JavaScript Interview Questions

who else wants to nail that interview?

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

Dedicated to my wife Nagehan and our daughter Yaprak,

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 **AJAX** web applications. I have worked in 4 start-ups before, and I'm currently a Mobile Front End Engineer at Jive Software.

The stuff I love to architect is a **responsive** and **intuitive** front-end, driven by amazingly well-organized **JavaScript** code. I dream of the death of Internet Explorer, and shudder at the horror of thousands of people still using the crazy thing.

I've experimented with most of the **JavaScript** frameworks around; written a handful myself. I have also fiddled with **NoSQL**, database queries, **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?!

My Timeline at a Glance

- I am currently a **Mobile Front-End 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 that, 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

- http://o2js.com/volkan
- http://geekli.st/volkan
- http://github.com/volkan

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 concepts summarized in this book. Say, for example, "*functional programming*" 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 a lot of 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 you the **red pill** and tell what the **JavaScript Engineering** interviews **really are**, and what you will need to know to get the job you want.

I haven't made up any question in this book. Every single 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 than 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** to that answer: Simply memorizing the answers presented in this book will be of little use, if any.

Rather than focusing the needles on individual pine trees, try to **see the forest**:
Try to realize the relatively few topic areas that these questions converge.
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 the book.

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

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 proof-read. However, it's human nature and mistakes, bugs, typos and errors are inevitable. 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: It won't be a secret process where there's the candidate (*i.e. you*) as the input; dark woo do 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 tell 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 e-mail me at volkan@o2js.com.

I hope you'll enjoy reading this book as much as I enjoy writing it. **Now go ace that interview!**

Good Luck!

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 can better fit for the purpose of this book*); I'd like to thank **Brett Langdon** and **Melih Önvural** for their patience throughout the editing, proof-reading, and copywriting process; I'd like to thank to all **geeklisters** who have been with me along the way. And last, but not the least; I'd like to thank my friends and my family, who've always supported me throughout the process.

The book is still in progress, and you've already done a great deal of job. And it would be impossible to create this book, to this level of quality, without you.

I wholeheartedly and truly appreciate your support.

Keep it up!

Who This Book is for

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 for **JavaScript Engineering** jobs (and obviously this book is highly **JavaScript**-focused), anyone who wants ideas on how to **ace** in the technical and non-technical parts of an **Engineering** job interview process can benefit from this book.

> 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 minute details on how to prepare for a **technical interview** in general, and how to prepare for a **JavaScript Engineering** interview in particular. And, as they say, "**Devil is in the details**".

This book is for "**doers**"; not spectators or readers. 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. 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 over you.

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**.

Enough said; let's begin.

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 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 non-technical questions. Even when you are asked a technical question, the recruiter will be not only evaluating your answer, but she also will be looking at subtle behavioral cues, in order to assess how **confident**, **comfortable**, and **knowledgeable** you are.

> There's no point in proceeding with the interview if you're not the kind of candidate in their mind.

While you won't get an offer barely 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: **Your employer will take 90% of less ability to 10% of more attitude, every day of the week**. That's why you have to show certain character traits in order to take things to the next level.

Trying to take control of the overall process (*like, delaying an interview, or creating a sense of urgency to get an offer quickly*) can be *risky*:

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

Remember, the goal of an interview is to eliminate as much 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 will always 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. And 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 much 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 bunch of answers and spitting them out to the interviewer.

And, **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 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 is 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.

Technical Interview Topics

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

Preorder your exclusive copy to read.

Warm Up Questions

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

Preorder your exclusive copy to read.

The Building Blocks

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

Preorder your exclusive copy to read.

Algorithmic Complexity and Big-O Notation

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.*"

And let alone a six-year old, any seasoned developer will need to read the above definition several times to have an 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, in deed, 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 an 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:

```
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:

```
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**), and the code will alert **`17`**.

Note that **`bar`** can still refer to **`x`**, and **`tmp`**, even though 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.

Closure Interview Questions

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 section. And some of those articles 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 concept in your own words.

Here is a very basic example of a closure:

```
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. In deed, 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?

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

    for (var 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 a **de facto** closure questions, and you will encounter **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 once more.

Human brain is a strange machine: If you cannot find an immediate exit point, you start to delve into more and more 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, via 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 in terms of 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 in deed 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 learn to **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**:

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

    for (var 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 link it may be bad in terms of 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 **re-use** the same code. However we cannot take this for granted.

Lets't try a different approach, then:

```
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?:

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

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

    for (var 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 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?

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

    return function() {
        var args = [];

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

        for(var 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.

We will be seeing more examples of currying in the functional programming section.

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.

```
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 **`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:

```
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;

    var 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:

```
var Empire = {droids:{}};

Empire.droids.C3PO = (function() {

    // Private variables:
    var series = '3PO';
    var 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

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

implement Keys.isEnter, Keys.isTab, Keys.isShift, and Keys.isLeftArrow functions.

Answer:

The naïve approach would be something like:

```
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:

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

var key = null;

for (var 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

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

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

create a "Factory Method" that creates objects, given different names and professions.

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:

```
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 **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*).
- **Functional programming** implementations, such as **memoization**, **partial functions**, and **currying** are practical implementations of **JavaScript closures** in real life applications. We'll have more of these n the functional programming section.

Closures are one of the most tricky parts of JavaScript.

Once you grok the concept fully,

and practice some additional functional kung-fu to sharpen your skills,

you can easily dive into **any** tricky subject **without fear**.

Excited?

That was just a small sample of what you'll have when you **preorder the book**.

Pre-order the book before anyone else

(including *recruiters* and *decision-makers*)

and use this to your advantage **;)**