuple
 */
type Parameters<T extends (...args: any) => any> = T extends (
  ...args: infer P
) => any
  ? P
  : never;

/**
 * Obtain the parameters of a constructor function type in a tuple
 */
type ConstructorParameters<
  T extends new (...args: any) => any
> = T extends new (...args: infer P) => any ? P : never;

/**
 * Obtain the return type of a function type
 */
type ReturnType<T extends (...args: any) => any> = T extends (
  ...args: any
) => infer R
  ? R
  : any;

/**
 * Obtain the return type of a constructor function type
 */
type InstanceType<T extends new (...args: any) => any> = T extends new (
  ...args: any
) => infer R
  ? R
  : any;

/**
 * Marker for contextual 'this' type
 */
interface ThisType<T> {}
```

> Advanced Types: https://www.typescriptlang.org/docs/handbook/advanced-types.html

Extending Types

> Diskussion: https://github.com/Microsoft/TypeScript/pull/13604

Old Method

```
type UserEvent = Event & {UserId: string}
```

New Method (TS ^2.2)

```
type Event = {
   name: string;
   dateCreated: string;
   type: string;
}

interface UserEvent extends Event {
   UserId: string;
}
```

An Array of a certain type at least with e.g. 2 elements

>> Source StackOverflow

See *Rest Elements in tuple types* here: https://www.typescriptlang.org/docs/handbook/release-notes/overview.html#rest-elements-in-tuple-types

```
let foo: [Foo, Foo, ...Foo[]];
```

And see additionally *User-defined type guards* here: https://www.typescriptlang.org/docs/handbook/advanced-types.html#user-defined-type-guards

To define a type guard, we simply need to define a function whose return type is a type predicate:

```
function isAtLeastTwoFoos(x: Foo[]): x is [Foo, Foo, ...Foo[]] {
  return x.length >= 2;
}

if (isAtLeastTwoFoos(fooArray)) {
  foo = fooArray; // okay
}
```

Same Type of Elements

```
type UnionKeys<U> = U extends U ? keyof U : never;
```

```
const test = <T>(x: T & Record<keyof T, Record<UnionKeys<T[keyof T]>,
number>>) => true;

const x = test({
  a: {
    x: 1,
    y: 3,
    z: 5
  },
  b: {
    x: 1,
    y: 2,
    z: 7

  },
})
```

An Excluding List for Initializer

```
interface MyExcludingList {
  moonWalking: Function;
  dancing: Function;
}

let exclude: keyof MyExcludingList;

class WhatEver {
  public id!: number;
  public name!: string;
  public date!: Date;
  public moonWalking!: Function;

  constructor(
    prop: {
      [prop in keyof WhatEver]?: WhatEver[prop] extends
MyExcludingList[typeof exclude]
        ? never
        : WhatEver[prop];
    }
  ) {
    Object.assign(this, prop);
  }
}

const whatEver = new WhatEver({
  id: 0,
  name: 'Michael Jackson',
  // moonWalking:
  // No value exists in scope for the shorthand property 'moonWalking'
});
```