TYPESCRIPT TYPE SYSTEM

The type system in typescript is designed to be **optional** so that your javascript is typescript.

TypeScript does not block JavaScript emit in the presence of Type Errors, allowing you to progressively update your JS to TS.

Now let's start with the syntax of the TypeScript type system. This way you can start using these annotations in your code immediately and see the benefit. This will prepare you for a deeper dive later.

BASIC ANNOTATIONS

As mentioned before Types are annotated using `:TypeAnnotation` syntax. Anything that is available in the type declaration space can be used as a Type Annotation.

The following example demonstrates type annotations can be used for variables, function parameters and function return values:

```
var num: number = 123;
function identity(num: number): number {
    return num;
}
```

PRIMITIVE TYPES

The JavaScript primitive types are well represented in the TypeScript type system. This means `string`, `number`, `boolean` as demonstrated below:

```
var num: number;
var str: string;
var bool: boolean;

num = 123;
num = 123.456;
num = '123'; // Error

str = '123';
str = 123; // Error

bool = true;
bool = false;
bool = 'false'; // Error
```

ARRAYS

TypeScript provides dedicated type syntax for arrays to make it easier for you to annotate and document your code. The syntax is basically postfixing `[]` to any valid type annotation (e.g. `:boolean[]`). It allows you to safely do any array manipulation that you would normally do and protects you from errors like assigning a member of the wrong type. This is demonstrated below:

```
var boolArray: boolean[];

boolArray = [true, false];
console.log(boolArray[0]); // true
console.log(boolArray.length); // 2
boolArray[1] = true;
boolArray = [false, false];

boolArray[0] = 'false'; // Error!
boolArray = 'false'; // Error!
boolArray = [true, 'false']; // Error!
```

INTERFACES

Interfaces are the core way in TypeScript to compose multiple type annotations into a single named annotation. Consider the following example:

```
interface Name {
    first: string;
    second: string;
}

var name: Name;
name = {
    first: 'John',
    second: 'Doe'
};

name = {          // Error : `second` is missing
    first: 'John'
};
name = {          // Error : `second` is the wrong type
    first: 'John',
    second: 1337
};
```

Here we've composed the annotations `first: string` + `second: string` into a new annotation `Name` that enforces the type checks on individual members. Interfaces have a lot of power in TypeScript and we will dedicate an entire section to how you can use that to your advantage.

INLINE TYPE ANNOTATION

Instead of creating a new `interface` you can annotate anything you want INLINE using : `{ /*Structure*/ }`. The previous example presented again with an inline type:

```
var name: {
    first: string;
```

```
    second: string;
};
name = {
    first: 'John',
    second: 'Doe'
};

name = {          // Error : `second` is missing
    first: 'John'
};
name = {          // Error : `second` is the wrong type
    first: 'John',
    second: 1337
};
```

Inline types are great for quickly providing a one off type annotation for something. It saves you the hassle of coming up with (a potentially bad) type name. However, if you find yourself putting in the same type annotation inline multiple times it's a good idea to consider refactoring it into an interface (or a `type alias` covered later in this section).

SPECIAL TYPES

Beyond the primitive types that have been covered there are few types that have special meaning in TypeScript. These are `any`, `null`, `undefined`, `void`.

ANY

The any type holds a special place in the TypeScript type system. It gives you an escape hatch from the type system to tell the compiler to bugger off. any is compatible with any and all types in the type system. This means that anything can be assigned to it and it can be assigned to anything. This is demonstrated in the example below:

```
var power: any;

// Takes any and all types
power = '123';
power = 123;

// Is compatible with all types
var num: number;
power = num;
num = power;
```

If you are porting JavaScript code to TypeScript, you are going to be close friends with any in the beginning. However, don't take this friendship too seriously as it means that it is up to you to ensure the type safety. You are basically telling the compiler to not do any meaningful static analysis.

NULL AND UNDEFINED

The `null` and `undefined` JavaScript literals are effectively treated by the type system the same as something of type `any`. These literals can be assigned to any other type. This is demonstrated in the below example:

```
var num: number;
var str: string;

// These literals can be assigned to anything
num = null;
str = undefined;
```

:VOID

Use `:void` to signify that a function does not have a return type:

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

GENERICS

Many algorithms and data structures in computer science do not depend on the actual type of the object. However you still want to enforce a constraint between various variables. A simple toy example is a function that takes a list of items and returns a reversed list of items. The constraint here is between what is passed in to the function and what is returned by the function:

```
function reverse<T>(items: T[]): T[] {
    var toreturn = [];
    for (let i = items.length - 1; i >= 0; i--) {
        toreturn.push(items[i]);
    }
    return toreturn;
}

var sample = [1, 2, 3];
var reversed = reverse(sample);
console.log(reversed); // 3,2,1

// Safety!
reversed[0] = '1';     // Error!
reversed = ['1', '2']; // Error!

reversed[0] = 1;       // Okay
reversed = [1, 2];     // Okay
```

Here you are basically saying that the function reverse takes an array (items: T[]) of some type T (notice the type parameter in reverse<T>) and returns an array of type T (notice : T[]). Because the reverse function returns items of the same type as it takes, TypeScript knows the reversed variable is also of type number[] and

will give you Type safety. Similarly if you pass in an array of string[] to the reverse function the returned result is also an array of string[] and you get similar type safety as shown below:

```
var strArr = ['1', '2'];
var reversedStrs = reverse(strArr);

reversedStrs = [1, 2]; // Error!
```

In fact JavaScript arrays already have a `.reverse` function and TypeScript does indeed use generics to define its structure:

```
interface Array<T> {
 reverse(): T[];
 // ...
}
```

This means that you get type safety when calling `.reverse` on any array as shown below:

```
var numArr = [1, 2];
var reversedNums = numArr.reverse();

reversedNums = ['1', '2']; // Error!
```

UNION TYPE

Quite commonly in JavaScript you want to allow a property to be one of multiple types e.g. a string or a number. This is where the union type (denoted by | in a type annotation e.g. string|number) comes in handy. A common use case is a function that can take a single object or an array of the object e.g.:

```
function formatCommandline(command: string[]|string) {
    var line = '';
    if (typeof command === 'string') {
        line = command.trim();
    } else {
        line = command.join(' ').trim();
    }

    // Do stuff with line: string
}
```

INTERSECTION TYPE

`extend` is a very common pattern in JavaScript where you take two objects and create a new one that has the features of both these objects. An **Intersection Type** allows you to use this pattern in a safe way as demonstrated below:

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U> {};
    for (let id in first) {
        result[id] = first[id];
```

```
        }
        for (let id in second) {
            if (!result.hasOwnProperty(id)) {
                result[id] = second[id];
            }
        }
        return result;
    }

    var x = extend({ a: "hello" }, { b: 42 });

    // x now has both `a` and `b`
    var a = x.a;
    var b = x.b;
```

TUPLE TYPE

JavaScript doesn't have first class tuple support. People generally just use an array as a tuple. This is exactly what the TypeScript type system supports. Tuples can be annotated using :[typeofmember1, typeofmember2] etc. A tuple can have any number of members. Tuples are demonstrated in the below example:

```
var nameNumber: [string, number];

// Okay
nameNumber = ['Jenny', 8675309];

// Error!
nameNumber = ['Jenny', '867-5309'];
```

Combine this with the destructuring support in TypeScript, tuples feel fairly first class despite being arrays underneath:

```
var nameNumber: [string, number];
nameNumber = ['Jenny', 8675309];

var [name, num] = nameNumber;
```

TYPE ALIAS

TypeScript provides convenient syntax for providing names for type annotations that you would like to use in more than one place. The aliases are created using the `type SomeName = someValidTypeAnnotation` syntax. An example is demonstrated below:

```
type StrOrNum = string|number;

// Usage: just like any other notation
var sample: StrOrNum;
sample = 123;
```

```
sample = '123';

// Just checking
sample = true; // Error!
```

Unlike an `interface` you can give a type alias to literally any type annotation (useful for stuff like union and intersection types). Here are a few more examples to make you familiar with the syntax:

```
type Text = string | { text: string };
type Coordinates = [number, number];
type Callback = (data: string) => void;
```

TIP: If you need to have hierarchies of Type annotations use an `interface`. They can be used with `implements` and `extends`

TIP: Use a type alias for simpler object structures (like `Coordinates`) just to give them a semantic name. Also when you want to give semantic names to Union or Intersection types, a Type alias is the way to go.

SUMMARY

Now that you can start annotating most of your JavaScript code we can jump into the nitty gritty details of all the power available in the TypeScript's Type System.