Slay the **JavaScript** Interview.

100 answers that every developer needs to know.

Slay the JavaScript Interview

100 answers that every developer needs to know

JSTips

Contents

ASIN: B08K9B5675

Cover design by: Maxi Albella

Introduction

Interviewing for a job in tech can be an intimidating process. You might have years of experience or just finished a bootcamp looking for your first job. In either case, to prepare for any question that may be asked is how you prevent being caught off-guard.

To help you get ready, I gathered the 100 answers to the most popular and most crucial questions to ensure that you do a terrific technical interview. The most critical step to landing a job is being properly prepared for the interview, and this book is here to help you through the process.

Each page contains a short, clear, no fluff sentence to give you maximum understanding of the concepts. Some questions are reinforced with a short code snippet, and others with a link to my favorite books and articles.

The purpose of this book is to share lessons and observations I have picked up from +30 interviews with TOP tech companies and through writing to +5 million readers on JSTips.co.

I hope you enjoy it and wish you success. Thanks for reading.

Joel

JavaScript

What exactly is JavaScript?

JavaScript is an interpreted, object-oriented, lightweight language that is used most frequently as part of web pages and applications, and that nowadays, you can find it running in spaceships.

Further readings:

> Documentation - w3schools Documentation - MDN You Don't Know JS Book Series - Kyle Simpson

What is ECMAScript?

ECMAScript, or popularly abbreviated as ES, is a standardized scripting language specification. ECMAScript has several implementations, the most popular being JavaScript.

Further readings:

Documentation - ECMAScript® Language Specification

What is ES6?

ES6 represented version 6 of the ECMAScript language, a significant step forward for a smoother process of large scale development. ES6 made substantial additions and sugars for patterns that drastically reduced the need for extensive boilerplates.

Further readings:

> Documentation - w3schools Documentation - ECMAScript 6: Feature Overview & Comparison

What is the V8 engine which powers JavaScript?

V8 is a JavaScript engine built for Google Chrome and chromium-based web browsers. V8 achieves high performance by implementing an in-memory-only architecture enabled by the design of the compiler and runtime of the language.

Further readings:

> Official Site - v8.dev Course - The V8 JavaScript Engine Guide - V8 Runtime Overview Documentation - WebAssembly Article - Just-In-Time Compiler

What makes JavaScript a dynamic language? Is it strongly or weakly typed?

JavaScript is dynamic and weakly typed since a variable is assigned a type, and the type is determined during runtime by looking at the first value assigned to it.

```
// Declaring a variable as a string
let thisIsAString = "I am a string";
console.log(typeof thisIsAString);
// Output: string

// Changing the type to number
thisIsAString = 1;
console.log(typeof thisIsAString);
// Output: number
```

How is JavaScript processed: compiled or interpreted?

JavaScript is an interpreted language that compiles the script to executable bytecodes with the assistance of JIT (Just-In-Time) compilation. This means the engine makes full use of the power of both interpreter and compiler to translate the JavaScript code on the browser at runtime.

Further readings:

> Article - A crash course in just-in-time (JIT) compilers - Mozilla Talk - JavaScript Engines - Franziska Hinkelmann

Is JavaScript case-sensitive?

JavaScript is a case-sensitive language. As a result, keywords, variables, and function names must be written using appropriate letter capitalization to avoid errors during code execution.

```
let name = "JS Tips";
console.log(Name);
// Output: ReferenceError: Name is not defined
```

How does the Garbage Collector work?

Mark-and-sweep is used in JavaScript as its Garbage Collector to decide whether a piece of code requires memory or not. The Garbage Collector first marks all roots as active, and then any children that can be reached from a root are kept as active. Memory not linked to the roots is considered garbage and will be collected and freed.

Further readings:

> Documentation - Memory Management - Mozilla Guide - Garbage collection - javascript.info

How does memory leak occur?

Memory leaks occur when some vulnerable piece of code fails to let go of an allocated part of memory that is no longer needed, causing problems like a slow execution, program crashing, and high latency. Some common causes of memory leaks include global variables, multiple references between objects, closures, and use of timers.

Further readings:

> Article - 4 Types of Memory Leaks in JavaScript and How to Get Rid Of Them - Auth0

Why is JavaScript a single threaded language?

JavaScript is a single thread language as it processes the code in sequence and finishes one operation before moving to the next. JavaScript Engines have one memory heap and call stack to emulate multi threading.

Further readings:

Article - Concurrency model and the event loop - Mozilla

What do Service Workers do?

A service worker is a JavaScript file that executes in the background to assist offline-first web applications, push notifications, and background syncs. It runs in parallel to the main application on a different thread and cannot interact directly with the webpage, but it still communicates with messages.

Further readings:

> Article - Service Workers: an Introduction - Google Dev Article - How JavaScript works - Session Stack

What is a PWA?

PWAs utilize service workers to build a user-friendly web app that can access native functionality of mobile devices. A Progressive Web App is akin to a traditional web application, except it is more robust, efficient, and ensures that it will work properly even when offline. Furthermore, PWAs enable web apps to load faster since they work in the background, and the end user is given a more seamless experience.

Further readings:

> Article - What are Progressive Web Apps? - WebDev Talk - JSConf:Modern PWA Cheat Sheet - Maximiliano Firtman

Values & Types

What are JavaScript's data types?

JavaScript is a dynamic language that doesn't require types to be explicitly declared. Variables do not have to be explicitly associated with any particular data types, and any variable can be assigned and then reassigned values of all types.

ECMAScript defines nine data types divided into two categories, primitive and non-primitive. Primitive data types represent a single value and are immutable, including `undefined`, `null`, `Boolean`, `Number`, `String`, `BigInt`, and `Symbol`. The `Object` type is the only non-primitive data being. arrays and functions are also treated as objects. The nine data type `null` is just a special value representative of "nothing".

```
// Primitives
console.log(typeof undefined); // "undefined"
console.log(typeof null); // "null"
console.log(typeof true); // "boolean"
console.log(typeof 0); // "number"
console.log(typeof 10n); // "bigint"
console.log(typeof "foo"); // "string"
console.log(typeof Symbol("id")); // "symbol"

// Non-Primitive
console.log(typeof {}); // "object"
console.log(typeof null); // "object"
console.log(typeof alert); // "function"
```

What does the typeof operator do?

`typeof` is a unary operator placed before an operand to indicate the type of value it evaluates to. It returns a string representing the type of value.

```
console.log(typeof true); // "boolean"
console.log(typeof false); // "boolean"

console.log(typeof 3000); // "number"
console.log(typeof 3.14); // "number"
console.log(typeof NaN); // "number"
console.log(typeof Infinity); // "number"

console.log(typeof "foobar"); // "string"
console.log(typeof `foobar`); // "string"

console.log(typeof {}); // "object"
```

How do instanceof and typeof differ?

The `typeof` operator gets the type of a data type value, and `instanceof` determines whether an object belongs to a particular constructor function.

```
function City() {};

const buenosAires = new City();

console.log(typeof buenosAires);
// Output: object

console.log(buenosAires instanceof City);
// Output: true

console.log(buenosAires instanceof Object);
// Output: true
```

What is JSON?

JSON is an open file format that stores text in name-value pairs and array data types. A JSON text will always have an extension of `.json`. It is used when you want to transfer files across networks, and it is useful for human reading.

```
{
    "url": "https://www.jstips.co",
    "isCool": true,
}
```

What is Type Coercion in JavaScript?

A process called type coercion is performed when applying operators to different types' values. It is possible since JavaScript is a weakly-typed language, which "coerces" one value type to another value type depending on what types are being applied for the conversion.

```
console.log("11" + 5); // 115
console.log(true + false); // 1
console.log(11 + 5 + "number"); // "16number"
console.log(true == 1); // true
```

What is the difference between == and === operators?

== operator or type coercion equality, automatically alters values to `string` before comparison, in contrast to === operator or strict equality operator, which follows strict comparison based on the data type of given variables.

```
console.log(0 == false); // true
console.log(0 === false); // false
console.log(1 == "1"); // true
console.log(1 === "1"); // false
console.log(null == undefined); // true
console.log(null === undefined); // false
```

What is undefined value?

`undefined` in JavaScript denotes that a variable has been reserved, but the value has not yet been assigned. Therefore, `undefined` is a variable in the global scope that cannot be overridden. Moreover, `undefined` is a valid data type.

```
// Declaring a variable
let firstName;
// Declaring and assign a value to a variable
let lastName = null;

console.log(firstName); // Output: undefined
console.log(lastName); // Output: null

console.log(typeof firstName); // Output: "undefined"
console.log(typeof lastName); // Output: "object"
```

What is null value?

The `null` value is a JavaScript's primitive value and indicates an intentional absence of value to show that the variable has been declared but not assigned with a value.

```
// Declaring a variable and assign no value
var foo = null;

console.log(foo); // null
console.log(typeof foo); // "object"
```

What are the differences between null and undefined?

`null` represents an empty value that does not exist, whereas `undefined` represents a variable that has not been assigned yet. Second, `null` is an `object` whereas `undefined` is not. When a check for equality == to `null` and `undefined` is done, it shows `true` due to type-coercion, but the two values previously were not the same.

```
let a = null;
let b;

console.log(typeof a);
// Output: "object"
console.log(typeof b);
// Output: "undefined"

console.log(a == b) // true
console.log(a === b) // false
console.log(null == undefined); // true
console.log(null === undefined); // false
```

What is NaN, and why is it used?

`NaN` stands for "Not a Number". It is a value that indicates a number format issue between two computations or an error in certain operations. It is assigned to the result of an operation that can't be completed due to an error.

```
console.log(1 * undefined);
// Output: NaN

console.log(parseInt("I'm a number!"));
// Output: NaN
```

What is Number.isNaN() method?

The `Number.isNaN()` method determines whether the passed value is an illegal number or not. If the value is does not count as a number, then `Number.isNaN()` returns `true`. Otherwise, it returns `false`.

```
console.log(Number.isNaN(123)); // false
console.log(Number.isNaN(-1.23)); // false
console.log(Number.isNaN(5-2)); // false
console.log(Number.isNaN(0)); // false
console.log(Number.isNaN("123")); // false
console.log(Number.isNaN("Hello")); // true
console.log(Number.isNaN("2005/12/12")); // true
console.log(Number.isNaN("")); // false
console.log(Number.isNaN(true)); // false
console.log(Number.isNaN(undefined)); // true
console.log(Number.isNaN("NaN")); // true
console.log(Number.isNaN(NaN)); // true
console.log(Number.isNaN(0 / 0)); // true
console.log(Number.isNaN(null)); // false
```

What is a void operator?

In JavaScript, the keyword `void` can be used to explicitly specify an expression to be evaluated without returning a value. It is commonly used in client-side JavaScript, where the browser should not render the value.

```
function getYear() {
  return 2020;
};

console.log(getYear());
// Output: 2020

console.log(void getYear());
// Output: undefined

// Useful use case
button.onclick = () => void getYear();
```

What is the difference between Map and WeakMap?

`WeakMap` is a type of `Map` that prevents a memory leak by holding the keys' reference objects weakly. If the object is deleted, then the `WeakMap` will release the object and all the references it holds.

Further readings:

> Article - Difference between Map and WeakMap in JavaScript - Matt Zenuert

What is a Regular Expression?

A regular expression(shortened as regex) is a function that is used to create comparisons between character strings, providing a high degree of flexibility when searching and replacing. This is a fundamental building block used to construct search and replace patterns in programs written in JavaScript.

```
console.log(/abc/.test("abcde"));
// Output: true
console.log(/abc/.test("abxde"));
// Output: false

// Splitting commas or spaces into an array
console.log("one, two three".split(/(?:,| )+/));
// Output: ["one", "two", "three"]
```

Further readings:

Article - Regular Expressions - Eloquent JavaScript

What is the purpose of JavaScript symbols?

A `Symbol` is a primitive data type in JavaScript that represents a unique identifier. A `Symbol` can be particularly useful when used as a property of an object. Once a symbol is created, its value is kept private, and for JavaScript's internal use.

```
let id1 = Symbol("id");
let id2 = Symbol("id");

console.log(id1 == id2); // false

let ids = {
  [id1]: 1,
  [id2]: 2,
};

console.log(ids["id"]); // Output: undefined
console.log(ids[id1]); // Output: 1
```

What is the JavaScript ternary operator?

The ternary operator is a shortcut for the `if` statement. It consists of three operands; a question mark, a condition, and an expression to execute if the condition is true, followed by a colon and another expression to execute if it's false.

```
let age = 26;

// condition ? expression if true : expression if false
let drink = (age >= 21) ? "Beer" : "Juice";

console.log(drink); // "Beer"

// Equivalent to:
let drink;
if ((age >= 21)) {
    drink = "Beer";
} else {
    drink = "Juice";
}

console.log(drink); // "Beer"
```

What is the rest parameter?

The rest parameter in JavaScript enables us to pass a large number of parameters to a function that we will then be able to access through an array.

```
function sum(...args) {
  return args.reduce((ac, cu) => ac + cu);
};

console.log(sum(2, 4, 6, 8));
// Output: 20
```

What is the spread operator?

In JavaScript, the spread operator refers to the use of an ellipsis of three dot to add elements to an array, combining arrays or objects into one larger one and spreading an array inside the arguments of a function.

```
// Concatenating arrays and objects
let arr1 = [1,2,3];
let arr2 = [4,5];
let newArray = [...arr1,...arr2];
console.log(newArray);
// Output: [ 1, 2, 3, 4, 5 ]

// Copying array elements
let arr = ["a","b","c"];
let newArray = [...arr];
console.log(newArray);
// Output: ["a", "b", "c"]

// Expanding arrays
let arr = ["a","b"];
let newArray = [...arr,"c","d"];
console.log(newArray);
// Output: ["a", "b", "c", "d"]

// Merging objects
const userBasic = {
    name: "Jen",
    age: 22,
};
const userMoreInfo = {
    country: "Argentina",
    city: "Córdoba",
};
const user = {... userBasic, ... userMoreInfo};
// Output: {  name: "Jen",  age: 22, country: "Argentina", city: "Córdoba" }
```

What are the differences between the rest and spread operators?

The syntax is the same, but its overall purpose alters its name. The spread operator allows us to add items to an array, merge arrays or objects, and spread one iterable into a function's arguments. The rest parameters let us pass numerous arguments to a function and access them as an array.

```
// Using rest to access unlimited numbers of parameters
const numbers = (...args) => {
  // Using spread to merge the items in an array
  return [0, ...args];
};

console.log(numbers(1, 2, 3, 4, 5, 6));
// Output: [0, 1, 2, 3, 4, 5, 6]
```

Variables

Are const variables immutable?

Variables declared using `const` are not immutable. `const` means that the variable will always reference the same object, the reference is immutable but not the value.

```
const number = 11;
const cat = {
  name: "Puchi"
};

number = 9;
// Output: TypeError: invalid assignment to const "number"

console.log(number);
// Output: 11

// You can override object properties
cat.name = "Rufi";
console.log(cat);
// Output: { name: "Rufi" }
```

What is Hoisting?

Hoisting is a JavaScript feature that move the scope of variables and function declarations to the top of the code before code executes.

```
console.log(num);
// Returns undefined, as only declaration was hoisted, no initialization has happened at th

var num; // Declaration
num = 6; // Initialization
```

What are the primary differences between `let` and `var`?

The main difference lies in scope, `let` is accessible only inside the block where it is declared, whereas `var` will be accessible globally. Moreover, variables declared using `var` and `let` will be hoisted, but variables declared using `let` will not be initialized.

```
// Creating global scoped variables
var foo = "Foo";
let bar = "Bar";
console.log(window.foo); // Foo
console.log(window.bar); // undefined

// Block scope rules
{
    var block_foo = "Foo";
    let block_bar = "Bar";
}
console.log(block_foo);
// Output: Foo
console.log(block_bar);
// Output: block_bar is not defined

// Hoisting
console.log(myVar); // Output: undefined
var myVar = "My var";
console.log(myVar); // Output: My var

console.log(myLetVar); // Output: ReferenceError
let myLetVar = "My let var";
console.log(myLetVar); // Output: My let var
```

How `this` works in JavaScript?

The `this` object refers to the current execution context. The execution content(aka scope) binds a different value of `this` depending on how the function is called. In the global scope, `this` refers to the global object window. Whenever a function is called with the `new` operator, `this` is bound to the new object being created. Inside an object, the object before the dot is what the `this` keyword is bound to.

Further readings:

> Article - Understanding the “this” Keyword in JavaScript - Better Programming Article - Understanding “This” in JavaScript - CodeMentor

What is scope in JavaScript?

The scope of a variable refers to the lifetime and accessibility of variables within the code context and is generally of two types, global and local. A global variable is accessible outside the block where it is declared, whereas local variables will only be accessible within the block where it is declared.

```
// Initializing a global variable
var globalVariable = "I'm a global variable";

const testingScopes = function() {
  // Initializing a local variable only visible in this block
  let localVariable = "I'm a local variable";
  // Logging a global variable as functions has access to the outer scope
  console.log(globalVariable + " inside a function!");
};

console.log(globalVariable);
// Output: "I'm a global variable"
console.log(localVariable);
// Output: ReferenceError: localVariable is not defined
testingScopes();
// Output: "I'm a global variable inside a function!"
```

What are the differences between declaring and initializing variables?

Declaring a variable reserves the name in memory at the current scope whereas initializing a variable sets the value of the variable.

```
function scopeDemo() {
  // Declaring name variable
  let name;
  // Initializing name variable
  name = "Praful";
  // Declaring and Initializing age variable
  let age = 55;
};
```

What is the Temporal Dead Zone?

The temporal dead zone is the time between the creation of a variable's binding and its declaration. In ES6, variables using the `let` or `const` keywords raise a ReferenceError if accessed before their declaration (within their scope).

```
function myFunction(){
  console.log(sayHello);
  var sayHello = "Hello World!";
};
myFunction();
// Output: undefined

function myFunctionTwo() {
  console.log(sayBye);
  let sayBye = "Hello World!";
};
myFunctionTwo();
// Output: ReferenceError: greeting is not defined
```

Arrays

When to utilize `call`, `apply` or `bind`?

You can use `call` and `apply` to invoke a function immediately with custom context and parameters. `bind` returns a new function with the contexts passed (`this`). These are common tools typically used on the pre-ES6 times. Nowadays, `call` and `apply` have been almost totally replaced by the spread operator.

```
const player = {
  name: "Roger",
  lastName: "Fedrer"
};

const getPlayer = function(sport) {
  return `${this.name} ${this.lastName} plays ${sport}!`;
};

// Using bind to set a new context
const asTennisPlayer = getPlayer.bind(player);
console.log(asTennisPlayer("tennis"));
// Output: "Roger Federer plays tennis"

// Using call to set a new context
console.log(getPlayer.call(player, "baseball"));
// Output: "Roger Federer plays baseball"

// Using apply to set a new context
console.log(getPlayer.apply(player, ["soccer"]));
// Output: "Roger Federer plays soccer"
```

What is the use of the array `slice` method?

The `slice` method returns a selection of elements from an array as a new array while not modifying the original one. The method takes two arguments, start and end, which represent the range of the elements to be sorted.

```
const myArray = [1, 2, 3, 4, 5];

// Copy two items
const newArray = myArray.slice(1, 3);
console.log(newArray);
// Output: [2, 3]

// Copy an entire array to a new object
const copyArray = myArray.slice();
console.log(copyArray);
// Output: [1, 2, 3, 4, 5]
```

What is the use of the array `splice` method?

The `splice` method modifies an array by either removing, replacing, or adding elements to it. The first argument gives the array index for modified elements. The second argument is the number of elements to be removed. Additional succeeding arguments are added to the array.

```
const myIceCream = ["chocolate", "vanilla", "strawberry"];

// Insert "mint" at index 2
myIceCream.splice(2, 0, "mint");
console.log(myIceCream);
// Outup: ["chocolate", "vanilla", "mint", "strawberry"]

// Remove "vanilla" at index 1
myIceCream.splice(1, 1);
console.log(myIceCream);
// Output: ["chocolate", "mint", "strawberry"]

// Replace "strawberry" at index 2
myIceCream.splice(2, 1, "mango");
console.log(myIceCream);
// Output: ["chocolate", "mint", "mango"]
```

How is `slice` different from `splice`?

`slice` is used to pick elements from an array that you do not intend to alter. `splice` is normally used to actually change the original array by inserting and removing elements. Nowadays, `slice` is used to clone an array by calling it without arguments.

```
const food = ["burrito", "pasta", "noodles"];

// Using slice to pick elements form an array based on elements index
const myFood = food.slice(1, 2);
// myFood is ["pasta"]
// food is ["burrito", "pasta", "noodles"]

// Using splice to update the original array based on elements index
food.splice(1, 1, "burger");
// Output: ["pasta"]
// food is ["burrito", "burger", "noodles"]
```

Functions

What are functions in JavaScript?

Functions are pieces of reusable code; they can be called again and again by referencing their name. Some functions have no name, are defined "on-the-fly," and are known as anonymous.

```
// Declaring a function named as game
function sayHello(){
  // Content of the function
  console.log("Hello!");
};

// Assigning an anonymous function to a variable
const sayBye = () =>{
  // Content of the function
  console.log("Bye!");
};

// Declaring an anonymous function
() => console.log("How am I?");
```

What are arrow functions?

Arrow functions are shorthand notations for anonymous functions that simplify function scoping. They do not bind their own `this` but use the one from the nearest enclosing lexical scope. Arrow functions can't be utilized as constructors and are more efficient than typical functional expressions as they don't create a new scope.

```
// ES5
var multiplyES5 = function(x, y) {
  return x * y;
};

// ES6
const multiplyES6 = (x, y) => {
  return x * y;
};

const shorterMultiplyES6 = (x, y) => x * y;
```

What is an IIFE?

An IIFE (Immediately Invoked Function Expression) is a method to execute a function as soon as it is created. In addition, IIFE's do not populate global objects and therefore are a useful solution to isolate variable declarations.

```
(function() {
  // Code here is private
})();

(() => {
  // You can define it with arrow functions as well
}());
```

What's an anonymous function?

An anonymous function is different from a regular function because it has no name, is not accessible after its creation, and is not stored in the memory.

```
let anonymous = function() {
    console.log("Anonymous function");
};

let arrowAnonymous = () => console.log("Anonymous function");

// IIFE is an anonymous function as well
(function() {
    console.log("IIFE");
})();
```

How are arguments different from properties?

Arguments are values that are passed to a function when the function is invoked, whereas parameters are simply variables listed as part of the function definition.

```
// Function definition
const sum = (parameter1, parameter2) => {
  return parameter1 + parameter2;
};

// Invoking sum function with arguments 1 and 2
sum(1, 2);
```

What are closures in JavaScript?

Closures are a way of implementing context privacy by wrapping a function together with reference to its surrounding environment. Closure functions have access to the scope in which they are created and not the scope in which they are executed, and they are created when the function is first defined.

```
/*
 * Example of simple closure
 */
function parentFunction() {
  let outerVar = "I am in the parent scope!";
  function innerFunction() {
    // Variable outerVar is stored in the scope
    console.log(outerVar);
  }
  return innerFunction;
}

const myClosure = parentFunction();
myClosure();
// Output: I am in the parent scope!

/*
 * Example of useful closure
 */
function multiply(a) {
  return function innerMultiply(b) {
    // Variable a is store in the scope
    return a * b;
  }
}

const double = multiply(2);
double(3);
// Output: 6
double(8);
// Output: 16
```

When are function definitions not hoisted in JavaScript?

Assigning a function to a variable does not hoist its function definition.

```
console.log(foo());
// Output: Hello!

// Entire function hoisted
function foo() {
  return "Hello!";
};

console.log(foo2());
// Output: Uncaught TypeError: foo2 is not a function

// Only definition hoisted as foo2 is a variable
var foo2 = function() {
  return "Bye!";
};

console.log(foo2());
// Output: Bye!
```

What is a function constructor?

Functions become constructors when they are called with the `new` operator, used to create new objects.

```
// Using User function as a constructor
function User(name, lastname) {
  return {
    name: name,
    lastname: lastname
  };
};

const myUser = new User("Buzz", "Lightyear");
console.log(typeof myUser); // "Object"
```

What is the difference between a method and a function?

JavaScript functions and methods are essentially the same things. The only difference is that a method is a function that belongs to an object, and `this` refers to the object itself.

```
// Example of function declaration
function hello() {
  console.log("I'm a function");
};

const myObject = {
  name: "method",
  // Example of method declaration
  hello: function() { return "My name is" + this.name }
};
```

Objects

What is an `Object`?

Objects are the elementary components of JavaScript. An `Object` is a collection of properties, and a property is a relationship between a key and a value. Nearly all things in JavaScript are instances of `Object`, which is situated at the top of the inheritance hierarchy.

What are the options for creating objects in JavaScript?

Using object literals is just one of three ways of creating new objects. The next option is using the constructor or the `new` operator, using a function or class as a base, followed by using the `object.create()` method to set it up with its prototype.

```
// Object literal
const myFirstObject = {
  property: "nice"
};

// Using a function as constructor
const createMyObject = function() {
  this._property = "nice";

  return {
    property: this._property
  };
};
const mySecondObject = new createMyObject();

// Class syntax
class MyClass {
  property = "nice" ;
};
const myThirdObject = new MyClass();

// Using Object.create()
Object.create({ name: "Jstips" });
```

How does the `Object.create()` method work?

`Object.create()` method returns a new object with the specified prototype object and properties. The `Object.create()` method takes two parameters: an object that will be used as the new object's prototype, and an object with the new object's properties.

```
const car = {
  drive: () => "I need oil!"
};

const myCar = Object.create(car, {
  color: { value: "blue" }
});

console.log(myCar.color);
// Output: "blue"

console.log(myCar.drive());
// Output: "I need oil!"
```

What is the difference between Shallow and Deep copy?

A deep copy means that all values and sub-values of the variable are copied and the reference removed from the original variable; however, shallow copy means sub-values keep the reference to the original variable.

```
var car = { brand: "Tesla", model: "S", canFly: false };

// Shallow copy.
var myCar = car;

// Shallow copy using assign
let myCar = Object.assign({}, car);

// Deep copy.
let myCar = JSON.parse(JSON.stringify(car))
```

What are the benefits of Getters and Setters?

In JavaScript, getters and setters are used to access and set properties of an object. Getter methods are used to access an object's properties and act as a wrapper around the object's properties to gain more control over operations with them. Setter methods are used to change an object's values and check the data's validity or mutate it before setting a property.

```
let book = {
  _title: "Slay the JavaScript interview",
  _year: null,
  get fullInfo() {
    return `${this._title} - ${this._year}`;
  },
  set year(year) {
    this._year = year;
  }
};

book.year = 2020;
console.log(book.fullInfo);
// Output: Slay the JavaScript interview - 2020
```

How to prevent modification of objects?

There are three preventive methods. `Object.preventExtensions()` prevents the addition of any new properties. `Object.Seal()` prevents modifications and makes existing properties as non-configurable. Lastly, `Object.Freeze()` can prevent the modification of existing property, value, and attribute.

Is there a downside to having the methods set within the constructor?

In Javascript, when a method is bound to the `this` keyword, the method is provided to only that particular instance and doesn't have anything to do with the constructor. Also, you need to know that any attached method gets redefined on every new instance we create, which could impact our application's memory usage. You should use the prototyping approach to prevent this non-recommended pattern.

```
// Factory function pattern
function Car(name) {
  this.name = name;
  this.color = "blue";
  this.getColor = function() {
    return this.color;
  };
};

// Example
for (let i = 0; i < 1000; i++) {
  // getColor() is copied 1000 times in memory
  const myCar = new Car("Tesla" + i);
}
```

Inheritance

What is prototype based inheritance?

Prototype-based inheritance is an object oriented programming style wherein object properties and methods can be shared through objects that can be copied and extended by delegation that act as prototypes.

Further readings:

> Article - 3 Different Kinds of Prototypal Inheritance: ES6+ Edition - JavaScript Scene Article - Understanding Prototypes and Inheritance in JavaScript - Digital Ocean

What is the name of the relationship among objects within prototype inheritance?

On Prototype Inheritance, delegation is the name of the relationship between objects. It refers to a state in which one object transfers, or delegates, some of its responsibilities to another object of the same type.

What is the prototype chain?

In JavaScript, each object has a private property that links to another object known as the prototype and continues this chaining until the object has a `null` prototype. This concept is known as the prototype chain, and its core mechanism enables the inheritance system in JavaScript.

Further readings:

> Article - The prototype chain in depth - Debugger

What is the difference between `__proto__` and `prototype`?

The main difference is that `__proto__` is a property of a class instance, whereas a `prototype` is a property of a class constructor. Moreover, `__proto__` is the object used in the lookup chain to resolve methods and properties. `prototype` is the object used to build `__proto__`.

What is a class in JavaScript??

A JavaScript class is, essentially, a sugar-syntax for creating objects that emulate the classical inheritance model by using the prototypal inheritance.

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return this.name;
  }
}

let myCat = new Animal("Lex");
console.log(myCat.sayHi());
// Output: "Lex"

// Proof of prototypal inheritance: Animal is a function
console.log(typeof Animal);
// Output: "function"
```

Further readings:

> Article - How To Use Classes in JavaScript - DigitalOcean Article - The Complete Guide to JavaScript Classes - Dmitri Pavlutin

How do you extend a class?

The `extends` keyword is used in class declarations to create a subclass which inherits properties and methods from a parent class. This can be used to create custom classes as well as subclasses built-in objects.

```
class User {
  constructor(email) {
    this.email = email;
  }

  getEmail() {
    return this.email;
  }
};

class Customer extends User{
  constructor(email, plan) {
    super(email)
    this.plan = plan;
  }

  getFullInfo() {
    return this.getEmail() + " " + this.plan;
  }
};

const myCusmer = new Customer("hey@mail.com", "Pro");

console.log(myCusmer.getFullInfo());
// Output: hey@mail.com Pro
```

What are class fields?

JavaScript has the concept of Class Fields, also called class properties, which are variables that keep information and can offer private and public members.

```
class MyClass {
  a = 1; // public variable
  #b = 2; // private variable that cannot access outside of Myclass
};
```

Promises

What is a promise?

A promise is an object that wraps an asynchronous task and notifies when it's complete. A promise produces a single value as a result of an asynchronous operation, and its state can be either pending, fulfilled, or rejected.

```
const prepareDrink = (cant) => new Promise(resolve => {
  setTimeout(() => resolve("mojitos"), cant * 2000);
});

prepareDrink(2).then((result) => {
  console.log(`My ${result} are done`);
});

// Output: My mojitos are done.
```

What are the three states of a promise?

Promises in JavaScript represent results of asynchronous operations, which are of three types: pending, which is the initial state of a promise, fulfilled, representing a complete action, or rejected, which represents a failed action.

Further readings:

> Article - What is a Promise? - JavaScript Scene Documentation - Promise - Mozilla

What is promise chaining?

Promise chaining enables you to tell JavaScript what to do next after an asynchronous job has occurred. It allows you to assign the next task to be executed after an asynchronous job is complete through `Promise` methods like `then()`, `catch()`, or `finally()`.

```
const parse = () => {}; // your parse code
const add = () => {}; // your add code

myPromise().then(parse).then(add).then((result) => {
    console.log(result); // print result
});
```

What is the promise executor?

All `Promise` instances accept a method as an argument called the executor. This executor takes two methods as arguments: resolve and reject. Within the executor, if resolve is called, the `Promise` instance becomes fulfilled. If an exception is thrown, reject should be called instead, and the `Promise` instance becomes rejected.

```
const executor = (resolve, reject) => {
  setTimeout(() => resolve("I'm done"), 1000);
};

new Promise(executor).then(result => {
  console.log(result);
  // Output after 1000ms: I'm done
});
```

What is the function of `all()` method in the `Promise` object?

The `Promise.all()` method takes in an array of Promises and returns a single Promise that is resolved when all of the elements in the Promise array have resolved successfully.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, "one");
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "two");
});

Promise.all([p1, p2]).then(function(value) {
  console.log(value); // Output: ["one", "two"]
});
```

In a `Promise`, what does the `race()` method do?

`Promise.race()` in JavaScript returns a promise that is fulfilled or rejected after a promise in the passed array fulfills or rejects. If the iterable passed is empty, the promise will be pending indefinitely.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, "one");
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "two");
});

Promise.race([p1, p2]).then(function(value) {
  console.log(value); // Output: two
  // Both resolve, but p2 is faster
});
```

What is an async function?

Async functions allow writing promise-based code as if it were synchronous, but without halting the execution thread. It operates asynchronously via the event-loop. An async function can contain an `await` expression that pauses the execution and waits for the passed Promise's resolution.

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("done!"), 1000);
});

async function myDemo() {
  // await works only inside async functions
  let result = await promise; // wait until the promise resolves
  console.log(result);
};

myDemo();
// Output after 1000ms: "done!"
```

What are the advantages and disadvantages of using promises over callbacks?

The pros of using promises include increased readability, reduced coupling, and integrated error handling, as well as better definition and organisation of asynchronous logic flow. A disadvantage of promises is they can only return one object and not return multiple arguments at the same time. In addition to that, promises kill asynchronous non-blocking input-output.

The Event Loop

What's the event loop?

The event loop is the very core of how the JavaScript engine executes asynchronous programming. In an effort to maintain the illusion of a true multithreaded environment, the JavaScript execution engine uses a single thread to execute all operations via a few smart data structures called call stack and event queue.

Further readings:

> Article - What is an event loop in JavaScript? - Educative Article - The JavaScript Event Loop - Flavio Copes Article - JavaScript Event Loop Explained - Fronteed Weekly Talk - Event Loop - JSConf

What is the call stack in JavaScript?

A call stack is a mechanism for the interpreter to track its position during multiple function calls in the script. The record contains details of which function is currently being executed, what functions are called within that function, and so on.

Further readings:

Documentation - Call stack - Mozilla

How can JavaScript be run in a different thread?

JavaScript is single-threaded; to run it in different threads, you can use web workers and run a script on a parallel thread with the main one. Using web workers, you can do essential jobs on the main thread and let the web worker do computationally intensive tasks.

Further readings:

> Article - JavaScript Web Workers - SitePoint Documentation - Using Web Workers - Mozilla

What is the Message Queue and how does it work?

The Message Queue handles all asynchronous code in JavaScript. It is a queue wherein user-initiated events are laid down, such as clicks, ajax calls, and promises. The Message Queue processes all queued methods every time a user's call stack is empty.

What is the function of `setTimeout`?

The `setTimeout()` function allows call to a function after a predefined time period expressed in milliseconds. A separate timer thread in the browser counts down after calling `setTimeout` and when time elapses it adds the callback function to the browser's thread execution stack.

```
setTimeout(() => console.log("Hi there!"), 1500);
// Output after 1.5 sec: Hi there!
```

What is the use of `setInterval`?

The `setInterval()` method in JavaScript calls and executes a function or a block of code repeatedly after a predetermined time interval.

```
// Repeat with the interval of 5 seconds
let timerId = setInterval(() => console.log("Hi!"), 5000);
```

How are Debouncing vs Throttling techniques different?

The main difference is that throttling is used when you need to ensure that a function executes at a minimum given interval. In contrast, debouncing enforces that a function not be called again until a certain amount of time has passed without it being called and allows us to "group" multiple sequential calls in a single one.

Further readings:

> Article - Debouncing and Throttling - CSS Tricks Article - Debounce vs Throttle: Definitive Visual Guide - Redd

When should you use `requestAnimationFrame`?

`requestAnimationFrame` is a method used to animate objects that allow you to execute code on the next available screen repaint. Rendering happens on the browser's own time, so it's much smoother than typical animations implemented with `setInterval();`

```
const myObject = document.getElementById("#object-to-animate");

let position = 0;

requestAnimationFrame(() => {
  position += 10;
  myObject.style.top = position + "px";
});
```

What is the event flow?

An event flow represents the order in which events are received and handled on a web page, and is generally accomplished by event capturing, event sourcing, and event bubbling.

Further readings:

> Article - Event Flow - Tutorial Spark Guide - Bubbling and capturing - javascript.info

How does the event bubbling work?

Event bubbling occurs when an event starts at an element and then proceeds to bubble up towards its parent and ancestors. When an event is triggered in a nested block element, it begins at the deepest declared element and works its way to the top element.

Event Flow

Target -> Body -> HTML -> Document -> Window

```
<ul class="list">
    <li class="item">Eggs</li>
    <li class="item">Milk</li>
    <li class="item">Chocolate</li>
</ul>
```

```
const list = document.querySelector(".list");

list.addEventListener("click", event => {
  console.log(event.target.className);
}, false);
// Output: item
```

How does the event capturing work?

In the event capturing concept or in the top to bottom event flow process, an event trigger is transferred from the parent element to the innermost DOM element / children that are inside the same nested hierarchy.

Event Flow

Window -> Document -> HTML -> Body -> Target

```
<ul class="list">
    <li class="item">Eggs</li>
    <li class="item">Milk</li>
    <li class="item">Chocolate</li>
</ul>
```

```
const list = document.querySelector(".list");

list.addEventListener("click", event => {
  console.log(event.target.className);
}, true);
// Output: item
```

What is event delegation technique?

Event bubbling is a foundational concept behind event delegation in browsers. A single parent element can have an event handler bound to it so that the listener handles the event every time it occurs with any of its child nodes. The premise behind event delegation is that we should look at the page for elements that will be there when the page initially displays rather than listening to the elements themselves. Whenever you need to handle events on multiple elements, this technique is fast and reliable.

```
<div id="buttons">
  <button class="buttonClass">Click me</button>
  <button class="buttonClass">Click me</button>
  <button class="buttonClass">Click me</button>
</div>
```

```
// Attaches the listener the the parent element
document.getElementById("buttons")
    .addEventListener("click", event => {
        if (event.target.className === "buttonClass") {
            console.log("Click!");
        }
    });
```

Further readings:

Article - A Simple Explanation of Event Delegation - Dmitri Pavlutin

How does the `preventDefault` method work?

`preventDefault` is used to suspend the default browser actions of a certain event without stopping the event completely from bubbling up the DOM. `preventDefault` stops event actions from occurring if those actions are cancellable, meaning that the default action the event includes will not occur.

```
<input type="checkbox" class="checkbox">
```

```
document
  .querySelector(".checkbox")
  .addEventListener("click", (event) => {
    event.preventDefault();
    // The browser prevents from checking the box.
  });
```

How `stopPropagation` works?

The `stopPropagation` method is used to stop the event bubbling from the current element and prevent the parent event handlers from being executed.

```
<div id="parent">
  <button id="child">Hey!</button>
</div>

// Adding a listener to the button
document
  .querySelector("#child")
  .addEventListener("click", event => {
    // Stopping propagation
    event.stopPropagation();
    });

document
  .querySelector("#parent")
  .addEventListener("click", event => {
    // Alert is never fired as propagation was stopped
    alert("Parent event fired!");
    });
```

What's the difference between Target and currentTarget in the event context?

The `target` refers to the DOM element that triggers an event. Otherwise, `currentTarget` indicates the DOM element that is being listened to.

```
<ul class="todo-list">
  <li class="item">Walk your dog</li>
</ul>
```

```
const list = document.querySelector(".todo-list");

list.addEventListener("click", e => {
  console.log(e.target);
  // Output: <li class="item">Walk your dog</li>
  console.log(e.currentTarget);
  // Output: <ul class="todo-list"></ul>
});
```

Storage

What is a cookie?

A cookie is a string stored on the web browser used to persist only a limited bits of information. JavaScript can access these by using the `document.cookie` property, which provides interfaces for creating, retrieving, and deleting them.

```
// Setting a cookie
document.cookie = "userId=Mike216";

// Reading all cookies
const cookies = document.cookie;

// Updating a cookie
document.cookie = "userId=newId";

// Deleting a cookie removing the value
document.cookie = "userId=";
```

What are the differences between cookies and local storage?

Cookies are used for client-side and server-side purposes to store relatively few pieces of information. Unlike them, `localStorage` is only accessed on the client-side and can hold larger amounts of information. Cookies can only hold strings, but you can save all kinds of data as a JSON in localStorage. Moreover, `localStorage` data does not expire, whereas cookie data can be set to expire after a particular period.

What makes sessionStorage different from localStorage?

The difference between `localStorage` and `sessionStorage` is that while both do not expire, data in `sessionStorage` is not shared between tabs, and it's cleared as a page session ends, like when you close the page.

What is IndexedDB?

IndexedDB is an alternative to store data on the user's browser. It's more powerful than localStorage since it allows you to perform complex queries regardless of network availability, and your app continues to function whether the network is online or not.

Further readings:

> Guide - Working with IndexedDB - Google Dev Article - A quick but complete guide to IndexedDB and storing data in browsers - Freecodecamp

Functional Programing

What is functional programming about?

Functional programming is an approach where functions are used to express what the program should do rather than how to do it(imperative way). Functional programming's key characteristics consist of pure functions, immutability, and preventing side-effects.

Further readings:

Book - Functional-Light JavaScript: Pragmatic, Balanced FP in JavaScript Book - Functional Programming in JavaScript

What is a pure function?

A pure function is a function that returns the same output every time the same input is given as an argument. Pure functions do not produce side effects and are the cornerstone of functional programming.

```
// Pure function example
const add = (x, y) => x + y;
add(2, 4);
// Output: 6
```

What is a thunk function?

A thunk function is a function in JavaScript that delays the evaluation of value. These functions do not take any arguments and generate values only when the thunk function is invoked.

```
// Example of thunk function
const sum = (x, y) => {
  return () => {
    return x + y;
  };
};

// You can store the return value in a variable.
const thunk = sum(5, 2);

// Or use it directly
console.log(sum(5, 2)());
// Output: 7

// You can also use it in other calculations.
const twice = thunk() * 2;
// Output: 14
```

What is memoization?

Memoization is a technique that allows a function to be optimized by ensuring that a function does not run continuously for the same input. This is done by storing previous results and using it while the same data are passed to a method. Memoization only works when working with pure functions.

Further readings:

> Article - How to use Memoize to cache JavaScript function results and speed up your code - Freecodecamp

What is a higher order function?

Functions are described as higher order if they take other functions as arguments or return other functions as values. Higher-order functions allow you to use composition to write simpler and more elegant code.

```
const multiplyBy = (x) => (y) => x * y;

let multiplyBy5 = multiplyBy(5);

console.log(multiplyBy5(10));
// Output: 50

console.log(multiplyBy5(2));
// Output: 10
```

What is a currying function?

A currying function takes multiple arguments and transforms it into sequential functions, each of which has only one argument. In this way, an n-ary function can be turned into a unary function, and the last function returns the results of all the arguments together in a single function.

```
// Normal definition
function multiply(a, b, c) {
  return a * b * c;
};
console.log(multiply(1, 2, 3));
// Output: 6

// Simple curry function definition
function multiply(a) {
  return (b) => {
    return (c) => {
      return a * b * c;
    };
  };
};
console.log(multiply(1)(2)(3));
// Output: 6
```

Further Reading:

> Currying in JavaScript Lodash curry JavaScript currying

What is Functional Inheritance?

The functional method does not implement a prototypical pattern. Inheritance is accomplished by creating an object by a base function, very much alike to a constructor, but without using `new` keyword. The function takes over the parent's closure scope, thereby keeping some properties private and extending the existing object with new properties and methods.

```
// Base object constructor function
function Animal(data) {
  var that = {}; // Create an empty object
  that.name = data.name; // Add it a "name" property
  return that; // Return the object
};

// Create a child object, inheriting from the base Animal
function Cat(data) {
  // Create the Animal object
  var that = Animal(data);
  // Extend base object
  that.sayHello = function() {
    return "Hello, I'm " + that.name;
  };
  return that;
};

// Usage
var myCat = Cat({ name: "Rufi" });
myCat.sayHello();
```

Acknowledgements

Special thanks to Leonardo Apiwan, Fran Caballero, and Jane Tracy for reviewing my drafts in great and granular detail. You've pushed the level of the book to a new and different level. To Maxi Albella, thank you for the best cover designer one could ever ask for. To the JavaScript community that helped me so much, thank you.

Slay the **JavaScript** Interview.

100 answers that every developer needs to know.