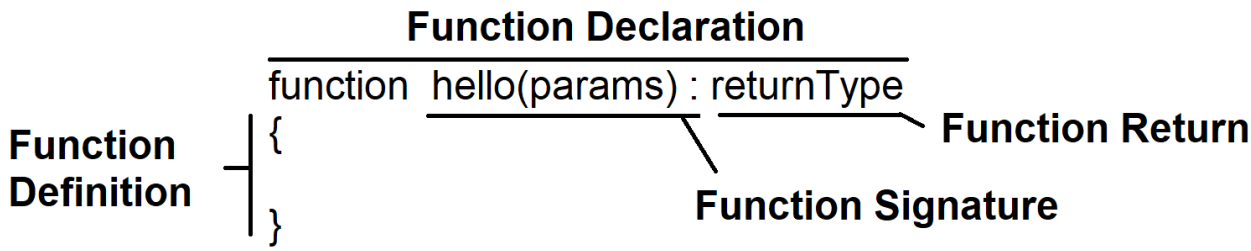
TypeScript Functions and Methods

- In computer programming the actions are defined by using
    - Function
    - Method
    - Procedure
- Function is intended to return a value.
- Method is intended to handle functionality without returning a value.
- Procedure may or may not return a value.
- JavaScript supports only functions, that have the behaviour of Method and Procedure.
- JavaScript supports methods from version ES5.
- TypeScript supports functions outside the class and methods with in a class.
- However, TypeScript methods have the behaviour of functions and procedures.
- Function will not have any restricted access.
- Methods can have restricted access, it can be configured as public, private and protected.

Function

- Function definition must be outside the class.
- Function will have a declaration, signature, return type and definition.
- Declaration specifies its accessibility and scope.

- Signature defines how it is accessed or used from any location.
- Definition specifies the functionality.
- Return type defines the behaviour of function definition. It defines the ability of function.



Function Return Types:

- A function can be configured to return a value or not to return a value.
- A function is configured to return a value when it is defined as an expression.
- Expression will take values, evaluate a value and return value.

  Syntax:

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

- Parameters “a, b” will take value

- “a+b” evaluate a value.
- “return” will return the value.
- If a function is intended to just handle a functionality but not evaluate a value then it is defined with return type as “Void”.
- **Void** is used to define an ability to function so that it can’t return a value. But it can handle specific functionality.
- **Void** is an operator

```
function Print():void
{
    console.log("Print Funciton");
}
```

FAQ: Can a function with Void return type have a return statement in definition?

A. **Yes. But the code defined after return statement is not reachable to compiler.**
   We use this technique for configure “Stubs” in a program.
   Even it is not reachable to compiler while executing it is compiled. Hence it must have a definition for “to be implemented”

Function with Return Type

- A function is defined with return type in order to handle as expression.
- It evaluates a value.
- Functions are strongly typed. A function can return only the type of value defined as function return type.

Ex:

```
function captcha():string
{
   let a:number = Math.random() * 10;
   let b:number = Math.random() * 10;
   let c:number = Math.random() * 10;
   let d:number = Math.random() * 10;
   let e:number = Math.random() * 10;
   let f:number = Math.random() * 10;
   let code = `${Math.round(a)} ${Math.round(b)} ${Math.round(c)} ${Math.round(d)} ${Math.round(e)} ${Math.round(f)}`;
   return code;
}
class Demo
{
```

```
    public Print():void {
        console.log(`Verify Code : ${captcha()}`);
    }
}
let obj = new Demo();
obj.Print();
obj.Print();
obj.Print();
```

FAQ: What can be the return type defined for a function?

A. It can be any type. Primitive or Non-Primitive.

Ex:

```
function Product():any
{
    let obj = {
        Name: "Samsung TV",
        Price: 45000.55
    };
    return obj;
```

```
}
let tv = Product();
console.log(`Name=${tv.Name}\nPrice=${tv.Price}`);
for(var property in Product())
{
    console.log(`${property}:${Product()[property]}`)
}
```

FAQ: Can a function be defined with more than one return type?

A.Yes. By using Union of Types.

Ex:

```
function pagestatus():string | number
{
    let page:string = 'about.html';
    if(page=='home.html') {
        return 'Request Status : OK';
    } else {
        return 404;
    }
```

```
}
console.log(pagestatus());
```

Ex: **Using a Method**

```
class Demo
{
   public pageName:string;
   public StatusMessage():string | number {
      if(this.pageName=="home.html") {
         return "Response Status - OK";
      } else {
         return 404;
      }
   }
}
let obj = new Demo();
obj.pageName = "home.html";
console.log(obj.StatusMessage());
```

Function Parameters

- Same like constructor parameters.
- The parameters defined in function signature are known as “Formal Parameters”
- The parameters passed into a function call are known as “Actual Parameters”.

```
function name(param)
{

}

name(paramValue)
```

Formal Parameter

Actual Parameter Arguments

- Multiple Parameters
- Optional Parameters
- Rest Parameters

Function Recursion

- Recursion is a technique where a function is called within the context of same function.
- Usually we use recursion for performing “Batch Operations”

Ex:

```
function fact(n:number):number
```

```
{
    if(n<=0)
    {
        return 1;
    } else {
        return n * (fact(n-1));
    }
}
console.log(`Factorial of 5 is ${fact(5)}`);
```

Anonymous Function

- A function defined without a name is known as Anonymous function.
- Anonymous function is configured with in the parenthesis “( )”.
- You can access anonymous function by using signature or definition.

Ex:

```
(function(msg:string){
    console.log(msg);
})("Welcome to TypeScript")
```

- Anonymous functions are used for implementing call back mechanism.
- Call back is a mechanism where function automatically executes according to the state and situation. It doesn’t require an explicit call.

Ex:

```
let password:string = "admin12";
let verifyPassword:any = [function(){}, function(){}];


verifyPassword[0] = function(){
   console.log("Password Verified");
}
verifyPassword[1] = function(){
   console.log("Invalid Password");
}
function overloadedFunction(success, failure)
{
   if(password=="admin123")
   {
      return success();
   } else {
```

```
        return failure();
    }
}
overloadedFunction(verifyPassword[0],verifyPassword[
1]);
```

Ex:

```
let functions = [
    function(){
        console.log("Password Verified");
    },
    function(){
        console.log("Invalid Password");
    }
];
var [success, failure] = functions;    // Array
Destruction


function VerifyPassword(pwd, success, failure){
    if(pwd=="admin123")
    {
        success();
```

```
    } else {
        failure();
    }
}
VerifyPassword("admin1",success, failure);
```

Ex:

```
let functions = [
    function(){
        console.log("Password Verified");
    },
    function(){
        console.log("Invalid Password");
    }
];
var [success, failure] = functions;

function VerifyPassword(pwd, success, failure){
    if(pwd=="admin123")
    {
```

```
        success();
    } else {
        failure();
    }
}
VerifyPassword("admin123",function(){console.log("Success..")},function(){console.log("Failure")});
```

Ex:

```
var welcome = function(msg:string)
{
    return msg;
}
console.log(welcome("Welcome to TypeScript"));
```

Lambda Notation for Functions

- Lambda allows a developer to minify any function.
- Minification is a technique of reducing the number of lines that you write but keep the functionality same.

- It is a technique of writing lengthy code in short hand method.
- Lambda is used for functions.

Syntax:

( ) → Function parameter less

(param) → Parameterized function

=> → value/ functionality to return

{ } → Logic to execute

Var hello = (param) => { statement1; statement2 };
Var hello = (param) => statement1;

Ex:

```
var hello = function(name:string) {
   return `Hello ! ${name}`;
}
var welcome  = (msg:string)=>msg;
var Print    = ()=>console.log("Print Function");
var Addition = (a:number, b:number)=>a+b;
```

```
var Multiply = ()=>{let a=4; let b=5; console.log(a*b)};

console.log(hello("John"));
console.log(welcome("TypeScript"));
Print();
console.log(`Addition= ${Addition(10,20)}`);
Multiply();
```

Ex: Filter Data without Lambda approach for function

```
let products:any[] = [
   {Name: "TV", Category: "Electronics"},
   {Name: "Nike Casuals", Category: "Footwear"},
   {Name: "Mobile", Category: "Electronics"},
   {Name: "Shirt", Category: "Fashion"}
];
let electronics:any[] = products.filter(
   function(product)
   {
      return product.Category=="Electronics"
   })
```

```
for(var item of electronics)
{
   console.log(item.Name);
}
```

Ex: Using Lambda Approach

```
let products:any[] = [
   {Name: "TV", Category: "Electronics"},
   {Name: "Nike Casuals", Category: "Footwear"},
   {Name: "Mobile", Category: "Electronics"},
   {Name: "Shirt", Category: "Fashion"}
];
let electronics:any[] =
products.filter(product=>product.Category=="Electro
nics");
for(var item of electronics)
{
   console.log(item.Name);
}
```