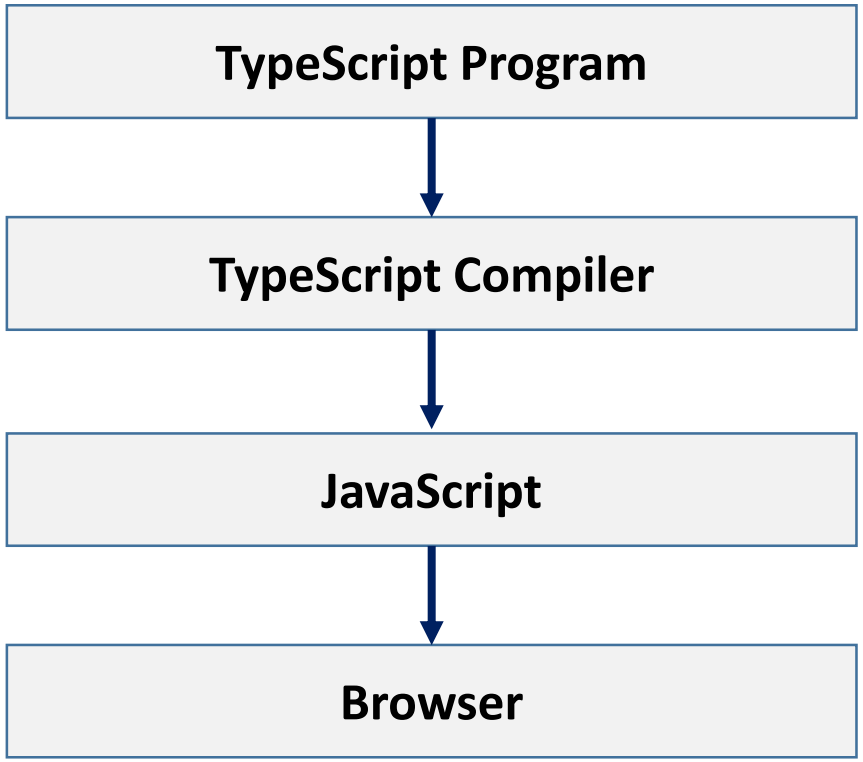
TypeScript Language

By
Srikanth Pragada

What is TypeScript?

- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
- TypeScript is pure object oriented with classes and interfaces.
- TypeScript is statically typed.
- It is projected as scalable JavaScript. It is known as JavaScript that scales.
- It is the preferred language to build Angular 2 applications as Angular 2 itself was written in TypeScript.
- It was designed by Anders Hejlsberg (designer of C#) at Microsoft.



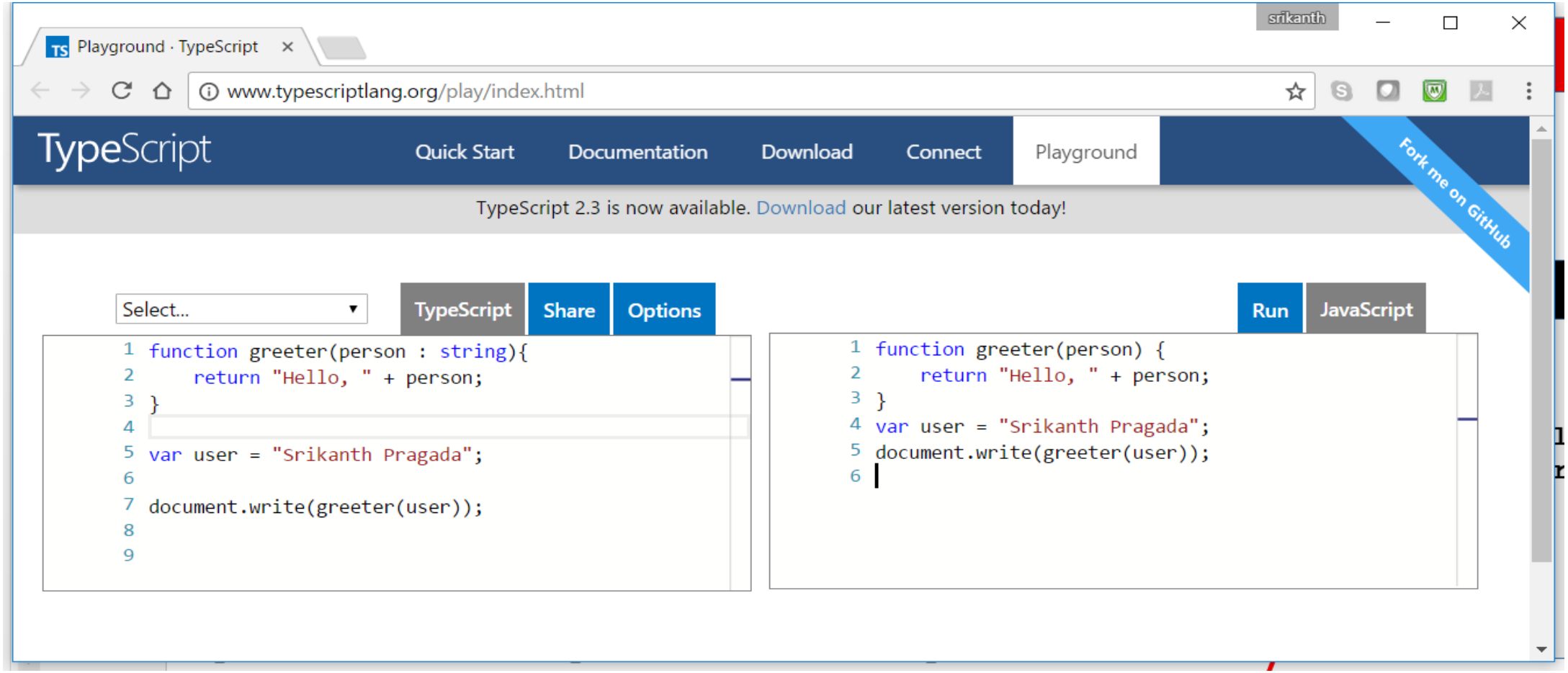
Installing TypeScript

- Install typescript using NPM (Node.js Package Manager).
- Or install TypeScript visual studio plugins.
- Visual Studio 2017 and Visual Studio 2015 Update 3 include TypeScript by default.

```
>npm install –g typescript
```

Using TypeScript Playground

- Allows you to write and test TypeScript using Browser.
- No need to install Node.js and TypeScript.
- Available at http://www.typescriptlang.org/play/index.html.
- Shows converted JavaScript immediately and allows you to run JavaScript in Browser.



Compiling TypeScript and running with Node.js

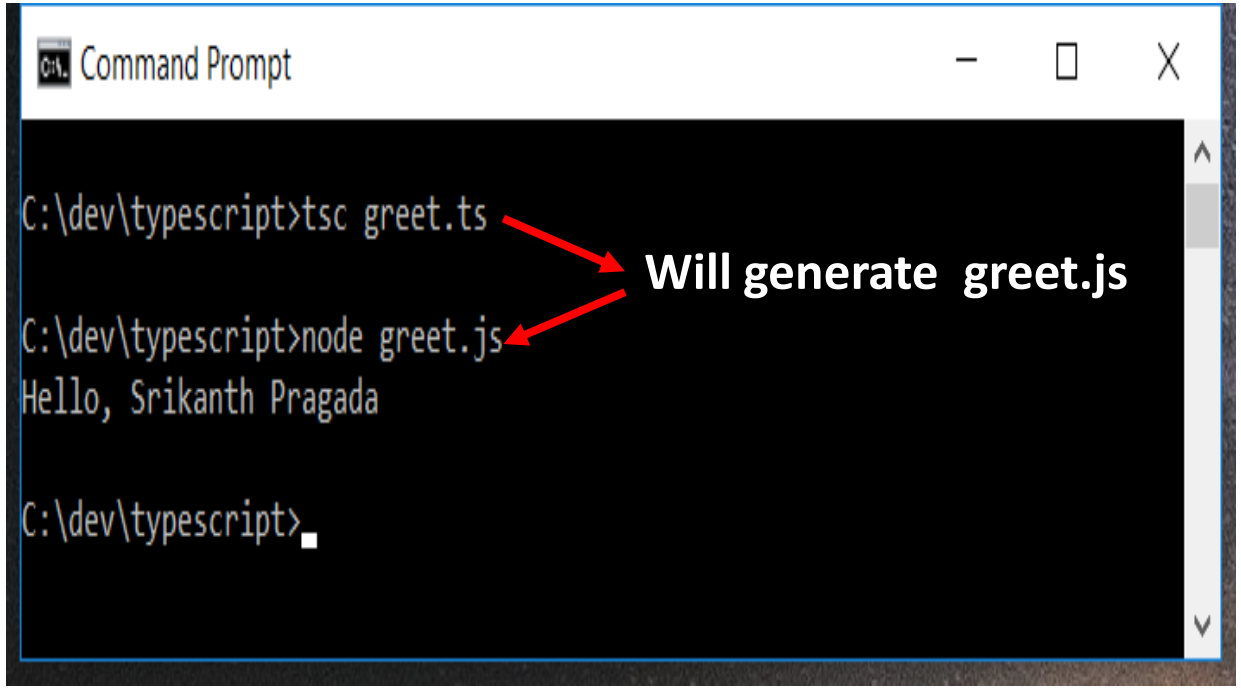
- Compile typescript file (.ts) using TSC (TypeScript Compiler).
- Run generated .js file with Node.js.
- In this mode, you can't use HTML DOM as it runs outside HTML.

greet.ts

```
function greeter(person : string){
    return "Hello, " + person;
}

var user = "Srikanth Pragada";

console.log( greeter(user));
```



Compiling TypeScript and running with Browser

- Compile typescript file (.ts) using TSC (TypeScript Compiler).
- Embed generated .js file in a HTML page using <script> tag.
- Run HTML page in Browser.
- It is possible to use HTML DOM as program is run inside HTML page.

hello.ts

```
function sayHello(person : string) {
    return "Hello, " + person;
}

var user = "Srikanth Pragada";

document.write(sayHello(user));
```

hello.html

```
<!DOCTYPE html>
<html>
  <head>
   <title>TypeScript Demo</title>
   <script src="hello.js"></script>
  </head>
  <body>

  </body>
</html>
```

```
> tsc hello.ts
```

→ Will generate hello.js

Identifiers

- Identifiers are names given to elements in a program like variables, functions etc.
- Identifiers can include both, characters and digits.
- Identifier cannot begin with a digit.
- Identifiers cannot include special symbols except for underscore (_) or a dollar sign ($).
- Identifiers cannot be keywords.
- Identifiers are case-sensitive.

Keywords

break	as	any	switch
case	if	throw	else
var	number	string	get
module	type	instanceof	typeof
public	private	enum	export
finally	for	while	void
null	super	this	new
in	return	true	false
any	extends	static	let
package	implements	interface	function
new	try	yield	const
continue	do	catch	

Data types

- Divided into three types – Any, built-in and user-defined.
- Any represents any type of value.
- Built-in types are provided by TypeScript.
- User-defined types are created by programmer such as classes and interfaces.

Built-in Data Types

Data type	Description
number	Double precision 64-bit floating point values. It can be used to represent both, integers and fractions.
string	Represents a sequence of Unicode characters.
boolean	Represents logical values, true and false.
void	Used on function return types to represent non-returning functions.
null	Represents an intentional absence of an object value.
undefined	Denotes value given to all uninitialized variables.

Any type

- The **any** data type is the super type of all types in TypeScript.
- It denotes a dynamic type.
- Using the **any** type is equivalent to opting out of type checking for a variable.

```
var  a : any;

a = 10;
console.log(a * a);  // will output 100

a = "Abc";
console.log(a + a);  // will output AbcAbc
```

Type Inference

- TypeScript can infer datatype of a variable based on the value assigned to it.

```
var st = "String"
var num = 10

console.log(typeof(st))      // string
console.log(typeof(num))     // number
```

Declaring variables

```
var identifier : [type-annotation] = value ;
```

```
var uname : string =  "Srikanth";

// variable's type is inferred from value
var uname =  "Srikanth";

// Type is any and value is undefined
var uname;
```

Variables declared with **let** are confined only to the block in which they are declared and not hoisted to the top of the block.

Variables declared with **var** are hoisted to the top of the function, so they can be accessed anywhere in the function.

Using const

- Keyword const is used to declare constants.
- Value of a constant cannot be changed.

```
const size = 10;
```

Type Assertion

- Type assertion is the process of converting variable from one data type to another.
- We need to put target type in <> in front of source variable or expression.
- Compiler decides the data type of the variable using type inference in the absence of explicit type declaration.
- Once data type of a variable is decided, its type cannot be changed.

```
var v1 : any;
v1 = "Srikanth";
var len = (<string>  v1).length;

console.log(typeof(len));    //  number
console.log(len);            //  8

var v2  = 10;                  // type of v2 is number

v2 = "20";    // Not allowed as v2 is of type number
```

Strings

- TypeScript uses double quotes (") or single quotes (') to surround string data.
- You can also use template strings, which can span multiple lines and have embedded expressions.
- These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

```
let uname : string = "Srikanth";
let subject : string = "Angular";
let sentence: string =
   `Hello,  I am ${uname} and I am your ${subject} trainer`;
```

Variable Scope

Global Scope

- ❑ Global variables are declared outside the programming constructs.
- ❑ These variables can be accessed from anywhere within your code.

Class Scope

- ❑ These variables are also called **fields**.
- ❑ Fields or class variables are declared within the class but outside the methods.
- ❑ These variables can be accessed using the object of the class. Fields can also be static. Static fields can be accessed using the class name.

Local Scope

- ❑ Local variables, as the name suggests, are declared within the constructs like methods, loops etc.
- ❑ Local variables are accessible only within the construct where they are declared.

Variable Scope - Example

```
var g : number = 1;    // Global scope
class Test
{
    static  sv : number = 2;     // Static variable
    iv  : number = 3;            // Field
    print(): void {
        var i : number = 4;     // Local scope
        console.log("Local                : " + i);
        console.log("Instance variable : " + this.iv);
        console.log("Static variable   : " + Test.sv);
        console.log("Global Variable   : " + g);
   }
}

var obj = new Test();
obj.print();
```

Loops

- while
- do.. While
- for
- for .. in
- for .. of

Loops - Examples

```
var i: number = 1;
while (i <= 10) {
    console.log(i);
    i++;
}

i = 1;
do{
    console.log(i);
    i++;
}
while (i <= 10);

for (i = 1; i <= 10; i++) {
    console.log(i);
}
```

Loops - Examples

```
var marks: number[] = [10,30,40];

// for in takes index of array
for (var idx in marks) {
    console.log(`Marks for student ${parseInt(idx)+1} are
                        ${marks[idx]}`);
}

// for of takes elements of array
for (var m of marks){
    console.log(m);
}
```

Enum

- Declares an enumeration, which is a collection of constants.

```
enum Payment { Cash, CreditCard, NEFT};

enum TransportMode { Road = 10, Rail = 20, Air = 30};

var tm = TransportMode.Air;

console.log(tm)                    // 30
console.log(TransportMode[tm])  // Air
```

Arrays

- An array is a collection of items of same type.
- In TypeScript an array is an object.
- It is possible to manipulate an array with methods and properties.

```
var marks: number[];
marks = [60, 70, 66];
console.log(marks.length)
console.log(marks[0]);
var subjects : string[] =  ["Java", "TypeScript", "Angular"];

// Use iterator
for (var sub of subjects)
    console.log(sub);

// Array Methods
subjects.push("jQuery");
console.log("Top Element  : " + subjects.pop());
```

Tuple

- A tuple is a heterogeneous collection of values.
- Individual elements are called as items.
- Tuples are index based and index starts at 0.
- Tuples are mutable, so we can manipulate them using methods and simple indexed access.
- It is possible to deconstruct a tuple – copy value to individual elements.

```
var tup1 = [10, "Abc", true];
console.log(tup1[0]);
console.log(tup1.length);
for (var v of tup1)
   console.log(v);

// change an item in tuple
tup1[2] = false;
// destructuring tuple
var [i1, i2, i3] = tup1;
console.log("Second Item : " + i2);
```

Destructuring

- The destructuring assignment syntax makes it possible to unpack values from arrays, or properties from objects, into distinct variables.
- It is a feature of ECMAScript 2015.

```
let input = [1, 2];
let [first, second] = input;

console.log(first);       // outputs 1
console.log(second);      // outputs 2
```

Functions

- TypeScript functions are created just like functions in other language like C and Java.
- We must use **function** to identify a function.
- Return type is given after : at the end of function declaration.
- Number of actual parameters and formal parameters must match in TypeScript.
- It is possible to explicitly mention that a parameter is optional by using **?** after parameter name.
- It is possible to assign default value to formal parameter by giving **=** followed by value after parameter. Any parameter with default value becomes an optional parameter.

Return type of function

```
function function_name(param[?]:datatype [= value],
                       param[?]:datatype [= value]): returntype
{
    // function body
}
```

Optional Parameter

Default Value for Parameter

Function Definition

```
function add ( n1: number, n2:number) : number{
    return n1 + n2;
}

console.log( add(10,20));
```

```
let sub = function( n1: number, n2:number) : number{
    return n1 - n2;
}

console.log(sub(50,20));
```

Optional Parameters

```
// Optional parameter - n2 declared with ? after parameter name

function mul(n1 : number, n2? : number) : number
{
    if (n2)  // if parameter is passed
        return n1 * n2;
    else
        return n1 * 10;
}

console.log( mul(10,20));
console.log( mul(10));
```

Default Parameters

```
// Setting second parameter n2 to default value
function div(n1 : number, n2 : number = 10) : number
{
    return n1 / n2;
}

console.log( div(100,5));
console.log( div(100));   // uses 10 for second parameter
```

Rest Parameters

```
// Rest parameters
function printMessage(message : string , ... names : string[])
{
      for(let n of names)
        console.log( message + " " + n);
}

printMessage("Hello", "Ben","Joe");
printMessage("Hi", "Scott","Anders","Tom");
```

```
Hello Ben
Hello Joe
Hi Scott
Hi Anders
Hi Tom
```

Function Overloading

- TypeScript cannot support two or more function with same name but different types of parameters as underlying JavaScript doesn't support that due to lack to datatypes.
- So, we have to create only one function definition but provide multiple function declarations (signatures) with different types of parameters.

```
// declare functions to be overloaded
function getTitles(author:string): string[];
function getTitles(status:Boolean): string[];

// function definition
function getTitles(value : any) : string[] {
   if( typeof(value) == 'string') {
   }
   else {
   }
}
```

```
books = getTitles('Jason')
books = getTitles(true)
```

Function Overloading

```
// declare functions to be overloaded
function f1(x: number): void;
function f1(s: string): void;
function f1(x: number, s: string): void;

function f1(n: any, s? : any): void
{
   console.log(`value is ${n}. Type is ${typeof (n)}`);
   if (s)
     console.log(`Second parameter is ${s}`);
}
```

```
f1("Abc");
f1(10);
f1(100, "PQR");
```

Lambda Functions

- Anonymous functions can be represented using lambda expressions or lambda statements.
- A lambda function contains the following components :
  - ✓ Parameters
  - ✓ Fat arrow
  - ✓ Statements or expression

```
(parameters) =>  Expression
```

```
// Lambda Expression
let nextEven = (n : number) => n % 2 == 0 ? n + 2 : n + 1;
```

```
// Lambda Block
let nextEven = (n: number) => {
     return n = n % 2 == 0 ? n + 2 : n + 1;
}
```

Classes

- TypeScript supports classes, which were introduced in **ES6**.
- A class may contains fields, constructors, and methods.
- We instantiate objects using **new** keyword followed by classname.
- Fields and methods are accessed using dot operator (.).
- Constructors are defined using keyword **constructor**.
- Classes can have static members that represent data and operations related to class, and declared using **static** keyword.
- Static members are accessed through classname.
- Classes can implement interfaces using **implements** keyword.

```
class class_name {
   // Members
}
```

Access Specifiers

public

A public data member has universal accessibility. Data members in a class are public by default.

private

Private data members are accessible only within the class that defines these members.

protected

A protected data member is accessible by the members within the same class and also by the members of the sub classes.

Product Class

```
class Product
{
    protected name :string;
    protected price : number;
    constructor(name :string, price : number) {
        this.name  = name;
        this.price = price;
    }
    print():void {
        console.log(this.name);
        console.log(this.price);
    }
}
```

```
let p1 = new Product("iPhone7 Plus", 70000);
p1.print();
```

Inheritance in classes

- Keyword **extends** is used to implement inheritance - create a new class from an existing class.
- TypeScript does NOT support multiple inheritance.
- Super class is accessed using **super** keyword.

DiscountProduct Class

```
class DiscountProduct extends Product {
    protected discountRate : number;
    constructor(name:string, price:number, discountRate:number){
      super(name,price);
      this.discountRate = discountRate;
    }
    print():void {
      super.print();
      console.log(this.discountRate);
    }
    getNetPrice(): number {
      return this.price - this.price * this.discountRate / 100;
    }
}
```

Using DiscountProduct Class

```
let dp = new DiscountProduct("Dell Laptop",65000,20);
dp.print();
console.log("Net Price : " +  dp.getNetPrice());
```

Interface

- An interface contains a collection of methods, properties and events.
- Interface contains only declarations and implementing classes provide definition.
- Interfaces are TypeScript only constructs. They are not converted to JavaScript.

```
interface interfacename {
    // members
}
```

```
interface Person {
    name : string;
    age  : number;
    toString: () => string;
}
// Inheritance in Interface
interface Student extends Person {
   course : string;
}
```

Interface

```
function print(v : Person) {
    console.log(v.toString());
}

let p1 : Person = {
      name : "Richards",
      age : 40,
      toString : function()  {
          return this.name + ":" + this.age;
      }
    };

print(p1);
```

Interface

```
function print(v : Person) {
    console.log(v.toString());
}

let s1 : Student = {
     name : "Mark",
     age : 20 ,
     course : "Angular",
     toString : function() {
       return this.name + ":" + this.age + ":" + this.course;
     }
};

print(s1);
```

Duck-typing

- In duck-typing, two objects are considered to be of the same type if both share the same set of properties.
- Duck-typing verifies the presence of certain properties in the objects, rather than their actual type, to check their suitability.
- The TypeScript compiler implements the duck-typing system that allows object creation on the fly while keeping type safety.