Chapter 1

Javascript: An Overview

1.1 What is Javascript?

JavaScript is a high-level, dynamic, untyped, and interpreted programming language. It has been standardized in the ECMAScript language specification. Alongside HTML and CSS, it is one of the three essential technologies of World Wide Web content production; the majority of websites employ it and it is supported by all modern web browsers without plug-ins or any kind of other extension.

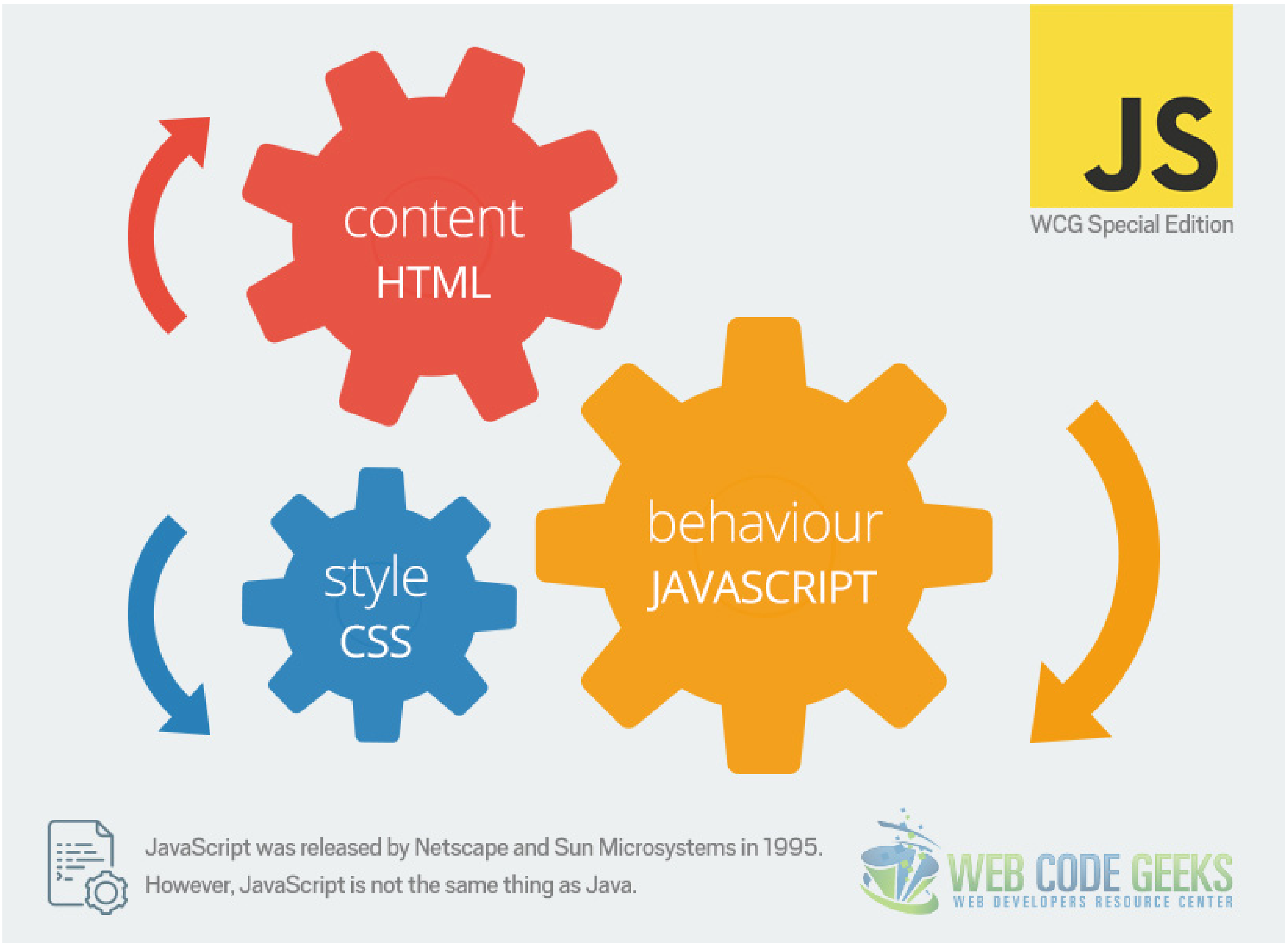
content
HTML
style
CSS
behaviour
JAVASCRIPT
JS
WCG Special Edition
JavaScript was released by Netscape and Sun Microsystems in 1995.
However, JavaScript is not the same thing as Java.
WEB CODE GEEKS
WEB DEVELOPERS RESOURCE CENTER

Figure 1.1: HTML, CSS & Javascript are the three most basic languages to create web content!

JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. It has an API for working with text, arrays, dates and regular expressions, but does not include any I/O, such as networking, storage or graphics facilities, relying for these upon the host environment in which it is embedded.

So, to conclude this paragraph, keep in mind answers to the question "What is Javascript?"

It is a programming language. It is an interpreted language. It is object-based programming. It is widely used and supported It is accessible to the beginner.

1.2 Javascript Code Implementation

1.2.1 Implementing Javascript

There are three ways to add JavaScript commands to your Web Pages:

- Embedding code
- Inline code
- External file

1.2.2 Writing Javascript

JavaScript code is typically embedded in the HTML, to be interpreted and run by the client's browser. Here are some tips to remember when writing JavaScript commands.

JavaScript code is case sensitive White space between words and tabs are ignored Line breaks are ignored except within a statement JavaScript statements end with a semi-colon: *;*

1.2.3 Objects

JavaScript supports programming with objects. Objects are a way of organizing the variables. The different screen elements such as Web pages, forms, text boxes, images, and buttons are treated as objects. Every object has its own properties and methods.

Properties define the characteristics of an object. Examples: color, length, name, height, width Methods are the actions that the object can perform or that can be performed on the object. Examples: alert, confirm, write, open, close .

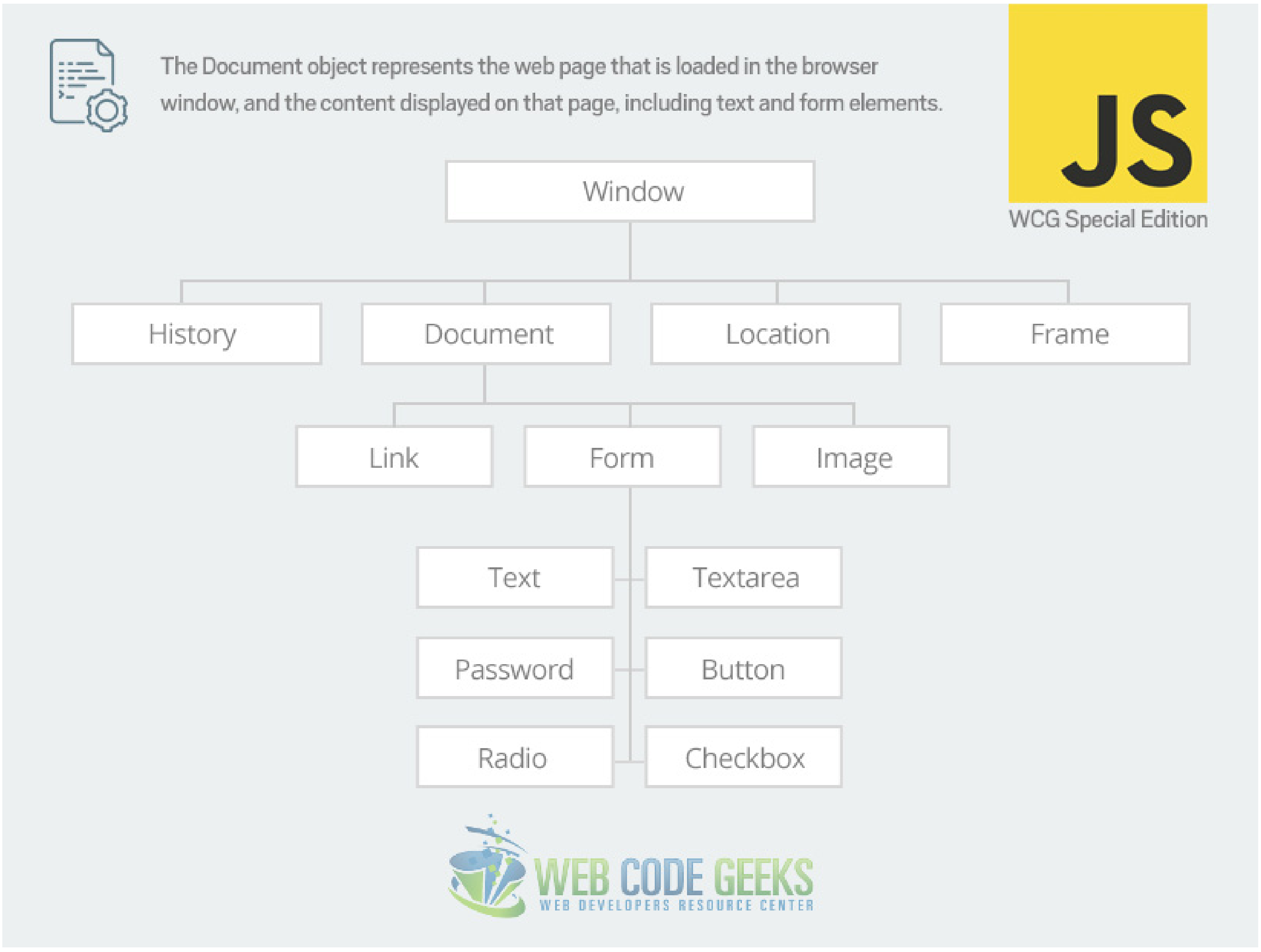
The Document object represents the web page that is loaded in the browser window, and the content displayed on that page, including text and form elements.
JS
WCG Special Edition
Window
History
Document
Location
Frame
Link
Form
Image
Text
Textarea
Password
Button
Radio
Checkbox
WEB CODE GEEKS
WEB DEVELOPERS RESOURCE CENTER

Figure 1.2: The Javascript Object Hierarchy

1.3 Javascript Events

The objects in a Web page are organized in a hierarchy. All objects have properties and methods. In addition, some objects also have "events". Events are things that happen, usually user actions, that are associated with an object. The "event handler" is a command that is used to specify actions in response to an event. Common events are:

- `onLoad` - occurs when a page loads in a browser
- `onUnload` - occurs just before the user exits a page
- `onMouseOver` - occurs when you point to an object
- `onMouseOut` - occurs when you point away from an object
- `onSubmit` - occurs when you submit a form
- `onClick` - occurs when an object is clicked

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page. When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable. Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

Chapter 2

Theoretical Questions

2.1 Can you name two programming paradigms important for JavaScript app developers?

JavaScript is a multi-paradigm language, supporting **imperative/procedural** programming along with **OOP** (Object-Oriented Programming) and **functional programming**. JavaScript supports OOP with **prototypal inheritance**.

2.2 What is functional programming?

Functional programming produces programs by composing mathematical functions and avoids shared state & mutable data. Lisp (specified in 1958) was among the first languages to support functional programming, and was heavily inspired by lambda calculus. Functional programming is an essential concept in JavaScript (one of the two pillars of JavaScript). Several common functional utilities were added to JavaScript in ES5.

Focus is on: - Function purity - Avoiding side-effects - Simple function composition

2.3 What is the difference between classical inheritance and prototypal inheritance?

Class Inheritance: instances inherit from classes (like a blueprint - a description of the class), and create sub-class relationships: hierarchical class taxonomies. Instances are typically instantiated via constructor functions with the `new` keyword. Class inheritance may or may not use the `class` keyword from ES6.

Prototypal Inheritance: instances inherit directly from other objects. Instances are typically instantiated via factory functions or `Object.create()`. Instances may be composed from many different objects, allowing for easy selective inheritance.

2.4 When is prototypal inheritance an appropriate choice?

There is more than one type of prototypal inheritance:

- **Delegation** (i.e., the prototype chain).
- **Concatenative** (i.e. mixins, `Object.assign()`).
- **Functional** (Not to be confused with functional programming. A function used to create a closure for private state/encapsulation).

Each type of prototypal inheritance has its own set of use-cases, but all of them are equally useful in their ability to enable **composition**, which creates **has-a** or **uses-a** or **can-do** relationships as opposed to the **is-a** relationship created with class inheritance.

2.5 What is asynchronous programming, and why is it important in JavaScript?

Synchronous programming means that, barring conditionals and function calls, code is executed sequentially from top-to-bottom, blocking on long-running tasks such as network requests and disk I/O.

Asynchronous programming means that the engine runs in an event loop. When a blocking operation is needed, the request is started, and the code keeps running without blocking for the result. When the response is ready, an interrupt is fired, which causes an event handler to be run, where the control flow continues. In this way, a single program thread can handle many concurrent operations.

User interfaces are asynchronous by nature, and spend most of their time waiting for user input to interrupt the event loop and trigger event handlers. Node is asynchronous by default, meaning that the server works in much the same way, waiting in a loop for a network request, and accepting more incoming requests while the first one is being handled.

This is important in JavaScript, because it is a very natural fit for user interface code, and very beneficial to performance on the server.

2.6 What are JavaScript data types?

In JavaScript, there are three primary data types, two composite data types, and two special data types.

Primary Data Types

- String
- Number
- Boolean

Composite Data Types

- Object
- Array

Special Data Types

- Null
- Undefined

2.7 What is the difference between "==" and "==="?

"==" checks only for equality in value whereas "===" is a stricter equality test and returns false if either the value or the type of the two variables are different. So, the second option needs both the value and the type to be the same for the operands.

2.8 What is an undefined value in JavaScript?

Undefined value means the:

- Variable used in the code doesn’t exist
- Variable is not assigned to any value
- Property doesn’t exist

2.9 What are the different types of errors in JavaScript?

There are three types of errors:

- **Load time errors**: Errors which come up when loading a web page like improper syntax errors are known as Load time errors and it generates the errors dynamically.
- **Run time errors**: Errors that come due to misuse of the command inside the HTML language.
- **Logical Errors**: These are errors that occur due to the wrong logic performed on a function.

2.10 Define event bubbling

JavaScript allows DOM elements to be nested inside each other. In such a case, if the handler of the child is clicked, the handler of parent will also work as if it were clicked too.

2.11 What is the significance, and what are the benefits, of including *use strict* at the beginning of a JavaScript source file?

`use strict` is a way to voluntarily enforce stricter parsing and error handling on your JavaScript code at runtime. Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions. In general, it is a good practice.

Some of the key benefits of strict mode include:

Makes debugging easier. Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions, alerting you sooner to problems in your code and directing you more quickly to their source.

Prevents accidental globals. Without strict mode, assigning a value to an undeclared variable automatically creates a global variable with that name. This is one of the most common errors in JavaScript. In strict mode, attempting to do so throws an error.

Eliminates this coercion. Without strict mode, a reference to a this value of null or undefined is automatically coerced to the global. This can cause many headfakes and pull-out-your-hair kind of bugs. In strict mode, referencing a a this value of null or undefined throws an error.

Disallows duplicate property names or parameter values. Strict mode throws an error when it detects a duplicate named property in an object (e.g., `var object ={foo:"bar", foo:"baz"};`) or a duplicate named argument for a function (e.g., `function foo(val1, val2, val1){}`), thereby catching what is almost certainly a bug in your code that you might otherwise have wasted lots of time tracking down.

Makes eval() safer. There are some differences in the way `eval()` behaves in strict mode and in non-strict mode. Most significantly, in strict mode, variables and functions declared inside of an `eval()` statement are not created in the containing scope (they are created in the containing scope in non-strict mode, which can also be a common source of problems).

Throws error on invalid usage of delete. The delete operator (used to remove properties from objects) cannot be used on non-configurable properties of the object. Non-strict code will fail silently when an attempt is made to delete a non-configurable property, whereas strict mode will throw an error in such a case.

2.12 What are Screen objects?

Screen objects are used to read the information from the client's screen. The properties of screen objects are -

- AvalHeight: Gives the height of client's screen
- AvailWidth: Gives the width of client's screen.

- ColorDepth: Gives the bit depth of images on the client’s screen
- Height: Gives the total height of the client’s screen, including the taskbar
- Width: Gives the total width of the client’s screen, including the taskbar

Chapter 3

Code Output Questions

3.1 What will the code below output to the console and why?

```
(function(){
  var a = b = 3;
})();

console.log("a defined? " + (typeof a !== 'undefined'));
console.log("b defined? " + (typeof b !== 'undefined'));
```

3.1.1 Answer:

Since both a and b are defined within the enclosing scope of the function, and since the line they are on begins with the var keyword, most JavaScript developers would expect typeof a and typeof b to both be undefined in the above example.

However, that is not the case. The issue here is that most developers incorrectly understand the statement var a = b = 3; to be shorthand for:

`var b =3; var a =b;`

But in fact, `var a =b =3;` is actually shorthand for:

`b =3; var a =b;`

As a result (if you are not using strict mode), the output of the code snippet would be:

```
a defined? false
b defined? true
```

3.2 What will the code below output to the console and why?

```
var myObject = {
    foo: "bar",
    func: function() {
        var self = this;
        console.log("outer func:  this.foo = " + this.foo);
        console.log("outer func:  self.foo = " + self.foo);
        (function() {
            console.log("inner func:  this.foo = " + this.foo);
            console.log("inner func:  self.foo = " + self.foo);
        }());
    }
};
myObject.func();
```

The above code will output the following to the console:

```
outer func:  this.foo = bar
outer func:  self.foo = bar
inner func:  this.foo = undefined
inner func:  self.foo = bar
```

In the outer function, both `this` and `self` refer to `myObject` and therefore both can properly reference and access `foo`.

In the inner function, though, `this` no longer refers to `myObject`. As a result, `this.foo` is undefined in the inner function, whereas the reference to the local variable `self` remains in scope and is accessible there. (Prior to ECMA 5, this in the inner function would refer to the global window object; whereas, as of ECMA 5, `this` in the inner function would be `undefined`.)

3.3 What will the code below output? Explain your answer.

```
console.log(0.1 + 0.2);
console.log(0.1 + 0.2 == 0.3);
```

An educated answer to this question would simply be: "You can’t be sure. it might print out "0.3" and "true", or it might not. Numbers in JavaScript are all treated with floating point precision, and as such, may not always yield the expected results."

The example provided above is classic case that demonstrates this issue. Surprisingly, it will print out:

```
0.30000000000000004
false
```

3.4 Consider the following code snippet:

```
for (var i = 0; i < 5; i++) {
  var btn = document.createElement('button');
  btn.appendChild(document.createTextNode('Button ' + i));
  btn.addEventListener('click', function(){ console.log(i); });
  document.body.appendChild(btn);
}
```

(a) What gets logged to the console when the user clicks on "Button 4" and why?

(b) Provide one or more alternate implementations that will work as expected.

3.4.1 Answer:

(a) No matter what button the user clicks the number 5 will always be logged to the console. This is because, at the point that the `onclick` method is invoked (for any of the buttons), the `for` loop has already completed and the variable `i` already has a value of 5. (Bonus points for the interviewee if they know enough to talk about how execution contexts, variable objects, activation objects, and the internal "scope" property contribute to the closure behavior.)

(b) The key to making this work is to capture the value of `i` at each pass through the `for` loop by passing it into a newly created function object. Here are three possible ways to accomplish this:

```
for (var i = 0; i < 5; i++) {
  var btn = document.createElement('button');
  btn.appendChild(document.createTextNode('Button ' + i));
  btn.addEventListener('click', (function(i) {
    return function() { console.log(i); };
  })(i));
  document.body.appendChild(btn);
}
```

3.5 What will the code below output to the console and why?

```
var arr1 = "john".split('');
var arr2 = arr1.reverse();
var arr3 = "jones".split('');
arr2.push(arr3);
console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));
console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1));
```

3.5.1 Answer:

The logged output will be:

```
"array 1: length=5 last=j,o,n,e,s"
"array 2: length=5 last=j,o,n,e,s"
arr1 and arr2 are the same after the above code is executed for the following reasons:
```

Calling an array object's `reverse()` method doesn't only return the array in reverse order, it also reverses the order of the array itself (i.e., in this case, `arr1`).

The `reverse()` method returns a reference to the array itself (i.e., in this case, `arr1`). As a result, `arr2` is simply a reference to (rather than a copy of) `arr1`. Therefore, when anything is done to `arr2` (i.e., when we invoke `arr2.push(arr3);`), `arr1` will be affected as well since `arr1` and `arr2` are simply references to the same object.

3.6 The following recursive code will cause a stack overflow if the array list is too large. How can you fix this and still retain the recursive pattern?

```
var list = readHugeList();

var nextListItem = function() {
    var item = list.pop();

    if (item) {
        // process the list item...
        nextListItem();
    }
};
```

3.6.1 Answer:

The potential stack overflow can be avoided by modifying the `nextListItem` function as follows:

```
var list = readHugeList();

var nextListItem = function() {
    var item = list.pop();

    if (item) {
        // process the list item...
        setTimeout( nextListItem, 0);
    }
};
```

The stack overflow is eliminated because the event loop handles the recursion, not the call stack. When `nextListItem` runs, if item is not null, the timeout function (`nextListItem`) is pushed to the event queue and the function exits, thereby leaving the call stack clear. When the event queue runs its timed-out event, the next `item` is processed and a timer is set to again invoke `nextListItem`. Accordingly, the method is processed from start to finish without a direct recursive call, so the call stack remains clear, regardless of the number of iterations.

3.7 What will the code below output to the console and why?

```
console.log(1 +  "2" + "2");
console.log(1 +  +"2" + "2");
console.log(1 +  -"1" + "2");
console.log(+"1" +  "1" + "2");
console.log( "A" - "B" + "2");
console.log( "A" - "B" + 2);
```

3.7.1 Answer:

The above code will output the following to the console:

```
"122"
"32"
"02"
"112"
"NaN2"
NaN
```

Why?

The fundamental issue here is that JavaScript (ECMAScript) is a loosely typed language and it performs automatic type conversion on values to accommodate the operation being performed. Let's see how this plays out with each of the above examples.

Example 1: `1 + "2" + "2"` **Outputs**: `"122"` **Explanation**: The first operation to be performed in `1 + "2"`. Since one of the operands (`"2"`) is a string, JavaScript assumes it needs to perform string concatenation and therefore converts the type of 1 to `"1"`, `1 + "2"` yields `"12"`. Then, `"12" + "2"` yields `"122"`.

Example 2: `1 + +"2" + "2"` **Outputs**: `"32"` **Explanation**: Based on order of operations, the first operation to be performed is `+"2"` (the extra + before the first `"2"` is treated as a unary operator). Thus, JavaScript converts the type of `"2"` to numeric and then applies the unary + sign to it (i.e., treats it as a positive number). As a result, the next operation is now `1 + 2` which of course yields `3`. But then, we have an operation between a number and a string (i.e., `3` and `"2"`), so once again JavaScript converts the type of the numeric value to a string and performs string concatenation, yielding `"32"`.

Example 3: `1 + -"1" + "2"` **Outputs**: `"02"` **Explanation**: The explanation here is identical to the prior example, except the unary operator is - rather than +. So `"1"` becomes `1`, which then becomes `-1` when the - is applied, which is then added to `1` yielding `0`, which is then converted to a string and concatenated with the final `"2"` operand, yielding `"02"`.

Example 4: `+"1" + "1" + "2"` **Outputs**: `"112"` **Explanation**: Although the first `"1"` operand is typecast to a numeric value based on the unary + operator that precedes it, it is then immediately converted back to a string when it is concatenated with the second `"1"` operand, which is then concatenated with the final `"2"` operand, yielding the string `"112"`.

Example 5: `"A" -"B" + "2"` **Outputs**: `"NaN2"` **Explanation**: Since the - operator can not be applied to strings, and since neither `"A"` nor `"B"` can be converted to numeric values, `"A" -"B"` yields NaN which is then concatenated with the string `"2"` to yield "NaN2".

Example 6: `"A" -"B" + 2` **Outputs**: `NaN` **Explanation**: As exlained in the previous example, `"A" -"B"` yields `NaN`. But any operator applied to NaN with any other numeric operand will still yield `NaN`.

3.8 What will be the output when the following code is executed? Explain.

```
console.log(false == '0')
console.log(false === '0')
```

3.8.1 Answer:

The code will output:

```
true
false
```

In JavaScript, there are two sets of equality operators. The triple-equal operator === behaves like any traditional equality operator would: evaluates to true if the two expressions on either of its sides have the same type and the same value. The double-equal operator, however, tries to coerce the values before comparing them. It is therefore generally good practice to use the === rather than ==. The same holds true for `!==` vs `!=`.

Chapter 4

Write Code Questions

4.1 What will the code below output to the console and why?

Create a function that, given a DOM Element on the page, will visit the element itself and all of its descendents (not just its immediate children). For each element visited, the function should pass that element to a provided callback function.

The arguments to the function should be:

a DOM element a callback function (that takes a DOM element as its argument)

4.1.1 Answer

Visiting all elements in a tree (DOM) is a classic Depth-First-Search algorithm application. Here's an example solution:

```
function Traverse(p_element,p_callback) {
   p_callback(p_element);
   var list = p_element.children;
   for (var i = 0; i < list.length; i++) {
       Traverse(list[i],p_callback);  // recursive call
   }
}
```

4.2 Write a `sum` method which will work properly when invoked using either syntax below.

```
console.log(sum(2,3));   // Outputs 5
console.log(sum(2)(3));  // Outputs 5
```

There might be quite some ways to do this, but one of them is shown below:

```
function sum(x) {
  if (arguments.length == 2) {
    return arguments[0] + arguments[1];
  } else {
    return function(y) { return x + y; };
  }
}
```

In JavaScript, functions provide access to an `arguments` object which provides access to the actual arguments passed to a function. This enables us to use the `length` property to determine at runtime the number of arguments passed to the function.

If two arguments are passed, we simply add them together and return.

4.3 Write a simple function (less than 80 characters) that returns a boolean indicating whether or not a string is a palindrome.

The following one line function will return `true` if `str` is a palindrome; otherwise, it returns false.

```
function isPalindrome(str) {
    str = str.replace(/\W/g, '').toLowerCase();
    return (str == str.split('').reverse().join(''));
}
```

For example:

```
console.log(isPalindrome("level"));                    // logs 'true'
console.log(isPalindrome("levels"));                   // logs 'false'
console.log(isPalindrome("A car, a man, a maraca"));  // logs 'true'
```

4.4 Functional Javascript

4.4.1 Given the following array, build me an array of cars with those colours:

```
var colors = ['blue', 'black', 'red'];
var cars = ...;
```

4.4.1.1 Answer:

```
var colors = ['blue', 'black', 'red'];
var cars = colors.map(buildCar);

function buildCar (color) {
    return new Car(color);
}
```

If they wrote a factory function wrapping Object.create in the first step, they can just map directly to that function, which makes this a bit more elegant.

4.5 Dynamic Objects

4.5.1 I've created an array of 1000 cars. When I call run on the first car, I want it to run as normal. When I call run on any of the others and any cars created in the future, I want it to run as normal, but also log The color car is now running to the console.

4.5.1.1 Answer:

```
cars[0].run = cars[0].run;

var oldRun = Car.prototype.run;
Car.prototype.run = function () {
    console.log("The " + this.color + " car is now running");
    return oldRun.apply(this, arguments);
};
```

This covers:

property lookup on objects the use of apply or call changing the behaviour of existing objects by modifying the prototype

4.6 Binding Shim

4.6.1 Can you shim the "bind" function?

4.6.1.1 Answer:

```
Function.prototype.bind = function (context) {
    var self = this;
    return function () {
        return self.apply(context, arguments);
    }
};
```

This can also be done by writing a **bind** function like underscore or CoffeeScript does, which is good if you don't want to be modifying JavaScript built-ins.

4.7 Animation

4.7.1 How could you implement moveLeft animation?

4.7.1.1 Answer:

Use `setInterval` that will place the element to the left position by some pixels in every 10ms. Hence, you will see the element moving towards the desired position. When you call setInterval, it returns a timeId. After reaching the desired location, you have to clear the time interval so that function will not be called again and again in every 10ms.

```
function moveLeft(elem, distance) {
  var left = 0;

  function frame() {
    left++;
    elem.style.left = left + 'px';

    if (left == distance)
      clearInterval(timeId)
  }

  var timeId = setInterval(frame, 10); // draw every 10ms
}
```

4.8 Memorization

4.8.1 How could you implement cache to save calculation time for a recursive fibonacci function?

4.8.1.1 Answer:

You could use poor man's memoization with a global variable. If fibonacci is already calculated it is served from the global memo array otherwise it is calculated.

```
var memo = [];

function _fibonacci(n) {
   if(memo[n]){
    return memo[n];
   }
   else if (n < 2){
     return 1;
   }else{
     fibonacci(n-2) + fibonacci(n-1);
   }
}
```

4.8.2 How could you cache execution of any function?

4.8.2.1 Answer:

You could have a method where you will pass a function and it will internally maintain a cache object where calculated value will be cached. When you will call the function with same argument, the cached value will be served.

```
function cacheFn(fn) {
    var cache={};

    return function(arg){
        if (cache[arg]){
           return cache[arg];
        }
        else{
           cache[arg] = fn(arg);
            return cache[arg];
        }
    }
}
```

4.8.3 How could you set a prefix before everything you log? for example, if you log(*my message*) it will log: "(app) my message"?

4.8.3.1 Answer:

Just get the arguments, convert it to an array and unshift whatever prefix you want to set. Finally, use apply to pass all the arguments to console.

```
function log(){
  var args = Array.prototype.slice.call(arguments);
  args.unshift('(app)');
  console.log.apply(console, args);
}

log('my message'); //(app) my message
log('my message', 'your message'); //(app) my message your message
```

Chapter 5

Rapid Fire - Tricky Questions

5.1 Is *false* false?

No. Because it's a string with length greater than 0. Only empty strings can be considered as false.

5.2 Is ' ' false?

No. Because it's not an empty string. There is a white space in it.

5.3 Is Boolean(function(){}) true or false?

`true`. If you pass a truthy value to Boolean, it will be true.

5.4 What is 2 + true?

3 - The plus operator between a number and a boolean or two boolean will convert boolean to number. Hence, true converts to 1 and you get the result of 2+1.

5.5 What is *6*+9?

69 - If one of the operands of the plus (+) operator is string it will convert the other number or boolean to string and perform a concatenation. For the same reason, "2"+true will return "2true".

5.6 What is the value of +*dude*?

NaN. The plus (+) operator in front of a string is an unary operator that will try to convert the string to number. Here, JavaScript will fail to convert the "dude" to a number and will produce NaN.

5.7 If you have var y = 1, x = y = typeof x; What is the value of x?

"undefined".

5.8 If var a = (2, 3, 5); what is the value of a?

5 - The comma operator evaluates each of its operands (from left to right) and returns the value of the last operand.

5.9 What is -5%2?

-1. the result of remainder always gets the symbol of the first operand

5.10 What is typeof arguments?

Object. Arguments are like arrays but not arrays. They have length, can be accessed by index but you can't push, pop and so on.

Chapter 6

Essential Questions

6.1 What is the drawback of creating true private methods in JavaScript?

One of the drawbacks of creating true private methods in JavaScript is that they are very memory-inefficient, as a new copy of the method would be created for each instance.

```
var Employee = function (name, company, salary) {
    this.name = name || "";       //Public attribute default value is null
    this.company = company || ""; //Public attribute default value is null
    this.salary = salary || 5000; //Public attribute default value is null

    // Private method
    var increaseSalary = function () {
        this.salary = this.salary + 1000;
    };

    // Public method
    this.dispalyIncreasedSalary = function() {
        increaseSlary();
        console.log(this.salary);
    };
};

// Create Employee class object
var emp1 = new Employee("John","Pluto",3000);
// Create Employee class object
var emp2 = new Employee("Merry","Pluto",2000);
// Create Employee class object
var emp3 = new Employee("Ren","Pluto",2500);
```

Here each instance variable `emp1`, `emp2`, `emp3` has its own copy of the increaseSalary private method. So, we don't use private methods unless it's necessary.

6.2 What will be the output of the code below?

```
var y = 1;
  if (function f(){}) {
    y += typeof f;
  }
  console.log(y);
```

The output would be `1undefined`. The `if` condition statement evaluates using `eval`, so `eval(function f(){})` returns `function f(){}` (which is true). Therefore, inside the if statement, executing `typeof f` returns `undefined` because the if statement code executes at run time, and the statement inside the if condition is evaluated during run time.

6.3 What is a "closure" in JavaScript? Provide an example.

A closure is a function defined inside another function (called the parent function), and has access to variables that are declared and defined in the parent function scope.

The closure has access to variables in three scopes:

Variables declared in their own scope Variables declared in a parent function scope Variables declared in the global namespace

```
var globalVar = "abc";

// Parent self invoking function
(function outerFunction (outerArg) { // begin of scope outerFunction
    // Variable declared in outerFunction function scope
    var outerFuncVar = 'x';
    // Closure self-invoking function
    (function innerFunction (innerArg) { // begin of scope innerFunction
        // variable declared in innerFunction function scope
        var innerFuncVar = "y";
        console.log(
            "outerArg = " + outerArg + "\n" +
            "outerFuncVar = " + outerFuncVar + "\n" +
            "innerArg = " + innerArg + "\n" +
            "innerFuncVar = " + innerFuncVar + "\n" +
            "globalVar = " + globalVar);

    }// end of scope innerFunction)(5); // Pass 5 as parameter
}// end of scope outerFunction )(7); // Pass 7 as parameter
```

`innerFunction` is closure that is defined inside `outerFunction` and has access to all variables declared and defined in the `outerFunction` scope. In addition, the function defined inside another function as a closure will have access to variables declared in the `global namespace`.

Thus, the output of the code above would be:

```
outerArg = 7
outerFuncVar = x
innerArg = 5
innerFuncVar = y
globalVar = abc
```

6.4 Write a mul function which will produce the following outputs when invoked:

`javascript console.log(mul(2)(3)(4));//output :24 console.log(mul(4)(3)(4));//output : 48`

Below is the answer followed by an explanation to how it works:

```
function mul (x) {
    return function (y) { // anonymous function
        return function (z) { // anonymous function
            return x * y * z;
        };
    };
}
```

Here the `mul` function accepts the first argument and returns an anonymous function, which takes the second parameter and returns another anonymous function that will take the third parameter and return the multiplication of the arguments that have been passed.

In JavaScript, a function defined inside another one has access to the outer function's variables. Therefore, a function is a first-class object that can be returned by other functions as well and be passed as an argument in another function.

A function is an instance of the Object type A function can have properties and has a link back to its constructor method A function can be stored as a variable A function can be passed as a parameter to another function A function can be returned from another function

6.5 How to empty an array in JavaScript?

For instance,

```
var arrayList =  ['a','b','c','d','e','f'];
```

6.5.1 One Solution

A solution would obviously be setting the array length to zero.

```
arrayList.length = 0;
```

The code above will clear the existing array by setting its length to 0. This way of emptying the array also updates all the reference variables that point to the original array.

6.6 What will be the output of the following code?

```
var output = (function(x){
    delete x;
    return x;
  })(0);

  console.log(output);
```

The output would be `0`. The `delete` operator is used to delete properties from an object. Here `x` is not an object but a local variable. `delete` operators don't affect local variables.

6.7 What is the difference between the function declarations below?

```
var foo = function(){
    // Some code
 };
```

```
function bar(){
    // Some code
 };
```

The main difference is the function foo is defined at `run-time` whereas function `bar` is defined at parse time.

6.8 What will be the output of the code below?

```
var Employee = {
  company: 'xyz'
}
var emp1 = Object.create(Employee);
delete emp1.company
console.log(emp1.company);
```

The output would be `xyz`. Here, `emp1` object has `company` as its prototype property. The `delete` operator doesn't delete prototype property.

`emp1` object doesn't have **company** as its own property. You can test it `console.log(emp1.hasOwnProperty('company'));//output :false`. However, we can delete the `company` property directly from the`Employee` object using `delete Employee.company`. Or, we can also delete the `emp1` object using the `__proto__` property `delete emp1.__proto__.company`.

6.9 Consider the following code:

```
(function() {
   var a = b = 5;
})();

console.log(b);
```

What will be printed on the console?

The code above prints `5`.

The trick of this question is that in the IIFE there are two assignments but the variable `a` is declared using the keyword `var`. What this means is that `a` is a local variable of the function. On the contrary, `b` is assigned to the global scope.

The other trick of this question is that it doesn't use strict mode (`'use strict';`) inside the function. If strict mode was enabled, the code would raise the error Uncaught ReferenceError: b is not defined. Remember that strict mode requires you to explicitly reference to the global scope if this was the intended behavior. So, you should write:

```
(function() {
   'use strict';
   var a = window.b = 5;
})();

console.log(b);
```

6.10 Hoisting

6.10.1 What's the result of executing this code and why?

```
function test() {
   console.log(a);
   console.log(foo());

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

test();
```

The result of this code is `undefined` and `2`.

The reason is that both variables and functions are hoisted (moved at the top of the function) but variables don't retain any assigned value. So, at the time the variable `a` is printed, it exists in the function (it's declared) but it's still `undefined`. Stated in other words, the code above is equivalent to the following:

```
function test() {
   var a;
   function foo() {
      return 2;
   }

   console.log(a);
   console.log(foo());

   a = 1;
}

test();
```

Chapter 7

General Questions

7.1 What is ‘this’ keyword in JavaScript?

this keyword is used to point at the current object in the code. For instance: If the code is presently at an object created by the help of the ‘new’ keyword, then ‘this’ keyword will point to the object being created.

7.2 What is the difference between ViewState and SessionState?

ViewState is specific to a page in a session. **SessionState** is specific to user specific data that can be accessed across all pages in the web application.

7.3 Does JavaScript support automatic type conversion?

Yes JavaScript does support automatic type conversion, it is the common way of type conversion used by JavaScript developers.

7.4 How can you read and write a file using JavaScript?

There are two ways to read and write a file using JavaScript

- Using JavaScript extensions
- Using a web page and Active X objects

7.5 How can you read and write a file using JavaScript?

The `parseInt()` function is used to convert numbers between different bases. `parseInt()` takes the string to be converted as its first parameter, and the second parameter is the base of the given string.

In order to convert 4F (of base 16) to integer, the code used will be -

```
parseInt ("4F", 16);
```

7.6 What is the use of Void(0)?

`Void(0)` is used to prevent the page from refreshing and parameter "zero" is passed while calling. `Void(0)` is used to call another method without refreshing the page.

7.7 What are JavaScript Cookies?

A function that is declared without any named identifier is known as an anonymous function. In general, an anonymous function is inaccessible after its declaration.

Anonymous function declaration:

```
var wcg = function() {

alert('I am anonymous');

};

wcg();
```

7.8 How can a particular frame be targeted, from a hyperlink, in JavaScript?

This can be done by including the name of the required frame in the hyperlink using the 'target' attribute.

```
<a href="newpage.htm" target="newframe">New Page</a>
```

7.9 Explain the role of deferred scripts in JavaScript?

By default, the parsing of the HTML code, during page loading, is paused until the script has not stopped executing. It means, if the server is slow or the script is particularly heavy, then the webpage is displayed with a delay. While using Deferred, scripts delays execution of the script till the time HTML parser is running. This reduces the loading time of web pages and they get displayed faster.

7.10 How can a particular frame be targeted, from a hyperlink, in JavaScript?

`innerHTML` content is refreshed every time and thus is slower. There is no scope for validation in innerHTML and, therefore, it is easier to insert rouge code in the document and, thus, make the web page unstable.

7.11 What does a timer do and how would you implement one?

Setting timers allows you to execute your code at predefined times or intervals. This can be achieved through two main methods: `setInterval();` and `setTimeout();` `setInterval()` accepts a function and a specified number of milliseconds. ex) `setInterval(function(){alert("Hello, World!"),10000)` will alert the "Hello, World!" function every 10 seconds. `setTimeout()` also accepts a function, followed by milliseconds. `setTimeout()` will only execute the function once after the specified amount of time, and will not reoccur in intervals.

7.12 Explain the concept of unobtrusive Javascript?

Unobtrusive JavaScript is basically a JavaScript methodology that seeks to overcome browser inconsistencies by separating page functionality from structure. The basic premise of unobtrusive JavaScript is that page functionality should be maintained even when JavaScript is unavailable on a user's browser.

7.13 What is "event delegation" and how does it work?

Event delegation makes use of two often overlooked features of JavaScript events: **event bubbling** and the **target element**. When an event is triggered on an element, for example a mouse click on a button, the same event is also triggered on all of that element's ancestors. This process is known as event bubbling; the event bubbles up from the originating element to the top of the DOM tree. The target element of any event is the originating element, the button in our example, and is stored in a property of the event object. Using event delegation it's possible to add an event handler to an element, wait for an event to bubble up from a child element and easily determine from which element the event originated.

7.14 What is the importance of tag?

- JavaScript is used inside <script> tag in HTML document. The tags that are provided the necessary information like alert to the browser for the program to begin interpreting all the text between the tags.
- The <script> tag uses JavaScript interpreter to handle the libraries that are written or the code of the program.
- JavaScript is a case sensitive language and the tags are used to tell the browser that if it is JavaScript enabled to use the text written in between the <script> and </script> tags.

7.15 What is the difference between window.onload and onDocumentReady?

The `window.onload` event won't trigger until every single element on the page has been fully loaded, including images and CSS. The downside to this is it might take a while before any code is actually executed. You can use `onDocumentReady` to execute code as soon as the DOM is loaded instead.

Chapter 8

A Final Note

I hope this list serves you as a refresher for what you already know about Javascript. Aside from being able to discuss the topics listed above, you should also be prepared to create whiteboard examples for each one. If you're a pro and use JavaScript everyday, you'll breeze through these.

However, what remains of a great importance is having the concentration to listen carefully and use your knowledge/experience alongside with logic to answer any question. The idea of the interview sometimes is not the actual programming language, but to identify if the candidate is able to solve real life problems on the spot.