**Table of content**

* [**Basic Types**](https://www.typescriptlang.org/docs/handbook/basic-types.html)
* [**Variable Declarations**](https://www.typescriptlang.org/docs/handbook/variable-declarations.html)
* [**Interfaces**](https://www.typescriptlang.org/docs/handbook/interfaces.html)
* [**Classes**](https://www.typescriptlang.org/docs/handbook/classes.html)
* [**Functions**](https://www.typescriptlang.org/docs/handbook/functions.html)
* [**Generics**](https://www.typescriptlang.org/docs/handbook/generics.html)
* [**Enums**](https://www.typescriptlang.org/docs/handbook/enums.html)
* [**Type Inference**](https://www.typescriptlang.org/docs/handbook/type-inference.html)
* [**Type Compatibility**](https://www.typescriptlang.org/docs/handbook/type-compatibility.html)
* [**Advanced Types**](https://www.typescriptlang.org/docs/handbook/advanced-types.html)
* [**Symbols**](https://www.typescriptlang.org/docs/handbook/symbols.html)
* [**Iterators and Generators**](https://www.typescriptlang.org/docs/handbook/iterators-and-generators.html)

*1.Basic Types*

* boolean
* number
* string
* array number[] or Array<number>
* tuple fixed array with differenet types **let** x: [string, number];
* enum
* any
* void
* null and undefined By default null and undefined are subtypes of all other types. However, when using the --strictNullChecks flag, null and undefined are only assignable to any and their respective types (the one exception being that undefined is also assignable to void).
* Never ... unreachable end of function used with error throws
* Object something that is non primitive

Type Assertion

Something is : any but we know that it is a string so x.length throws error we use it as

(<string>x).length or (x as string ).length

# *2.Variable Declarations*

* Var function scope
* Let block scope
* Const immutable

Array destructuring

**let** input = [1, 2];

**let** [first, second] = input;

* Object destructuring
* Tuple destructuring
* Spread operator ...array is used in an other array so it taks spread arrays every element and adds it in the new array It is also used as rest operator
* Default values... optonal paramenters

# *3.Interfaces*

* Optional properties
* Readonly properties can use readonly x : number
* ReadonlyArray<number> ... is in lib ,,, it cant be mutated besides ro: ReadonlyArray<T> as T[]
* Functon Typs

interface SearchFunc {

(source: string, subString: string): boolean;

}

Once defined, we can use this function type interface like we would other interfaces

let mySearch: SearchFunc;

mySearch = function(source: string, subString: string) {...}

* Indexable types

**interface** StringArray { [index: number]: string; }

* Interfaces extend other interfaces
* Class cant implement interface with constructor.
* Hybrid types is interface with properties and methods
* Interface can extend a class so its inherits its members but not implementations

# *4.Classes*

* Readonly modifiers ca be set in constructors ,,, using a   
  constructor(public a : number)
* Creates a public a member and sets it in constructor
* Getter setters
* Static properties is accesed on class itself class can be not static at all but have static members
* Abstract classes

# *5.Functions*

* Function types

**let** myAdd: (x: number, y: number) => number = **function**(x: number, y: number): **number** { **return** x + y; };

* **Optional and default parameteres**
* **Rest parameteres**

function buildName(firstName: string, ...restOfName: string[]) {

return firstName + " " + restOfName.join(" ");

} // employeeName will be "Joseph Samuel Lucas MacKinzie"

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");

* In function this is WINDOW object or undefined in strict mode
* In arrow function this is the object
* In JS we use let \_this = this; in objct syntax and use \_this variale in places of this object
* We can provide this in parameters
* Overloading in ts

**function** **pickCard**(x: {suit: string; card: number; }[]): **number**; **function** **pickCard**(x: number): {suit: string; card: number; }; **function** **pickCard**(x): **any** {}

**here we cant provide x as any only as a number or an object having a suit and a card ...**

# *6.Generics*

* **function** **identity**<**T**>(arg: T): **T** { **return** arg; }

we use it as

**let** output = identity<string>("myString"); OR

JUST LIKE THIS

**let** output = identity("myString");

* Generic classes

**class** GenericNumber<T> { zeroValue: T; add: (x: T, y: T) => T; }

* Genric constraint

**interface** Lengthwise { length: number; }

**function** **loggingIdentity**<**T** **extends** **Lengthwise**>(arg: T): **T** {

console.log(arg.length);

// Now we know it has a .length property, so no more error

**return** arg; }

* Factory

**function** **create**<**T**>(c: {**new**(): **T**; }): T { **return** **new** c(); }

# *7.Enums*