JavaScript Programming

Tiep Phan – SSE – July 2018

NASH TECH
The Power to Innovate

Agenda

- Course overview.
- Course contents.
- Examination.

Course Overview

- Introduction to JavaScript.
- Why learn JavaScript?
- The fundamentals of JavaScript.
- JavaScript new features: ES2015 → ES2018.

Course Contents

- Section 1: Data type, Operators, Variables, Scope, Program Structure.
- Section 2: Function, Higher-Order Function, Object/Class, Array.
- Section 3: Asynchronous Programming, Events, Block scope variables, Constant.
- Section 4: Arrow functions, Default parameter value, Rest, Spread, Class, Module.
- Section 5: Promise, Async/Await.

Section 1

Data type, Operators, Variables, Scope, Program Structure

Data Type

- Primitive Values.
- Objects.

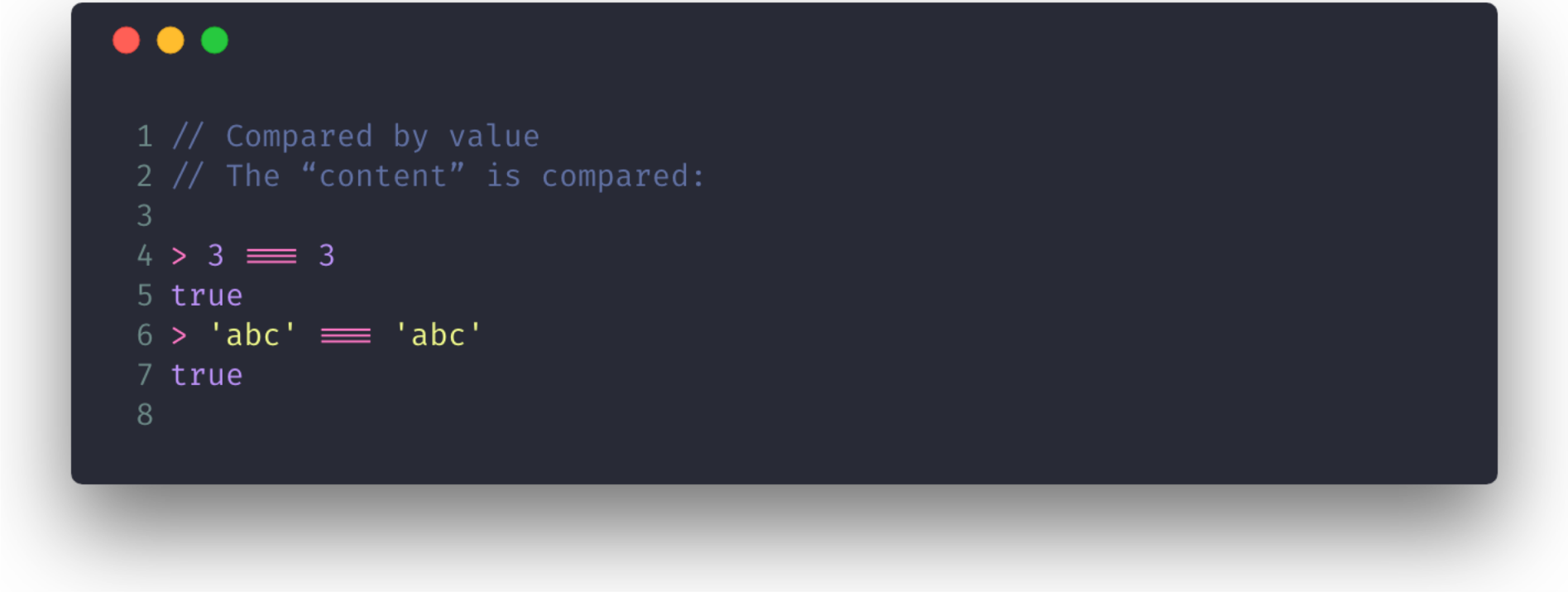
Primitive Values

- Boolean: true/false
- Number: 10, 3.14, …
- String: ‘abc’, …
- Null
- Undefined
- Symbol (ES2015)

Note: You can’t define your own primitive types.

Primitive Values

- Compared by value



```
1 // Compared by value
2 // The “content” is compared:
3
4 > 3 === 3
5 true
6 > 'abc' === 'abc'
7 true
8
```

Primitive Values

- Always immutable

```
// Always immutable
// Properties can't be changed, added, or removed:

> var str = 'abc';

> str.length = 1; // try to change property `length`
> str.length      // ⇒ no effect
3

> str.foo = 3; // try to create property `foo`
> str.foo      // ⇒ no effect, unknown property
undefined

```

Objects

- All nonprimitive values are *objects*.

```
// Plain Objects
var programmingLanguage = {
    name: 'JavaScript',
    spec: 'ECMAScript'
}

// Arrays, Map, Set, ...
var fruits = [ 'apple', 'banana', 'cherry' ]

// Regular Expressions
var regex = /^a+b+$/gi

```

Objects

- Compared by reference

```
> ({} === {})  // two different empty objects
false

> var obj1 = {};
> var obj2 = obj1;
> obj1 === obj2
true

```

Objects

- Mutable by default

```
> var obj = {};
> obj.k = 123; // add property `k`
> obj.k
123

> obj.v = 'xyz'; // add property `v`
> obj.v
xyz
```

Objects

- User-extensible

```
1  // Custom class
2  function Person(name) {
3      this.name = name;
4  }
5
6  Person.prototype.getInfo = function () {
7      return 'Person named ' + this.name;
8  };
9
10 > var me = new Person('Tiep');
11 > me.getInfo();
12 Person named Tiep
13
```

Null and Undefined

- *undefined* means “no value” (neither primitive nor object).
  - Uninitialized variables
  - Missing parameters
  - Missing properties have that nonvalue
  - Functions implicitly return it if nothing has been explicitly returned.
- *null* means “no object.” It is used as a nonvalue where an object is expected (as a parameter, as a member in a chain of objects, etc.).

Null and Undefined

- *undefined* and *null* are the only values for which any kind of property access results in an exception.

```
> function getName(x) { return x.name }

> getName(true)
undefined
> getName(0)
undefined

> getName(null)
TypeError: Cannot read property 'name' of null
> getName(undefined)
TypeError: Cannot read property 'name' of undefined
```

Null and Undefined

- *undefined* is not a keyword in JavaScript.

Null and Undefined

- *undefined* is not a keyword in JavaScript.

Wrapper Objects for Primitives

- Boolean
- Number
- String

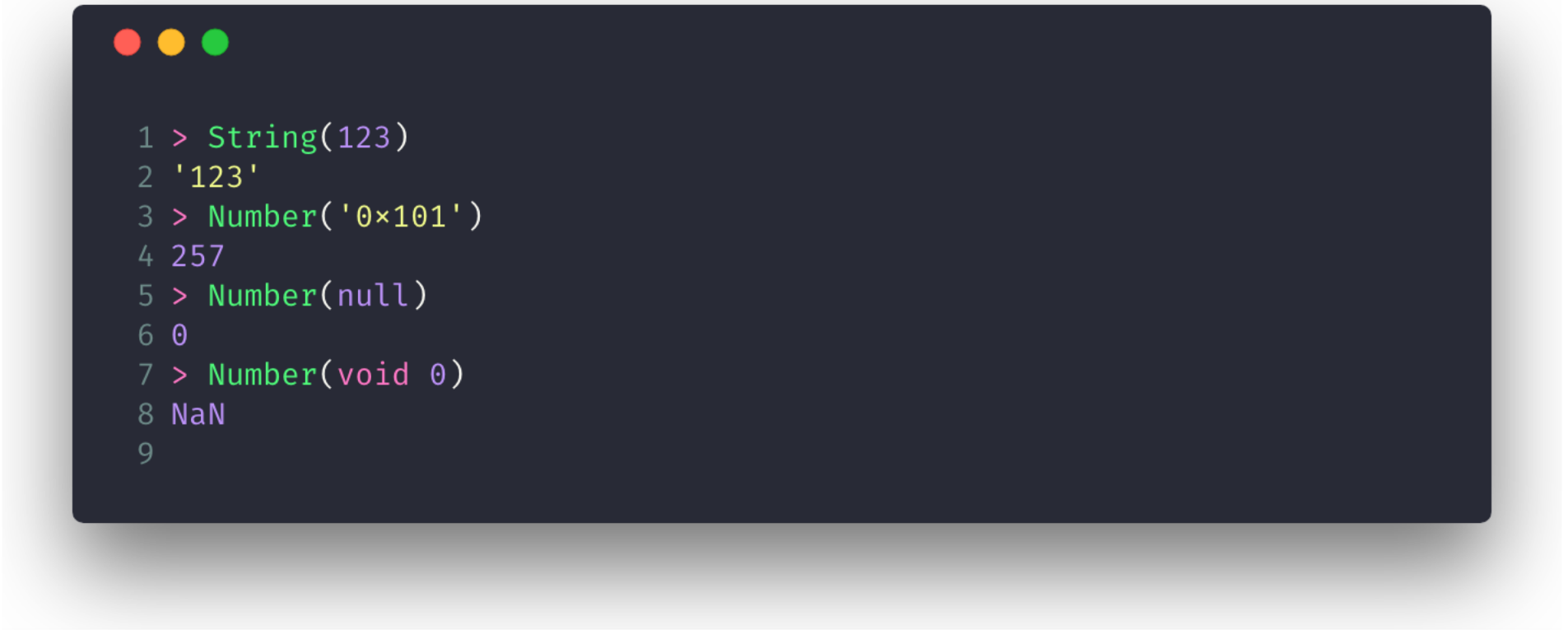
Wrapper Objects for Primitives

- As constructors, they create objects that are largely incompatible with the primitive values that they wrap

```
1 > var str = new String('abc')
2 > typeof str
3 'object'
4 > str === 'abc'
5 false
6
```

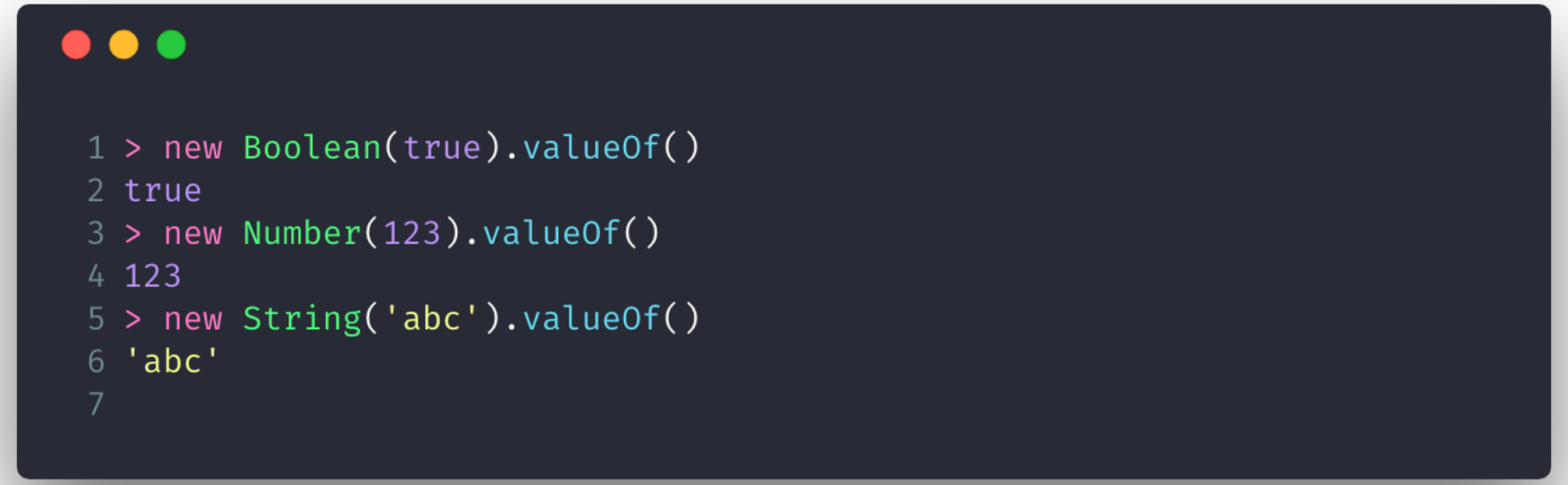
Wrapper Objects for Primitives

- As functions, they convert values to the corresponding primitive types



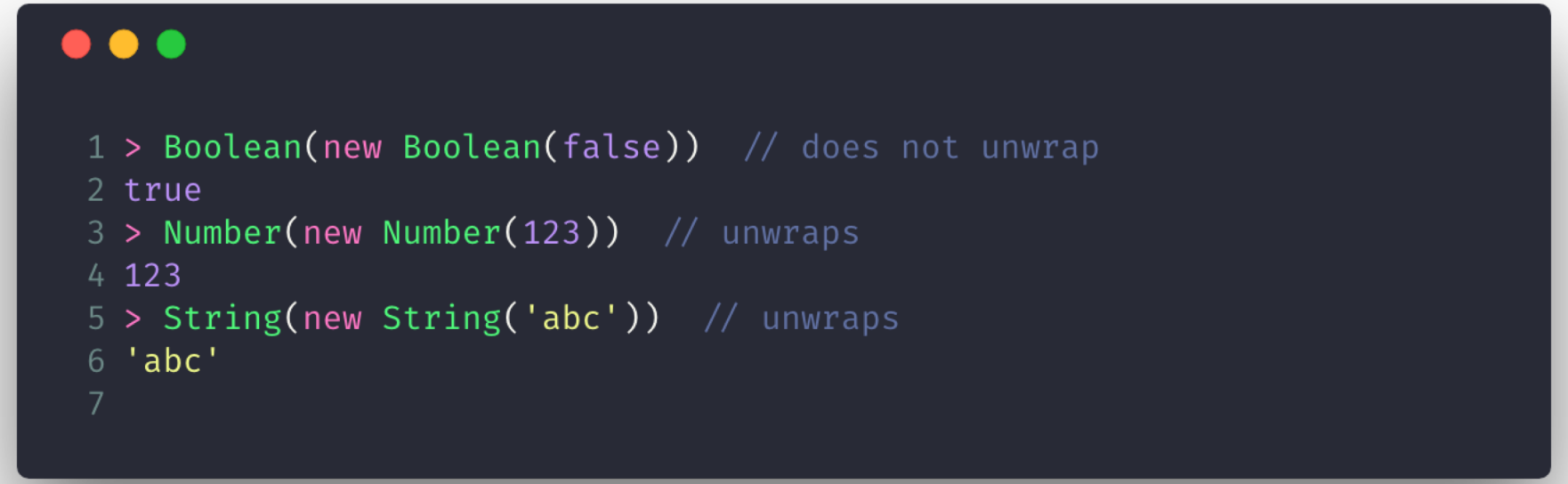
Un-wrapping Primitives

- Unwrap a primitive by invoking the method *valueOf()*. All objects have this method.



Un-wrapping Primitives

- Converting wrapper objects to primitives properly extracts numbers and strings, but not booleans.



```
> Boolean(new Boolean(false))  // does not unwrap
true
> Number(new Number(123))  // unwraps
123
> String(new String('abc'))  // unwraps
'abc'
```

Type Coercion

- *Type coercion* means the implicit conversion of a value of one type to a value of another type.
- Most of JavaScript's operators, functions, and methods coerce operands and arguments to the types that they need.

Truthy and Falsy Values

- undefined, null
- Boolean: false
- Number: 0, NaN
- String: ''
- **all objects are truthy**

Operators

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- String operators
- Conditional (ternary) operator
- Comma operator
- Unary operators
- Relational operators

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators

Operators

```
> console.log(1 == 1);
// expected output: true

> console.log("1" == 1);
// expected output: true

> console.log(1 === 1);
// expected output: true

> console.log("1" === 1);
// expected output: false
```

Operators

- JavaScript Comparison Algorithms:
  - Two strings are strictly equal when they have the same sequence of characters, same length, and same characters in corresponding positions.
  - Two numbers are strictly equal when they are numerically equal (have the same number value). NaN is not equal to anything, including NaN. Positive and negative zeros are equal to one another.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

Operators

- JavaScript Comparison Algorithms:
  - Two Boolean operands are strictly equal if both are true or both are false.
  - Two distinct objects are never equal for either strict or abstract comparisons.
  - An expression comparing Objects is only true if the operands reference the same Object.
  - Null and Undefined Types are strictly equal to themselves and abstractly equal to each other.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

Operators

- JavaScript Comparison Algorithms - **Equality Operators**:
  - When comparing a **number** and a **string**, the **string** is converted to a **number** value.
  - If one of the operands is Boolean, (1 if it is true and +0 if it is false).
  - If an **object** is compared with a **number** or **string**, JavaScript attempts to return the default value for the object. Operators attempt to convert the object to a primitive value, a String or Number value, using the **valueOf** and **toString** methods of the objects. If this attempt to convert the object fails, a runtime error is generated.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

Operators

- JavaScript Comparison Algorithms - **Equality Operators**:
  - If the **valueOf** method returns a primitive value, the **toString** method will never be called.
  - Note that an object is converted into a primitive if, and only if, its comparand is a primitive. If both operands are objects, they're compared as objects, and the equality test is true only if both refer the same object.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

Operators

- JavaScript Comparison Algorithms - **Equality Operators**:

```
> 1 == [1]
true
> 1 == ['1']
true
> 1 == true
true
> 2 == true
false
> 0 == false
true
> var obj = {
    valueOf: function() {
        return true;
    }
}

> 1 == obj
true
> true == obj
true
> 2 == obj
false
```

The Plus Operator

- If one of them is a string, the other is also converted to a string and both are concatenated.
- Otherwise, both operands are converted to numbers and added.

Unary Operators

- **Unary plus (+):** Tries to convert the operand into a number
- **Unary negation (-):** Tries to convert the operand into a number and negates after
- **Logical Not (!):** Converts to boolean value then negates it
- **Increment (++):** Adds one to its operand
- **Decrement (--):** Decrements by one from its operand
- **Bitwise not (~):** Inverts all the bits in the operand and returns a number
- **typeof:** Returns a string which is the type of the operand
- **delete:** Deletes specific index of an array or specific property of an object
- **void:** Discards a return value of an expression.

Unary Operators

```
1 var d = new Date('2018-10-10');
2 d.valueOf()
3 // output: 1539129600000
4 +d
5 // output: 1539129600000
6 -d
7 // output: -1539129600000
8
```

Variables, Scope

- Declaring and Assigning Variables.

Variables, Scope

- An **Identifier** is an **IdentifierName** that is not a **ReservedWord**.
- An identifier must start with $, _, or any character in the Unicode categories "Uppercase letter (Lu)", "Lowercase letter (Ll)", "Titlecase letter (Lt)", "Modifier letter (Lm)", "Other letter (Lo)", or "Letter number (Nl)".
- The rest of the string can contain the same characters, plus any *U+200C zero width non-joiner* characters, *U+200D zero width joiner* characters, and characters in the Unicode categories "Non-spacing mark (Mn)", "Spacing combining mark (Mc)", "Decimal digit number (Nd)", or "Connector punctuation (Pc)".

Variables, Scope

- Valid **Identifiers**.

```
1 var π = Math.PI;
2
3 var λ = function() {
4     return 10;
5 };
6
7
```

https://mathiasbynens.be/notes/javascript-identifiers

Variables, Scope

- Function scope, global scope, hoisting

```
1 var bob = {
2   name: 'Bob',
3   age: 53
4 }
5
6 if (bob.age > 50) {
7     var message = 'OK';
8 }
9
10 console.log(message)
11
```

Variables, Scope

- Function scope, global scope, hoisting

```
function blockScope() {
  'use strict';
  var arr = [];

  for (var i = 0; i < 5; ++i) {
    arr.push(function() {
      console.log(i * i);
    });
  }

  arr[3]();
}

blockScope();

// ouput: 25

```

Variables, Scope

- Function scope, global scope, hoisting

```
function blockScope() {
  'use strict';
  var arr = [];

  for (var i = 0; i < 5; ++i) {
    (function(j) {
        arr.push(function() {
          console.log(j * j);
        });
    })(i);
  }

  arr[3]();
}

// ouput: 9

```

Program Structure

- Conditionals: if-else, switch-case.
- Loops: while, do-while, for, for-in, for-of.

DEMO

NASH TECH
The Power to Innovate

Q&A

Section 2

Function, Higher-Order Function, Object/Class, Array

Functions

- A function is a JavaScript procedure—a set of statements that performs a task or calculates a value.
- To use a function, you must define it somewhere in the scope from which you wish to call it.

Functions

```
1 function sayHi(fname, lname) {
2     return 'Hi: ' + fname + ' ' + lname;
3 }
4
5 console.log(sayHi('Tiep', 'Phan'));
6
```

Functions

- Function declarations: A **function definition** (also called a **function declaration**, or **function statement**) consists of the ***function*** keyword, followed by:
  - The name of the function.
  - A list of parameters to the function, enclosed in parentheses and separated by commas.
  - The JavaScript statements that define the function, enclosed in curly brackets, { }.
- Function declarations are *hoisted* onto the beginning of the scope.

Functions

- Function expressions: A function expression may be a part of a larger expression.
  - One can define "named" function expressions (where the name of the expression might be used in the call stack for example) or "anonymous" function expressions.
- Function expressions are **not *hoisted*** onto the beginning of the scope, therefore they cannot be used before they appear in the code.

Functions

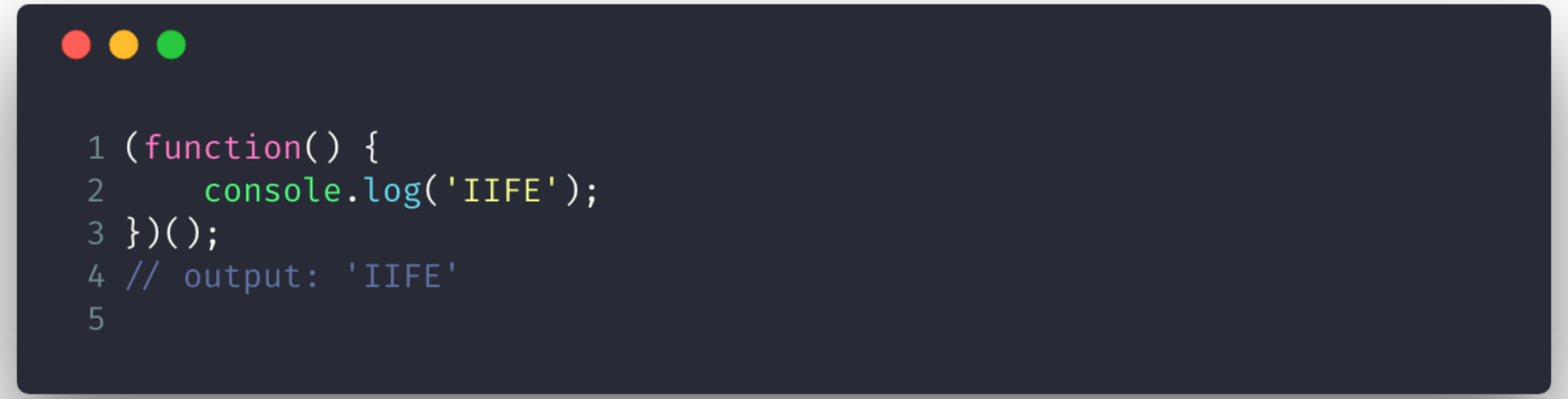
```
1 var sayHi = function (fname, lname) {
2     return 'Hi: ' + fname + ' ' + lname;
3 }
4
5 console.log(sayHi('Tiep', 'Phan'));
6
```

Functions

```
1 console.log(sayHiFn('Tiep', 'Phan Fn'));
2 console.log(sayHi('Tiep', 'Phan'));
3
4 var sayHi = function (fname, lname) {
5     return 'Hi: ' + fname + ' ' + lname;
6 }
7 // vs
8
9 function sayHiFn(fname, lname) {
10     return 'Hi: ' + fname + ' ' + lname;
11 }
12
```

Functions

- IIFE (Immediately Invokable Function Expression): function expressions that are invoked as soon as the function is declared.



```
1 (function() {
2     console.log('IIFE');
3 })();
4 // output: 'IIFE'
5
```

Functions

- The Three Roles of Functions in JavaScript:
  - Non-method function (normal function).
  - Constructor.
  - Method.

```
1 function HttpServer(host, port) {
2   this.host = host;
3   this.port = port;
4 }
5
6 HttpServer.prototype.print = function() {
7   console.log('Server running at ' + this.host + ':' + this.port);
8 }
9 // Constructor
10 var httpServer = new HttpServer('localhost', 80);
11
12 //method
13 httpServer.print();
14
```

Functions

- "Parameter" Versus "Argument":
  - *Parameters* are used to define a function.
  - *Arguments* are used to invoke a function.
- The arguments object: The arguments of a function are maintained in an array-like object.
  - Access 1st argument: `arguments[0]`
  - The total number of arguments is indicated by `arguments.length`

```
1 function boom() {
2   console.log(arguments.length)
3 }
4
5 boom(1, 2, 3);
6 // output: 3
7 boom('some string');
8 // output: 1
9
```

Closure

- You can nest a function within a function. The nested (inner) function is private to its containing (outer) function.
- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.
- When a function gets declared, it contains a function definition and a closure. The closure is a collection of all the variables in scope at the time of creation of the function.

Closure

```
1 function A(x) {
2   function B(y) {
3     function C(z) {
4       console.log(x + y + z);
5     }
6     C(3);
7   }
8   B(2);
9 }
10 A(1); // logs 6 (1 + 2 + 3)
11
```

Closure

```
1 function addHoF(a) {
2   return function(b) {
3     return a + b;
4   }
5 }
6
7 var add2 = addHoF(2);
8 var a6 = add2(4);
9 var a8 = add2(6);
10
```

What is this?

- The “this” keyword allows you to reuse functions with different contexts. Said differently, **the “this” keyword allows you to decide which object should be focal when invoking a function or a method.**

```
var user = {
  name: 'Bob',
  age: 28,
  greet() {
    console.log('Hello, my name is ' + this.name);
  }
}

user.greet();
```

What is this?

```
var john = {
    name: 'John'
};

john.greet = user.greet;

john.greet();
// output: Hello, my name is John

```

call(), apply(), and bind()

```
user.greet.call(john);
// output: Hello, my name is John
john.greet.apply(user);
// output: Hello, my name is Bob
user.greet = function(title) {
  console.log('Hello, ' + title + ': ' + this.name)
}
user.greet.call(john, 'Mr');
user.greet.apply(john, ['Mr']);

```

call(), apply(), and bind()

```
1 var greet = user.greet.bind(user);
2
3 greet('Mr');
4 greet.call(john, 'Mr');
5 greet.apply(john, ['Mr']);
6 // output: Hello, Mr: Bob
7
```

Higher-Order Function

- Functions that operate on other functions, either by taking them as arguments or by returning them, are called *higher-order functions*.

```
1 function addHoF(a) {
2   return function(b) {
3     return a + b;
4   }
5 }
6
7 var add2 = addHoF(2);
8 var a6 = add2(4);
9 var a8 = add2(6);
10
```

Higher-Order Function

```
function iter(arr, fn) {
  for (var i = 0; i < arr.length; i++) {
    fn && fn(arr[i], i);
  }
}
function plusVFn(v) {
  // TODO
}
iter([1, 2, 3], plusVFn(5));
// 6, 7, 8

```

Higher-Order Function

```
function iter(arr, fn) {
  for (var i = 0; i < arr.length; i++) {
    fn && fn(arr[i], i);
  }
}
function plusVFn(v) {
  return function(value, index) {
    console.log(v + value);
  }
}
iter([1, 2, 3], plusVFn(5));
// 6, 7, 8

```

Object/Class

- All objects in JavaScript are maps (dictionaries) from strings to values.
- A (key, value) entry in an object is called a *property*.
- The key of a property is always a text string (number).
- The value of a property can be any JavaScript value, including a function.
- *Methods* are properties whose values are functions.

Object/Class

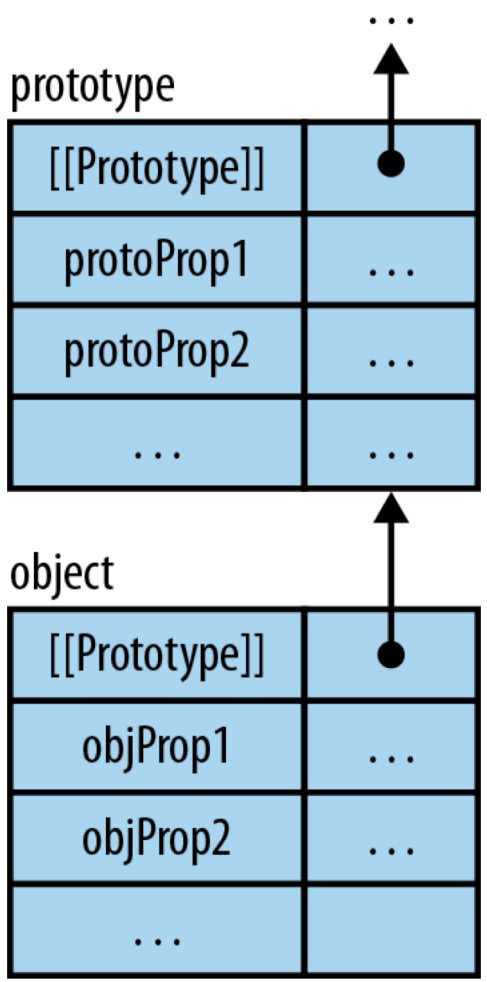
```
1 var user = {
2   name: 'Bob',
3   age: 28,
4   greet() {
5     console.log('Hello, my name is ' + this.name);
6   }
7 }
8
9 user.greet();
10
```

Object/Class

- Pitfall: Losing this When Extracting a Method
- Pitfall: Functions Inside Methods Shadow this

Object/Class

- The Prototype Relationship Between Objects



Object/Class

- The Prototype Relationship Between Objects

```
1 var proto = {
2     describe: function () {
3         return 'name: ' + this.name;
4     }
5 };
6 var obj = {
7     __proto__: proto,
8     name: 'obj'
9 };
10 // output: "name: obj"
11
```

Object/Class

- Constructors—Factories for Instances:

A *constructor function* (short: *constructor*) helps with producing objects that are similar in some way. It is a normal function, but it is named, set up, and invoked differently. This section explains how constructors work. They correspond to classes in other languages.

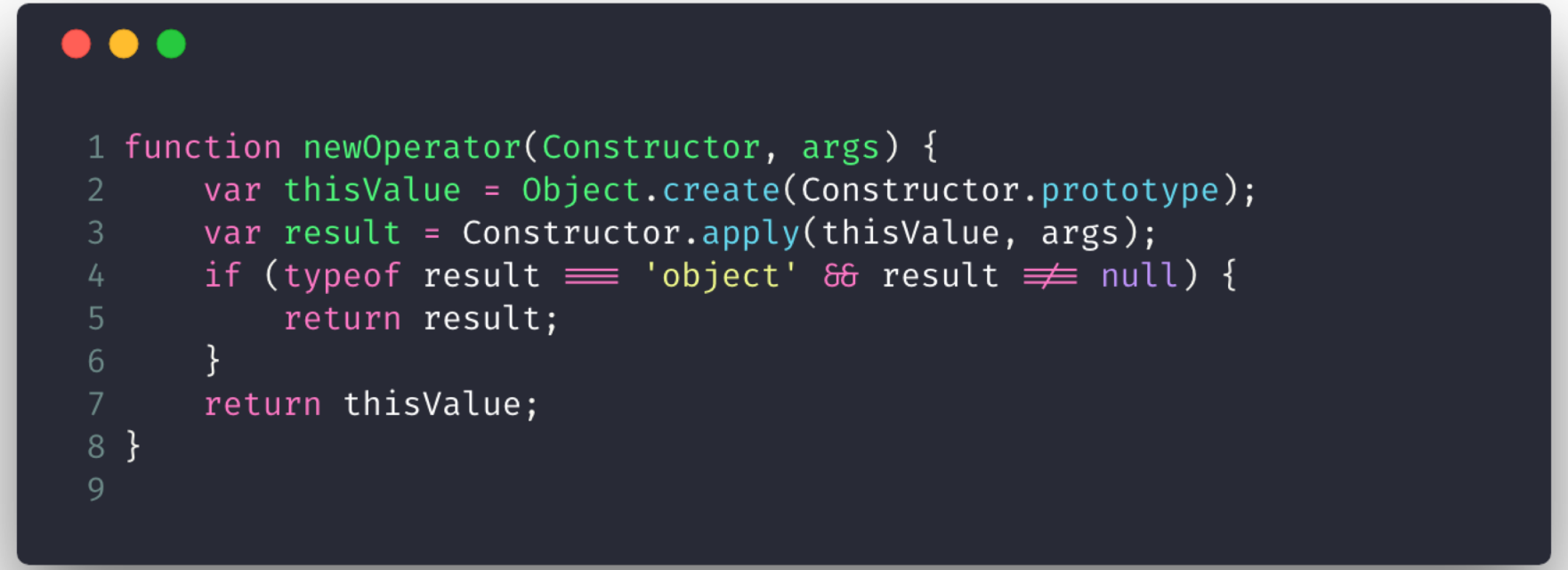
Object/Class

- Constructors:

```
1 function HttpServer(host, port) {
2   this.host = host;
3   this.port = port;
4 }
5
6 HttpServer.prototype.print = function() {
7   console.log('Server running at ' + this.host + ':' + this.port);
8 }
9 // Constructor
10 var httpServer = new HttpServer('localhost', 80);
11
12 //method
13 httpServer.print();
14
```

Object/Class

- The new Operator Implemented in JavaScript:



```
function newOperator(Constructor, args) {
    var thisValue = Object.create(Constructor.prototype);
    var result = Constructor.apply(thisValue, args);
    if (typeof result === 'object' && result !== null) {
        return result;
    }
    return thisValue;
}

```

Object/Class

- Inheritance in JavaScript:

```
1 function Shape(x, y) {
2     this.x = x;
3     this.y = y;
4 }
5 Shape.prototype.moveTo = function (x, y) {
6     console.log('From: (' + this.x + ',' + this.y + ')');
7     console.log('To: (' + x + ',' + y + ')');
8     this.x = x;
9     this.y = y;
10 };
11 function Rect(x, y, w, h) {
12     Shape.call(this, x, y);
13     this.w = w;
14     this.h = h;
15 }
16 Rect.prototype = Object.create(Shape.prototype);
17 Rect.prototype.constructor = Rect;
18 var r = new Rect(0,0, 100, 50);
19 r.moveTo(5, 10);
20 /*output:
21 From: (0,0)
22 To: (5,10)
23 */
24
```

Array

```
1 var pets = ['dog', 'cat'];
2 pets.length
3 // 2
4 pets.push('owl');
5 // ['dog', 'cat', 'owl']
6 var rank = [3, 4, 1, 4, 30, 0, 5];
7 rank.filter(function(point) {
8     return point ⩾ 5;
9 });
10 // [30, 5]
11 rank.map(function(point) {
12     return point * 1.2;
13 });
14 // [3.6, 4.8, 1.2, 4.8, 36, 0, 6]
15
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

DEMO

NASH TECH
The Power to Innovate

Q&A

Section 3

Asynchronous Programming, Events, Block scope variables, Constant

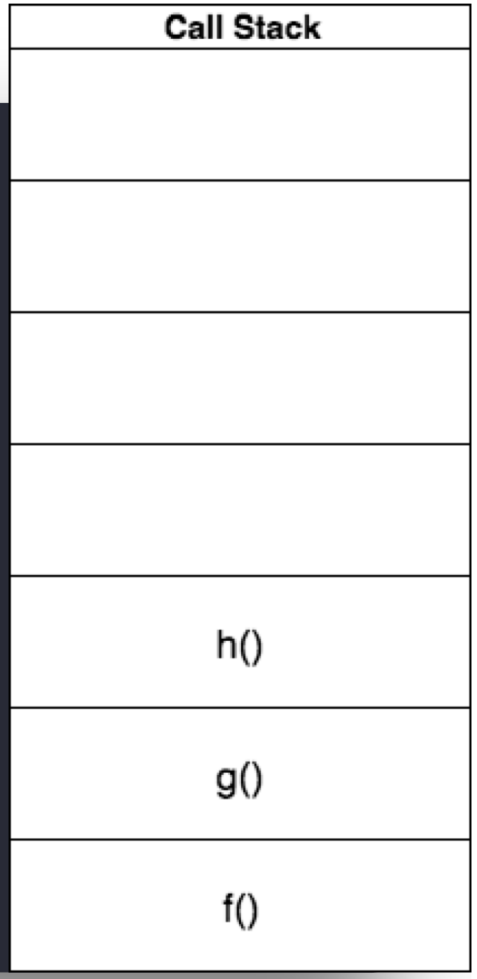
Asynchronous Programming

- Both JavaScript in browser, and in Node.js, are single-threaded.
- In *synchronous* programs, if you have two lines of code (L1 followed by L2), then L2 cannot begin running until L1 has finished executing.
- In *asynchronous* programs, you can have two lines of code (L1 followed by L2), where L1 schedules some task to be run in the future, but L2 runs before that task completes.

Asynchronous Programming

- Call Stack

```
1 function h(z) {
2   console.log(z);
3 }
4
5 function g(y) {
6   h(y + 2);
7 }
8
9 function f(x) {
10  g(x + 1);
11 }
12
13 f(5);
14
```



Asynchronous Programming

- Web Browser API

```
1 console.log('START ... ');
2 setTimeout(function() {
3   console.log('TIMER ... ');
4 }, 1000);
5 console.log('END ... ');
6
```

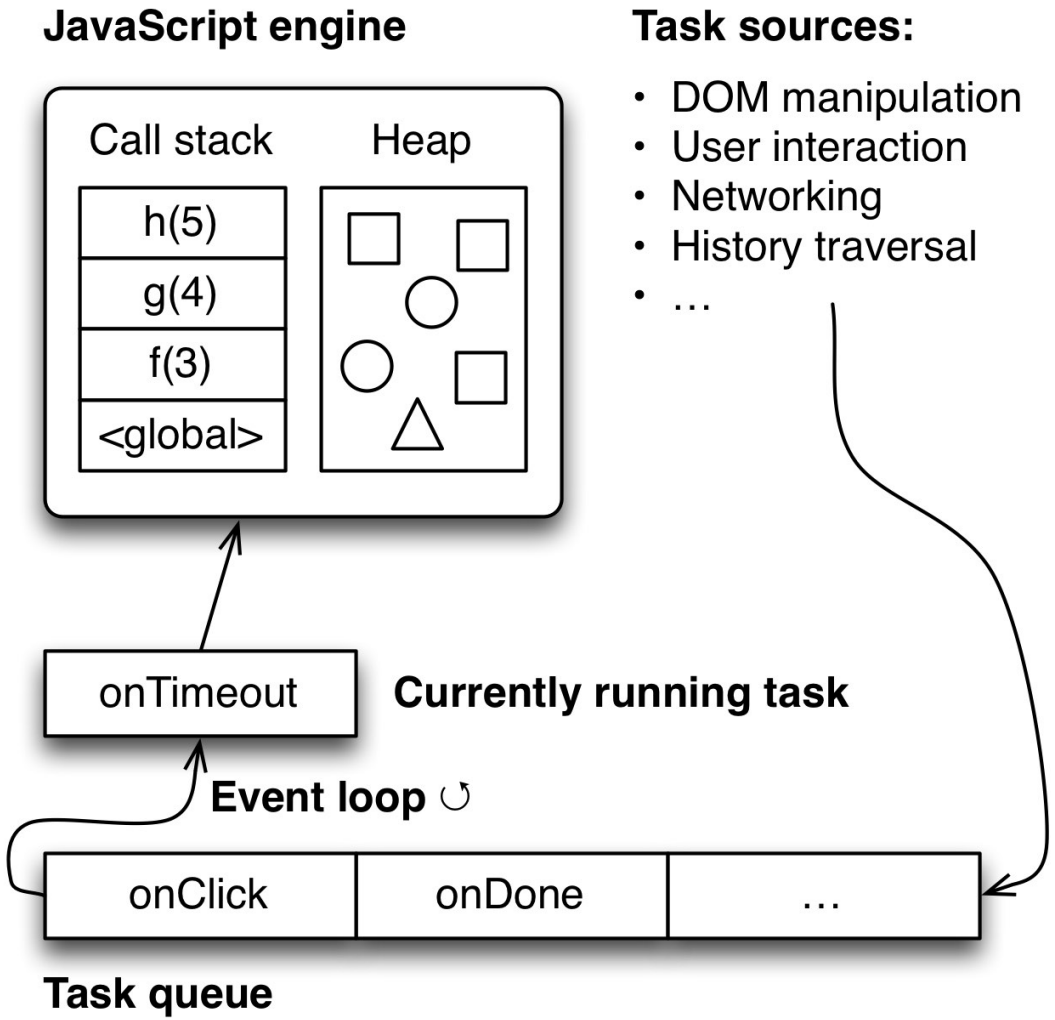
Asynchronous Programming

- Web Browser API

```
1 console.log('START ... ');
2 setTimeout(function() {
3   console.log('TIMER ... ');
4 }, 0);
5 console.log('END ... ');
6
```

Asynchronous Programming

- Event Loop



https://www.youtube.com/watch?v=8aGhZQkoFbQ

Asynchronous Programming

- Event Loop: The reality is that *all* JavaScript executes synchronously - it's the event loop that allows you to queue up an action that won't take place until the loop is available some time *after* the code that queued the action has finished executing.

Asynchronous Programming

- Callback:
  - Pros: Easy to understand (only for simple callback function).
  - Cons: Error handling becomes more complicated, Composition is more complicated (Callback hell)

Asynchronous Programming

- Callback:

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

Asynchronous Programming

- Callback:

```
try {
  setTimeout(function() {
    throw new Error("Woosh");
  }, 1000);
} catch (_) {
  // This will not run
  console.log("Caught!");
}

```

Asynchronous Programming

- AJAX

```
1 function fetchData(URL, done) {
2   var http = new XMLHttpRequest();
3   http.onreadystatechange = function() {
4     if (this.readyState == 4 && this.status == 200) {
5         done(http);
6     }
7   };
8   http.open("GET", URL, true);
9   http.send();
10 }
11 var API_URL = 'https://api.github.com/search/repositories?q=angular';
12 fetchData(API_URL, function(xhr) {
13     console.log(xhr)
14 });
15
```

Asynchronous Programming

- Asynchronous programming makes it possible to express waiting for long-running actions without freezing the program during these actions.
- JavaScript environments typically implement this style of programming using callbacks, functions that are called when the actions complete.
- An event loop schedules such callbacks to be called when appropriate, one after the other, so that their execution does not overlap.

Event

- HTML events are **"things"** that happen to HTML elements.

```
/**HTML
<button class="addToCart">Add to Cart</button>
<div class="output"></div>
*/
var btnAddToCart = document.querySelector('.addToCart');
var output = document.querySelector('.output');

btnAddToCart.addEventListener('click', function() {
  output.textContent = 'Added to Cart';
  setTimeout(function() {
    output.textContent = '';
  }, 3000);
});

```

https://jsbin.com/cibuyereme/edit?html,js,output

Event

- Event objects

```
/**HTML
<input type="text" data-maxlength="6" class="handleMaxLength">
*/
var handleMaxLength = document.querySelector('.handleMaxLength');
handleMaxLength.addEventListener('input', function(event) {
  var input = event.target;
  var maxLength = input.dataset['maxlength'] >> 0;
  if (input.value.length > maxLength) {
    input.value = input.value.substr(0, maxLength);
  }
});

```

https://jsbin.com/sugoviziyi/edit?html,js,output

Event

- Remove Listener

```
1 function handleAddToCart() {
2   output.textContent = 'Added to Cart';
3   setTimeout(function() {
4     output.textContent = '';
5   }, 3000);
6 }
7
8 btnAddToCart.addEventListener('click', handleAddToCart);
9 setTimeout(function() {
10   btnAddToCart.removeEventListener('click', handleAddToCart);
11 }, 5000);
12
```

https://jsbin.com/jodiqafulu/edit?html,js,output

Block Scope Variables

- let

```
1 var x = 1;
2
3 if (x === 1) {
4   var x = 2;
5
6   console.log(x);
7 }
8
9 console.log(x);
10
```

Block Scope Variables

- let

```
let x = 1;

if (x === 1) {
  let x = 2;

  console.log(x);
}

console.log(x);

```

Block Scope Variables

- let

```
1  function blockScope() {
2    'use strict';
3    var arr = [];
4
5    for (let i = 0; i < 5; ++i) {
6      arr.push(function() {
7        console.log(i * i);
8      });
9    }
10
11   arr[3]();
12 }
13
14 blockScope();
15
```

https://www.tiepphan.com/es2015-var-let-const-keywords/

Block Scope Variables

- const

```
1 function constant() {
2   'use strict';
3   const PI = 3.14;
4   PI = 3.5;
5 }
6
7 constant();
8 // TypeError: Assignment to constant variable
9
```

https://www.tiepphan.com/es2015-var-let-const-keywords/

Block Scope Variables

- const

```
1 function constant() {
2   'use strict';
3   const obj = {'name': 'constant'};
4
5   obj.type = 'Something go wrong!';
6   console.log(obj);
7 }
8
9 constant();
10
```

https://www.tiepphan.com/es2015-var-let-const-keywords/

Block Scope Variables

- const

```
1 function constant() {
2   'use strict';
3   const arr = [];
4   arr.push(1, 2, 3);
5
6   console.log(arr);
7
8   arr.splice(1, 1);
9   console.log(arr);
10  arr.push(4, 7, 9);
11  console.log(arr);
12  arr.shift();
13  console.log(arr);
14 }
15
16 constant();
17
```

https://www.tiepphan.com/es2015-var-let-const-keywords/

DEMO

NASH TECH
The Power to Innovate

Q&A

THANK YOU

www.nashtechglobal.com