JavaScript Interview Questions

Who Else Wants to Nail that Interview?

by **Volkan Özçelik** • j4v4ScR1p7 h4X0r @ o2js.com

Dedicated to my wife Nagehan, our daughter Yaprak, and our son Toprak ,

whose loves, hugs, and smiles make every day the best day ever.

About the Author

Hi, I'm Volkan Özçelik: Jack of all Trades, Samurai of **JavaScript**.

Since **2003**, I've been doing front-end development on client-heavy web, desktop, and mobile applications.

The stuff I love to architect is a **responsive** and **intuitive** user interface, driven by amazingly well-organized **JavaScript**. I've experimented with most of the **JavaScript** frameworks around; written a handful myself. I have also fiddled with **NoSQL**, **ASP.net**, **C#**, **PHP**, **Java**, **Python**, **Django**, and some **Ruby**, and I keep coming back to the front-end. Sure, I can process a form on the server, reload the page and spit out a table in the glimpse of an eye, but where's the fun in that?!
java

My Timeline at a Glance

★ I am currently a **Technical Lead** at Cisco;

★ Before that I was a **Mobile Software Engineer** at Jive Software;

★ Before that I was a **JavaScript Hacker** at SocialWire;

★ Before that, I was a **VP of Technology** at GROU.PS;

★ Before that, I was a **JavaScript Engineer** at **LiveGO**, a social mash-up (gone to dead pool; **R.I.P.**);

★ Before, I was the **CTO** of Turkey's largest business network **cember.net** (got acquired by Xing A.G.; **R.I.P.**);

Other Places to Find Me

★ volkan.io

★ linkedin.com/in/volkanozcelik

★ github.com/v0lkan

Table of Contents

Preface

If you are like me, who skips prefaces and jump straight into the core of the subject, I wish you'll make an exception, because this one has some useful information. If you are still inclined to skip the preface here's the **bottom line up front**:

> If you just read this book and do nothing, you'll learn just a few things, but this will **not** even be **tangentially close** to what you would learn when you spend some time trying to work through the problems **on your own** before diving into the answers.

Therefore, I **highly suggest** you deeply research the ideas summarized in this book.

Take, for example, "*functional programming*". It may sound trivial when you simply read the definition of it; however there's an entire school of mathematics (called *Lambda Calculus*) behind it. Moreover, it requires significant effort and much practice to get used to thinking functionally, especially if you're from an object-oriented background.

Have you ever read the bibliography of a book? If not, starting with this book you'll be looking at a compilation of the most useful set of reference material and links **ever**.

> If you want to work in a rock star company, **be a rock star**:
> Do your homework: don't just skim, but **digest** this book.
> Browse any links cited; read through any supporting materials given.

In the last couple of years I have been to **at least a hundred** technical interviews on **JavaScript Engineering**, Front-End Development, and related positions. So, rather than giving you some HR Manager's perspective on what interviews should look like, I'm going to hand over the **red pill** and tell you what the **JavaScript Engineering** interviews **really are**, and what you will need to know to get the job you want.

I have made every effort to ensure that the information presented in this book is complete, coherent, and correct. All the code has been double tested, all the text has been proofread several times. However, mistakes, bugs, typos and errors are inevitable; it's human nature. If you find such problems please feel free to shoot me a mail.

You'll find this book a great way to get prepared for the entire interview process for a **JavaScript Engineering** position.

After having read this book, **JavaScript Engineering** interviews will not be a **black box** to you: That is, it won't be a secret process where there's the candidate (*i.e., you*) as the input; dark voodoo magic happening in between, and hopefully a job offer as an output.

I am confident that you'll find this book useful in getting the job you want. I also hope you will find it an entertaining exploration into the mysterious world of **JavaScript**. If you want to offer feedback on how you feel about this book, share your knowledge and thoughts on any particular topic or problem, or provide a problem from one of your recent interviews, I'd be more than happy to hear from you: Please email me at volkan@o2js.com.

I hope you'll enjoy reading this book as much as I enjoyed writing it.

Now, go ace that interview!

Good Luck!

V. Özçelik

Acknowledgements

Organizing and composing a book (*even if it's an ebook*) is not easy. It requires **hard work** from many people.

I'd like to thank **Bobby Rubio** for the wonderful cover illustration
(*I can't imagine an image that better fits the purpose of this book*);

I'd like to thank **Edward Schaefer**, **Brett Langdon** and **Melih Önvural** for their patience throughout the editing, proofreading, and copywriting process;

I'd like to thank **Ken Brumer**, for his grammatical corrections, code analyses, and improvement suggestions;

I'd like to thank **Marc Grabanski**, for giving me the opportunity to introduce **JavaScript Interview Questions** to the most targeted audience possible, because there is nothing more satisfying for a writer than knowing the fact that what he has crafted will be consumed by a brilliant audience who can get the most value out of it. Frontend Masters is a legendary community, and I'm honored to have the privilege of adding value to it;

I'd like to thank all the **geeklisters**, and **frontend masters** who have been with me along the way;

And last, but not the least; I'd like to thank my friends and my family, for having supported me throughout the process.

You have done a great deal of a job, and it would be impossible to create this book, to this level of quality, without you.

I wholeheartedly and truly appreciate your support.

Thank you!

Status of This Book

The book's current version is **`1.0.12`**, so it's **complete**, and stable.

If you see any errors, please e-mail volkan@o2js.com; I will be adding them to the book's errata.

This version of "**JavaScript Interview Questions**" was published on **Thursday, May 5, 2016**.

Who This Book Is For

I **highly recommend** you read this chapter.

It's not the usual "*this book is for these developers of that background...*" yadda yadda.

Before I get into who this book **is** for, I guess I should tell you who this book **isn't** for: This book **isn't** for those looking for quick remedies. Nor is this book a **magic wand** that will instantly transform you into a rockstar candidate. It is **not** for the lazy, or for the close-minded, or for those who think that the overall job interview process is a **"virtual reality"** to keep you out anyway. This book **isn't** for those who have their minds made up, or know it all. As you will see, this book is a **radical departure** from virtually all other interview books available.

This book **is** for people who live and breathe **JavaScript**. This book is for those who are **bold enough** to investigate the depth and breadth of **JavaScript**. Although the intended audience of this book is the **curious** people seeking **JavaScript Engineering** jobs (*and obviously this book is highly JavaScript-focused*), anyone who wants ideas on how to **ace** the technical and nontechnical parts of an **Engineering** job interview process can benefit from this book.

This book is **not** written to distinguish the good parts of **JavaScript** from the bad. Nor is it meant to be a definitive reference guide. It's **not** a cookbook containing **JavaScript** framework recipes either. Those books **have** already been written, and they have been written **well**.

In contrast, this book has **intentionally** been written to give the reader a framework-agnostic and **accurate** perspective on **JavaScript;** and then to teach the reader how to use this gained perspective in a JavaScript programming interview. To achieve this goal, it thoroughly analyzes actual real-life interview questions instead of simply providing a group of useless items to memorize. In essence, this book teaches the reader *how to think* **in JavaScript**.

If you are one of those who have only used **JavaScript** under the cloak of libraries (*i.e., jQuery, Prototype, etc.*), it's my hope that the material in this book will transform you from a library user into a **JavaScript** rockstar.

Libraries create a "**black box**" that can be beneficial in some regards but detrimental in others. Tasks may be completed fast and efficiently, but you have no idea how or why, which will have their consequences such as memory leaks, and poor performance; and the **"how"**s and **"why"**s are the only things that really matter when you are in an interview.

To be honest, this book entails far more than what its title promises. Once you complete this book, in the suggested way, you **will** slam-dunk that interview, that's for sure; and there is more than that: You will have a solid understanding of **JavaScript**, and you will have the necessary courage to look behind the veil (*i.e., the source code*) of any **JavaScript** framework you plan to use.

So just **who is this book for?** This book is for anyone who desires an improved **perspective** on **JavaScript interviews**, and who wants to leverage this new perspective on a consistent basis in the playing of the game of "**virtual reality**".

This book is for those who understand the value and necessity of **hard work**, and are willing to put forth an effort to learn:

This book will teach you lesser-known details on how to prepare for a **technical interview** in general, and how to prepare for a **JavaScript Engineering** interview in particular.

As they say, "**The Devil is in the details**".

If you have only a couple of days before that critical interview, and you are expecting this book to be a last-minute game changer, with hundreds of questions and answers to memorize, then this book is **not** for you. Either **change your mindset** or stop reading it right now!

This book is for "**doers**"; not spectators or skimmers. This book is for **believers**, for those who instinctively say "*yes that makes sense*" and dive in right away. This book is also for **skeptics**, who may doubt the efficacy and value of the material, but are **open-minded** enough to begin and give things an honest chance. This book is for all the frustrated candidates who have been into a great interview, but haven't received the expected outcome (*i.e., an offer*) from the process.

Contrary to most of the programming interview books on the market, this book will **teach** you **how to think**, instead of forcing you memorize a set of "*how do you balance an unbalanced binary tree?*" kind of questions.

Rather, this book will show you all the most important features of **JavaScript** and how they **fit together**.

After you finish this book, you will have a **solid foundation of knowledge** that you can build upon to knock out **any** kind of interviewing challenge thrown at you.

Enough said; let's begin.

How to Read This Book

I haven't made up any question in this book. Every one of them has been gathered from one or more interviews. The only thing I did was to alter the nature, setup and wording of some of the questions, so that they are different from the original ones; and that's not a big deal, because **there's no spoon**: As you'll see soon, it's not the question that's important, it's **the road** that leads to it. It is **not** the answer, but rather it's **how** you **approach** the answer: Simply memorizing the answers presented in this book will be of little use, if any.

Rather than focusing on the needles on individual pine trees, try to **see the forest**:
Try to realize the relatively few topic areas that these questions converge on.
That way, you will be able to handle **anything** thrown at you.

Learning by watching is never as effective as learning by doing. If you want to get the most out of this book, **work out the problems yourself**. I suggest the following method: After you read a problem, close this book right away; try to answer the questions yourself; then search the Internet and fine-tune your answer; write down your answer; after you are sure that you're done with your answer, compare and contrast with what you see in this book.

The more you work on yourself, the better your overall understanding will be.

How This Book Is Organized

This book can be roughly split into two parts:

- ★ Technical interview topics;
- ★ And the behavioral interview process.

Here is a deeper drill-down of each part:

Technical Interview Topics

Chapter 2: Algorithmic Complexity and the Big-O Notation

The **Big-O** notation is something you just have to know before going to an interview. Do not ever walk into an interview without knowing how to make a **Big-O** analysis. You may think that Big-O is not the solution to all problems; there are many misconceptions about it that sometimes we take for granted, and it really needs an update, and these do not prevent you from the fact that you should know how to make a Big-O analysis to get a programming job in the Silicon Valley.

Chapter 3: Warm Up Questions

In this chapter, we will go over a set of sample questions that can be answered with less effort and in a relatively short amount of time if you know the subject matter. Most of these questions are asked in the initial phases of an interview (*either during phone screening, or in an interview immediately after the phone screen*).

A few recruiter friends of mine agree that **roughly 80% of candidates are eliminated during these initial warm-up questions**. Do not memorize them though, because it is highly unlikely that you will come across the same question that I present in this book in an interview. **Try to understand how to approach the problem** instead, and make sure you clearly show your thought process to the interviewer.

Chapter 4: The Building Blocks

These are the important ideas that you have to master to have a solid understanding of **JavaScript**.

Learn this chapter well before going on to the next chapter, because these building blocks also lay the foundations of more advanced **JavaScript Patterns**.

Chapter 5: Patterns of JavaScript Architecture

In this chapter we will cover a selected subset of design patterns that are asked in the interviews most.

This won't be an extensive chapter about everything and the kitchen sink about **JavaScript** patterns. Instead, we will cover the most frequently used (*hence the most frequently asked in interviews*) ones.

I will be compiling **an extensive list** of **JavaScript** patterns in the sequel to this book:
"JavaScript Interview Questions – Deluxe Edition".

Chapter 6: More Than JavaScript

Knowing **JavaScript** alone will rarely guarantee you a job, if ever.

JavaScript lives in an ecosystem. So you have to be familiar with other technologies such as **CSS3**, **DOM**, **HTML5**, **CSS Preprocessors**, **Build Tools**, etc. You have to **be a rounded developer** too. So you'd have to know **how the HTTP protocol works** (note: I'm not talking about "HTML"; you have to know **HTTP** at the protocol level; because at every three of five interviews that you'll experience, one systems and operations wizard will ask you this:
"Now, explain me what happens when the user types http://unicornsandrainbows.com/ to the browser, tell me about DNS, Caches, Proxies, Headers, Packets... all that jazz! I wanna hear'em all!";
additionally you should know what an Oauth flow is, how many flavors of Oauth are there. You should also have a fair understanding of web application security, usability, and accessibility.) – If you want to be a **JavaScript Engineer**, and haven't looked at **Node.JS**, or you haven't played with a Map/Reduce **noSQL** document store such as **MongoDB**; you're gonna have a bad time. I will be giving links and pointers on what to study along with the material covered in this book to be a **Rockstar JavaScript Engineer**.

Being a rockstar is not an easy task, if it were everyone would be working in their dream companies. It's hard work. It's much sacrifice, and it really pays off on the long run.

The Behavioral Interview Process

Chapter 1: Behavioral Tips and Tricks

There's a reason I wrote this section first, before the "technical" portion: **This part is important! Treat it that way**.

Chapter 7: The Nontechnical Parts

These are as important, and sometimes more important than the technical parts of the interview. Learn them well. Practice them a lot. You will have a better chance to survive, so to speak, if you know how the system works.

Chapter 8: An Overview of the Behavioral Interview Process

Contrary to popular belief an interview does not consist of entering a room, answering a couple of questions, and getting out. It is way more than that. **An interview is a multifaceted process**; and to ace that interview you have to know every part of the process very well.

Chapter 9: Sharpen Your Katana Forever

Self improvement is a lifelong process. Improving yourself only during your job search is the most ineffective (*read: "dumbest"*) action that you can take. Always, and continuously improve. You can start by reading every reference material in the bibliography of this book.

Chapter 10: How to Create a Killer Resumé

There is only one goal of your resumé, and it is to get you an interview.
So if you have a killer resumé, your chances of getting an interview dramatically increase.

Creating a resumé requires much more time and effort than most people tend to think; and using a shiny template for your resume **will not** grab attention of the interviewer. In this chapter we will learn **how to make your resumé stand out**.

You'll be surprised that it's not what you expect.

Chapter 11: The Interview

In this chapter, we will cover all the stages of an interview; how to prepare for an interview; common mistakes and misconceptions about the interview; what the two most important interview questions are, and how to answer them; how to ask questions; and how to close an interview.

Chapter 12: How to Handle Offers

This is also a multipart process. There are proper ways to **receive** an offer, **accept** an offer and **decline** an offer. This is the most **delicate** part of the overall process. If you are extended an offer, then next couple of years of your professional life may depend on how well you handle that offer.

Chapter 13: The Postlude

This is an overview of what we've covered in the former chapters.

Although it's two pages long, I believe it's important enough to deserve its own chapter. Because, well... it **is** important!

Conventions Used in This Book

Code will be colored using syntax highlighting. This will help you understand the code, but you will be just fine reading this material on a monochrome e-book reader such as the Kindle Touch.

Besides syntax highlighting the code, the text in this book is colored to distinguish between JavaScript words and keywords, JavaScript code, and regular text:

- ★ Inline verbatim text, like source code samples, keywords, and terminal/console commands, and the like inside paragraphs, will be **`bold monotype`**,
- ★ If there's a part of the code that is being discussed, its background will be highlighted.
- ★ The source codes have a link to their associated github page. You will need a github account and please notify me, to be able to access the source code.

The excerpts on the following page demonstrate these semantics.

Color Coding in the Source Code

```
// 0004_singleton_object_create.js

var baz = Object.create(Object.prototype,  {

    // foo is a regular "value property".
    foo: { writable:true, configurable:true, value: "hello" },

    // bar is a getter-and-setter (accessor) property.
    bar: {
        configurable: false,
        get: function() { return 10 },
        set: function(value) { console.log("Setting `o.bar` to", value) }
    }
});

// will alert "hello".
alert(baz.foo);
```

Emphasized Sections of the Source Code

```
// 0031_closure_event_handler.js

function createHandler(node, i) {
    node.onclick = function() {
        alert(i);
    };
}

window.onload = function() {
    var node = null;
    var i = 0;

    for (var i = 0; i < 10; i++) {
        node = document.createElement('a');
        node.innerHTML = 'Click' + i;

        createHandler(node, i);

        document.body.appendChild(node);
    }
};
```

Source Code Within Text

The **ref** function encapsulates a copy of the variables and scope when it is created. So when it's called for the second time, the value of **tmp** will be incremented (since it's still hanging around

`bar`'s **closure**); the code will alert **`17`**. Note that **`bar`** can still refer to **`x`**, and **`tmp`**, even though function **`foo`** has returned after the **`var ref = foo(2);`** assignment.

Read the Source, Luke

You can **`git clone`** the source code used in this book, along with other helpful materials from its github repository. Since the repository is **private**, you need to send me your github username first. Just send me your github username, and I'll grant you access to the repository.

I highly suggest you join the repository, because I may put extra source code and some hidden gems in there which will not be present in this book.

Not only can you access theses **hidden gems**, but joining the repository has other benefits too: One obvious advantage is that you will find a gang of hungry minds to get in touch with. You can open discussions, share **your** particular interviewing experiences, and see that you are not alone (*in my honest opinion, no matter how well designed it is, any job interview* **sucks**. *Interviews are merely virtual realities where the physics of the world is governed by the interviewer; and I'm confident that the entire* **#jsiq** *community feels the same: The industry as a whole needs a paradigm shift; and that's one of the reasons I'm writing this book*).

You are more than welcome to open issues, ask questions, create wikis, and contribute any way you like to **#jsiq github repository**.

If you want to contribute to the source code, you are welcome too: Just keep in mind to **work on your own branch** and issue a **pull request**.

If you're new to **github**, learn.github is a good place to get started with **git**. For a deeper drill down you might want to read **Scott Chacon**'s Pro Git Book, or you might like try git immersion for a hands-on experience; **Ben Lynn**'s git magic will enchant you if you are one of those who think "**work is play**", for the video-learner types **Ralf Elbert**'s screencast is an excellent starting point, and for those "gimme my flowchart" learners **Mark Lotado**'s Visual Git Guide is a great start.

Using Code Examples in This Book

You can use the source code accompanying however you like, given that your actions do not violate "fair use".

In general, you may use the code in this book in your programs and documentations. You do not need to contact for permission unless you're reproducing a significant portion of the code.

For instance writing a program that uses chunks of the code in this book **does not** require permission; distributing a CD-ROM of the code in this book, however, **does** require permission. Similarly, answering a question by citing the code in this book does not require permission.

I appreciate, but don't require, attributions. A typical form of attribution to a particular page would be as follows:

> Özçelik, Volkan; "JavaScript Interview Questions / Algorithmic Complexity and Big-O Notation", page 36

I believe the reader is clever enough to understand what "**fair**" use is.

When in doubt, feel free to contact me.

Spread theWord

You'll never know how hard writing a technical book is unless you start writing one yourself. The book that you're reading right now is a self-publishing book. That means I'm doing all the writing, publishing, and marketing myself. I'm spending a great deal of time and effort to create this book; and I'm doing my best to ensure that this book is of great content and quality. This is unimaginably freaking hard work! If you like what you read in this book, I'd truly appreciate your help.

"**JavaScript Interview Questions**" is a book that I myself would have bought instantly without a second thought. It's also a book that I would recommend to anyone, because I believe that anyone from novice to professional (*even those who are not directly involved with* **JavaScript**) will find something valuable in this book.

I pay attention to every detail, I **reread** and **refine** and **distill** the material repeatedly. This attention to detail is what creates "a tension" between the way I'd like this book to be, and what it is right now; and it's this tension that gradually evolves the book to an even-better state. In fact, you can be a part of this **(r)evolution**:

Tweet #jsiq

You can use **#jsiq** hashtag for this book.
You can tweet **http://bit.ly/interviewz** as a link to share.

Share #jsiq on Facebook

"**JavaScript Interview Questions**" does not have a Facebook page, for I simply don't have time to maintain it. That's why I decided to focus on what I do best (*i.e., writing the darn thing*) and let the community spread the news. So feel free to share **http://o2js.com/interview-questions/** wherever you can. It really helps and I truly appreciate your efforts.

Refer #jsiq to Your Friends

"JavaScript Interview Questions" can be ordered at **http://o2js.com/interview-questions/** .
If you like the book, refer to your friends. If the book is helpful to you, why not help others too?

Write a Review in Your Blog About #jsiq

I'd love to read your reviews about the book.
Just don't forget to **send me the link**, so I can cite and promote it in the final version of this book too.

Chapter 1
Behavioral Tips and Tricks

There's a reason I wrote this section first, before the "*technical*" portion: **This part is important! Treat it that way.**

Hard skills (*such as learning a new programming language*) are **easy**,
and **soft skills** (*such as learning to appear positive, assertive, confident, and trustworthy*) are **hard**.

Before diving into technical questions, I'd like to tell you about the psychology of an interview, and different techniques the interviewer may use to probe your knowledge.

Behavior and **personality** are important traits in everyday work life, because *people are hired for technical reasons, and they are fired for personal reasons*. Thus, one of the goals of the interviewer is to assess how you fit to the overall culture of the company; and every technical interview includes implicit and explicit nontechnical questions. Even when you are asked a technical question, the recruiter will not only be evaluating your answer, but she will also be looking at subtle behavioral cues, to assess how **confident**, **comfortable**, and **knowledgeable** you are.

There's no point in continuing with the interview if you're not the kind of candidate in their minds.

While you won't get an offer purely on the strengths of your characteristics and traits alone;
appearing uninterested, or unsure about the subject you are talking about can definitely lose you an offer.

Interviewers Ain't No Dumb

Being a rockstar in your field is simply not enough, unless you show some manners:
You have to show certain character traits to take things to the next level.

Trying to take control of the overall process (*like, delaying it, or creating a false sense of urgency*) can be *risky*:

Don't try to trick the interviewer.
Don't ever think that you can outsmart them.

Remember, the goal of an interview is to eliminate as many candidates as possible to find the **perfect fit**. So the default answer in any job interview is **NO**. The interviewer does not know everything; they only know what you tell them. Worse, they don't even know what you've told them. **They believe in what they think you have told them**. So being prepared is not enough. You'll always have to deliver your answers **clearly**, **forcefully** and **enthusiastically**. They have to get excited about it, so that you can turn that NO into a **YES**.

It's not what you do that's important, it's what the interviewer thinks you do.

It's your **hard work**, **persistence**, **confidence**, along with your knowledge, skills, and abilities, that will change their minds. Remember: the interviewer is, and always will be, in the control of the overall process.

Don't outsmart your interviewer.

For example given "*What is your biggest weakness?*" question (*a question you will always get, while the format and wording of it may differ on different occasions*); giving an answer like "*My biggest weakness is that my professional network is in Boston, but I'm looking to relocate to LA.*" (*as suggested in a Harvard Business Review Article*) is **completely and utterly wrong**.

Here's why:

The purpose of the "weakness" question is to see how you honestly can express a (*guess what*) **weakness**, and what you are doing to overcome it. Anyone can see that the above answer is a "made up" one to skip that part of the interview.

Think of yourself in the shoes of the interviewer for a second. Wouldn't you feel humiliated?
And do you think humiliating your potential employer is the smartest way to get a job offer? – I don't think so.

Don't outsmart your interviewer (*that's the third time I'm writing it*).

Here's the correct way to approach the above scenario:

When you are asked for your weakness, have some guts and **talk about a real weakness related to your job role**, and **what you do to overcome it**. That is to say; clearly state your weakness, and then clearly state what you are doing to improve in that area. I assume you're wise enough **not** to choose a weakness that's core to the success of your job.

What I mean, **do not** choose something like:

"*My weakness is...* **well...** *I am really experienced with jQuery, but I cannot even write a cross-browser click event handler without a supporting library such as Prototype, Dojo, or jQuery. But I do try to improve myself.*"

If I were the interviewer (*I was*) and got that answer from a candidate (*I did*) I'd have made my decision at that moment.

Be Confident

No matter how skillful you may be, your behavioral traits will affect the hiring decision of the company.

There's no such thing as a behavioral interview. You will be assessed about your behaviors during the entire interview process. Your manners are as important as, and even more important than your technical skill set.

Smile and be confident. Smiling is the most powerful universal gesture. It's a more powerful tool than you can imagine. And unless there's a problem with your facial muscles, you **can** smile. Give it a try, you'll be amazed how it positively affects the overall outcome.

Interviewers may probe you for your knowledge on the subject, by asking questions like... *"Are you sure it works that way?"*, *"I never knew that was possible?"*, "*I looked at your github project, and FooController.js seems to have an error around line 128, is that check really necessary?*"

Be prepared for those kinds of questions. Those questions all inquire whether you are self-assured about the answer you convey, or you have simply memorized a group of answers and are spitting them out to the interviewer.

Do not outsmart your interviewer. If you don't know an answer to a question, honestly say that you don't, and ask whether they can give you some additional information to help you in your thought process.

Giving a confident "**no**" as an answer is way superior to being unsure, hesitating, and mumbling.

Confidence is the key for any question thrown at you, no matter how trivial it may be. During your answer make sure you don't raise suspicion. Know what to say, and confidently answer what you've been asked for.

Also make sure that you answer the question you're being asked **right away**. For instance the answer to the famous question "*What's your GPA?*" is a floating point number, as in "*my GPA is 2.8*". If you start answering that question with a sentence that starts with **"well..."**, it will clearly indicate that you are not prepared for the question, and you are trying to soften or hide things.

Don't outsmart your interviewer; just answer directly what you've been asked for. Be calm and confident.

...

This was just a brief introduction to emphasize the importance of your manners during the interview. We'll come to tips and tricks of how to behave, and look at frequently asked behavioral interview questions and examine them in depth at the end of this book.

A Few Words About Technical Interviews

Technical interviews give you the chance to show your knowledge, skills, and abilities.
Needless to say, they are the key decision factors, for companies, on whom to hire and whom not to.

Contrary to popular belief (*and what's written in many interview books*), technical interview questions are **not difficult**. Don't get me wrong, though. These questions are not pieces of cakes, either. If they were, then anybody could have answered them; and there would be no point in the interviewer asking the question first.

However, **it's not 2005** and people are not asking impossible to solve brain teasers anymore; or at least the number of such companies is significantly decreasing. Even companies like Google ban brain teaser questions.

Moreover, there are legal aspects to it:

> *A brain-teaser-type test used in a hiring process might be illegal if it can be shown to have disparate impact on some job applicants and is not supported by a validation study demonstrating that the test is related to successful performance on the job.*

Companies outside the United States are regulated by different laws, of course.

> If you meet a firm that does not audit the skill set that you will be using during your work, and rather asks "how many golf balls you can fit into a school bus" then they are looking for **walking scientific calculators**, instead of **pragmatic**, **proactive**, and **creative** individuals.
> I don't know you, and I'd prefer to work in a **better** environment.

Besides, those questions don't seem to have any validity on determining how qualified you are as a candidate.

Study shows (*see the above link*) a job screening procedure that works reasonably well is a **work-sample test**: In a work-sample test, **the applicant does an actual task** or **group of tasks** like what **the applicant will do on the job if hired**.

> I know there are **managers** and **recruiters** reading this book too. If you are on the other side of the desk; acting as a hiring decision-maker, it's a good idea to think about how to incorporate a **work-sample test** into your hiring processes. Honestly, brain teasers are just a stupid banking tradition and it's time to burry them in the ground.

Nonetheless, some of the questions may "*seem to be hard*", with the intention to see how you tackle a problem when you don't immediately see the solution; and you'll soon see that the important thing is not knowing the answer, but **how you approach the question**.

Don't get frustrated if you don't see the solution right away.

The important thing is to break the problem into sub problems, approach it analytically and logically, and solve it step by step; and while you do that, make sure that you **keep talking, and always explain what you are doing**. That's the only way that the interviewer will understand how you tackle the problem.

Apart from the *legal* and *practical* aspects, one reason that the problems are not hard is the **limitations of the interview environment**: The interviewer has to ask questions that have to be easily explained and solved in a **reasonable time**; but the questions should not be too easy as they have to check your expertise and domain knowledge. Otherwise the interview would have no point at all.

Also keep in mind that in any technical interview, the code you are writing in the interview is most probably the only piece of your code that your interviewer will see. – I've seen companies analyze my **Github** profile, and **open-source projects**. I even had interviews based on open source code I've written at **github**, discussing ways to improve it -- but that's a trace amount of evidence to assume that the majority of the companies do that. So **interactivity** is the key.

Don't assume that people will read your open source code (*or even look at your resumé for more than ten seconds*). It's a fact of life that a considerable percentage of the interviewers *will have only a few minutes* to read your resumé before the interview.

A second thing to keep in mind in interviews is "**honesty**". If you have seen a problem before, **mention it**.

Interviewers are not dumb, and they'll understand you're blurting out something that you've memorized before.

These are the traits you should have as a candidate. Next up, we'll see what you've been waiting for:

An extensive collection of "JavaScript Interview Questions", along with the core concepts that you need to ace them.

So let's begin. Shall we?

Chapter 2
Algorithmic Complexity and the **Big-O** Notation

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 3
Warm-Up Questions

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 4
The Building Blocks

Most of the topics and questions that are mentioned in this section are mentioned in almost all interviews.

As always, I will be presenting them in most-to-least frequently asked order. When you finish this section (*and by "finish" I do* **mean**, **intend**, *and* **insist** *that you also read all supporting material*), you will have gained enough knowledge to build your own JavaScript framework (*seriously, I'm not kidding*).

This section also constitutes the foundation of the following **"JavaScript Patterns"** section, because you cannot properly use a **JavaScript** pattern without thoroughly understanding how **JavaScript** works internally (*that's what we'll try to cover in this "the Building Blocks" section*).

To repeat, during your interview process, you will be asked **most of the topics covered in this section**. These may be either *"definition"* questions (*possibly asked on a phone screen*), asking you to define what the subject matter is, or they can be "*walk me through this code piece*" kinds of problems, where you will be given a piece of code, and asked why it behaves the way it behaves.

Sometimes the questions may get trickier, where most of the time the interviewer also knows that what she's asking is lesser-known parts of the language; but she'll expect any rockstar candidate to know about them anyway.

And if you are not a rock star candidate, why should they hire you?

Remember, failing to give a satisfactory answer to any of the following topics
will be equivalent to kissing your job offer goodbye.

At any rate, if you are applying for a **JavaScript Engineering** job (*and if you are reading this book, chances are that you are*), you will have to know **all the minute details** of the language; and our journey continues with the often-misunderstood, yet the most crucial part of the language, namely: "**Closures**". Shall we?

Closures

Answering closure-related interview questions will be a cakewalk once you understand how **closures** work.

Actually, **closures** will be an important part of your everyday work as a **JavaScript Engineer**:

You'll be using them ubiquitously. Hence, you are expected to understand them very well. Otherwise it's unavoidable that you'll mess things up and give birth to hard to find **execution-context**-related, and **scope**-related bugs.

Closures are one of the most powerful (*yet misunderstood*) features of **JavaScript** (*EcmaScript*). They cannot be properly exploited without understanding them. They are, however, relatively easy to create, even accidentally, and their creation has potentially harmful consequences. To avoid accidentally encountering the drawbacks and to take advantage of the benefits they offer, it is necessary to understand their mechanism.

Moreover, a thorough knowledge of **closures** is crucial in understanding how memory leaks are formed between the **DOM World** and the **JS World**, and how to avoid them.

There are more examples of **closure-related memory leaks** in this excellent MDN Tutorial. Although some of the leaks mentioned in the article have been mitigated in the newer versions of IE (*and most of them have never existed in other browsers nonetheless*), it's a good practice to have a clear understanding about them. Most of the time those closures are the first places to further investigate for **performance improvements**, **refactoring**, and **optimization**.

One further clarification regarding "**memory leaks**" and **closures**:

Closures use memory, but they don't cause memory leaks since **JavaScript** by itself cleans up its own circular object chains that are not referenced. IE memory leaks involving closures are created when it fails to disconnect **DOM** attribute values that reference closures, thus creating a "circular reference". For details, see this MDN Article on Memory Management, this MSDN Article on how Script Garbage Collectors work, this article on memory management and recycling, and Martin Wells' article on writing high-performance garbage-collector-friendly code.

What Is a JavaScript Closure

Here's the definition of a closure from wikipedia:

> A **closure** (also **lexical closure** or **function closure**) is a **function** together with a *referencing environment* for the **non-local variables** of that function. A closure allows a function to access variables outside its immediate lexical scope. An "**upvalue**" is a **free variable** that has been bound (*closed over*) with a closure. The closure is said to "close over" its upvalues. The referencing environment **binds** the non-local names to the corresponding variables in **scope** at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime of the closure itself. When the closure is *entered* at a later time, possibly from a different scope, the function is executed with its non-local variables referring to the ones captured by the closure.

"*If you can't explain it to a six-year old, you really don't understand it yourself.*"

Let alone a six-year old, any seasoned developer will need to read the above definition several times to have any idea of what it says.

Closures are not hard to understand once the core concept is learned. However, they are impossible to understand by reading academic definitions like the one above.

A **closure** is basically formed every time there's an instantiation of a **function**. By that token, every **function definition** is, indeed, a **closure**; but most of the time what's meant by a closure is a "*function defined inside another function*", or "*a function returning another function*".

> Technically, in **Javascript**, every **function** is a **closure**. It always has access to variables defined in the surrounding scope; but what the interviewer means by "*closures*" is most of the time a "**function**" **within another** "**function**".

Here's an example:

```
// 0026_closure_sample.js

function foo(x) {
    var tmp = 3;

    function bar(y) {alert(x + y + (++tmp));}

    return bar;
}

foo(2)(10); // will alert "16".
foo(2)(10); // will alert "16" again.
```

Whenever you see the **`function`** keyword within another function, the inner function has access to variables in the outer function, where we say that the inner function **closes over** the scope of the outer function. The above example will alert **`16`**, since the **`bar`** function will close over the variable **`tmp`**. **`bar`** can also access the argument **`x`**, which was defined as an argument to **`foo`**. The inner function **`bar`** *closes over* the variables of **`foo`**, before function **`foo`** exits.

When creating a **closure**, the form in which the function is defined does not make any difference either:
It could be either a function declaration or a function expression.

Let's make things more fun and slightly modify the former code:

```
// 0027_closure_modified.js

function foo(x) {
    var tmp = 3;

    function bar(y) {alert(x + y + (++tmp));}

    return bar;
}

// `ref` will close over a copy of current function `foo`s scope <tmp=3, x=2>.
var ref = foo(2);

ref(10); // Will alert "16" <tmp=4, x=2>.
ref(10); // Will alert "17" <tmp=5, x=2>.
```

The **ref** function encapsulates a copy of the variables and scope when it is created. So when it's called for the second time, the value of **tmp** will be incremented (since it's still hanging around **bar**'s **closure**); the code will alert **17**.
Note that **bar** can still refer to **x**, and **tmp**, even though the function **foo** has returned after **var ref = foo(2);** assignment.

The reason the above code works that way is the fact that **JavaScript** uses **lexical scoping**.

> **Lexical scope** means functions are executed using the variable scope in effect when the function was **defined**. It has nothing to do with the scope in effect when the function is called. This fact is **crucial** to unlocking the power of **closures**.

For a more detailed review on closures, read Angus Croll's excellent article, Juri Zaytsev's Use Cases for JavaScript Closures, and Jim Ley's FAQ notes on Closures; or if you are a visual type of person, you may want to have a look at Ben Nadel's visual explanation on closures, and if you like analogies read how Derek relates closures to marriage.

> **Closures** are the building blocks of many **functional programming** questions that you'll be asked on follow up interviews. **Study them well**.

There are a number of articles out there that explain closures. I've shared a couple of the valuable ones with you in the previous paragraph. Some of the closure-related articles that you might find on the web take for granted that everyone has developed in about fifteen other languages before. Although language-agnostic academic articles on *lexical closures* are nice to read, those articles can be better comprehended after seeing how closures work in the wild – so it turns out to be a *causality dilemma*.

Starting from the next page, you'll find a series of **closure-related** interview questions. After going through the sample questions in this section, my humble goal is try to convey to you **what closures are**, **how they work**, and more importantly **how you can specifically benefit from them**.

As any other interview question bundle in this book, this section is just a **representative set**.
So reading them cover to cover will be of no use at all. **Do not memorize them**.
Instead, focus on **what the interviewer tries to learn about you** when she asks those questions.

What is a "JavaScript closure"; when would you use one?

Answer:

If you've done your homework, the answer is obvious.

However, the interviewer would want the explanation "*in your own words*".

You'll need an **authentic**, **less formal** answer such as:

> **From a simplified perspective**, a **closure** is an enclosed scope of the local variables of a function.
> This scope is kept alive after the function returns.
> Or in other words, a closure is a stack frame that's not deallocated when the function returns.
> A **closure** is a special kind of object that combines two things: **a function**, and any **local variables** that were in-scope **at the time that the closure was created**.

The above definition is **not** a rigorous definition of a closure; and that's exactly what the interviewer is looking for.

> **Do not memorize** a formal definition; express the idea **in your own words**.

Here is a very basic example of a **closure**:

```
// 0028_closure_greet.js

function greet(name) {
    return function() { alert('Hello ' + name); };
}
```

As for the **second part** of the question (*i.e., when would you use one*), the following should suffice:

> **Closures** reduce the need to pass state around the application. The *inner function* has access to the variables in the *outer function* so there is no need to store the state data at a global place that the inner function can get it.

In a typical scenario the *inner function* will be called after the *outer function* has exited. The most common example of this is when the i*nner function* is being used as an **event handler**. In this case you get no control over the arguments that are passed to the event handling function; so using a closure to **isolate state data** can be very convenient.

In the world of compiled programming languages the benefits of closures are less noticeable.
In **JavaScript**, however, closures are incredibly powerful for two reasons:

1) **JavaScript** is an *interpreted language*[1].
So **instruction efficiency** and **scope conservation** are important for faster execution.

2)**JavaScript** is a *lambda language*[2].
This means, **JavaScript** functions are **first class objects** that define **scope;** and scope from a parent function is accessible to child functions. Indeed, the only scope in **JavaScript** is "**function scope**". This is **crucial**, since a variable can be declared in a parent function; used in a child function; and retain value even after the child function returns. That means a variable can be reused by a function many times with a value already defined by the **last iteration of that function**.

As a result, closures are incredibly powerful and provide superior efficiency in **JavaScript**.

[1] Not %100 true, since engines like V8 and TraceMonkey compile **JavaScript** into machine code before executing it; rather than interpreting it.

[2] As coined in JavaScript the Good Parts.

What happens when the code below gets executed?

```
// 0029_closure_typical.js

window.onload = function() {
    var node = null, i = 0;

    for (i = 0; i < 10; i++) {
        node = document.createElement('a');

        node.innerHTML = 'Click' + i;
        node.onclick = function() {alert(i);};

        document.body.appendChild(node);
    }
};
```

Discussion:

This is the "**de facto**" closure question, and you will come across **many** variants of this during your interviews.

> If you do not immediately see *the elephant in the room*, you should read the links and reference material given at the "What is a JavaScript Closure" section again.

Human brain is a strange machine: If you cannot find an immediate exit point, you start to delve into increasingly convoluted decision trees, and thought patterns.

Let's assume for a moment that we don't see the obvious problem in the question above.
Then, one thing to focus can be the **`window.onload`** function itself:

We may be overriding a former **`window.onload`** implementation. Therefore, we may need to cache it, before running our function to prevent side-effects (i.e., **`var oldLoad = window.onload || function() {}; window.onload = function() {oldLoad(); ... do stuff ...}`**).

Besides, the assignment above uses DOM Level 0 event registration, and there are more modern ways to do it, maybe we should use a **cross-browser DOM Event Helper**, to begin with.

The same is true for **`node.onclick`**.

If we had an **events** library, we could have used it as something like:

```
lib.on('click', node, function(){...});
```

...

Another thing is, if window has already **loaded** and we are registering the **`window.onload`** function, through some kind of lazy-loading mechanism or asynchronous script injection; our **`window.onload`** function will not run at all, because the **`load`** event of window has already been fired. So we may need to implement a cross-browser way to check the readyState of the document (which is better than window.onload) and run it instead; or we may need to check the availability of an element, and run our method after that if that's more efficient for responsiveness.

...

Further, what if, for some reason, our code runs multiple times? Shall we append the link elements to the document over and over again?; or is it better to create a *container node*, clear the contents of that container, and append the elements to the container when we execute the loader function a second time? Or can we use some functional magic to ensure that the code runs once and only once, maybe?

Yet another thing we can focus on can be the way we create nodes:

Instead of using **`innerHTML`**, how about **`document.createTextNode`**? Isn't it more preferred?

And instead of appending a link element to the body at each iteration, we can create a document fragment, append to the document fragment and when the loop exits. Then we can append the fragment to body to minimize repaints and reflows.

"Exploring alternate paths, and thinking outside the box" is something that you should do, especially if you don't quite know the concept. However, you should do it **AFTER** *honestly telling the interviewer about that.*

An interviewer will appreciate this kind of thought process, **and it will not hide** the fact that you do not know closures.

Honestly, if somebody gives me an answer with this level of detail, I won't give a darn whether she knows closures. She will be a **strong candidate**; and I will consider her for a follow-up interview. But that's me, and the interviewer may have sharper and stricter rules.
Learn your stuff and **don't risk your chances**.

Exploring different paths helps at times, and there are rare cases that the problem you are working with is indeed algorithmically complex, or involves obscure parts of the language to solve it elegantly. However, most of the time that's not the case; especially if it's asked you in an interview. We've already seen that an interview has specific limitations that makes it hard to ask really complex problems.

When you find yourself whirling into increasingly complex solutions,
take your **Occam's Razor**, and **think simple**.

Answer:

I know; those who've seen what's wrong with the question are jumping in their seats up and down.
Without keeping you waiting any further, here's the answer that the interviewer **expects** from you:

Inside the for loop of the `window.onload` function 10 different *HTML link elements* are created; and **at each iteration** of the for loop, an anonymous function literal is assigned to the `click` event handler of the created *HTML link element*. These function literals are said to **close over** the variable `i`. So even after `window.onload` function executes, and all the nodes get inserted into the **DOM**; the event handling functions will be able to access the latest value of the variable `i` (*which is* `10`).

Therefore clicking **any of the links** inserted to the page will alert `10`.

Fix the former example so that each link alerts the number associated with it. i.e., the first link should alert 0, the second link should alert 1... etc.

Answer:

When you correctly answer the former question; this is generally the *follow-up question*.

Remember; "**i**" in your function is bound at the time of **executing** the function, not the time of **creating** the function:

> When you create the closure, **i** is a reference to the variable defined in the outside scope. It is not a copy of it as it was when you created the closure; and it will be evaluated at the time of execution.

The usual fix of that is introducing another scope by adding an additional closure and using it as an **IIFE**:

```
// 0030_closure_typical_fixed.js

window.onload = function() {
    var node = null, i = 0;

    for (i = 0; i < 10; i++) {
        (function(i) {
            node = document.createElement('a');
            node.innerHTML = 'Click' + i;

            node.onclick = function() {alert(i);};

            document.body.appendChild(node);
        }(i));
    }
};
```

This will work as expected, alerting a different number when clicking a different link.

That's normally the answer that the interviewer **expects**.
Though the above code is not ideal in a production environment. Here's why:

Declaring anonymous functions inside a loop, the **JavaScript** interpreter will create a separate Function instance at each iteration; and if there are many such links, it may be bad for performance.

> Note that although separate **Function** objects are created, it does not mean that they share the same code; for instance Chrome's V8 JavaScript engine is clever enough to **reuse** the same code. However we cannot take this for granted.

Lets't try a different approach, then:

```
// 0031_closure_event_handler.js

function createHandler(node, i) {
    node.onclick = function() {
        alert(i);
    };
}

window.onload = function() {
    var node = null;
    var i = 0;

    for (var i = 0; i < 10; i++) {
        node = document.createElement('a');
        node.innerHTML = 'Click' + i;

        createHandler(node, i);

        document.body.appendChild(node);
    }
};
```

The above code works as expected; and we are not creating function objects inside the for loop; or, are we?

What we did is nothing but to **encapsulate** the function creation logic into the **`createHandler`** method.

If you trace the code, you'll see that we are still creating a separate anonymous function at each iteration.

We might be slightly better off, because of getting rid of the nested closures. What else can we do?:

```
// 0032_expando_properties.js

function node_click() {alert(this.expando);};

window.onload = function() {
    var node = null,
        i = 0;

    for (i = 0; i < 10; i++) {
        node = document.createElement('a');
        node.innerHTML = 'Click' + i;
        node.expando = i;

        node.onclick = node_click;

        document.body.appendChild(node);
    }
};
```

The above implementation shares a single common **`node_click`** function across all iterations, so it's **better**.

Also the **`node_click`** event handler does not close over the **`node`** variable (*unlike the previous implementation*), so it's also free from closure-related *memory leaks*. Moreover, we got rid of two closures by simply defining an expando property on the node object. That's an acceptable tradeoff.

As you may have seen from the above discussion, the thing that's important is **not** the solution.

Per contra, it's **the way** you logically **approach** the problem.

Simply **memorizing** a solution and parroting it to the interviewer **will not be of any use**.

Draw a picture in the interviewer's mind; and let the interviewer see what's inside your head.

What does the following code do? How can you improve it?

```
// 0033_question_bind_arguments.js

function bindArguments(context, delegate) {
    var boundArgs = arguments;

    return function() {
        var args = [], i;

        for(i=2; i < boundArgs.length; i++) {
            args.push(boundArgs[i]);
        }

        for(i=0; i < arguments.length; i++) {
            args.push(arguments[i]);
        }

        return delegate.apply(context, args);
    };
}

function calculate(a, x, b, c) {return (a*x + b / c) * this.factor; }

var bound = bindArguments({factor:4}, calculate, 10);

res = bound(1, 30, 60);

alert(res);
```

Answer:

That code is a JavaScript **currying** implementation.

The **`bindArguments`** function above takes a context, a **delegate**, then **curries** a variable number of arguments. So:

```
bindArguments({factor:4}, calculate, 10)(1, 30, 60);
```

will be equivalent to:

```
calculate(10, 1, 30, 60);
```

where **`this`** refers to an object literal of:

```
{factor:4}
```

So **`res`** will be equivalent to:

```
(10*1 + 30 / 60) * 4
```

Which will be **42**.

As per the *second part* of the question:

Once we understand what the function does; it's easy to improve it further:

> As of **JavaScript 1.8.5** (*EcmaScript 5th Edition*), there is a native alternative to binding a **context** and **arguments** to a function: `Function.prototype.bind`. So the following `bindArguments` implementation will be faster, since it's using native methods.

```
// 0034_bind_arguments.js

function bindArguments(context, delegate) {
    return delegate.bind(
        context,
        Array.prototype.slice.call(arguments).splice(2)
    );
}
```

We use **`Array.prototype.slice.call`** to convert **`arguments`** object into an **`Array`**.
Because **`arguments`** is an Array-like object and to use **`splice`** method we need to convert it into an actual **`Array`**.

But **`Function.prototype.bind`** is not supported in all user agents.
So we need to do a feature detection, and use a *fallback method* if it's not supported, as in the following example:

```
// 0035_bind_arguments_improved.js

var bindArguments = (
    !!Function.prototype.bind
) ?
function(context, delegate) {
    return delegate.bind(context,
        Array.prototype.slice.call(arguments).splice(2)
    );
} :
function(context, delegate) {
    var slice = Array.prototype.slice,
        args = slice.call(arguments).splice(2);

    return function() {
        return delegate.apply(context,
            args.concat(slice.call(arguments))
        );
    };
};
function calculate(a, x, b, c) {return (a*x + b / c)*this.factor;}

var bound = bindArguments({factor:4}, calculate, 10);
res = bound(1, 30, 60, 50);

alert(res);
```

A typical use of JavaScript closures is to create "so called" 'private members'. Can you give an example of this?

Answer:

Technically **JavaScript** does not support private access modifiers (hence the "*so called*" phrase in the question). However, you can isolate **private static functions** inside a **closure** to have some privacy.

This is one of the building blocks of the module pattern.

Here's an example:

```
// 0036_hide_your_privates.js

var Empire = {droids:{}};

Empire.droids.C3PO = (function() {

    // Private variables.
    var series = '3PO',
        prefix = 'C';

    // Private method.
    function toString() {
        return prefix + '-' + series;
    }

    return {

        // Public members.
        manufacturer: 'Cybot Galactica',
        serialNumber: '190e0696-b7db-4401-9e72-b742302e2b10',

        // Public method.
        toString: function() {
            return this.manufacturer + '/' +
                this.serialNumber + '/' +
                toString();
        }
    }
}());

// This alert will polymorphically use the toString method of the object;
// and alert "Cybot Galactica/190e0696-b7db-4401-9e72-b742302e2b10/C-3PO".
alert(Empire.droids.C3PO);
```

Given the following mapping implement Keys.isEnter, Keys.isTab, Keys.isShift, and Keys.isLeftArrow functions.

```
// 0037_question_keys.js

var Keys = {
   Enter: 13,
   Shift: 16,
   Tab: 9,
   LeftArrow: 37
};
```

Answer:

The naïve approach would be something like:

```
// 0038_keys_naive.js

var Keys = {
   Enter: 13,
   Shift: 16,
   Tab: 9,
   LeftArrow: 37,

   isEnter: function(key) {return key === this.Enter;},
   isShift: function(key) {return key === this.Shift;},
   isTab: function(key) {return key === this.Tab;},
   isLeftArrow: function(key) {return key === this.LeftArrow;}
};
```

If it was that straightforward, the interviewer would not have asked the question in the first place.

So let's add some spice to it:

```
// 0039_keys_dynamic.js

var Keys = {Enter:13, Shift:16, Tab:9, LeftArrow:37};

var key;

for (key in Keys) {
    if (Keys.hasOwnProperty(key)) {
        Keys['is' + key] = (function(key) {
            return function(k) {return k === key;};
       }(Keys[key]));
    }
}

// Will alert "true".
alert(Keys.isEnter(13));
```

Given the following code snippet create a "Factory Method" that creates objects, given different names and professions.

```
// 0040_question_factory.js

var message = "I'm " + name + ", and I am a " + profession + "!";

{
    shout: function() {alert(message);},
    shoutAsync: function() {
        setTimeout(function() {alert(message);}, 1000);
    }
};
```

Answer:

Another useful implementation of **closures** is Factory Methods; and the interview question ascertains your knowledge about factory methods. Here's a simple implementation for the above example:

```
// 0041_warrior_factory.js

var WarriorFactory = {
    create: function(name, profession) {
        var message = "I'm " + name + ", and I am a " + profession + '!';

        return {
            shout: function() {alert(message);},
            shoutAsync: function() {
                setTimeout(function() {alert(message);}, 1000);
            }
        };
    }
};

var warrior1 = WarriorFactory.create('Akechi Mitsuhide', 'Samurai');
var warrior2 = WarriorFactory.create('Kumawakamaru', 'Ninja');

warrior1.shout();
warrior1.shoutAsync();

warrior2.shout();
warrior2.shoutAsync();
```

Summary

That was a sample set of closure-related **JavaScript** interview questions.

Here are some final remarks:

★ Most of the time you use **closures** without even knowing. For instance **jQuery** functions like **$.ajax**, **$.click**, **$.live**, and **$.delegate** *internally* use **closures** in their implementations, and you pass **function literals** to those functions as event-handling callbacks, and form **closures**.

★ Whenever you see a **function** defined in another **function**, a **closure** is formed.

★ A **closure** keeps **copies of** (*for primitives*) and **references to** (*for objects*) variables in the scope that it's called.

★ There's only one scope in (ES5) **JavaScript** and it is **function scope**.

★ You can best visualize a **closure**, as a frozen set of **local** variables that's created just on the entry of a **function**.

★ A new set of local variables is created every time a **function** with a **closure** is called.

★ You can have **nested closures** (*i.e., functions within functions within functions...*).

★ **Functional programming** implementations, such as **memoization**, **partial functions**, and **currying** are practical implementations of **JavaScript closures** in real life applications.

Closures are one of the most tricky parts of JavaScript.
Once you grok the idea fully,
and practice some additional functional kung-fu to sharpen your skills,
you can easily dive into **any** tricky subject **without fear**.

Chapter 5
Patterns of JavaScript Architecture

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 6
More Than JavaScript

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 7
The Nontechnical Parts

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 8
An Overview of the Behavioral Interview Process

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 9
Sharpen Your Katana **Forever**

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 10
How to Create a Killer Resumé

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 11
The Interview

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 12
How to Handle Offers

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 13
The Postlude

This section is only available to those who order "**JavaScript Interview Questions**".

Chapter 14
Conclusion

This section is only available to those who order "**JavaScript Interview Questions**".