TypeScript

TypeScript is JavaScript with added syntax for types.

TypeScript is a syntactic superset of JavaScript which adds static typing.

Why TypeScript?

JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.

In JavaScript, function parameters and variables don't have any information! So developers need to look at documentation, or guess based on the implementation.

TypeScript allows specifying the types of data being passed around within the code and has the ability to report errors when the types don't match.

For example, TypeScript will report an error when passing a string into a function that expects a number. JavaScript will not.

TypeScript uses compile time type checking. Which means it checks if the specified types match before running the code, not while running the code.

TypeScript Compiler: -

TypeScript is transpiled into JavaScript using a compiler.

Within your npm project, run the following command to install the compiler.

npm install typescript –save -dev

npx tsc gives version of ts

Configuring the compiler: -

The compiler can be configured using a tsconfig.json file.

npx tsc –init // create tsconfig.json

This is one way to quickly get started with TypeScript.

TypeScript Simple Types: -

There are three main primitives in JavaScript and TypeScript.

boolean - true or false values.

number - whole numbers and floating point values.

string - text values like "TypeScript Rocks"

There are also 2 less common primitives used in later versions of Javascript and TypeScript.

bigint - whole numbers and floating point values but allows larger negative and positive numbers than the number type.

symbol are used to create a globally unique identifier.

Type Assignment: -

When creating a variable, there are two main ways TypeScript assigns a type. Explicit and Implicit.

Explicit Type: -

writing out the type.

let firstName: string = "Dylan"; // typeof firstName gives string.

Explicit type assignment are easier to read and more intentional.

Implicit Type: -

TypeScript will "guess" the type, based on the assigned value.

let firstName = "Dylan"; //it will guess string

Implicit assignment forces TypeScript to infer the value. implicit type assignment are shorter, faster to type, and often used when developing and testing.

Error In Type Assignment: -

TypeScript will throw an error if data types do not match.

let firstName: string = "Dylan"; // type string
firstName = 33; // attempts to re-assign the value to a different type.

prog.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.

Implicit type assignment would have made firstName less noticeable as a string, but both will throw an error.

let firstName = "Dylan"; // inferred to type string

firstName = 33; // attempts to re-assign the value to a different type.

prog.ts(2,1): error TS2322: Type 'number' is not assignable to type 'string'.

Unable to Infer: -

TypeScript may not always properly infer what the type of a variable may be. In such cases, it will set the type to any which disables type checking.

// Implicit any as JSON.parse doesn't know what type of data it returns so it can be "any" thing...
const json = JSON.parse("55");
// Most expect json to be an object, but it can be a string or a number like this example
console.log(typeof json); //number

This behavior can be disabled by enabling noImplicitAny as an option in a TypeScript's project tsconfig.json. That is a JSON config file for customizing how some of TypeScript behaves.

TypeScript Special Types: -

Type: any

any is a type that disables type checking and effectively allows all types to be used.

Example without any: -

```
let u = true;
u = "string"; // Error: Type 'string' is not assignable to type 'boolean'.
Math.round(u); // Error: Argument of type 'boolean' is not assignable to parameter of type 'number'.
```

Example with any: -

Setting any to the special type any disables type checking.

```
let v: any = true;
v = "string"; // no error as it can be "any" type
Math.round(v); // no error as it can be "any" type
```

any can be a useful way to get past errors since it disables type checking, but TypeScript will not be able provide type safety, and tools which rely on type data, such as auto completion, will not work.

Type: unknown

unknown is a similar, but safer alternative to any.

TypeScript will prevent unknown types from being used, as shown in the below example.

```
let w: unknown = 1;
w = "string"; // no error
w = {
  runANonExistentMethod: () => {
    console.log("I think therefore I am");
  }
} as { runANonExistentMethod: () => void}
// w.runANonExistentMethod(); // Error: Object is of type 'unknown'.
if(typeof w === 'object' && w !== null) {
  (w as { runANonExistentMethod: Function }).runANonExistentMethod();
}
// Although we have to cast multiple times we can do a check in the if to secure our type and have a safer casting.
```

unknown is best used when you don't know the type of data being typed. To add a type later, you'll need to cast it.

Casting is when we use the "as" keyword to say property or variable is of the casted type.

Type: never

never effectively throws an error whenever it is defined.

```
let x: never = true; // Error: Type 'boolean' is not assignable to type 'never'.
```

never is rarely used, especially by itself, its primary use is in advanced generics.

Type: undefined & null

undefined and null are types that refer to the JavaScript primitives undefined and null respectively.

```
let y: undefined = undefined; // undefined type
let z: null = null; // object type
```

These types don't have much use unless strictNullChecks is enabled in the tsconfig.json file.

TypeScript Arrays: -

```
const names: string[] = [];
names.push("Dylan"); // no error
// names.push(3); // Error: Argument of type 'number' is not assignable to parameter of type 'string'.
```

Readonly: -

The readonly keyword can prevent arrays from being changed.

```
const names: readonly string[] = ["Dylan"];
names.push("Jack"); // Error: Property 'push' does not exist on type 'readonly string[]'.
```

Type Inference: -

TypeScript can infer the type of an array if it has values.

```
const numbers = [1, 2, 3]; // inferred to type number[]
numbers.push(4); // no error
// comment line below out to see the successful assignment
numbers.push("2"); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.
let head: number = numbers[0]; // no error
```

TypeScript Tuples: -

Typed Arrays

A tuple is a typed array with a pre-defined length and types for each index.

Tuples are great because they allow each element in the array to be a known type of value.

To define a tuple, specify the type of each element in the array.

```
// define our tuple

let ourTuple: [number, boolean, string];

// initialize correctly

ourTuple = [5, false, 'Coding God was here'];
```

order matters in our tuple. if we try to set them in the wrong order, it will throw an error.

// We have no type safety in our tuple for indexes 3+

A good practice is to make your tuple readonly.

```
// define our readonly tuple
const ourReadonlyTuple: readonly [number, boolean, string] = [5, true, 'The Real Coding God'];
// throws error as it is readonly.
ourReadonlyTuple.push('Coding God took a day off');
```

Named Tuples: -

Named tuples allow us to provide context for our values at each index.

```
const graph: [x: number, y: number] = [55.2, 41.3];
```

Destructuring Tuples: -

Since tuples are arrays we can also destructure them.

```
const graph: [number, number] = [55.2, 41.3];
const [x, y] = graph;
```

TypeScript Object Types: -

```
const car: { type: string, model: string, year: number } = {
  type: "Toyota",
  model: "Corolla",
  year: 2009
};
```

Type Inference

TypeScript can infer the types of properties based on their values.

```
const car = {
  type: "Toyota",
};
car.type = "Ford"; // no error
car.type = 2; // Error: Type 'number' is not assignable to type 'string'.
```

Optional Properties: -

Optional properties are properties that don't have to be defined in the object definition.

Example without an optional property

```
const car: { type: string, mileage: number } = { // Error: Property 'mileage' is missing in type '{ type: string; }' but required in type '{ type: string; mileage: number; }'.
  type: "Toyota",
};
car.mileage = 2000;
```

Example with an optional property

```
const car: { type: string, mileage?: number } = { // no error
  type: "Toyota"
};
car.mileage = 2000;
```

Index Signatures: -

Index signatures can be used for objects without a defined list of properties.

```
const nameAgeMap: { [index: string]: number } = {};
nameAgeMap.Jack = 25; // no error
nameAgeMap.Mark = "Fifty"; // Error: Type 'string' is not assignable to type 'number'.
```

TypeScript Enums: -

An enum is a special "class" that represents a group of constants (unchangeable variables).

Enums come in two flavors string and numeric.

Numeric Enums - Default

By default, enums will initialize the first value to 0 and add 1 to each additional value.

```
enum CardinalDirections {
  North,
  East,
  South,
  West
}
let currentDirection = CardinalDirections.North;
// logs 0
console.log(currentDirection);
// throws error as 'North' is not a valid enum
currentDirection = 'North'; // Error: "North" is not assignable to type 'CardinalDirections'.
```

Numeric Enums - Initialized

You can set the value of the first numeric enum and have it auto incremented from that.

```
enum CardinalDirections {
  North = 1,
  East,
  South,
  West
}
console.log(CardinalDirections.North); // logs 1
console.log(CardinalDirections.West); // logs 4
```

Numeric Enums - Fully Initialized

You can assign unique number values for each enum value. Then the values will not incremented automatically.

```
enum StatusCodes {
  NotFound = 404,
  Success = 200,
  Accepted = 202,
  BadRequest = 400
}
console.log(StatusCodes.NotFound); // logs 404
console.log(StatusCodes.Success); // logs 200
```

String Enums: -

Enums can also contain strings. This is more common than numeric enums, because of their readability and intent.

```
enum CardinalDirections {
  North = 'North',
  East = "East",
  South = "South",
  West = "West"
};
console.log(CardinalDirections.North); // logs "North"
console.log(CardinalDirections.West); // logs "West"
```

TypeScript Type Aliases and Interfaces: -

TypeScript allows types to be defined separately from the variables that use them.

Aliases and Interfaces allows types to be easily shared between different variables/objects.

Type Aliases: -

Type Aliases allow defining types with a custom name (an Alias).

Type Aliases can be used for primitives like string or more complex types such as objects and arrays.

```
type CarYear = number

type CarType = string

type CarModel = string

type Car = {

  year: CarYear,

  type: CarType,

  model: CarModel

}

const carYear: CarYear = 2001

const carType: CarType = "Toyota"
```

```
const carModel: CarModel = "Corolla"
const car: Car = {
  year: carYear,
  type: carType,
  model: carModel
};
```

Interfaces: -

Interfaces are similar to type aliases, except they only apply to object types.

```
interface Rectangle {
  height: number,
  width: number
}
const rectangle: Rectangle = {
  height: 20,
  width: 10
};
```

Extending Interfaces: -

Extending an interface means you are creating a new interface with the same properties as the original, plus something new.

```
interface Rectangle {
  height: number,
  width: number
}
interface ColoredRectangle extends Rectangle {
  color: string
}
const coloredRectangle: ColoredRectangle = {
  height: 20,
  width: 10,
  color: "red"
};
```

TypeScript Union Types: -

Union types are used when a value can be more than a single type.

Union | (OR)

```
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code}.`)
}
printStatusCode(404); // My status code is 404.
printStatusCode('404'); // My status code is 404.
```

Union Type Errors

you need to know what your type is when union types are being used to avoid type errors.

```
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code.toUpperCase()}.`) // error: Property 'toUpperCase'
//does not exist ontype 'string | number'.
 // Property 'toUpperCase' does not exist on type 'number'
}
```

TypeScript Functions: -

TypeScript has a specific syntax for typing function parameters and return values.

Return Type

The type of the value returned by the function can be explicitly defined.

```
// the `: number` here specifies that this function returns a number
function getTime(): number {
  return new Date().getTime();
}
```

If no return type is defined, TypeScript will attempt to infer it through the types of the variables or expressions returned.

The type void can be used to indicate a function doesn't return any value.

Parameters: -

```
function multiply(a: number, b: number) {
  return a * b;
}
```

Optional Parameters

By default TypeScript will assume all parameters are required, but they can be explicitly marked as optional.

```
// the `?` operator here marks parameter `c` as optional
function add(a: number, b: number, c?: number) {
  return a + b + (c || 0);
}
```

Default Parameters

For parameters with default values, the default value goes after the type annotation.

```
function pow(value: number, exponent: number = 10) {
  return value ** exponent;
}
```

TypeScript can also infer the type from the default value.

Named Parameters

```
function divide({ dividend, divisor }: { dividend: number, divisor: number }) {
  return dividend / divisor;
}
```

Rest Parameters

the type must be an array as rest parameters are always arrays.

```
function add(a: number, b: number, ...rest: number[]) {
  return a + b + rest.reduce((p, c) => p + c, 0);
}
```

Type Alias

Function types can be specified separately from functions with type aliases.

```
type Negate = (value: number) => number;

// in this function, the parameter `value` automatically gets assigned the type `number` from the type `Negate`

const negateFunction: Negate = (value) => value * -1;
```

If no parameter type is defined, TypeScript will default to using any, unless additional type information is available for parameters and Type Alias.

TypeScript Casting: -

Casting is the process of overriding a type.

Casting with as: -

A straightforward way to cast a variable is using the as keyword, which will directly change the type of the given variable.

```
let x: unknown = 'hello';
console.log((x as string).length); //5
```

casting doesn't actually change the type of the data within the variable, for example the following code will not work as expected since the variable x is still holds a number.

```
let x: unknown = 4;
console.log((x as string).length); // prints undefined since numbers don't have a length.
```

Casting with <>: -

```
let x: unknown = 'hello';
console.log((<string>x).length); // 5
```

This type of casting will not work with TSX, such as when working on React files.

Force casting: -

To override type errors that TypeScript may throw when casting, first cast to unknown, then to the target type.

```
let x = 'hello';
console.log(((x as unknown) as number).length);
```

Property 'length' does not exist on type 'number' // 5