# TypeScript

Matt Waismann

October 26th, 2021

## 1 Introduction

TypeScript stands in an unusual relationship to JavaScript. TypeScript offers all of JavaScript's features, and an additional layer on top: TypeScript's type system.

For example, JavaScript provides language primitives like *string* and *number*, but it doesn't check that you've consistently assigned these. TypeScript does.

## 2 Types by Inference

TypeScript knows the JavaScript language and will generate types for you in many cases. For example, in creating a variable and assigning it to a paritcular value, TypeScript will use the value as its type.

```
let helloWorld = "Hello World";
console.log(typeof helloWorld);
//[LOG]: "string"
```

## 3 let, var, and const variable declarations

What is *let*? In JavaScript, there's several ways to declare variables. Those desclared by the *var* keyword are scoped to the immediate function body(hence the function scope) while *let* variables are scoped to the immediate enclosing clock denoted by {} . The reason why *let* was introduced to the language was because function scope is confusing and was one of the main sources of bugs in JavaScript.

At the top level, *let*, unlike *var*, does not create a property on the global object:

```
var foo = "Foo"; //globally scoped
let bar = "Bar"; // not allowed to be globally scoped

console.log(window.foo); // Foo
console.log(window.bar); // undefined
```

Another motivation for let statements is to free up memory when not needed in a ceratin block. It's encouraged that variables are existent only where they are needed.

Variables can be declared using *const* similar to var or let declarations. The const makes a variable a constant where its value cannot be changed. Const variables have the same scoping rules as *let* variables (block scoped). Const declares a variable that is immutable. The main argument for using *const* is for the readability of your code. It signals to the reader that you're not going to assign to the variable. This is also in line with the "principle of least privilege'

# 4 Defining Types

To create an object with an inferred type which includes *name: string* and *id: number*, you can write:

```
const user = {
  name: "Hayes",
  id: 0,
};
```

You can explicity describe this object's shape (yes shape) using an *interface* declaration:

```
interface User {
  name: string;
  id: number;
}
```

You can then declare that a JavaScript object conforms to the shape your new interface by using syntax like *: TypeName* after a variable declaration:

```
const user: User = {
  name: "Hayes",
  id: 0,
};
```

If you provide an object that doesn't match the interface you have provided, TypeScript will warn you.

Since JavaScript supports classes and object-oriented programming, so does TypeScript. You can use an interface declaration with classes:

```
interface User {
  name: string;
  id: number;
}

class UserAccount{
  name: string;
  id: number:

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

const user: User = new UserAccount("Murphy", 1);
```

You can use interfaces to annotate parameters and return values to functions:

```
function getAdminUser(): User {
  // ...
}

function deleteUser(user: User) {
  // ...
}
```

There is already a small set of primitive types available in JavaScript: *boolean*, *bigint*, *null*, *number*, *string*, *symbol*, and *undefined*, which you can use in an interface. TypeScript extends this list with a few more, such as *any* (allow anything), *unkown* (ensure someone using this type declares what the type is), *never* (it's not possible that this type could happen), and *void* (a function which returns *undefined* or has no return value).

The == operator in JavaScript (and hence TypeScript) can do unexpected type conversions. In Javascript *1==1* is *true*. The === operator avoids this. Comparing different types with === is always false.

# 5 Primative Types

In Java, primative types are the most basic data types available within the Java language. There are 8: **boolean, byte, char, short, int, long, float, and double**.These types serve as the building blocks of data manipulation in Java.

Such types serve only one purpose - containing pure, simple values of a kind.

In Python, there are only four primitive data types **Integer, Float, String, and Boolean**.

In JavaScript, there are 7 primitive data types: **string, number, bigint, boolean, undefined, symbol, and null**.

# 6 Enums

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript.

Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

```
enum Direction {
  Up = 1,
  Down,
  Left
  Right,
}
```

Above, we have a numeric enum where *Up* is initialized with *1*. All of the other following members are auto-incremented from that point on. In other words, *Direction.Up* has the value *1*, *Down* has *2*, *left* has *3*, and so on. In fact, if we didn't even initialize anything then *Up* would start with 0, *Down* with *1* and so on. This auto-incrementing behavior is useful for cases where we might not care about the member values themselves, but do care that each value is distinct from other values in the same enum.

Using an enum is simple: just access any member as a property off of the enum itself, and declare types using the name of the enum:

```
enum UserResponse {
  No = 0,
  Yes = 1,
}

function respond(recipient: string, message: UserResponse): void {
  // ..
}

respond("Princess Caroline", UserResponse.Yes);
```

String enums are a similar concept but don't have auto-incrementing behavior

```
enum Direction{
  Up = "UP",
  Down = "DOWN",
  left = "LEFT",
  right = "RIGHT"
}
```

Each enum member has a value associated with it which can be either constant or computed.

# 7 Objects vs Enums

In modern TypeScript, you may not need an enum when an object with *as const* could suffice:

```
const enum EDirection{
  Up,
  Down,
  Left,
  Right,
}

const ODirection = {
  Up: 0,
  Down: 1,
  Left: 2,
  Right: 3,
} as const;
```

The biggest argument against using enums is that since JavaScript does not contain enums, so your codebase cannot be aligned with the state of JavaScript.