Module 5

THE EVOLUTION OF JAVASCRIPT

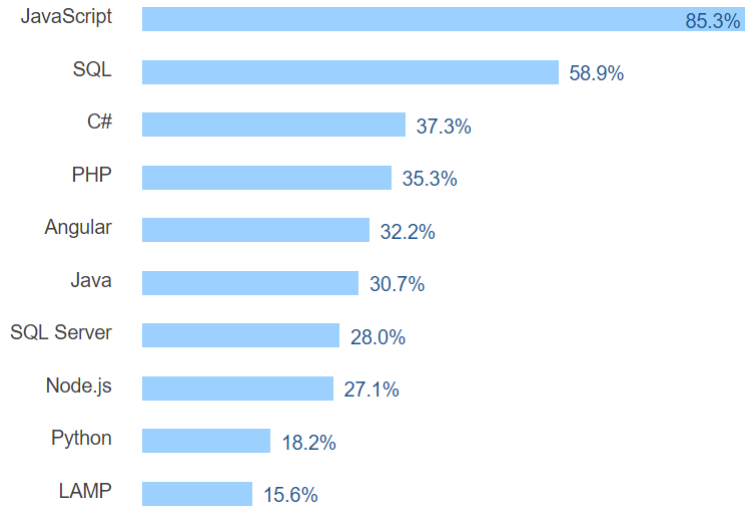
JavaScript

> JavaScript is the most commonly used programming language on earth.

> Even Back-End developers are more likely to use it than any other language.

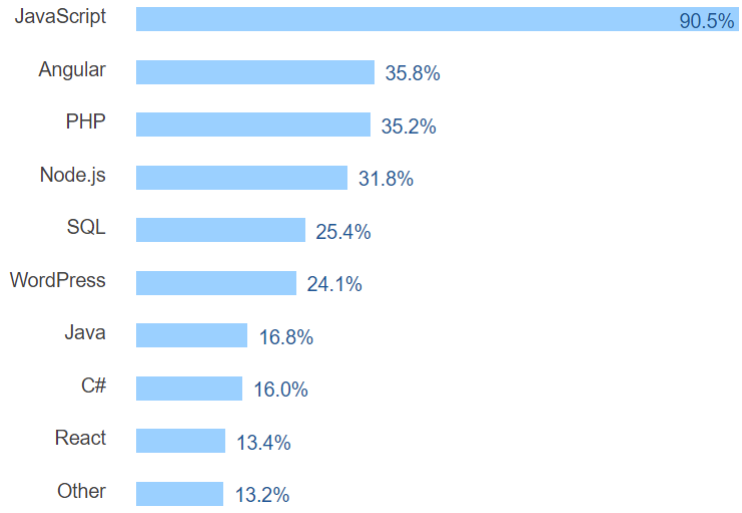
Most Popular Technologies

Fullstack



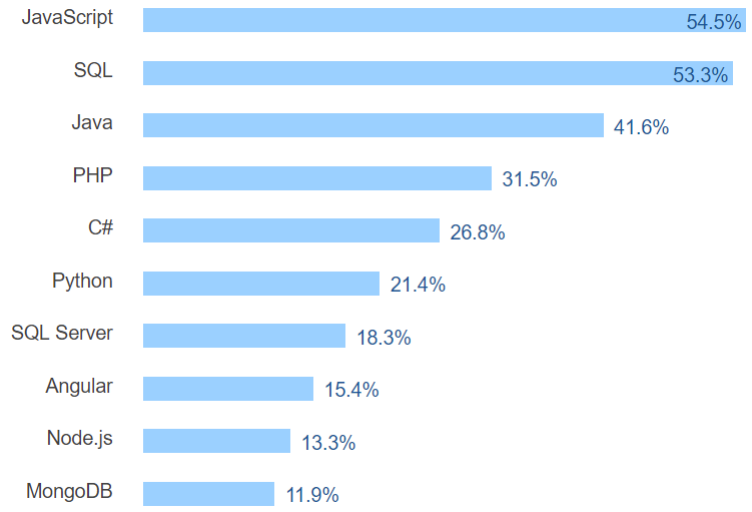
Most Popular Technologies

Frontend



Most Popular Technologies

Backend



Quiz #1

```
> 1<2<3
> 3>2>1
> [0]+1;
> [0]-1;
> [] == true;
> ![] == true;
> [] + [];
> [] + {};
> A = [0]
> A == A
> A == !A
> A = [null, undefined, [] ]
> A == ",,"
> Math.min < Math.max
```

Answers

```
> 1<2<3                    true
> 3>2>1                    false
> [0]+1;                   01
> [0]-1;                   -1
> [] == true;              false
> ![] == true;             false
> [] + [];                 ""
> [] + {};                 "[object Object]"
> A = [0]
> A == A                   true
> A == !A                  true
> A = [null, undefined, [] ]
> A == ",,"                true
> Math.min < Math.max      false
```

Quiz #2

```
function getLabel() {
  var count = 0,
      concatenation = ""
  ['dog','cow','rabbit']
        .forEach(function(animal,i){
               count = i+1
               concatenation += animal + ' '
         })
  return
 {
        'count' : count,
        'animals' : concatenation
  }
}
```

Answer

```
function getLabel() {
  var count = 0,
      concatenation = "" ;

  ['dog','cow','rabbit'].forEach(function(animal,i){
                count = i+1
                concatenation += animal + ' '
  })

  return
  {
        'count' : count,
        'animals' : concatenation
  }
}
```

ES5

ECMASCRIPT 5

JavaScript Objects

- A JavaScript object is an unordered collection of properties.
- Everything in a JavaScript program is an object.
- To make a new object:
  - x=new Object();
    - Makes a clone of the default Built-In Object called Object.
    - Assigns it to a global variable x.
    - x is empty except for the built-in properties and methods.

New Instances of Built-In Objects

- Create a new empty object and place a reference to that object in the global variable x

```
x = new Object();
// alternatively: x = {};
```

- Create an instance of a Number object with value 1 and place a reference to that object in the global variable one

```
one = new Number(1);
// alternatively: one = 1;
```

- Create an instance of a String object with a value "HelloWorld" and place a reference to that object in the global variable x

```
s = new String("Hello World")
// alternatively: s = "Hello World";
```

Instance Variables Are Called Properties

```
x = new Object();
x.top = 5;
```

> The assignment algorithm works like this:
  - Check if object x has a property named top
  - If it does not:
    - Add a new property to x
    - Name the property top
  - Overwrite the value of the property top with the new value, in this example: 5.

Object Literal Notation (1/2)

`x = new Object();` is the same as `x = {};`

```
x = new Object();
x.top = 5;
x.left = 10;
x.y = new Object();
```

is the same as

```
x = {
    top : 5,
    left : 10,
    y : {}
};
```

Object Literal Notation (2/2)

```
x = new Object();
x.top = 5;
x.left = 10;
x.y = new Object();
x.y.color = "red";
x.y.state = true;
```

is the same as

```
x = { top : 5,
      left : 10,
      y : { color: "red", state: true }
};
```

Accessing Properties (1/2)

> Dot notation is equivalent to Associative Array syntax

> Given the example so far:

```
x = {top:5, left:10, y:{}};
```

`x.top` is the same as `x['top']`

`x.left` is the same as `x["left"]`

`x.y` is the same as `x['y']`

Accessing Properties (2/2)

- A property name can be any value that can be evaluated as a string.
- For example, the following are legitimate properties in the JavaScript language.

```
x = new Object();
x[1] = "one";
x[1.1] = "one dot one";
x[-1] = false;
X['10']= 5;// Also accessed using x[10]
```

- **x** is not an not Array object.
  - The numbers are converted to strings, and used as property names.

Arrays

- Arrays extend the Object with an additional feature:
  - The ability to use an integer index to access member values.

Arrays – Basic Syntax

```
// creates an empty array object
a = new Array();
// the same a previous statement
a = new Array(0);
x = a.length; // assigns 0 to x
y = a[0]; // y gets value 'undefined'
// creates an array object with length 7 b = new
Array(7);
x = b.length; // assigns 7 to x
y = b[0]; // y gets 'undefined' value
...
y = b[6]; // y gets 'undefined' value
```

Length and Object Properties

> Arrays are objects with an additional feature:
  – The ability to use integer index to access members.

```
a = new Array();
//adds sting to 1st position in array
a[0] = "uno";
//adds an object to 2nd position in array
a[1] = {dos:2};
x = a.length; // assigns 2 to x
// puts a boolean value into PROPERTY
a['aim'] = true;
x = a.length; // assigns 2 to x
y = a.aim; // assigns true to y
```

Arrays – Object Literal Notation

> The following two array declarations are equivalent:

```
c = new Array("uno", "dos", "tres")
c = ["uno", "dos", "tres"];
```

> Square brackets indicates the declaration of an Array

Set Array Length Dynamically

```
// creates an array object with length 4
a = ["uno", "dos", "tres", "cuatro" ];
```

> Increasing the length adds positions with value undefined:

```
// adds 3 new positions: a[4],a[5],a[6]
a.length = 7;
```

> Decreasing the length deletes the extra positions:

```
// the array has 2 positions: a[0],a[1]
a.length = 2;
```

> Assigning a value to a new position sets the array's length to one less than that position:

```
a[50] = "cincuenta";
y = a.length // y is assigned value 51.
```

Multi-Dimensional Arrays

> Multi-dimensional arrays are arrays of arrays:

```
a =
[
  ["uno", "dos", "tres"],
  ["un", "deux","trois"],
  ["eins", "zwei", "drei"],
];
// y is assigned value "deux"
y = a[1][1];
```

Strings Are Objects

> Strings extend an Object with a string value.

Basic Syntax

> The value is not a property of the String Object.

```
x = new String("123");
x.status = true;
y = x; // y gets "123"
z = x.status // z gets true
```

> Typically, do not use a String's Object properties

Use Single or Double Quote Marks

> You create a string using the String object's constructor method or with a simple assignment.
> Equivalent initializations:

```
a = 'This is a String';
a = "This is a String";
a = new String('This is a String');
a = new String("This is a String");
```

Escaping Quote Marks

> To embed a single or double quote inside a string, use the other quotation marks, or precede the quote mark with a backslash.

> Equivalent:

```
b = "It's using single 'quotation' marks.";
b = 'It\'s using single \'quotation\' marks.';
```

The length Property

> The length property returns the number of characters in the string.

> For example:

```
c = "123";
d = c.length; // d gets value 3
```

> Assigning string length has no effect:

```
c = "123";
c.length = 2;
d = c.length; // d gets value 3
```

Concatenation

> The + operator is used to concatenate strings:

```
a = "one";
b = "two";
// c gets value "one and two"
c = a + " and " + b;
```

> Concatenation can cast Numbers to String

> Left-to-right operator precedence applies:

```
x = 2 + 3 + ","; // x gets value "5,"
```

Functions

> Functions extend objects with an additional feature:

– It can be executed.

JavaScript Functions Are Objects

> The JavaScript text of a function is called the function's body
> Functions have an additional feature:
  – The body is code that can be executed

```
function sayHello() {
  alert("Hello World");
}
// dialog appears with message
// "Hello World"
sayHello();
sayHello.top = 5;
// y is assigned value 5
y = sayHello.top;
```

Calls and Object References

> Parentheses appended to the function name indicate a call to the function.
> No parentheses appended to the function name, indicates a reference to the function object.

```
function one() {
  return 1; // returns the value 1
}
y = one(); // sets y to the value 1
z = one;
y = z(); // sets y to the value 1
one.two = 2;
alert( one() + one.two)
```

JavaScript Methods

> The ECMAScript specification states that a function stored in a property of another object is called a method.

```
function myArea( ) {
  return ( this.height * this.width);
}
function myRectangle( t, l, h, w) {
  this.top = t;
  this.left = l;
  this.height = h;
  this.width = w;
  this.area = myArea;
}
x = new myRectangle( 0, 0, 2, 3);
y = x.area(); // y is assigned value: 6
```

Function Parameters

> Parameters are the same as local variables.

```
function twiceAlert( msg1 , msg2 ){
  alert(msg1);
  alert(msg2);
}
```

> Parameters and local variables are completely separate from a Function object's properties.
  - Parameters and local variables belong to an instance of a call to the code in the Function body.
  - Properties belong to the Function object.

No Parameter Checking

> No compile-time or runtime argument checking:

```
function twiceAlert( msg1, msg2 ) {
  alert(msg1);
  alert(msg2);
}
twiceAlert('Hello World');
twiceAlert('Hello World', "Hello
Again",
"Ignored Greeting");
```

Constructors Are Functions

> A constructor is used to create new object instances.

> When a function is invoked with the new operator, that function is acting as a constructor in JavaScript.

```
function myPoint( x, y) {
  this.x = x;
  this.y = y;
}
x = new myPoint( 10, 27);
// x gets a new myPoint object
y = new myPoint( 17, 25);
// y gets a new myPoint object
alert( x.x + y.y);
```

Constructors Are Functions

> In a constructor, the keyword **`this`** refers to the new instance of the object.

```
function myPoint( x, y) {
  this.x = x;
  this.y = y;
  var z = 10;
  this.z = z;
}
```

Constructors Are Functions

> Parameters are local variables.
> Variables are not properties:
  – this.x is not the same as the parameter x;
  – this.y is not y.
  – this.z is not z.
> The declaration of an function object instantiates an original instance of that object.

Constructor Inner-Workings

1. Create a new empty object.
2. Add the constructor's prototype property to the new object's prototype chain, to resolve property references.
3. Execute the constructor's function body using the new object for the value of this, and using the current call scope.

> If Step 3 returns an object, return that object; otherwise return the newly created object (Step 1).

Prototype Inheritance

> When a property is accessed, the value of first instance of a property with that name found in the prototype chain is used.

```
function myPoint( x, y ) {
  this.x = x;
  this.y = y;
}
x = new myPoint(50, 100);
alert( x.x + "," + x.y+ "," + x.z);
myPoint.prototype.z = 150;
alert(x.x + "," + x.y+ "," + x.z );
alert( myPoint.x + "," + myPoint.y +
"," + myPoint.z);
```

Prototype Inheritance

> Altering a property value shadows other same-named properties in the Prototype chain.

```
function myPoint( x, y ) {
  this.x = x;
  this.y = y;
}
x = new myPoint(50, 100);
myPoint.prototype.z = 150;
alert( x.z );
x.z = 33;
alert( x.z);
// dialog appears with message: 150
alert( myPoint.prototype.z);
```

Prototype Inheritance

> All of the properties of the constructor's prototype are available to the child objects

> Even methods that are added to the constructor's prototype after the children were instantiated.

```
function myPoint( x, y ) {
  this.x = x;
  this.y = y;
}
x = new myPoint(50, 100);
x.helloWorld() // Error
myPoint.prototype.helloWorld =
function(){ ("x=" + this.x +", y="
            + this.y))};
x.helloWorld();
```

Prototype Inheritance

> Prototypical inheritance even applies to Built-In Objects.

```
x = new String("ABCDEFG");
x.helloWorld() // Error
String.prototype.helloWorld =
function() {alert("Hello World")};
x.helloWorld(); // alerts: Hello World.
```

The Function Keywords and Objects

> **`function`** (lower case f) is a reserved keyword like new.
> The function keyword is not the same as the name of the **`Function`** (upper case F) built-In Object.
> The **`function`** keyword is used as part of a special syntax to declare a new instance of **`Function`** Objects.
> Typically, do not use **`Function`**

The Function Keywords and Objects

- **`function`** (lower case f) is a reserved keyword like new.
- The function keyword is not the same as the name of the **`Function`** (upper case F) built-In Object.
- The **`function`** keyword is used as part of a special syntax to declare a new instance of **`Function`** Objects.
- Typically, do not use **`Function`**

Creating Objects

- Creating an empty object:

```
var obj_1 = {};
var obj_2 = new Object();
var obj_3 = Object.create(null);
```

- Creating an object:

```
var person = {
    "full-name" : "John Doe",
    age: 35,
     address: {
        address_line1: "Clear Trace, Glaslyn, Arkansas",
        "postal code": "76588-89"
    }
};
```

Creating Objects

> Creating an object with new:

```
function Tree(type, height, age) {
        this.type = type;
        this.height = height;
        this.age = age;
}

var mapleTree =
    new Tree("Big Leaf Maple", 80, 50);
```

Accessing Object Properties

```
var myObject = {
   name: "luggage",
   length: 75,
   specs: {
    material: "leather",
    waterProof: true
   }
}
```

> Getting and setting properties:

```
myObject["name"];                  // "luggage"
myObject.name;                     // "luggage"
myObject.specs.material;           // "leather"
myObject["specs"]["material"];     // "leather"
myObject.width;                    // undefined
myObject.tags.number;              // TypeError thrown
myObject["name"] = "suitcase";     // name : "suitcase"
myObject.name = "suitcase";        // name : "suitcase"
myObject.width = 40;               // creates a new property
myObject.tags.number = 6;          // TypeError thrown
```

Object Methods

> Object method example:

```
var myObj = {
        print: function() {
            console.log("Hello World!");
        }
};

myObj.print();
```

Creating Arrays

> Initializing an array:

```
var myNewEmptyArray = [];
var numbers = [1, 2, 3, 4, 'five'];
```

> Creating an array with new:

```
var myNewEmptyArray = new Array();
var myNewNonEmptyArray = new Array(15);
var numbers = new Array(1, 2, 3, 4, 'five');
```

Iterating Through Objects and Arrays

> Iterating object properties:

```
for(var key in obj) {
  if (obj.hasOwnProperty(key)) {
    // no parent properties
    console.log(obj[key]);
  }
}
```

> Iterating arrays:

```
for (var i = 0; i < c.length; i++) {
        console.log(c[i]);
}
```

JavaScript Arrays as Collections

> To modify arrays and treat them as collections, you can use several methods:

- `array.pop()` removes and returns the last element.
- `array.push(value)` adds the value at the end.
- `array.shift()` removes and returns the first element.
- `array.unshift(value)` adds the value at the start.
- `array.indexOf(value)` gets the index of value.
- `array.join(", ")` joins all items in a single string.

JavaScript Arrays as Collections

- To modify arrays and treat them as collections, you can use several methods:
  - `array.sort(function)` sorts all items by using the provided function. The function takes two arguments and returns less than 0 if the first argument comes first, 0 if the arguments are equal, and greater than 0 if the second comes first.

JavaScript RegExp Object

- A regular expression is an object that describes a pattern of characters.
- When you search in a text, you can use a pattern to describe what you are searching for.
- A simple pattern can be one single character.
- A more complicated pattern can consist of more characters, and can be used for parsing, format checking, substitution and more.
- Regular expressions are used to perform powerful pattern-matching and "search-and-replace" functions on text.

Example

```
var pat1=new RegExp("e");
document.write(pat1.test("The best things in life are free"));
var pat2=new RegExp("e");
document.write(pat2.exec("The best things in life are free"));
```

Modifiers

Modifier	Description
I	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

Brackets

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character NOT between the brackets
[0-9]	Find any digit between the brackets
[^0-9]	Find any digit NOT between the brackets
(x|y)	Find any of the alternatives specified

Metacharacters

Metacharacter	Description
.	Find a single character
\w	Find a word character
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character
\s	Find a whitespace character
\S	Find a non-whitespace character
\b	Find a match at the beginning/end of a word
\B	Find a match not at the beginning/end of a word
\0	Find a NUL character
\n	Find a new line character
\f	Find a form feed character

Metacharacters

Metacharacter	Description
\r	Find a carriage return character
\t	Find a tab character
\v	Find a vertical tab character
\xxx	Find the character specified by an octal number xxx
\xdd	Find the character specified by a hexadecimal number dd
\uxxxx	Find the Unicode character specified by a hexadecimal number xxxx

Quantifiers

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero/more occurrences of n
n?	Matches any string that contains zero/one occurrences of n
n{X}	Matches any string that contains a sequence of X n's
n{X,Y}	Matches any string that contains a sequence of X to Y n's
n{X,}	Matches any string that contains a sequence of at least X n's
n$	Matches any string with n at the end of it
^n	Matches any string with n at the beginning of it
?=n	Matches any string that is followed by a specific string n
?!n	Matches any string that is not followed by a specific string n

RegExp Object Methods

Method	Description
compile()	Deprecated in version 1.5. Compiles a regular expression
exec()	Tests for a match in a string. Returns the first match
test()	Tests for a match in a string. Returns true or false
toString()	Returns the string value of the regular expression

Example

```
meyveler= [ "elma", "armut", "kiraz", "karpuz",
            "kavun", "muz"]
pattern= new RegExp("^k.*z$")
for (meyve in meyveler)
     if( pattern.test(meyveler[meyve]) )
        console.log(meyveler[meyve]);
pattern= new RegExp("^[a-z]{1,4}$")
for (meyve in meyveler)
     if( pattern.test(meyveler[meyve]) )
        console.log(meyveler[meyve]);
```

Object Cloning

```
function clone(obj) {
   var target = {};
   for (var i in obj) {
    if (obj.hasOwnProperty(i)) {
     target[i] = obj[i];
    }
   }
   return target;
  }
```

Object Cloning

```
var oldObject = {
   a: 1,
   b: 2,
   c: 3,
   d: 4,
   e: 5,
   f: function() {
    return 6;
   },
   g: [7, 8, 9]
  };
```

Object Cloning

```
var newObject= clone(oldObject);
```

Object Cloning

```
newObject= JSON.parse(
    JSON.stringify( oldObject)
);
```

Object Cloning

```
var newObject = {};
jQuery.extend(newObject, oldObject);
```

Object Cloning

```
newObject= {} ;
newObject= Object.create(oldObject);
```

How to remove a property

```
function Circle(x,y,radius){
    this.x= x;
    this.y= y;
    this.radius= radius;
    this.area= function(){
        return 3.1415 * radius * radius;
    }
}
```

How to remove a property

```
delete Circle.prototype.x
unitCircle= new Circle(0,0,1);
console.log(unitCircle.area());
delete unitCircle.area;
console.log(unitCircle.area());
undefined is not a function
```

ECMASCRIPT 6

ECMAScript 6.0

- Const
  - **const PI = 3.141593**
- Binary and octal literals
  - **0b1101010010**
  - **0o03427**
- Template strings

```
message = `Hello ${customer.name},
want to buy ${card.amount} ${card.product} for
a total of ${card.amount * card.unitprice} bucks?`
```

- Tagged template strings

```
String.raw`This is a text
with multiple lines.
Escapes are not interpreted,
\n is not a newline.`;
```

ECMAScript 6

- Default parameters
  - function f (x, y = 7, z = 42) {
    return x + y + z
    }
    f(1) === 50
- Rest Parameter
  - function f (x, y, ...a) { return (x + y) * a.length }
    f(1, 2, "hello", true, 7) === 9
- Spread
  - var params = ["hello", true, 7];
    var other = [1, 2, ...params];
    // [1, 2, "hello", true, 7]
    f(1, 2, ...params) === 9;
    var str = "foo";
    var chars = [...str]; // ["f", "o", "o"]

ECMAScript 6

- Lexical **this**

```
this.nums.forEach(
   (v) => {
            if (v % 5 === 0)
             this.fives.push(v)
})
```

- Earlier

```
var self = this;
this.nums.forEach(function (v) {
       if (v % 5 === 0)
            self.fives.push(v);
       }
);
```

ECMAScript 6

> Variable with block scope

```
for (let i = 0; i < a.length; i++){
    let x = a[i]
}
```

> Functions with block scope

```
{
    function foo () { return 1 }
    foo() === 1 {
      function foo () { return 2 }
      foo() === 2
    }
    foo() === 1
}
```

ECMAScript 6

> ES5 emulation

```
(function () {
    var foo = function () { return 1; }
    foo() === 1;
    (function () {
        var foo = function () {
            return 2;
        }
        foo() === 2;
    })();
    foo() === 1;
})();
```

ECMAScript 6

> =>

```
pairs = evens.map(
   v => ({ even: v, odd: v + 1  })
);
nums.forEach(v => {
  if (v % 5 === 0) fives.push(v)
})
```

> Simplified notation when creating objects

```
obj = { x, y, foo(){} }
obj.x
obj.foo()
```

ECMAScript 6

> Type Symbol

Symbol() !=== Symbol()

Symbol("foo") !== Symbol("foo")

ECMAScript 6

> Decomposition of structures, complex assignments

```
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
[ b, a ] = [ a, b ]

var { op, lhs, rhs } = getASTNode()
var { op:a, lhs:{ op: b }, rhs:c } = getASTNode()
```

> Decomposition vs. default values

```
var list = [ 7, 42 ]
var [ a = 1, b = 2, c = 3, d ] = list
a === 7
b === 42
c === 3
d === undefined
```

ES6 – set/weakSet map/weakMap

```
let s = new Set()
s.add("hello").add("goodbye").add("hello")
s.size === 2
s.has("hello") === true
for (let key of s.values()) // insertion order
  console.log(key)

let m = new Map()
m.set("hello", 42)
m.set(s, 34)
m.get(s) === 34
m.size === 2
for (let [ key, val ] of m.entries())
   console.log(key + " = " + val)
```

ES6 – Array/String – New Methods

> Array

```
[ 1, 3, 4, 2 ].find(x => x > 3) // 4
```

> String

```
"foo".repeat(3)
"hello".startsWith("ello", 1)    // true
"hello".endsWith("hell", 4)      // true
"hello".includes("ell")          // true
"hello".includes("ell", 1)       // true
"hello".includes("ell", 2)       // false
```

ECMAScript 6.0 –Proxy

```
// Proxying a normal object
var target = {};
var handler = {
  get: function (receiver, name) {
    return `Hello, ${name}!`;
  }
};

var p = new Proxy(target, handler);
p.world === "Hello, world!";
```

ECMAScript 6.0 – Reflection

- has, deleteProperty, defineProperty
- ownKeys
- getPrototypeOf, setPrototypeOf
- preventExtensions, isExtensible

ES6 Classes

```
class Shape {
   constructor (id, x, y) {
     this.id = id;
     this.move(x, y);
   }
   move (x, y) {
     this.x = x;
     this.y = y;
   }
}
class Rectangle extends Shape {
   constructor (id, x, y, width, height) {
      super(id, x, y);
      this.width  = width;
      this.height = height;
   }
}
```

ES6 Classes: static components

```
class Rectangle extends Shape {
…
  static defaultRectangle () {
    return new Rectangle("default", 0, 0, 100, 100)
  }
  get dxdy() { return this.x * this.y }
}

var r = Rectangle.defaultRectangle()
r.dxdy === 10000
```

Modules in ES5

- **AMD -** Asynchronous Module Definition
  - **JQuery, DoJo**
- **CommonJS**
  - **Curl (client), NodeJs**
- **ES Harmony**
  - **Base for modules in ES6**

Solution:

ECMAScript 6 Modules

```
//  lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

//  someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))

//  otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

ECMAScript 6.0 - generatory

```
Function *range (start, end, step) {
   while (start < end) {
      yield start start += step
   }
}

for (let i of range(0, 10, 2)) {
   console.log(i) // 0, 2, 4, 6, 8
}
```

```
function msgAfterTimeout (msg, who, timeout) {
   return new Promise((resolve, reject) =>
   {
      setTimeout( () =>
       resolve(`${msg} Hello ${who}!`),timeout)
   })
}

msgAfterTimeout("", "Foo", 100)
 .then( (msg) =>
     msgAfterTimeout(msg, "Bar", 200),
       (msg) =>{
        console.log(`done after 300ms:${msg}`)
     });
```

ECMAScript 6.0 – Promises

```
function msgAfterTimeout (msg, who, timeout) {
   return new Promise((resolve, reject) =>
   {
      setTimeout(()=>resolve(`${msg} Hello ${who}!`),
                                 timeout)
   })
}

msgAfterTimeout("", "Foo", 100)
 .then( (msg) => msgAfterTimeout(msg, "Bar", 200),
        (msg) =>{ console.log(`done after 300ms:${msg}`)}
      );
```

ECMAScript 6.0

> Transcompiler:
  http://babeljs.io/

PROMISES

Promisses ES6 – plain old callback

```
fs.readFile('countries.json', 'utf8', (err, data) => {
    if (err) {
        console.error("error while reading json file!");
        throw err;
    }

    countries = JSON.parse(data);

    // Mapped and Reduced Countries
    mappedCountries = countries.map(country => {
        return {
            "code": country._id,
            "name": country.name,
            "population": country.population,
            "surfaceArea": country.surfaceArea,
            "continent": country.continent
        }
    }).reduce((list, country) => {
        list.push(country);
        list[country.code] = country;
        return list;
    }, []);


});
```

Promises ES6: callbacks – what is wrong?

```
1  fs.readdir(source, function (err, files) {
2    if (err) {
3      console.log('Error finding files: ' + err)
4    } else {
5      files.forEach(function (filename, fileIndex) {
6        console.log(filename)
7        gm(source + filename).size(function (err, values) {
8          if (err) {
9            console.log('Error identifying file size: ' + err)
10         } else {
11           console.log(filename + ' : ' + values)
12           aspect = (values.width / values.height)
13           widths.forEach(function (width, widthIndex) {
14             height = Math.round(width / aspect)
15             console.log('resizing ' + filename + 'to ' + height + 'x' + height)
16             this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
17               if (err) console.log('Error writing file: ' + err)
18             })
19           }.bind(this))
20         }
21       })
22     })
23   }
24 })
25
```

Promises ES6 – promise

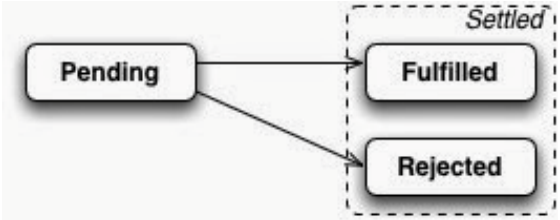
> Promise.all(iterable)
> Promise.race(iterable)
> Promise.reject(reason)
> Promise.resolve(value)

```
readFilePromisified('config.json')
    .then(function (text) { // (A)
        var obj = JSON.parse(text);
        console.log(JSON.stringify(obj, null, 4));
    })
    .catch(function (reason) { // (B)
        // File read error or JSON SyntaxError
        console.error('An error occurred', reason);
    })
```

> Promise.prototype.catch()
> Promise.prototype.then()

Promises ES6 – promise

> A promise states:
  – **Pending**: the result hasn't been computed, yet
  – **Fulfilled**: the result was computed successfully
  – **Rejected**: a failure occurred during computation



Promises ES6 – Promise.all

> Promise.all waits for all fulfillments (or the first rejection).

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, "one");
});
var p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, "two");
});
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 3000, "three");
});
var p4 = new Promise((resolve, reject) => {
  setTimeout(resolve, 4000, "four");
});
var p5 = new Promise((resolve, reject) => {
  reject("reject");
});

Promise.all([p1, p2, p3, p4, p5]).then(value => {
  console.log(value);
}, reason => {
  console.log(reason)
});
```

ADVANCED ES6 AND ES7 FEATURES

Generators

> Enable us to get an item on demand
  - Do not require waiting until all items have been generated before using one item
  - Allow us to have infinite sequences
  - Reduce memory usage

Generators

```
var generator= function* (){
  for (let i=0;i<1000;++i) yield i;
}
var futures= generator();
do {
  received= futures.next();
  console.log(received.value);
} while(! received.done);
```

TYPESCRIPT

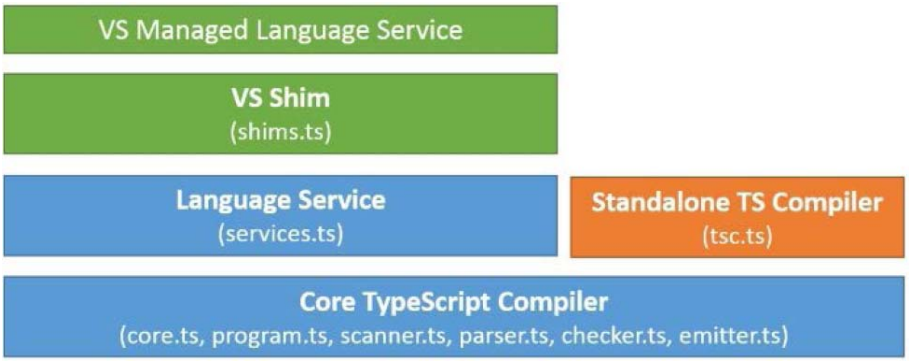
The TypeScript architecture

Design Goals

- Statically identify JavaScript constructs that are likely to be errors.
- High compatibility with the existing JavaScript code
  - TypeScript is a superset of JavaScript
- Provide a structuring mechanism for larger pieces of code.
- TypeScript adds class-based object orientation, interfaces, and modules.
- Impose no runtime overhead on emitted programs.
- Align with the current and future ECMAScript proposals
- Be a cross-platform development tool
  - Microsoft released it under open source Apache license

TypeScript components

> The TypeScript language is internally divided into three main layers



TypeScript components

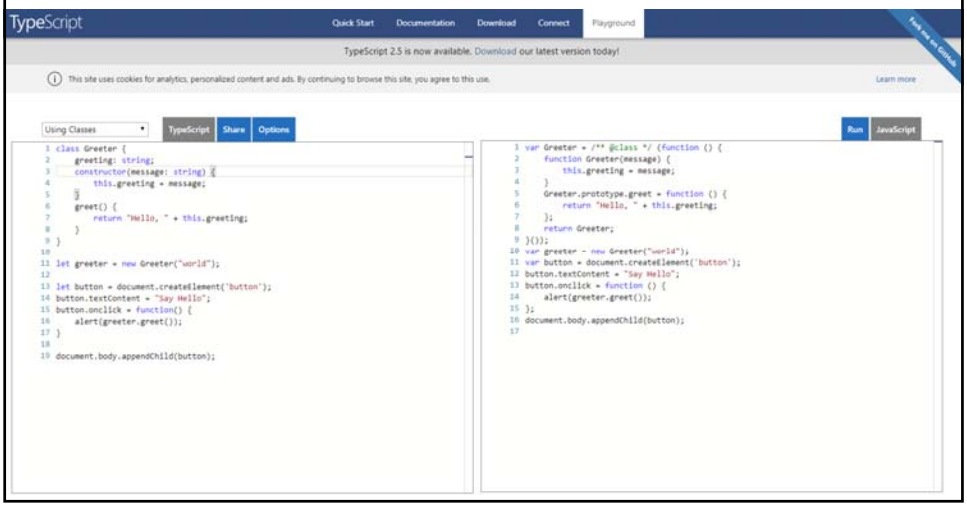
Each of these main layers has a different purpose:

> The language
  – Features the TypeScript language elements.
> The compiler
  – Performs the parsing, type checking, and transformation of your TypeScript code to JavaScript code.
> The language services
  – Generates information that helps editors and other tools provide better assistance features such as IntelliSense or automated refactoring.
> IDE integration

TypeScript language features

> The easiest and fastest way to start writing some TypeScript code is to use the online editor

http://www.typescriptlang.org/Playground



npm

> Node.js package manager

npm install --save typescript

> Compiling

tsc test.ts

– Produces **test.js**

> **tsc –w test.js**

– watches for changes and recompiles

> **npm install --save ts-node**

> **ts-node test.ts**

Types

- TypeScript is a typed superset of JavaScript
- TypeScript added optional static type annotations to JavaScript in order to transform it into a strongly typed programming language.
- Strong typing allows the programmer to express his intentions in his code, both to himself and to others in the development team.
- Typescript's type analysis ***occurs entirely at compile time*** and adds ***no runtime overhead*** to program execution.

Optional static type notation

- For a variable, the type notation comes after the variable name and is preceded by a colon:

```
var counter; // unknown (any) type
var counter = 0; // number (inferred)
var counter : number; // number
var counter : number = 0; // number
```

Basic types

Data Type	Description
Boolean	Whereas the string and number data types can have a virtually unlimited number of different values, the Boolean data type can only have two. They are the literals `true` and `false`. A Boolean value is a truth value; it specifies whether the condition is true or not. `var isDone: boolean = false;`
Number	As in JavaScript, all numbers in TypeScript are floating point values. These floating-point numbers get the type `number`. `var height: number = 6;`

String	You use the string data type to represent text in TypeScript. You include string literals in your scripts by enclosing them in single or double quotation marks. Double quotation marks can be contained in strings surrounded by single quotation marks, and single quotation marks can be contained in strings surrounded by double quotation marks.
	`var name: string = "bob";`
	`name = 'smith';`

Basic types

Data Type	Description

Array	TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of the two ways. In the first, you use the type of the elements followed by `[]` to denote an array of that element type:
	`var list:number[] = [1, 2, 3];`
	The second way uses a generic array type, `Array`:
	`var list:Array<number> = [1, 2, 3];`

Enum	An enum is a way of giving more friendly names to sets of numeric values. By default, enums begin numbering their members starting at 0, but you can change this by manually setting the value of one to its members.
	`enum Color {Red, Green, Blue};`
	`var c: Color = Color.Green;`

Basic types

Data Type	Description

Any	The any type is used to represent any JavaScript value. A value of the any type supports the same operations as a value in JavaScript and minimal static type checking is performed for operations on any values.
	`var notSure: any = 4;`
	`notSure = "maybe a string instead";`
	`notSure = false; // okay, definitely a boolean`
	The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt in and opt out of type checking during compilation. The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:
	`var list:any[] = [1, true, "free"];`
	`list[1] = 100;`

Void	The opposite in some ways to any is `void`, the absence of having any type at all. You will see this as the return type of functions that do not return a value.
	`function warnUser(): void {`
	`  alert("This is my warning message");`
	`}`

Basic types

> The value null is a literal: representation of no value.

```
var TestVar; // variable is not initialized
alert(TestVar); // shows undefined
alert(typeof TestVar); // shows undefined
var TestVar = null;
alert(TestVar); // shows null
alert(typeof TestVar); // shows object
```

> In TypeScript, we will not be able to use null or undefined as types:

```
// Error, Type expected
var TestVar : null;
// Error, cannot find name undefined
var TestVar : undefined;
```

var, let, and const

> Variables declared with **var** are scoped to the nearest function block

```
var mynum : number = 1;
```

> Variables declared with let are scoped to the nearest enclosing block

```
let isValid : boolean = true;
```

> The const keyword creates a constant that can be global or local to the block in which it is declared.

```
const apiKey : string =
       "0E5CE8BD-6341-4CC2-904D-C4A94ACD276E";
```

Union types

> Union types are used to declare a variable that is able to store a value of two or more types.

```
var path : string[]|string;
path = '/temp/log.xml';
path = ['/temp/log.xml', '/temp/errors.xml'];
path = 1; // Error
```

Type guards

```
var x: any = { /* ... */ };
if(typeof x === 'string') {
  // Error, 'splice' does not exist on 'string'
  console.log(x.splice(3, 1));
}
// x is still any
x.foo(); // OK
```

Type aliases

> TypeScript allows us to declare type aliases by using the **type** keyword:

```
type MyArray = Array<string|number|boolean>;
type MyNumber = number;
type NgScope = ng.IScope;
type Callback = () => void;
```

> Type aliases are exactly the same as their original types
  – Simply alternative names.

Classes

> ES6 adds class-based object orientation to JavaScript

> TypeScript is based on ES6