JavaScript

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

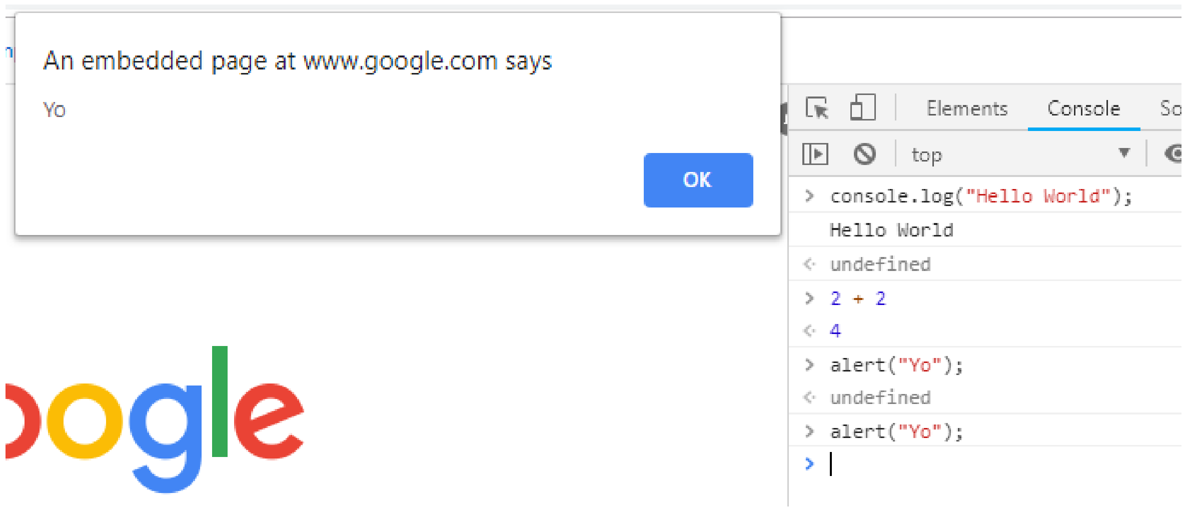
Authorized & published by Summitworks Technologies Inc

Agenda

- Introduction
- Data Types
- Variables
- Console.log
- Alerts, Prompts, Confirms
- Document.write
- Operators
- Conditional statements
- Functions
- Parameters
- Function Arguments
- Variable Scope
- Comments
- Objects
- Creating Objects
- Accessing Properties
- Setting Properties
- Methods
- Executing Methods
- this keyword
- JavaScript HTML DOM Elements
- JavaScript HTML DOM EventListener
- Event Bubbling or Event Capturing?
- The removeEventListener() method
- JavaScript Timing Events
- The setTimeout() Method
- The setInterval() Method
- ECMAScript

Introduction

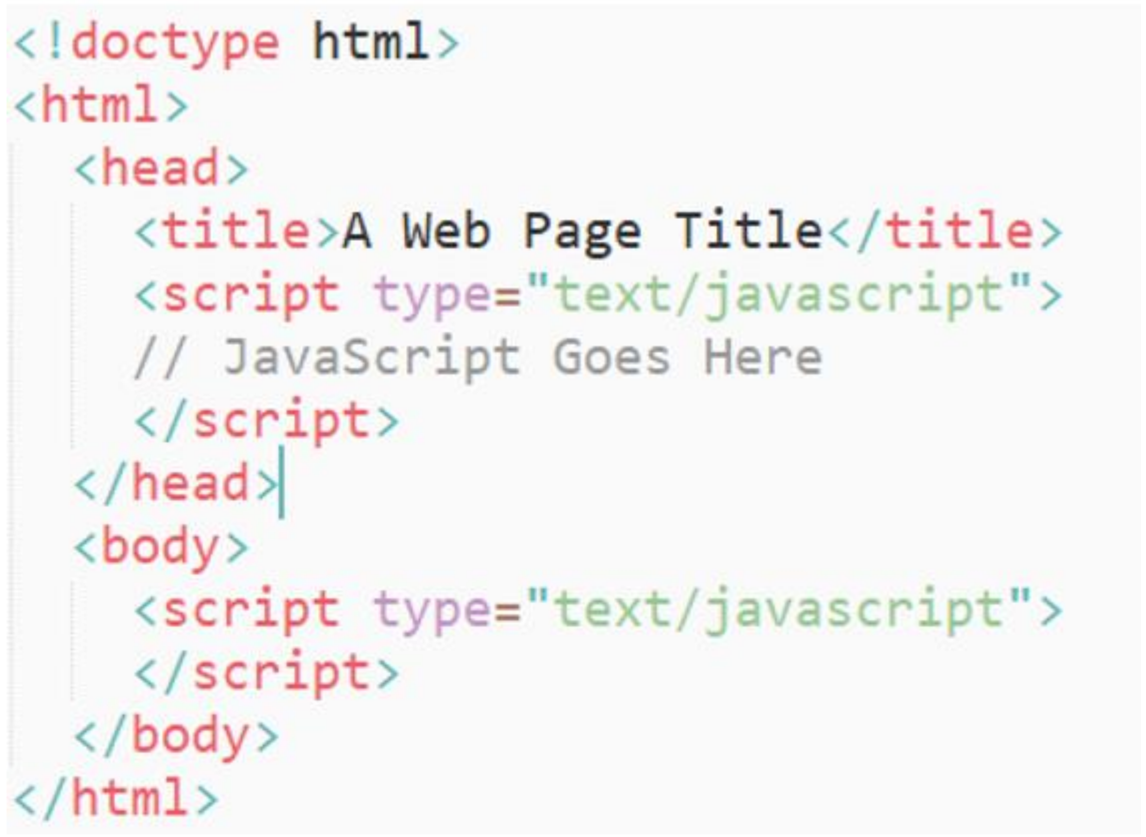
- JavaScript is the third of the three fundamental programming languages of the modern web (along with HTML, CSS).
- JavaScript allows developers to create dynamic web applications capable of taking in user inputs, changing what's displayed to users, animating elements, and much more.
- All browser has JavaScript engines (Firefox engine is Spider monkey and Chrome is v8) and previously JavaScript runs only in browser.
- Later in 2009, Ryan Dahl developed Node JS using C++ that includes Google's JavaScript V8 engine. Now we can run JavaScript outside browser.
- As every browser has JavaScript engine, we can easily run JavaScript code in browser without any additional tool.


An embedded page at www.google.com says

Yo

```
> console.log("Hello World");
  Hello World
< undefined
> 2 + 2
< 4
> alert("Yo");
< undefined
> alert("Yo");
> |
```

JavaScript placement on your webpage



```html
<!doctype html>
<html>
  <head>
    <title>A Web Page Title</title>
    <script type="text/javascript">
    // JavaScript Goes Here
    </script>
  </head>
  <body>
    <script type="text/javascript">
    </script>
  </body>
</html>
```

Data types

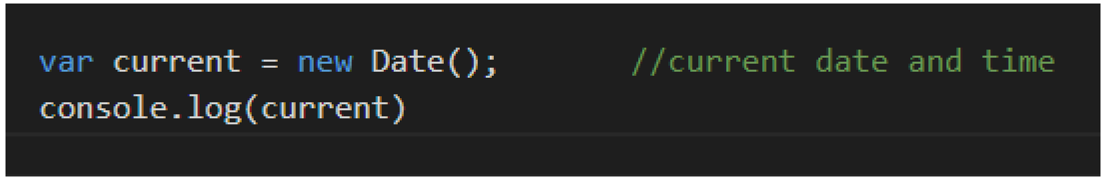
- A datatype is a specialized form of information (number, string. etc.)
- JavaScript supports the following data types:
  - String: represents a list of characters. (ex: “abc”, ‘Ind*3j2n’)
  - Number: represents an integer or decimal value. (ex: 100, 12.0383)
  - Boolean: represents either true or false. (ex: true or false)
  - Object: are special data types which has properties that are identified as key/value pairs.

```
ex: Cat = {
          name: ‘Mittens’,
          age: 2,
        }
```

  - Undefined: represents a value not yet defined
  - Null: represents an empty value
  - Lets look into object in detail in coming slides.

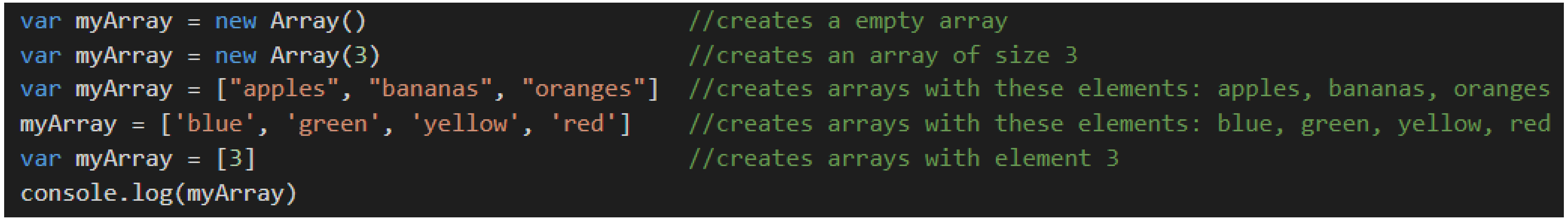
Data types cont….

- Few built-in object based data types are:
  - Date: represents date/time
    Date.now() //represents current time



```
var current = new Date();        //current date and time
console.log(current)
```

  - Array: represents a collection of data



```
var myArray = new Array()                       //creates a empty array
var myArray = new Array(3)                      //creates an array of size 3
var myArray = ["apples", "bananas", "oranges"]  //creates arrays with these elements: apples, bananas, oranges
myArray = ['blue', 'green', 'yellow', 'red']    //creates arrays with these elements: blue, green, yellow, red
var myArray = [3]                               //creates arrays with element 3
console.log(myArray)
```

Variables

- JavaScript variables are containers for storing data values.

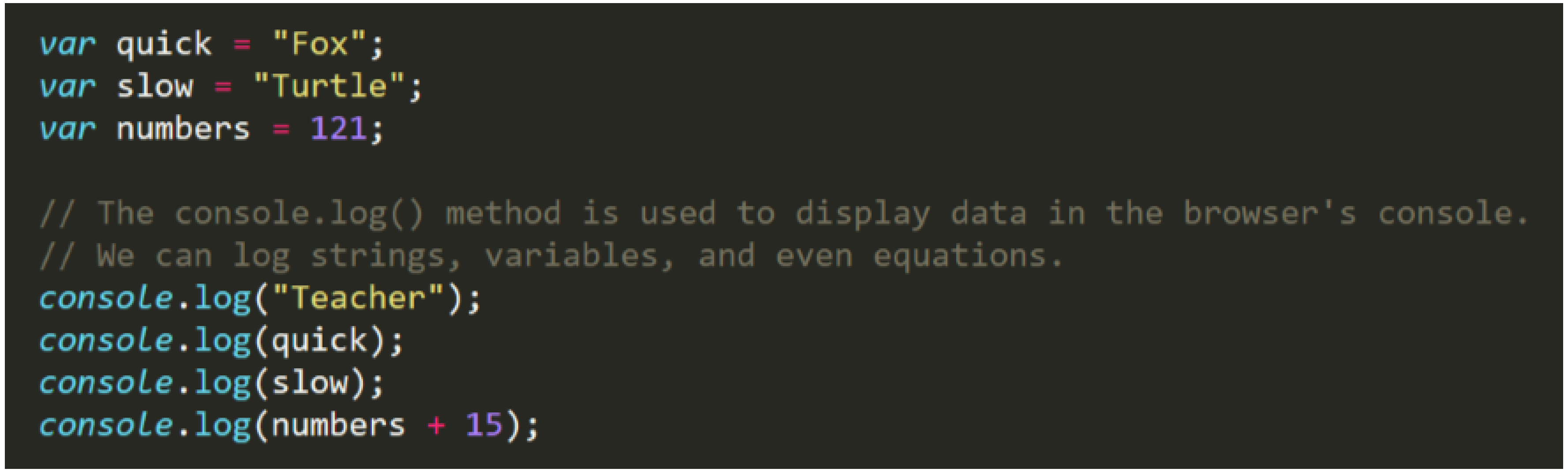
```
var x = 5;
var y = 6;
var z = x + y;
```

  In this example, x, y and z are variables
- JavaScript variables can hold numbers like 100 and text values like "John Doe".
- Variables are case-sensitive: “name” is not equal to “Name”.

Var Keyword	Variable name	Assignment	Value	Termination
var	**name**	**=**	**“Nicholas”**	**;**

Console.log

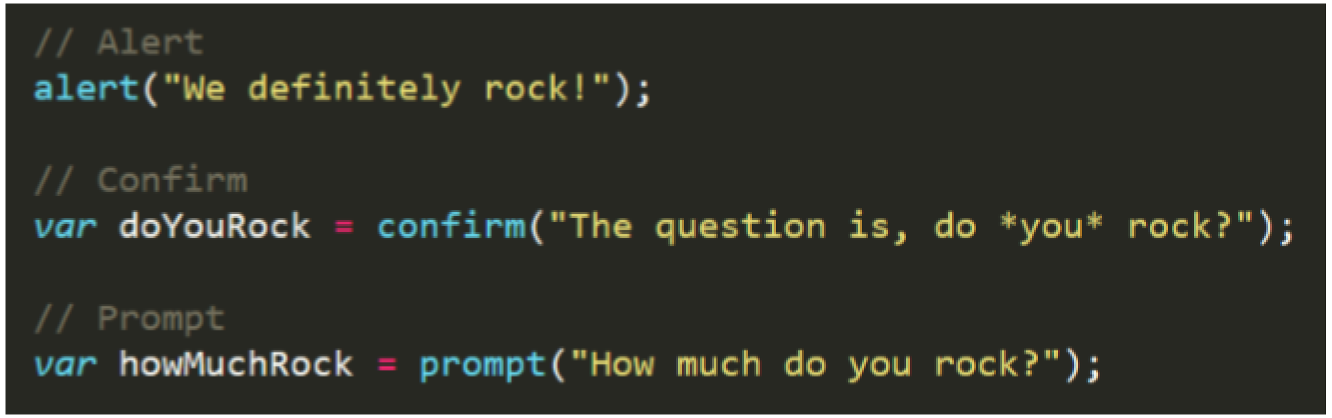
- console.log is a quick expression used to print content to the debugger.
- It is a very useful tool to use during development and debugging.



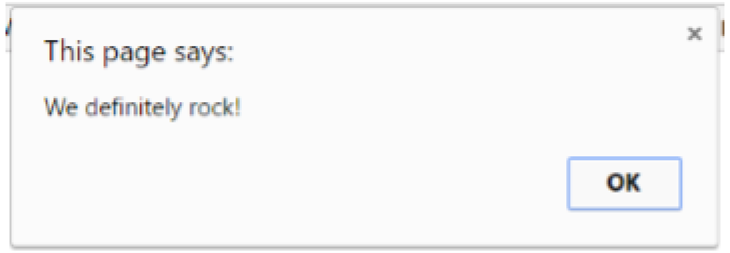
```
var quick = "Fox";
var slow = "Turtle";
var numbers = 121;

// The console.log() method is used to display data in the browser's console.
// We can log strings, variables, and even equations.
console.log("Teacher");
console.log(quick);
console.log(slow);
console.log(numbers + 15);
```

Alerts, Prompts, Confirms

- alerts, confirms, and prompts will create a popup box in the browser when run.
- These are also useful for development and debugging.

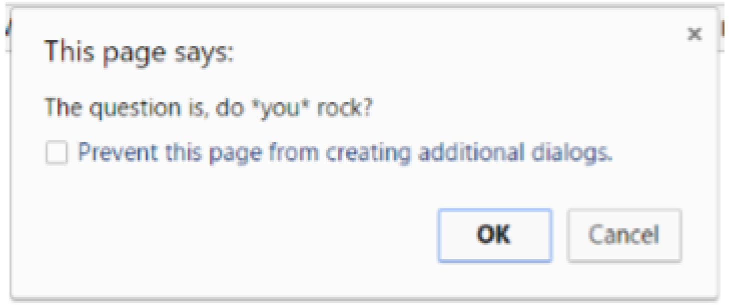


```
// Alert
alert("We definitely rock!");

// Confirm
var doYouRock = confirm("The question is, do *you* rock?");

// Prompt
var howMuchRock = prompt("How much do you rock?");
```



This page says:

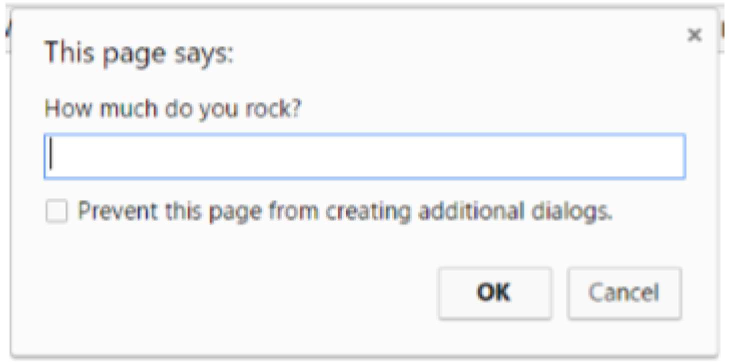
We definitely rock!



This page says:

The question is, do *you* rock?

☐ Prevent this page from creating additional dialogs.



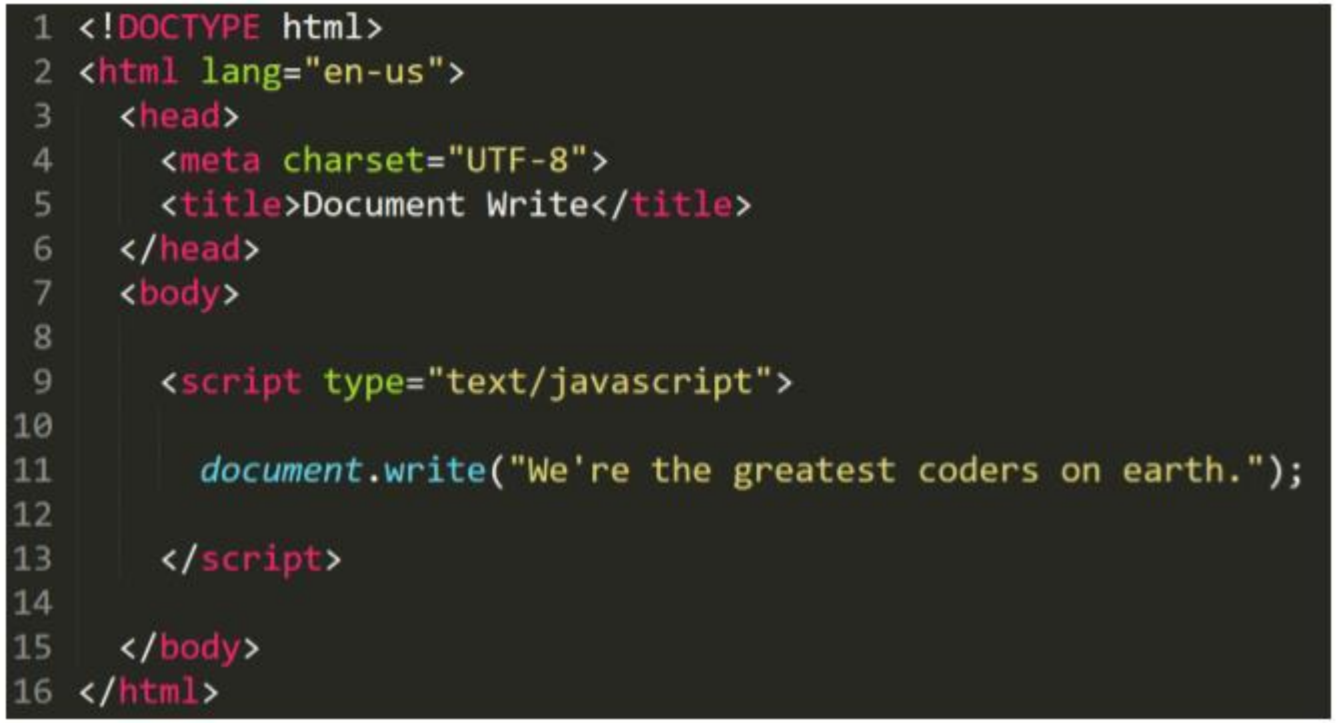
This page says:

How much do you rock?

☐ Prevent this page from creating additional dialogs.

Document Write: Writing to HTML

- document.write( ) is used in JavaScript to directly write to the HTML page



```html
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <meta charset="UTF-8">
    <title>Document Write</title>
  </head>
  <body>

    <script type="text/javascript">

      document.write("We're the greatest coders on earth.");

    </script>

  </body>
</html>
```

Operators

- An Operator is a symbol used to perform a calculation or comparison.

• Addition (+)	a + b
• Subtraction (-)	a – b
• Multiplication (*)	a * b
• Division (/)	a / b
• Modulus (%)	a % b
• Increment (++)	a ++
• Decrement (--)	a—

- You can combine or concatenate 2 strings by using the plus operator (+)

  ex: var a = “Lazy” + “alligator”;

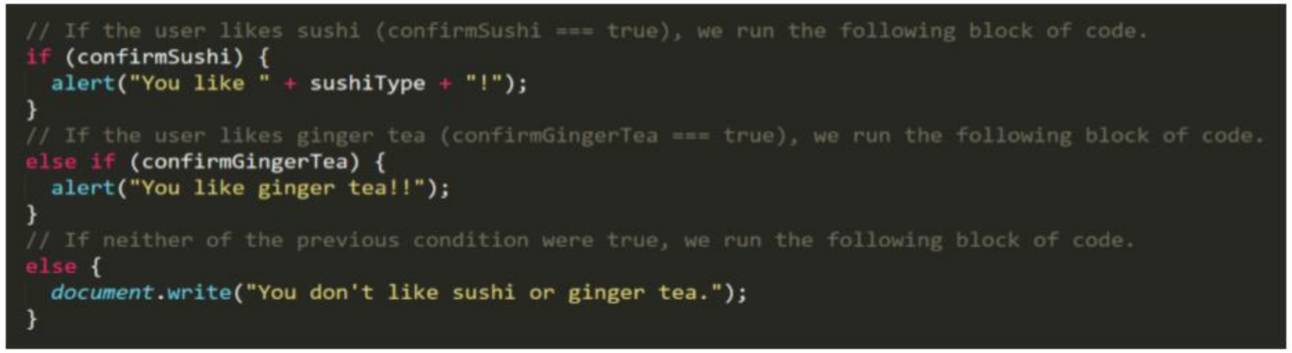
  a is now “Lazy alligator”

Conditional Statements

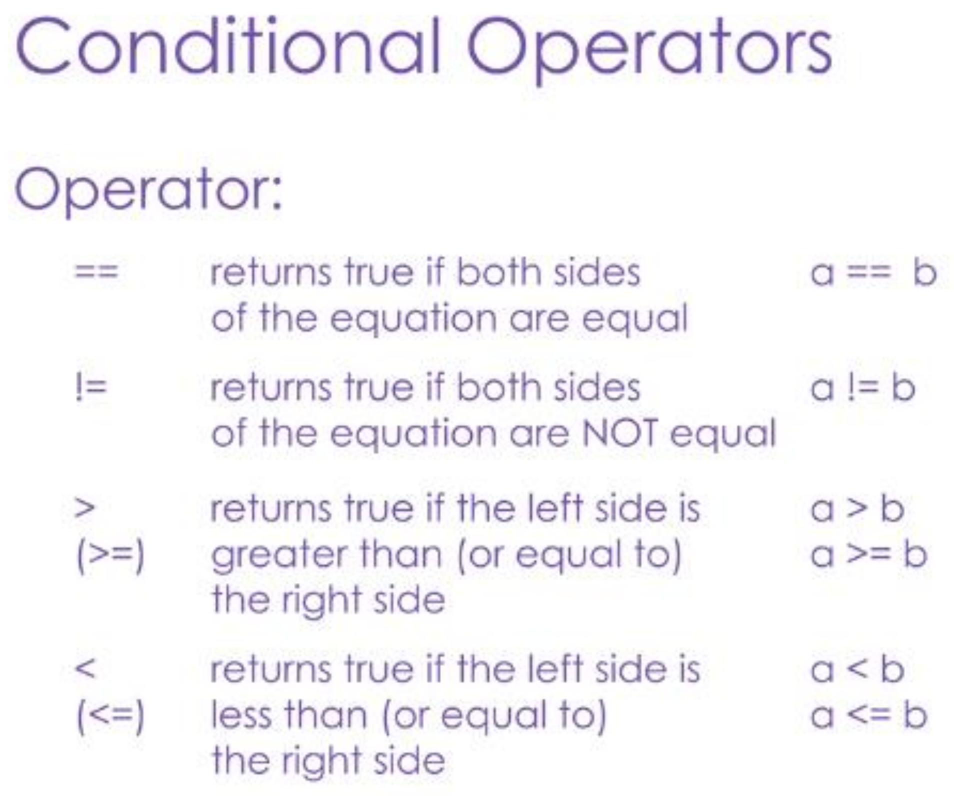
- Very often when you write code, you want to perform different actions for different decisions.
- You can use conditional statements in your code to do this.
- In JavaScript we have the following conditional statements:
  - Use if to specify a block of code to be executed, if a specified condition is true
  - Use else to specify a block of code to be executed, if the same condition is false
  - Use else if to specify a new condition to test, if the first condition is false
  - Use switch to specify many alternative blocks of code to be executed

If/Else Statements

- The if statement executes a statement if a specified condition is truthy. If the condition is falsy, another statement can be executed.



```
// If the user likes sushi (confirmSushi === true), we run the following block of code.
if (confirmSushi) {
  alert("You like " + sushiType + "!");
}
// If the user likes ginger tea (confirmGingerTea === true), we run the following block of code.
else if (confirmGingerTea) {
  alert("You like ginger tea!!");
}
// If neither of the previous condition were true, we run the following block of code.
else {
  document.write("You don't like sushi or ginger tea.");
}
```



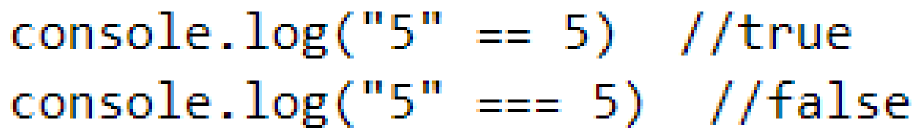
Conditional Operators

Operator:

==	returns true if both sides of the equation are equal	a == b
!=	returns true if both sides of the equation are NOT equal	a != b
> (>=)	returns true if the left side is greater than (or equal to) the right side	a > b a >= b
< (<=)	returns true if the left side is less than (or equal to) the right side	a < b a <= b

Comparing different Types

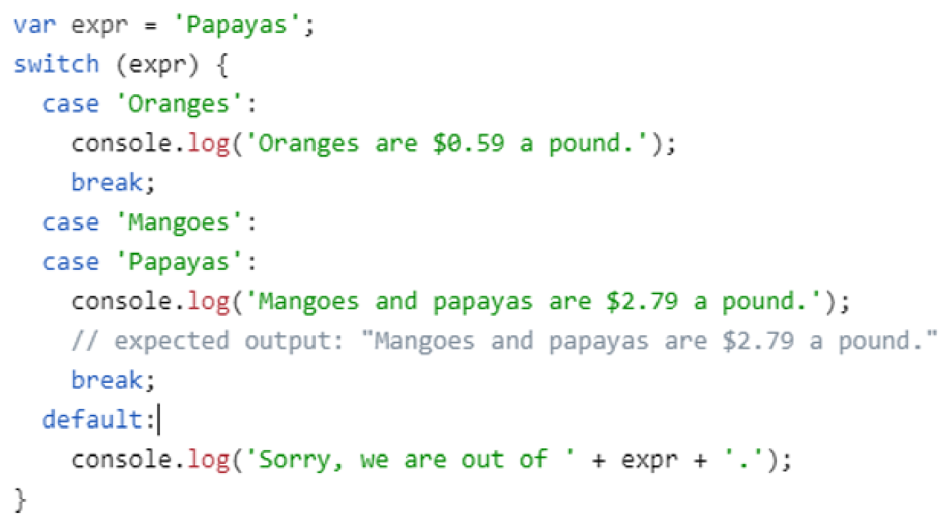
- When using triple equals (===) in JavaScript, we are testing for strict equality. This means both the type and the value we are comparing have to be the same.
- When using double equals (==) in JavaScript we are testing for loose equality. This means only value has to be same.



```
console.log("5" == 5)  //true
console.log("5" === 5)  //false
```

Switch Statements

- The switch statement evaluates an expression, matching the expression's value to a case clause, and executes statements associated with that case, as well as statements in cases that follow the matching case.
- **Expression:** An expression whose result is matched against each case clause.
- **Case valueN Optional:** A case clause used to match against expression. If the expression matches the specified valueN, the statements inside the case clause are executed until either the end of the switch statement or a break.
- **Break**: Basically is used to execute the statements of a single case statement. If no break appears, the flow of control will fall through all the subsequent cases until a break is reached or the closing curly brace '}' is reached.
- **Default Optional:** A default clause; if provided, this clause is executed if the value of expression doesn't match any of the case clauses.



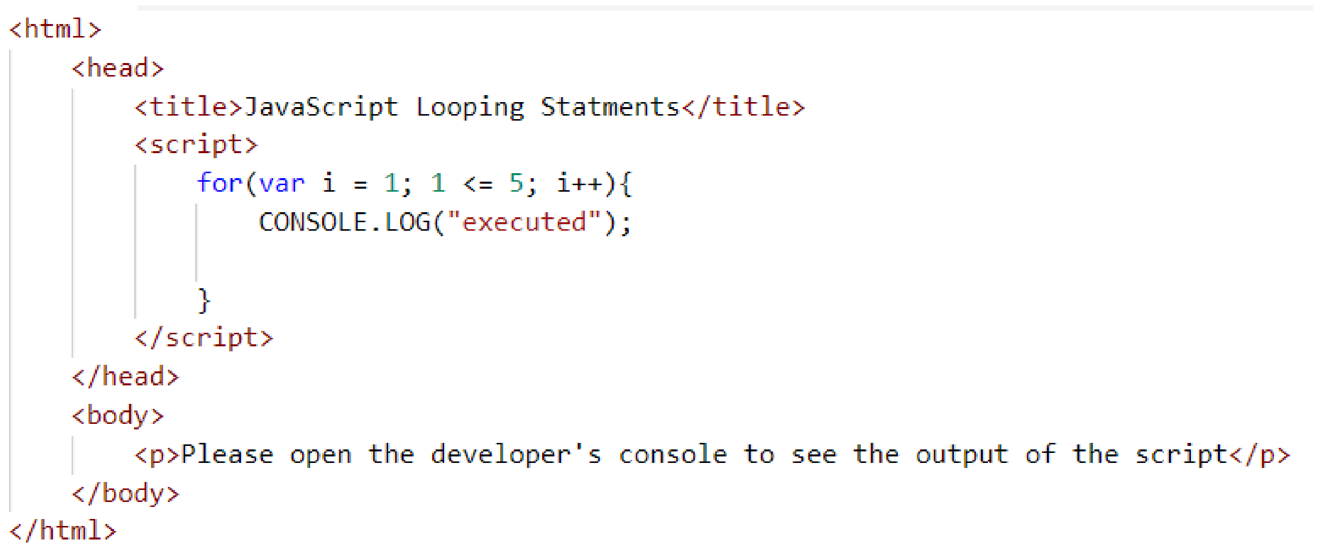
```
var expr = 'Papayas';
switch (expr) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
    break;
  case 'Mangoes':
  case 'Papayas':
    console.log('Mangoes and papayas are $2.79 a pound.');
    // expected output: "Mangoes and papayas are $2.79 a pound."
    break;
  default:
    console.log('Sorry, we are out of ' + expr + '.');
}
```

Loops

- There are several ways to execute a statement or block of statements repeatedly. In general, repetitive execution is called looping. It is typically controlled by a test of some variable, the value of which is changed each time the loop is executed. JavaScript supports many types of loops: for loops, while loops, do...while loops.

For Loop

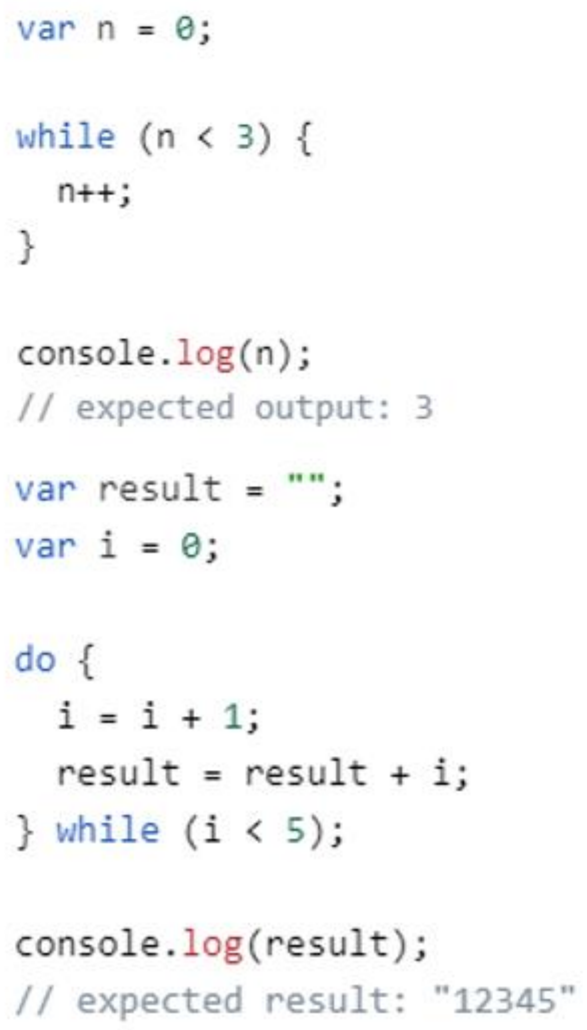
- **For Loop**: The **for statement** creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement to be executed in the loop.
- for ([initialization]; [condition]; [final-expression]) statement



```
<html>
    <head>
        <title>JavaScript Looping Statments</title>
        <script>
            for(var i = 1; 1 <= 5; i++){
                CONSOLE.LOG("executed");

            }
        </script>
    </head>
    <body>
        <p>Please open the developer's console to see the output of the script</p>
    </body>
</html>
```

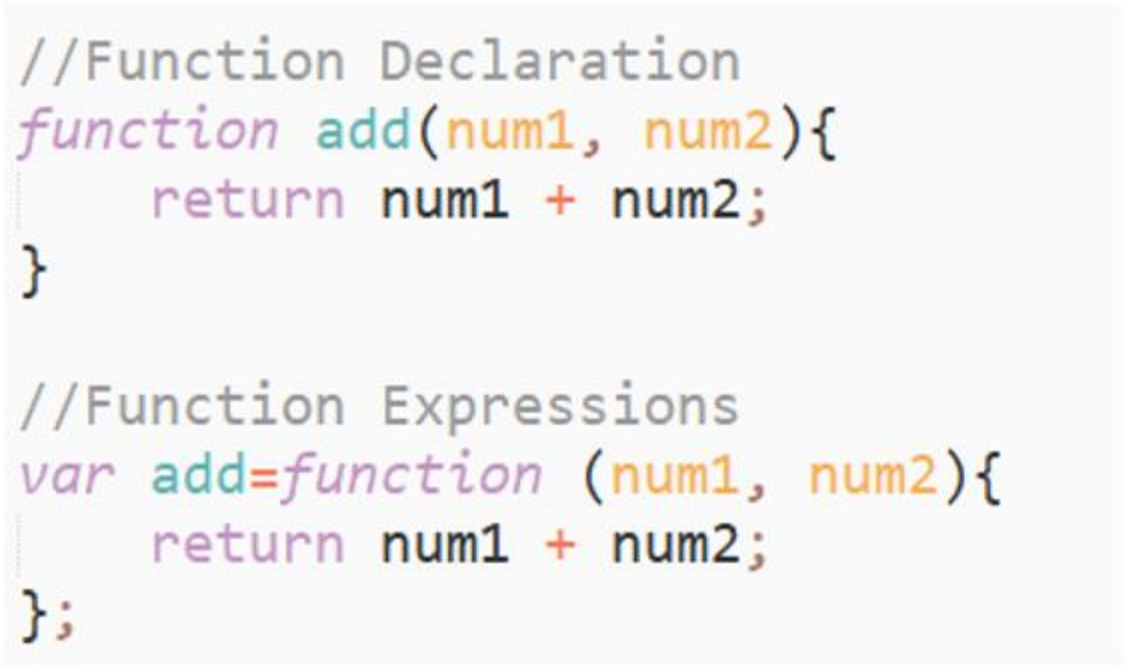
While Loop

- **While Loop**: The while statement creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.
- **Do While Loop**: The do...while statement creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

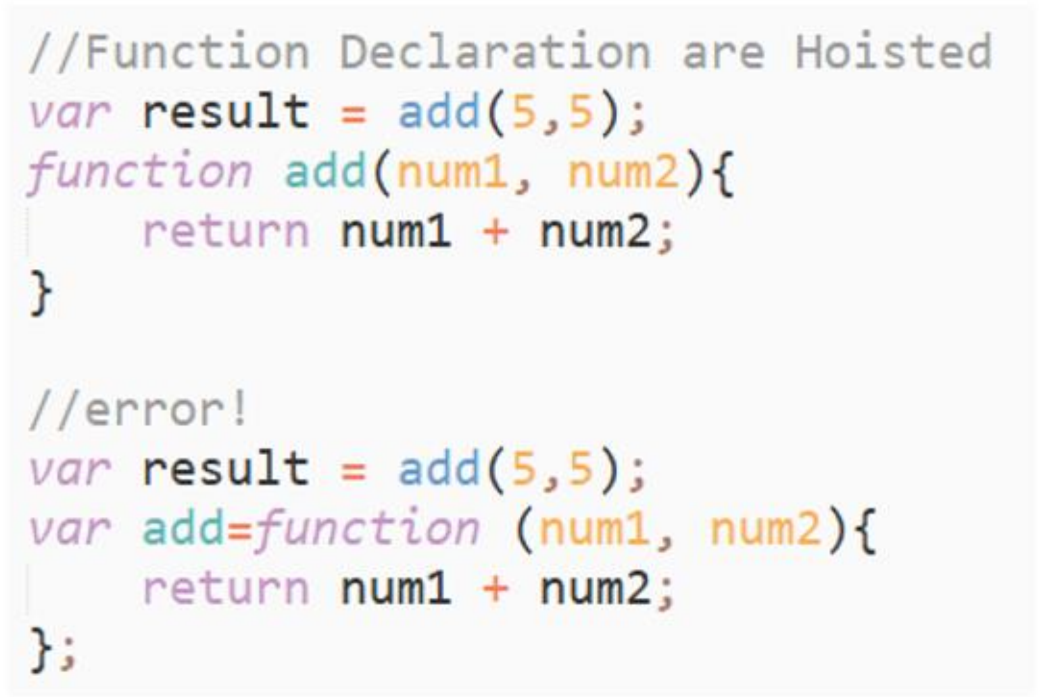


```
var n = 0;

while (n < 3) {
  n++;
}

console.log(n);
// expected output: 3

var result = "";
var i = 0;

do {
  i = i + 1;
  result = result + i;
} while (i < 5);

console.log(result);
// expected result: "12345"
```

Function Declaration vs. Expressions



```
//Function Declaration
function add(num1, num2){
    return num1 + num2;
}

//Function Expressions
var add=function (num1, num2){
    return num1 + num2;
};
```

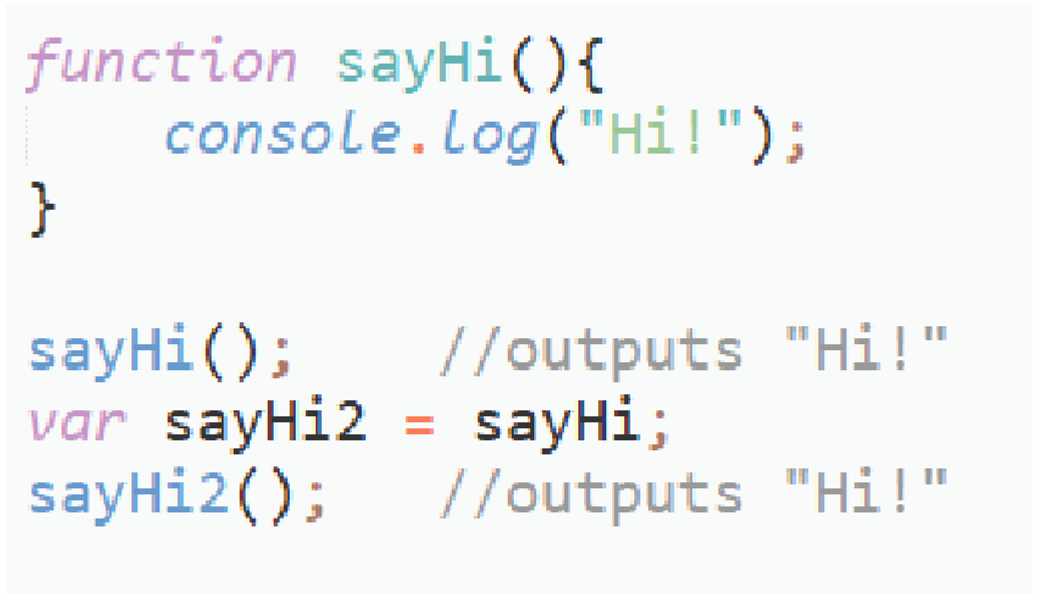


```
//Function Declaration are Hoisted
var result = add(5,5);
function add(num1, num2){
    return num1 + num2;
}

//error!
var result = add(5,5);
var add=function (num1, num2){
    return num1 + num2;
};
```

- Function Declaration are hoisted to the top of the context when the code is executed. That means you can actually define a function after it is used in code without generating an error

Functions as Values

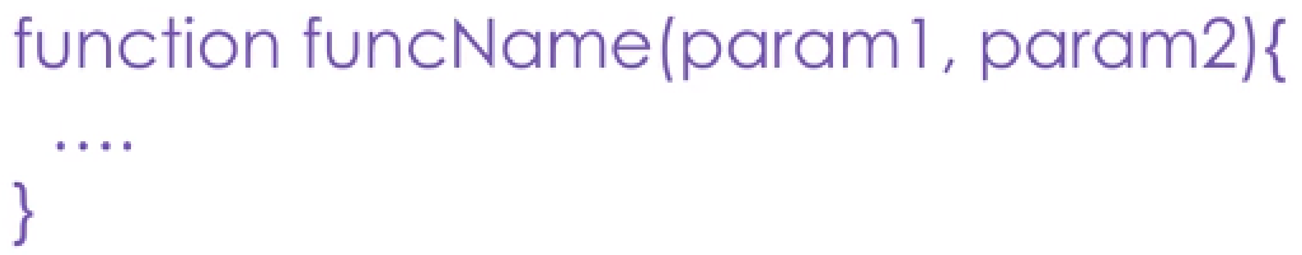
Function can be assigned to variables, added to objects, passed to another functions as arguments, and returned from functions.



```
function sayHi(){
    console.log("Hi!");
}

sayHi();    //outputs "Hi!"
var sayHi2 = sayHi;
sayHi2();   //outputs "Hi!"
```

Parameters

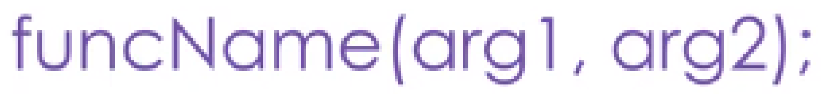
- Parameters are values that you can pass to a function.
- Parameters copy the value of what's provided.
- To create a function with parameters, declare parameter names inside of the parenthesis.
- In JavaScript, any number of parameter can be passed to any function.



```
function funcName(param1, param2){
  ....
}
```

Function Arguments

- To pass value to a function, you will call that function with it's name and with what's called arguments.
- This arguments directly corresponds to the parameter of your method. (ex. arg 1 corresponds to param1, arg 2 corresponds to param2 etc.)



```
funcName(arg1, arg2);
```

Variable Scope

- JavaScript Variables are said to have function scope. That means variables declared within a function using var keyword do **NOT** exist outside of the function. It is considered as local and will give undefined if we try to access it outside the function.
- If you declare a variable without using the var keyword, then the variable will be available outside of the function.
- This process is called Hoisting. It is the process of assigning loose variables as global.

Comments

- A comment is a statement that is NOT processed.
- Use a comment to leave notes for yourself and other developers.

Single-Line Comments

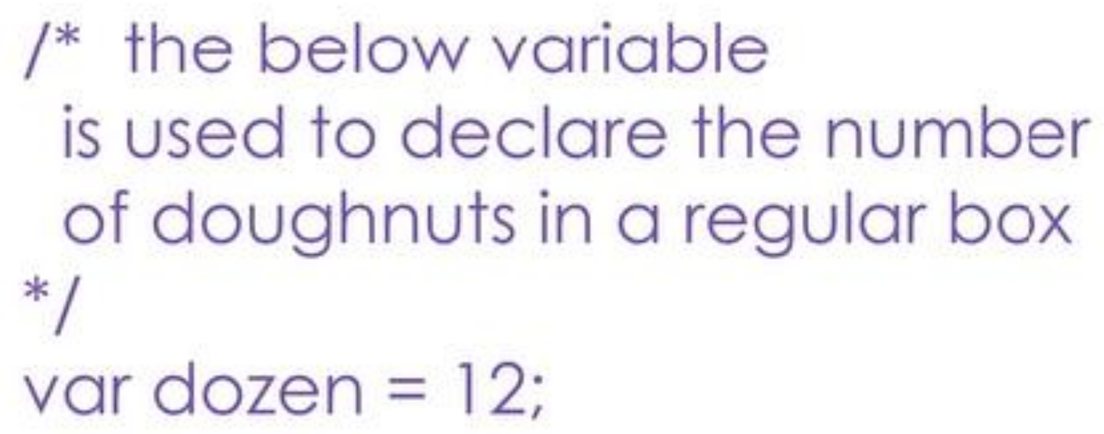
- Only covers a span of one line of code.
- A single line comment is specified using two forward-slashes(//).

```
// this is a comment.
var a = 12;
var elem =
document.getElementById('btn');
...
```

Comments

Multi-Line Comments

- As multi-line comment is started using a forward slash followed by an asterisk (/*)
- A multi-line comment is ended using an asterisk followed by a forward slash (*/).



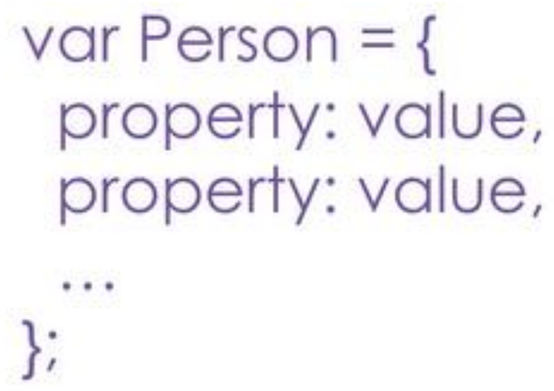
```
/*  the below variable
 is used to declare the number
 of doughnuts in a regular box
*/
var dozen = 12;
```

Objects

- An object represents a real-world entity.
- Objects have properties and methods.
- Objects are great way to group property and behavior in to one container.

Creating Objects – Literal notation

- You can use the **literal** notation to define an object.
- Use curly braces to contain the key/value pairs.



```
var Person = {
  property: value,
  property: value,
  ...
};
```

Creating Objects – Constructor notation

- You can use the **constructor** notation to define an object.
- Use the new keyword with the object’s constructor function.

```
var Person = new Object();
```

Accessing Properties

- You can access an object’s properties value by using the **dot-notation**.
- Use a period after the object name followed by the name of the property.

  ex. Person.name

- You can access an object’s properties value by using the **array-notation**.
- Access the property similar to how you would an array.

  ex. Person[“name”]

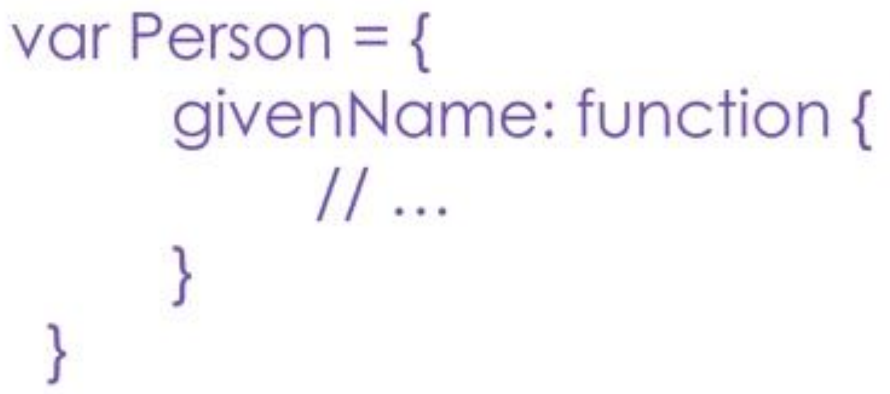
Setting Properties

- You can set properties of an object by using the **dot-notation** or **array-notation** followed by the assignment.

```
Person.name = "Adam";

Person["name"] = "Adam";
```

Methods

- A method is a function that is assigned as a property of an object.



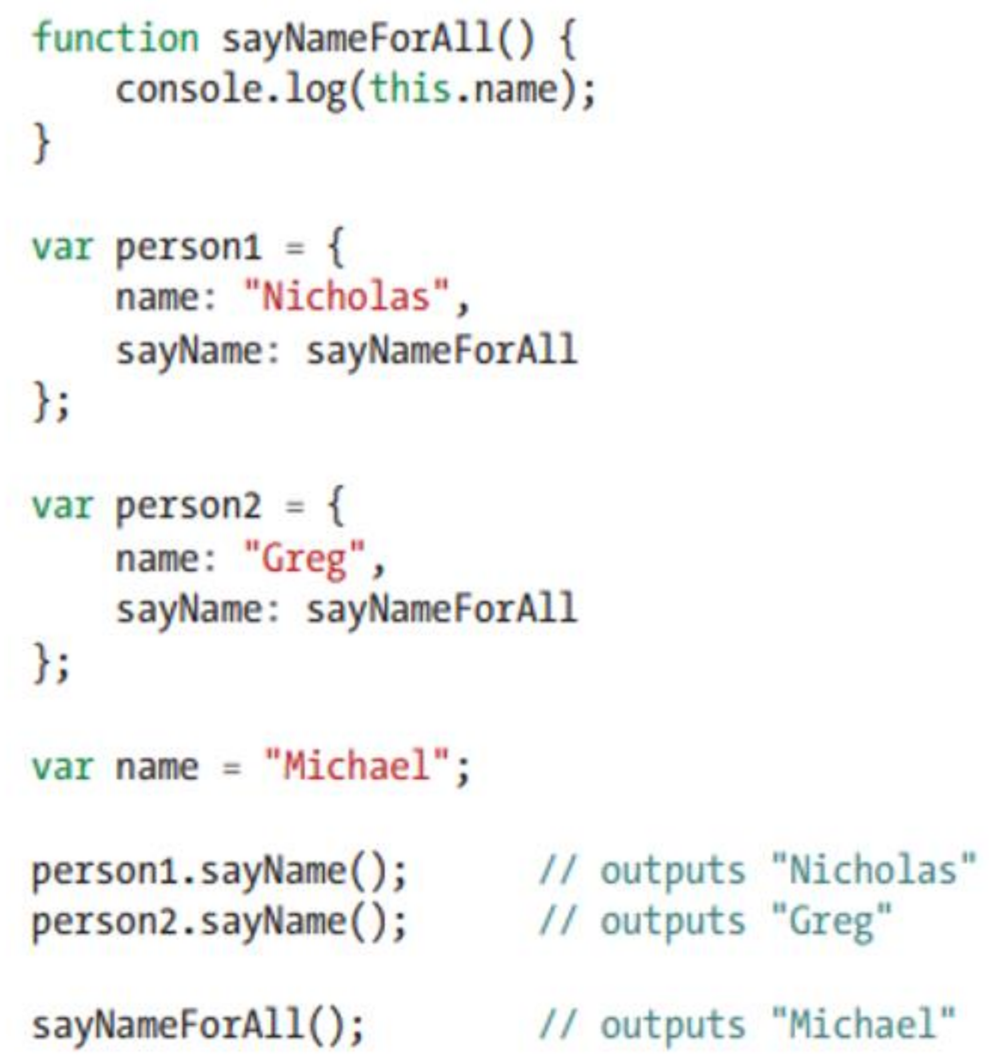
```
var Person = {
        givenName: function {
                // ...
        }
  }
```

Executing Methods

- You can call a method similar to as you would a function, except that you use the dot-notation.

```
person.givenName();

person.methodTwo(param1, param2);
```

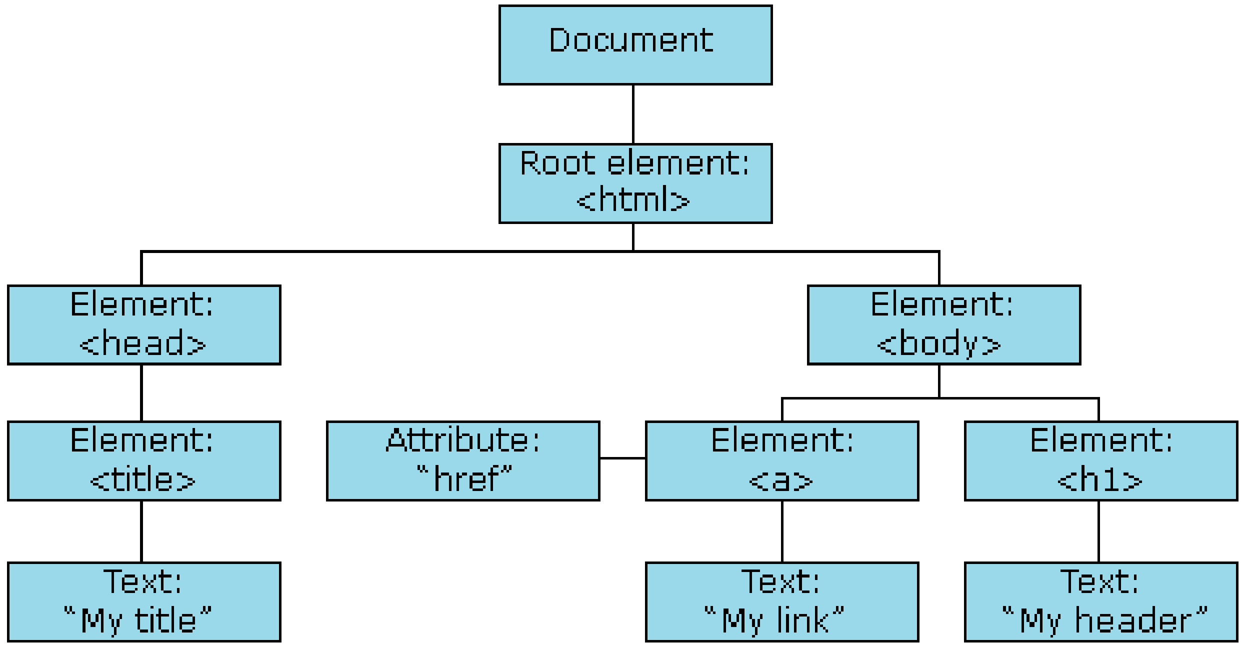
this keyword Example



```
function sayNameForAll() {
    console.log(this.name);
}

var person1 = {
    name: "Nicholas",
    sayName: sayNameForAll
};

var person2 = {
    name: "Greg",
    sayName: sayNameForAll
};

var name = "Michael";

person1.sayName();      // outputs "Nicholas"
person2.sayName();      // outputs "Greg"

sayNameForAll();        // outputs "Michael"
```

JavaScript HTML DOM Elements

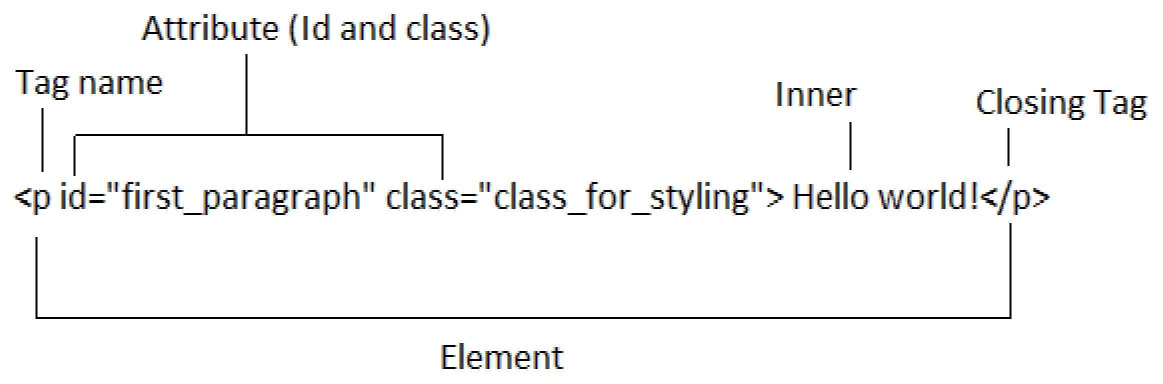
- We can access DOM elements
- Edit the DOM elements properties
- Create/remove DOM elements



JavaScript HTML DOM Elements

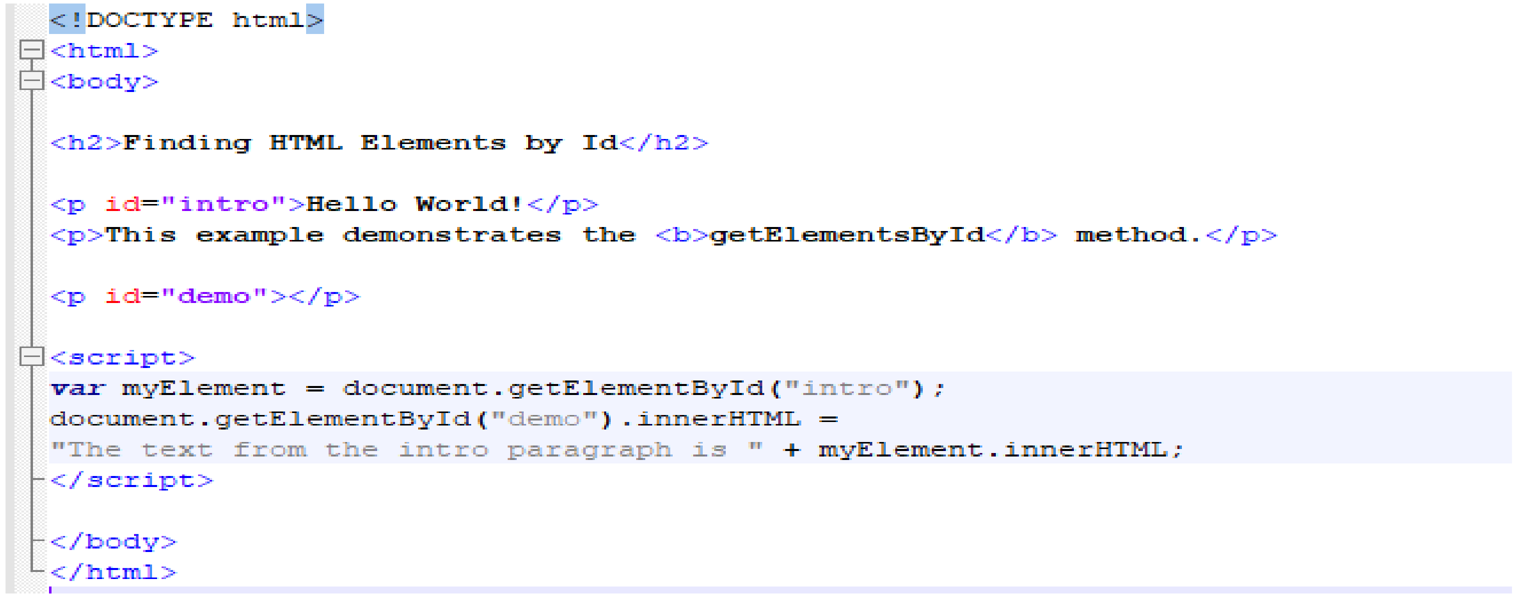
Finding HTML elements

- We need to find the elements for DOM manipulation. There are several ways to do this:
  - Finding HTML elements by id
  - Finding HTML elements by tag name
  - Finding HTML elements by class name


Attribute (Id and class)
Tag name
Inner
Closing Tag
<p id="first_paragraph" class="class_for_styling"> Hello world!</p>
Element

JavaScript HTML DOM Elements

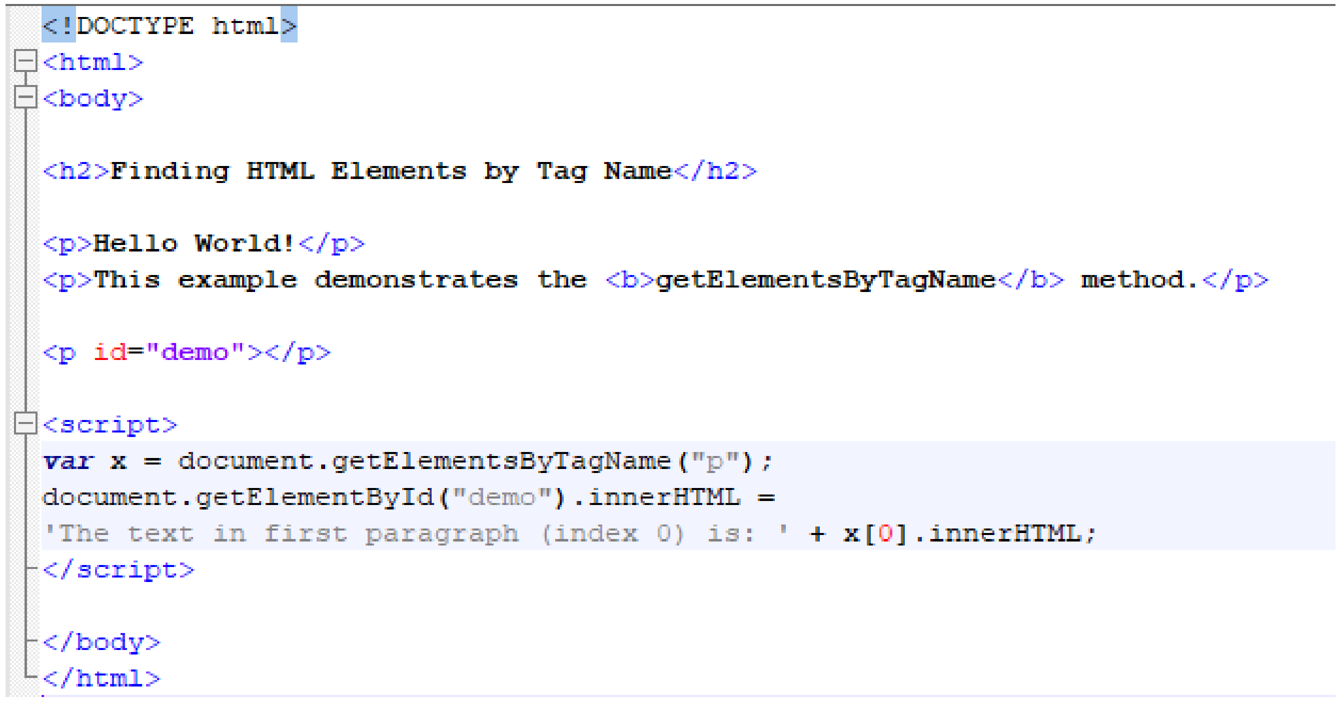
- **Finding HTML elements by id**
  - The easiest way to find an HTML element in the DOM, is by using the element id.
  - This example finds the element with id="intro":
  - If the element is found, the method will return the element as an object (in myElement).
  - If the element is not found, myElement will contain null.



```
<!DOCTYPE html>
<html>
<body>

<h2>Finding HTML Elements by Id</h2>

<p id="intro">Hello World!</p>
<p>This example demonstrates the <b>getElementsById</b> method.</p>

<p id="demo"></p>

<script>
var myElement = document.getElementById("intro");
document.getElementById("demo").innerHTML =
"The text from the intro paragraph is " + myElement.innerHTML;
</script>

</body>
</html>
```

JavaScript HTML DOM Elements

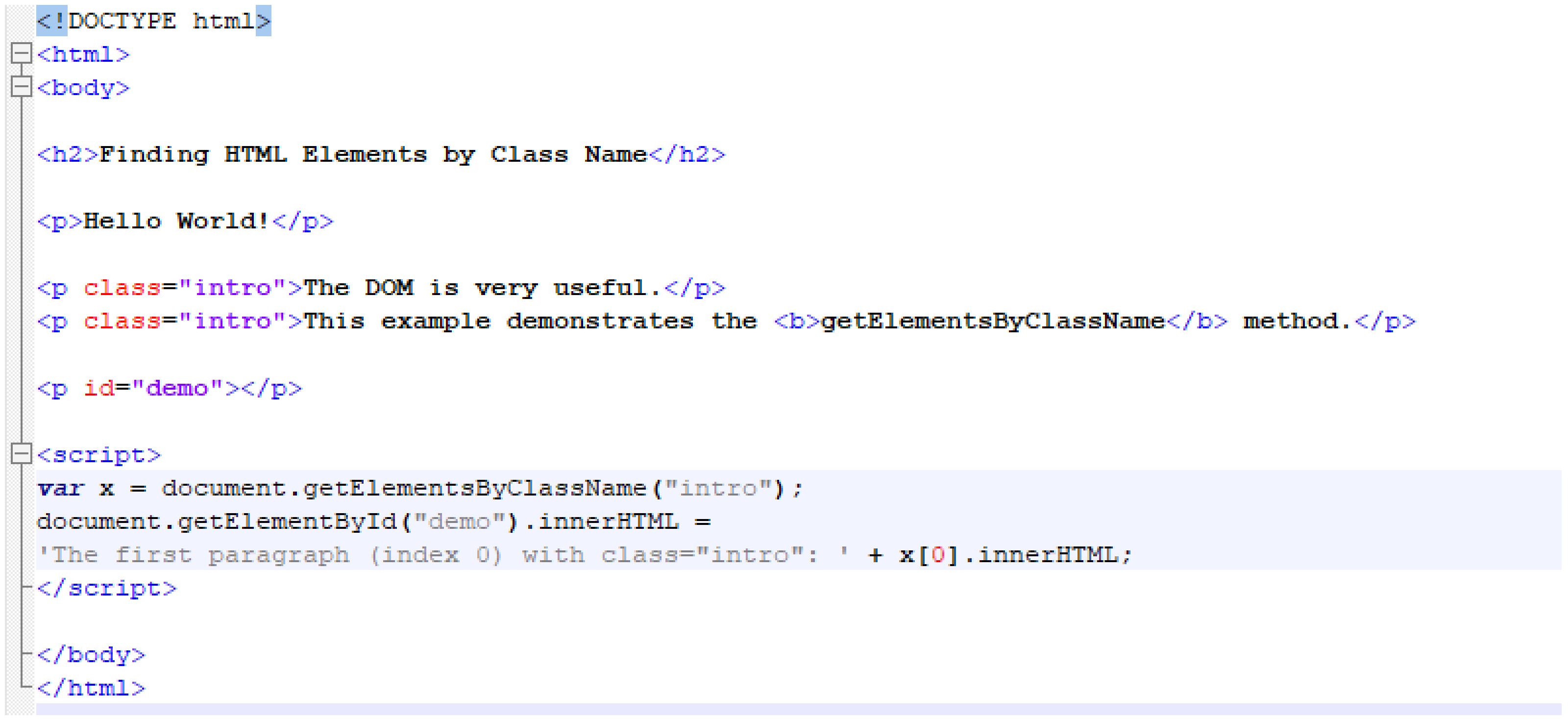
- **Finding HTML Elements by Tag Name**
  - This example finds all <p> elements:
  - This example finds the element with id="main", and then finds all <p> elements inside "main":



```
<!DOCTYPE html>
<html>
<body>

<h2>Finding HTML Elements by Tag Name</h2>

<p>Hello World!</p>
<p>This example demonstrates the <b>getElementsByTagName</b> method.</p>

<p id="demo"></p>

<script>
var x = document.getElementsByTagName("p");
document.getElementById("demo").innerHTML =
'The text in first paragraph (index 0) is: ' + x[0].innerHTML;
</script>

</body>
</html>
```

JavaScript HTML DOM Elements

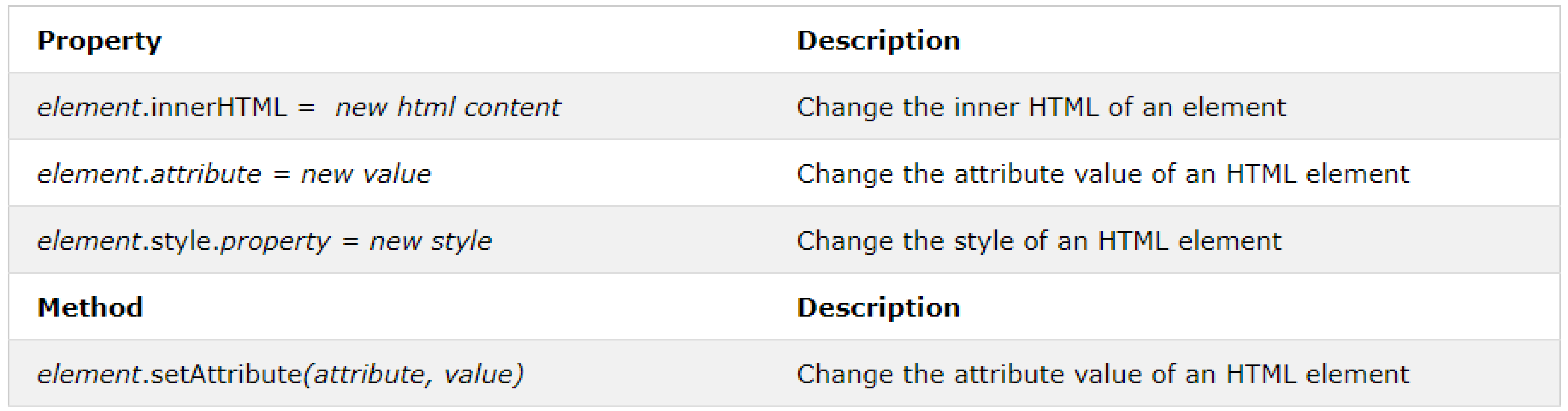
- **Finding HTML Elements by Class Name**
  - If you want to find all HTML elements with the same class name, use getElementsByClassName().
  - This example returns a list of all elements with class="intro".



```
<!DOCTYPE html>
<html>
<body>

<h2>Finding HTML Elements by Class Name</h2>

<p>Hello World!</p>

<p class="intro">The DOM is very useful.</p>
<p class="intro">This example demonstrates the <b>getElementsByClassName</b> method.</p>

<p id="demo"></p>

<script>
var x = document.getElementsByClassName("intro");
document.getElementById("demo").innerHTML =
'The first paragraph (index 0) with class="intro": ' + x[0].innerHTML;
</script>

</body>
</html>
```

JavaScript HTML DOM Elements

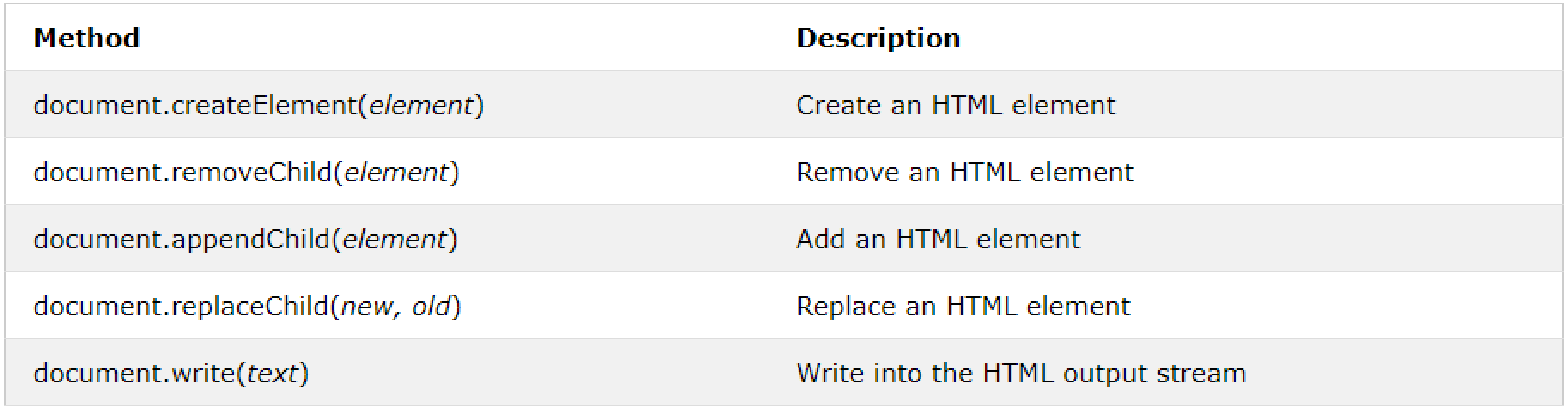
- **Changing DOM Elements properties**



Property	**Description**
element.innerHTML = *new html content*	Change the inner HTML of an element
element.attribute = *new value*	Change the attribute value of an HTML element
element.style.*property* = *new style*	Change the style of an HTML element
Method	**Description**
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element

JavaScript HTML DOM Elements

- **Create/remove DOM elements**



Method	**Description**
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element
document.replaceChild(*new, old*)	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

JavaScript HTML DOM EventListener

- **Changing DOM Elements properties**
  - The addEventListener() method attaches an event handler to the specified element.
  - You can easily remove an event listener by using the removeEventListener() method.
- **Syntax**

```
element.addEventListener(event, function, useCapture);
```

  - The first parameter is the type of the event (like "click" or "mousedown").
  - The second parameter is the function we want to call when the event occurs.
  - The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

Event Bubbling or Event Capturing?

- There are two ways of event propagation in the HTML DOM, bubbling and capturing.
- Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a <div> element, and the user clicks on the <p> element, which element's "click" event should be handled first?
- In bubbling the inner most element's event is handled first and then the outer:

  i.e. the <p> element's click event is handled first, then the <div> element's click event.
- In capturing the outer most element's event is handled first and then the inner:

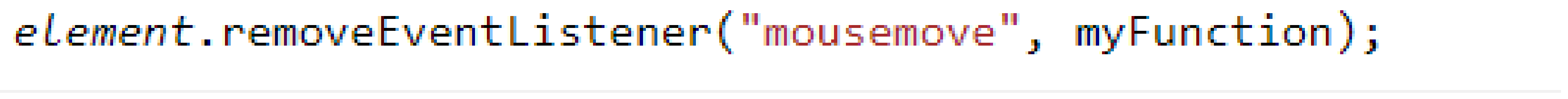
  i.e. the <div> element's click event will be handled first, then the <p> element's click event.
- With the addEventListener() method you can specify the propagation type by using the "useCapture" parameter:

```
addEventListener(event, function, useCapture);
```

- The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

The removeEventListener() method

- The removeEventListener() method removes event handlers that have been attached with the addEventListener() method:



```
element.removeEventListener("mousemove", myFunction);
```

JavaScript Timing Events

- Window object represents an open window in a browser.
- The window object allows execution of code at specified time intervals. These time intervals are called timing events.
- The two key methods to use with JavaScript are:
  - setTimeout(function, milliseconds)

    Executes a function, after waiting a specified number of milliseconds.
  - setInterval(function, milliseconds)

    Same as setTimeout(), but repeats the execution of the function continuously.

The setTimeout() Method

- The setTimeout() method executes one time after the given time interval.
- The method takes 2 parameters
  - The first parameter is a function to be executed.
  - The second parameter indicates the number of milliseconds before execution.

The setInterval() Method

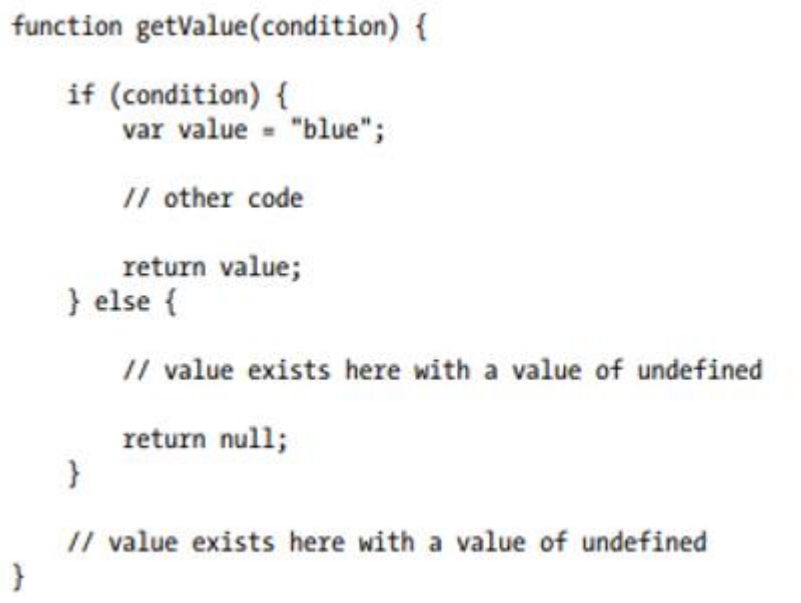
- The setInterval() method repeats a given function at every given time-interval.
- The method takes 2 parameters
  - The first parameter is the function to be executed.
  - The second parameter indicates the length of the time-interval between each execution.

ECMAScript

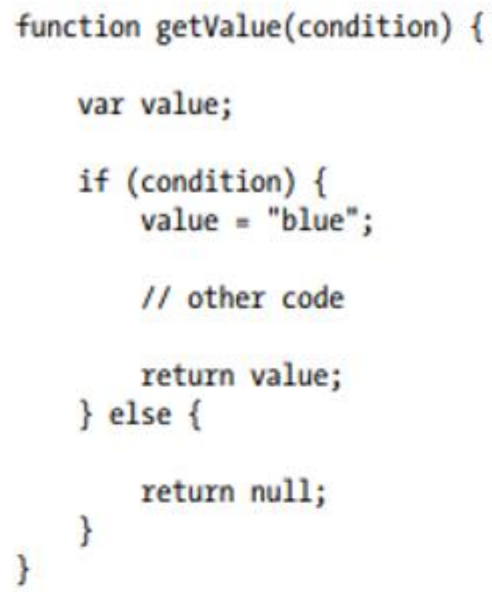
- The JavaScript core language features are defined in the ECMA-262 standard.
- The language defined in this standard is called ECMAScript.
- What you know as JavaScript in browsers and in Node.js is actually a superset of ECMAScript.

var Declarations and Hoisting

- Variable declarations using var are treated as if they're at the top of the function (or in the global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called hoisting.



```
function getValue(condition) {

    if (condition) {
        var value = "blue";

        // other code

        return value;
    } else {

        // value exists here with a value of undefined

        return null;
    }

    // value exists here with a value of undefined
}
```



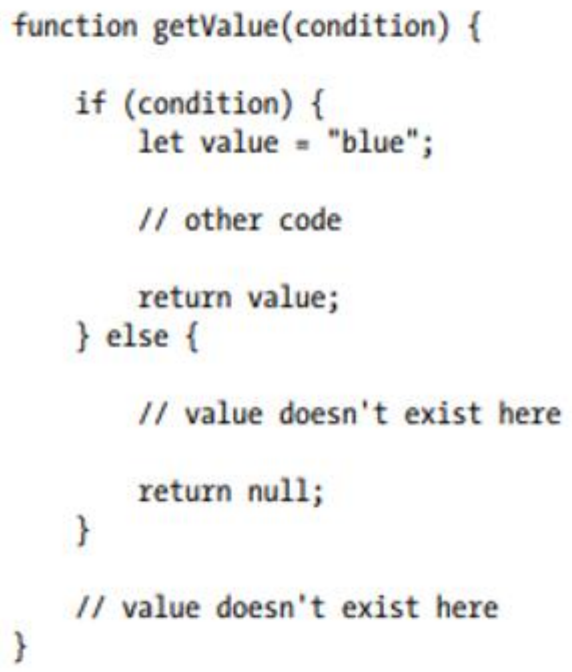
```
function getValue(condition) {

    var value;

    if (condition) {
        value = "blue";

        // other code

        return value;
    } else {

        return null;
    }
}
```

Block-Level Declarations

- Block-level declarations declare bindings that are inaccessible outside a given block scope. Block scopes, also called lexical scopes, are created in the following places:
  - Inside a function
  - Inside a block (indicated by the { and } characters)

Block-Level Declarations: let Declarations

- The let declaration syntax is the same as the syntax for var.
- You can basically replace var with let to declare a variable but limit the variable’s scope to only the current code block .
- Because let declarations are not hoisted to the top of the enclosing block, it’s best to place let declarations first in the block so they’re available to the entire block.

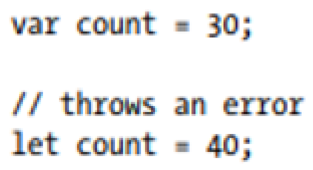


```
function getValue(condition) {

    if (condition) {
        let value = "blue";

        // other code

        return value;
    } else {

        // value doesn't exist here

        return null;
    }

    // value doesn't exist here
}
```

Block-Level Declarations: let Declarations

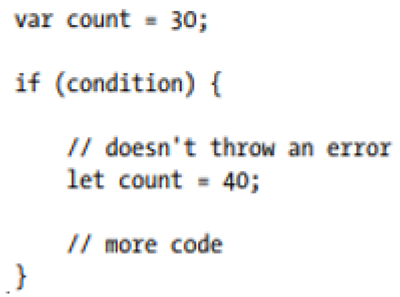
No Redeclaration

- If an identifier has already been defined in a scope, using the identifier in a let declaration inside that scope causes an error to be thrown.



```
var count = 30;

// throws an error
let count = 40;
```

- Conversely, no error is thrown if a let declaration creates a new variable with the same name as a variable in its containing scope



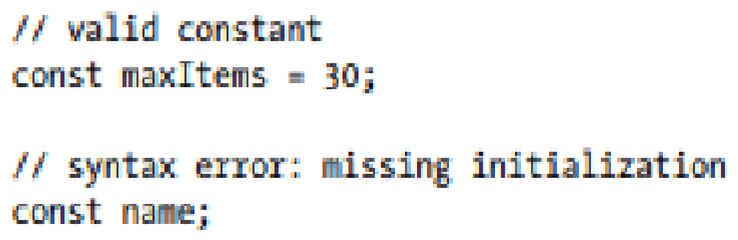
```
var count = 30;

if (condition) {

    // doesn't throw an error
    let count = 40;

    // more code
}
```

- This let declaration doesn’t throw an error because it creates a new variable called count within the if statement instead of creating count in the surrounding block. Inside the if block, this new variable shadows the global count, preventing access to it until execution exits the block.

Block-Level Declarations: const Declarations

const Declarations

- Bindings declared using const are considered constants, meaning their values cannot be changed once set.
- For this reason, every const binding must be initialized on declaration



```
// valid constant
const maxItems = 30;

// syntax error: missing initialization
const name;
```

Constants vs. let Declarations

- Constants, like let declarations, are block-level declarations. That means constants are no longer accessible once execution flows out of the block in which they were declared, and declarations are not hoisted.
- In another similarity to let, a const declaration throws an error when made with an identifier for an already defined variable in the same scope. It doesn’t matter whether that variable was declared using var (for global or function scope) or let (for block scope).
- Despite those similarities, there is one significant difference between let and const. Attempting to assign a const to a previously defined constant will throw an error in both strict and non-strict modes.
- However, unlike constants in other languages, the value a constant holds can be modified if it is an object.

```
if (condition) {
    const maxItems = 5;

    // more code
}

// maxItems isn't accessible here
```

```
var message = "Hello!";
let age = 25;

// each of these throws an error
const message = "Goodbye!";
const age = 30;
```

```
const maxItems = 5;

// throws an error
maxItems = 6;
```

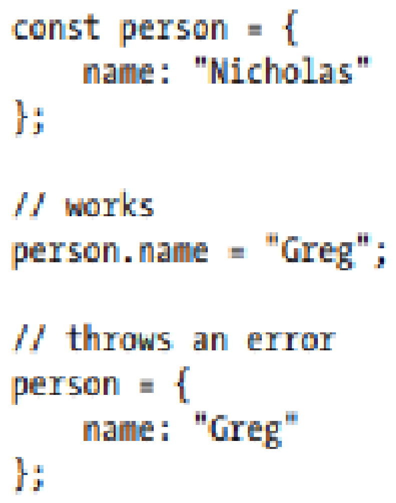
var vs constants vs let Declarations

Var	Let	Const
Hoisted	Not hoisted	Not hoisted
Not Block level scope	Block level scope	Block level scope
Redeclaration allowed	Redeclaration not allowed	Redeclaration not allowed
Same variable with var and let not allowed in same scope	Same variable with var and let not allowed in same scope	Same variable with var, let and const not allowed in same scope
Same variable with var and let allowed in different scope	Same variable with var and let allowed in different scope	Same variable with var, let and const allowed in different scope
-	-	Considered as constant, cannot be changed
Not necessary	Not necessary	Must be initialized on declaration

Block-Level Declarations

Object Declarations with const

- A const declaration for an object, prevents modification of the binding, not of the value. That means const declarations for objects don’t prevent modification of those objects.



```
const person = {
    name: "Nicholas"
};

// works
person.name = "Greg";

// throws an error
person = {
    name: "Greg"
};
```

Arrow Function

One of the most interesting new parts of ECMAScript 6 is the arrow function. Arrow functions are, as the name suggests, functions defined with a new syntax that uses an arrow (=>)

```
let reflect = value => value;

// effectively equivalent to:

let reflect = function(value) {
    return value;
};
```

One argument

```
let getName = () => "Nicholas";

// effectively equivalent to:

let getName = function() {
    return "Nicholas";
};
```

No argument

```
let sum = (num1, num2) => num1 + num2;

// effectively equivalent to:

let sum = function(num1, num2) {
    return num1 + num2;
};
```

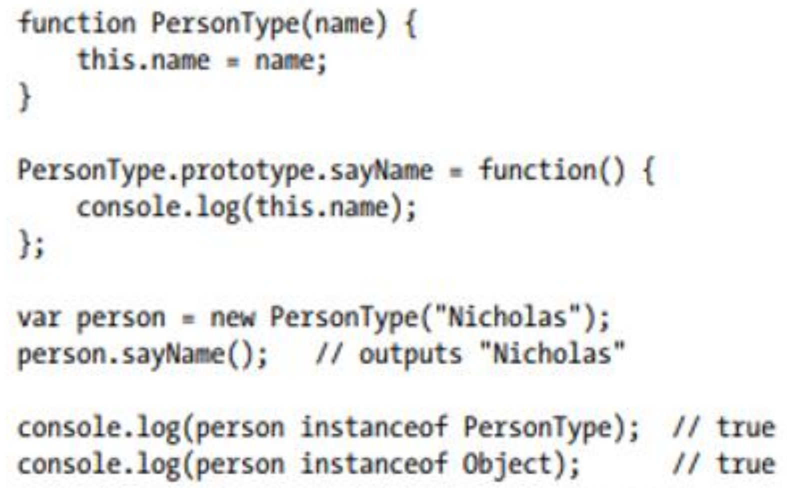
More than one argument

```
let sum = (num1, num2) => {
    return num1 + num2;
};

// effectively equivalent to:

let sum = function(num1, num2) {
    return num1 + num2;
};
```

More statements inside the function

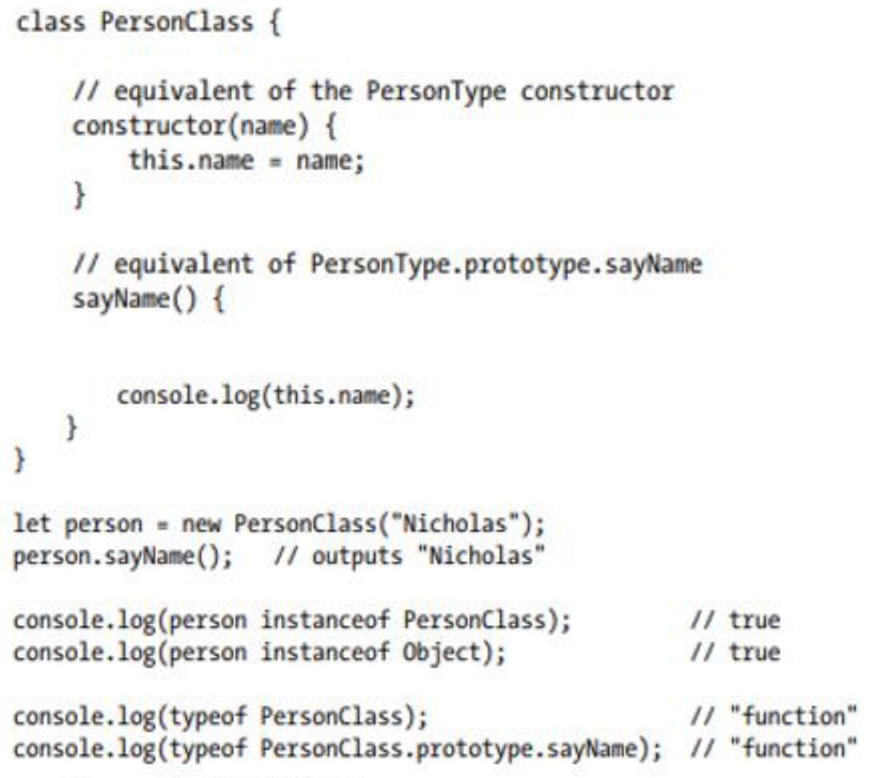
Classes

Class declarations begin with the class keyword followed by the name of the class. The rest of the syntax looks similar to concise methods in object literals but doesn't require commas between the elements of the class.



```
function PersonType(name) {
    this.name = name;
}

PersonType.prototype.sayName = function() {
    console.log(this.name);
};

var person = new PersonType("Nicholas");
person.sayName();   // outputs "Nicholas"

console.log(person instanceof PersonType);  // true
console.log(person instanceof Object);      // true
```

Class-like structure in ECMA5



```
class PersonClass {

    // equivalent of the PersonType constructor
    constructor(name) {
        this.name = name;
    }

    // equivalent of PersonType.prototype.sayName
    sayName() {


       console.log(this.name);
    }
}

let person = new PersonClass("Nicholas");
person.sayName();   // outputs "Nicholas"

console.log(person instanceof PersonClass);         // true
console.log(person instanceof Object);              // true

console.log(typeof PersonClass);                    // "function"
console.log(typeof PersonClass.prototype.sayName);  // "function"
```

Class in ECMA6