NODE.JS

Server Side Javascript

Node.js – an intro

- In 2009 Ryan Dahl created Node.js or Node, a framework primarily used to create highly scalable servers for web applications.
  - Node.js is an open source, cross-platform runtime environment for server-side JavaScript.
  - Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute code.
  - It is written in C++ and JavaScript.
  - You write Node.js code in JavaScript, and then V8 compiles it into machine code to be executed.
  - You can write most—or maybe even all—of your server-side code in Node.js, including the webserver and the server-side scripts and any supporting web application functionality.

Node.js – an intro

- It’s a highly scalable system that uses asynchronous, non-blocking I/O model (input/output), rather than threads or separate processes
- It is not a framework like jQuery nor a programming language like C# or JAVA; Its primarily a Javascript engine
- It's a new kind of web server that has a lot in common with other popular web servers, like Microsoft’s Internet Information Services (IIS) or Apache
- With Node.js, you can build many kinds of networked applications.
  - For instance, you can use it to build a web application service, an HTTP proxy, a DNS server, an SMTP server, an IRC server, and basically any kind of process that is network intensive.

Traditional Programming Limitations

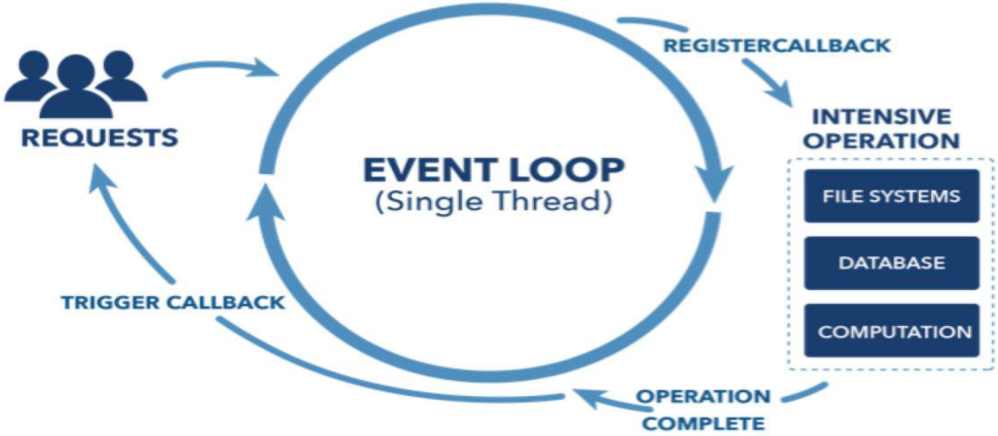
- In traditional programming I/O is performed in the same way as it does local function calls. i.e. Processing cannot continue until the operation is completed.
    - When the operation like executing a query against database is being executed, the whole process/thread idles, waiting for the response. This is termed as “Blocking”
- Event-driven programming or Asynchronous programming is a programming style where the flow of execution is determined by events.
- Events are handled by event handlers or event callbacks
    - An event callback is a function that is invoked when something significant happens like when the user clicks on a button or when the result of a database query is available.

```
result = query('SELECT * FROM posts WHERE id = 1');
do_something_with(result);
```

```
query_finished = function(result) {
    do_something_with(result);
}
query('SELECT * FROM posts WHERE id = 1', query_finished);
```

Event loop

- An event loop is a construct that mainly performs two functions in a continuous loop — **event detection and event handler triggering**.
  - In any run of the loop, it has to detect which events just happened.
  - Then, when an event happens, the event loop must determine the event callback and invoke it.
- This event loop is just one thread running inside one process, which means that, when an event happens, the event handler can run without interruption. This means the following:
  - There is at most one event handler running at any given time.
  - Any event handler will run to completion without being interrupted.



Why Node.js

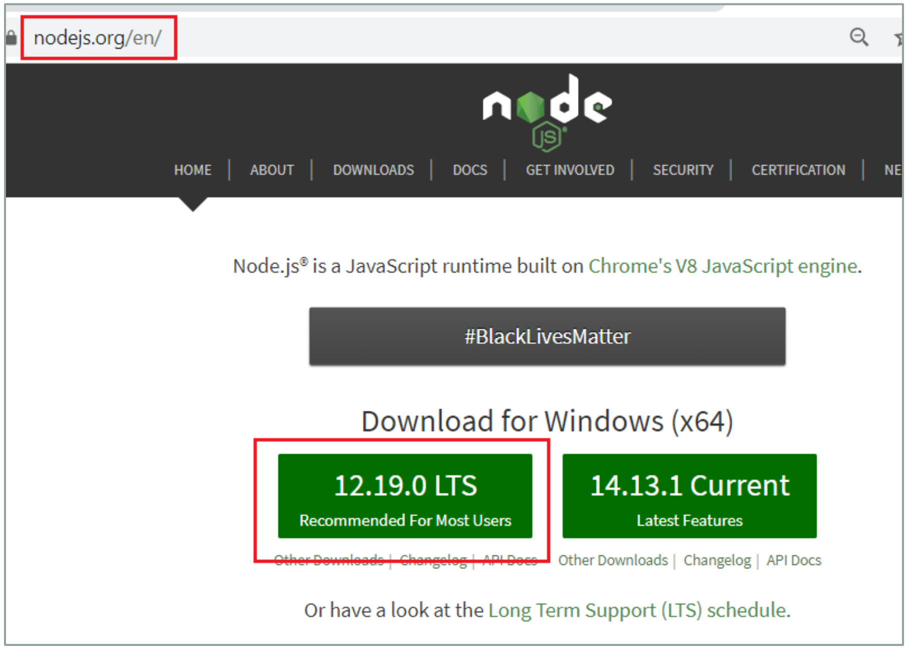
- Node allows developers to write server side code using javascript
  - Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.
- Perfect for data-intensive real-time applications that run across distributed devices

- **Node.js is really two things: a runtime environment and a library**

- **What Node is NOT!**
  - Node is **not** a webserver. By itself it doesn't do anything. Node.js is just another way to execute code on your computer. It is simply a JavaScript runtime.

Setting up Node

- To install and setup an environment for Node.js:
- Download the latest version of Node.js installable archive file from https://nodejs.org/en/

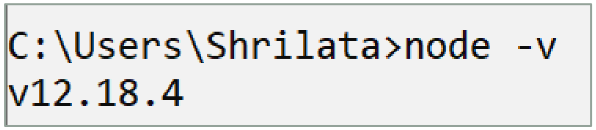

nodejs.org/en/
node js
HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | CERTIFICATION | NE
#BlackLivesMatter
12.19.0 LTS
Recommended For Most Users
14.13.1 Current
Latest Features
Other Downloads | Changelog | API Docs
Other Downloads | Changelog | API Docs

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

Download for Windows (x64)

Or have a look at the Long Term Support (LTS) schedule.

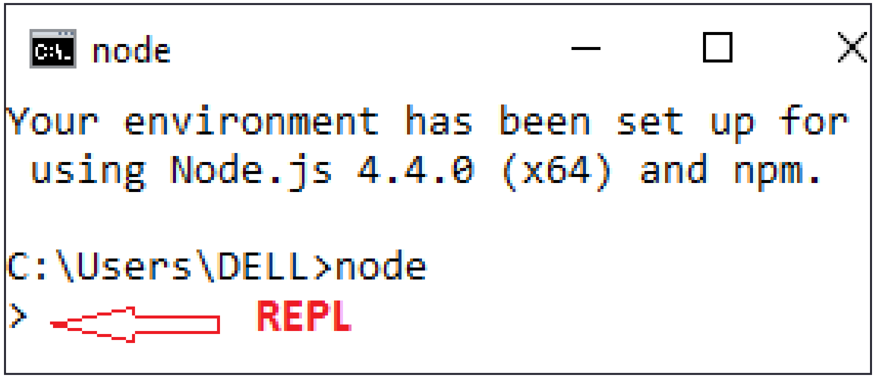
- Double click to run the msi file
- To verify if the installation was successful, enter the command node –v in the terminal window.



```
C:\Users\Shrilata>node -v
v12.18.4
```

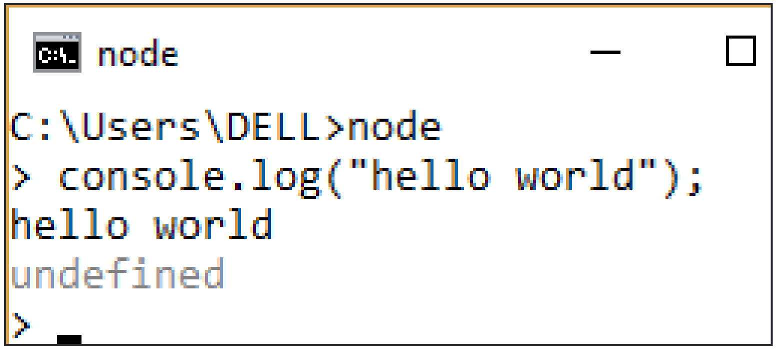
Using the Node CLI : REPL (Read-Eval-Print-Loop)

- There are two primary ways to use Node.js on your machines: by using the Node Shell or by saving JavaScript to files and running those.
  - Node shell is also called the Node REPL; a great way to quickly test things in Node.
  - When you run “node” without any command line arguments, it puts you in REPL



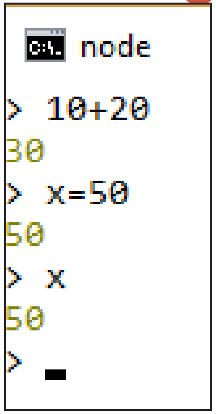
```
Your environment has been set up for
 using Node.js 4.4.0 (x64) and npm.

C:\Users\DELL>node
>
```

REPL

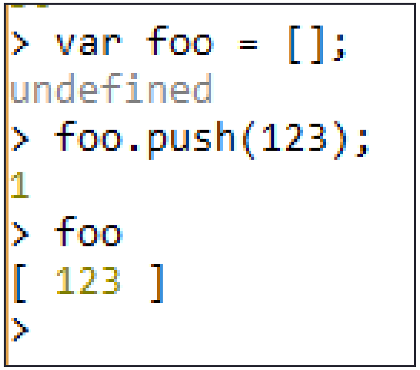


```
C:\Users\DELL>node
> console.log("hello world");
hello world
undefined
> _
```

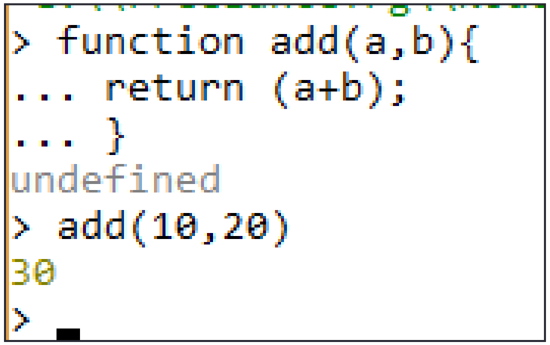
Using the REPL



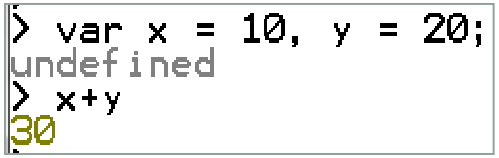
```
node
> 10+20
30
> x=50
50
> x
50
> _
```



```
> var foo = [];
undefined
> foo.push(123);
1
> foo
[ 123 ]
>
```

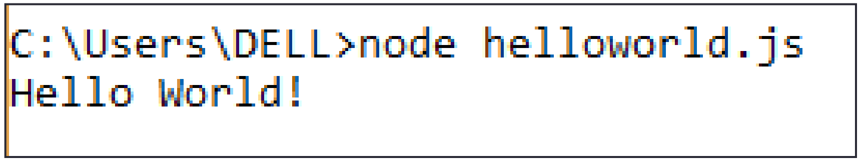


```
> function add(a,b){
... return (a+b);
... }
undefined
> add(10,20)
30
> _
```



```
> var x = 10, y = 20;
undefined
> x+y
30
```

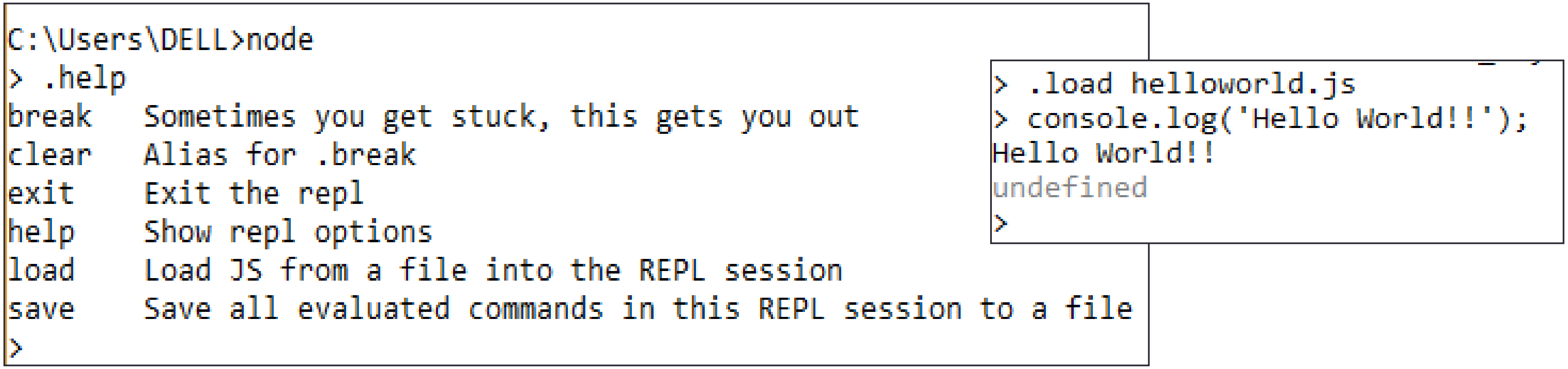
- You can also create a js file and type in some javascript.



```
C:\Users\DELL>node helloworld.js
Hello World!
```

```
//helloworld.js
console.log("Hello  World!");
```

- To view the options available to you in REPL type .help and press Enter.

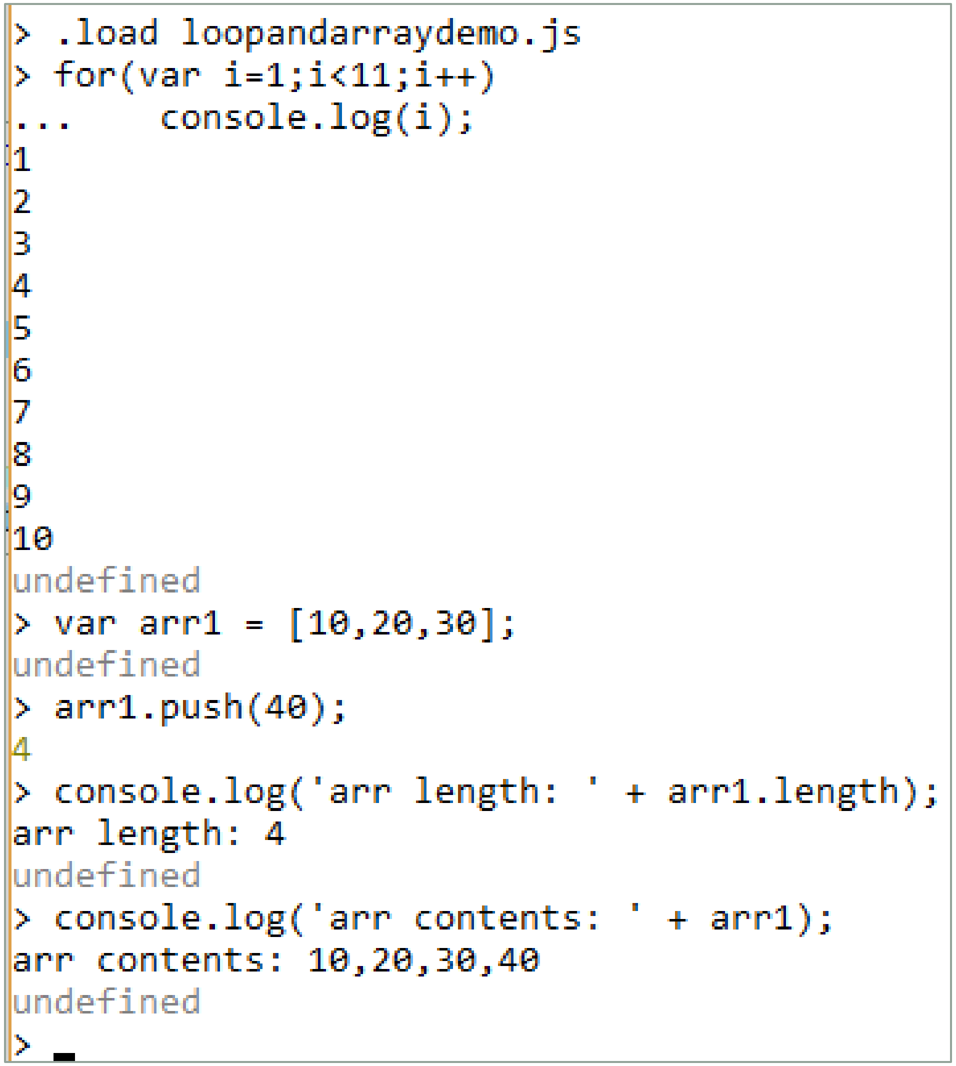


```
C:\Users\DELL>node
> .help
break    Sometimes you get stuck, this gets you out
clear    Alias for .break
exit     Exit the repl
help     Show repl options
load     Load JS from a file into the REPL session
save     Save all evaluated commands in this REPL session to a file
>
```

```
> .load helloworld.js
> console.log('Hello World!!');
Hello World!!
undefined
>
```

Demo

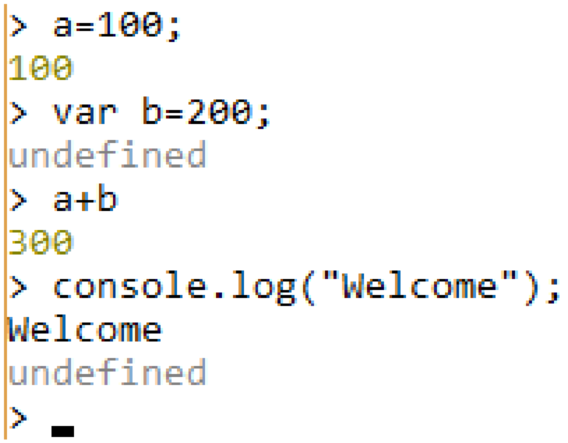
```
//loopAndArrayDemo.js
for(var i=1;i<11;i++)
   console.log(i );

var arr1 = [10,20,30];
arr1.push(40);

console.log('arr length: ' + arr1.length);
console.log('arr contents: ' + arr1);
```



```
> .load loopandarraydemo.js
> for(var i=1;i<11;i++)
...     console.log(i);
1
2
3
4
5
6
7
8
9
10
undefined
> var arr1 = [10,20,30];
undefined
> arr1.push(40);
4
> console.log('arr length: ' + arr1.length);
arr length: 4
undefined
> console.log('arr contents: ' + arr1);
arr contents: 10,20,30,40
undefined
> _
```

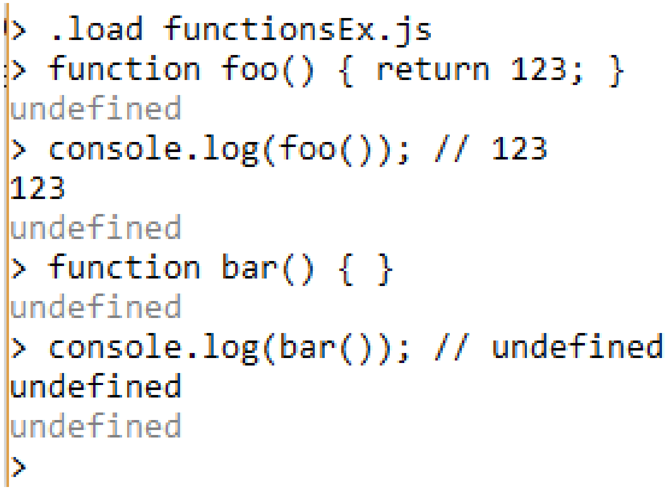
Variables and functions (Recap)

- Variables can be declared as usual.
  - But, if **var** keyword is not used, then the value is stored in the variable and printed.
  - Whereas if **var** keyword is used, then the value is stored but not printed.
  - You can print variables using **console.log()**.



```
> a=100;
100
> var b=200;
undefined
> a+b
300
> console.log("Welcome");
Welcome
undefined
> ▬
```

- **Functions:**
  - All functions return a value in JavaScript. If no explicit return statement, func returns undefined.



```
> .load functionsEx.js
> function foo() { return 123; }
undefined
> console.log(foo()); // 123
123
undefined
> function bar() { }
undefined
> console.log(bar()); // undefined
undefined
undefined
>
```

- **Anonymous Function**
  - A function without a name

```
var foo2 = function () {
          console.log('foo2');
     }
foo2();    // foo2
```

Higher-Order Functions (Recap)

- Since JavaScript allows us to assign functions to variables, we can pass functions to other functions.
  - Functions that take functions as arguments are called *higher-order functions*
  - *Eg, geolocation.getCurrentPosition(func1, func2)*
  - *Eg $(document).ready(function(){});*
  - *Eg*

```
setTimeout(function () {
          console.log('2 secs have passed since demo started');
   }, 2000);
```

```
setTimeout(f1, 2000);
---------------------------------
function f1 () {
          console.log('2 secs have passed since demo started’);
}
```

Node js Modules

- A module in Node.js is a logical encapsulation of code in a single unit.
  - Since each module is an independent entity with its own encapsulated functionality, it can be managed as a separate unit of work.

- Consider modules to be the same as JavaScript libraries.
  - A set of functions you want to include in your application.
  - Module in Node.js is a simple or complex functionality organized in JavaScript files which can be reused throughout a Node.js application.

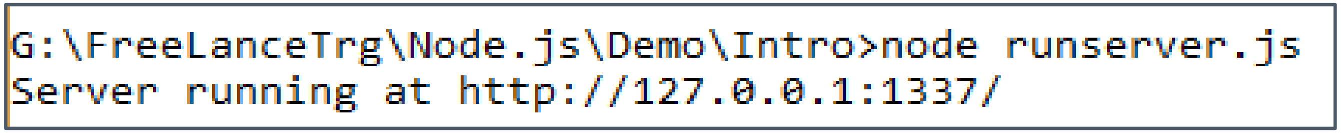
- Node.js uses a module architecture to simplify creation of complex apps
  - For ex, the http module contains functions specific to HTTP. Eg http.createServer()

Node js Modules

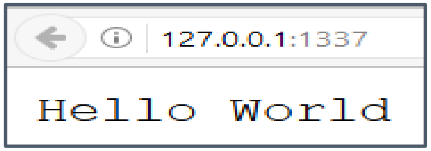
- Node.js has a set of built-in modules which you can use without any further installation.
- Built-in modules provide a core set of features we can build upon.
    - To include a module, use the require() function with the name of the module.

```
//RunServer.js
var http = require("http");
function process_request(req, res) {
   var body = ‘Hello World\n';
   var content_length = body.length ;
   res.writeHead(200, {
              'Content-Length': content_length,
              'Content-Type': 'text/plain‘   });
    res.end(body);
}
var srv = http.createServer(process_request);
srv.listen(1337, '127.0.0.1’);
console.log('Server running at http://127.0.0.1:1337/');
```

```
//alternatively
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
  })
  .listen(1337, '127.0.0.1');
```



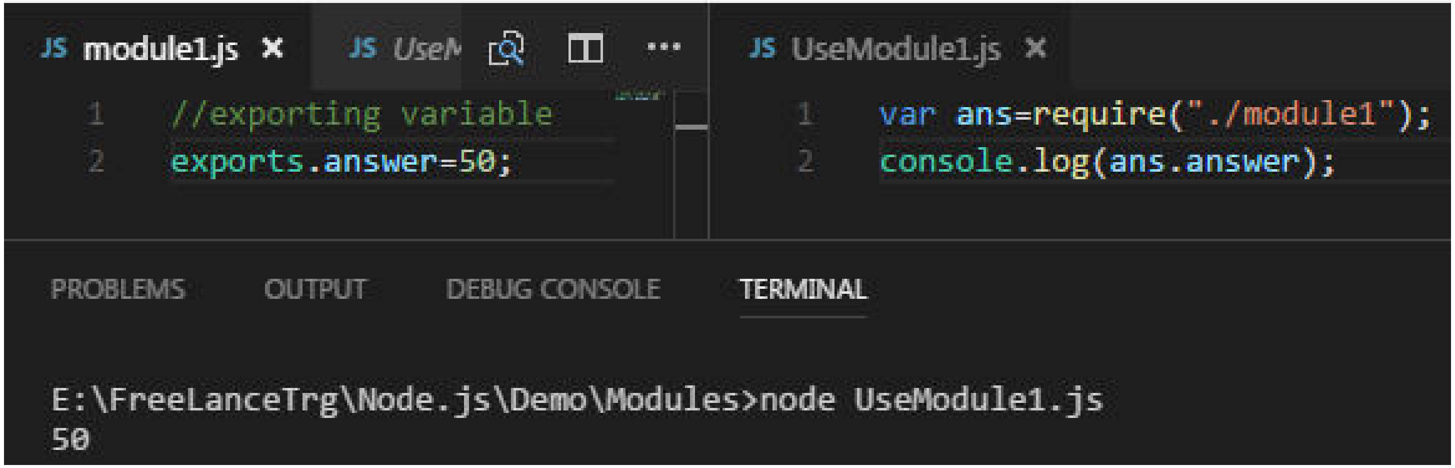
```
G:\FreeLanceTrg\Node.js\Demo\Intro>node runserver.js
Server running at http://127.0.0.1:1337/
```



```
Hello World
```

Create Your Own Modules

- You can create your own modules, and easily include them in your applications.



```
//exporting variable
exports.answer=50;
```

```
var ans=require("./module1");
console.log(ans.answer);
```

```
E:\FreeLanceTrg\Node.js\Demo\Modules>node UseModule1.js
50
```

  - exports object is a special object created by the Node module system which is returned as the value of the require function when you include that module.
  - module is just a plain JavaScript object with an exports property.

Create Your Own Modules

- You can create your own modules, and easily include them in your applications.

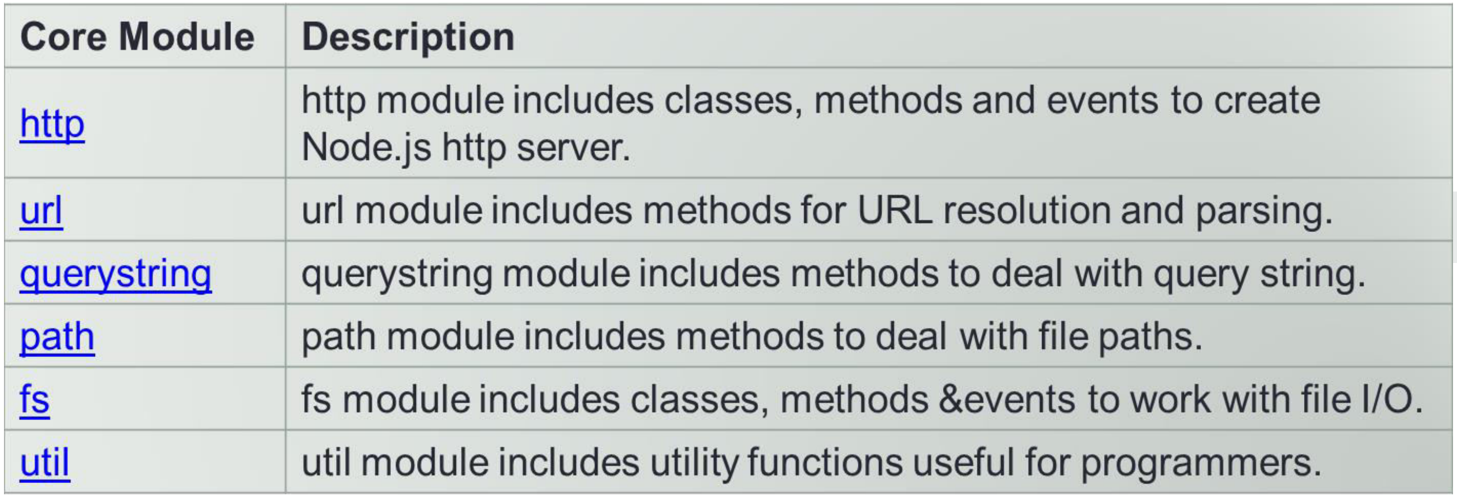
```
//module2.js
exports.sayHelloInEnglish  = function(){
          return "Hello";
     };
exports.sayHelloInSpanish  = function(){
          return "Hola";
      };
```

```
//UseModule2.js
var greet=require("./module2");
console.log(greet.sayHelloInSpanish());     //Hola
```

Node.js Module

- Node.js includes three types of modules:
  - Core Modules
  - Local Modules
  - Third Party Modules

- Core Modules
  - Unlike other programming technologies, Node.js doesn't come with a heavy standard library. The core modules of node.js are a bare minimum, and the rest can be cherry-picked via the NPM registry.
  - In order to use Node.js core or NPM modules, you first need to import it using require() function: var module = require('module_name');



Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods &events to work with file I/O.
util	util module includes utility functions useful for programmers.

Third Party Modules

- Node.js also has the ability to embedded external functionality or extended functionality by making use of custom modules.
- These modules have to be installed separately (using NPM)
  - Apart from writing our own modules and core modules, we will frequently use the modules written by other people in the Node community and published on the Internet (**npmjs.com**).
- Summarizing:
  - NPM is a command line tool that installs, updates or uninstalls Node.js packages in your application.
  - It is also an online repository for open-source Node.js packages.
- NPM is a command line tool that installs, updates or uninstalls Node.js packages in your application.
  - NPM is included with Node.js installation.
  - After you install Node.js, verify NPM installation : **npm -v**
- npm manages Node modules and their dependencies

NPM (Node Package Manager)

- Installing Packages
  - In order to use a module, you must install it on your machine.
  - To install a package, type npm install, followed by the package name
- There are two ways to install a package using npm: globally and locally.
  - **Globally** − This method is generally used to install development tools and CLI based packages. To install a package globally, use the following code.
  - npm install -g <package-name>
  - Eg to install Typescript : npm install -g typescript
  - **Locally** − This method is generally used to install frameworks and libraries. A locally installed package can be used only within the directory it is installed. To install a package locally, use the same command as above without the -g flag.
  - npm install <package-name>
  - Eg : To install cookie parser in Express : npm install --save cookie-parser

NPM

- **Installing a package using NPM**
- *$ npm install [-g] <Package Unique Name>*
- **To remove an installed package**
- *npm uninstall [-g] < Package Unique Name>*
- **To update a package to its latest version**
- *npm update [-g] < Package Unique Name>*

- **package.json**
  - The package.json file in Node.js is the heart of the entire application.
  - It is basically the manifest file that contains the metadata of the project.
  - package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.
    - It must be located in project's root directory.
  - To create package.json use npm init

TYPESCRIPT

Pre-reqs:

- HTML
- CSS
- Basic JavaScript
- OOP concepts

TypeScript

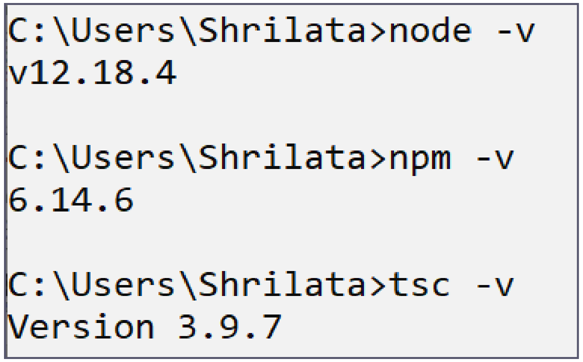
- Whats wrong with Javascript?
  - Not suitable for large applications
  - Lacks strong typing : means some errors might pop up only at run time

- Typescript : is a free and open-source programming language developed by Microsoft
  - Is a **typed** superset of JavaScript
  - Transpilation compiles TypeScript to JavaScript
  - Is object oriented with classes, interfaces and statically typed
  - “TypeScript is JavaScript for application-scale development.”
  - Provides data types and strong typing
  - It is portable as it runs on any browser, any host and device

- Components of TypeScript
  - **Language :** comprises of the syntax, keywords and type annotations
  - **The TypeScript Compiler : (tsc)** converts the instructions written in TypeScript to its JavaScript equivalent.

Getting Started

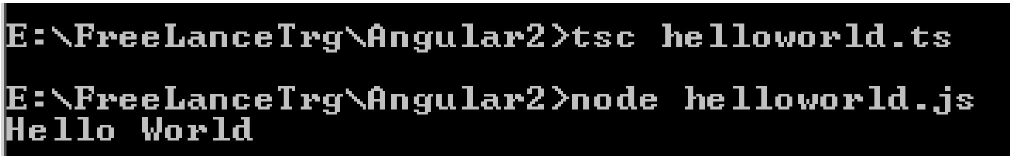
- To write, compile and run typescript code.
  - Install NodeJS, followed by TypeScript into local systems
  - Install TypeScript as follows:
  - npm install –g typescript (-g installs typescript so that it is accessible globally across all applications on this comp)
  - Node version 4.6.x or greater, npm 3.x.x or greater
  - To check version of node and npm installed : node –v and npm –v and tsc –v



```
C:\Users\Shrilata>node -v
v12.18.4

C:\Users\Shrilata>npm -v
6.14.6

C:\Users\Shrilata>tsc -v
Version 3.9.7
```

  - Create a .ts file for first TS application:
  - Compile: tsc helloworld.ts
  - Run : node helloworld.js

```
//helloworld.ts
var message:string = "Hello World”
console.log(message)
```



```
E:\FreeLanceTrg\Angular2>tsc helloworld.ts

E:\FreeLanceTrg\Angular2>node helloworld.js
Hello World
```

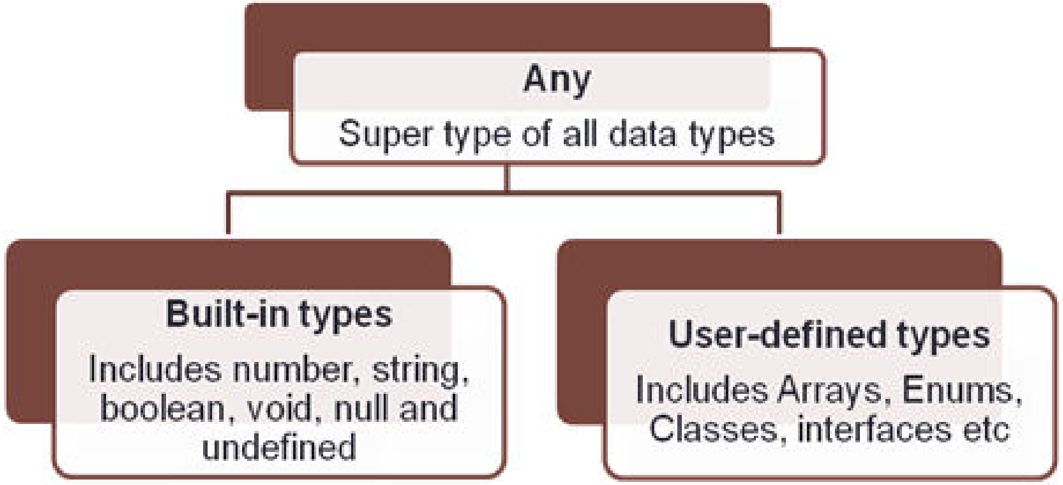
TypeScript Basic Syntax

- Identifiers are names given to elements in a program like variables, functions etc. The rules for identifiers are:
  - Can include both, characters and digits. However, cannot begin with a digit.
  - Cannot include special symbols except for underscore (_) or a dollar sign ($).
  - Cannot be keywords.
  - Must be unique and cannot contain spaces
  - Are case-sensitive.
  - **Valid :** firstName, first_name, num1, $result
  - **Invalid** : var, first name, first-name , 1number
- Semicolons are optional:
- Comments
  - **Single-line comments (//)** − Any text between a // and the end of a line
  - **Multi-line comments (/* */)** − These comments may span multiple lines.

```
//this is single line comment
/* This is a
   Multi-line comment  */
```

TypeScript Types

- Any type : is the super type of all types in TypeScript; denotes a dynamic type.
  - Using the any type is equivalent to opting out of type checking for a variable
- **Built-in types**



Any

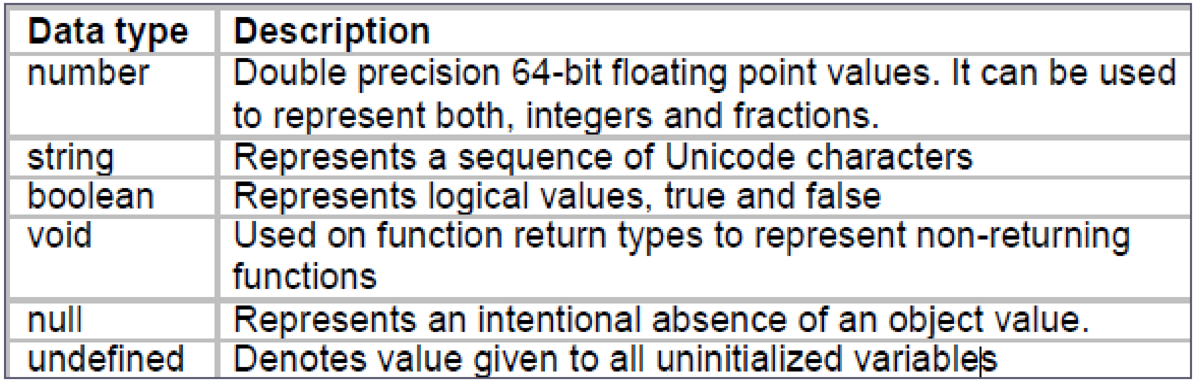
Super type of all data types

Built-in types

Includes number, string, boolean, void, null and undefined

User-defined types

Includes Arrays, Enums, Classes, interfaces etc



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

TypeScript Types

- Array: Egs
  - var jobs: Array<string> = ['IBM', 'Microsoft', 'Google'];
  - var jobs: string[] = ['Apple', 'Dell', 'HP'];
  - We specify the type of the items in the array with either the Array<type> or type[ ] notations

- Enums : They work by naming numeric values.
  - Eg : fixed list of roles a person may have could be written as:
  - enum Role {Employee, Manager, Admin};
  - var role: Role = Role.Employee;

- Any : is the default type if we omit typing for a given variable.
  - Having a variable of type any allows it to receive any kind of value

    var something: any = 'as string';

    something = 1;

    something = [1, 2, 3];

Variable Declaration in TypeScript

- Declare a variable by using the **var** keyword:
- var identifier : [type-annotation] = value ;

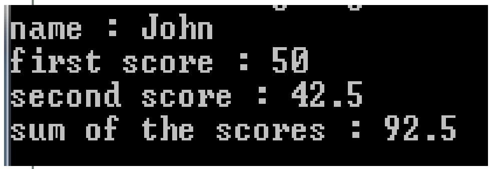
- When you declare a variable, you have four options −

1. Declare its type and value in one statement.
   var name:string = ”mary”
2. Declare its type but no value. In this case, the variable will be set to *undefined*.
   var name:string;
3. Declare its value but no type.
   var name = ”mary” //The type is inferred from the value. Here, type string
4. Declare neither value not type. In this case, the data type of the variable will be **any** and will be initialized to *undefined*.
   var name;

Examples

- TypeScript will try to infer as much of the type information as it can in order to give you type safety with minimal cost of productivity during code development.
- Eg : var foo = 123;
- foo = '456'; // Error: cannot assign `string` to `number`

```
var pname:string = "John";
var score1:number = 50;
var score2:number = 42.50
var sum = score1 + score2
console.log("name : "+ pname)
console.log("first score: "+ score1)
console.log("second score: "+ score2)
console.log("sum of the scores: "+ sum)
```



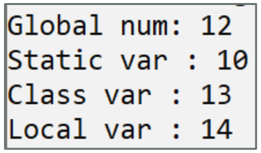
```
name : John
first score : 50
second score : 42.5
sum of the scores : 92.5
```

```
var any1; // any value. same as not having a static type
var num1: number; // number type
num1 = 1;              // set after the fact.
var num2: number = 2;          // initialized and typed
var num3 = 3;          //typed as a number via type inference
var str1 = num1 + 'some string';
console.log(typeof(str1))          //string
```

Variable Scope

- Variables can be of the following scopes −
  - Global Scope − Global variables are declared outside the programming constructs; can be accessed from anywhere within your code.
  - Class Scope − also called fields; are declared within the class but outside the methods; 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 are declared within the constructs like methods, loops etc; accessible only within the construct where they are declared.

```
var global_num = 12          //global variable

class Numbers {
   num_val = 13;             //class variable
   static sval = 10;         //static field
   storeNum():void {
        var local_num = 14;    //local variable
        console.log("Local var : " + local_num);
   }
}

console.log("Global num: "+global_num)
console.log(“Static var : “ + Numbers.sval)   //static variable
var obj = new Numbers();
console.log("Global num: "+obj.num_val)
obj.storeNum();
```



```
Global num: 12
Static var : 10
Class var : 13
Local var : 14
```

If you try accessing the local variable outside the method, it results in a compilation error.

TypeScript Operators

- Arithmetic operators (+,-,*,/,%)
- Assignment operators (=,+=, -=, /=, *=, %=)
- Comparison operators (==, !=, < <=, > >=)
- Boolean operators (&&, ||, !)
- Bitwise operators (&, |, !, ^, >>, >>>)
- String operators (=, +, +=)
- Ternary/conditional operator (Test ? expr1 : expr2).
  - Eg var result = num > 0 ?"positive":"negative"
- Type Operator (typeof)

```
var num = 12
console.log(typeof num);    //output: number
```

- Instanceof : used to test if an object is of a specified type or not (more later)

Language constructs

```
if(boolean_expression) {
   // statements
}

if(boolean_expression) {
   // statements
} else {
   // statements
}
```

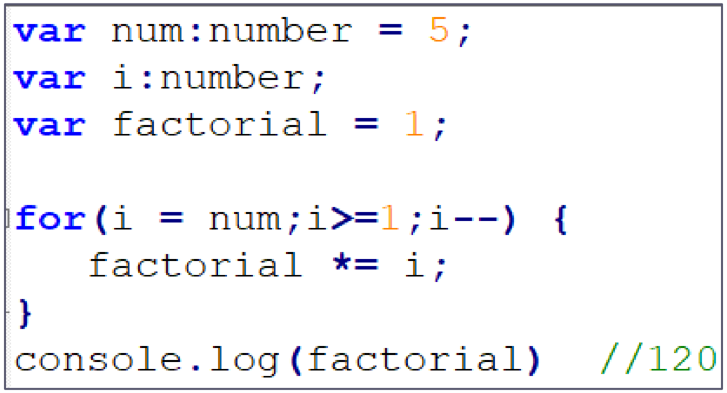
```
//example
var num:number  = 12;
if (num  % 2==0) {
   console.log("Even");
} else {
   console.log("Odd");
}
```

```
switch(variable_expression) {
   case constant_expr1: {
      //statements;   break;
   }
   case constant_expr2: {
      //statements;   break;
   }
   default: {
      //statements;   break;
   }
}
```

```
//example
var grade:string = "A";
switch(grade) {
   case "A": {
      console.log("Excellent");
      break;
   }
   case "B": {
      console.log("Good");
      break;
   }
default: {
      console.log("Invalid  ");
      break;
   } }
```

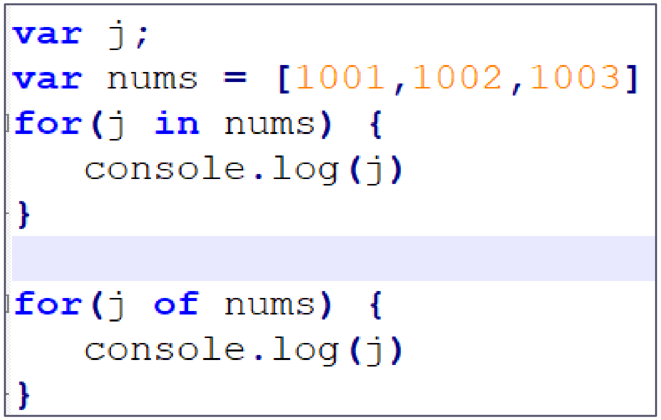
Language constructs

```
for (initialvalue; condition; step) {
   //statements
}
```

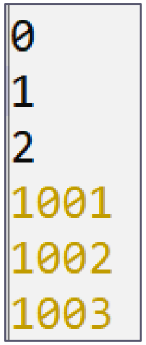


```
var num:number = 5;
var i:number;
var factorial = 1;

for(i = num;i>=1;i--) {
   factorial *= i;
}
console.log(factorial)  //120
```

```
// for...in loop
for (var val in array/tuple/collection) {
   //statements
}
```



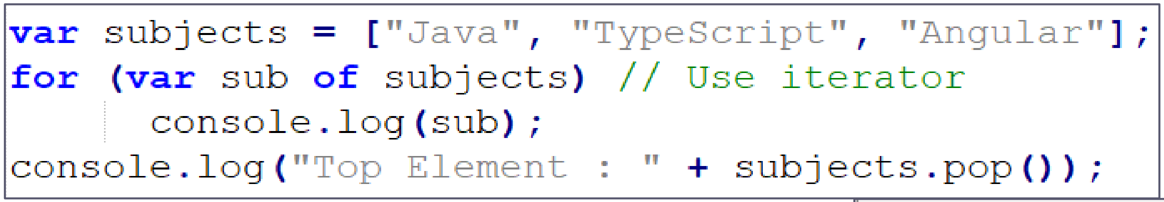
```
var j;
var nums = [1001,1002,1003]
for(j in nums) {
   console.log(j)
}

for(j of nums) {
   console.log(j)
}
```



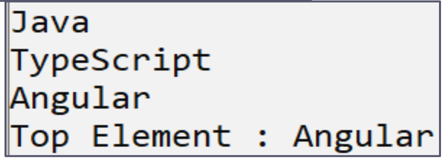
```
0
1
2
1001
1002
1003
```

```
while(condition) {
   // statements
}
```

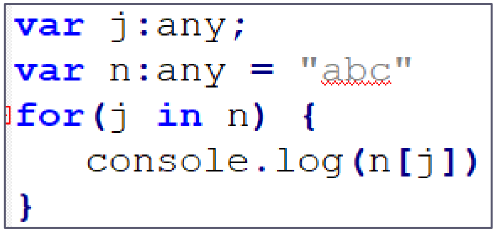
```
do {
   //statements
} while(condition)
```



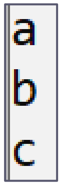
```
var subjects = ["Java", "TypeScript", "Angular"];
for (var sub of subjects) // Use iterator
        console.log(sub);
console.log("Top Element : " + subjects.pop());
```



```
Java
TypeScript
Angular
Top Element : Angular
```



```
var j:any;
var n:any = "abc"
for(j in n) {
   console.log(n[j])
}
```



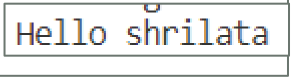
```
a
b
c
```

TypeScript Functions

```
function fname()[:return_type] {
      //statements
      [return value;]
}
```

```
function fname( param1 [:datatype],
                param2 [:datatype]) {
}
```

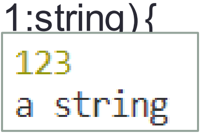
```
function greetText(name: string): string {
    return "Hello " + name;
}
console.log("Shrilata");
```



```
Hello shrilata
```

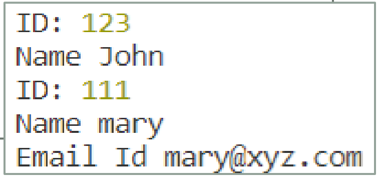
```
function greet():string {  //function returns a string
   return "Hello World"
}

function caller() {   //func with no args no return
     var msg = greet()    //function greet() invoked
     console.log(msg)
}
caller()    //invoke function
```

```
//parameterized functions
function test(n1:number, s1:string){
   console.log(n1)
   console.log(s1)
}

test(123,"a string")
```



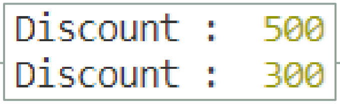
```
123
a string
```

```
//optional parameters
function disp(id:number,   name:string,   email?:string) {
   console.log("ID:", id);
   console.log("Name",name);

   if(email!=undefined)
   console.log("Email Id",email);
}
disp(123,"John");
disp(111,"mary","mary@xyz.com");
```



```
ID: 123
Name John
ID: 111
Name mary
Email Id mary@xyz.com
```

```
//Default Parameters
 function calc(price:number, rate:number = 0.50) {
   var discount = price * rate;
   console.log("Discount : ",discount);
}
calc(1000)
calc(1000,0.30)
```



```
Discount :  500
Discount :  300
```

Rest parameters

- Rest parameters (...argumentName for the last argument) allow you to quickly accept multiple arguments in function and get them as an array.

```
function iTakeItAll(first, second, ...allOthers) {
    console.log(allOthers);
}
iTakeItAll('foo', 'bar'); // []
iTakeItAll('foo', 'bar', 'bas', 'qux'); // ['bas','qux']
```

```
function addNumbers(...nums:number[]) {
   var i;
   var sum:number = 0;

   for(i = 0;i<nums.length;i++) {
        sum = sum + nums[i];
   }
   console.log("sum of the numbers",sum)
}
addNumbers(1,2,3)
addNumbers(10,10,10,10,10)
```

Javascript Recap : Anonymous Function

- Functions that are not bound to an identifier (function name) are called as **anonymous functions**.
  - These functions are dynamically declared at runtime.
  - Can accept inputs and return outputs, just as standard functions do.
  - Variables can be assigned an anonymous function. Such an expression is called a function expression.

```
var res = function( [arguments] ) { ... }
```

```
var msg = function() {
   return "hello world";
}
console.log(msg())
```

```
 var area = function (radius:number) {
       return Math.PI * radius * radius;
   };
 console.log(area(5));         // 78.5
```

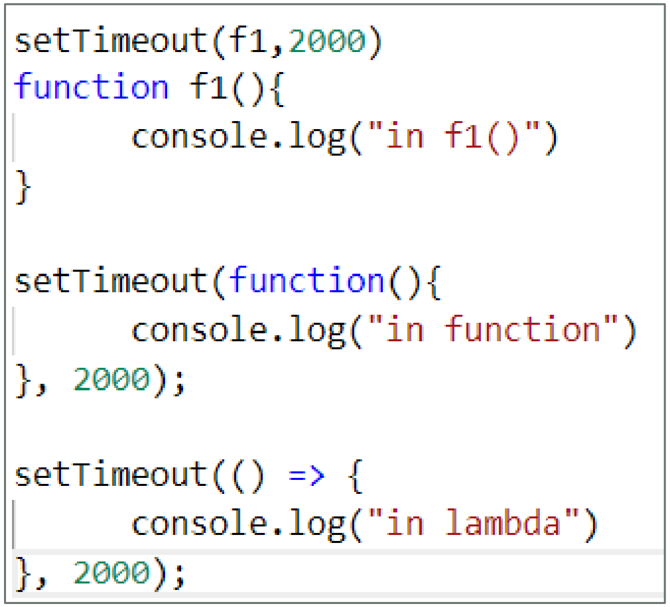
Arrow (Lambda)Functions

- Fat arrow (=>) is also called a lambda function.
  - Lambda refers to anonymous functions
  - There are 3 parts to a Lambda function : Parameters (optional), fat arrow notation/lambda notation (=>) , Statements (represent the function logic)
  - Syntax:

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: (param1, param2, …, paramN) => { return expression; }

// Parentheses are optional when there's only one parameter name:
(singleParam) => { statements }
singleParam => { statements }

// A function with no parameters should be written with a pair of parentheses.
() => { statements }

// Parenthesize the body of function to return an object literal expression:
params => ({foo: bar})  //notice the parenthesis around the JSON object

// Rest parameters are supported
(param1, param2, ...rest) => { statements }
```

Arrow (Lambda)Functions

- Fat arrow => functions are a shorthand notation for writing functions.
  - In ES5, whenever we want to use a function as an argument we have to use the function keyword along with {} braces like so:



```
setTimeout(f1,2000)
function f1(){
      console.log("in f1()")
}

setTimeout(function(){
      console.log("in function")
}, 2000);

setTimeout(() => {
      console.log("in lambda")
}, 2000);
```

```
// ES5-like example
var data = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach(function(line) { console.log(line); });
```

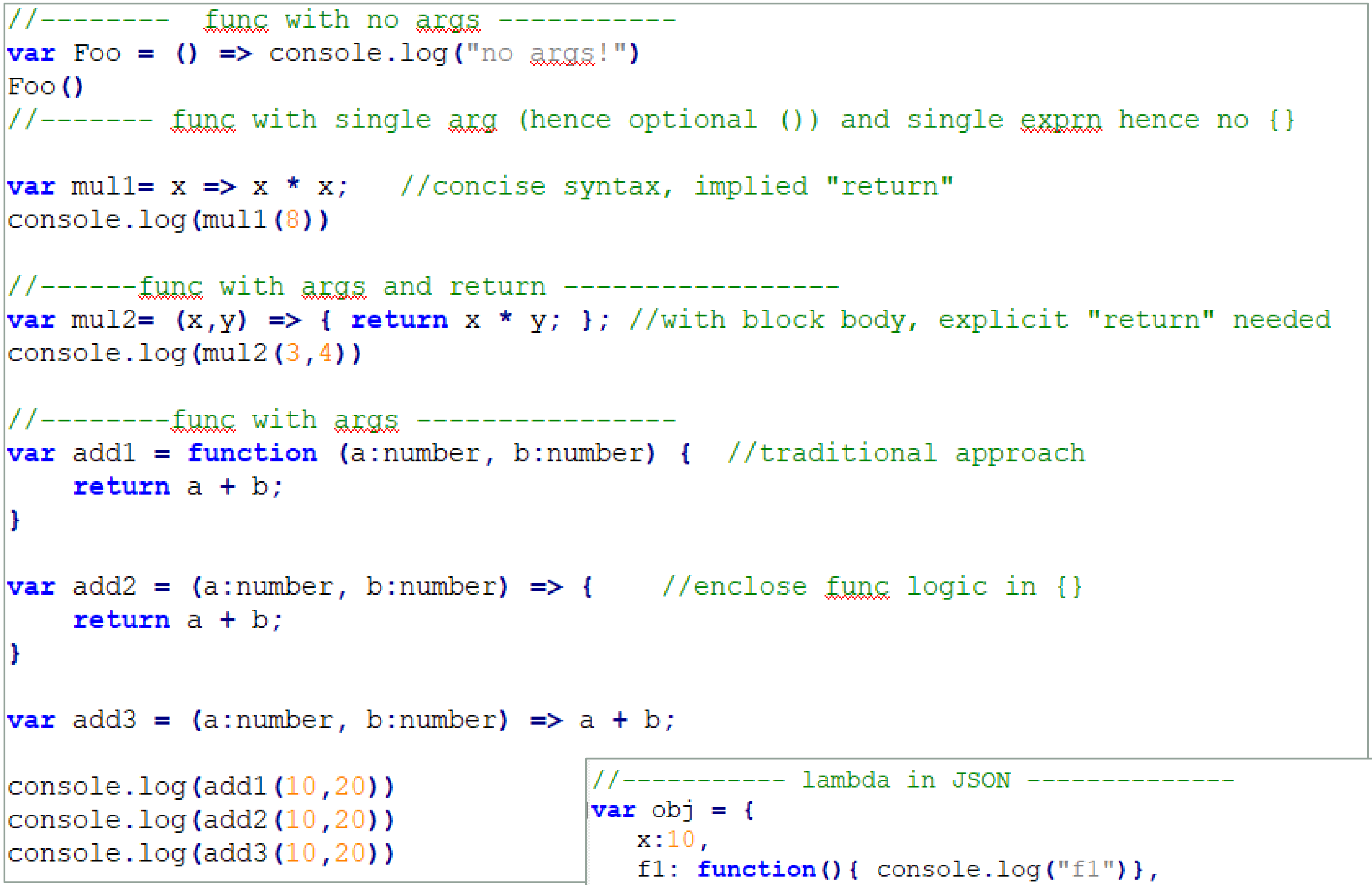
```
// Typescript example
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach( (line) => console.log(line) );
```

Normal function	Lambda function
var foo = function (x) { return 10 + x; };	var foo = (x:number)=>10 + x

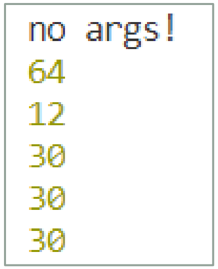
var foo = function (x) {
x = 10 + x; console.log(x);
};
foo(100);
	var foo = (x:number)=> {
	x = 10 + x ; console.log(x)
	}
	foo(100)

var display = function (x) {
console.log("The function got " + x);
};
display(12);
	var display = x=> {
	console.log("The function got "+x)
	}
	display(12)

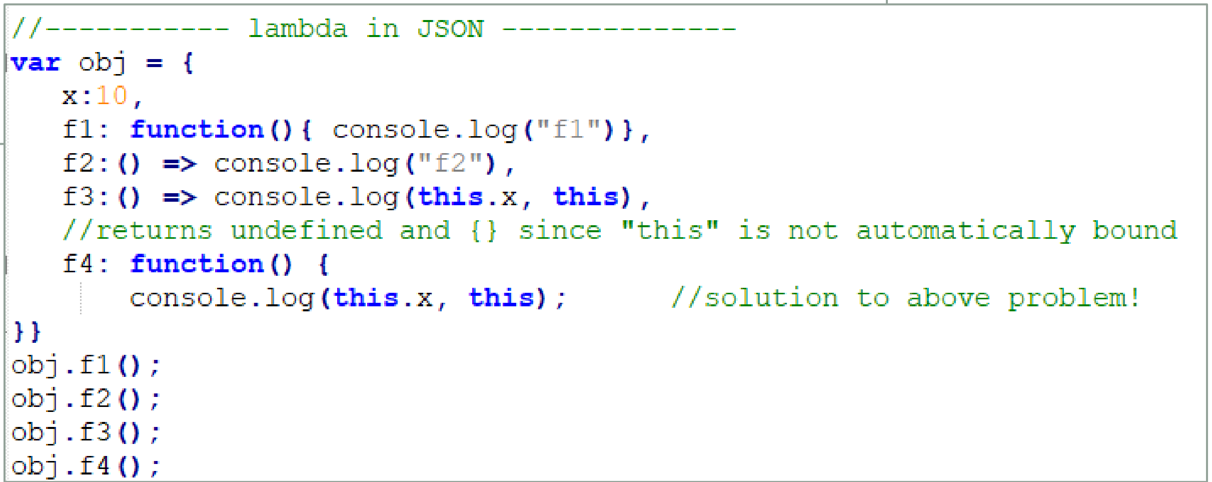
var disp = function () {
console.log("Function invoked");
};
disp();
	var disp =()=> {
	console.log("Function invoked");
	}
	disp();



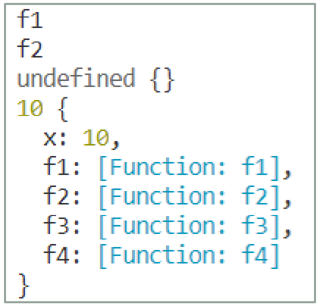
```
//--------  func with no args -----------
var Foo = () => console.log("no args!")
Foo()
//------- func with single arg (hence optional ()) and single exprn hence no {}

var mul1= x => x * x;   //concise syntax, implied "return"
console.log(mul1(8))

//------func with args and return -----------------
var mul2= (x,y) => { return x * y; }; //with block body, explicit "return" needed
console.log(mul2(3,4))

//--------func with args ----------------
var add1 = function (a:number, b:number) {  //traditional approach
    return a + b;
}

var add2 = (a:number, b:number) => {    //enclose func logic in {}
    return a + b;
}

var add3 = (a:number, b:number) => a + b;

console.log(add1(10,20))
console.log(add2(10,20))
console.log(add3(10,20))
```



```
no args!
64
12
30
30
30
```



```
//----------- lambda in JSON --------------
var obj = {
   x:10,
   f1: function(){ console.log("f1")},
   f2:() => console.log("f2"),
   f3:() => console.log(this.x, this),
   //returns undefined and {} since "this" is not automatically bound
   f4: function() {
       console.log(this.x, this);     //solution to above problem!
}}
obj.f1();
obj.f2();
obj.f3();
obj.f4();
```



```
f1
f2
undefined {}
10 {
  x: 10,
  f1: [Function: f1],
  f2: [Function: f2],
  f3: [Function: f3],
  f4: [Function: f4]
}
```

Arrays : Ways of creating array:

- var array_name[:datatype]; //declaration
- array_name = [val1,val2,valn..] //initialization
- var array_name[:data type] = [val1,val2…valn] //declaration+initialization
- var arr_name:data_type[] = new Array(size) //using the Array object
- var arr_name:data_type[] = new Array(val1, val2…) //comma separated values

```
var arr1:string[];
arr1= ["1","2","3","4"]
console.log(arr1[0]);  //1
```

```
var nums:number[] = [1,2,3,3]
console.log(nums[0]);  //1
```

```
var names:string[] = new
Array(“Joy",“Roy",“Leo”)
for(var i = 0;i<names.length;i++) {
   console.log(names[i])
}
```

```
var arr2:number[] = new Array(4)
for(var i = 0;i<arr2.length;i++) {
   arr2[i] = i * 2 ;
   console.log(arr2[i])
}
```

```
var arr:Array<number> = [1,2,3,4]; // using generics
```

- You can pass to the function a pointer to an array by specifying the array's name without an index.

Allow a function to return an array.

```
function disp():string[] {
   return new Array("Mary","Tom","Jack","Jill")
}
var nums:string[] = disp()
for(var i in nums) {
   console.log(nums[i])
}
```

```
var names:string[] = new Array("Mary","Tom","Jack","Jill")
function disp(arr_names:string[]) {
   for(var i = 0;i<arr_names.length;i++) {
      console.log(names[i])
   }
}
disp(names)
```

Array functions

- **concat()** :joins this array with two or more arrays. & returns a new array

```
var alpha = ["a", "b", "c"];  var numeric = [1, 2, 3];
var alphaNumeric = alpha.concat(numeric);  //a,b,c,1,2,3
```

- **filter()** : creates a new array with all elements that pass the test implemented by the provided function.

```
var nums = [1, 2, 3, 21, 22, 30];
var evens = nums.filter(i => i % 2 == 0);
```

```
function isBigEnough(element, index, array) {
  return (element >= 10);
}
var passed = [12, 5, 8, 130, 44].filter(isBigEnough);
console.log(passed );   //12,130,44
```

- pop() method removes the last element from an array and returns that element.
- shift() method removes the first element from an array and returns that element.
- push() method appends the given element(s) in the last of the array and returns the length of the new array.
- reverse() Reverses the order of the elements of an array
- slice() method Extracts a section of an array and returns a new array.
- join() : joins all the elements of an array into a string. array.join(separator);

```
var arr = new Array("First","Second","Third");
var str = arr.join();    //if no separator given, defaults to , (comma)
console.log("str : " + str );    //First,Second,Third
```

Array functions

- **forEach()** :calls a function for each element in the array and returns created array

```
var arr = ['a', 'b', 'c'];
arr.forEach(function(element) {
    console.log(element);
}); //a,b,c
```

- **map()** : creates a new array with the results of calling a provided function on every element in this array.

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
console.log("roots is : " + roots ); //1,2,3
```

```
var numbers = [1, 2, 3, 4];
var doubled = numbers.map(i => i * 2);
var doubled = [for (i of numbers) i * 2]; //same as above
console.log(doubled); // logs 2,4,6,8
```

for...of

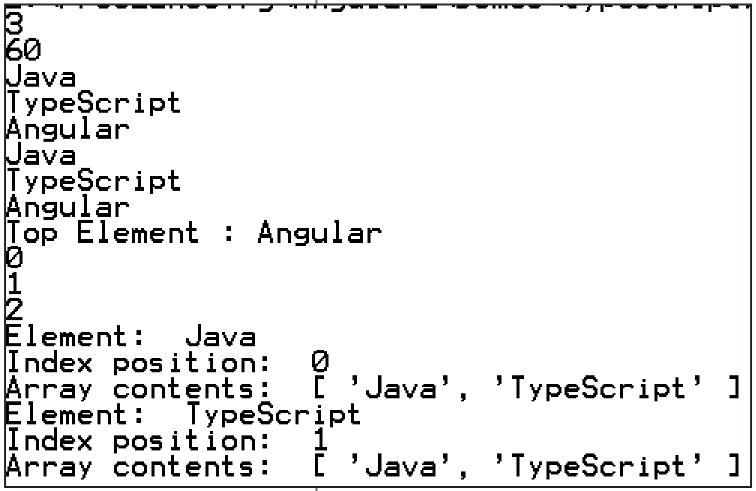
```
//In Javascript
var someArray = [9, 2, 5];
for (var i in someArray) {
    console.log(someArray[i]); // 0,1,2
}
```

```
// in TypeScript
var someArray = [9, 2, 5];
for (var item of someArray) {
    console.log(item); // 9,2,5
}
```

```
//TS can go going through a string character by character
var hello = “hello all";
for (var char of hello) {
    console.log(char); // hello all
}
```

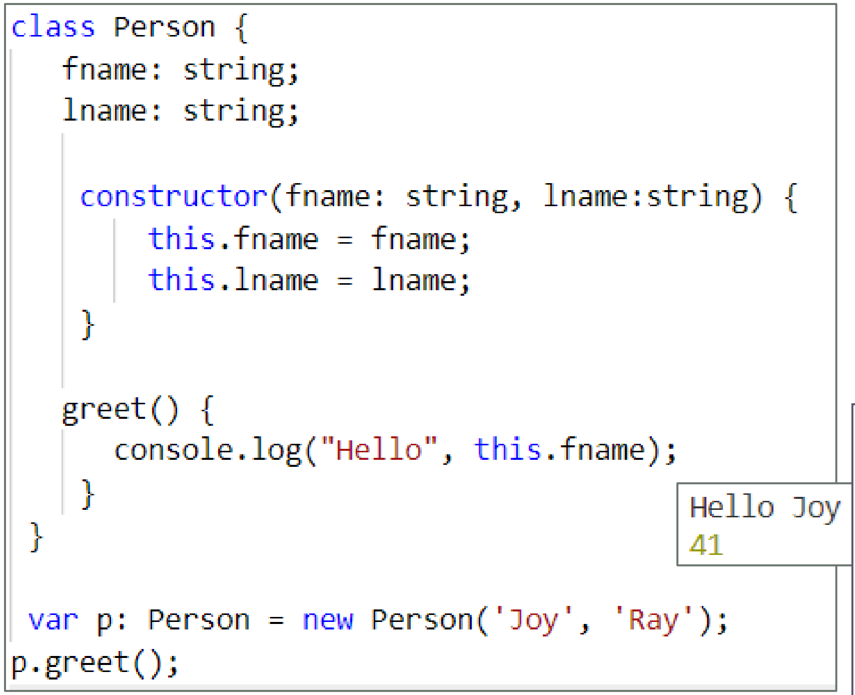
- If TypeScript can see that you are not using an array or a string it will give you a clear error *"is not an array type or a string type"*;

```
var marks = [60, 70, 66];
console.log(marks.length)
console.log(marks[0]);
var subjects = ["Java", "TypeScript", "Angular"];
for (var i = 0; i < subjects.length; i++)
      console.log(subjects[i]);
for (var sub of subjects) // Use iterator
      console.log(sub);
console.log("Top Element : " + subjects.pop());
var j;
var nums = [1001,1002,1003]
for(j in nums) {
   console.log(nums[j])
}
// Print all elements
subjects.forEach((v,idx,a) => {
    console.log("Element: ",v);
    console.log("Index position: ", idx);
    console.log("Array contents: ", a);
});

//v-element, idx-index pos, a-array contents
```



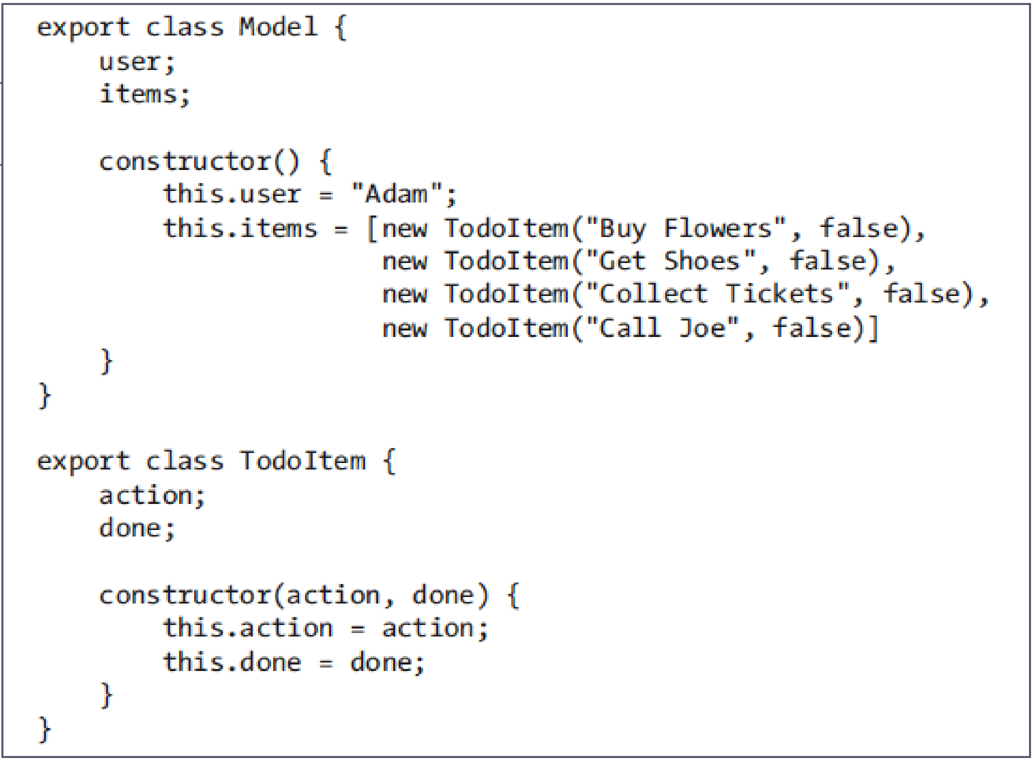
```
3
60
Java
TypeScript
Angular
Java
TypeScript
Angular
Top Element : Angular
0
1
2
Element:  Java
Index position:  0
Array contents:  [ 'Java', 'TypeScript' ]
Element:  TypeScript
Index position:  1
Array contents:  [ 'Java', 'TypeScript' ]
```

Typescript class



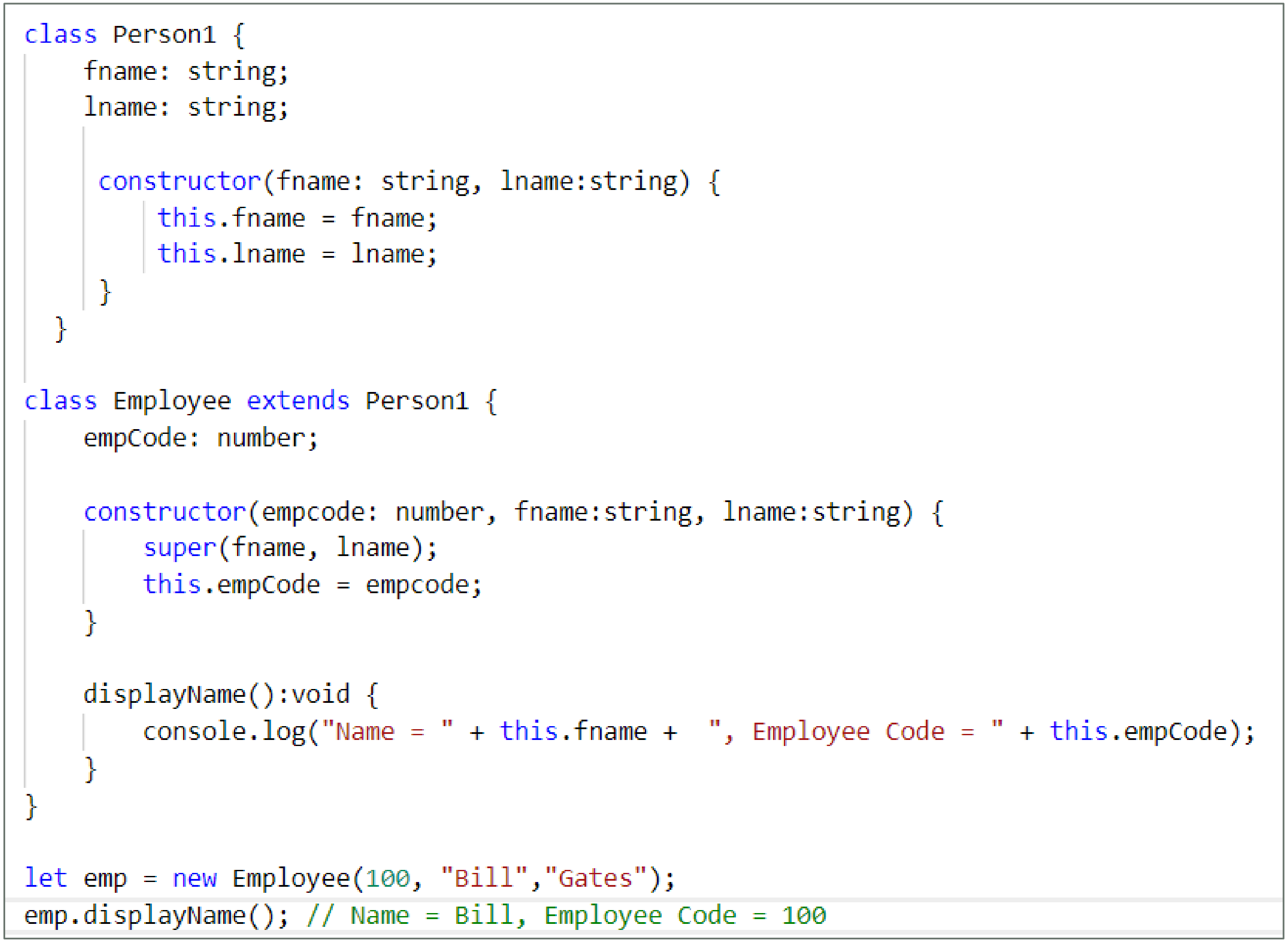
```
class Person {
   fname: string;
   lname: string;

    constructor(fname: string, lname:string) {
        this.fname = fname;
        this.lname = lname;
    }

   greet() {
      console.log("Hello", this.fname);
    }
 }

 var p: Person = new Person('Joy', 'Ray');
p.greet();
```

```
Hello Joy
41
```



```
export class Model {
    user;
    items;

    constructor() {
        this.user = "Adam";
        this.items = [new TodoItem("Buy Flowers", false),
                      new TodoItem("Get Shoes", false),
                      new TodoItem("Collect Tickets", false),
                      new TodoItem("Call Joe", false)]
    }
}

export class TodoItem {
    action;
    done;

    constructor(action, done) {
        this.action = action;
        this.done = done;
    }
}
```

Inheritance



```
class Person1 {
    fname: string;
    lname: string;

     constructor(fname: string, lname:string) {
         this.fname = fname;
         this.lname = lname;
     }
  }

class Employee extends Person1 {
    empCode: number;

    constructor(empcode: number, fname:string, lname:string) {
        super(fname, lname);
        this.empCode = empcode;
    }

    displayName():void {
        console.log("Name = " + this.fname +  ", Employee Code = " + this.empCode);
    }
}

let emp = new Employee(100, "Bill","Gates");
emp.displayName(); // Name = Bill, Employee Code = 100
```

static property

- Classes support static properties that are shared by all instances of the class. A natural place to put (and access) them is on the class itself
  - You can have static members as well as static functions

```
class MyClass{
    static instances = 0;
    constructor(){
        MyClass.instances++;
    }
}
var s1 = new MyClass();
var s2 = new MyClass();
console.log(MyClass.instances); // 2
```

```
class StaticMem {
   static num:number;

   static disp():void {
      console.log("The value of num:"+ StaticMem.num)
   }
}
StaticMem.num = 12     // initialize  the static variable
StaticMem.disp()      // invoke the static method
```

- instanceof operator : returns true if object belongs to the specified type.

```
class Person{}
var obj = new Person()
var isPerson = obj instanceof Person;   //true
console.log(" obj is an instance of Person " + isPerson);
```

Let

- Let allows to define variables with true block scope; unlike Javascript that recognises only function-scope, not block scope
  - That is if you use let instead of var you get a true unique element disconnected from what you might have defined outside the scope.

```
var foo = 123;
if (true) {
    var foo = 456;
}
console.log(foo); // 456
```

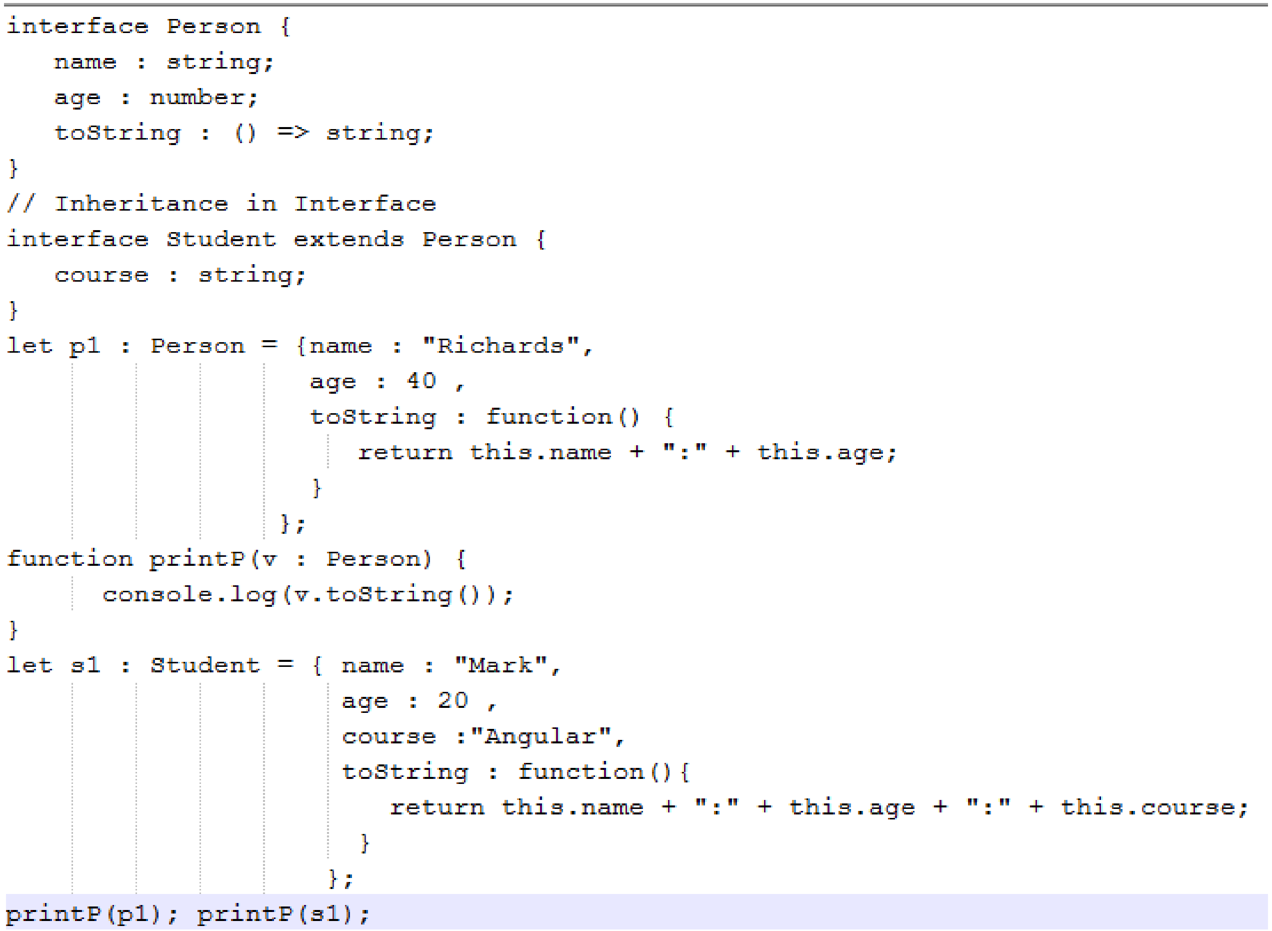
```
let foo = 123;
if (true) {
    let foo = 456;  //block scoped
}
console.log(foo); // 123
```

  - Another place where let would save you from errors is loops.
  - its better to use let whenever possible as it leads to lesser surprises for new and existing multi-lingual developers.

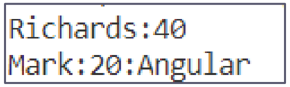
```
var index = 0;
var array = [1, 2, 3];
for (let index = 0; index < array.length; index++) {
    console.log(array[index]);
}
console.log(index); // 0
```

Interfaces

- An interface contains a collection of methods, properties and events.
  - Interfaces are TypeScript only constructs. They are not converted to JavaScript.
  - By default, all the members in an interface are public.
  - class [ClassName] implements [InterfaceName]



```
interface Person {
   name : string;
   age : number;
   toString : () => string;
}
// Inheritance in Interface
interface Student extends Person {
   course : string;
}
let p1 : Person = {name : "Richards",
                   age : 40 ,
                   toString : function() {
                      return this.name + ":" + this.age;
                   }
                 };
function printP(v : Person) {
      console.log(v.toString());
}
let s1 : Student = { name : "Mark",
                     age : 20 ,
                     course :"Angular",
                     toString : function(){
                        return this.name + ":" + this.age + ":" + this.course;
                      }
                    };
printP(p1); printP(s1);
```



```
Richards:40
Mark:20:Angular
```

const

- Const : offered by ES6 / TypeScript.
  - It allows you to be immutable with variables.
  - To use const just replace var with const: const foo = 123;

```
// Low readability
if (x > 10) {
}
```

```
// Better!
const maxRows = 10;
if (x > maxRows) { }
```

  - const declarations must be initialized : const foo; // ERROR
  - A const is block scoped:

```
const foo = 123;
if (true) {
    const foo = 456; // Allowed as its a new variable limited to this `if` block
}
```

  - A const works with object literals as well : const foo = { bar: 123 };
  - However it still allows sub properties of objects to be mutated:

```
const foo = { bar: 123 };
foo.bar = 456; // Allowed!
console.log(foo); // { bar: 456 }
```

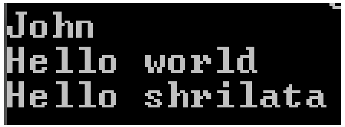
Modules

- Modules provide the possibility to group related logic, encapsulate it, structure your code and prevent pollution of the global namespace
  - Modules are executed within their own scope, not in the global scope
  - This means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported
  - Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported

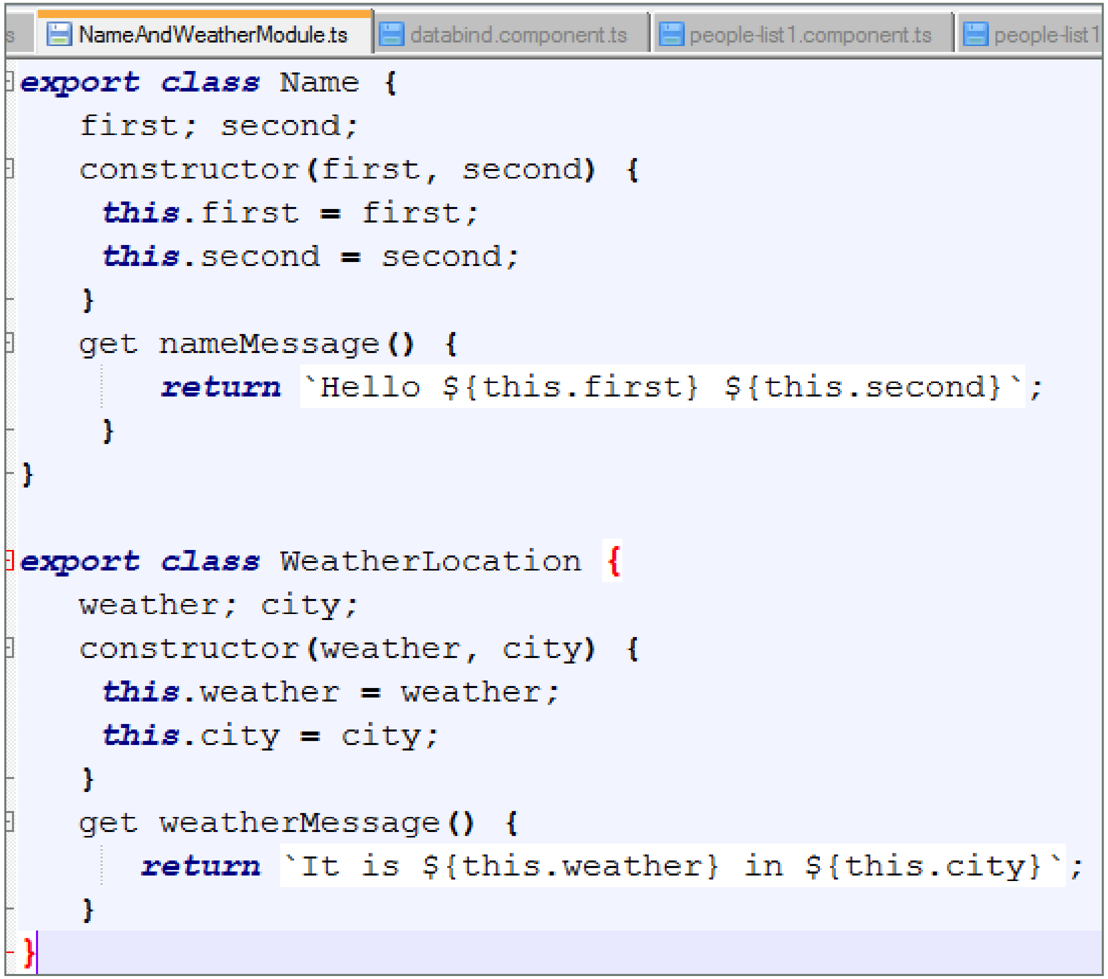
```
//module1.ts
export var fname:string = "John";
export function run() {  return "Hello world" }
export class Greeter {
    constructor(public msg:string){}
    greet(){
        console.log("Hello " + this.msg);
    }
}
```

Using an import in module2.ts not only allows you to bring in stuff from other files, but also marks the file module2.ts as a module

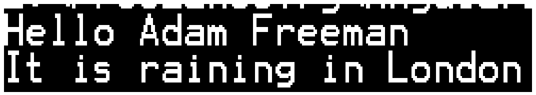
```
//module2.ts
import {fname} from "./module1";
import {run as r} from "./module1";
console.log(fname);

let r1 = r();
console.log(r1);

import {Greeter} from './module1';
export function run() {
    var greeter = new Greeter("shrilata");
    greeter.greet();
}
run();
```



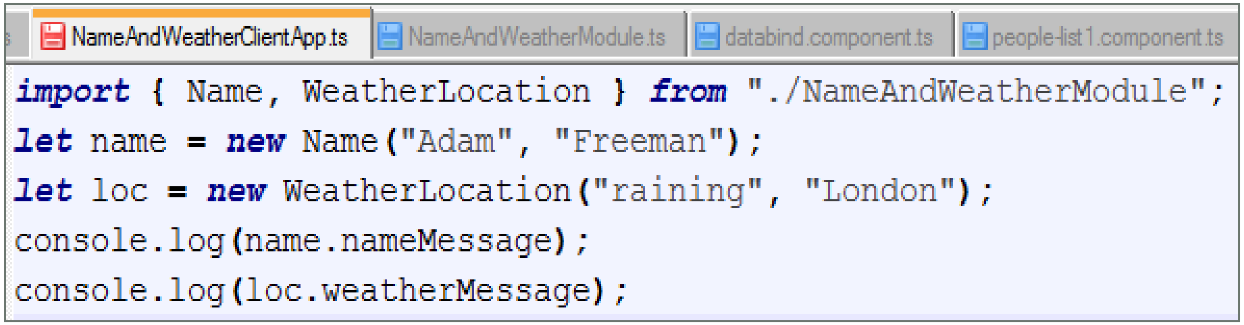
```
John
Hello world
Hello shrilata
```



```
export class Name {
   first; second;
   constructor(first, second) {
    this.first = first;
    this.second = second;
   }
   get nameMessage() {
      return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
   weather; city;
   constructor(weather, city) {
    this.weather = weather;
    this.city = city;
   }
   get weatherMessage() {
     return `It is ${this.weather} in ${this.city}`;
   }
}
```



```
Hello Adam Freeman
It is raining in London
```



```
import { Name, WeatherLocation } from "./NameAndWeatherModule";
let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
console.log(name.nameMessage);
console.log(loc.weatherMessage);
```