

# Basics in TypeScript

# Topics

- Basics
- Objects
- Enums
- Arrays
- Functions
- Classes and Interfaces
- Generics
- Basic Types
- Type Operators
- Working with Types
- Debugging

# Basics

- Install with `npm i -g typescript`
- Run with `tsc file-name`
- tsconfig.json: -outDir,

# Basics: JS vs TS

- larger than js
- static typing
- code completion

# Basics: Type safety

- Type safety: Using types to prevent programs from doing invalid things
- Statically Typed: I made a mistake when I compile the program
- Dynamically Typed: I made a mistake when I RUN the program
- TypeScript is Statically Typed differently from JavaScript that is Dynamically Typed

# Basics: Compiler and Compiling Process

- TypeScript Compiler (TSC)
- How compilation works: text (code) -> compiler -> abstract syntax tree (AST) -> bytecode -> feed it into runtime and get results
- but before: TS code -> TS AST -> Typechecker -> JS Source
- Typechecker: A special program that verifies that your code is typesafe
- Important fact: when TSC compiles your code from TypeScript to JavaScript, it won't look at your types

# Basics: TypeChecker

- Type System: A set of rules that a typechecker uses to assign types to your program
- General Rule: Type Explicitly Declared and Type Automatically Inferred
- Typescript does both: It can infer from example and you can declare it
- Good Programming style: Write where necessary, infer if it's possible

# Basics: tsconfig.json

- Every TypeScript project should include a file called tsconfig.json in its root directory
- It's a configuration file where you can set different properties of the compiling process
- You can set: which file should be compiled, which directory compile them to, which version of JavaScript to emit
- you can configure the tsconfig file also by command line



## Basics: tslint.json

- tslint.json for configuration and management of the code formatting style

# Types

- string, boolean, number
- any
- undefined
- null

```
let n: number = 10;
```

# Types: Basic Types

- A set of values and the things you can do with them
- Example: boolean, string, number

# Types: TypeScript's type hierarchy

- Every type extends `unknown`
- `any` extends `unknown`
- `number`, `bigint`, `boolean`, `string`, `symbol`, `Object` types extend `any`
- bounds of types: a variable upper bound (in type) is `number`. It cannot be a `string` or more than a `number` it's not assignable

# Types: Fundamentals

- any:
  - avoid, if you can
  - use it when you and the typechecker are not be able to infer the type
  - you can do everything and it can be everything
  - working with any is like working in JavaScript, without TypeChecker
- unknown

## Types: Fundamentals (2)

- boolean: as always, for the moment
- number: as always, for the moment
  - use\_separators: 1\_000\_000
- bigint: as always, defined by n
- string

# Types: Fundamentals (Object)

```
let a = {  
  b: 'x'  
}  
console.log(a.b);  
let b = {  
  c: {  
    d: 'f'  
  }  
}  
let a: {b: number} = {  
  b: 12  
}  
let c: {  
  firstName: string  
  lastName: string  
} = {firstName: 'john', lastName: 'barrowman' }
```

# Type: Classes

```
class Person {  
    constructor (  
        public firstName: string, // public is shorthand for  
        public lastName: string // this.firstName = firstName  
    ) {}  
}  
c = new Person('matt', 'smith');
```



# Types: Type Aliases

- Use it for DRYing up repeated complex types

```
type Age = number
type Person = {
  name: string
  age: Age
}
```

# Types: Union and Intersection Types

```
type Cat = {name: string, purrs: boolean}
type Dog = {name: string, barks: boolean, wags: boolean}
type CatOrDogOrBoth = Cat | Dog
type CatAndDog = Cat & Dog

let b: CatAndDog {
  name: 'Domino',
  barks: true,
  purrs: true,
  wags: true
}
```

# Types: Fundamentals (3)

- symbol
  - alternative to string keys in object and in map
  -
- Objects
- Arrays
- Tuples
- null, undefined, void and never
- Enums

# Types: Arrays

```
let a = [1,2,3] // number[]
var b = ['a','b'] // string[]
let c: string[] = ['a'] // string[]
let d = [1,'a'] // (string | number)[]
const e = [2,'b'] // (string | number)[]
let f = ['red']
f.push('blue')
f.push(true) // Error TS2345: Argument of type 'true' is not assignable to parameter of type 'string'
let g = [] // any[]
g.push(1) // number[]
g.push('red') // (string | number)[]
let h: number[] = [] // number[]
h.push(1) // number[]
h.push('red') // Error TS2345: Argument of type '"red"' is not assignable to parameter of type 'number'
```

# Types: Tuples

```
let a: [number] = [1]
// A tuple of [first name, last name, birth year]
let b: [string, string, number] = ['malcom', 'gladwell', 1963]

b = ['queen', 'elizabeth', 'ii', 1926] // Error TS2322: Type 'string' is not assignable to type 'number'

// With optional element
let trainFares: [number, number?][] = [
  [3.75],
  [9.25, 7.70],
  [10.50]
]
// Equivalently
let moreTrainFares: ([number] | [number, number])[] = [
  //...
]
```

# Types: readonly Tuples and Arrays

```
let as: readonly number[] = [1,2,3]
let bs: readonly number[] = as.concat(4)
as[4] = 5 // Error readonly
as.push(6) // Error readonly
```

## Types: null, undefined, void and never

- undefined: used in case of something hasn't been defined yet
- null means an absence of value
- void: used in function that not return a value
- never: used in function that never returns a value, e.g throws an exception or cycle indefinitely

# Types: Enums

```
enum Language {  
    English, Spanish, Russian  
}  
enum Language {  
    English = 0, Spanish = 1, Russian = 2  
}  
let myFirstLanguage = Language.Russian  
let mySecondLanguage = Language['English']  
// You can split the definition in two parts  
enum Language {  
    English = 0,  
    Spanish = 1  
}  
enum Language {  
    Russian = 2  
}
```



## Types: Enums (2)

```
enum Language {  
    English = 100,  
    Spanish = 200 + 300,  
    Russian // inferred 501  
}  
  
enum Color {  
    Red = '#c10000',  
    Blue = '#007ac1',  
    Pink = 0xc10050,  
    White = 255  
}  
  
let d = Color[6] // doesn't give you an error  
  
const enum Language {  
    English,  
    Spanish,  
    Russian  
}  
  
let a = Language.English  
let b = Language.Tagalog // error!!!  
let d = Language[6] // error!!!
```

# Functions

- Return type
- Optional parameters
- Narrowing
- Optional call
- Optional operator

# Functions: Declaring and Invoking Functions

```
function add(a: number, b: number) {  
    return a + b  
}  
function add(a: number, b: number): number {  
    return a + b  
}
```

# Functions: Optional and Default Parameters

```
function log(message: string, userId?: string) {  
    let time = new Date().toLocaleTimeString()  
    console.log(time,message,userId || 'Not signed in')  
}  
function log(message: string, userId = 'Not signed in') {  
    let time = new Date().toISOString()  
    console.log(time, message, userId)  
}
```

# Functions: Variable number params

```
function sum(numbers: number[]): number {  
    return numbers.reduce( (total, n) => total + n, 0 )  
}
```

## Function: Variable number params (2)

- Arity in functions is defined by the number of parameters that are expected to be passed
- A function is called Variadic, if the number of parameter is not fixed
- A function is called fixed-arity function, if the number of parameter is given and fixed

## Function: Variable number params (3)

- A safe way to realize a variadic function in TypeScript is that in which you use the rest operator, declaring the type of the array

```
function sumVariadicSafe(...numbers: number[]): number {  
    return numbers.reduce((total, n) => total + n, 0)  
}
```

# Function: call, apply and bind

```
function add(a: number, b: number): number {  
    return a + b  
}  
add(10,20) // evaluates to 30  
add.apply(null, [10,20]) // by spreading, evaluates to 30  
add.call(null, 10, 20) // by not spreading, evaluates to 30  
add.bind(null, 10, 20)() // by not spreading and not directly invoking, evaluates to 30
```



# Classes and Interfaces

# Classes and Interfaces: General Structure

```
class Person {  
  name: string;  
  age: number;  
  
  constructor (name: string, age: number) {  
    this.name = name;  
    this.age = age;  
  }  
  
  greet(): void {  
    console.log(`Hi, my name is ${this.name} and I'm ${this.age} years old`);  
  }  
}  
  
const henry = new Person("Henry", 30);  
henry.greet(); // Hi, my name is Henry and I'm 30 years old
```

# Classes and Interfaces: Access Modifier

- `public` (default): accessible from anywhere
- `protected` : accessible from class and subclasses
- `private` : accessible only from the class

# Classes and Interfaces: Inheritance

```
class Vehicle {  
    switchOn(): void {  
        console.log("Vehicle switched on!");  
    }  
}  
  
class Car extends Vehicle {  
    drive(): void {  
        console.log("Driving the car!");  
    }  
}  
  
const tesla = new Car();  
tesla.switchOn();  
tesla.drive();
```

# Classes and Interfaces: readonly property

```
class Book {  
    readonly title: string;  
  
    constructor(title: string) {  
        this.title = title;  
    }  
}  
  
const book = new Book("1984");  
book.title = "Animal Farm";
```

# Classes and Interfaces: Getter and Setter

```
class Product {  
    private _price: number;  
  
    constructor(price: number) {  
        this._price = price;  
    }  
  
    get price(): number {  
        return this._price;  
    }  
  
    set price(val: number) {  
        if (val > 0)  
            this._price = val;  
    }  
}
```

# Classes and Interfaces: Static Properties and Methods

```
class Math {  
    static PI = 3.14;  
  
    static squared (x: number): number {  
        return x * x;  
    }  
}  
console.log(Math.PI);  
console.log(Math.squared(5));
```

# Classes and Interfaces: Interfaces and Implementations

```
interface Animal {  
    name: string;  
    makeSound(): void;  
}  
class Dog implements Animal {  
    name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
  
    makeSound(): void {  
        console.log("Bau!");  
    }  
}
```



# Classes and Interfaces: super call

- super call: if the child class overrides a method defined on its parent class, the child instance can make a super call to call its parent's version of the method

# Classes and Interfaces: this as return type

```
class Set {  
    has(value: number): boolean {  
        // ...  
    }  
    add(value: number): this {  
        // ...  
    }  
}
```

# Classes and Interfaces: Type Aliases and Interfaces (1)

- Comparison between type aliases and interfaces

```
type Sushi = {  
    calories: number  
    salty: boolean  
    tasty: boolean  
}  
interface Sushi = {  
    calories: number  
    salty: boolean  
    tasty: boolean  
}
```

# Classes and Interfaces: Type Aliases and Interfaces (2)

- Comparison between type aliases and interfaces

```
type Food = {  
    calories: number  
    tasty: boolean  
}  
type Sushi = Food & {  
    salty: boolean  
}  
type Cake = Food & {  
    sweet: boolean  
}
```

```
interface Food {  
    calories: number  
    tasty: boolean  
}  
interface Sushi extends Food {
```

# Classes and Interfaces: Type Aliases and Interfaces (3)

- Differences between Type Aliases and Interfaces:
  - Inheritance: Interfaces can be extended
  - Interface merging: type can be extended only with `&` (intersection), and it cannot be redefined
  - Union and Intersection: only type aliases can create advanced type with union and intersection
  - Type is better to use with primitive types and tuples

# Handling Errors

# Handling Errors: Base syntax

```
try {  
    let result = riskyFunction();  
    console.log(result);  
} catch(error) {  
    console.error("An error occurred!", error);  
} finally {  
    console.log("This runs no matter what.");  
}
```

# Handling Errors: Typed error handling

```
try {  
    throw new Error("Something went wrong");  
} catch (err: unknown) {  
    if (err instanceof Error) {  
        console.error("Error:", err.message);  
    } else {  
        console.error("Unknown error");  
    }  
}
```



# Handling Errors: Custom error classes

```
class InvalidValueerro extends Error {  
  constructor(value: string) {  
    super(`Invalid value: ${value}`);  
    this.name = "InvalidValueError";  
  }  
}  
  
function check(value: string) {  
  if (value !== "ok") {  
    throw new InvalidValueError(value);  
  }  
}
```