TypeScript Cheatsheet

Typescript is a superset of Javascript which adds optional static typing to the language. It is very popular for large scale applications and helps reduce a lot of errors in the development mode only.

In this cheatsheet you'll find all the things you need to know to get started with typescript.

How to compile TypeScript code to Javascript?

- Open terminal and move to correct directory
- Type following command in terminal and press enter

```
tsc <typescript_f.ts> <javascript_f.js>
```

- This will compile the typescript file *typescript_file.ts* into a javascript file called *javascript_f.js*
- If we don't provide a name for the javascript file, typescript will automatically create one js file with the same name as that of typescript file

How to make compiler work continuously?

Open terminal and type

```
tsc <typescript_f.ts> -w
```

This means typescript's gonna keep watching *typescript_f.ts* and hence will automatically compile it to javascript every time we make a change

What does tsconfig file do?

- When working on projects, we will have multiple, nested typescript and javascript files so we can't compile them all with one command. For that we use `tsconfig.json` file
- Open the terminal in root directory and run

```
tsc --init
```

- Find and add the required root and output directory path in tsconfig file
- Now typescript will automatically be compiled based on the provided directories

Type Basics

- We can change the value of a variable but not its type

```
let character = "Deekayy";

character = 20; // type is number
character = "John"; type is string
```

- We don't have to tell typescript what type we're using as it uses *inference* to identify it based on the value
- We can also declare what type of variable we expect as a function argument

```
const circ = (diameter: number) => {
return diameter * Math.PI
}

circ("Hello") // can't take strings
circ(5) // numbers are welcome
```

- This checks for the type happen before we compile it to javascript that's why we won't see it in the compiled javascript file

Arrays

If array has all elements of one type we can't mutate it to add element of different type

```
let names = ['John', 'Sam', 'Deekayy'];

names.push('Josh'); ✅ // strings are welcome
names.push(5); ❌ // can't push number type
names[1] = 3 ❌ // can't change string to number
```

We can have an array with different types but we have to do that right when we declare the array

```
let names = ['John', 2, true];

names.push('Josh'); ✅ // strings are welcome
names.push(5); ✅ // can push number
names[1] = 3 ✅ // can change string to number
```

Also, once we declare an array with a name, we can't assign a value of different type to that variable

Objects

The properties of an object works the same way as a variable in typescript, they can't be assigned a different type once declared

```
// define an object
let person = {
name: 'mario',
belt: 'black',
age: 30,
};

person.age = 40; ✅
person.name = 'John'; ✅
person.age = '17'; ❌
```

We can't assign new properties to an object once it is defined

```
person.skills = ['coding', 'cricket']; ❌
```

Explicit Types

Giving explicit type to a variable means manually making the variable be a specific type so we can't add a value of any other type to it

```
let name: string;
let age: number;
let isLoggedIn: boolean;

name: 'Deekayy'; // ✅
age: '20'; // ❌
```

array

If we need to declare an array of a specific type we can do that in the following way

```
let skills: string[]; // means array can only take strings

skills = ['coding', 'teaching']; // ✅
skills = [true, '20']; // ❌ can't add boolean

// !note that the above isn't an empty array so we can't use push method here
// for that it should be `let skills = string[] = [];`
```

`union` types

If we have to use multiple types inside an array, we can use `union types` denoted by `|`, it basically represents `or`

```
let mixed: (string | number)[] = []; // this will allows us to have strings as well as numbers inside the array

mixed.push('John'); // ✅ a string
mixed.push(20); // ✅ a number
mixed.push(false); // ✅ a boolean
```

We can also use union type for variables

```
let gameID: string | number;

gameID = 'achbfjbyug221'; // ✅ a string
gameID = 254; // ✅ a number
gameID = true; // ✅ a boolean
```

objects

We can explicitly make a variable object

```
let personOne: object;

personOne = { name: 'Deekayy', age: 20 }; // ✅
personOne = ['John', 'Joe']; // ✅, array is an object too
personOne = 'John'; // ❌
```

We can also be more specific if we want, for example

```
let personTwo: {
  name: string,
  age: number,
};

personTwo = { name: 'Deekayy', age: 20 }; // ✅
personTwo = { name: 'Deekayy', age: '20' }; // ❌, age can't be string
personTwo = { name: 'Deekayy', skills: ['coding'] }; // ❌, can't change properties
```

`any` type

It allows to have a variable of any type, i.e, we can edit the variable to a value of any type we want

```
let age: any = 25;

age = true; // ✅ can be made boolean
age = 'hello'; // ✅ can be made string
age = [20, 25]; // ✅ can be made array
age = { today: 25, yesterday: 24 }; // ✅ can be made object
```

It basically reverts typescript back to javascript which can be useful at certain times

Functions

If we declare a function with name `x` typescript will automatically assign it type function and then later we can't assign any other type of value to `x`

We can explicitly define a function w/o assigning it a value

```
let greet: Function; // should be capital F

// some lines of code

greet = () => console.log('hello'); // ✅
greet = 'hello'; // ❌ can't be string
```

Parameters

We can specify the type of parameters passed inside the functions

```
const age = (n: number, name: string) => {
  console.log(`${name} is ${n} years old`);
};
```

We can also pass optional parameters

```
const age = (n: number, name: string, DOB?: number | string) => {
  console.log(`${name} is ${n} years old`);
  console.log(DOB); // will be undefined if not passed
};
```

We can pass default parameters and hence there's no need for optional one since there's always a default value

```
const age = (n: number, name: string, gender: string = 'male') => {
  console.log(`${name} is ${n} years old`);
  console.log(gender); // will be male if not passed
};
```

returns

When we return something from a function, typescript can *inference* on it's own the type of the return result

```
const add = (a: number, b: number) => {
  return a + b;
};

const result = add(5, 3); // typescript will automatically know result is a number type
```

We can still explicitly define the type of value returned from the function so it's easy for others to get what we return from a big function

```
const add = (a: number, b: number): number => {
  return a + b;
};
```

- incase we don't return anything from a function type script returns a value called *void* for those functions, although when it's compiled to javascript it becomes undefined
- *void* basically represnets absence of a return statement inside a function in typescript

Type Aliases

If we have complex type definitions at several different place, let's say for multiple function parameters, we can create type aliases for them right at the start so we can use them as many places as we want

```
// usual way
let DOB: string | number;
const greet = (user: { name: string, uId: string | number }) => {
  console.log('hey');
};

// using type aliases
type StringNum = string | number;
type ObjWithName = {
  name: string,
  uId: StringNum,
};

let DOB: StringNum;
const greet = (user: ObjWithName) => {
  console.log('hey');
};
```

Now we can use the type aliases for any other functions or variables we create w/o having to repeat ourselves

Function Signatures

Used to define the structure of a function, i.e., how many parameters, what are their types, what the function returns

```
// func signature
let calc: (a: number, b: number, c: string) => number;
// 3 parameters and should return a number

// func defined
calc = (n1: number, n2: number, action: string) => {
  if (action === 'add') {
    return n1 + n2;
    // we return a number
  } else {
    return n1 - n2;
    // if we don't add this else, func will return void and throw errors
  }
};
```

We can get as specific as we want with these function types and signatures

Example -

```
// alias
type person = {
  name: string,
  age: number,
};

// func signature
let logDetails: (obj: person) => void;

// func declaration
logDetails = (ninja: person) => {
  console.log(`${ninja.name} is ${ninja.age} years old`);
};

const durgesh = {
  name: 'Deekayy',
  age: 20,
};

// function call
logDetails(durgesh);
```

DOM & Type Casting

- If we select an element with `querySelector` using it's tag we might get a little error as there may be a case when the element might not be there and hence the variable will be null
- This can be solved by adding an `!` sign after the selection if we're damn sure that the element will be present in the code `js const form = document.querySelector('form')! // the `!` ensures the form tag is present`
- The type of the `form` variable here would be **HTMLFormElement** only because we selected it using the tag selector
- If we used an ID or a class selector instead it wouldn't be the case. We would have to do type casting then. `js const form = document.querySelector('.input-form') as HTMLFormElement // we don't need to use `!` as we defined the type for the variable and hence it won't be null`

Classes

Access Modifiers

- `readonly` - only allows to read inside & outside a class but not change the value
- `private` - only allows us to read & change inside a class
- `public` - we can change & read value outside or inside a class (default one)

```
class Invoice {
  //----> first way
  // readonly client: string;
  // private details: string;
  // public amount: number;

  // constructor(c: string, d: string, a: number) {
  //   this.client = c;
  //   this.details = d;
  //   this.amount = a;
  // }

  //----> second way (only works if we have access modifier in front)
  constructor(
    readonly client: string,
    private details: string,
    public amount: number
  ) {}

  format() {
    return `${this.client} owes Rs. ${this.amount} for ${this.details}`;
  }
```

Interfaces with Objects

- Used to structure all the different properties and methods on am object of a class
- Unlike class, it isn't used to create objects, just defines the structure

```
interface IsPerson {
  name: string;
  age: number;
  speak(a: string): void;
  spend(a: number): number;
}

// ❌ wrong way
const john: IsPerson = {
  name: 'john',
};

// ✅ correct way
const peter: IsPerson = {
  name: 'peter',
  age: 20,
  speak(text) {
    console.log(text);
  },
  spend(amount) {
    return amount;
  },
};
```

Interfaces with Classes

Same thing as before but just used for classes instead of objects

```
interface HasFormatter {
  format(): string;
}

export class Invoice implements HasFormatter {
  constructor(
    readonly client: string,
    private details: string,
    public amount: number
  ) {}

  format() {
    return `${this.client} owes Rs. ${this.amount} for ${this.details}`;
  }
}
```

- Now the class has to make sure it has a format method as defined in the interface and all it's objects should follow the same structure too
- It just structures our code a bit well and makes it bit error prone for future

Example -

```
import { Invoice } from './classes/Invoice.js';
import { Payment } from './classes/Payment.js';
import { HasFormatter } from './interfaces/HasFormatter.js';

let docOne: HasFormatter;
let docTwo: HasFormatter;

docOne = new Invoice('Mario', 'web design', 200);
docTwo = new Payment('Susan', 'video editing', 50);

// an array that only accepts objects that follow HasFormatter
let docs: HasFormatter[] = [];

docs.push(docOne);
docs.push(docTwo);

console.log(docs);
```

Generics

- Allows to create blocks of code which can be used with different types
- The `<T>` below is a generic (can be any letter) and it will track all the properties that are there on the passed argument (object here)

  ```
  const addUID = <T>(obj: T) => {
  let uid = Math.floor(Math.random() * 100);
  return { ...obj, uid };
  };

  let docOne = addUID({ name: 'john', age: 40 });
  ```

- The above example will work for all things - arrays, objects, strings but we can also get more specific if we want

  ```
  const addUID = <T extends object>(obj: T) => {
  let uid = Math.floor(Math.random() * 100);
  return { ...obj, uid };
  };
  ```

  or

```
const addUID = <T extends {name: string, age: number}>(obj: T) => {
let uid = Math.floor(Math.random() * 100);
return { ...obj, uid };
};
```

Generics with Interfaces

Allows us to have generic types for interfaces properties

```
interface Resource<T> {
  uid: number;
  resourceName: string;
  data: T; // means data can be anything
}

// example 1 (data needs to be an object)
const doc1: Resource<object> = {
  uid: 1,
  resourceName: 'person',
  data: { name: 'shaun' },
};

// example 2 (data needs to be an array of string)
const doc1: Resource<object> = {
  uid: 2,
  resourceName: 'shoppingList',
  data: ['bread', 'milk', 'eggs'],
};
```

Enums

A way to specify descriptive constants and associate each one of them with a numeric value

```
enum ResourceType {BOOK, AUTHOR, FILM, DIRECTOR, PERSON}

interface Resource<T> {
  uid: number;
  resourceName: ResourceType;
  data: T;
}

const doc1: Resource<object> = {
  uid: 1,
  resourceName: ResourceType.PERSON, // behind the scenes it will give a index of 4 & hence this will resource type 5th
  data: { name: 'shaun' },
};
```

Tuples

Just like array but the types of the elements is fixed and it cannot be changed

```
// ARRAYS
let arr = ['john', 20, false];

// type can be modified
arr[0] = false;
arr[1] = 'peter';

// TUPLES
let tup: [string, number, boolean];
tup = ['susan', 23, true];

// type cannot be modified but value can be
tup[0] = false; // 
tup[0] = 'charlie'; // 
```

Conclusion

Thank you for using this cheatsheet!

If you found it helpful please check out more of my work on yodkwtf.com or follow me on twitter. I also run a small youtube channel called Yodkwtf Academy.