Chapter 6: Functions

Section 6.1: Optional and Default Parameters

Optional Parameters

In TypeScript, every parameter is assumed to be required by the function. You can add a **?** at the end of a parameter name to set it as optional.

For example, the `lastName` parameter of this function is optional:

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

Optional parameters must come after all non-optional parameters:

```
function buildName(firstName?: string, lastName: string) // Invalid
```

Default Parameters

If the user passes **`undefined`** or doesn't specify an argument, the default value will be assigned. These are called *default-initialized* parameters.

For example, "Smith" is the default value for the `lastName` parameter.

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
buildName('foo', 'bar');      // firstName == 'foo', lastName == 'bar'
buildName('foo');             // firstName == 'foo', lastName == 'Smith'
buildName('foo', undefined);  // firstName == 'foo', lastName == 'Smith'
```

Section 6.2: Function as a parameter

Suppose we want to receive a function as a parameter, we can do it like this:

```
function foo(otherFunc: Function): void {
    ...
}
```

If we want to receive a constructor as a parameter:

```
function foo(constructorFunc: { new() }) {
    new constructorFunc();
}

function foo(constructorWithParamsFunc: { new(num: number) }) {
    new constructorWithParamsFunc(1);
}
```

Or to make it easier to read we can define an interface describing the constructor:

```
interface IConstructor {
    new();
```

```
}

function foo(contructorFunc: IConstructor) {
    new constructorFunc();
}
```

Or with parameters:

```
interface INumberConstructor {
    new(num: number);
}

function foo(contructorFunc: INumberConstructor) {
    new contructorFunc(1);
}
```

Even with generics:

```
interface ITConstructor<T, U> {
    new(item: T): U;
}

function foo<T, U>(contructorFunc: ITConstructor<T, U>, item: T): U {
    return new contructorFunc(item);
}
```

If we want to receive a simple function and not a constructor it's almost the same:

```
function foo(func: { (): void }) {
    func();
}

function foo(constructorWithParamsFunc: { (num: number): void }) {
    new constructorWithParamsFunc(1);
}
```

Or to make it easier to read we can define an interface describing the function:

```
interface IFunction {
    (): void;
}

function foo(func: IFunction ) {
    func();
}
```

Or with parameters:

```
interface INumberFunction {
    (num: number): string;
}

function foo(func: INumberFunction ) {
    func(1);
}
```

Even with generics:

```
interface ITFunc<T, U> {
    (item: T): U;
}

function foo<T, U>(contructorFunc: ITFunc<T, U>, item: T): U {
    return func(item);
}
```

Section 6.3: Functions with Union Types

A TypeScript function can take in parameters of multiple, predefined types using union types.

```
function whatTime(hour:number|string, minute:number|string):string{
    return hour+':'+minute;
}

whatTime(1,30)          //'1:30'
whatTime('1',30)        //'1:30'
whatTime(1,'30')        //'1:30'
whatTime('1','30')      //'1:30'
```

TypeScript treats these parameters as a single type that is a union of the other types, so your function must be able to handle parameters of any type that is in the union.

```
function addTen(start:number|string):number{
    if(typeof number === 'string'){
        return parseInt(number)+10;
    }else{
        else return number+10;
    }
}
```

Section 6.4: Types of Functions

Named functions

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

Anonymous functions

```
let multiply = function(a, b) { return a * b; };
```

Lambda / arrow functions

```
let multiply = (a, b) => { return a * b; };
```