Check out those Types!

Typescript Cheat Sheet

Jon Hilton

Typescript quick start

A handy reference to the fundamental building blocks of Typescript.

Basic Types

Boolean

```
let completed: boolean = false;
```

String

```
let name: string = 'Sally';
```

Number

```
let aNumber: number = 20;
```

Array

```
let ages: number[] = [21, 30, 70]

// equivalent but less common in real-world apps
let ages: Array<number> = [21, 30, 70]
```

Enum

```
enum Gender { Male, Female, Unspecified };
let gender: Gender = Gender.Female;

// override the enum values (underlying numbers)
enum Gender { Male = 1, Female = 2, Unspecified = 3 };
```

Any

For interacting with javascript code or situations where the type is simply unknown and you want to avoid compile-time errors...

```
let something: any = 1;
something = "a string";
something = false;

// all work because type is "any"
```

Any for arguments

```
function handleData(age: any){
    console.log(age);
    // age could be anything!
}
```

Any for return values

```
function handleData(age: any) : any {
    if(age !== undefined){
        let message = `you are ${age} years old`;
        return message;
    }
    return false;
    // we know it's a string but we could return anything
}
```

Pro Tip: Try to avoid over-use of any as it somewhat defeats the point (and benefits) of using Typescript in the first place!

Void

Mostly used in function declarations, to indicate that the function does not return a value.

```
function log(message: string): void {
    console.log(message);
}
```

Object

Anything which isn't a primitive (isn't a number, boolean, string etc.);

```
function callApi(){
    let thing:object = backend.get('/api/users');
    thing = 42; // error: can't assign a number

    thing = {name: 'bob'}; // works, we've assigned an object
}
```

Template literals (aka Template strings)

A handy way to concatenate strings without the usual ugly syntax.

```
// without Template strings
let greeting = 'Hello ' + name + ', how are you today?';

// with Template strings
let greeting = `Hello ${name}, how are you today?`;
```

The difference is that subtle *backtick* used instead of an apostrophe.

Interfaces

```
interface IUser {
    firstName: string;
    lastName: string;
}
```

Interfaces make it easy to declare the "shape" of an object.

Objects defined using the interface will trigger compile errors if you a) don't specify values for required fields or b) try to assign a value to a field which isn't defined.

```
// doesn't compile because firstName, LastName missing
let me: IUser = {};

// works
let me: IUser = { firstName: 'Jon', lastName: 'Hilton' };

// doesn't compile because IUser has no an 'age' property
me.age = 21;
```

Required vs optional properties

```
interface IUser {
    firstName: string;
    lastName: string;
    age?: number;
}
```

Here `age` is optional. Everything else is required.

```
// doesn't compile because firstName, LastName missing
let me: IUser = {};

// works, age is optional
let me: IUser = { firstName: 'Jon', lastName: 'Hilton' };

// works because IUser does specify an age property
me.age = 21;

// fails compilation (mistyped name of age property)
me.aeg = 21;
```

Union Types

Sometimes, an object can be one of several types.

You can let the compiler know that multiple types are valid by declaring a union type.

```
let ambiguous: string | number;
ambiguous = 'Not any more';
ambiguous = 21;
```

Functions

For the most part functions in TS work just as functions in Javascript do with the obvious difference being that you can specify argument and return types.

Here's a named function with it's argument and return types specified.

```
function subtract(x: number, y: number): number {
    return x - y;
}
```

And here's an anonymous equivalent.

```
let subtract = function (x: number, y: number): number {
    return x - y;
}
```

Note that we haven't explicitly defined the type of `subtract` here. The Typescript compiler has inferred it from the type of our function.

This is the same as...

```
let subtract: (x:number, y:number) => number;
```

This declares a variable (`subtract`) and indicates that it's type is a function which takes two `number` arguments and returns a `number` ...

From here we can then assign our anonymous function to `subtract` .

```
subtract = function (x: number, y: number): number {
    return x - y;
}
```

Putting it all together gives us...

```
let subtract: (x:number, y:number) => number
    = function (x: number, y: number): number {
    return x - y;
}

let result = subtract(10, 1);
alert(result);
```

Optional arguments

With default values

```
let add = function(x: number, y:number, z:number = 0): number {
    return x + y + z;
}

add(1,2); // 3
add(10,10,10); //30
```

If you specify a default value for an argument then don't pass in a value for it when you call the function, the default will be used.

Without default values

```
let greet = function(firstName: string, lastName?: string): string {
    if(lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
```

Here no default will be used, if you don't pass the optional parameter in, the function needs to check whether it has a value and act accordingly.

Rest Parameters

If you've used C# you may have come across the `params` keyword which lets you declare that any number of arguments can be passed to a function and added to an array which holds each of the passed values.

Typescript's equivalent is Rest parameters.

```
function greetTheRoom(...names: string[]){
    console.log('hello ' + names.join(", "));
}

// 'hello Jon, Nick, Jane, Sam'
greetTheRoom('Jon', 'Nick', 'Jane', 'Sam');
```

When you call a function with a Rest parameter, you can specify as many values as you like (or none).

Arrow functions

If you've used C# you've probably used Lambdas. Javascript/Typescript has them too...

```
let sayHello = () => console.log('hello');

// is equivalent to...

var sayHello = function () {
    return console.log('hello');
};
```

And with arguments...

```
let sayHello = (name:string) => console.log(`hello ${name}`);

// is the same as...

var sayHello = function (name) {
    return console.log("hello " + name);
};
```

Classes

Javascript is very much a functional language but in recent times has introduced the concept of "classes" as a way to structure your code and objects.

Typescript supports this too...

```
class Calculator {
    private result: number;

    constructor(){
        this.result = 0;
    }

    add(x:number) {
        this.result += x;
    }

    subtract(x:number) {
        this.result -= x;
    }

    getResult():number {
        return this.result;
    }
}

// use it
let calculator = new Calculator();
calculator.add(10);
calculator.subtract(9);
console.log(calculator.getResult()); // 1
```

Modifiers

Everything in a Typescript class defaults to public.

Private modifier

You can mark class members as private and this will prevent it being accessed outside of the class.

```
class Person {
    private passportNumber: string;

    constructor(passportNumber: string){
        this.passportNumber = passportNumber;
    }
}

let person = new Person("13252323223");

// error: passportNumber is private
console.log(person.passportNumber);
```

Readonly modifier

Readonly members must be initialized when they're declared, or in the constructor

```
class Person {
    private readonly passportNumber: string;

    constructor(passportNumber: string){
        this.passportNumber = passportNumber;
    }

    changePassport(newNumber: number){
        // error: cannot assign to passportNumber because readonly
        this.passportNumber = newNumber;
    }
}
```

Parameter properties

If you find yourself declaring fields, passing values in via constructors and assigning those values to the fields, you can use a handy shorthand to save writing so much boilerplate code.

```
class Person {
    constructor(private readonly passportNumber: string){
    }
}
```

This is exactly the same as...

```
class Person {
    private readonly passportNumber: string;

    constructor(passportNumber: string){
        this.passportNumber = passportNumber;
    }
}
```

When we add accessibility modifiers (`private` , `public` , `readonly` etc.) to our constructor parameters, Typescript automatically declares a class member and assigns the parameters' value to it.

Static properties

You can also declare static members on a class.

```
class Salutations {
    static Mr(name: string){
        return "Mr " + name;
    }
    static Mrs(name: string){
        return "Mrs " + name;
    }
}

// hello Mr Brian Jones
console.log ('hello ' + Salutations.Mr('Brian Jones'));

// greetings Mrs Joy Smith
console.log ('greetings ' + Salutations.Mrs('Joy Smith'));
```

Here we access the static members via the class name (`Salutations`) without having to create an instance of the class.

Where next?

That covers some essential Typescript syntax.

It's worth remembering that Typescript adheres pretty closely to the latest Javascript specs.

This is good if you're just getting started because your knowledge of Typescript will generally transfer to Javascript too.

Typescript layers some Type safety on top of Javascript and handles the compilation back to a version of Javascript which your browser can understand (see relevant sections in "front-end in four hours" for more details).

So now it's over to you. Crack open your favourite text editor and get hacking :-)