2

Types, Variables and Functions

Type annotations, Primitive and Special types, Object types, Functions

Type annotations

- Optional but recommended feature
- Allows for type checking

- Can be applied to
  - Variables
  - Input parameters in functions
  - Return types

```
function Add(left: number, right: number): number {
    return left + right;
}
```

Type annotations

- Defined explicitly

```
var x: number;
```

- Defined implicitly (type inference)

```
var x = 8;
```

Variables

- **var**: used to declare a **function-scoped** variables

```
var x: number;
```

- **const**: used to declare constant values, same scope as let

```
const x = 8;
```

- **let**: used to declare **block-scope** variables

```
let x: number;
```

Primitive types

number

- Represent double precision (64bits)
- Used for all number type values: integer, float

```
var n: number;
n = 10
n = 15.2
```

Primitive types

`string`

- Represent sequence of characters
- Stored as UTF-16
- String values are surrounded by single (‘) or double (“) quotation marks

```
var text: string;
text = "hello world"
text = 'hello'
```

Primitive types

`boolean`

- Represent a logical value : **`true`** or **`false`**

```
var b: boolean;
b = true
b = false
```

Primitive types

`null`

- Represent a null value: empty or doesn't exist.
- It is possible to reference null type value itself
- null is **both** type and value

```
var n = null;
var n: null;
```

Primitive types

`undefined`

- Represent an undefined value: a variable is not assigned to any value
- By default Typescript assign **undefined** and not null
- It is possible to reference undefined type value itself
- undefined is **both** type and value

```
var u;
var u = undefined;
var u: undefined;
```

Special types

any

- Top-type of the type system
- All types is assignable to any type
- any is assignable ony to any

```
let value: any;
value = 42; // OK
value = "Hello World"; // OK
value = null; // OK
```

Special types

unknown

- Indicates an unknown type of the stored value
- All types is assignable to unkown type
- unknown is assignable only to **unkown** itself and **any**

```
let value: unknown;
let some : any;
value = some; // OK
value = true; // OK
value = "Hello World"; // OK
```

Special types

never

- Indicates the value that will never occur, cannot have any value
- Used to indicates logic error
- Example: a function that will never reach it's end (infinite loop)

```
function keepProcessing(): never {
    while (true) {
            console.log('I always does something and never ends.')
    }
}
```

Special types

void

- Used with functions that return nothing
- There is no meaning to assign void to variables
- void type can have undefined or null values

```
function sayHi(): void {
    console.log('Hi!')
}
```

Special types

Array

- Used to store multiple values of same type
- Using brackets (**[]**)

```
let fruits: string[] = ['Apple', 'Orange', 'Banana'];
```

- Using generic array type: **Array<T>**

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

Special types

Tuple

- A Tuple is an array that can contain two values of different types
- Using brackets (**`[T1, T2]`**)

```
var tuple: [number, string] = [2, "hello"];
```

- **`number[]`** and **`[number]`** are two different types,

Type assertion

- Allows to set the type of a value **explicitly** after its declaration
- Prevent compiler from inferring the type
- Type assertion has no runtime effect
- Using angular brackets **<T>** or **as** keyword

```
let code: any = 123;
let employeeCode = <number> code;
//or
let employeeCode =  code as number;
```

Type aliases

- Type aliases give a type a new name
- Does **NOT** create a new type
- Using **type** keyword

```
type ID = number
var n : ID = 10
```

```
interface Person {
    //properties here
}

type User = Person
```

Object types

- In Typescript, objects can also be described by their properties
- Typing objects helps to avoid runtime errors

```
// Object type literal
let user: {id: string, email: string};

user.id    //OK
user.email //OK
user.name  //Error, name property does not exist
```

Object types

- Object types can be named using **interface** or **type**

```
interface User {
   id: string;
   email: string;
}
```

```
type User = {
   id: string;
   email: string;
}
```

```
var user: User
```

Object types

Optional properties

- Mark a property as optional by adding question mark (**?**)
- Providing values for optional properties is not required

```
interface PaintOptions {
    shape: Shape;
    xPos?: number; //Optional
    yPos?: number; //Optional
 }
var paint: PaintOptions  = { shape : Shape.Square} //OK
```

Object types

Optional chaining

- Access optional properties using optional chaining operator (**?**)
- Returns **undefined** instead of throwing an error

```
interface Customer {
    name: string;
    email?: string;
    address?: {
        type?: string;
        city?: string;
        state?: string;
    }
}
```

```
// Previous way
if (customer.address && customer.address.city) {
    let city = customer.address.city;
}

//New way
let city = customer.address?.city;
```

Object types

Non-null assertion

- Non-null assertion operator (**!**) assert that a value is **not null** and **not undefined**
- **CAUTION**: Throws an error if value is null or undefined

```
var customer: Customer = {name: "John", email: "test@test.com"}

var address = customer.address!
var city = address.city  //Error, address is undefined
```

Duck typing

- Duck typing is a technique used to compare two-objects
  - Same type if same properties and members
  - Also known as Structural typing
  - Takes into account only members of a type

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck ”

Duck typing

- Two objects are considered to be the same type of both share the same properties

```
interface A {
  count: number
}

var a: A
```

=

```
interface B {
  count: number
}

var b: B
```

```
b = a; //correct
a = b; //correct
```

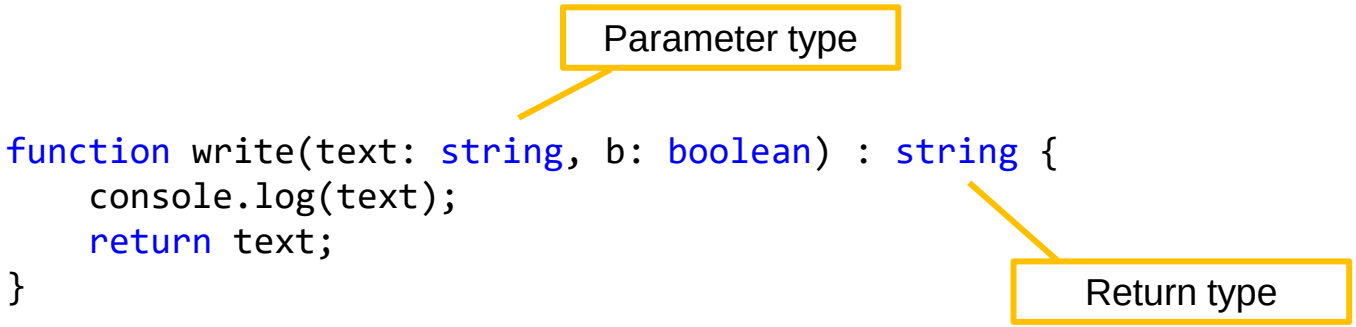
Duck typing

- Duck-typing system allows object creation **on the fly** while keeping type safety

```
interface IPoint {
  x: number
  y: number
}

var a: IPoint

var b = {x: 1, y: 2}

a = b //Correct
b = a // Also correct
```

Functions

- Declaration



- Return type can be inferred implicitly

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

Functions

- Optional parameters

default value

optional

```
function write(text: string, b1: boolean = false, b2?: boolean) : string {
    console.log(text);
    return text;
}
```

Functions

- Arrow function
  - More readable version of functions

```
var fn = (a: number, b: number): number => {
   return a+b;
}
```