



Introduction to JavaScript

Module Overview

This module introduces participants to the fundamentals of JavaScript, es6 and TypeScript.



Module Objective

At the end of this module, students should be able to:

- Create user interfaces and websites.
- Use basic JavaScript concepts
- use functions, arguments, strings
- Use object de-structuring
- Understand Typescript fundamentals
- use the typescript types, object and function types
- Understand the typescript OOPS concept



Introduction to JavaScript

What is JavaScript?

JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.

JavaScript was first known as LiveScript, but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name LiveScript. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers.

The ECMA-262 Specification defined a standard version of the core JavaScript language.

- JavaScript is a lightweight, interpreted programming language.

- Designed for creating network-centric applications.
- Complementary to and integrated with Java.
- Complementary to and integrated with HTML.
- Open and cross-platform

Why to Learn Javascript

- Javascript is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning Javascript:
- Javascript is the most popular programming language in the world and that makes it a programmer's great choice. Once you learnt Javascript, it helps you developing great front-end as well as back-end softwares using different Javascript based frameworks like jQuery, Node.JS etc.
- Javascript is everywhere, it comes installed on every modern web browser and so to learn Javascript you really do not need any special environment setup. For example Chrome, Mozilla Firefox , Safari and every browser you know as of today, supports Javascript.
- Javascript helps you create really beautiful and crazy fast websites. You can develop your website with a console like look and feel and give your users the best Graphical User Experience.
- JavaScript usage has now extended to mobile app development, desktop app development, and game development. This opens many opportunities for you as Javascript Programmer.
- Due to high demand, there is tons of job growth and high pay for those who know JavaScript. You can navigate over to different job sites to see what having JavaScript skills looks like in the job market.
- Great thing about Javascript is that you will find tons of frameworks and Libraries already developed which can be used directly in your software development to reduce your time to market.

Advantages of JavaScript

The merits of using JavaScript are –

- Less server interaction – You can validate user input before sending the page off to the server. This saves server traffic, which means less load on your server.
- Immediate feedback to the visitors – They don't have to wait for a page reload to see if they have forgotten to enter something.

- Increased interactivity – You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard.
- Richer interfaces – You can use JavaScript to include such items as drag-and-drop components and sliders to give a Rich Interface to your site visitors.

Limitations of JavaScript

We cannot treat JavaScript as a full-fledged programming language. It lacks the following important features –

- Client-side JavaScript does not allow the reading or writing of files. This has been kept for security reason.
- JavaScript cannot be used for networking applications because there is no such support available.
- JavaScript doesn't have any multi-threading or multiprocessor capabilities.

Once again, JavaScript is a lightweight, interpreted programming language that allows you to build interactivity into otherwise static HTML pages.



JavaScript Basics

JavaScript can be implemented using JavaScript statements that are placed within the `<script>... </script>` HTML tags in a web page. You can place the `<script>` tags, containing your JavaScript, anywhere within your web page, but it is normally recommended that you should keep it within the `<head>` tags.

The `<script>` tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of your JavaScript will appear as follows.

```
<script...>
```

```
JavaScript code
```

```
</script>
```

The script tag takes two important attributes –

- **Language** – This attribute specifies what scripting language you are using. Typically, its value will be javascript.

Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.

- **Type** – This attribute is what is now recommended to indicate the scripting language in use and its value should be set to "text/javascript".

So, your JavaScript segment will look like –

```
<script type="text/javascript">
document.write("JavaScript is a simple language for javatpoint learners");
</script>
```

- The script tag specifies that we are using JavaScript.
- The text/javascript is the content type that provides information to the browser about the data.
- The document.write() function is used to display dynamic content through JavaScript. We will learn about document object in detail later.

JavaScript versions

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997. ECMAScript is the official name of the language. ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6. Since 2016 new versions are named by year (ECMAScript 2016 / 2017 / 2018).

Version	Official Name	Description
ES1	ECMAScript 1 (1997)	First edition
ES2	ECMAScript 2 (1998)	Editorial changes
ES3	ECMAScript 3 (1999)	Added regular expressions & try/catch
ES4	ECMAScript 4	Not released
ES5	ECMAScript 5 (2009)	Added "strict mode", JSON support, String.trim(), Array.isArray(), & Array iteration methods.
ES6	ECMAScript 2015	Added let and const, default parameter values, Array.find(), & Array.findIndex()
ES6	ECMAScript 2016	Added exponential operator & Array.prototype.includes
ES6	ECMAScript 2017	Added string padding, Object.entries, Object.values, async functions, & shared memory
ES6	ECMAScript 2018	Added rest / spread properties, asynchronous iteration, Promise.finally(), & RegExp

JavaScript Keywords

JavaScript statements often start with a keyword to identify the JavaScript action to be performed. Our Reserved Words Reference lists all JavaScript keywords. Here is a list of some of the keywords:

Keyword	Description
var	Declares a variable
let	Declares a block variable
const	Declares a block constant
if	Marks a block of statements to be executed on a condition
switch	Marks a block of statements to be executed in different cases
for	Marks a block of statements to be executed in a loop
function	Declares a function
return	Exits a function
try	Implements error handling to a block of statements

JavaScript Variables

Like many other programming languages, JavaScript has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the var keyword as follows.

```
<script type = "text/javascript">
  <!--
    var money;
    var name;
  //-->
</script>
```

You can also declare multiple variables with the same var keyword as follows –

```
<script type = "text/javascript">
  <!--
    var money, name;
  //-->
</script>
```

Storing a value in a variable is called variable initialization. You can do variable initialization at the time of variable creation or at a later point in time when you need that variable.

For instance, you might create a variable named money and assign the value 2000.50 to it later. For another variable, you can assign a value at the time of initialization as follows.

```
<script type = "text/javascript">
  <!--
    var name = "Ali";
    var money;
    money = 2000.50;
  //-->
</script>
```

JavaScript Datatypes

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

JavaScript allows you to work with three primitive data types –

- Numbers, eg. 123, 120.50 etc.
- Strings of text e.g. "This text string" etc.
- Boolean e.g. true or false.

JavaScript also defines two trivial data types, null and undefined, each of which defines only a single value. In addition to these primitive data types, JavaScript supports a composite data type known as object. We will cover objects in detail in a separate chapter.

Note – JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.



JavaScript Arrow Functions

Arrow functions are introduced in ES6, which provides you a more accurate way to write the functions in JavaScript. They allow us to write smaller function syntax. Arrow functions make your code more readable and structured.

Arrow functions are anonymous functions (the functions without a name and not bound with an identifier). They don't return any value and can declare without the function keyword. Arrow functions cannot be used as the constructors. The context within the arrow functions is lexically or statically defined. They are also called as Lambda Functions in different languages.

Arrow functions do not include any prototype property, and they cannot be used with the new keyword.

Syntax for defining the arrow function:

```
const functionName = (arg1, arg2, ?..) => {  
  //body of the function
```

There are three parts to an Arrow Function or Lambda Function:

- Parameters: Any function may optionally have the parameters.
- Fat arrow notation/lambda notation: It is the notation for the arrow (=>).
- Statements: It represents the instruction set of the function.

Let us try to understand it with an example.

In the following example, we are defining three functions that show Function Expression, Anonymous Function, and Arrow Function.

```
// function expression
var myfun1 = function show() {
  console.log("It is a Function Expression");
}

// Anonymous function
var myfun2 = function () {
  console.log("It is an Anonymous Function");
}

//Arrow function
var myfun3 = () => {
  console.log("It is an Arrow Function");
};

myfun1();
myfun2();
myfun3();
```

Output:

```
It is a Function Expression
It is an Anonymous Function
It is an Arrow Function
```

Syntactic Variations

There are some syntactic variations for the arrow functions that are as follows:

- **Optional parentheses for the single parameter**

```
var num = x => {
  console.log(x);
}

num(140);
```


Output:

```
140
```

- **Optional braces for single statement and the empty braces if there are no parameters required.**

```
var show = () => console.log("Hello World");  
show(); }
```

Output:

```
Hello World
```

Arrow Function with Parameters

If you require to pass more than one parameter by using an arrow function, then you have to pass them within the parentheses

Example:

```
var show = (a,b,c) => {  
  console.log(a + " " + b + " " + c );  
}  
show(100,200,300);
```

Output:

```
100 200 300
```

Arrow Function with default parameters

In ES6, the function allows the initialization of parameters with default values, if there is no value passed to it, or it is undefined. You can see the illustration for the same in the following code:

```
var show = (a, b=200) => {  
  console.log(a + " " + b);  
}  
show(100);
```

In the above function, the value of b is set to 200 by default. The function will always consider 200 as the value of b if no value of b is explicitly passed.

Output:

```
100 200
```

The default value of the parameter 'b' will get overwritten if the function passes its value explicitly. You can see it in the following example:

Example:

```
var show = (a, b=200) => {  
  console.log(a + " " + b);  
}  
show(100,500);
```

Output:

```
100 500
```

Arrow Function with Rest parameters

Rest parameters do not restrict you to pass the number of values in a function, but all the passed values must be of the same type. We can also say that rest parameters act as the placeholders for multiple arguments of the same type.

For declaring the rest parameter, the parameter name should be prefixed with the spread operator that has three periods (not more than three or not less than three).

You can see the illustration for the same in the following example:

```
var show = (a, ...args) => {  
  console.log(a + " " + args);  
}  
show(100,200,300,400,500,600,700,800);
```

Output:

```
100 200,300,400,500,600,700,800
```

Arrow Function without Parentheses

If you have a single parameter to pass, then the parentheses are optional.

Example:

```
var show = x => {  
  console.log(x);  
}  
show("Hello World");
```

Output:

```
Hello World
```



Activity One

Follow the steps given above for all examples of JavaScript arrow functions and execute all programs.



ES6 Strings

JavaScript string is an object which represents the sequence of characters. Generally, strings are used to hold text-based values such as a person name or a product description.

In JavaScript, any text within the single or double quotes is considered as a string. There are two ways for creating a string in JavaScript:

- By using a string literal
- By using string object (using the new keyword)

Let us elaborate both ways of creating a string in JavaScript.

- **By using a string literal**

The string literals can be created by using either double quotes or by using single quotes. The syntax for creating string literal is given below:

```
var stringname = "string value";
```

- **By using String object (using the new keyword)**

Here, we will use a new keyword for creating the string object. The syntax for creating the string object is given below:

```
var stringname = new String ("string literal");
```

String Properties

There are some properties of the string that are tabulated as follows:

S.no.	Property	Description
1.	constructor	It returns the constructor function for an object.
2.	length	It returns the length of the string.
3.	prototype	It allows us to add the methods and properties to an existing object.

- **JavaScript string constructor property**

The constructor property returns the constructor function for an object. Instead of the name of the function, it returns

the reference of the function.

Syntax:

```
string.constructor
```

Example:

```
var str = new String("Hello World");  
console.log("Value of str.constructor is: "+str.constructor);
```

Output:

```
Value of str.constructor is: function String() { [native code] }
```

- **JavaScript string length property**

As its name implies, this property returns the number of characters or the length of the string.

Syntax:

```
string.length
```

Example:

```
var str = new String("Hello World");  
console.log("The number of characters in the string str is: "+str.length);
```

Output:

```
The number of characters in the string str is: 11
```

- **JavaScript string prototype property**

It allows us to add new methods and properties in an existing object type. It is a global property which is available with almost all the objects of JavaScript.

Syntax:

```
object.prototype.name = value;
```

Example:

```
function student(name, qualification){  
  this.name = name;  
  this.qualification = qualification;  
}  
student.prototype.age = 20;  
var stu = new student('Daniel Grint' , 'BCA');  
console.log(stu.name);  
console.log(stu.qualification);  
console.log(stu.age);
```

Output:

```
Daniel Grint  
BCA  
20
```

String Methods

There are four string functions available in ES6, which are tabulated as follows:

S.no.	Methods	Description
1.	startsWith	It determines whether a string begins with the characters of a specified string.
2.	endsWith	It determines whether a string ends with the characters of a specified string.
3.	includes	It returns true if the specified argument is in the string.
4.	repeat	It returns a new string repeated based on specified count arguments.

startsWith() method

It is a case-sensitive method, which determines whether the string begins with the specified string characters or not. It

returns true if the string begins with the characters and returns false if not.

Syntax:

```
string.startsWith(searchValue, startPosition)
```

This method includes two parameters that are as follows:

- **searchValue:** It is the required parameter of this method. It includes the characters to be searched for at the start of the string.
- **startPosition:** It is an optional parameter. Its default value is 0. It specifies the position in the string at which to begin searching.

Example:

```
var str = 'Welcome to javaTpoint :);  
console.log(str.startsWith('Wel',0));  
console.log(str.startsWith('wel',0));
```

Output:

```
true  
false
```

endsWith() method

It is also a case-sensitive method that determines whether a string ends with the characters of the specified string or not.

Syntax:

```
string.endsWith(searchvalue, length)
```

The parameters of this method are defined as follows:

- **searchValue:** It is the required parameter that represents the characters to be searched at the end of the string.
- **length:** It is an optional parameter. It is the length of the string that will be searched. If this parameter is omitted, then the method will search in the full length of the string.

Example:

```
var str = "Welcome to Edubridge.";
console.log(str.endsWith("to", 10))
console.log(str.endsWith("To", 10))
```

Output:

```
true
false
```

includes() method

It is a case-sensitive method that determines whether the string contains the characters of the specified string or not. It returns true if the string contains characters and returns false if not.

Syntax:

```
string.includes(searchValue, start)
```

Let's understand the parameters of this method.

- **searchValue:** It is a required parameter. It is the substring to search for.
- **start:** It represents the position where to start the searching in the string. Its default value is **0**.

Example:

```
let str = "hello world"
console.log(str.includes('world',5));
console.log(str.includes('World', 11))
```

Output:

```
true
false
```


repeat() method

It is used to build a new string that contains a specified number of copies of the string on which this method has been called.

Syntax:

```
string.repeat(count)
```

This function has a single argument.

- **count:** It is a required parameter that shows the number of times to repeat the given string. The range of this parameter is from 0 to infinity.

Example:

```
var str = "hello world";  
console.log(str.repeat(5));
```

Output:

```
hello worldhello worldhello worldhello worldhello world
```

ES6 Template Literals

Template literals are a new feature introduced in ECMAScript 2015/ ES6. It provides an easy way to create multiline strings and perform string interpolation. Template literals are the string literals and allow embedded expressions.

Before ES6, template literals were called as **template strings**. Unlike quotes in strings, template literals are enclosed by the **backtick** (```) character (key below the **ESC** key in QWERTY keyboard). Template literals can contain placeholders, which are indicated by the dollar sign and curly braces (**`${expression}`**). Inside the backticks, if we want to use an expression, then we can place that expression in the (**`${expression}`**).

Syntax:

```
var str = `string value`;
```

Multiline strings

In normal strings, we have to use an escape sequence `\n` to give a new line for creating a multiline string. However, in template literals, there is no need to use `\n` because string ends only when it gets backtick(```) character.

Let us try to understand it with the following example.

Example:

```
// Without template literal
console.log('Without template literal \n multiline string');

// With template literal
console.log(`Using template literal
multiline string`);
```

Output:

```
Without template literal
multiline string
Using template literal
multiline string
```

String Interpolation

ES6 template literals support string interpolation. Template literals can use the placeholders for string substitution. To embed expressions with normal strings, we have to use the `${}` syntax.

Example 1:

```
var name = 'World';
var cname = 'javaTpoint';
console.log(`Hello, ${name}!
Welcome to ${cname}`);
```

Output:

```
Hello, World!
Welcome to javaTpoint
```

Example 2:

```
var x = 10;

var y = 20;

console.log(`The product of the variables ${x} and ${y} is:

${x*y}`);
```

Output:

```
The product of the variables 10 and 20 is:
200
```

Tagged templates

Tagged templates are one of the more advanced form of template literals. Tagged template literals allow us to parse template literals with a function.

The first argument of the tag function contains an array having string values, and the remaining arguments are related to the expression. The writing of tagged literal is similar to the function definition, but the difference occurs when the tagged literals get called. There are no parentheses () required to call a literal. Let us see the illustration for the tagged templates.

Example 1:

```
function TaggedLiteral(str) {

  console.log(str);

}

TaggedLiteral `Hello World`;
```

Output:

```
[ 'Hello World' ]
```

Example 2:

We can also pass the values in a tagged literal. The value can be the result of some expression or the value fetched from the variable. We can see the illustration for the same in the following code:

```
function TaggedLiteral(str, val1, val2) {  
  console.log(str);  
  console.log(val1+" "+val2);  
}  
  
let text = 'Hello World';  
  
TaggedLiteral`Colors ${text} ${10+30}`;
```

Output:

```
[ 'Colors ', ' ', '' ]  
Hello World    40
```

Raw Strings

The template literal raw method allows the accessing of raw strings as they were entered. In addition to this, the `string.raw()` method exists for creating the raw strings as similar to the default template function. It allows us to write the backslashes as we would in a regular expression literal.

The `string.raw()` method is used to show the strings without any interpretation of backslashed characters. It is also useful to print windows sub-directory locations without require to use a lot of backslashes.

Example:

```
var rawText = String.raw`Hello \n World \n Welcome back!`  
  
console.log(rawText)
```

Output:

```
Hello \n World \n Welcome back!
```

String.fromCodePoint()

This method returns a string, which is created by using the specified sequence of Unicode code points. It throws a `RangeError` if there is an invalid code point is passed.

Example:

```
console.log(String.fromCodePoint(49))  
console.log(String.fromCodePoint(80, 76))
```

Output:

```
1  
PL
```



Activity Two

1. Write a JavaScript function to check whether a string is blank or not. Go to the editor

Test Data :

```
console.log(is_Blank(''));  
console.log(is_Blank('abc'));  
true  
false
```

2. Write a program using JavaScript string prototype property and the display the output as shown below:

```
Neha Tandon  
BE  
24
```

3. Using String methods and template literals, write a program to display the output as shown below:

```
My name is, Rohan!  
Welcome to Edubridge  
hello Rohanhello Rohanhello Rohanhello Rohanhello Rohan
```



Object De-structuring

It is similar to array de-structuring except that instead of values being pulled out of an array, the properties (or keys) and their corresponding values can be pulled out from an object.

When de-structuring the objects, we use keys as the name of the variable. The variable name must match the property (or keys) name of the object. If it does not match, then it receives an undefined value. This is how JavaScript knows which property of the object we want to assign.

In object de-structuring, the values are extracted by the keys instead of position (or index).

First, try to understand the basic assignment in object de-structuring by using the following example.

Example Simple Assignment:

```
const num = {x: 100, y: 200};  
const {x, y} = num;  
console.log(x); // 100  
console.log(y); // 200
```

Output:

```
100  
200
```

Let us understand the basic object de-structuring assignment.

Example - Basic Object de-structuring assignment

```
const student = {name: 'Arun', position: 'First', rollno: '24'};  
const {name, position, rollno} = student;  
console.log(name); // Arun  
console.log(position); // First
```

Output:

```
Arun  
First  
24
```

Object de-structuring and default values

Like array de-structuring, a default value can be assigned to the variable if the value unpacked from the object is **undefined**. It can be clear from the following example.

Example:

```
const {x = 100, y = 200} = {x: 500};  
console.log(x); // 500  
console.log(y); // 200
```

In the above example, the variables **x** and **y** have default values **100** and **200**. Then, the variable **x** is reassigned with a value of **500**. But the variable **y** is not reassigned, so it retains its original value. So, instead of getting **100** and **200** in output, we will get **500** and **200**.

Output:

```
500  
200
```

Assigning new variable names

We can assign a variable with a different name than the property of the object. You can see the illustration for the same as follows:

Example:

```
const num = {x: 100, y: 200};  
const {x: new1, y: new2} = num;  
  
console.log(new1); //100
```

In the above example, we have assigned the property name as x and y to a local variable, new1, and new2.

Output:

```
100
200
```

Assignment without declaration

if the value of the variable is not assigned when you declare it, then you can assign its value during destructuring. You can see the illustration for the same as follows:

Example:

```
let name, division;
({name, division} = {name: 'Anil', division: 'First'});
console.log(name); // Anil
console.log(division); // First
```

In the above example, it is to be noted that the use of parentheses () around the assignment statement is mandatory when using variable destructuring assignment without a declaration. Otherwise, the syntax will be invalid.

Output:

```
Anil
First
```

Object de-structuring and rest operator

By using the rest operator (...) in object destructuring, we can put all the remaining keys of an object in a new object variable.

Example:

```
let {a, b, ...args} = {a: 100, b: 200, c: 300, d: 400, e: 500}
console.log(a);
console.log(b);
console.log(args);
```


Output:

```
100
200
{ c: 300, d: 400, e: 500 }
```

Assigning new variable names and providing default values simultaneously

A property that is unpacked from an object can be assigned to a variable with a different name. And the default value is assigned to it in case the unpacked value is undefined.

Example:

```
const {a:num1=100, b:num2=200} = {a:300};
console.log(num1); //300
console.log(num2); //200
```

Output:

```
300
200
```

**Activity Three**

Follow the steps given above for all examples of object de-structuring and execute all programs.



Spread and Rest operator

Spread operator: The spread operator helps us expand an iterable such as an array where multiple arguments are needed, it also helps to expand the object expressions. In cases where we require all the elements of an iterable or object to help us achieve a task, we use a spread operator.

Syntax:

```
var var_name = [...iterable];
```

Rest operator: The rest parameter is converse to the spread operator. while spread operator expands elements of an iterable, rest operator compresses them. It collects several elements. In functions when we require to pass arguments but were not sure how many we have to pass, the rest parameter makes it easier.

Syntax:

```
function function_name(...arguments) {
```

Let's understand with examples.

Example 1: In the example below two arrays are defined and they're merged into one using the spread operator (...). The merged array contains elements in the order they're merged.

```
<script>
  var array1 = [10, 20, 30, 40, 50];
  var array2 = [60, 70, 80, 90, 100];
  var array3 = [...array1, ...array2];
  console.log(array3);
</script>
```

Output:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Example 2: In this example, appending an element to a given iterable. An array is defined and a value needs to be appended to it, so we use the spread operator to spread all the values of the iterable and then add the elements before or after according to the order we want.

```
<script>
  var array1 = [10, 20, 30, 40, 50];
  var array2 = [...array1, 60];
  console.log(array2);
</script>
```

Output:

```
[10, 20, 30, 40, 50, 60]
```

Example 3: In this example, we will copy objects using the spread operator. obj2 is given all the properties of obj1 using the spread operator(...). curly brackets must be used to specify that object is being copied or else an error gets raised.

```
<script>
  const obj = {
    firstname: "Divit",
    lastname: "Patidar",
  };
  const obj2 = { ...obj };
  console.log(obj2);
</script>
```

Output:

```
{
  firstname: "Divit",
```

```
    lastname: "Patidar"  
}
```

Example 4: In this example, rest parameter condensing multiple elements into a single element even when different numbers of parameters are passed into the function, the function works as we used the rest parameter. it takes multiple elements as arguments and compresses them into a single element or iterable. further operations are performed on the iterable.

```
<script>  
function average(...args) {  
    console.log(args);  
    var avg =  
        args.reduce(function (a, b) {  
            return a + b;  
        }, 0) / args.length;  
    return avg;  
}  
  
console.log("average of numbers is : "  
    + average(1, 2, 3, 4, 5));  
console.log("average of numbers is : "  
    + average(1, 2, 3));  
</script>
```

Output:

```
[1, 2, 3, 4, 5]  
"average of numbers is : 3"  
[1, 2, 3]  
"average of numbers is : 2"
```



Activity Four

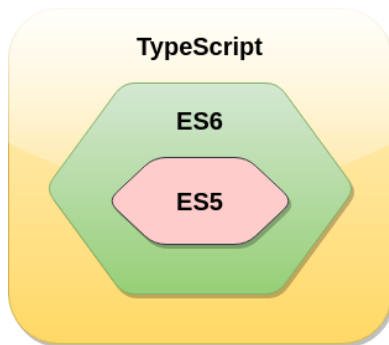
Follow the steps given above and solve all 4 examples and execute the programs.



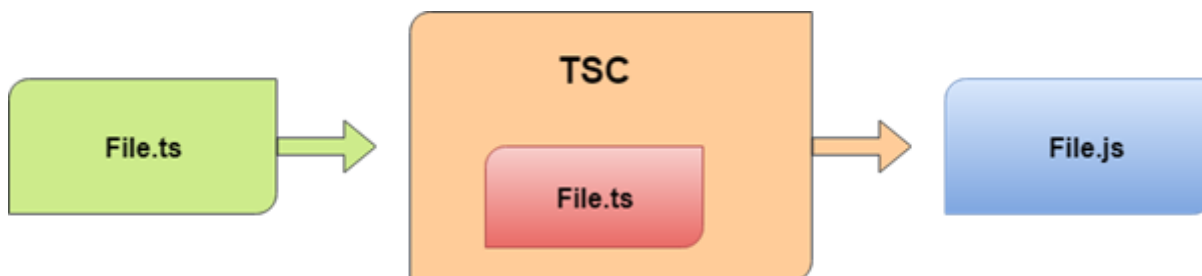
TypeScript Fundamentals

What is TypeScript?

TypeScript is an open-source pure object-oriented programming language. It is a strongly typed superset of JavaScript which compiles to plain JavaScript. It contains all elements of the JavaScript. It is a language designed for large-scale JavaScript application development, which can be executed on any browser, any Host, and any Operating System. The TypeScript is a language as well as a set of tools. TypeScript is the ES6 version of JavaScript with some additional features.



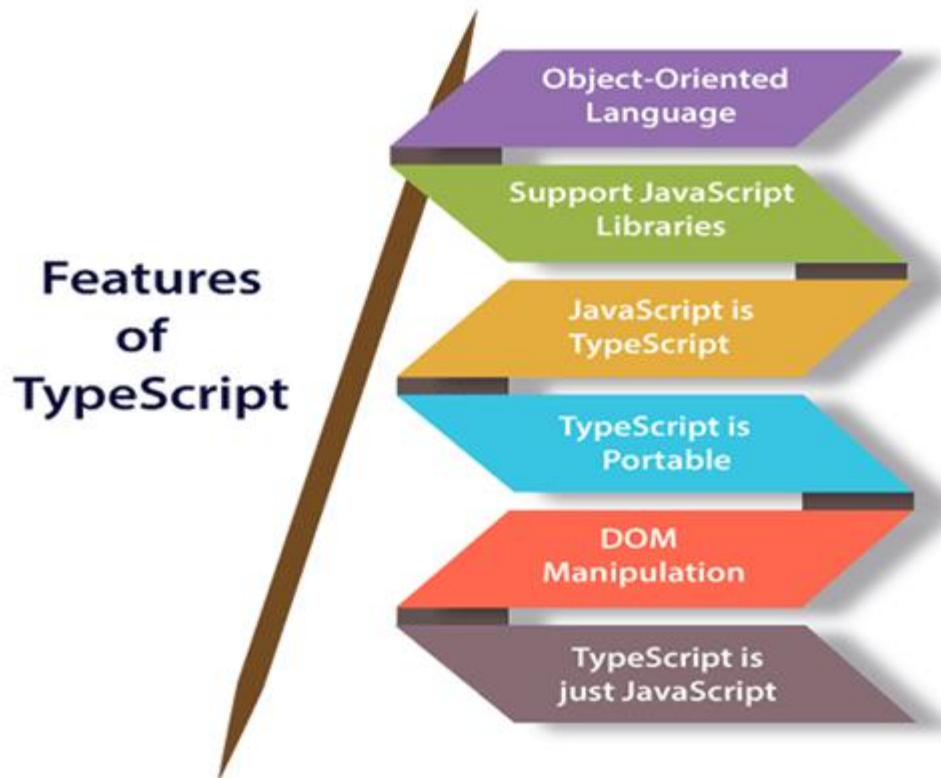
TypeScript cannot run directly on the browser. It needs a compiler to compile the file and generate it in JavaScript file, which can run directly on the browser. The TypeScript source file is in ".ts" extension. We can use any valid ".js" file by renaming it to ".ts" file. TypeScript uses TSC (TypeScript Compiler) compiler, which convert Typescript code (.ts file) to JavaScript (.js file).



Difference between JavaScript and Typescript

SN	JavaScript	TypeScript
1.	It doesn't support strongly typed or static typing.	It supports strongly typed or static typing feature.
2.	Netscape developed it in 1995.	Anders Hejlsberg developed it in 2012.
3.	JavaScript source file is in ".js" extension.	TypeScript source file is in ".ts" extension.
4.	It is directly run on the browser.	It is not directly run on the browser.
5.	It is just a scripting language.	It supports object-oriented programming concept like classes, interfaces, inheritance, generics, etc.
6.	It doesn't support optional parameters.	It supports optional parameters.
7.	It is interpreted language that's why it highlighted the errors at runtime.	It compiles the code and highlighted errors during the development time.
8.	JavaScript doesn't support modules.	TypeScript gives support for modules.
9.	In this, number, string are the objects.	In this, number, string are the interface.
10.	JavaScript doesn't support generics.	TypeScript supports generics.
11.	Example: <pre><script> function addNumbers(a, b) { return a + b; } var sum = addNumbers(15, 25); document.write('Sum of the numbers is: ' + sum); </script></pre>	Example: <pre>function addNumbers(a, b) { return a + b; } var sum = addNumbers(15, 25); console.log('Sum of the numbers is: ' + sum);</pre>

TypeScript Features



- **Object-Oriented language:** TypeScript provides a complete feature of an object-oriented programming language such as classes, interfaces, inheritance, modules, etc. In TypeScript, we can write code for both client-side as well as server-side development.
- **TypeScript supports JavaScript libraries:** TypeScript supports each JavaScript elements. It allows the developers to use existing JavaScript code with the TypeScript. Here, we can use all of the JavaScript frameworks, tools, and other libraries easily.
- **JavaScript is TypeScript:** It means the code written in JavaScript with valid .js extension can be converted to TypeScript by changing the extension from .js to .ts and compiled with other TypeScript files.
- **TypeScript is portable:** TypeScript is portable because it can be executed on any browsers, devices, or any operating systems. It can be run in any environment where JavaScript runs on. It is not specific to any virtual-machine for execution.
- **DOM Manipulation:** TypeScript can be used to manipulate the DOM for adding or removing elements similar to JavaScript.
- **TypeScript is just a JS:** TypeScript code is not executed on any browsers directly. The program written in TypeScript always starts with JavaScript and ends with JavaScript. Hence, we only need to know JavaScript to use it in TypeScript. The code written in TypeScript is compiled and converted into its JavaScript equivalent for the execution. This process is known as Trans-plied. With the help of JavaScript code, browsers can read the TypeScript code and display the output.

Components of TypeScript

The TypeScript language is internally divided into three main layers. Each of these layers is divided into sublayers or components. In the following diagram, we can see the three layers and each of their internal components. These layers are:

- Language
- The TypeScript Compiler
- The TypeScript Language Services

1. Language

It features the TypeScript language elements. It comprises elements like syntax, keywords, and type annotations.

2. The TypeScript Compiler

The TypeScript compiler (TSC) transform the TypeScript program equivalent to its JavaScript code. It also performs the parsing, and type checking of our TypeScript code to JavaScript code.

Browser doesn't support the execution of TypeScript code directly. So the program written in TypeScript must be re-written in JavaScript equivalent code which supports the execution of code in the browser directly. To perform this, TypeScript comes with TypeScript compiler named "tsc." The current version of TypeScript compiler supports ES6, by default. It compiles the source code in any module like ES6, SystemJS, AMD, etc.

We can install the TypeScript compiler by locally, globally, or both with any npm package. Once installation completes, we can compile the TypeScript file by running "tsc" command on the command line.

3. The TypeScript Language Services

The language service provides information which helps editors and other tools to give better assistance features such as automated refactoring and IntelliSense. It exposes an additional layer around the core-compiler pipeline. It supports some standard typical editor operations like code formatting and outlining, colorization, statement completion, signature help, etc.

TypeScript Variables

A variable is the storage location, which is used to store value/information to be referenced and used by programs. It acts as a container for value in code and must be declared before the use. We can declare a variable by using the var keyword. In TypeScript, the variable follows the same naming rule as of JavaScript variable declaration. These rules are-

- The variable name must be an alphabet or numeric digits.
- The variable name cannot start with digits.
- The variable name cannot contain spaces and special character, except the underscore(_) and the dollar(\$) sign.

In ES6, we can define variables using let and const keyword. These variables have similar syntax for variable declaration and initialization but differ in scope and usage. In TypeScript, there is always recommended to define a variable using let keyword because it provides the type safety.

The let keyword is similar to var keyword in some respects, and const is an let which prevents re-assignment to a variable.

Variable Declaration in TypeScript

The type syntax for declaring a variable in TypeScript is to include a colon (:) after the variable name, followed by its type. Just as in JavaScript, we use the var keyword to declare a variable.

When you declare a variable, you have four options –

1. Declare type and value in a single statement

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

2. Declare type without value. Then the variable will be set to undefined.

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

3. Declare its value without type. Then the variable will be set to any.

```
var [identifier] = value;
```

4. Declare without value and type. Then the variable will be set to any and initialized with undefined.

```
var [identifier];
```

Example: Variables in Typescript

```
var name:string = "Aditya";  
  
var score1:number = 47;  
  
var score2:number = 46.50  
  
var sum = score1 + score2  
  
console.log("name: "+name)  
  
console.log("first score: "+score1)  
  
console.log("second score: "+score2)  
  
console.log("sum of the scores: "+sum)
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10  
  
var name = "Aditya";  
  
var score1 = 47;  
  
var score2 = 46.50;  
  
var sum = score1 + score2;  
  
console.log("name: " + name);  
  
console.log("first score: " + score1);  
  
console.log("second score : " + score2);  
  
console.log("sum of the scores: " + sum);
```

The output of the above program is given below –

```
name: Aditya  
first score: 47  
second score: 46.5  
sum of the scores: 93.5
```

The TypeScript compiler will generate errors, if we attempt to assign a value to a variable that is not of the same type. Hence, TypeScript follows Strong Typing. The Strong typing syntax ensures that the types specified on either side of the assignment operator (=) are the same. This is why the following code will result in a compilation error –

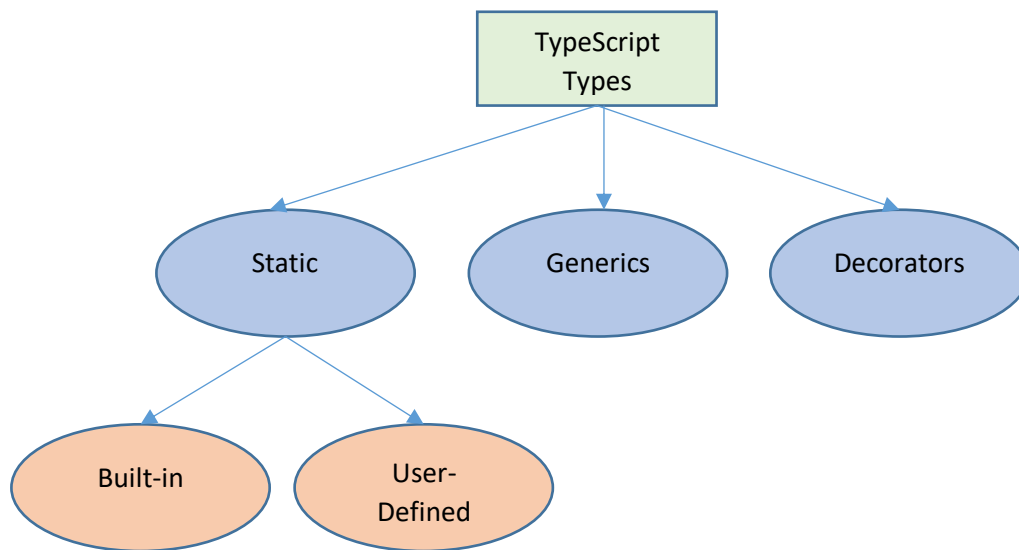
```
var num:number = "hello" // will result in a compilation error
```



TypeScript Types

The TypeScript language supports different types of values. It provides data types for the JavaScript to transform it into a strongly typed programming language. JavaScript doesn't support data types, but with the help of TypeScript, we can use the data types feature in JavaScript. TypeScript plays an important role when the object-oriented programmer wants to use the type feature in any scripting language or object-oriented programming language. The Type System checks the validity of the given values before the program uses them. It ensures that the code behaves as expected.

TypeScript provides data types as an optional Type System. We can classify the TypeScript data type as following.



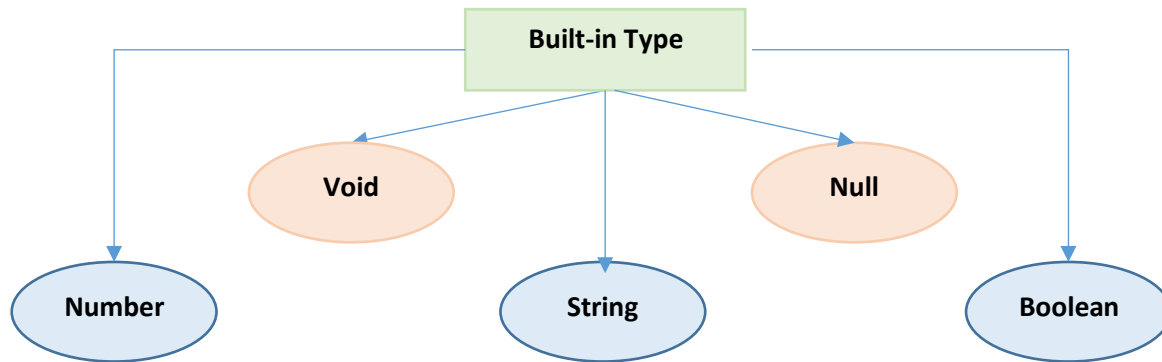
1. Static Types

In the context of type systems, static types mean "at compile time" or "without running a program." In a statically typed language, variables, parameters, and objects have types that the compiler knows at compile time. The compiler used this information to perform the type checking.

Static types can be further divided into two sub-categories:

Built-in or Primitive Type

The TypeScript has five built-in data types, which are given below.



- **Number**

Like JavaScript, all the numbers in TypeScript are stored as floating-point values. These numeric values are treated like a number data type. The numeric data type can be used to represent both integers and fractions. TypeScript also supports Binary(Base 2), Octal(Base 8), Decimal(Base 10), and Hexadecimal(Base 16) literals.

- **String**

We will use the string data type to represent the text in TypeScript. String type works with textual data. We include string literals in our scripts by enclosing them in single or double quotation marks. It also represents a sequence of Unicode characters. It embeds the expressions in the form of `$ {expr}`.

- **Boolean**

The string and numeric data types can have an unlimited number of different values, whereas the Boolean data type can have only two values. They are "true" and "false." A Boolean value is a truth value which specifies whether the condition is true or not.

- **Void**

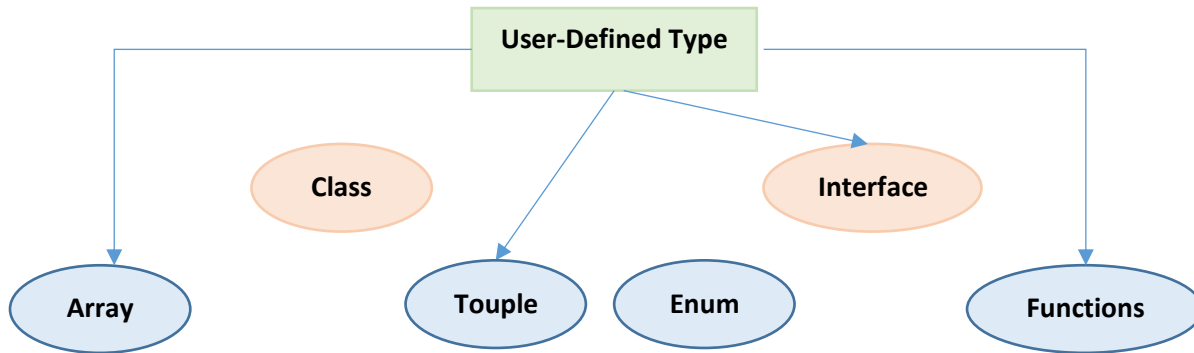
A void is a return type of the functions which do not return any type of value. It is used where no data type is available. A variable of type void is not useful because we can only assign undefined or null to them. An undefined data type denotes an uninitialized variable, whereas null represents a variable whose value is undefined.

- **Null**

Null represents a variable whose value is undefined. Much like the void, it is not extremely useful on its own. The Null accepts the only one value, which is null. The Null keyword is used to define the Null type in TypeScript, but it is not useful because we can only assign a null value to it.

User-Defined Type

TypeScript supports the following user-defined data types:



- **Array**

An array is a collection of elements of the same data type.

- **Touple**

The Tuple is a data type which includes two sets of values of different data types. It allows us to express an array where the type of a fixed number of elements is known, but they are not the same.

- **Interface**

An Interface is a structure which acts as a contract in our application. It defines the syntax for classes to follow, means a class which implements an interface is bound to implement all its members. It cannot be instantiated but can be referenced by the class which implements it. The TypeScript compiler uses interface for type-checking that is also known as "duck typing" or "structural subtyping."

- **Class**

Classes are used to create reusable components and acts as a template for creating objects. It is a logical entity which store variables and functions to perform operations. TypeScript gets support for classes from ES6. It is different from the interface which has an implementation inside it, whereas an interface does not have any implementation inside it.

- **Enums**

Enums define a set of named constant. TypeScript provides both string-based and numeric-based enums. By default, enums begin numbering their elements starting from 0, but we can also change this by manually setting the value to one of its elements. TypeScript gets support for enums from ES6.

- **Functions**

A function is the logical blocks of code to organize the program. Like JavaScript, TypeScript can also be used to create functions either as a named function or as an anonymous function. Functions ensure that our program is readable, maintainable, and reusable. A function declaration has a function's name, return type, and parameters.

2. Generic

Generic is used to create a component which can work with a variety of data type rather than a single one. It allows a way to create reusable components. It ensures that the program is flexible as well as scalable in the long term. TypeScript uses generics with the type variable `<T>` that denotes types. The type of generic functions is just like non-generic functions, with the type parameters listed first, similarly to function declarations.

Example:

```
function identity<T>(arg: T): T {  
    return arg;  
}  
  
let output1 = identity<string>("myString");  
let output2 = identity<number>( 100 );
```

3. Decorators

A decorator is a special of data type which can be attached to a class declaration, method, property, accessor, and parameter. It provides a way to add both annotations and a meta-programming syntax for classes and functions. It is used with "@" symbol.

A decorator is an experimental feature which may change in future releases. To enable support for the decorator, we must enable the `experimentalDecorators` compiler option either on the command line or in our `tsconfig.json`.

Example:

```
function f() {  
    console.log("f(): evaluated");  
    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {  
        console.log("f(): called");  
    }  
}  
  
class C {  
    @f()  
    method() {}  
}
```

**TypeScript Type Assertion**

In TypeScript, type assertion is a mechanism which tells the compiler about the type of a variable. When TypeScript determines that the assignment is invalid, then we have an option to override the type using a type assertion. If we use a type assertion, the assignment is always valid, so we need to be sure that we are right. Otherwise, our program may not work correctly.

Type assertion is explicitly telling the compiler that we want to treat the entity as a different type. It allows us to treat any as a number, or number as a string. Type assertion is commonly used when we are migrating over code from JavaScript to TypeScript.

Type assertion works like typecasting, but it does not perform type checking or restructuring of data just like other languages can do like C# and Java. The typecasting comes with runtime support, whereas type assertion has no impact on runtime. However, type assertions are purely a compile-time construct and provide hints to the compiler on how we want our code to be analyzed.

Example:

```
let empCode: any = 111;

let employeeCode = <number> code;

console.log(typeof(employeeCode)); //Output: number
```

In the above example, we have declared a variable `empCode` as of type `any`. In the next line, we assign a value of this variable to another variable named `employeeCode`. Here, we know that `empCode` is of type `number`, even though we declared it as `'any.'` when we are assigning `empCode` to `employeeCode`, we have asserted that `empCode` is of type `number`. Now the type of `employeeCode` is `number`.

TypeScript provides two ways to do Type Assertion. They are

- Using Angular Bracket `<>`
- Using as keyword

Using Angular Bracket `<>`

In TypeScript, we can use angular "bracket `<>`" to show Type Assertion.

Example:

```
let empCode: any = 111;

let employeeCode = <number> code;
```

Using as Keyword

TypeScript provides another way to show Type Assertion by using `"as"` keyword.

Example:

```
let empCode: any = 111;

let employeeCode = code as number;
```

Type Assertion with object

Sometimes, we might have a situation where we have an object which is declared without any properties. For this, the compiler gives an error. But, by using type assertion we can avoid this situation. We can understand it with the following example.

Example:

```
let student = { };  
  
student.name = "Rohit"; //Compiler Error: Property 'name' doesn't exist on type '{}'  
  
student.code = 123; //Compiler Error: Property 'code' doesn't exist on type '{}'
```

In the above example, we will get a compilation error, because the compiler assumes that the type of student is {} without properties. We can avoid this situation by using a type assertion, which can be shown below.

```
interface Student {  
  name: string;  
  code: number;  
}  
  
let student = <Student> { };  
student.name = "Rohit"; // Correct  
student.code = 123; // Correct
```

In the above example, we have created an interface Student with the properties name and code. Then, we used type assertion on the student, which is the correct way to use type assertion.



Object Types

An object is an instance which contains set of key value pairs. The values can be scalar values or functions or even array of other objects. The syntax is given below:

Syntax:

```
var object_name = {  
  key1: "value1", //scalar value  
  key2: "value",  
  key3: function() {  
    //functions  
  },  
  key4:["content1", "content2"] //collection  
};
```

As shown above, an object can contain scalar values, functions and structures like arrays and tuples.

Example: Object Literal Notation

```
var person = {  
    firstname:"Tom",  
    lastname:"Hanks"  
};  
  
//access the object values  
console.log(person.firstname)  
console.log(person.lastname)
```

On compiling, it will generate the same code in JavaScript.

The output of the above code is as follows –

```
PS C:\Users\Tanaya Deshpande\Desktop\Typescript> Node firstprogram.js  
Tom  
Hanks
```

TypeScript Type Template

Let's say you created an object literal in JavaScript as –

```
var person = {  
    firstname:"Tom",  
    lastname:"Hanks"  
};
```

In case you want to add some value to an object, JavaScript allows you to make the necessary modification. Suppose we need to add a function to the person object later this is the way you can do this.

```
person.sayHello = function(){ return "hello";}
```

If you use the same code in Typescript the compiler gives an error. This is because in Typescript, concrete objects should have a type template. Objects in Typescript must be an instance of a particular type.

You can solve this by using a method template in declaration.

Example: Typescript Type template

```
var person = {  
  firstName:"Vilas",  
  lastName:"Deshmukh",  
  sayHello:function() { } //Type template  
}  
person.sayHello = function() {  
  console.log("hello "+person.firstName)  
}  
person.sayHello()
```

On compiling, it will generate the same code in JavaScript.

The output of the above code is as follows –

```
PS C:\Users\tanaya.Deshpande\Desktop\Typescript> Node firstprogram.js  
hello Vilas
```

Objects can also be passed as parameters to function.

Example: Objects as function parameters

```
var person = {  
  firstname:"Tom",  
  lastname:"Hanks"  
};  
var invokeperson = function(obj: { firstname:string, lastname :string }) {  
  console.log("first name :"+obj.firstname)  
  console.log("last name :"+obj.lastname)  
}  
invokeperson(person)
```

The example declares an object literal. The function expression is invoked passing person object.

On compiling, it will generate following JavaScript code.

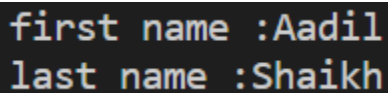
```
//Generated by typescript 1.8.10

var person = {
  firstname: "Aadil",
  lastname: "Shaikh"
};

var invokeperson = function (obj) {
  console.log("first name :" + obj.firstname);
  console.log("last name :" + obj.lastname);
};

invokeperson(person);
```

Its output is as follows –



You can create and pass an anonymous object on the fly.

Example: Anonymous Object

```
var invokeperson = function(obj:{ firstname:string, lastname :string}) {
  console.log("first name :"+obj.firstname)
  console.log("last name :"+obj.lastname)
}

invokeperson({firstname:"Sachin",lastname:"Tendulkar"});
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var invokeperson = function (obj) {
    console.log("first name :" + obj.firstname);
    console.log("last name :" + obj.lastname);
};
invokeperson({ firstname: "Sachin", lastname: "Tendulkar" });
```

Its output is as follows –

```
first name :Sachin
last name :Tendulkar
```



Activity Five

1. Refer to all the topics and examples given above for the following:
 - Object Literal Notation
 - TypeScript Type Template
 - Objects as function parameters
 - Anonymous Object
2. Follow the steps to perform and execute the programs.



TypeScript Functions

Functions are the most crucial aspect of JavaScript as it is a functional programming language. Functions are pieces of code that execute specified tasks. They are used to implement object-oriented programming principles like classes, objects, polymorphism, and abstraction. It is used to ensure the reusability and maintainability of the program. Although TypeScript has the idea of classes and modules, functions are still an important aspect of the language.

Function declaration: The name, parameters, and return type of a function are all specified in a function declaration. The function declaration has the following:

Syntax:

```
function functionName(arg1, arg2, ... , argN);
```

Function definition: It includes the actual statements that will be executed. It outlines what should be done and how it should be done. The function definition has the following

Syntax:

```
function functionName(arg1, arg2, ... , argN){  
  // Actual code for execution  
}
```

Function call: A function can be called from anywhere in the application. In both function calling and function definition, the parameter/argument must be the same. We must pass the same number of parameters that the function definition specifies. The function call has the following

Syntax:

```
functionName(arg1, arg2, ... , argM);
```

Types of Functions in TypeScript: There are two types of functions in TypeScript:

- Named Function
- Anonymous Function

1. Named function: A named function is defined as one that is declared and called by its given name. They may include parameters and have return types.

Syntax:

```
functionName( [args] ) { }
```

Example:

```
// Named Function Definition
function myFunction(x: number, y: number): number {
  return x + y;
}
// Function Call
myFunction(7, 5);
```

Output:

```
12
```

2. Anonymous Function: An anonymous function is a function without a name. At runtime, these kinds of functions are dynamically defined as an expression. We may save it in a variable and eliminate the requirement for function names. They accept inputs and return outputs in the same way as normal functions do. We may use the variable name to call it when we need it. The functions themselves are contained inside the variable.

Syntax:

```
let result = function( [args] ) { }
```

Example:

```
// Anonymous Function
let myFunction = function (a: number, b: number) : number {
  return a + b;
};
// Anonymous Function Call
console.log(myFunction(7, 5));
```

Output:

12

TypeScript has a specific syntax for typing function parameters and return values.

Return Type

The type of the value returned by the function can be explicitly defined.

```
// the `: number` here specifies that this function returns a number
function getTime(): number {
    return new Date().getTime();
}
```

**TypeScript OOPS**

TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class.

Creating classes

Use the class keyword to declare a class in TypeScript. The syntax for the same is given below –

Syntax:

```
class class_name {
    //class scope
}
```

The class keyword is followed by the class name. The rules for identifiers must be considered while naming a class.

A class definition can include the following –

- Fields – A field is any variable declared in a class. Fields represent data pertaining to objects
- Constructors – Responsible for allocating memory for the objects of the class

- Functions – Functions represent actions an object can take. They are also at times referred to as methods

These components put together are termed as the data members of the class.

Consider a class Person in typescript.

```
class Person {  
}
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10  
var Person = (function () {  
    function Person() {  
    }  
    return Person;  
})();
```

Example: Declaring a class

```
class Car {  
    //field  
    engine:string;  
  
    //constructor  
    constructor(engine:string) {  
        this.engine = engine  
    }  
  
    //function  
    disp():void {  
        console.log("Engine is : "+this.engine)  
    }  
}
```

The example declares a class Car. The class has a field named engine. The var keyword is not used while declaring a field. The example above declares a constructor for the class.

A constructor is a special function of the class that is responsible for initializing the variables of the class. TypeScript defines a constructor using the constructor keyword. A constructor is a function and hence can be parameterized.

The `this` keyword refers to the current instance of the class. Here, the parameter name and the name of the class's field are the same. Hence to avoid ambiguity, the class's field is prefixed with the `this` keyword.

`disp()` is a simple function definition. Note that the function keyword is not used here.

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var Car = (function () {
    //constructor
    function Car(engine) {
        this.engine = engine;
    }

    //function
    Car.prototype.disp = function () {
        console.log("Engine is : " + this.engine);
    };
    return Car;
})();
```

TypeScript Interface

An Interface is a structure which acts as a contract in our application. It defines the syntax for classes to follow, means a class which implements an interface is bound to implement all its members. We cannot instantiate the interface, but it can be referenced by the class object that implements it. The TypeScript compiler uses interface for type-checking (also known as "duck typing" or "structural subtyping") whether the object has a specific structure or not.

The interface contains only the declaration of the methods and fields, but not the implementation. We cannot use it to build anything. It is inherited by a class, and the class which implements interface defines all members of the interface.

When the Typescript compiler compiles it into JavaScript, then the interface will be disappeared from the JavaScript file. Thus, its purpose is to help in the development stage only.

Interface Declaration

We can declare an interface as below.

```
interface OS {  
    name: String;  
    language: String;  
}  
  
let OperatingSystem = (type: OS): void => {  
    console.log('The ' + type.name + ' has the fastest ' + type.language + ' performance of any Mac');  
};  
  
let ios = {name: 'M1 chip', language: 'single-core'}  
  
OperatingSystem(ios);
```

In the above example, we have created an interface OS with properties name and language of string type. Next, we have defined a function, with one argument, which is the type of interface OS.

Now, compile the TS file into the JS which looks like the below output.

```
let OperatingSystem = (type) => {  
    console.log('The ' + type.name + ' has the fastest ' + type.language + ' performance of any Mac');  
};  
  
let ios = {name: 'M1 chip', language: 'single-core'}  
  
OperatingSystem(ios);
```

Output:

```
PS C:\Users\tanaya.Deshpande\Desktop\Typescript> tsc firstprogram.ts  
PS C:\Users\tanaya.Deshpande\Desktop\Typescript> node firstprogram.js  
The M1 chip has the fastest single-core performance of any Mac
```

Use of Interface

We can use the interface for the following things:

- Validating the specific structure of properties
- Objects passed as parameters
- Objects returned from functions.

Interface Inheritance

We can inherit the interface from the other interfaces. In other words, Typescript allows an interface to be inherited from zero or more base types.

The base type can be a class or interface. We can use the "extends" keyword to implement inheritance among interfaces.

The following example helps us to understand the interface inheritance more clearly.

Syntax:

```
child_interface extends parent interface{  
}
```

Example:

```
interface Person {  
  name:string  
  age:number  
}  
interface Employee extends Person {  
  gender:string  
  empCode:number  
}  
let empObject = <Employee>{};  
empObject.name = "Aabha"  
empObject.age = 31  
empObject.gender = "Female"  
empObject.empCode = 18  
console.log("Name: "+empObject.name);  
console.log("Employee Code: "+empObject.empCode);
```

Output:

```
Name: Aabha  
Employee Code: 18
```

Let us take the above example, which helps us to understand the **multiple interface** inheritance.

Example:

```
interface Person {  
  name:string  
}  
  
interface PersonDetail {  
  age:number  
  gender:string  
}  
  
interface Employee extends Person, PersonDetail {  
  empCode:number  
}  
  
let empObject = <Employee>{};  
empObject.name = "Gautam"  
empObject.age = 35  
empObject.gender = "Male"  
empObject.empCode = 11  
console.log("Name: "+empObject.name);  
console.log("Employee Code: "+empObject.empCode);
```

Output:

```
Name: Gautam  
Employee Code: 11
```

Array Type Interface

We can also use interfaces to describe the array type. The following example helps us to understand the Array Type Interface.

Example:

```
// Array which return string
interface nameArray {
  [index:number]:string
}

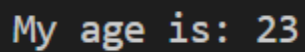
// use of the interface
let myNames: nameArray;
myNames = ['Smriti', 'Jhulan', 'Deepti'];

// Array which return number
interface ageArray {
  [index:number]:number
}

var myAges: ageArray;
myAges =[21, 23, 25];
console.log("My age is: " +myAges[1]);
```

In the above example, we have declared nameArray that returns string and ageArray that returns number. The type of index in the array is always number so that we can retrieve array elements with the use of its index position in the array.

Output:

My age is: 23

Interface in a class

TypeScript also allows us to use the interface in a class. A class implements the interface by using the implements keyword. We can understand it with the below example.

Example:

```
// defining interface for class
interface Person {
  firstName: string;
  lastName: string;
  age: number;
  FullName();
  GetAge();
}

// implementing the interface
class Employee implements Person {
  firstName: string;
  lastName: string;
  age: number;
  FullName() {
    return this.firstName + ' ' + this.lastName;
  }
  GetAge() {
    return this.age;
  }
  constructor(firstN: string, lastN: string, getAge: number) {
    this.firstName = firstN;
    this.lastName = lastN;
    this.age = getAge;
  }
}

// using the class that implements interface
let myEmployee = new Employee('Nishant', 'kapoor', 30);
let fullName = myEmployee.FullName();
let Age = myEmployee.GetAge();
console.log("Name of Person: " + fullName + '\nAge: ' + Age);
```

In the above example, we have declared Person interface with firstName, lastName as property and FullName and GetAge as method/function. The Employee class implements this interface by using the implements keyword. After implementing an interface, we must declare the properties and methods in the class. If we do not implement those properties and methods, it throws a compile-time error. We have also declared a constructor in the class. So when we instantiate the class, we need to pass the necessary parameters otherwise it throws an error at compile time.

Output:

```
Name of Person: Nishant kapoor  
Age: 30
```