Programster's Blog

Tutorials focusing on Linux, programming, and open-source

Typescript Cheatsheet

TypeScript

Below is my cheatsheet for programming in Typescript. A large amount of the content here comes form Introduction to TypeScript which I recommend you watch if you are just getting started.

Basic Types

- Number
- String
- Boolean
- Array
- Tuple
- Enum
- Any
- Void
- Null
- Undefined
- Never

Creating Variable With Types

```
var x: number = 3;
var y: string = "hello world";

// Old fashioned javascript without types still works
var y = "hello";
```

Copy to clipboard

Functions with Types

```
function hello(name: string)
{
    console.log("hello " + name);
}

hello("John");
hello(5); // Will raise an error
```
Copy to clipboard

Enums

Enums are very handy when you know there is a set range of values. For example my `environment` attribute of my config, should always be `dev`, `staging`, or `production`

```
enum Environment
{
    Dev = "dev",
    Staging = "staging",
    Production = "production"
}
```
Copy to clipboard

Interfaces

Interfaces can be useful in the traditional way for classes to implement them and then be used in dependency injection etc.

```
interface AnimalInterface
{
    name: string;
    sayName(): void
}
```
Copy to clipboard

However, they are also brilliantly useful for telling your IDE how a JSON variable is made up. For example, here I am telling my IDE about my `config` global variable that was dumped by the PHP backend:

```
enum Environment
{
    Dev = "dev",
    Staging = "staging",
    Live = "live"
}

interface Config
{
    environment: Environment;
    setting1Enabled : boolean;
    setting2Enabled : boolean;
    redirectUrl: string;
}

// Global js variables declared in php world
declare var config: Config;
```
Copy to clipboard

Below is how I might pass the config to the Typescript/Javascript from the server.

```
<?php
$config = array(
    "environment" => "dev",
    "setting1Enabled" => true,
    "setting2Enabled" => false,
);
?>

<script>
var config = "<?= json_encode($config, JSON_UNESCAPED_SLASHES); ?>";
</script>
```
Copy to clipboard

Optional Interface Attributes/Functions

Typescript has a concept of optional attributes/functions. These are denoted with a `?` at the end like so:

```
interface MyInterface
{
    mandatoryAttribute: string;
    optionalAttribute?: number;
    optionalFunction?: Function;
}
```
Copy to clipboard

Classes

Creating A Basic Class

Below is a very basic class for creating a dog with a name.

```
class Dog
{
    private name: string;

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

    sayName() {
        console.log("My name is " + this.name );
    }
}

var myDog: Dog = new Dog("Rover");
myDog.sayName();
```
Copy to clipboard

Note that `constructor` is a keyword that allows you to use the `new` functionality later to create an instance of the class.

Using An Interface

Carrying on from the previous example, we can use an interface for if we wanted to create other animal classes such as `Cat` later.

```
interface AnimalInterface
{
    name: string;
    sayName(): void
}

class Dog implements AnimalInterface
```

```
{
    name: string = '[Animal]';

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

    sayName() {
        console.log("My name is " + this.name );
    }
}

var myDog: Dog = new Dog("Rover");
myDog.sayName();
```
Copy to clipboard

Notice how an interface can force a class that uses the interface to have specified member variables, not just methods. However, I was unable to use private in this example without errors being raised in Netbeans. I am not sure if this is a bug with the plugin I am using.

Accessors (Getters and Setters)

If you configure Typescript to be targeting ECMAScript 5+ then you can use the accessors as demonstrated below:

```
class Dog
{
    _name: string = '[Animal]';

    get name(): string { console.log("Using name accessor"); return this._n
    set name(name: string) { console.log("Using name setter"); this._name =

    constructor(name: string) {
        this._name = name;
    }

    sayName() {
        console.log("My name is " + this._name );
    }
}

var myDog: Dog = new Dog("Rover");
```

```
myDog.name = "Sally";
console.log("My dogs name is " + myDog.name);
```
Copy to clipboard

If you execute the compiled javascript for the script above, you will get the following output:

```
Using name setter
Using name accessor
My dogs name is Sally
```
Copy to clipboard

As you can see, even though we have no public `name` variable, since we have a `name` setter and getter, we can `name` like a property of the class as shown by these lines:

```
myDog.name = "Sally";
console.log("My dogs name is " + myDog.name);
```
Copy to clipboard

These will use the get and set functions accordingly, which may perform additional functionality other than just setting a member variable. This could be something like performing validation of the input data.

Extending Classes / Creating Child Classes

In the example below we create a `Dog` class that extends the `Animal` class. The dog class simply has one more method, which is `bark` but still has the methods of the parent class.

```
class Animal
{
    _name: string;

    constructor(name: string) {
        this._name = name;
    }

    sayName() {
        console.log("My name is " + this._name );
    }
}

class Dog extends Animal
{
    bark()
    {
        console.log("Woof");
    }
}

var myDog: Dog = new Dog("Rover");
myDog.sayName();
myDog.bark();
```

Copy to clipboard

References

- Typescriptlang.org - Basic Types