v.1.1.0 Last Update: **January 18, 2017**

JavaScript Interview Questions

Who Else Wants to Nail that Interview?

by **Volkan Özçelik** • j4v4ScR1p7 h4X0r @ volkan.io

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

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

Always be curious;

Never stop learning;

And may the odds ever be in your favor!

V.Özçelik

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?!

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.Özcelik

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 **`1.1.0`**, 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 **Wednesday, January 18, 2017**.

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 **CSS**, **DOM**, **HTML**, **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://02js.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://02js.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

Big-O is a **mathematical** concept, used to analyze the **asymptotic** behaviors of functions, first introduced by number theorist **Paul Bachmann** in **1894**. Though, if you are not a mathematician, you'll like this explanation on NIST better.

Foremost, do not even walk into an interview without knowing what a *Big-O Analysis* is.
It's simply something that you **must** know if you expect to get a job.

There's generally more than one solution to any given computer science problem. These solutions will often be in the form of different algorithms, and you will generally want a quick (*and dirty*) way to have an idea of which algorithm is better without using a profiler. This is where **Big-O** analysis helps.

Sounds quite boring, right?

At any part of the interview, you may be asked to assess a **Big-O** analysis of the code you've written. This may be a question the interviewer asks to probe your knowledge about **algorithmic efficiency** and **complexity analysis**. It's also possible that you may have implemented a naïve solution to a given problem, and the solution can be further refined (*usually by caching frequently used data structures in memory, which is known as the space-time tradeoff*). In that case, you refactor your code for an algorithm change, then redo the **Big-O** analysis after improving the efficiency of your code.

Big-O measures how well an algorithm will "**scale**" when you increase the amount of "**things**" it operates on. Those "**things**" can be members of an array, entries of a dictionary, nodes of a graph, characters in a String... and the like, depending on the algorithm in question.

Big-O can be used to describe how fast an algorithm will run, or it can describe other behaviors such as how much memory an algorithm will use. Most of the time, the interviewer will be interested in the former (*i.e., runtime complexity of your code*) when she asks for a **Big-O**.

There's no mechanical approach to the **Big-O** analysis of a piece of code. The most thorough explanation I have seen to this date is this StackOverflow thread given by Alejandro Santos.

Another thing you need to mention during a **Big-O** analysis is amortized time.

Let me try to explain the idea in simpler terms:

If you do an operation say a million times, you don't really care about the worst-case or the best-case of that operation. What you care about is how much time is taken **in total** when you repeat the operation a million times.

So it doesn't matter if the operation is very slow occasionally, as long as that "occasion" is rare enough to be neglected.

Amortized time doesn't have to be constant; you can have **linear** and **logarithmic** "*amortized*" time... etc.

Big-O Explained for the Eight-Year Old

For a "*beginner's guide*" with examples you can also read Rob Bell's blog post on the subject.

Although there's a huge list of complexity classes, for practical purposes, the list below is mostly sufficient for almost all **Big-O** analyses you'll do in interviews.

O(1)

This is "**constant time**". Time required is independent of the input size.
Hash lookups are generally considered O(1).

O(log(N))

That's "**logarithmic time**". It gets slower as the input size grows, but once the input size is large enough, there won't be enough change to worry about. For every element, you're doing something that only needs to look at **log N** of the elements. This is usually because you know something about the elements that lets you make an **efficient choice**.

O(N)

This is "**linear time**". So ten times more input will roughly take ten times longer to complete.

O(N log(N))

"**Better than quadratic time**". Increasing the input size hurts, but it's still manageable.
It's okay if the input size increases a little bit. Just watch out for multiples, since it performs worse than **O(N)**.

Most efficient sorts are an example of this, such as merge sort.

O(N^2)

"**Quadratic time**". For every element, you're doing something with every other element, such as comparing them.
Bubble sort is an example of this.

O(N!)

Do something for all possible permutations of the **N** elements.
Traveling salesman is an example of this.

How to Carry Out a Big-O Analysis

In **Big-O** analysis, the most important thing is to determine what **N** (*i.e., the input size*) is:
It can be **the items in an array**, **the letters in a String**, **the number of keys in a Dictionary,** and the like.

After deciding **what N is**, the next step is to determine **how many times the N input items are examined in terms of N**.

In an interview, generally, it will be pretty straightforward to figure this out.

Then you **eliminate all higher order terms**, and **remove all constant factors**.

Remember; when you are asked to comment on the performance of a solution, the interviewer is generally asking for a **Big-O runtime analysis**. Algorithms that run in **linear** or **constant** time are *preferred*.

If a solution you implement appears to run in **quadratic time**, and the interviewer asks you to optimize it, it's generally possible to make it run in **linear time**, by introducing some **extra memory** by creating a **cache** struct, or a **lookup table**. (*note that dictionary lookup operations are considered to be constant time, and an initial loop to prepare a storage can be a negligible cost, if the size of the sample space increases*).

Big-O Is **not** a Silver Bullet

In this section we'll discuss why using **Big-O** only is **not adequate** in real-life scenarios. – In most of the interviews you won't be asked about specific drawbacks and limitations of **Big-O**. All your interviewer wants to know will be a quick and dirty *Big-O runtime analysis,* as it will show you know the concept. However, it's always good to know practical caveats of using the Big-O notation. Who knows what an interviewer reading this book will ask you in your next interview?

At its heart, **Big-O** assumes that "*more work*" means "*longer execution*"; and you should know the limitations, and the dangers of solely relying on the **Big-O** notation. **Big-O** gives us a **tool** to *quickly assess* an algorithms' efficiency and execution time without having to write the code and do experiments. Remember, **speed** and **accuracy** do not always go hand in hand.

Don't fall into the seductive trap of optimization without measurement.

The **additive** assumptions of **Big-O** (*i.e., more work means longer execution time*) was valid until early 1990s when computers did most of the work "sequentially". However, in the last two decades, Moore's law has allowed us to build multicore machines that are far more complex; and in the recent years, the sole focus of computer scientists has been to increase "latency hiding" (*i.e., modern machine architecture is designed to minimize stalls for long latency operations. So the length of the operation, and the parallelization of the operation has an impact on the overall efficiency*).

You may say this issue is *tangentially related* to a *Big-O runtime analysis*, since *architectural complexity* has nothing to do with the *asymptotic behavior* of an algorithm.

That's not the only issue, however. The running time of an algorithm will change based on the size of the input.
Big-O doesn't tell us much about *actual* runtime relative to other algorithms, or even relative to itself.

For example, **within a certain input range**, an O(N^2) algorithm which uses an array is very likely going to be faster than an O(N) algorithm which uses a linked list. The reason is that each operation in an array is way faster than the same operation in a linked list.

Another example: If we have one algorithm that is O(N^2) and one that is O(N log N), the second one may actually be *slower* for the input we are using, but it will scale better as the size of the input increases (*i.e., there exists some* **C** *where "if* **N > C***" the second algorithm is faster*).

Moreover, **Big-O** has to be used carefully for **parallelism**. **Shared memory**, **locks**, **pipes**, **messages**... all have a (*not necessarily constant*) cost, and the most efficient algorithms will be those which take these costs into account in exactly the right way, and maybe in a **counterintuitive** way: For example, in some cases it is faster to recompute a structure than to fetch it from wherever it is stored; which certainly goes against what we would usually do, that is, *use storage rather than computation to speed things up*.

Another example, not exactly related to parallelism, is **memory-hierarchy friendly algorithms**; algorithms that explicitly exploit spatial and temporal locality for (*sometimes major*) speedups. The model assumes memory access is **not** O(1), replacing memory accesses with different costs, *not always constant*, depending on which level you read from or write to.

While I agree that these may be too much of a detail to be asked in an interview, I want to communicate that what's done with the **Big-O** analysis can be roughly summarized as simplifying a complex problem, by disregarding some of the implementation details that make it hard to compute. That's somewhat similar to model-driven architecture, where you abstract representations of knowledge and activities to make it easier to tackle with.

Abstractions are everywhere in an engineer's life. Indeed, what an Engineer does most of the time is to **divide and conquer** complex domain problems into smaller and easier to manage sub-problems. As an engineer you'll sometimes need to disregard harder-to-control specifics of a problem to come up with a solution within given budget, timing, and accuracy requirements.

Chapter 3
Warm-Up Questions

Interviewing (*in the Silicon Valley*) is a **multifaceted** process. Starting from your job application, to receiving an offer you will have a sequence of interviews:

- ★ First you'll have **initial screening** phone interviews;
- ★ Then you may have a couple of **video interviews**;
- ★ Then a group of **intermediate interviews** where you'll most probably be writing code on etherpad;
- ★ Then **serial on site interviews** where you will interview with several people **during the day**;
- ★ And after that, you may have a **simulation interview**: a hands on coding session using some of the company's codebase; trying to solve a similar challenge that you might face in a typical workday in the company.

The highest number of rejections happen **in the first few steps**.

The first steps focus more on evaluating the applicant's **interest** in the company and her motives as well as her overall passion to be employed in the company. The middle steps are designed to evaluate the abstract thinking abilities, problem solving skills, and overall domain knowledge; and the latest steps are more specialized for the position that an applicant has applied for, to make sure about the abilities of the applicant to fulfill the requirements they have in mind.

Depending on your proximity to the company, you may be doing some of those interviews over video conferencing software (*like Skype*); sharing your computer screen and development environment.

Every step in the sequence, will be **harder** than the previous one, and it will generally **take longer** than the previous one.

Acing one interview is **not enough**.

Unless you **ace all interviews**, you won't get an offer.

It's a tiring and stressful process.

And you have to **show "your best"** in each and every single one of those interviews.

The questions in this section are the ones that are generally asked in the beginning phase of this interviewing hassle. They will be asked either on phone screens, or on the technical interview that immediately follows the phone screen.

These will be easy questions, that you can do in less than a few minutes. If the candidate (*i.e., you*) is unable to answer those questions, than the interviewing process will end there.

The format of the questions in this section, and in the proceeding sections, will be as follows:

★ At each page a question will be asked and the remaining of the page will be left blank.

★ And on the following page you'll see the answer to that question.

This will enable you to **work on the question on your own** before looking at the answer on the next page.

A Quick Reminder Before You Dive In

When this book was written, *EcmaScript 6 (ES6)* was just being standardized; therefore the majority of the questions assume an older version of JavaScript (*i.e., ES5)* where things like the const keyword, arrow functions, destructuring assignment, classes, generators, template literals, lexical scoping, and many other features do not exist.

When answering your questions keep that in mind.

For instance while ES5 does not have a **`const`** keyword, ES6 has.

So, **do your homework** and ask yourself whether there is a simpler/better/different answer to the question than the one you see in the text where you can utilize a more modern version of JavaScript. – Also think about the pros and cons of doing so (*in terms of performance issues? backwards compatibility, standardization, ease-of-use, etc.*)

I'm working on a second edition of this book where I'll update the answers and add more questions in these particular areas. – That being said, by the time of this writing, I am swamped with many other projects. So I'm not sure when (*and even if*) I will complete this next edition.

If you want to get notified about the next version, **please send me an email** so that I can inform you when it's done.

How do you declare constants and enums in JavaScript?

Answer:

That's a tricky question. A **constant** is a variable that cannot change. Assigning a value to a constant variable will throw an exception. An **enum**, is a datatype that you can enumerate in a **for each** loop.

JavaScript neither has **constant**, nor **enum** datatypes implemented in it's current version[1]. After telling this, you may ask further clarification from the interviewer, and ask what he exactly means by **constant**, and **enum**?

In Java constants and enums are written in **ALL_CAPS** by convention. So you can take a similar approach in defining constants and enums:

```
// 0001_constants_and_enums.js

// Global Constant
var CLIENT_HOST = 'http://o2js.com/';

// Global Enum
var Screen = {
    SMALL: 0,
    TABLET: 1,
    DESKTOP: 2,
    WIDE: 3
};

var User = {

    // Class Constant
    DATA_SET: 'users',

    // Class Enum
    type: { GUEST: 1, EDITOR: 2, ADMIN: 3 }
};

var key = null;

alert(CLIENT_HOST);

for (key in Screen) {
    if (Screen.hasOwnProperty(key)) {alert(key + ' ' + Screen[key]);}
}

alert(User.DATA_SET);

for (key in User.type) {
    if (User.type.hasOwnProperty(key)) {alert(key + ' ' + User.type[key]);}
}
```

That is, if a variable is written in **`ALL_CAPS`**, you assume it's a constant by contract, and you don't modify it.

---

[1] This will change in the near future. Note that you can freeze objects as of **JavaScript** 1.8, and you can define constants in EcmaScript 6.

What are the different ways you can define "classes" in JavaScript?

Answer:

This is also a tricky question. **JavaScript** uses prototypal inheritance.

Strictly speaking, you don't have classes in the current implementation of **JavaScript**[2].

You can create **class-like structures**, however; and you can **construct objects** with the new operator.

Assuming that's what the interviewer is asking you about, here are different ways to initialize objects in **JavaScript**:

Create a Singleton/Namespace Using an Object Literal

```
// 0002_singleton.js

var app = {
    config: {
        constants: {
            ProviderType: {
                TWITTER: 1,
                FACEBOOK: 2,
                IMAP: 3
            }
        }
    }
};

alert(JSON.stringify(app));
```

Create a Namespace Using Object() Constructor

We can create a similar structure by using `new Object();` instead of an object literal:

```
// 0003_singleton_object.js

var app2 = new Object();
app2.config = new Object();
app2.config.constants = new Object();
app2.config.constants.ProviderType = new Object();
app2.config.constants.ProviderType.TWITTER = 1;
app2.config.constants.ProviderType.FACEBOOK = 2;
app2.config.constants.ProviderType.IMAP = 3;

alert(JSON.stringify(app2));
```

This code does exactly the same thing as the former one. However, the object literal notation is more readable, aesthetically pleasing, and is relatively faster to execute (*also see this performance test, for a more extensive comparison*).

---

[2] Note that, by the time of this writing, there has been an avid discussion going on how to implement classes in EcmaScript 6. So we will see more formal classical behavior in the future versions of **JavaScript**, without needing to write any boilerplate object oriented code.

Aside

As we are on the subject of performance, I'd like to add a side note:

Most of the benchmarks and performance tests rely on JavaScript timers; and often times they behave strangely, because of the single thread which they are in. Additionally, any task that takes **less than fifteen milliseconds** is highly prone to errors. **JavaScript time is not accurate**, and should be repeated **hundreds of times** to reduce the error rate to bearable margins. Even worse, that's an operating-system-level limitation in windows-based systems. To be precise: "*The default timer resolution on Windows is* **15.6 milliseconds**". That is to say, the internal representation of the clock of the **JavaScript** engine is updated every 15.6 milliseconds only. Therefore **any action that takes less than 15.6 milliseconds will be rounded down to zero**.*You would need to have tests running for, at least, 750ms before you could safely reduce the error overhead of the browser to 1%.*

We'll cover more performance topics and questions in the following sections. Now back to our question:

Using "Object.create()" to Create Objects

For modern browsers, we can also use Object.create to create objects.

```
// 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);}
    }
});

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

Using "Constructor Functions" to Initialize Objects

The below is just a quick example of so called "Object Oriented" JavaScript:

```
// 0005_constructor_functions.js

var instanceCount = 0;

function BaseProvider() {
    instanceCount++;
}

BaseProvider.prototype.count = function() {
    return instanceCount;
}

function TwitterProvider(username) {
    this.username = username;

    var dummy = 20;

    function privateMethod() {
        alert(dummy);

        dummy--;
    }

    this.privilegedMethod = function() {
        privateMethod();

        return dummy > 18;
    };
}

TwitterProvider.prototype = new BaseProvider();
TwitterProvider.constructor = TwitterProvider;

// ref: https://dev.twitter.com/docs/api/1.1
TwitterProvider.prototype.tweet = function(message) {
    var kApiEndpoint = 'https://api.twitter.com/1.1/statuses/update.json',
        data = {url: kApiEndpoint, status: message};

    // Assuming we are using a library to do AJAX POST requests.
    ajax.post(data);
};

var volkan = new TwitterProvider('@linkibol');

volkan.tweet();

// This will alert "true".
alert(volkan.privilegedMethod());

// This will alert "false".
alert(volkan.privilegedMethod());
```

"Constructor Function" Returns an "Object"

When you read EcmaScript Language specification (*section 13.2.2*) regarding the **new operator** carefully, you'll see that a constructor can indeed return an object. Here's how that section is formally defined:

When the **[[Construct]]** property for a **Function** object **F** is called, the following steps are taken:

1. Create a new native ECMAScript **object**.
2. Set the **[[Class]]** property of **Result(1)** to "**Object**".
3. Get the value of the prototype property of **F**.
4. If **Result(3)** is an object, set the **[[Prototype]]** property of **Result(1)** to **Result(3)**.
5. If **Result(3)** is not an object, set the **[[Prototype]]** property of **Result(1)** to the original **Object** prototype object as described in **15.2.3.1**.
6. Invoke the **[[Call]]** property of **F**, providing **Result(1)** as the **this** value and providing the argument list passed into **[[Construct]]** as the argument values.
7. If **Type(Result(6))** is **Object** then return **Result(6)**.
8. Return **Result(1)**.

In human-readable terms: that if a constructor returns an Object[3], then that object will be used; if the constructor function returns a non-object value (like `null`, undefined, `"hello world"`, `42`...) a new object is returned.

On the other hand, returning any *non-object* from the constructor is practically the same as exiting the constructor. (The constructor will return a new object with a **prototype** if one was specified.)

[3] *An* **object** *in* **JavaScript** *is any collection of key-value pairs. If it's not a primitive (undefined, null, boolean, number, string) it's an* **object.**

The following is an example of a constructor that returns an object:

```
// 0006_constructor_function_returns_object.js

var Account = function(accountNumber) {
    var num = accountNumber;

    return {
        getAccountNumber: function() {
            return num;
        },
        setAccountNumber: function(newNumber) {
            num = newNumber;
        }
    }
}

var bar = new Account('42');

alert(bar.getAccountNumber());
```

Using Immediately Invoked Function Expressions with the new Operator

Albeit rarely used, that's also a way to create objects. Here's an example:

```
// 0007_singleton_object.js

var user1 = (new function(id, name) {
    this.id = id;
    this.name = name;
    this.toString = function() {
        return this.id + ' / ' + this.name;
    };
}('vvic', 'Vector Victor'));

// This will alert "vvic / Vector Victor".
alert(user1);

var user2 = (new function() {
    this.id = 'jdoe';
    this.name = 'John, Doe';
    this.toString = function() {
        return this.id + ' / ' + this.name;
    };
}());

// This will alert "jdoe / John, Doe".
alert(user2);
```

How do you create private variables and methods in JavaScript?

Answer:

Since there's no access modifier in **JavaScript**[4], the best definition we can rely on is **Douglas Crockford**'s terminology:

> **Public Properties**: Any data stored on the instance.
>
> **Private Properties**: Data is stored in the environment is only accessible to the constructor and functions created inside the constructor.
>
> **Privileged Methods**: Functions created inside the constructor, and are added to the instance.

We'll explore some of these usages in the following pages.

[4] By the time of this writing, there is a discussion around how to implement access modifiers in EcmaScript 6.

Use Private Methods Inside Constructor

Here's an example of using **private members** and **privileged methods** inside a **constructor**:

```
// 0008_private_methods_inside_constructor.js

function MyLovelyConstructor(argument1, argument2) {

    // Public members.
    this.publicMember1 = argument1;
    this.publicMember2 = argument2;

    // This is a common pattern to encapsulate self reference.
    var that = this;

    // This is a private member.
    var privateValue = 42;

    // This is a private method.
    function privateMethod(newValue) {

        // This will refer to the global object [window] or
        // undefined (for strict mode) - So we use "that" instead.
        privateValue = newValue;

        that.publicMember1++;
        that.publicMember2++;

        that.publicMethod();
    }

    privateMethod(argument2);

    // This method is public. And it can access to the private
    // members of the `MyLovelyConstructor` object.
    // One disadvantage of "privileged methods" is that they
    // have to be assigned whenever the object is constructed.
    this.privilegedMethod = function() {alert(privateValue);};
}

// This method is bound to the prototype chain of the object,
// and assigned "only once". This is better in terms of performance
// than using a privileged method and initializing it at every constructor call.
MyLovelyConstructor.prototype.publicMethod = function() {alert('geronimo!');}

var test = new MyLovelyConstructor(10, 20);
```

Using Immediately Invoked Function Expressions

Another approach is exporting a **module** as a return value of an **IIFE**:

```
// 0009_iife.js

var app = (function() {
    var privateValue = 42;

    var exports = {};

    function privateFunction() {
        exports.publicMember++;
        exports.publicMethod();
    }

    exports = {
        publicMember: 42,
        publicMethod: function() {
            alert('hi');
        },
        callPrivate: function() {
            privateFunction();
        }
    };

    return exports;
}());
```

In the above code, **`privateValue`** and **`privateFunction`** are only accessible inside the **IIFE** closure's scope.

Aside

Interestingly, in some user agents even the **function scope** doesn't provide 100% privacy.

Which proves nothing but **eval is evil**.

Hyper-Private Variables

You can even restrict the scope of private variables, using what Peter calls "hyper private variables":

```
// 0010_hyper_private_variables.js

function Person(first, age) {

    // Private methods.
    var getFirstName, setFirstName, getAge, setAge;

    (function() {

        // `_first` is a hyper-private variable.
        // `_first` is only visible to getFirstName and setFirstName.
        var _first;

        getFirstName = function() {

           // Capitalize the name.
           return _first.charAt(0).toUpperCase() + _first.substr(1);
        };

        setFirstName = function(v) {_first = v;};
    })();

    (function() {

        // `_year` is a hyper-private variable.
        // `_year` is only visible to `getAge` and `setAge`.
        //
        // Store birth year instead of age so that `getAge` always returns current age.
        var _year;

        getAge = function() {return (new Date()).getYear() - _year;};

        setAge = function(v) {
            if (v < 0) {throw new Error('Negative ages are not allowed!');}
            _year = (new Date()).getYear() - v;
        };
    })();

    // `sayInfo` is public method.
    // Cannot access `_first` or `_age` directly.
    // So we must use the getters.
    this.sayInfo = function() {
        alert(getFirstName() + ' ' + getAge());
    };

    // Cannot access `_first` or `_age` directly.
    // So we must use the setters.
    setFirstName(first);
    setAge(age);
}

var gisele = new Person('Gisele', 27);

gisele.sayInfo();
```

Using Immediately Invoked Function Expressions to Enclose a Single Method

Or you can use an **IIFE** for a single method:

```
// 0011_iife_enclosure.js

var app = {
    Helper: {
        generateFoo: (function() {
            var methodPrivateData = 42;

            return function() {
                return methodPrivateData * Math.random();
            };
        }()),

        executeBar: function() {

            // Do stuff.
        }
    }
};
```

Having Fun with "Isolated Objects"

If you get comfortable enough with those ideas, you can make a mixture of those, along with some reflection as in my humble isolated objects sample:

```
// 0012_isolated_objects.js

//platform.js
var platform = {};
(function(platform) {
    var players = {}, totalPlayerCount = 0;

    // The `Player` class is only visible to this module,
    // with an exception (see [1]).
    function Player(name) {this.id = totalPlayerCount++;this.name = name;this.score = 0;}

    Player.prototype.incrementScore = function() {this.score++;};

    platform.GameController = {
        addPlayer: function(name) {
            if(!name) {return;}if(players[name]) {return;}
            players[name] = new Player(name);
        },
        incrementScore: function(name) {
            if(!name) {return;}

            var player = players[name]; if(!player) {return;}

            player.incrementScore();

            if(typeof successCallback !== 'function') {return;}

            // [1]
            // We dynamically bind the player instance
            // to the context of the delegate.
            successCallback.apply(player, []);
        }
    };
}(platform));

// consumer.js
(function(platform){
    var controller = platform.GameController;
    var kRustu = 'Rüştü Rençber', kGokhan = 'Gökhan Zan';

    controller.addPlayer(kRustu);controller.addPlayer(kGokhan);

    controller.incrementScore(kRustu, function() {

        // `this` refers to the Player object. But you can only access it here.
        // That is to say, access to the `Player` instance
        // is "isolated" to this callback only.
        alert(this.id);    // Will alert '0'
        alert(this.name);  // Will alert 'Rüştü Rençber'
        alert(this.score); // Will alert '1'
    });
}(platform));
```

A real-life application of "Isolated Objects" pattern may be found at 02.unit.core module of **02.js** JavaScript Framework.

Or you can have a naming convention for private members just like we did in the "constants question".

```
// 0013_private_naming_convention.js

var app = {
    String: function() {

        // underscore ( _ ) means that it's intended for private access.
        this._buffer = [];

        var i = 0,
            len = 0,
            buf = this._buffer;

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

app.String.prototype.toString = function() {
    return this._buffer.join('');
};

app.String.prototype.concat = function() {
    this._buffer = this._buffer.concat(
        Array.prototype.slice.call(arguments)
    );

    return this.toString();
};

var phrase = new app.String('Hello ', 'world.');

// Hello world.
alert(phrase);

phrase.concat(' Hello ', 'stars; ', 'Hello', ' universe!');

// Hello world. Hello stars; Hello universe!
alert(phrase);
```

Aside

Don't Abuse the Use of Private Variables

Sometimes trying to encapsulate every member as private variables, and creating getters and setters for each of those variables makes things more complicated than necessary. If there is a **getter**, and a **setter** for that variable, why not make that variable public in the first place?!
JavaScript is not Java. Embrace it's flexibility.

Take the following example:

```
// 0014_aside_widget.js

var Widget = function() {
    var borderWidth, width;

    this.getWidth = function() {return width + borderWidth*2;}
    this.setWidth = function(w) {width = w;}
    this.getBorderWidth = function() {return borderWidth;}
    this.setBorderWidth = function(b) {borderWidth = b;}
    this.getRealWidth = function() {return width;}
};

var widget = new Widget();

widget.setWidth(10);
widget.setBorderWidth(20);

alert(widget.getWidth());
```

The above code does nothing but protect **you** from... **yourself**. Abusing the use of private variables indeed makes code **less readable** and **harder to maintain**.

Remember **Occam's razor**?: If there's a **simpler** approach, probably it's **better**:

```
// 0015_aside_widget_simplified.js

var Widget = function(params) {
    this.borderWidth = params.borderWidth || 0;
    this.width = params.width || 0;
};

Widget.prototype.getWidth = function() {
    return this.width + this.borderWidth * 2;
};

var widget = new Widget({width: 10, borderWidth: 20});

// Do stuff with `widget`.
```

What does the following code do?

```
// 0016_question_set.js

var a = [1, 2, 3, 4];
var b = [3, 4, 5];
var c = [3, 4, 5];

var set = {add: function(item) {this[item] = item;}};

set.add(a);
set.add(b);
set.add(c);
```

Answer:

The question seems trivial. Don't be deceived by its simplicity, however.

The question measures whether you know what a set data structure is; and whether you know how keys in **JavaScript** objects work at a deeper level.

According to ECMA 262, Section 11.2.1, keys in **JavaScript** objects can only be **`String`**s.
If we assign a non-string value as a key to an object, it will be **coerced** to a **`String`**.

What this means for our example is:

```
this[item] = item;
```

will coerce item to a **`String`** (*`item.toString()`*, to be specific) and this string representation will be used as the key.

So,

```
alert(set['1,2,3,4'] === a);
```

will alert **`true`**.

To analyze it further you can use something similar the following loop:

```
// 0017_set_dump.js

for (var key in set) {
    alert(key);
    alert(typeof key);
    alert(set[key]);
    alert(set[key] === a);
    alert(set[key] === b);
    alert(set[key] === c);
}
```

Note that in our example, **`b`** and **`c`** are different objects.

```
alert(b === c);
```

will alert **`false`**.

You see how easily you can implement a **set** data structure in **JavaScript**.

Just use an object literal and voila! you are all **set** (*pun intended*).

Let's dive a little deeper, and write a **count** method for our sets, to get the total number of items of our **set**:

```
// 0018_set_count.js

var set = {
    add: function(item) {this[item] = item;},
    count: function() {
        var counter = 0,
            key;

        for (key in this) {
            if (this.hasOwnProperty(key)) {
                if (typeof this[key] === 'object') {
                    counter++;
                }

            }
        }

        return counter;
    }
};
```

> There are more efficient ways of writing the **`count`** method; like checking for the existence of **`item`** key in the **`add`** method, and increasing a **`_count`** member of the set if what we are adding has not been added before. This way, the **`count`** method will simply return **`_count`** instead of iterating through all the elements of the set. I'm leaving this as an exercise to the reader.

Note that

```
alert(set.count());
```

will alert **`2`**.

Why? Because the **String** representations of **`b`** and **`c`** are the same (*i.e., "2,3,4"*), and only the last added item will be stored as a member of **`set`**.

For further proof:

```
alert(set[b] === c);
```

will alert **`"true"`**, because the object **`b`** will be **coerced** to **b.toString()** before being evaluated as a key of **`set`**.

Isn't **JavaScript** an amusing language?

What's wrong with the following code?

```
// 0019_question_samurai.js

this.jsiq = {};

(function(jsiq) {
    if (!jsiq) {throw 'root namespace is not defined';}

    // A private variable.
    var name = '';

    function Samurai(theName) {name = theName;}
    Samurai.prototype.getName = function() {return name;};

    // Exports.
    jsiq.Samurai = Samurai;
})(this.jsiq);

var akamatsu = new jsiq.Samurai('Akamatsu Mitsusuke'),
    anayama = new jsiq.Samurai('Anayama Beisetsu');
```

Answer:

What's expected from the above code sample is to create different **`Samurai`**s with different names.
However, this will not be the case, since the **`name`** variable will be statically shared among all the **`Samurai`** instances.

We can test this with the following code:

```
// 0020_samurai_test.js

alert(akamatsu.getName()); // will alert "Anayama Beisetsu".
alert(anayama.getName());  // will alert "Anayama Beisetsu".
```

You may think that sharing a static context is a drawback for the module pattern as demonstrated by the above usage. However, it's actually a **misuse** of the pattern; or we can see this as an **antipattern**.

In software engineering, an **anti-pattern** (or *antipattern*) is a pattern that may be commonly used but is **ineffective** and/or **counterproductive** in practice.

Why? Because the core idea behind **modules** is to **implement namespaces**. Namespaces are **unique**, so modules **should** behave similar to **singletons**. Unlike what it's meant for, if the module code is used as a **factory method** (*as in here*) then two distinct ideas are being conflated, which will result in the above side effects.

Once you learn the **module pattern**, you may think that it's a hammer for every possible nail.
It's not always the case.

Let's fix the code to make it work as expected:

```
// 0021_samurai_fixed.js

this.jsiq = {};

(function(jsiq) {
    if (!jsiq) {throw 'root namespace not defined';}
    jsiq.Samurai = function(name) {this.name = name;};
})(this.jsiq);

var akamatsu = new jsiq.Samurai('Akamatsu Mitsusuke'),
    anayama = new jsiq.Samurai('Anayama Beisetsu');

alert(akamatsu.name); // will alert "Akamatsu Mitsusuke".
alert(anayama.name);  // will alert "Anayama Beisetsu".
```

Simplicity is better, isn't it?

What does the following code try to achieve?

```
// 0022_lock_unlock.js

this.jsiq = (function(me) {
    var privates = me.privates = me.privates || {};

    var lockPrivates = me.lockPrivates || function() {
        delete me.privates;
        delete me.lockPrivates;
        delete me.unlockPrivates;
    };

    var unlockPrivates = me.unlockPrivates || function() {
        me.privates = privates;
        me.lockPrivates = lockPrivates;
        me.unlockPrivates = unlockPrivates;
    };

    return me;
}(this.jsiq || {}));
```

Answer:

The above structure is an example of "**module augmentation**", you can view the **module pattern** section for a variety of different example usages of the **module pattern**.

One usage of the **module pattern** is that it gives you the ability to split your codebase into logically coherent "**modules**". One limitation of this approach is, though, each file maintains its own execution context; therefore each file has its own private state; and the state cannot be shared across the files: You cannot get access to the private state of other files.

When you analyze the above code closely, you can see that it's meant to alleviate this restriction.

A straightforward approach to the problem is to use a `jsiq.privates` namespace to share state across the files. However, this will make the `jsiq.privates` namespace **publicly available** to **everyone**, so there won't be any privacy at all.

If you ask me, that's no big deal unless you know that **privates** namespace is meant to be used "*privately*" within the modules; and trying to protect you from yourself overcomplicates things. Yet, our beloved interviewer thinks that that's not true.

In the above code the `lockPrivates` and `unlockPrivates` methods are used to fix this public access issue.
Here's how:

Unless the `lockPrivates` method is called (*possibly after we're sure that all the modules have been initialized successfully*) any file can access the `jsiq.privates` namespace of the module.

After the `lockPrivates` method is called, the modules can still access the privates through cached `var privates` copies. After calling the method there will be no such `jsiq.privates` namespace, and the privates will not be publicly accessible. Moreover `jsiq.unlockPrivates` will also be hidden, so it's impossible to reveal the privates namespace again from outside the modules.

If we need to grant public access to the privates at a later time (*may be when we want to lazy load another module*), the *cached* `unlockPrivates` method can be called through any function **inside** any of the module closures; and after we're done with our modification, we can all `lockPrivates` again to disable public access to the privates namespace of the module once more.

Analyze the following code in terms of performance and memory utilization.

Can you improve it?

```
// 0023_counter_factory.js

var CounterFactory = {
    create: function() {
        return (function() {
            var me = {
                count: 0,
                max: 5,
                isPastMax: function() {
                    return this.count > this.max;
                }
            };

            return {
                count: function() {return me.count;},
                reset: function() {me.count = 0;},
                increment: function() {
                    me.count++;

                    if (me.isPastMax()) {this.reset();}
                }
            };
        }());
    }
};

var counters = [],
    i;

for (i = 0; i < 10000; i++) {
    counters.push(CounterFactory.create());
}
```

Answer:

If the interviewer asks whether you can improve a piece of code, then the code clearly needs some improvement and refactoring.

Generally what should be done will be **hidden in the question** itself. The question asks about the memory implications; so our focus should be on things that will increase the memory utilization. The piece of code that obviously consume memory, is this **for** loop:

```
for (i = 0; i < 10000; i++) {
    counters.push(CounterFactory.create());
}
```

An obvious thing to do to decrease memory consumption is to decrease the number of iterations in the loop. If the question was that easy, the interviewer would not have asked it in the first place.

So let's focus on the **`CounterFactory`**'s object creation:

```
var CounterFactory = {
    create: function() {
        return (function() {
            ...
            return {
                ...
            };
        }());
    }
};
```

We have **10,000** instances of the Counter **module**! This provides us **encapsulation**; and this encapsulation comes with a price, because each **closure** comes with its own **execution context**. Therefore every method and attribute defined for every counter instance we create separate instances. This is bad for memory utilization.

Sometimes this is acceptable, if the number of instances to be created is low, and the module's memory footprint is small.

> If you've got a module that goes on for a few hundreds of lines, and you intend to make a few hundred instances, expect it to be eating up a **substantial** amount of memory.

Yes, we've successfully identified the problem. Now it's time to propose a solution. In this case, the ideal solution is to get rid of the module, and use pseudo-private variables instead, as follows:

```
// 0024_counter_factory_improved.js

var Counter = {
    _count: 0,
    _max: 5,
    _isPastMax: function() {return this._count > this._max;},

    reset: function() {this._count = 0;},
    increment: function() {this._count++;},
    count: function() {return this._count;}
};

var extend = function(destination, source) {
    if (!!Object.extend) {
        return Object.extend(destination, source);
    }

    var key = null;

    for (key in source) {
        if (source.hasOwnProperty(key)) {

            // This is a "by ref" assignment.
            destination[key] = source[key];
        }
    }
}

var counters = [],
    i;

for (i=0; i<10000; i++) {
    counters.push(extend({}, Counter));
}
```

This has some advantages over the previous one:

Since the assignment of the functions are by ref, only references to functions are passed to each object unlike the former one (where a separate function instance was created for every method of every Counter instance). Furthermore we've gotten rid of the closures, and it will also decrease the memory utilization.

What else can we do? Let's see:

```
// 0025_counter_factory_improved_simplified.js

function Counter() {
    this._count = 0;
    this._max = 5;
};

Counter.prototype = {
    _isPastMax: function() {return this._count > this._max;},

    increment: function() {this._count++;},
    reset: function() {this._count = 0;},
    count: function() {return this._count;}
};

var counters = [],
    i = 0;

for (i=0; i<10000; i++) {
    counters.push(new Counter());
}
```

This is also equally memory-efficient, **simpler**, and **better**.

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*[5].
   So **instruction efficiency** and **scope conservation** are important for faster execution.

2) **JavaScript** is a *lambda language*[6].
   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**.

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

[6] 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 **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

Here's the definition from Wikipedia:

> In **software engineering** (or computer science), a **design pattern** is a general repeatable solution to a commonly occurring problem in **software design**. A design pattern is not a finished design that can be transformed directly into **code**. It is a description or template for how to solve a problem that can be used in many different situations. **Algorithms** are not thought of as design patterns, since they solve **computational** problems rather than **design** problems.

Knowing design patterns and knowing the elements of a design pattern are very important. Needless to say, during your interviews, you may have one or more pattern-related questions. The interviewer may either ask you to list a couple of your favorite design patterns, and how you implement them in your projects; or she will name a specific pattern and want you to discuss its *typical implementations*.

> Even if you're not directly asked about software design patterns, it'll be wise to show your knowledge of patterns wherever *appropriate*; "appropriate" being the keyword:
> Trying to *show off* without knowing the pattern's **context**, **intent**, **motivation of use**, **structure** (*UML Diagrams*), **consequences** (*i..e. benefits and liabilities*), specific **implementations**, known **issues**, typical **usage examples**, **co-requisites** (*i.e., other patterns that this pattern utilizes*), and **related patterns** to the pattern you're talking about... will be your highway to *"we will call you later" hell* -- if you know what I mean.

So simply knowing the name of the pattern will not take you anywhere.
You also have to know about the structure of the design pattern.

Aside

When it comes to patterns, the Gang of Four (or **GoF***) patterns are generally considered the foundation for all other patterns.

They are categorized in three groups:

Creational, **Structural**, and **Behavioral**.

Design Patterns are **language-agnostic**. That is, you can implement a software design pattern in any programming language. However, some of these patterns are more applicable to **JavaScript** than others.

** The authors of the "Design Patterns: Elements of Reusable Object-Oriented Software" came to be known as the "Gang of Four." – The name of the book ("Design Patterns: Elements of Reusable Object-Oriented Software") was too long for e-mail, so "the book by the GoF" became a shorthand name for it. After all, it isn't the ONLY book on patterns. That long book name got shortened to "GOF book", which is pretty cryptic the first time you hear it.*

Each pattern typically has it's own **benefits**, **liabilities**, and known **issues**; and, most of the time each pattern has a group of variants.

Take **MVC**, for example. It has many **MV*** variants: **MVP**, **MVVM**, **MVA**, **Passive View**, and a dozen of others are approximately the same thing, tweaked for different requirements. Fowler's GUI Architectures is a nice work summarizing the need for **MVC** and its evolution to **MVP**. Also, Oliver Steele's Web MVC discusses how **MVC** started from the desktop, to server-side **MVC**, and then transferred some of the responsibility to the client; evolving into the current **fat-client** single-page **RIA** architectures.

Hint:
If an interviewer asks you what your favorite pattern is, and your answer is **Singleton**, your chances of getting the job decrease by **30%**. If you cannot name any other pattern other than Singleton, afterwards, your chances decrease by another **30%**.
True story.

The evolution and diversification of a given pattern is always due to the requirement to better fit it to different problem domains: **One architecture cannot fit them all**.

> Using a software design pattern, without fully understanding it's **benefits**, **liabilities**, and implementation **complexity** is akin to giving a gang of monkeys a box of hand grenades.
> The result will be "**destructive**" (*yet amusing*).

Even if you know that nobody will be asking you about patterns (*and you'll never know*), it's always a good idea to know about typical **JavaScript** design patterns. Even if you don't realize, you will be using them in your work all the time.

And once you dive into the ocean of patterns, it's huge! Pattern Oriented Software Architecture (**POSA**) is a very diverse topic to be covered in a single book. Indeed, there are several volumes of books written about it.

Here, we will be looking at a group of patterns that are widely used in **JavaScript Engineering**, and **Large-Scale JavaScript Application Development**.

In this chapter we will review design patterns as they relate to **JavaScript**, starting from the most frequently used to least frequently used ones. Note that this is just a *limited set* of the most commonly used patterns.

For a wider collection of **JavaScript** patterns, you can look at Shi Chuan's compilation, and for ajax-specific patterns you may want to give ajax patterns a go. Addy Osmani summarizes a bunch of essential design patterns too. Also, Stoyan has a very nice book about **JavaScript** patterns.

You'll find yourself using different combinations of these patterns depending on the nature and requirements of the project you're working on. You may even want to show off your pattern knowledge in the tougher *hands-on coding questions* that you will be asked once you nail the first few initial screening interviews.

In the following sections, we'll be examining each of the patterns in depth. You have seen some tricky interview questions that use important **JavaScript** patterns in the Warm Up Questions section. The following sections about patterns will include fewer interview questions. Instead we will discuss **implementation details**, **variations** and usage **examples** of important patterns.

Continuation-Passing Style

Continuation-Passing Style is a good pattern to start this section with. Because it's a pattern that's woven into **JavaScript**. Most of us use it all the time, without knowing. Moreover, continuations are also the building blocks for **Promises**. Thus before diving into Promises, it's better to have a fair understanding of continuation passing style.

Here's the definition from Wikipedia.

> Continuation-passing style (*CPS*) is a style of programming in which control is passed explicitly as a **continuation**. A function written in **CPS** takes an extra argument: an explicit "continuation" i.e., **a function of one argument**. When the **CPS** function has computed its result value, it "returns" it by calling the continuation function with this value as the argument.

For the interested, a **continuation** is...

> A **continuation** is a data structure that represents the computational process at a given point in the process's execution ... Continuations are useful for encoding other control mechanisms in programming languages such as **exceptions**, **generators**, **coroutines**, and so on.

I don't know you, but to me the book definition of a "**continuation**" feels rather vague and unsatisfying. In my honest opinions, a powerful construct as a continuation requires a solid pedagogical foundation to make them "lovable". We will try to establish this foundation with tips, tricks, and examples.

> **Continuations** are the building blocks of many other patterns, and they are especially useful in functional programming. So learn them well before studying other patterns.

Formal definitions are rigorously written to cover all possible interpretations and to avoid confusion and pitfalls. This makes them often hard to understand without reading multiple times; and when the interviewer asks what "**CPS**" is, she would expect you to answer it **in your own words**. So let us rephrase the above quote so that a five-year-old can understand:

> Continuation-passing style is a programming approach where
>
> 1. No function is allowed to return to its caller, **ever**.
> 2. To achieve this, functions take a "**callback**" to "**invoke**" upon their return value.

If it's still not clear, the following questions will make it crystal-clear.

Write an identity function, using continuation-passing style.

Answer:

An identity function (ref) is simply a function that returns its argument.

```
// 0042_identity_normal.js

function identity(arg) {
    return arg;
}
```

Let's write this function in continuation-passing style:

```
// 0043_identity_cps.js

function identity(arg, delegate) {
    delegate(arg);
}
```

Instead of returning, we **delegate**, **always**. Keep this in mind, and rewriting any function in continuation-passing style would be a cakewalk.

Aside

You may wonder what's the use of an identity function at all: **JavaScript** is a **functional** language. And **functions are first-class citizens** in **JavaScript**. That's why we often pass functions as arguments to other functions. In that regard, the **constant function** (*which always returns the same value for any argument*) and the **identity function** (*which returns its own argument*) play a similar role as in "**zero**" and "**one**" in multiplication, loosely-speaking.

This is particularly useful in constructs like iterators.
Take the following code, for example:

```
// 0044_aside_why_identity.js
function any(items, iterator) {
    iterator = iterator || function(x){return x;};

    var result = false, len = items.length, i;

    for (i = 0, i < len; i++) {
        result = !!iterator.call(context, items[i], i);

        if (result) {return true;}
    }

    return false;
}
```

By using the identity function as a replacement for **`iterator`** when it does not exist, the following two statements would work identically:

```
// 0045_aside_identity_maybe.js

any([false, true, false], function(item){return !!item;});
any([false, true, false]);
```

In short, we simply invoke a method by passing a **continuation**. The **continuation** is nothing but a **function** that takes a single argument, which is the value that we are supposed to return. The method accepting the continuation decides whether and when to call the continuation.

The "will call you maybe" aspect of continuations is what makes them the building blocks of Promises.
The Hollywood-style nature of **CPS** is also useful in distributed computing.

To put it simpler:

If functions are first-class citizens, a continuation is a **first-class return point**.

Write a factorial function in JavaScript, using continuation-passing style.

Answer:

If you go for as many interviews as you can (*and I strongly suggest you do*), eventually someone will ask you to write a factorial function, or a function that calculates the nth number of a Fibonacci sequence, or to create a function to do multiplication by using addition only. These are de-facto "recursion" interview questions.

This question, adds some spice to a boring "recursive Factorial implementation" question, which a veteran candidate might have been asked at least a dozen times (*if not more*). The trick is, the interviewer is asking you to approach the subject matter from a different perspective by combining your knowledge of **recursion** with your knowledge of **CPS**.

> In any "well-designed" interview **always be prepared for surprises**. The interviewer's aim is to ask you out of the ordinary questions to see your critical thinking abilities. Often you will see questions that require you to connect different pieces of the puzzle (*i.e., different patterns and techniques*) to come to a conclusion. – That correlates with what you will be doing in your job too: combining a variety of tools and skills to solve a business problem.

Let's start by a naïve recursive factorial implementation:

```
// xxx_yyy.js


function factorial(n) {
    if (n === 0) {return 1;}

    return n * factorial(n-1);
}
```

This implementation has its own drawbacks. Like calling it with a big number will result in a stack overflow error. Now let's write it in a **tail-call-optimized** fashion:

```
// aaa_bbb.js

function factorial(remaining, accumulator){
    if (remaining === 0) {return accumulator;}

    return factorial(remaining - 1, remaining * accumulator); // <-- tail call
}
```

One **caveat** you should know is that writing it in a "*tail-call-optimized*" manner will not actually optimize anything. I've witnessed the surprise in an interviewer's face when I mentioned her this fact, after doing a tail call optimization for an interview question that she asked.

When dealing with **JavaScript**, do not take anything for granted.
Doubt everything and do your research.

JavaScript does not do tail call optimization (it is a feature yet to be implemented in EcmaScript 6). There are ways to overcome this and make our good-old factorial function run without getting a "*RangeError: Maximum call stack size exceeded*" or similar exceptions, running in constant stack space.

I've written the function in tail form here, because it's generally easier to convert a recursive function to CPS, when it's written in tail form. Here is how we do it:

```
// 0046_factorial_recursive.js

function factorial(n, accumulate) {
    if (n === 0) {
        accumulate(1);

        return;
    }

    factorial(n-1, function(k) {accumulate(n * k);});
}
```

As always, we **do not return**, and we **yield** the continuation to another function instead.

The flavor of the function has changed, but it's still recursive. Here is how the call stack unwinds for **`n = 3`**:

```
//0047_factorial_stack_unwind.js

factorial (3,            function fn3(k){ log(k)   } )
    factorial (2,        function fn2(k){ fn3(3*k) } )
        factorial (1,    function fn1(k){ fn2(2*k) } )
            factorial(0, function fn0(k){ fn1(k)   } )
                fn0(1);
            fn1(1);
        fn2(2*1);
    fn3(3*2*1);
log(3*2*1);
```

When tracking down recursive algorithms, I've found that giving names to each lexical closure (***`fn0, fn2, fn2, fn3`*** *in the above example*) makes it easier to unwrap the stack in my mind. It may be trivial to see how the function operates in this simple factorial implementation. However, things may not be always this simple: Interviewers constantly seek ways to surprise candidates, to filter our their rockstar candidate from the crowd; and you are given a more complex algorithm in an interview, you may find this trick useful. It will also clearly show the interviewer that you *can think recursively*.

Let us use this function in an example. The function below will log **`6`** to the console:

```
// 0048_factorial_cps_example.js

factorial(3, function(n) {
    console.log(n);
});
```

Convert the CPS factorial function so that it executes each iteration asynchronously.

Oh those interviewers! Once you solve something they always ask for something more complicated that you have to build on top of your former solution. All this hassle is to ensure that you are **not** copying an answer that you've memorized a night before the interview (which you should **not!**). The interviewer wants to figure out whether you are able to logically reason and construct complex structures from simpler building blocks. That's what engineering is all about, and that's what you will be doing in your job anyway.

Since factorial function can call the continuation at any time, it is trivial to make things async:

```
// 0049_factorial_async.js

function factorial(n, accumulate) {
    if (n === 0) {
        setTimeout(function() {accumulate(1);}, 1000);

        return;
    }

    setTimeout(function() {
        factorial(n-1, function(k) {accumulate(n * k);});
    }, 2000);
}

factorial(3, function(n) {console.log(n);});
```

This opens many possibilities, which is relatively harder to implement if you take a non-CPS approach. For instance let's assume we have an "echo" web method that does nothing but return the parameter sent (i.e., an **`HTTP GET`** request to **`http://api.foo.bar/echo/10`** displays **`10`** in the response body). Then we can write a totally useless remote factorial that uses this echo web method:

```
// 0050_factorial_rpc.js

function factorial(n, accumulate) {
    if (n === 0) {
        execAjaxAsync('http://api.foo.bar/echo/1', function(res) {
            accumulate(res);
        });

        return;
    }

    execAjaxAsync('http://api.foo.bar/echo/' +(n-1), function(res) {
        accumulate(n*res);
    });
}

factorial(3, function(n) {console.log(n);});
```

Aside

The astute reader might realize the similarities between this approach, and **Node.JS API**.

This is **not** a coincidence. **Node.JS** encourages, **requires** even, that you write all your code in continuation-passing style. The entire **Node.JS** API is written in **CPS**. This is a community-made decision starting from **Node.JS** version 0.1.3. Before that, the API was using Promises. After much back and forth discussions, the community finally settled to go with the simpler **CPS** style and leave promise-related implementation details to the developers.

Simplicity had been **Node.JS**'s core focus from the day it started; and that's a good thing. At the time of this writing, there are still occasional flamewars regarding whether sticking with **CPS** was a good thing for **Node.JS**.

While we're on the subject of Promises, be sure not to miss the point of promises, RTFM, and know that it's not simply a pattern for flattening the callback hell; it's way more than that. What makes the promises pattern shine is how elegantly it handles and propagates edge cases and errors. We'll dive into this topic deeper in the Promises section.

Implement a simple `fetch(url, onSuccess, onFail)` **function that makes an** `XMLHttpRequest`**.** `onSuccess` **should be called upon success, and** `onFail` **should be called upon failure. Do not take cross-browser differences into account.**

Answer:

You may not have realized that all the popular **AJAX** libraries use the **CPS** Pattern when handling **AJAX** requests. Here's a very simple implementation for the above question:

```
// 0051_ajax_cps.js

var noop = function() {};

function fetch(url, onSuccess, onFail) {
    var request = new XMLHttpRequest(),
        successHandler = onSuccess || noop,
        errorHandler = onFail || noop,
        kComplete = 4,
        kSuccess = 200,
        kVerb = 'GET';

    function handleReadyStateChange() {
        var isSuccessful = request.readyState === kComplete &&
                    request.status === kSuccess;

        if (isSuccessful) {
            successHandler(request.responseText, url, request);

            return;
        }

        errorHandler(url, request);
    }

    request.onreadystatechange = handleReadyStateChange;

    request.open(kVerb, url, true);
    request.send(null);

    return request;
}
```

And you have a bare-bones **AJAX** fetcher in just a few lines of code. There's more to do to make the code reusable; like handling cross-browser implementation differences, handling **`POST`** and other type of requests (i.e., **`DELETE`**, **`PUT`**...), caching, enabling sync requests... These may be the follow-up questions that the interviewer will ask once you complete your implementation.

Assume there exists a factorial server at `http://example.com/factorial/n` **that displays the "factorial of n" in its** `HTTP` **response. Create a async combinatorial function by making calls to this server.**

Answer:

In these kind of questions if there's something that is not clear, or if there's something that you don't quite remember, feel free to ask the interviewer. You don't need to know the combinatorial formula by heart. In fact you don't have to know any formula, or any API; and if you ask, the interviewer will gladly help you.

The combinatorial formula is:

$$C(n, r) = n! / ((n-r)! * r!)$$

Now let us implement this using continuation-passing style.

First let's use the **fetch** function in the former question to create a remote factorial proxy:

```
// 0052_factorial_service_cps.js

function factorial(n, result) {
    fetch('http://example.com/factorial/' + n, function(n) {
        result(n);
    });
}
```

Then we can come up with an async combinatorial implementation as follows:

```
// 0053_combine_cps.js

function combine(n, r, result) {
    factorial(n, function (factorial_n) {
        factorial(n-r, function (factorial_nr) {
            factorial(r, function (factorial_r) {
                result(factorial_n / (factorial_nr * factorial_r));
            });
        });
    });
}

combine(10, 2, function(n) {
    console.log(n);
});
```

Modify the CPS combine function to handle errors if n or r is less than zero. Assume that for n < 0, the remote factorial web method returns an "HTTP 200" response with a plain text error message.

Answer:

Once you are using callbacks, or continuations, it's not possible to use standard exception handling mechanisms, because by the time the continuation function is called (*i.e., sometime in the future*), it's impossible to reach the call stack of the past. Let's see this with a naïve implementation, first:

```
// 0054_factorial_exception.js

function factorial(n, result) {
    fetch('http://example.com/factorial/' + n, function(n) {
        n = parseInt(n, 10);

        if (n === NaN) {

            // n has a "non-numeric" detailed error message.
            throw n;
        }

        result(n);
    });
}
```

The problem is, when we **`throw n`**, there is nobody around to catch it. It's impossible to catch this inside the **`combine`** function, because this exception is thrown in a completely different scope and time.

Generally the first thing that comes to your mind would be something that the interviewer would be expecting, too. So do not hesitate to propose your solution thinking that the interviewer will see your implementation as premature. She will, indeed, be expecting a naïve answer.

Just **do not forget** to indicate the apparent drawbacks of your implementation, because what she cares about will be **the way** you approach the problem, and **how** you **analyze** the current situation.

Trust me, she won't care less about how perfect your solution is. After you indicate the downsides of your solution, she will be probing you to implement a solution to overcome these issues.

One way CPS deals with the exceptional cases, is to use a second error handler continuation. So our factorial function becomes something like this:

```
// 0055_factorial_errback.js

function factorial(n, result, error) {
    fetch('http://example.com/factorial/' + n, function(n) {
        n = parseInt(n, 10);

        if (n === NaN) {

            // n has the detailed "non-numeric" error message.
            error(n);

            return;
        }

        result(n);
    });
}
```

And our combinatorial implementation that handles error cases will turn out to be something like this:

```
// 0056_combine_errback.js

function combine(n, r, result, error) {
    factorial(n, function (factorial_n, error) {
        factorial(n-r, function (factorial_nr, error) {
            factorial(r, function (factorial_r, error) {
                result(factorial_n / (factorial_nr * factorial_r, error));
            });
        });
    });
}

combine(10, 2, function(n) {
    console.log(n);
}, function(description){
    console.log(description);
});
```

Some might consider carrying the error handler continuation all the way through the chain of factorials more *involved* than necessary. Additionally, the overall callback hell does not look quite appealing to the eye.

We will be handling these two cases in the Promises section, next.

Method Chaining

Method chaining is the building block of some of the more complex patterns such as promises, and deferred execution.

If you have played with UNIX pipes before, chaining would look very familiar to you.

Just like UNIX pipelines, **chaining** enables us to perform a series of transformations on an object one after another:

```
// 0057_method_chain.js

// Normally, these functions should return a new state.
// We're returning the state itself for the sake of simplicity.
function doSortMagic(state, args) {return state;}
function doCatMagic(state, args) {return state;}
function doAwkMagic(state, args) {return state;}
function doSedMagic(state, args) {return state;}

var jsiq = {
    state: null,
    cat: function() {
         this.state = doCatMagic(this.state, arguments);

         // Returning the object itself enables us chain it.
         return this;
    },
    sort: function() {
        this.state = doSortMagic(this.state, arguments);

        return this;
    },
    awk: function() {
        this.state = doAwkMagic(this.state, arguments);

        return this;
    },
    sed: function() {
        this.state = doSedMagic(this.state, arguments);

        return this;
    }
}

// Chain the methods.
jsiq.cat('file1', 'file2').sort('-n').awk('{print $2}').sed('s/@/ at /g');
```

This really is not that complicated, and the key is to return the original object in each method to allow chaining. As long as you continue to return the jsiq object you can execute another method on it, or else you will break the chain.

JavaScript chaining can be very useful; it provides a fluent interface, and *arguably* more readable code, especially when you want to preform a series of actions on a single object. As with every pattern there's a fine line between use and abuse of chaining. If you add one or two chained methods onto an object that is still readable. If you start adding four, five, or even nine, then your code becomes harder not only to read but to maintain.

Promises

While coding, most of the time we take certain features for granted: **Sequential programming** for instance. Writing down a series of instructions that does one thing after another.

In general, given the same input data, a sequential program will always execute the same sequence of instructions and it will always produce the same results.
Sequential programs' execution is **deterministic**.

The sequential paradigm has the following two characteristics:

★ The textual **order of statements** specifies their order of execution;

★ Successive statements must be executed **without any overlap** (in time) with one another.

Neither of these properties applies to concurrent programs.

Two different methodologies (*i.e., sequential programming versus concurrent programming*) support two different needs. If you have a problem driven by events, then you should use an event-driven methodology. If you need to perform procedures on defined data, but you are not worried about what's happening elsewhere, then obviously you want to use a more **sequential** style.

However, if you are writing code in **Javascript** that uses blocking I/O or other long running operations, *sequential programming* is out of the question because blocking the only thread in the system is a very bad idea. The solution is to implement algorithms using asynchronous **callbacks** as **continuations**.

For an in-depth overview of these ideas you may want to read a not-very-short introduction to **Node.JS**, then demystify non-blocking IO, and then try to understand **Node.JS** event loop, and finally have some **Node.JS** in a restaurant.

Let's start with a simple **Node.JS** example to have some context:

```
// 0058_nodejs_cps.js

var http = require('http'),
    fs = require('fs');

http.createServer(function(request, response) {
   request.on('end', function () {
      // Intentionally removed code that sets headers, etc. for simplification.

      fs.readFile('jsiq.txt', 'utf-8', function(error, data) {
         fs.writeFile('jsiq.txt', ++parseInt(data), function(err) {
             if (err) {
                 response.end('Error');

                 return;
             }

             response.end('This page was refreshed ' + data + ' times!');
         });
      });
   });
}).listen(8080);
```

The above code sample creates a server, binds to an event on the server, reads a file when there's a request, increments the data in the file, writes the data back to the file, and displays a response to the client.

This leaves us with the so called callback hell problem.

In the above code there is a 4-level-deep nested asynchronous callbacks. In a larger application this might create code that is hard to read and maintain.

Flatten the code in the last page, to decrease the nesting of callbacks.

Answer:

Actually, callback hell is not a big problem, and you can organize your code to solve it; and you don't need to know about promises at all to do this. So let's take an initial stab at the problem:

```
// 0059_callback_hell_flattened.js

var http = require('http'),
    fs = require('fs');


function handleWriteFile(err) {
    var response = this.response,
        data = this.data;

    if (err) {
        response.end('Error');

        return;
    }

    response.end('This page was refreshed ' + data + ' times!');
}

function handleReadFile(error, data) {
    var response = this.response;

    fs.writeFile('jsiq.txt', ++parseInt(data), handleWriteFile.bind({
        response: response,
        data: data
    }));
}

function handleRequestEnd() {
    var request = this.request,
        response = this.response;

    fs.readFile('jsiq.txt', 'utf-8', handleReadFile.bind({ response : response }));
}

function serve(request, response) {
    request.on('end', handleRequestEnd.bind({
        request: request,
        response: response
    }));
}

http.createServer(serve).listen(8080);
```

Look Ma! No nested callbacks!

However, this solution has certain drawbacks, in terms of **reusability**, and **exception handling**; and to understand why, we have to travel to the wonderful world of promises, first.

What Is a Promise?

Here's the formal definition from Promises/A+ Spec:

A **promise** represents a **value** that **may not be available yet**.
The primary method for interacting with a promise is its **then** method.

The specification is more detailed than that. I would recommend you stop reading this section, and read and understand the spec. This will make following the rest of the text much easier.

CommonJS Promises are one of the many standards that CommonJS project specifies.

Here's a quick terminology, to put the rest of the discussion in context:

Fulfillment:

> When a successful promise is fulfilled, all the pending callbacks are called with the value. If more callbacks are registered in the future, they will be called with the same value.
> Fulfillment is the asynchronous analog for returning a value.

Rejection:

> When a promise cannot be fulfilled, a promise is "rejected" which invokes the errbacks that are waiting and remembers the error that was rejected for future errbacks that are attached.
> Rejection is the asynchronous analog for throwing an exception.

Resolution:

> A promise is resolved when it makes progress toward fulfillment or rejection. A promise can only be resolved once, and it can be resolved with a promise instead of a fulfillment or rejection.

Callback:

> A function executed if a promise is fulfilled with a value.

Errback:

> A function executed if a promise is rejected, with an exception.

The Promise Contract

Each promise should also satisfy the following properties:

There Is One Resolution or Rejection

A promise is resolved only one time. It will never be fulfilled if it has been rejected or rejected if it has been fulfilled.

Listeners Are Executed One Time

An individual callback or errback will be executed once and only once. This follows from the first rule of the contract.

Promises Remember Their State

A promise that is resolved with a value remembers the fulfillment.

If a callback is attached in the future to this promise, it will be executed with the **previously resolved value**.

The same is true of errbacks. If a promise is rejected and an errback is attached after the rejection, it will be executed with **the rejected value**.

Promises behave the same way regardless of whether they are already resolved or resolved in the future.

Bubbling

The value returned by a callback is bubbled up the chain of promises. The same is true of rejections. This allows for rejections to be handled at a higher level than the function that called the promise-returning function that is rejected. This also allows promise-based APIs to compose responses for rejections and successes by changing the value or error as it is bubbled.

> Don't get intimidated by the above definitions if you don't understand them the first time you read it. It's really hard to learn promises by reading the promise contract. I would suggest you to play with a promise-supporting library (*such as Q*), get used to that library's idioms, then come and read the promise contract above. Believe me, it will make much more sense, then.
>
> Promises is a harder concept to get used to. Give some time for the overall idea to sink in.

If you've got yourself familiarized with the concept of promises, we can continue with a sample interview question, that covers the basics of promises.

Given the following code;

```
// 0060_question_promise.js


api.fetch('/url/to/attendees')
        .then(fetchAllAttendeeDetails)
        .then(calculateAverageAge)
        .then(displayResult, displayError);
```

Assume:

- ★ **`api.fetch`** returns a promise that resolves with an array of urls.
- ★ **`api.promise`** returns a CommonJS promise.
- ★ **`api.when`** returns a promise that gets fulfilled when all of its arguments (*which are CommonJS promises*) are fulfilled.
- ★ All promises returned by the `api` methods conform to **Promises/A+** specs.

Also assume that the attendees will receive a discount if their average age is less than 20.

Implement:

`fetchAllAttendees`**,** `calculateAverageAge`**,** `displayResult` **and** `displayError` **functions.**

Answer:

At its very basic meaning, a promise is an object with a **`then`** function. This **`then`** function executes some time in the future, when some condition occurs (such as receiving an asynchronous response from an **AJAX** request).

From the question we understand that the first request returns a promise that resolves with a list of urls; and for each url in that list we would be doing further requests to the server to get some data (*the age of the participant*).

The tricky part is, we need all the ages of the participants readily available to calculate their average; and since each request we make to the server is asynchronous, we need a mechanism to wait until we get all the responses back.

A naïve approach would be something like:

```
// 0061_promise_answer.js

var remaining = 0,
        age = 0;

function incrementAge(number) {
    age += number;

    remaining--;

    if (remaining === 0) {
        calculateAverageAge();
    }
}

function fetchAllAttendeeDetails(data) {
    var i, len;

    remaining = data.length;

    peopleTotal += len;

    for (i = 0 len = data.length; i < len; i++) {
        api.fetch(data[i].url).then(incrementAge);
    }
}
```

We basically wait for all the remaining items to finish, by decrementing the **`remaining`** counter. Although it could work, there are several problems with this approach: First, we're doing three things at once in the **`incrementAge`** function; we decrease the counter, increase the total age, and compute the average if a certain condition is met. This implementation is harder to reuse, harder to refactor, and prone to errors.

Whenever possible any function should do one thing, and it should do it well.

In an interview, this initial response is okay; and the interviewer will probe you to enhance your solution. When we look at the question, we see that the API has an **`api.when`** method, which will make keeping the counter of remaining requests, redundant. This will help us rewrite **`incrementAge`** as follows:

```
// 0062_increment_age_refined.js

function incrementAge(number) {
    age += number;
}
```

This is way better! Now let's see how we can use it:

So let's see how we make it happen using **`api.when`**:

```
//0063_promise_with_when.js

var ageTotal = 0,
    peopleTotal = 0;

function incrementAge(age) {ageTotal += age;}

function displayResult() {
    var averageAge = Math.floor(ageTotal/peopleTotal);

    if (averageAge < 20) {
        console.log('Eligible for discount');

        return;
    }

    console.log('Not eligible for discount.');
}

function displayError(error) {
    console.log('An error occured.');
    console.log(error);
}

function pushRequest(i, data, requests) {
    var url = data[i].url;

    requests.push( api.fetch(url).then(incrementAge) );
}

function fetchAllAttendeeDetails(data) {
    var requests = [], i;

    peopleTotal = data.length;

    for (i = 0; i < peopleTotal; i++) {
        pushRequest(i, data, requests);
    }

    return api.when.call(null, requests);
}

api.fetch('url/to/attendees')
    .then(fetchAllAttendeeDetails)
    .then(calculateAverageAge)
    .then(displayResult, displayError);
```

In **`fetchAllAttendeeDetails`**, we pass an array of **promises** to **`api.when`** as arguments.
Each of these arguments have this form:

```
api.fetch(url).then(incrementAge)
```

Each of those **promises** resolve as we fetch the data from the server with **`api.fetch`**.

> **`then`** function returns a **new promise** (P1) that's fulfilled with the return value of its argument. If the argument (which is a **`function`**) also returns a **promise** (P2), then the initial promise (P1) does not get fulfilled until the returned promise (P2) gets fulfilled first. Then **`incrementAge`** is called with the resolution of the promise that **`api.fetch`** returns.

Since, **`then`** also returns a **new promise** (*according to Promises/A+ specification*), that returned promise also resolves with the return value of **`incrementAge`**.

`fetchAllAttendeeDetails` calls **`api.when`** which returns a promise that only resolves when all of its arguments (which are promises, too) resolve.

All of these mean that **`calculateAverageAge`** has the **`ageTotal`** readily available, by the time it's called.

Also, it's important to note that if an error occurs, at any part of this long chain of promises; the rest of the promises will be **rejected**, and eventually **`displayError`** method will be called; and if everything goes as expected, without errors, and rejected promises, then **`displaySuccess`** method will be called.

Well, that's a mouthful of text, that the reader can only understand if she has read the **Promises/A+ specification** fully.

Have you understood it all at once? You're not alone. Promises are intimidating when you see them for the first time.

It's the monad curse, that makes the promises complicated. The only way to break this curse, is to bisect several promise-related examples and interview questions, and try to see how they chain, and cascade. I'll try to show how the promises flatten the asynchronous execution flow, and make it appear like sequential, as clearly as I can; however, nothing substitutes hands-on practice.

> **The only surefire way to learn promises is to actually use them.**

And there's still something smelly with this solution.

We are using **`ageTotal`** and **`peopleTotal`** globals. Using globals is most of the time a code smell.

So let's optimize our code further:

From the question text, we know that **`api.promise`** returns a promise. Let's use it:

```
// 0064_promise_when_refined.js

function displayResult(averageAge) {
    if (averageAge < 20) {
        console.log('Eligible for discount');

        return;
    }

    console.log('Not eligible for discount.');
}

function displayError(error) {
    console.log('An error occurred.');
    console.log(error);
}

function fetchAllAttendeeDetails(data) {
    var requests = [],
        ageTotal = 0,
        peopleTotal = data.length,
        i;

    function incrementAge(age) {ageTotal += age;}

    function pushRequest(i, data, requests) {
        var url = data[i].url;

        requests.push( api.fetch(url).then(incrementAge) );
    }

    for (i = 0; i < peopleTotal; i++) {
        pushRequest(i, data, requests);
    }

    return api.when.call(null, requests).then(function() {
        var p = api.promise();

        p.resolve(Math.floor(ageTotal/peopleTotal));

        return p;
    });
}

api.fetch('url/to/attendees')
.then(fetchAllAttendeeDetails).then(displayResult, displayError);
```

We've returned a **promise** that resolves with the average age we want.

We have also inlined **`ageTotal`** and **`peopleTotal`**; so that they are local to our functions, and they are not polluting the global namespace.

Actually we don't need to use **`api.promise`** at all, because **`then`** returns a **promise** that resolves with the return value of its argument. So we can rewrite that part as follows:

```
// 0065_promise_when_refined_alternate.js
...
    return api.when.call(null, requests).then(function() {
        return Math.floor(ageTotal/peopleTotal);
    });
...
```

This will do exactly the same thing and free us from creating a redundant promise object.

> In an interview, it's totally normal that you may not come up with the most ideal answer. The interviewer will be looking at **how** you approach the problem; **not** what you provide as an answer. The more you **progressively improve the solution** by **logical reasoning**; the more they like you.

Design a simple promise library; you don't need to make it fully Promises/A+ compliant.

Answer:

At a basic level, a promise is a simple entity. It is initialized with a **`PENDING`** state, and it can transition to either **`FULFILLED`**, or **`REJECTED`** states with a resolution **value**, or a rejection **reason**, respectively. When the promise is **`FULFILLED`**, or **`REJECTED`**, we say that the promise is **resolved**. When a promise is **resolved**, it is **immutable**; meaning: neither its state nor its resolution value or rejection reason can change. Also a promise manages a **queue** of delegates that we register to the promise using its **`then`** method. Let's create a constructor that considers all of these:

```
// 0066_simple_promise_ctor.js

    var PromiseState = {
        PENDING: 1,
        FULFILLED: 3,
        REJECTED: -1
    };

    function Promise() {
        this.queue = [];
        this.state = PromiseState.PENDING;
    };
```

Nothing fancy. We just initialized the promise with a state, and created a processing queue. A promise should also have a then method which populates the queue.

```
// 0067_simple_promise_then.js

    Promise.prototype.then = function(callback, errback) {
        this.queue.push({
            callback: callback || null,
            errback: errback || null
        });

        var shallFireImmediately = this.state !== PromiseState.PENDING;

        if (shallFireImmediately) {
            fire(this, this.result);
        }

        return this;
    };
```

That's equally easy to construct, if we've read the specs before.

And we should be able to resolve, or reject a promise. These two operations are like *freezing* the promise in time. After you resolve, or reject a promise; changing the value or state of it is not allowed.

```
// 0068_simple_promise_resolve_reject.js

    Promise.prototype.resolve = function(value) {
        if (this.state === PromiseState.REJECTED) {
            throw "Promises/A+ Violation: Promise state should be immutable.";
        }

        if (this.state === PromiseState.FULFILLED && value !== this.result) {
            throw "Promises/A+ Violation: Promise resolution should be immutable.";
        }

        this.state = PromiseState.FULFILLED;

        fire(this, value);
    };

    Promise.prototype.reject = function(error) {
        if (this.state === PromiseState.FULFILLED) {
            throw "Promises/A+ Violation: Promise state should be immutable";
        }

        if (this.state === PromiseState.REJECTED && error !== this.result) {
            throw "Promises/A+ Violation: Promise error should be immutable.";
        }

        this.state = PromiseState.REJECTED;

        fire(this, error);
    };
```

And when we fire a promise, we basically shift the latest callbacks from the execution queue and run them.

One thing to pay attention to is the fact that any exception thrown when executing a success callback, or an error callback should result in the promise being rejected with that exception as a reason. Lets' see how we can do this:

```
// 0069_simple_promise_fire.js

    function fire(promise, value) {
        var item, delegate;

        promise.result = value;

        while (promise.queue.length) {
            item = promise.queue.shift();
            delegate = promise.state === PromiseState.REJECTED ?
                item.errback : item.callback;

            if (delegate) {
                try {
                    promise.result = delegate.call(null, promise.result);
                } catch(e) {
                    promise.state = PromiseState.REJECTED;
                    promise.result = e;
                }
            }
        }
    }
```

Also note that we pass the result of the former promise into the next delegate in queue, which helps the then methods nicely chain together. The next **`then`**, resolves with the return value of the current one; and if an error is thrown, the next **`then`** is rejected with the same error, too.

Let's combine all of these methods together, and see how we can come up with a very basic promise implementation:

```
// 0070_simple_promise_combined.js

(function(window) {
    'use strict';

    var PromiseState = {PENDING: 1, RESOLVED: 3, REJECTED: -1};

    function fire(promise, value) {
        var item, delegate; promise.result = value;

        while (promise.queue.length) {
            item = promise.queue.shift();
            delegate = promise.state === PromiseState.REJECTED ?
                item.errback : item.callback;

            if (delegate) {
                try {promise.result = delegate.call(null, promise.result);}
                catch(e) {
                    promise.state = PromiseState.REJECTED;
                    promise.result = e;
                }
            }
        }
    }

    function Promise() {this.queue = []; this.state = PromiseState.PENDING;}

    Promise.prototype.then = function(callback, errback) {
        this.queue.push({callback: callback || null, errback: errback || null});
        var shallFireImmediately = this.state !== PromiseState.PENDING;
        if (shallFireImmediately) {fire(this, this.result);}

        return this;
    };

    Promise.prototype.resolve = function(value) {
        if (this.state === PromiseState.REJECTED) {
            throw 'Promises/A+ Violation: Promise state should be immutable.';}
        if (this.state === PromiseState.FULFILLED && value !== this.result) {
            throw 'Promises/A+ Violation: Promise resolution should be immutable.';}
        this.state = PromiseState.FULFILLED;

        fire(this, value);
    };

    Promise.prototype.reject = function(error) {
        if (this.state === PromiseState.FULFILLED) {
            throw 'Promises/A+ Violation: Promise state should be immutable';}
        if (this.state === PromiseState.REJECTED && error !== this.result) {
            throw 'Promises/A+ Violation: Promise error should be immutable.';}
        this.state = PromiseState.REJECTED;

        fire(this, error);
    };

    window.Promise = Promise;
}(this));
```

There are several things omitted in the implementation. For instance, when a success handler of promise **A** returns a promise **P2**, then **P1** should essentially become **P2**, and **P1** should remain pending until **P2** is resolved; however, in an interview, a solution similar to this one would be sufficient.

I strongly suggest the reader download the Promises/A+ Tests and make this implementation pass all the tests there.

Aside

For those who want to get technical, **promises** are nothing but a special form of **Monads**.

Understanding **Monads** is a nontrivial task though; and it's out of the scope of this book; and trust me, no one will be asking you about them in an interview. However, once you understand them, the way you look at programming will change; and I strongly suggest you **dive into monads**.

Monads come with a curse, though. Once you are enlightened, and you can wrap your head around the whole thing; it's virtually impossible to express what you've learned in simple terms. That's why there is a plethora of **Monad** tutorials on the Internet, none of which make any sense to someone new to the subject matter.

Do you like burritos? Then you'll love Monads. Because they are just like burritos.

The thing is, once you are able to wrap your head around monads; it's virtually impossible to tell the idea to others.

Module

The **JavaScript Module Pattern** is arguably the most important JavaScript design pattern. It's a very clever way with dealing with static instances, enabling us to create a pseudo-private encapsulation to hide the module's internals while exporting only the necessary functionality.

The **Module Pattern** is excellent for encapsulating component-specific knowledge.

It was first coined by Eric Miraglia, years ago at Yahoo! User Interface Blog.

Especially when developing external widgets to sites, this pattern is a gold mine, because your widget's logic will be encapsulated inside a **closure**, and it will **not** pollute the global namespace and it will not mess with the site's global **JavaScript** code.

John Resig, was one of the first people evangelizing the module pattern in a library (i.e., **jQuery**). Also Christian Heilmann is one of the pioneers in motivating people to show love to the module pattern.

Node.JS uses modules too. It has it's own way of **importing** and **exporting** stuff (which is defined by the **CommonJS** standards) **modularly**.

A Basic Module

Here's how we basically define a module:

```
// 0071_basic_module.js

jsiq.DemoModule = (function () {
    return  {
        publicProperty: 'accessible as jsiq.DemoModule.publicProperty',
        publicMethod: function() {alert('this is a public method');}
    };
}());
```

Private Static Scope

And as we've seen before, you can add private variables and functions to a **module**:

```
// 0072_private_static_scope.js

jsiq.DemoModule = (function () {
    var privateMember = 10;

    function privateMethod() {alert(privateMember);}

    return {
        publicProperty: 'accessible as jsiq.DemoModule.publicProperty',
        publicMethod: function() {alert('this is a public method');}
    };
}());
```

Caching Commonly-Used Globals as Parameters

You may want to improve performance by caching frequently used globals as the module's parameters.

```
// 0073_module_cache_globals.js
(function(jsiq, window, document, UNDEFINED) {
    // Do stuff with `window`, `document`, and `jsiq`.
}(this.jsiq, this, this.document));
```

The above example caches commonly used **`window`**, and **`document`** objects within the module's scope to decrease scope-lookup and increase the overall performance of the module.

Why we use an **`UNDEFINED`** constant, instead of the **`undefined`** keyword is a totally different story:

`undefined` is *not* a keyword, it's a property of the global object. So it's value can be overridden with something like **`undefined = 42;`**. That's really easy to do if you use Yoda Conditions and mistype "=" instead of "==".

The same is true for **`NaN`**, it is also a property of the global object, but it's harder to mistakenly override the value of **`NaN`**. Besides you should check whether something is a number using **`isNaN()`** function anyway.

The "Revealing" Module

The revealing module is another way to write the module pattern. Instead of defining all the private methods inside the **IIFE**, and the public methods within the returned object; you write all methods within the **IIFE** and just "reveal" which ones you wish to make public in the return statement.

```
// 0074_revealing_module.js

this.jsiq = (function() {
    function publicMethod() {}

    function privateMethod() {
        publicMethod();
    }

    function anotherPublicMethod() {
        privateMethod();
    }

    return {
        publicMethod: publicMethod,
        anotherPublicMethod: anotherPublicMethod
    };
}());
```

There are a few advantages to the revealing module pattern:

- ★ All the **functions** are declared and implemented in the same place.
  That is arguably more readable, and creates less confusion;
- ★ **private functions** can now access **public functions**, if necessary;
- ★ And you will be using **`this`** keyword less. Instead of calling **`this.publicFunction()`**, you can directly call **`publicFunction()`**. That's good, because what **`this`** resolves to may end up being something different than what's originally intended.

Once you get used to the *functional* nature of **JavaScript**, start thinking functionally, and stop thinking in OO paradigms; you'll recognize you'll be using the **`this`** keyword less and less.

If you are using the **`this`** keyword in your code, you are intentionally or unintentionally referring to a **bound context**. This usage creates dependencies that can grow out of your control if you don't manage carefully.

The less you use the **`this`** keyword, the less you'll need to worry about these.

Using Module as a Façade

Another variation is to use the module structure as a Façade to sweep our mess under the carpet.

Here's an example:

```
// 0075_module_as_facade.js

jsiq.doDailyChoires = (function() {
    function setTableForBreakfast() {}
    function getKidsReadyForSchool(){}
    function shopGroceries() {}
    function takeOutTrash() {}
    function batheChildren() {}
    function putChildrenToBed() {}

    return function(isWeekend) {
       setTableForBreakfast();

       if (!isWeekend) {
           getKidsReadyForSchool();
           shopGroceries();
           takeOutTrash();
       }

       batheChildren();
       putChildrenToBed();
    };
}());

jsiq.doDailyChoires(true);
jsiq.doDailyChoires(false);
```

Extending an Existing Module

This pattern is generally used when creating plugins to libraries. One such jQuery Plugin Development Pattern is described extensively by Mike.

Here is a simplified example of the idea:

```
// 0076_extending_module_strict.js

var jQuery = (function($) {
    $.fn.pluginMethod = function() {

        ...

    }

    ...

    return $;
}(this.jQuery));
```

The above code is flexible because you don't even need the "**`var jQuery =`**" or the "**`return $`**" statement near the end. **`jQuery`** will still be extended with the new method without them. It's, actually, bad for performance to return the entire **`jQuery`** object and reassign it.

What if **`this.jQuery`** is not defined yet?
We can use a somewhat **loose augmentation** to handle that case:

```
// 0077_extending_module_loose.js

var jQuery = (function($) {
    var fn = $.fn || ($.fn = {});

    $.fn.pluginMethod = function() {

        ...

    }


    ...


    return $;
}(jQuery || {}));
```

Method Overriding in Modules

If you know the order which the modules are loaded, it's easy to override methods:

```
// 0078_module_method_overriding.js

var jsiq = (function (me) {
    var oldMethod = me.moduleMethod || function(){};

    me.moduleMethod = function () {

        // You can call cache method if required.
        oldMethod();

        // Do other stuff.
    };

    return me;
}(this.jsiq));
```

Cloning a Module

You can clone a module with a pattern similar to below:

```
// 0079_module_clone.js

var jsiq2 = (function (old) {
    var me = {},
        key = null;

    for (key in old) {
        if (old.hasOwnProperty(key)) {
            me[key] = old[key];
        }
    }

    var base_moduleMethod = old.moduleMethod;

    me.moduleMethod = function () {

        // Things to do before calling the cached method:
        doPreExecutionStuff();

        //
        // You can call the cached base method if necessary.
        //
        // Hint: you can context-bind using
        // `base_moduleMethod.apply(me, arguments)`
        // instead of simply calling `base_moduleMethod()`.
        //
        base_moduleMethod();

        // Things to do after calling the cached method:
        doPostExecutionStuff();
    };

    return me;
}(this.jsiq));
```

Note that in the above **`for`** loop, properties which are **`Object`**s or **`Function`**s will **not** be duplicated; instead, they will exist as multiple references to a single object: Changing one will change the other. This could be fixed for objects with a **recursive** cloning process, and it could be fixed for functions using **`Function.prototype.apply`** or **`Function.prototype.bind`**. I am leaving those implementation details as an exercise to the reader.

Submodules

I'm adding this one for the sake of completeness.

If you've observed all the examples so far; the following code sample will be obvious. Here's how to create a sub-module:

```
// 0080_submodules.js

jsiq.fooBarBaz = (function () {
    var me = {};

    // Do stuff.

    return me;
}(this.jsiq));
```

At times, you may need to create sub-modules.

To be honest, in the recent 10 years, I have never needed to create a sub-module, ever.

Your situation may vary, of course.

Immediately-Invoked Function Expressions (**IIFE**)

I **strongly disagree** to the claim that IIFE is not a closure.

An IIFE (or *immediately-invoked function expression*), as its name implies, is a function which is immediately invoked after it has been defined. One advantage of **IIFE** is that it creates it's own scope so that application logic may be encapsulated and protected from global reach.

Actually, a **JavaScript module** is nothing but an **IIFE** that returns an **Object** as a public interface as an **export**.

Similar to modules, **IIFE**s also create a separate execution scope that lives even after the function is executed. This means by definition that modules and **IIFE**s are "technically" nothing but different variations of closures.

Here's how an **IIFE** looks like:

```
// 0081_very_simple_iife.js

(function(){
    alert('Hello world');
}());
```

If you have done your homework on closures, then **IIFE**s are really nothing unfamiliar.

Explain what the following code does:

```
// 0082_question_iife.js

function foo(name){ alert('hello ' + name); }('#jsiq');
```

Answer:

Chances are that you might not have spotted anything problematic in the code.

Once you look closer though, and compare it with the original IIFE example, you can see that it has a few parenthesis missing (*fast forward to the end of the answer to see the missing parens*).

What you expect, at a first glance is for the code to immediately execute and alert "Hello #jsiq".

However, the code is equivalent to this (*i.e., a function declaration followed by an arbitrary expression*):

```
// 0083_named_function_expression_misuse.js

function foo(name){ alert('hello ' + name); }
('#jsiq');
```

Running the code will not work as expected.
You can read more on EcmaScript function definition semantics on Dmitry's excellent article.

One way to fix this is to **hint** the JavaScript interpreter that we indeed want to use an expression, not a declaration.
For instance, unary operators require an expression:

```
// 0084_unary_operator_before_named_function_expression.js

!function foo(name){ alert('hello ' + name); }
('#jsiq');
```

Here are some other interesting ways to force an expression:

```
// 0085_force_expression_variations.js

// Assigning a variable.
var f = function foo(name){alert('hello ' + name);}('#jsiq');

// Using the comma operator.
0, function foo(name){alert('hello ' + name);}('#jsiq');

// Using logical operators.
true && function foo(name){alert('hello ' + name);}('#jsiq');

// Using `new` to call the function as a constructor.
new function foo(name){alert('hello ' + name);}('#jsiq');
```

However, the more traditional fix (*that the interviewer expects*) is putting the parentheses into the correct places to define precedence, so that the statement is executed as a function expression. Here's how:

```
// 0086_iife_fixed.js

( function foo(name){alert('hello ' + name);}('#jsiq') );
```

For further investigation on the subject, you can read this github page on named function expression and the **MDN** introduction to functions and function scope.

Asynchronous Module Definition (AMD)

A section on modules would not be complete without mentioning asynchronous module definition (*AMD*). amdjs wiki defines it as:

> The *Asynchronous Module Definition* (**AMD**) API specifies a mechanism for defining modules such that the module and its dependencies can be asynchronously loaded. This is particularly well suited for the browser environment where synchronous loading of modules incurs performance, usability, debugging, and cross-domain access problems.

Indeed **AMD** is currently a workaround, which is supposed to be a part of the language in the future.

There are several **AMD** frameworks around, the most commonly used ones are require.js and curl.js. Both of these libraries provide mechanisms to resolve module dependencies and reuse modular code without having to worry about the script load order; because the dependencies are resolved and lazy-loaded at runtime.

Indeed, **Node.JS** also implements a *slightly different* CommonJS specification for its built-in modules. That's another reason why modules are so popular and everyone is talking about **AMD** and **CommonJS** modules these days.

> Arguably, **CommonJS** specification is a better fit for server-side modules,
> and **AMD** is a better fit for the browser environment.

When you read the documentation on **Node.JS** modules (*and you should*) you'll quickly notice the similarity between how an **AMD** module is defined; and while you're into node modules you'd want to view this huge and growing list of **Node.JS** modules.

Creating a **CommonJS** module is really easy:

The first thing you need to do is, to *define* your module sing a **`define`** function.

A define function that takes an optional **id**, a set of **dependencies**, and a **factory function** that's basically a closure for the module (*Remember? Every function is a closure by definition*).

Here's an example:

```
// 0087_amd_require.js

define('jsiq', [
    'utils', 'dom', 'ajax'
], function (
    utils, dom, ajax
){
    return {
        render: function() {
            return ajax('/api/endpoint').then(function(responseHtml) {
                dom.select('#masterPage').html(responseHtml);
            };
        }
    };
});
```

The above definition enables us to use the included module methods, without explicitly loading them beforehand.

How the **`define`** function is implemented will depend on the particular **AMD** library; but the overall logic will be the same: *All the dependencies will be resolved before the factory function executes*.

We can use the **`render`** function of the above **`jsiq`** module without needing to know which other modules it depends on. The chain of dependencies will be automatically resolved for you. Here is an example that shows this usage:

```
// 0088_amd_render.js

define('main', [
    'jsiq'
], function (
    jsiq
) {
    // The factory function will called when all the dependencies are resolved.
    jsiq.render();
});
```

That's is the beauty of the **AMD** syntax. And, arguably, the ugliness of the **AMD** syntax is that you need to enclose your module logic in it's own factory function, provide dependencies as parameters, and keep the number and order of the parameters in sync, when you need to add or remove a dependency. – **CommonJS** modules have a much cleaner syntax. And it is possible to write your code in CommonJS module format, and then convert it to **AMD**, too. For a detailed summary, I highly suggest you to read **require.js** documentation to learn more.

Each **AMD** library uses a kind of script loader to load the dependencies.

Yet, loading the script is the easy part, what's relatively harder is to come up with a robust way of **resolving** and dynamically loading dependencies.

Summary

Some of the patterns are more frequently used than others.

Write in the language you're writing; **do not** try to force **Java** patterns into **JavaScript**. JavaScript has its own beauties and its own way of dealing with things. Embrace the good parts of it, instead of fighting against the language.

In addition someone working with patterns without knowing its strengths, areas of implications, and liabilities is something that we should avoid. Besides, each pattern comes with its own level of abstraction, and it makes code harder to trace. So you should be absolutely sure that using the pattern is worth the additional level of complexity it causes.

Don't use patterns to show off your colleagues that you know kung-fu.

Remember that simplicity is the ultimate sophistication.

Always think simple; and when in doubt **KISS**.

Chapter 6
More Than JavaScript

To become a rockstar **JavaScript Engineer**, knowing all the nitty-gritty details of **JavaScript** is necessary, and it is not sufficient. You will also have to be knowledgeable about the environments that interact with JavaScript.

It is highly unlikely that you will land a **JavaScript** Engineering job, if the only thing you know is **JavaScript**.
You should be proficient in other areas, like:

- ★ A server-side language and platform (like **Java**, **ASP.net/C#**, **Ruby on Rails**, **Django/Python**...)
- ★ A Relational **DBMS** (like **MS SQL Server**, **MySQL**, **PostgreSQL**, **Sybase**...)
- ★ A **NoSQL** Document Store (such as **MongoDB**, or **Redis**...)
- ★ **HTML**, **CSS**, and the **DOM API**
- ★ How Browsers Work (how render trees are formed, what a **repaint** is, what a **reflow** is, what an **animation frame** is, what causes memory leaks, how the JavaScript garbage collector works...)
- ★ How the HTTP works as a protocol (proxies, **DNS**, headers, cache validation, **gzip** compression...)
- ★ Web application security concepts (such ass **XSS**, **CSRF**, and how to mitigate them...)
- ★ Web application performance best practices
- ★ **Node.JS** (this is a **must**)

The internet is full of extensive badass lists to sift through and take your knowledge to the next level. I too have initially started compiling a list of resources for this chapter.

After reaching a list of hundreds of quality links, I realized that it would confuse the reader rather then helping.

So I will provide you **three great sites**. Once you've gone through all the material in those sites, you can consider yourself to be adequately knowledgeable to go to a **JavaScript Engineering** interview.

★ Web Platform – Although the content of the site requires some organization, it has extremely valuable material that is frequently updated.

★ Mozilla Developer Network – All parts of **MDN** (docs, demos, and the site itself) are created by an open community of developers.

★ HTML5 Rocks – **HTML5** is the ubiquitous platform for the web. Whether you're a mobile web developer, an enterprise with specific business needs, or a serious game developer looking to explore the web as a new platform, HTML5 Rocks has something for you.

Last, but not the least, you might want to look at **HTML5 Hub** as a bonus.

These are more than you would need to get you started with; if you need more resources on a particular topic, just shoot me an email; I would be glad to help.

Aside

Along with good resources, the Internet is also full of ill advice, incomplete and faulty information.

So when studying for a certain front-end development topic, make sure you **cross-check** the things you learn by looking at several different resources.

You might be surprised that some sites that come up first in Google, are, in fact, not that credible.

It all boils down to **keeping and open mind** and being ready and eager to learn new things.

Chapter 7
The Nontechnical Parts

The **nontechnical** parts of an interview, where the **behavioral interview** questions that you will be asked, are as important as the technical parts of the interview.

Note that I am not at all an expert on human behavior, or psychology. I have a special interest in science and neurology, but that's out of topic right now. I am just sharing my **honest** *observations and notes that I've taken during my interviews. This is my* **personal experience**. *It is* **what worked for me**. *It may not be the only way around; and your mileage may vary, of course.*

For the interested, there is a gazillion "*behavioral interview questions and their answers*" available on the **Internet**. My humble advice, though, is to **approach these techniques with a grain of salt**. Personal experience has shown me that some of those techniques are just **misdirections** and **ill advices** given by those who have never been to an interview at all.

If you are still seeking for questions to *memorize*, you are reading the wrong book; and I'm sorry this book hasn't taught you anything until this point. **There's no spoon**; memorization is **not** a solution. If you know yourself, and if you know your career history then you are good to go; **you don't need to memorize anything**.

Of course you will need to **rehearse** your knowledge and **your** answers to certain key questions **every day**; that's **not** memorization. That's **preparation**. You need to **be prepared at all times**.

If you have made so far with this book, you should have seen that it takes a **Zen**-like approach and aims to give you some **perspective**, and let you **control** how you learn things **your way**. Every individual is **unique**, and the only way to completely "get" a concept is to **doubt**, **experiment**, try to **refute**, **verify**, **retry**, **play**, and figure it out **your way**.

I consider all the readers of this book as the members of a **Dojo**. In dojos, **it's the road that's important**. So, it's **not** what you read; it's **how you apply what you read** to yourself that matters.

Similarly, if you think the intention of this section is to provide you with great answers to tough behavioral interview questions, **you are totally missing the point**. In that case, I suggest you read the preface once again.

...

For the technical part, it's easy to experiment with things and sharpen your skill set:

★ You **can** have a **technical blog**, and write about your findings;

★ You **can** have a **github** account, **share** your **open source projects**;

★ You **can** contribute to others' open source projects;

★ You **can** join programmer **communities**, **mailing lists** and **news lists**;

★ You **can** attend **meet-ups** and **events**;

★ There are even brick and mortar **dojos**, where you **can** hack around with like-minded individuals.

Hint: Replace all the "**can**"s on the above bullet points with "**must**"s.

...

If you want to be good at what you do (*which is a* **prerequisite** *for getting a decent job*) **you must do all the above**. And that's necessary, yet not sufficient. There's also the **nontechnical** part of the equation:

The only way to experiment with the behavioral part of the puzzle is, though, to actually **go for as many interviews as possible**. I cannot stress the importance of this more: **Go for as many interviews as you can**. Psychologists state that "behavior can be learned"; and when it comes to behavior, **learning by doing** is the only surefire way of achievement.

Nevertheless, in any interview there are things that you must do, behaviors that you should avoid, and certain aspects of the overall process that you should pay special attention to. The interviewer will be looking for certain **characteristics** and **traits** of yours and checking whether they match against the culture of the company.

The interviewer will try to gain an understanding of your **drive for achievement**, your **strategic thinking**, your **analytic skills**, your **leadership skills** and your **team-management skills**, your **thirst for continuous improvement**, your **decision-making skills**, your **judgement**, your **mentoring skills** (*i.e., how you develop others around*), your **team-working skills**... and more.

Since the interviewer cannot see inside of your head, the only way to express these to her is through...

★ **What you say** and **how you say it**;

★ The tone and pitch of **your voice**;

★ How you **stress** certain words;

★ Your **facial expressions** and your **body language**.

You cannot learn these by reading a book; you need to **practice** every single day.

Don't believe me? Just record your speech as if you were in an interview, answering certain questions, wait for a week, and re-listen to the recording, to see **how awful you sound**. Next week, do the same thing again, and see your **progress**.

Repetition and persistence, no matter how boring the task is, eventually leads to success.

I **repeat**, the only way you can experience the real world, is to get your butt out of your cave, *take the red pill*, and actually **go for as many interviews as possible**.

Sharpen your behavioral skills before you need them;
Or it will be too late and you'll need to learn it the hard way.

The following sections are dedicated to open the *Pandora's Box* of behavioral interviewing. Shall we?

Above Everything: Be Honest; Tell the Truth

If you forget everything you read, remember this:

Candor and honesty always wins against cunningness in any situation.
Do not bend the facts; do not try to trick the interviewer; do not pretend to be something you aren't; and **do not lie**.

Truth is **not** your enemy, but a "made up" answer is.

There are no exceptions to this: Never lie, always be honest, and **don't outsmart your interviewer**.

You may think that the interview went well, and you've successfully tricked the interviewer; and when you start interviewing with another company, you will wonder why "*that excellent job opportunity, which you were a perfect match*" never responded back to you. Because someone has tagged you as an "**unreliable**" and "**unprofessional**" **liar**, and has shared this with her professional network, resulting in an eternal kiss of death for you. Please keep in mind that news spreads very fast in the business world.

Honesty and ethical behavior is nonnegotiable.

You cannot have an excuse to lie, to be unethical, or to be dishonest.

Hiding the truth is the biggest mistake you can ever make. You will not be able to avoid the truth. While covering up one weakness, you will tell about another. Moreover, if the interviewer suspects that you are hiding the truth (*and hiding the truth is an alias for "lying"*) then everything you've said and will say is compromised. It's really not worth it. No matter how bad your circumstances are, you can still get a job – **if** you don't lie.

It's **not okay** to exaggerate your experience either. Even some minor exaggerations in your resumé will come back to bite you. Telling about things that are not representative of your current skill set is **unprofessional** and **dishonest**.
And it will be seen that way by the interviewer too. Nobody will hire a dishonest person.

Polishing your resumé is one thing, and making things different from what they actually are is another. Don't make things bigger than they are. Never claim credit for accomplishments you haven't done.

It's a lose-lose situation: If you lie, the interviewer detects that you lie, you are **blacklisted** and you lose **forever**. If you pass through all the interviews with your lies; it's highly probable that you won't be equipped to tackle with the challenges of the job. Eventually you'll get fired, and it will be hard to explain why you got fired without **more lies** in your next interview, and you lose more.

Moreover, you will have more than one interview with the company, to reach to an "offer" stage.
And believe it or not, your brain is not as efficient as you think in remembering which lie you've told to which person.

Further, if you're applying to a company in the same field as your last job, it's fair to think someone there may have worked at one of your past employers, and may be familiar with your old job. You really don't want to be called out on a skill your resumé says you have but that you can't demonstrate.

Don't dare to lie, **ever**! Else, your lies will haunt you **forever**.

Don't Blame Anyone or Anything

I will keep this short. Don't blame anyone or anything. Don't blame your boss, don't blame your coworkers, don't blame your company, don't blame the sector, don't blame your bad luck. Don't accuse people and don't judge people.

Express the interviewer the **facts** about the situations, **not** your **feelings** about them.

Blaming is a clear indication that you are an immature whiner. Nobody likes to see a whiner in their office.

And if you talk about your former company and coworkers in an accusing and unprofessional manner; then it's highly probable that you will talk about the company you're talking to alike. Even if you won't, and you promise you won't, from the perspective of the employer, a company's reputation is everything. For the employer, it's not worth taking the risk.

Utilize Every Opportunity

Always remember this as a guideline:

It ain't over, until someone tells you it's over.

Interviewing is not a single event, it's a **multifaceted process**.
During the process, never lose your hope, never lose your self-esteem, and use every opportunity to your advantage.

In your job search, there are things that are under your control (*like using your professional network, having a well-prepared cover letter and a resumé, getting prepared for the interview to the best of your abilities*), and certain things are not under your control. It's important to be prepared and perfectly utilize everything which are under your control.

The following sections discuss the things that are **under your control**, and what you can do to **make use of them** most. You will have more of the things that you should pay special attention to at the "Common Interview Mistakes and Misconceptions" section.

Starting from the next page, we'll specially look at things that **you control** to increase the odds of getting an offer.

And *may the odds be ever in your favor*[7].

[7] This is a quote from "the Hunger Games" that I really like. It's in your hands to change the odds to your favor.

Have Tailor-Made Cover Letters and Resumés

Don't just send a plain jane "*my resumé is enclosed*" email. Create a **genuine**, **well-prepared** message specifically describing **point by point** how your **knowledge**, **skills**, and **abilities** correlate with the position's **requirements**, and **responsibilities**.

Corollary:

Don't tell about your skills that are **not** *related to the job, either. It's a waste of time.*

For instance, don't write how good you are at game programming, if the job does not require that.

The same is true for your resumé. **Do not even think of sending a generic resumé to everyone**.

The Subject of the Interview Is **not** the Company; It Is **You**

One of the **ill** advises wandering on the Internet is that you should do whatever it takes to learn about the company's culture. This is **wrong**, and it's merely a **waste of time**. The culture of your next workplace will be determined by your prospective boss (*whom you don't know yet*) not by the company.

Believe me, there are better things to occupy your time. Don't try to get too much information about the company's culture. Also don't focus too much on the company either. A few hours of research is okay, but it's not worth spending your days researching.

Focusing too much on the company is the biggest mistake you will ever make.

Do you wonder why? Because **the interview is about you**, not about the company. So focus on your **knowledge**, **skills** and **abilities** and how those **relate** to the job you are going to do. If you can ask a few friends that work there, that's okay. More than that, is an inefficient use of your time.

Remember? The interview is about you, your **knowledge**, your **skills**, and your **abilities**. The interviewer does not give a damn about what you know about the company. So don't waste your time for excessively researching the company. Of course study what the company does, and read recent press releases and news about the company. However, don't think that it's your primary focus; and don't spend days on that.

Research the job description instead. Try to find what people in similar jobs do. Ask your network if there are any people in similar positions that you are applying. Read the job description thoroughly and try to correlate it with your professional knowledge.

The Cover Letter is Your "Elevator Pitch"

Those introductory **tailor-made** messages, which accompany a resumé, are also known as "*cover letters*".

The cover letter is your **elevator pitch**, and your resumé is your **sales presentation**. Specially align the wording of your cover letter, and your resumé with company's **culture**, and the **requirements** of the job.

Back in the days where there was no Internet, people were printing their resumés in the finest paper they'd find, add an additional covering *letter*, and put those two into an envelope, and send it to the *hiring manager* either by themselves, or through a reference.

Those days have gone long before. Yet neither the process, nor the importance of the cover letter has changed at all. It's only the technology and the medium that the message is transferred has changed: Resumés are still either directly emailed to a recruiter, or sent through an insider reference who works in the company; and the cover letter (*i.e., the email message accompanying your digital resumé*) still preserves its relevance.

The cover letter is not just a wrapper for the resumé. It's your "*Hello, is it me you're looking for?*"

A well-prepared "cover letter" gets your resumé read.

There is Only One Purpose of a Cover Letter

Getting your resumé read is the sole and only purpose of your cover letter; and **getting an interview** is the sole and only purpose of your resumé. Therefore it logically follows that **if your cover letter sucks, you won't get an interview**.

Thus, never apply for a job using a generic cover letter. It **severely decreases** the chances of your resumé getting read. Needless to say, never apply **without** a cover letter too; and in your cover letter, be specific in **content** by telling **how** your knowledge, skills and abilities **match** what the company is looking for.

Since your cover letter has to match with the company's needs and wants, you have to read the job posting and research the company to show how your background gets them what they need **specifically**.

A smart person, at this point, may ask "**and** how do I learn about the company's culture and the job's requirements?" Here's how:

Do your homework, and read **anything** and **everything** you can find about the company.
In the information age we're in, where **STFW** is dead easy, it's not an excuse to be unprepared.

Prepare a Brief and Effective Cover Letter

While writing your cover letter, pay extra attention to be **accurate** and **brief**. Three paragraphs is good enough, more than that is an overkill. Also if your paragraphs are more than five or six sentences, it's again too long, and it won't get read. Why? Because executives are **busy** people, and most of them have high D personalities, which means **they want brevity** and anything that's "bubble talk" will be filtered out and will not get read.

Sending a cover letter that does not get read, is worse than sending no cover letter at all.

So how do you prepare a brief, yet effective cover letter?:

First, **state your interest**; explicitly naming the job you are interested in and how you learned about that job. Then, describe exactly **how** your skill set will help **the company** achieve its goals. **Relate** what you've done so far with the job's **responsibilities**. Make it **authentic** and **personal**.

Have a Distinct Cover Letter for Every Job You Apply for

Don't be lazy, include a different **cover letter** with each of your applications, tailored specifically to the application you're applying for; you'll see that the time you spent will pay off. Keep in mind that if a hiring manager has hundreds of applications in her inbox (*that's* **always** *the case*) a well-prepared cover letter will jump you straightly in front of the line.

If you are emailing your resumé, your mail body is the cover letter.
If you're applying through a website the "additional information" text area is the cover letter.

Write each cover letter **from scratch**. **Don't** cut and paste content from previous cover letters and personalize later.
If you take your time and start with a blank cover letter, you'll see that you will express yourself better in terms of how you fit the job's requirements and responsibilities. If you are filling a *"I read your job posting on -blank- and I'm really excited to learn that -the company name- is hiring a -the position-..."* template, **delete** it immediately.

If you are not willing to **invest** a couple of minutes to your cover letter,
then why should the company invest any time to learn about you?

And there's no shame in expressing your excitement. There's no harm in telling the company that it's your dream job and you can do anything to get it (**if** *that's true*). Appearing cool and hiding your excitement is **not effective**.

Don't try to be cool. Be **effective**.

Have a Distinct Resumé for Every Job You Apply for

Creating a different resumé for every single job you apply for is crucial. Every job requires different skills and traits of yours; and including every single thing you have done in your resumé is **strategically incorrect**. If you do so, you waste valuable space and you dilute your resumé by evening things out with the skills that are not required for the job. Indeed, you should **focus** on **what the job position requires**, and create a resumé that best matches those requirements.

Keep in mind not to lie in your resumé, as they will ask about any single thing you've written on it; and your lies will come back to bite you.

---

jobs@company.com is an Alias for **/dev/null**

For heavens' sake, don't you even think of sending your resumé to hr@foo.com, or jobs@baz.com. Those predefined emails are bombarded with junk mail and have special sorting and filtering systems. So your resumé will be in the line of a thousand other resumés waiting to be read.

Do your homework:

- ★ **Find an insider** and send your resumé through her; **LinkedIn** is a great tool for those.
- ★ Or find a way to introduce yourself to the **decision-maker** (*i.e., your potential boss*), who will then send you through the company's recruiting pipeline himself (*this will make sure that your resumé is at least read*);
- ★ Or find and **contact the recruiter** in charge of finding talent for that company.
- ★ And before anything; **start with your personal network**. (*You may not have a direct link to the decision-maker, but may be a friend of your friend's friend may have*).

There is no Such Thing as a "Perfect Match"

Unlike what most of us think, **there is no such thing as a "perfect match"**. You will be learning some of the skills that your job requires along the way, mostly in the first couple of months of your employment.

If you **honestly** think that you can do **80%** of the things that are listed on the job post, then go for it and apply for that job.

Know Things Happening Beyond Your Desk

Repeatedly using "I don't know", "I was not responsible for that", "I am not quite sure how that worked out" will indicate that you have **insulated** yourself inside your cubicle and you are unaware of the dynamics of the project you are working on.

Have **curiosity** and learn what others are doing in your company. Showing a lack of a holistic view of your current job, also means that you are **not** inclined towards collaboration. It's a **red flag**.

Be Careful With Your Vocabulary

This includes both **"written"** and **"verbal"** vocabulary:

Think twice what you say; and **make sure there's no grammar errors in anything you write**; including cover letters, resumés, "thank you" notes, email correspondences... everything.

Especially on resumés, spelling errors are **inexcusable**, and shows that you are lazy enough not to use the **spel-cheeker** (*pun intended*) of your word processing software. Grammar errors, have some "leeway", since where to use a colon versus a semicolon; or where to put a comma is open to debate even between grammarians; but if you write "your" instead of "you are"; or if you write "there" instead of "their"... then your chance of getting an interview is **not more than zero**.

Avoid passive voice and vague sentences. Be **clear**, **crisp**, **descriptive**, and **quantitative**.

For instance "I like to <*verb*>" is a disastrous phrase to use (written or verbal). It implies you like, but you don't do.

Second, your phrases should be in "first person singular" (**I**) or "first person plural" (**We**).

For example, answering "*what is your style of working with a team?*" question with a "first **you** need to do this, then **you** need to do that..." kind of answer has two mistakes in a row:

★ It implies a **sequence** of events, where your answer should be independent of time;

★ And using **"you"** implies that you don't actually have those traits and you are describing an ideal persona who has them.

Use first person singular ("I do this, I do that, and I do those"); or first person plural ("To work as a team, we do this, we do that, and we do those.")

Get Rid of Your Excu"but"s

I don't know you, and it's crazy;

and *a single conjunction will make you like me, maybe?*

"**But**" is a very **dangerous** word:

★ It makes people think **there's a catch**.

★ It **negates** everything you've said before.

★ It **reduces** the positivity of your argument.

An **"excubut"**, is a positive-starting sentence, needlessly cut in the middle with a **"but"** to convey an **excuse**.
Let's see with an example:

I know that SVG is very important to your project, **but** *I'm starting to learn it.*

versus

I know that SVG is very important to your project, **and** *I'm starting to learn it.*

The former sentence implies that you are a reckless rookie, trying to learn SVG, while the latter implies that you know the importance of SVG in the project; you have taken control of the situation **and** you **are** learning what you need to learn. Hell, you **can** take initiative! **And** you **are** a self-motivated learner. What else can an interviewer want from you?!

Don't Talk About Your Peers All the Time

It's good to have team working skills; and you must give credit to your peers where it's due; and the interview is about **you**. Therefore focus on **your** accomplishments first. Talk about the results you've achieved, how you led some parts of the project, how you tackled with problems.

If you talk about your peers all the time you may be seen as either the least productive member of the team, or you may be seen as a person who's unable to express herself. Either is equally bad.

Like it or not, interview is a **selling opportunity**. It's where you **sell** your **knowledge**, **skills**, and **abilities** the best way you can. Your **contribution** is what makes you **worthwhile**. Keep your career history file up to date to have an in-depth track record of your contributions; and talk about them when the time comes.

Interviews are the only place where being humble and egoless works **against** you.

Don't be a Snob

If you are unwilling to work certain types of tasks, or if you limit what you like to work within a set of boundaries; it's also a red flag. It's okay that you have preferences in working in certain areas; and you **must** acknowledge your willingness to perform any relevant tasks that need to be done.

That's freaking teamwork in the end of the day; and if you are not a team player, you are **out**.

Don't Turn the Weakness Question into a "Positive"

We've covered this before, in the beginning of this book.

To put it simply, **interviewers are not idiots**; and treating them as such will work against you.

Be **honest**, and come up with a weakness that can be improved and won't ruin your chances of getting the job.
And make sure you express what you are doing to improve in that area.

Know What to Say A Priori

You have to share a lot about yourself, and you don't have time to tell your whole life story; so you should do your homework well, and have to know what's important about your background that **most relates** to the **company** and your **job**. Deliver the **key things** that you believe **identifies you**, **separates you from the crowd**, and **differentiates you**.

If they say no, it is not because you can't do the job. It will be because **you've not told them enough** that they are convinced that you can do it. You have to prepare.

Show Your Passion

Demonstrating a lack of energy and passion is a complete "no no". Even if the job you are talking about is not your greatest choice, it's something that you **want** to do. Otherwise, you would not be interviewing in the first place.
Don't look like an apathetic, uninterested individual.

You are obliged to show your best in an interview.
Having a bad night with your significant other will not change that obligation.

Regardless of the source of the apathy, the interviewer will think you are nothing but a waste of time.

If you have **energy** and **attitude** and can express your **interest** and **why** your **knowledge**, **skills** and **abilities** align with the job **requirements** and **responsibilities**, in a **logical** and **coherent** way, you will really **stand out** in the crowd.

Interviewers want to see exciting, and energized professionals, who always have a good attitude at every moment.

Eagerness, **energy** and **manners** are the only way to break the activation energy to keep the interview going. The interviewer will not care whether you are an introvert, or a low key individual; **you just have to be energetic**. Furthermore, the DISC model states that even though it makes you uneasy a bit, **any behavior can be learned** when you **practice** enough. Why not start with a few cues; like having an open body language, **leaning forward**, and **smiling**.

This is a "*virtual reality*" **game**, set up between **you** and **what you desire**. Just get over it. Because you don't care doesn't change the facts. **Learn** to be **energetic**, **proactive**, **open**, and **enthusiastic**; **show your best**; or suffer with the fact that you'll have **a very hard time**.

Motivate Yourself Before the Interview

No matter how skilled you are, a third party who has minimal knowledge in your skill set, questioning your abilities is a stressful process. Moreover, the interviewer **knows** that you are trying hard to show the best of you.

> Don't make them think "If what I saw is the best of this guy, I cannot imagine him in the office!"

No matter how hard you try, you can't seem overly energetic. So do everything to motivate yourself up:

Listen to your favorite music, use a joke to laugh and relax, look at the picture of your child, your spouse and smile.

Don't Take it Personal

She will probably take notes throughout the interview, so don't be put off by the lack of eye contact; and interviewing requires a special skill set too; and some of the people you interview, despite being great technical rockstars, may not be so good at interviewing. So don't take anything personal. **Do not be emotional**. You are there to show the best of yourself, and tell them about the **facts**. That's it.

Don't be Greedy

The job interview is not the time or place to ask about advancement opportunities or how to become the CEO (*unless you are being interviewed for that position, of course*). You need to be interested in the job at hand. Sure, the company wants to see that you're ambitious, and they also want **assurances** that you are committed to the job you're being hired for.

> Don't focus on the future roles instead of the job you are being interviewed for.
> Don't come with an agenda that this job is just a stepping stone to bigger and better things.

Another example of greediness is to follow-up earlier than it's necessary. Yes, you should follow-up, and it's a **step** in the interviewing process. However, if the interviewer receives a resumé submission for a job posting and then get a second email from you within 24 hours asking about the submission; even if you are qualified, it will decrease the odds of you getting hired. Qualified candidates with realistic expectations and an understanding of business acumen. **Don't make this mistake**. Be sure to give someone at least a couple of days to respond.

Check Your Phone and Email Daily

Make sure that your phone has an answering system or voice mail and evaluate your recording. The message should be something like, *"Hi, this is Bob, I'm not available, please leave a message."*.

If it's something like *"Dude, Bob's gone high; I'm not really away, I'm just ignoring you. Whatcha say? Say it right now!"* change it **immediately**! You got the message.

I am sure you do, but I want to remind again: Make sure you check your email **every day** including weekends.
And use an email address that is just your name, for instance bob.parker@gmail.com.
And never use something like _crazybob79_@hotmail.com.

The Subject Line is More Important than You Think

Why? Because your email will be **forwarded** to other people: managers, recruiters, heads of departments, friends of the recipients who are decision-makers in different companies and sectors.

The title should clearly state it's purpose and grab attention to get read.

Which one of the subjects do you think will be opened and read by a hiring decision-maker?

"*Regarding Your Job Opening*"

versus

"*Resume of Volkan Ozcelik - Looking for 'JavaScript Engineering' Positions - Can Relocate*"

An email with the former subject line will be directly sent to the **Later/Maybe** folder.
And the second mail will be **immediately** read, and, more importantly, **forwarded** to relevant people.

Our job is to make the life of the decision-maker easier.
She should understand what the email is all about by simply looking at the subject line.

Here are a few properties of the second subject:

★ **No accented letters** (é, Ö, ç... etc) these may be incorrectly encoded by the email client, of the recipient;

★ Your **name**, followed by **what you are looking for**, followed by **additional details** to help the reader;

★ **The subject line should not be too long** (else it might get truncated by the email client of the recipient).

The **subject line**, and the **content** of your email are important for getting your resumé read.
And without someone reading your resumé, you won't get an interview at all.

Keep Your Professional Network Warm

Nobody but you are responsible for your own career development; and no matter how skillful you are, without a professional network, nobody will hear you. If don't have a professional network up to this point, start building one.

Having a professional network is not just something that's nice to have.
It's a **vital requirement** for extending your hands to great opportunities.

For some reason, people think that asking their network for help is incredibly difficult. **It's not**:
It's basic human *behaviors* and *politeness*.

Make sure to keep in touch with your network. **Reduce the tension before you can ask for a favor**:

Go to professional gatherings, go out to lunches, visit your colleagues' offices... Ask how they are, give them **favors**. Actually **give thrice more than you take**. You cannot have a network ready to help you at the instant you need; you need to prepare it long before.

Networking is more important in the Silicon Valley than anywhere else in the known universe.

Here's another reason why you need to grow a professional network, if that's still not obvious:

Job vacancies fall into two categories: those that are **advertised to the public**; and those that are sometimes known as **"hidden gem"**s – these are not given to recruiters, and only people inside the company know about them.

So first warm up your network. Indeed, **always keep your network warm**, because you will need an active professional network **long before** you'll be in need for a job.

Reconnect with the people in your professional network and touch base for a couple of times before asking for a favor. Remember: **You don't kiss on your first date**.

So begin by **reconnecting**. Then give them some idea of what kind of a job you are looking for, because your professional friends cannot read your mind; and "Bob, I'm looking for a job" is too vague to aid your friend help you.
And **set reminders** in your calendar to **follow-up**, because you are not the only thing that's important to them.

Chapter 8
An Overview of the Behavioral Interview Process

First things first: [there's no such thing as a behavioral interview]: You will always be gauged by your manners during all the interviews (*technical interviews included*) you go for with the company. However, especially in the initial steps, you will have interviews specially designed to measure how you culturally fit to the company.

A behavioral interview generally consists of the following sequence of steps:

- ★ Introduction and First Impressions;
- ★ **They Ask, You Answer**;
- ★ You Ask, They Answer;
- ★ Closing;
- ★ Follow-Up.

Most of the time it's the highlighted part that people think as an "interview".
Conversely, an "interview" extends way beyond that.

Lets analyze each of the parts of an interview in order:

Introduction and First Impressions

In the first five or ten minutes you will be getting to know each other.
The important thing to keep in mind is that **you are in the interview the moment you enter the room.**

You **are** in the interview the moment you apply for the job –
so I assume you are wise enough to **"sanitize" your online presence** and publicly visible data.

Sanitizing your online presence is more important than you may think: The first step in landing an interview is get past the screens. Screeners are humans who examine the applicants. Most of them google you to learn more about you. Before even applying for a job, sharpen and sanitize your social profile: Check your public profiles to make sure that there are no surprises that can turn off a potential employer.

Furthermore, **do not** consider the introduction step as an unimportant chitchat phase:

Most probably you will be talking about recent local, global, and industry news. Make sure you follow recent **global** events (*disasters, tournaments, politics...*), recent **local** events, recent **industry** events (*conferences, meet-ups, hack challenges...*), and recent **technical** happenings in the sector (*a new version of a framework, that new cool iPhone that everybody's talking about...*). In short, be aware of the world around you: You're living in it.

Being a geek doesn't give you an excuse for not knowing what's happening in the world around.

And remember to **show energy** and **smile**. Smiling is the least you can do to make the **most** difference. Try it, you'll see.

Their Questions, Your Answers

That's what most people **incorrectly** think what the interview is all about. If you are one of those, you are **wrong**. This is just a **part** of the interview. If you succeed in this part and fail in the other parts, then you don't have a chance. We will be specifically looking at "the Most Important two Interview Questions" in a following section.

Your Questions, Their Answers

After you've answered their questions, it will be your turn to ask questions. This is often an underestimated opportunity. This is the phase that not only you know what you are doing, but you also can lead a conversation and take initiative.
We will cover this in depth in "Your Turn to Ask Questions" section.

Closing

At the end of the interview, the interviewer will want to inform you about what will happen next. If she does not, do not hesitate to ask. This is important, and unlike written in most of the interview articles, this is **not** closing.

Closing begins when the interviewer finishes what she has to say.

Closing is the only time in the overall process that you have **full control** and can **truly** make an impact.
Yet, most of us are so afraid that we don't use this *hidden gem*.

Interviews are not for the faint-hearted. **Be brave!**

Closing is where you **thank** the interviewer and express (*again*) **why you want an offer** (*this should be based on your skills, characteristics and traits;* **not** *on how you desperately need the job*), and **why you are a good match for the firm**.

This is very important. Of course you want an offer, otherwise why should you go for an interview in the first place; and firmly expressing it in the closing of **every interview** is neither bragging, nor whining. It's a **bold** action that, if done correctly, will differentiate you **miles apart** from the others. Remember, you'd **use every opportunity** to your advantage.

Acting cool will not help you at all. Being **effective** will.
When the time comes, have the **courage** to **close that interview**.

Follow-Up

> Once it's over, it ain't over.

You are **expected** to do certain things. One of them is to **thank** your interviewer with a letter or an email at least; and the other is to **follow up** after a couple of days asking how the process is going.

Thanking the interviewer is *de rigor* and you are considered an unprofessional new kid on the block (*to say the least*) if you don't.

And contrary to popular belief, following up after a few days has passed does not show you're weak, or you're hopelessly in need of the job. Instead, it shows that you are an adult, not afraid of expressing her interest in the company.

Any organization should be professional enough to update the candidates once the hiring process is over.
So it's not abrupt to regularly ask about the process until you get an answer.

Shift Your Paradigms

The interview extends **before** and **beyond** the time you talk with the interviewer.
Interview starts at the time you contact the company, and continues until you get an offer.

Following up is also a part of the interview. If you haven't received an answer from the company, it means that the company has not made up their mind, yet; and following up maximizes your chances for an offer. If you don't hear from the company, follow up **each week** until you get a definite response from them.

Chapter 9
Sharpen Your Katana **Forever**

Self improvement is a **lifelong** process.

Improving yourself only during your job search is **the most ineffective action** that you can take. Always, and continuously improve. You can start by reading every reference material in the bibliography of this book, to begin.

Second, know yourself, know the market, know which technologies are hot:
Develop marketable and transferable properties.

Third, have a continuously evolving **development plan** for yourself.
Creating a self-development plan is similar to creating a software project plan.

We've seen before that optimization without measurement, is a seductive trap.
So follow these steps not to fall into this trap:

- ★ Conduct a **requirements analysis**: Create a **list** of what you need to **learn**, what you need to **practice**, and **prioritize** that list by the **importance of the skill** to the job roles you are looking at, and **ease of learning the skill** to the level that the job requires; respectively.
- ★ **Practice** your **existing skills; learn new skills**. Learn one (*at most two*) skill at a time. Once you pass the **benchmark** that indicates that you know **enough** of that skill **for the job requirement**; continue your studies with the next skill in the list.
- ★ Corollary to the above line: Devise a **benchmark** to measure your progress.
- ★ **Rinse and repeat**.

This is a **lifelong** practice. **Do it**, even if you are not seeking for a job at all.
You will get noticed by being better at things that make you "**promotable**".
And, in case of an interview, you will have a hell of a lot of things to talk about.

If you've worked in the same company for many years, your experience may be seen as "*not transferrable*", meaning that you can do an excellent job at your current company, but when given a different set of tools, techniques, and team members, you may come across a steep learning curve. That's the interviewer's **subjective** feeling of course; and it's the interviewer who's in charge of the interview.

So how can you make the interviewer feel **positive** about you? By constantly **improving** yourself; By regularly writing at your technical **blog**; by **sharing** your **open source** code, by joining in open source projects, reading others' code, and

contributing to what others have written; by **collaborating** with like-minded people; by going to developer **meet-ups** and hack **events**; by having hobby technical projects; in short, **by having a geek reputation outside your office**.

Extend your professional reputation out of your office, so that you don't have a hard time convincing the interviewer you are worthwhile.

Fourth, **always keep your resumé up-to-date**. Your resumé is **not** a piece of paper that you just update before a job search. It's a **subset** of your career history file. It should be perfect at all times. We'll come to that in the next section.

The last, but not the least, have an in depth knowledge about at least some subset of your overall skill set:

Bouncing from one job to another, and accumulating knowledge by osmosis in different areas, and therefore being a "jack of all trades" is a good trait. Having experience working in diverse environments is definitely a positive, **if and only if**, you also **gather deeper skills** and experience along your way.

Focus not only to gain breadth, but also focus to gain depth.

Chapter 10
How to Create a Killer Resumé

I have a inalterable rule that a resumé, even in the IT sector, should not exceed one page. So here's what I do:

I have a **career history file** I keep up to date at all times:
It includes all my **jobs**, the **responsibilities** I had, and what I did to **accomplish** my goals.

Since your resumé should be a single-page document, you'll need to **cherry-pick** from your achievements; and given that you also have to create different resumés for different job positions, it logically follows that you should store those achievements **somewhere**. You don't need a fancy database, or any GTD porn software. Indeed **a plain text file** is the best: It's a universal format where you can quickly edit share and update anywhere without needing any special software.

Update your career history text file **every 3 months**, or sooner, **religiously**.

If you add items to your career history file on the go, when the time strikes (*and it does strike at the most unexpected time, due to Murphy's laws*) you will have an excellent resumé in no time.

And **no**, your git commit logs are not the most efficient way to record your achievements.
And, honestly, when's the last time you've entered a meaningful message to your source control system's commit log?

...

Every three months, I update my career history file. If I do something great along the way, I update it earlier. Nonetheless, I've **set up reminders** to quarterly update the document.

Even if I don't have anything to add, I read what I have written carefully and make corrections, if needed.
And without exception, I always find something to add.

If you have not prepared a career history document before, and you think all the information will magically pop out of your brain; there is no such magical wand to make it happen. So you have to do it in traditional, non-magical ways. There is much heavy-lifting to be done:

Spare at least **a weekend without distraction** to create your first career history file.

In fact, in the last 30 years, companies have turned the responsibility of managing peoples' career to (*guess whom*) people themselves. Therefore you, and only you, are responsible for your career path. You have to capture and document every single accomplishment with as much detail as possible. So if you have not already done so, you have to go back to your past and review your professional history **right now**.

Seriously, if you don't have a career history file, close this book right away and create one.

You will be asked about your accomplishments, and your resumé is built of your accomplishments. Your accomplishments are your stock in the trade, if you will, in the free job market. They are the supporting **evidence** to **prove** whether you're capable enough to get the job you want.

These Are not the Droids You're Looking For...

Would you persuade, speak of interest, not of reason. – **Benjamin Franklin**

Your success in any interview (*behavioral or not*) highly depends on your **persuasive ability**. Let me **persuade** you why:

You know everything you have done so far, right?
Yet, your interviewer has only briefly overviewed your projects (*if you are lucky*) ;
and has gazed at your resumé for a minute (*if you are lucky too*).

Contrary to popular belief, **your resumé won't get you hired**.
The goal of your resumé is to get you an interview.
And once you are in the interview that goal is **over**.
And **you are on your own** to **persuade** the interviewer.

How the interview goes – not your knowledge, nor skills, nor abilities – is what matters.

Yes, it is not fair, and that's life. **Welcome to the real world of adult decisions and consequences**.

This is already covered in the beginning of this book. It's what the interviewer thinks what you've done, that's important.
Your **success** depends on your **persuasive abilit**y of how you express the facts (*without bending the truth*, of course).

Being **honest** and **persuasive** requires great **preparation** and **practice**.

Which also means that if you want to nail that interview, you have to **practice** and you have to **be prepared** at all times; because your "**persuasive ability**", which your **success** depends on, *depends on* how prepared you are.

Persuaded?

Good.

Don't Let Your Resumé Collect Dust

> What the interviewers want to see is simple: **An outstanding resumé**. If you have a resumé that **stands out** so much, they will talk to you even if your skill set is not quite aligned with the job at hand. They won't care, and take a chance to talk to you (*and that what your resumés sole purpose is, in the first place*) because your resumé will sing to them that you are a **professional**.

The worst thing you can do is let your resumé collect dust (*most probably because you have been at the same job for a while*). **You never know what will happen tomorrow**. The past few years have shown me through the experience, and I learned the hard way, that you can get laid off in a minute and then you are left scrambling to get a new job.

At that moment, if you have an outdated resumé; it's a headache trying to get it up-to-date in that level of stress. Additionally, since you have not kept your track record for a while, you've probably forgotten some of the things you have done: You can't remember dates, you can't remember what you did... and so on.

> That's why you should try to **update your resumé at least four times a year**.

In this way, you have less to remember when you decide it's time to make a career change.

In the former section, we've discussed that you should have tailor-made resumés for every job you apply for.
And we've not dived into the "how to"s of it. Let's expand the topic a little:

Once a manager picks up your resumé, it has a **ten-second lifetime** on the average. Why? Because your resumé is one of the 200-300 or more applications that the manager receives daily, and sifting through resumés is **only** a part of her job.

> If your resumé does not get past that first ten seconds barrier then it's gone to the **Later/Maybe** folder to be reviewed "**later**" (*which, I call the "once in a couple of months" folder*).

Don't let your resumé fade away. The goal of your resumé is to get an interview "at that moment", not after an indeterminate amount of time. To have an excellent resumé, make sure you update your career history file regularly.

Content is the King (*in Resumés too*)

The bottom line is: **Make your resumé stand out in the crowd**. No, not using that shiny new template.

And for God's sake do not use an animated resumé or a video resumé, you are a developer, not a Hollywood star.

You want your **professionalism** to stand out; not your new cute template.

The only way to make your resumé shine is through its content, **not** through its context.

Make Sure Your Achievement Bullets Make Their Points

Your achievements define you. Get them ship shape.

Don't pay anyone to fix your resumé. If you can type, you can create a decent resumé. Here's how:

Stress your achievements in your resumé; not simply list your past employers and job titles.
Be **quantitative** and **specific** in expressing your achievements.

There's mountains and valleys of difference between

> *"Implemented a spam filter, using industry best practices, Enterprise Java Beans, and Microsoft Biztalk Services."*

versus

> *"Increased inbox deliverability rate from 30% to over 90% in two weeks, designing a new email bounce-back system;"*

The second one will immediately attract the interviewer's eyeballs; and he will ask how you did that in the interview. And there you can explain what cool technologies you used to implement the solution.

Here are the three important things to keep in mind when writing achievement bullets:

- ★ **Be quantitative**, give numbers whenever you can.
- ★ Use **action verbs** (*like "implemented", "created", "researched", "developed", "managed", "mentored"...*)
- ★ **Give only the necessary details**. Too much information will degrade the value of the message you convey.

Action, Result, Method;

Use "**action**, **result**, **method**;" as your template; and you will never fail:

"**Increased** *inbox deliverability rate* **from 30% to over 90%** *in two weeks,* **designing a new bounce-back system;**"

Action **Result** **Method**

A Resumé is not a CV; Don't Tell Your Whole Life Story

CV, or *Curriculum Vitae* can be translated from Latin as "the course of my life". It's a boringly **detailed** document about everything and anything you did in your professional life. That's how a **career history file** is created especially in European countries. In contrast, an American-style resumé is a more distilled version:

A **resumé** is a marketing brief, where the product that's marketed is "you", so to speak.
Whereas a **CV** is akin to **pages and pages of** business plan.

You don't need an "executive summary" in your resumé, you don't need an "objectives" section either.
Both of them are nothing but useless wasted space. Besides, **a single page does not need a summary**.

Don't waste your precious time and energy crafting a "summary". Your achievement bullets should summarize your career in the blink of an eye. Spend your time (*and space*) on them. It's the most important part of your resumé;
make sure the achievements are **quantitative**: show **your ability to be successful at areas important to the company**.

Three simple questions will help you in this. For each bullet, ask yourself:

★ **Why** you did the activity;

★ **What** the definition of **success** in that activity is;

★ And **how** you can **qualify** and **quantify** that success.

And then incorporate your answers into the achievement bullet.

Be a minimalist, when it comes to formatting: **No tables**, **no lines**, **no horizontal rules**, **no indentation**.
The more formatting your resumé has, the more it will mess up with the companies' applicant tracking systems.

Your resumé does **not** need to look fancy with sidebars, headings, images, cute fonts and all that jazz. Instead, a plain vanilla, single page resumé, written with a **10pt Times New Roman** font, with valuable **content** will **shine** amongst all the "eye candy" templated, –yet empty – resumés.

On a Single Sheet of Paper, Whitespace is Gold

Don't bloat your resumé. It should adequately summarize your career history, and it should be **at most one page** (no exceptions to this rule of thumb – to this date I'm writing these lines I have been in nine different jobs in the recent ten years of my professional life, and my resumé is still one page, and every single recruiter sees it, thanks me for how brief and well-prepared my resumé is).

If you have a two-page resumé, just know that **the second page will never be get read**.

Okay, if you're the CEO of Apple Inc., you can have one and a half page; but two pages is more than necessary, and more than two pages will send your resumé directly to the shredder.

Given the "one page" rule, whitespace becomes of premium value in resumés. Therefore each achievement bullet in your resumé should not exceed one line. Which forces you to restrict the details you are going to give, and that's a good thing. Here's why:

A **resumé** is like the smalltalk you have at a dinner with your date. Make sure you leave some **mystery** for the next level. Write enough information to **make her wonder** and ask further details. And don't give too little information that the interviewer concludes you're not worth her time.

It's a delicate balance to make, and the **one page rule** ensures that balance.

...

The **achievement bullets** are the heart of your resumé. Spend a considerable amount of time on them and get them ship-shape. Keep in mind that **you** are responsible for your achievements, and you'll be asked for **specific details** of them in an interview.

Use a single sheet of paper.
If you use two, the second one won't get read.
If you use three, your entire resumé will be skipped, and it won't get read.

Since space is invaluable in a single sheet of paper, we should not waste it.

Whitespace is overrated, it means **nothing** on a resumé.

Get rid of the "*skills and tools*" section (*i.e., the list of all the cool programming languages that you know*).
Your professional experience, and your **achievement bullets** should reflect your skill set already.
The languages you use, for instance, should be included in **responsibilities** body of your particular job descriptions.

Similarly *"professional profile"* and *"executive summary"* are nothing but bubble talk. Remove them too. Heck, it's not a marketing report you are preparing. It's simply ridiculous for a single sheet of paper to have an "executive" summary.

Remove the *"objectives statement"*. It is a waste of space and limits your choices. What if your objective is "to **collaborate** in a team-like environment" (*a bubble talk you can see in 80% of all "objective" statements*); and the employer of your dream job does not extend you an offer because he wants a strong, self-motivated, jack of all trades **individual**, who can do his task **on his own**. Or what if your objective is to work in a "startup", and your prospective employer is a big firm with a startup-like culture (*which is the best of both worlds*); offering you much better benefits (*like an entire library of books to study, free premium tickets to developer events, 1/5 of the company time to develop your hobby projec*t) and decides not to hire you, thinking that you will be happier with a core team of less than ten individuals. Remove your objective statement; you won't lose anything. Trust me.

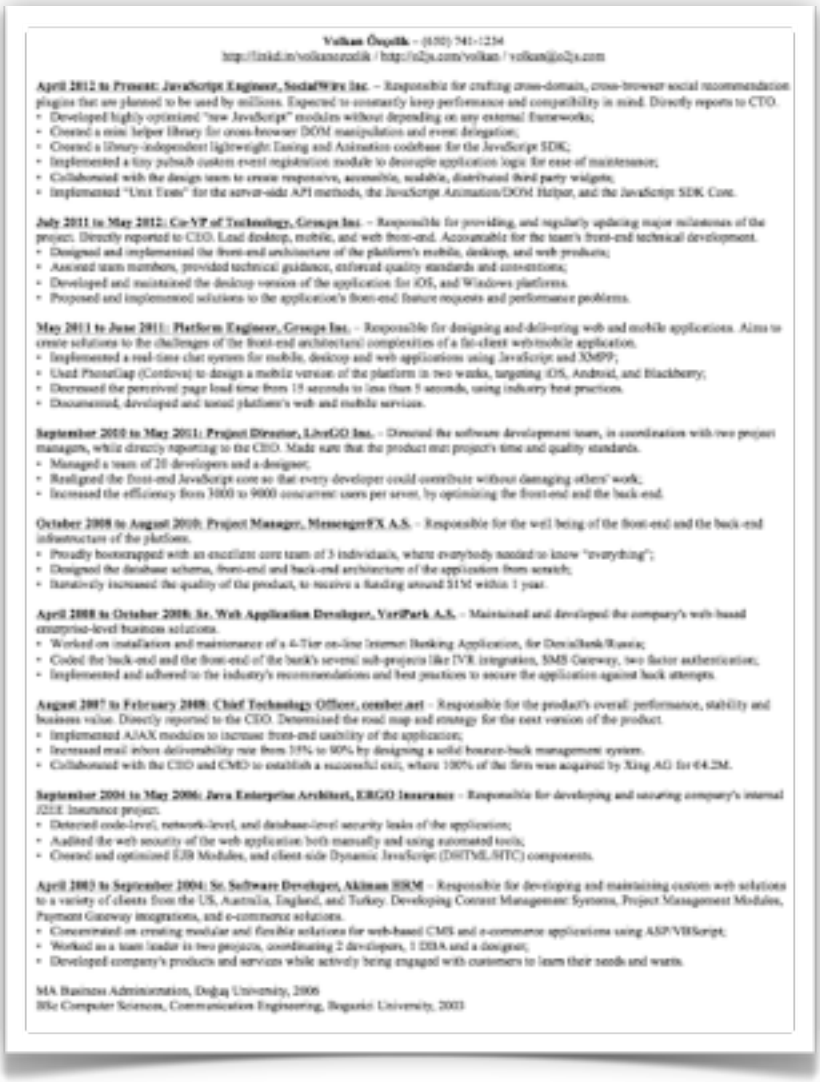
The objectives statement is dead. **Seriously!** – Pull it off of your resumé. Use that space for more relevant things, such as your **accomplishments**, to help you land the job you're applying for.

And I beg you on my knees, for God's sake, please do not include your hobbies, and references. The line "*references available upon request*" is a complete waste of space. If the interviewer **wants** a reference she **will request** it even if you don't hint in your resumé, and they will get it. Even they may ask for them along with your resumé upfront.

Which leaves the **majority** of your resumé to your **job history**. The only places that are not job-related are your contact information (*name, address, phone, and a non-work email, centered and bold at the top*) and your academic information (*a single line if you are a university graduate; two or three lines if you have MA and Ph.D.*), and that's it.

Use a single text block for each job; and separate each text block with a single **carriage return** and an empty line. Someone looking at your resumé from **ten feet** should clearly see how many jobs you have been to (*that "someone" can be a coworker of the hiring manager; while the hiring manager has left your resumé on her desk among a pile of other things*)

Here's how that kind of a resumé looks like from ten feet:



Volkan Özçelik - (650) 741-1234
http://linkd.in/volkanozcelik / http://n2js.com/volkan / volkan@n2js.com

April 2012 to Present: JavaScript Engineer, SocialWire Inc. – Responsible for crafting cross-domain, cross-browser social recommendation plugins that are planned to be used by millions. Expected to constantly keep performance and compatibility in mind. Directly reports to CTO.
- Developed highly optimized "raw JavaScript" modules without depending on any external frameworks;
- Created a mini helper library for cross-browser DOM manipulation and event delegation;
- Created a library-independent lightweight Easing and Animation codebase for the JavaScript SDK;
- Implemented a tiny pub/sub custom event registration module to decouple application logic for ease of maintenance;
- Collaborated with the design team to create responsive, accessible, scalable, distributed third party widgets;
- Implemented "Unit Tests" for the server-side API methods, the JavaScript Animation/DOM Helper, and the JavaScript SDK Core.

July 2011 to May 2012: Co-VP of Technology, Groups Inc. – Responsible for providing, and regularly updating major milestones of the project. Directly reported to CEO. Lead desktop, mobile, and web front-end. Accountable for the team's front-end technical development.
- Designed and implemented the front-end architecture of the platform's mobile, desktop, and web products;
- Assisted team members, provided technical guidance, enforced quality standards and conventions;
- Developed and maintained the desktop version of the application for iOS, and Windows platforms.
- Proposed and implemented solutions to the application's front-end feature requests and performance problems.

May 2011 to June 2011: Platform Engineer, Groups Inc. – Responsible for designing and delivering web and mobile applications. Aims to create solutions to the challenges of the front-end architectural complexities of a fat-client web/mobile application.
- Implemented a real-time chat system for mobile, desktop and web applications using JavaScript and XMPP;
- Used PhoneGap (Cordova) to design a mobile version of the platform in two weeks, targeting iOS, Android, and Blackberry;
- Decreased the perceived page load time from 15 seconds to less than 5 seconds, using industry best practices.
- Documented, developed and tested platform's web and mobile services.

September 2010 to May 2011: Project Director, LiveGO Inc. – Directed the software development team, in coordination with two project managers, while directly reporting to the CEO. Made sure that the product met project's time and quality standards.
- Managed a team of 20 developers and a designer;
- Realigned the front-end JavaScript core so that every developer could contribute without damaging others' work;
- Increased the efficiency from 3000 to 9000 concurrent users per server, by optimizing the front-end and the back-end.

October 2008 to August 2010: Project Manager, MessengerFX A.S. – Responsible for the well being of the front-end and the back-end infrastructure of the platform.
- Proudly bootstrapped with an excellent core team of 3 individuals, where everybody needed to know "everything";
- Designed the database schema, front-end and back-end architecture of the application from scratch;
- Iteratively increased the quality of the product, to receive a funding around $1M within 1 year.

April 2008 to October 2008: Sr. Web Application Developer, VeriPark A.S. – Maintained and developed the company's web based enterprise-level business solutions.
- Worked on installation and maintenance of a 4-Tier on-line Internet Banking Application, for DenizBank/Russia;
- Coded the back-end and the front-end of the bank's several sub-projects like IVR integration, SMS Gateway, two factor authentication;
- Implemented and adhered to the industry's recommendations and best practices to secure the application against hack attempts.

August 2007 to February 2008: Chief Technology Officer, comber.net – Responsible for the product's overall performance, stability and business value. Directly reported to the CEO. Determined the road map and strategy for the next version of the product.
- Implemented AJAX modules to increase front-end usability of the application;
- Increased mail inbox deliverability rate from 35% to 90% by designing a solid bounce-back management system.
- Collaborated with the CEO and CMO to establish a successful exit, where 100% of the firm was acquired by Xing AG for €4.2M.

September 2004 to May 2006: Java Enterprise Architect, ERGO Insurance – Responsible for developing and securing company's internal J2EE Insurance project.
- Detected code-level, network-level, and database-level security leaks of the application;
- Audited the web security of the web application both manually and using automated tools;
- Created and optimized EJB Modules, and client-side Dynamic JavaScript (DHTML/HTC) components.

April 2003 to September 2004: Sr. Software Developer, Akiman HRM – Responsible for developing and maintaining custom web solutions to a variety of clients from the US, Australia, England, and Turkey. Developing Content Management Systems, Project Management Modules, Payment Gateway integrations, and e-commerce solutions.
- Concentrated on creating modular and flexible solutions for web-based CMS and e-commerce applications using ASP/VBScript;
- Worked as a team leader in two projects, coordinating 2 developers, 1 DBA and a designer;
- Developed company's products and services while actively being engaged with customers to learn their needs and wants.

MA Business Administration, Doğuş University, 2006
BSc Computer Sciences, Communication Engineering, Bogazici University, 2003

That resumé will shine like a star on the hiring manager's desk.

As you can see from the sample image on the former page, the most important part of any resumé is the "job history". Each job history "**block**" starts with the date of the job, your title, and the company name, **all <u>underlined;</u>** and that's the **only** thing that's underlined in the resumé which helps clearly and visually separate job blocks from each other.

Then **without adding a newline** the "key responsibilities" part follows. Your responsibilities is almost a rewrite of your job description at that company. If you don't have a formal and written job description, ask your employer for one; and then comes the achievement bullets.

Responsibilities and **achievements** are completely different things; don't mix the two. Your achievements are "what you do" to **actualize**, the tasks and responsibilities you had in your job.

Make it clear what your job is (*i.e., your responsibilities*) and how well you did your job (*i.e., your achievement bullets*). Choose your wording carefully, so that it is easy for the interviewer to see those areas distinctly.

Your resumé is **not** a plain advertisement paper; it's a **coherent** and **linked** career history document.
And it is meant to draw a picture in the mind of the decision-maker.

Also do not indent anything. It's a waste of precious space. Flush everything to the left margin.

Indentation is overrated. Flush everything with the left margin; gain some precious whitespace.

At the bottom of your resumé goes the education information.
Each degree in a **single line** (BS or BA, then the major, followed by the school, location and your graduation date).

To repeat, you don't need to do fancy makeup for your resumé.
A well-written single page resumé will attract the eyeballs of everyone.

I want to stress the **responsibilities** and **accomplishments** parts again; because it's where most people get confused:

> **Responsibilities** are **what** you did; **accomplishments** are **how well** you did it; don't mix the two.

And don't you ever think that your resumé will get you a job.
The only thing a well-written resumé will get you is "**an interview**". Use that chance wisely to get closer to a job!

List your jobs in **reverse chronological order:** Four or five achievement bullets in the most recent one, and three or two bullets in the rest of the jobs. Of course you have done a lot more than that during your job history; and limiting the number of achievements will force you to **cherry-pick** achievements that are the most aligned with the requirements of the job that you apply.

Your Resumé Will Be Printed

Believe it or not, a printed copy of your resumé will be filed and distributed among the company. Most HR departments are "conventional" in that regard. That's another reason for you to confine your resumé to a single page: The second page may get lost, or torn apart. It's simply not worth taking the risk.

Also, use a serif font (**Times New Roman** is a safe choice). Serif fonts are specially designed to be readable on the paper.

A ten point typeface is big enough. You can use 11 or 12 points if you are early in your career. Don't use a font larger than 12 points anywhere in the document, except for your contact information at the top; and even there a 10 point font is okay if you need a couple of extra lines for your achievement bullets.

Chapter 11
The Interview

If you do not go after what you want, you'll never have it. If you don't ask, the answer will be always no; and if you don't step forward, you will always be in the same place. So, go after your dreams, know what you want, and step ahead.

The people who get ahead in this world are people who get up and look for the circumstances they want; and if they can't find them, make them.

George Bernard Shaw

You would be extremely lucky to go for an interview completely unprepared and sail through by a combination of your sparkling personality and endless wisdom (*which no doubt you* **do** *possess*).

Remember, **this is not a game** (*unless you see yourself in a never ending "virtual reality" game of consecutive interviews – been there, done that*). It's not a game: There are no second chances in an interview; no retries, no extra lives...
Hence, you have to be top notch, and be prepared to talk about yourself in a **professional** manner.

This is not the time to hide your light under a bush. It's a selling situation.
You are selling your **knowledge**, **skills** and **abilities** from the moment you apply for the job.

And before even applying for a job, it's advisable to do some **self reflection**. Observe yourself while you're at work, and make a **SWOT analysis** of yourself. See what you are good at, and which areas you need to improve. This is a potentially life-changing decision after all. See "Preparing for an Interview" section for more details on how to do this.

And there's one more thing:

What happens during the interview usually depends on the job position you are applying for, too. The company would take different approaches for entry-level, and senior-level positions; and they would prefer to err on the side of caution (*i.e., they'd prefer not to hire a good candidate, than to hire an awful candidate*).

From a certain perspective it's like gambling. Don't take any rejection personally, and continue to improve your skills by building something real (*hint:* **open source**).

Preparing for an Interview

Interview is a **sales opportunity**, where the product being sold is your **knowledge**, **skills** and **abilities**; and the customer is the interviewer. To be good in selling the product, you should know what the customer wants beforehand. Okay, you are being recruited for a job; and when you dive a bit deeper, you'll come to a conclusion that if you **know** how the interviewer thinks, then you will be one step close to convincing her. It's really as simple as this:

> The interviewer is **literally** trying to close her eyes and picture you doing the job.

Chances are that you may not have done exactly the same job: the only thing in the head of the interviewer is the skills and traits the job requires; she'll be trying to relate what you are saying to how they fit to the job's requirements **all the time**.

> You have to convince the interviewer that you have the skills that matches the job's requirements.

It's that simple!

And the next question is, how do you learn about what the job requires:

An obvious starting point is the job advertisement. Read it carefully, and note down what's expected from an ideal candidate. However, that's not enough. Read job descriptions of similar roles in different companies. Take notes about the backgrounds, projects, presentations, articles, contributions, and strengths of the people working in similar roles. Consolidate all you've gathered under **skills**, **abilities**, **experience**, and **qualifications** categories. This way, you'll have an extensive knowledge of what the job requires, which parts of the job strongly match your skill set, and which parts of the job you need to further improve yourself to be on par with the position's requirements.

This will help you **formulate** a basis for a strategy to sharpen your skill set as quickly as possible.

I know it's a tough job. If it were easy, everybody would be working in their dream jobs; and being technically prepared is just a part of the equation. You have to **behaviorally prepare** yourself, **rehearse** your elevator pitch, **review** your **professional history** repeatedly[8] -- Don't worry, you won't seem "rehearsed" because under the level of stress you are at, you will eventually be forgetting things, adding bits and pieces that come to your mind at that instant: It will seem **well prepared** and **natural**.

[8] I suggest making a text file that tells about yourself and your greatest challenges and read it **thrice a day**, until it become second nature.

Remember; the worst answers are those given with little or no preparation. You have to think about your answers ahead of time. Practice your career history, challenges, and achievements until you can deliver **any** answer without thinking.

You'll see that once you know all the pieces of the puzzle, it's just combining different pieces and deliver an answer that the interviewer **wants**; and until you reach that level, you have to work on your background **every single day**.

You know, it's **your** background; **you** are responsible to deliver any piece of it effortlessly, **any time**.

> An interview can happen at **any time**, it does not need to be a formerly scheduled meeting at the headquarters of a company. Even if you don't think it is an interview, it does not change the fact that it **is** an interview. Don't be surprised when that causal lunch with a colleague of yours turns out to be an interview.
>
> What if all of a sudden and out of nowhere she asks: "*hey John, I heard that you are good at JavaScript; can you tell me what's the hardest JavaScript challenge you've solved ever?*".
>
> Are you ready to answer that question after two margaritas? What if the answer to that question would turn out to be a contributing factor in the company calling you to a "formal" interview? Would you have spent some time to be prepared?
> **Always be prepared**.

Always try to see the questions through the interviewer's eyes.
What is it that they want, why are they asking the question?

Review your professional **history** and identify all of your **accomplishments** that align with the company's **requirements**.
If you follow this logic, you will need to highlight different sets of accomplishments for different jobs you apply for.
Which, in turn, means that you should have separate cover letters, and resumés for each job you apply for.

And don't assume, since you write those things down, they will be magically engraved into your long term memory.
To put something into the long term memory, there's only one tried and true way: **Practice every freaking day**. Ask behavioral questions to yourself, and answer them, **record your voice**, listen to it after a couple of days. Rinse and repeat.

Recording your voice and listening to it after a day may feel awkward initially;
however, it's important, and you'll love this technique once you get used to it.

If you spend half an hour each day, you will be the most prepared person the interviewer has ever talked to.

Common Interview Mistakes and Misconceptions

No matter how prepared you are, you have to pay special attention to certain things in an interview.

This section covers a list of common interview mistakes. Some of these may seem obvious. Yet, knowing that something is wrong does not necessarily mean that you don't do these in the interviews.

While reading this section, think of your former interviews and **visualize** whether you've fallen into the traps listed here.

Not Listening to Questions

To put it simply, if someone asks you a three part question, the answer should have three parts; and your answer should have exactly three parts **in the same order** as the question's.

When answering a question; do not be emotional, just state the facts instead. Do not start your answer with an excuse.
If you want to give some supporting evidence about why you did what you did; first answer the question and then give your supporting evidence. **Not** the other way around.

Answer what the interviewer asks **directly** and **specifically**. Answer the interviewer's questions and **then** tell what you think is important. Don't do this in reverse. It's the biggest mistake in answering interview questions.

Ignoring the Interviewer

You don't outrank the interviewer and you don't control the interview process. Do not try to **deflect** their question by asking another question. **Do not try to outsmart the interviewer**. Do not try to run, or hide.

> There's only one piece of the **process** that's totally under your control: That's the closing. Until that time comes you're not in charge. Acting **"as if"** so, is the dumbest action you'll ever take.

Talking too Much

Don't throw the kitchen sink at the interviewer. Know what your answer is. Think about how you will convey your answer first. Answer what's being asked, and elaborate on the subject if necessary; and **then be quiet**.

> Know what the interviewer wants to learn and deliver your answer professionally.

A prepared candidate **thinks**, **delivers** and then **shuts up**.
The unprepared tries to talk themselves out of their mistakes.

Going Down Into the Rabbit Hole of Mistakes

As we're on the subject of mistakes, know that there's is no perfect interview. Everybody makes mistakes in interviews. It's inevitable; but after having answered a question, if you think "*oh boy! I blew that*" and you spend the next few minutes distracted, not paying attention to the next questions, you'll mess up with the rest of the interview.

It's okay to make a mistake; build a bridge and get over it.

Everyone makes mistakes; and the interviewer knows that you are human and can make mistakes.
When you make a mistake, just let it go.

Thinking that You Are in Total Control of the Timing

Interviewers ain't no dumb. Do not attempt to influence the timing of an offer in advance. Ever!

Do not reschedule interviews. Always be responsive. Return calls and emails as soon as possible.

Never mute your interest, it's dumb.

Your purpose, is to get an offer. Until you get a bit more control over the situation (*that's when you* **have** *the offer*), **do not** attempt to slow things down, or speed things up.

A Promise is **not** an Offer

This one is a conceptual mistake, because you are a human and you want to reduce stress and uncertainty, you'll exaggerate any positive signal you see.

The purpose of an interview is to get an offer. A promise of an offer, a **suggestion** that you are a strong candidate, even the promising sentence "we will extend you an offer" is **not** an offer.

If you don't have an offer, you don't have an offer.
An **offer** is a formal entity with very specific parts.

We are so afraid of being judged; that the moment we have a hint of a positive, we relax.

In a group of interviews, for example, I got the very **promise** of an offer, and the position got frozen due to strategy changes in the company, and I'm not extended the offer.

Erring on the wrong side is better than having false hopes.
Assume that what you've been told is an obfuscation designed to get you relaxed.

It's like your date saying "y*ou dressed nice*" on your first date;
and you assume everything is gonna be perfect.

Have I told you that the human brain is an interesting machine?
Under uncertainty, we are prone to believe in anything, to reduce our stress level.

Until you have an offer, you should be going after every offer you can get your hands on.

Until you are extended an offer never give up, and **never believe the nice things you are told**.

Don't Talk About Compensation and Benefits too Early

Most probably you know this already:
It's unprofessional and unethical to talk about your compensation and benefits before you're extended an offer.

> God says to me with a kind of smile,
> "Hey how would you like to be God awhile
> And steer the world?"
>
> "Okay," says I, "I'll give it a try.
>
> Where do I set?
> How much do I get?
> What time is lunch?
> When can I quit?"
>
> "Gimme back that wheel," says God.
> "I don't think you're quite ready yet."
>
> **Shel Silverstein**

If you don't have an offer, it makes no sense to talk about your salary and benefits.

You are not Obliged to Accept an Offer

This is an **extension** to the above discussion.

There are two parts to every job search. **Getting** offers, and then **taking** offers. Don't mix the two.

Feel no obligation to accept an offer.

And if you **do** accept an offer, then you've made a choice against the opportunity cost of accepting other offers.
Be an adult, be **thankful** and **grateful**.

The Most Important Two Interview Questions

In this section, I'll elaborate on the most important two questions.
If you prepare for them well enough, you can answer **"any"** behavioral question thrown at you.

If you have been to at least one interview, then you must have already heard about them. These questions are:

- ★ "What is your greatest achievement?"
- ★ "Tell me about yourself."

Fear not; we will cover them both in excruciating detail. We will see why these questions are important, what is the reason behind asking them, and how to successfully answer them.

"What Is Your Greatest Achievement?"

This question is the most frequently asked behavioral interviewing question.
It centers on the theory that the best way to predict future behavior is by projecting from past behavior.

This question will come to you in different forms, all asking for the same thing:

- ★ Give me an example of your accomplishments.
- ★ What is the greatest optimization you have done this far?
- ★ Tell me about something you did in your job that you're proud of.
- ★ If you wanted to tell a success story, what would that be?
- ★ What is the biggest challenge you've overcome?

Most think that the good old "tell me about yourself" question is the most important behavioral interview question.
And they are **wrong** again. The "*greatest achievement*" question is **the most important behavioral interview question** you'll ever get in the interview.

Be Prepared

Prepare for this question in advance, because **you will be asked several times**. Yes, you should be well prepared to deliver any bullet point the interviewer asks about your resumé. Yet, there are certain accomplishments that are better-suited for the job you are being interviewed for. On different occasions, a different set of accomplishments may shine more than the others.

Choose your accomplishments wisely, and prepare for them.

Avoid Sorting by Time at All Costs

You won't want to give a "*and then... blah blah... and then... blah blah... and then...*" answer. It will sound **awful**.

Contrarily, you should organize the core of your answer around facts; such that, you should be able to aggregate everything that happened in such a way that it all relates to the key drivers of your action.

Draw a **time-independent** picture and make the interviewer "**see**" what you are doing.

Providing **Meta-Information** *is* **not** *Repeating Yourself*

Let me tell you more about the importance of introducing meta-information about your story: It will enable the interviewer to think about the concepts beforehand; she will prepare a stage on her mind; if she has prior knowledge of the concepts, it would be much easier for her when you introduce the details of those concepts later.

You may say something like "So *let me tell you* how I cut the server load by 50%" then walk her through the story. This kind of **meta communication**, where you explain to her what you are going to say, and then walk her through a visual story is **very powerful**. It's like telling the core of the story up front as in "I did this; and here is the back stage of it".

Tell the interviewer where your story will end, so they can **follow** you as you **lead** them.
This way, they will always be thinking about the end result on the back of their minds.

Your starting sentence should describe what you have achieved in the end.

Sentence by sentence **paint a picture** of the initial situation, and **how** you have improved that situation to achieve success. Describe your **overall approach** and give a **high level overview** of **why** you did what you did, **how** you attacked the problem. If the circumstances were different, would you have approached the problem the same way? If not, tell the interviewer about **alternatives**, and what made you choose your current sequence of actions among those alternatives.

The interviewer must be able to close her eyes and **visualize** you doing what you are talking about.

Finish with a brief statement of **results**. Describe what the **success** achieved for **you**, your **team**, and your **company**.

"Tell Me About Yourself."

Almost all interviews begin with this question.

If you're quarterly maintaining your career history file, answering this question will be a cakewalk.

The question will have different flavors:

★ Tell me about yourself;

★ Walk me through your resumé;

★ Give me a quick rundown of what you did so far.

This often leads to the most common interviewing fault: **revealing too much**.

Although there is no "universally accepted" 'too much'; a very long answer to this question may reveal your lack of preparedness in certain areas, or, who knows, an inconsequential bit of information that conflicts with your resumé.

Of course you are **honest** and **truthful**, therefore you should not be afraid of revealing too much.

When the interviewer says "tell me about yourself," what he or she is really doing is checking to see if you're **articulate** and **professional** on a multifaceted question, what your self-image is, how you express that image, and (*this is important*) how you handle **interruptions**.

The way you answer is as important as the things you say.

Give them the facts about your professional life.

Don't ask questions – as offered in certain blogs – it's useless. The only place in an interview to generate a conversation is "*the introductory chit chat*" and "*your time to ask questions*" parts. The rest is a **one way street** where they ask and you either answer, or ask for "more information" to validate your answer. You don't lead the process; you follow.

Further, don't try to **design** an answer that **twists** your professional career history into something that matches the company's requirements. Under the stress level you are at, you will definitely fail at that. Trying to bend the facts is another name for "telling lies". Truth and honesty is something you cannot sacrifice for any reason, in **any** interview.

Even an innocent lie has the potential to bite you in the back.

Spend five to six minutes for this answer. Most career books say that a two-minute answer is enough; however, giving a two-minute answer will leave things vague, and the interviewer will ask further questions to pull words out of your mouth. Don't be so discreet. If you have nothing to hide, why are you afraid of getting caught unprepared in the first place?

The other side of the coin is; if your answer exceeds six minutes, the interviewer will think you are throwing the kitchen sink at her: She'll lose her attention, get bored, and stop listening. You won't want that to happen.

I've seen this literally in the eyes of an interviewer, after having told a fifteen minute **"brief"**(?) explanation of my career history; and I learned this the hard way.
I'm sharing with you so you don't have to.

Interruptions Are Inevitable

There will be **interruptions**. Be prepared for them. Most probably the interviewer will ask why you did a certain thing in a certain way, while it could have been done in a different way. When you're interrupted, just save where you've left in the back of your head, answer the interviewer's question, and continue from where you left.

Interruptions are in the nature of any conversation:
The interviewer is not trying to be rude or anything. He is just trying to gain a further understanding of you.

Be prepared for interruptions, and don't let interruptions destroy the flow of your speech.

If you nail this question, the interviewer will have a persistent impression that you are **prepared**, you **know** what to talk about, and you can convey your message **forcefully**, yet **effortlessly**.

If there is only one thing you do with an interruption, it is to **remember** where you left off.

When you get a "why?" question, there is a very simple way to answer:

The best way to answer a "why?" question is to describe what your goal was at the time, and then to show how your choice served that objective.

When Your Turn to Ask Questions Comes...

Without exception, at the very end of any interview you will be asked a "*Do you have any questions for us?*" question.
It's an **overlooked** opportunity. You know that it will come, so **be prepared for it**.

First, this question is asked to see how your mind works, your ability to take the initiative and carry the conversation a bit.

Is it all you can do to answer the questions, or can you lead the discussion?

This section of the interview cannot make you, but it can obviously **break** you if you do it poorly.

If you are absolutely certain that you **will** be asked this question; not getting prepared for it is nothing but your laziness. And lazy people have no place in the professional world. It is dangerous to go in the interview without preparing for your questions in advance.

If you think that they will naturally pop out of your mind, good luck with it! Because, when the time for your questions comes, you would have been talking at least for half an hour, you would be tired, and no matter how well-prepared you are, you're under stress.

If you don't get prepared, those questions won't come.

If you respond with "*No, I have all my questions answered*", then you are pulling your own rope.
Worse than that, is asking the **wrong questions**.

For example, "*Am I allowed to work from home? If so, how often?*" will communicate that one of your top priorities is **not coming** to the office as much as possible. How wise!

Don't ask questions about benefits, PTOs, promotions, or God forbid... salary.
Those questions are a highway to hell.
Ask about **your role in the company**, and how you can **contribute** to the company.

Don't ask about the company, or the industry. The company expects their best candidate to have researched these in advance; and you are asking these because... wait for it... you are not their best candidate. Don't fall into this trap.

Don't ask "yes" or "no" questions either.
Your questions should allow the interviewer to talk about them for at least a minute or two.

And if you had an excellent interview, with many good information exchanges; if you've touched on interesting topics about your prospective role in the company, the tasks, duties, and requirements that you might have in the former parts of the interview; then it's not only appropriate to ask questions about them, **it's inappropriate not to**. Expand upon the topics you've talked about before, if you can. It will show that you are not blindly answering questions and you are paying attention to what the interviewer says as well. **She'll remember that**.

Going back to our dating dinner table *metaphor* again: A great conversation is going with your first date. She says "*talked about you long enough; let's talk about me. What do you want to learn about me?*" and suddenly you turn out to be a dull person asking the questions you've memorized, while she expects more **insightful** questions based on your former conversation.

The flip side of the coin is, if asked correctly, the questions you are going to ask are **invaluable assets** to you: Not only will you get the information you wanted, but the interviewer will also think that you are a critical thinker, who can take initiative when necessary. Ideally you should craft your questions in such a way that the interviewer should talk for a few minutes, then allow you to respond what they say; turning the discussion into a **two way conversation**.

Here are a couple of questions to get you started:

★ What projects can I contribute to right away?

★ What will be a typical day of mine at the company?

★ How do you define success, organizationally? (*should better be asked to a higher level executive*)

Rest assured, "*Do you have any questions for us*" is a question that is always asked.
Make sure you have something planned to ask.

This is the only place in this book I suggest memorizing something: **Memorize** a handful of questions before going to the interview. Treat the interview like a **conversation**. This is the only part of the interview that's 100% **conversational**. Just, don't let what you memorized veil more "to the point" questions you can take out of the overall interview process.

Don't stress out; be calm.
Your job, your responsibilities, and your contribution are good things to ask a question about.

And the last thing is, when you are done with your questions, stop.
This is showing respect to the interviewer's time. There will be some awkward pause, and that's normal.

It's not "That" Difficult to Talk About Salary

"Salary" discussions are something that people takes too emotionally. **Don't**.

First of all, your employer knows that there's a **free job market**, and she should offer a salary on par with your market value; otherwise you may choose an alternative; and even if you choose them, if you are not satisfied with your compensation, your employer very well knows that you'd be looking for other jobs. No professional employer will risk offering you an amount that you won't be satisfied, knowing that it'll increase the possibility of losing you in the future. Therefore if you are great at what you do, and you equally express this fact to them, then you will get a great offer.

Furthermore, salary is only a piece of your total compensation. Whatever your salary expectation is, you should be open to discussions; and believe it or not, salary does not buy job satisfaction.There are several other factors in overall job satisfaction; money tends to be one item out of many and often it is not the most important.

Having said that, salary discussions in an interview is inevitable; and there's a right way to approach salary discussions:

Keep your emotions out of the equation; know your market value;
and once you are asked, state how much you want to make freely.

We can divide the salary discussion into two parts.

★ Questions about your current salary;

★ Questions about how much you want to make.

Keep in mind that these are two **distinct** questions. Your current salary has nothing to do with how much you want to make in your next job. If you know the market conditions, and you have **marketable** and **transferrable** skills, then you are free to ask for what you are worth.

Answering these questions are **the simplest** thing you'll do in the overall interview process.
Do your research and come up with a number. That's it.

Don't beat your brains out, thinking hard about the "salary" questions. Don't exaggerate it and turn it out to something that you worry about; and don't be emotional in salary discussions.

What is Your Current Salary?

You'll have this question most probably in the introductory interviews, after passing the phone screen.
But it can come to you at any time.

When your current salary is asked, **say it freely**.

Most people take salary questions way too emotional. There's no need to do so; and approaching that way is counterproductive. The salary question is just yet another question; and the rules are simple: Answer what you have been asked for, and tell the truth. One advice I can give is to never lie, never stumble, and to confidently tell your salary.

Your salary is a number; and your salary does not start with "**well...**"

Another thing you have to consider is to **never** mention your salary unless you've been asked for.
Doing otherwise sends a message that salary is a "key factor" to you. It's **not** a good idea.

If you are trying to convince the company that money is more important than the job you are being interviewed for, then kiss that job goodbye.

As a rule of thumb, your interviews with the company should be about what **you** can do for the company; not the other way around. Which means that the interview is **not** the time to ask about the severance package, vacation time, sick leave, or health plan; and since the company has not extended you an offer, yet, you have no compensation or benefits at all.
If nothing, it's simply a waste of time to discuss about something that you don't have.

The company asks your salary to determine how well you are doing at your present job. Don't take it too emotional. It's totally independent of your real market value. Who knows, maybe you are a better match to the company you are talking to than you current employer. While your former employer was unable to use all of your skill set, everything you know will be beneficial to the company you are talking to; and in a fair job market, it's reasonable to expect a better match to be rewarded with a better compensation. Your response to the salary question should be straightforward:

"My base salary is this, and my bonus is that; Next year I believe my cash compensation will be this."

You might mention **stock options**, if you receive them in lieu of cash compensation.
But no matter what you do, **don't give out your total compensation as your salary**, it will severely damage you.

The following topic discusses the difference between your **compensation** and your **salary**:

Your Compensation is **not** *Your Salary*

Yet another common mistake is confusing **compensation** with **salary**.
It's virtually impossible for the interviewer to guess your total compensation based on your current salary.
And that's a good thing.

Salary, and **compensation** are two different things. Here's generally how your total compensation breaks down:

```
"Total cash compensation" = "Salary" + "Bonus";
```

```
"Compensation" =
    "Total cash compensation"
        + "Stock options"
        + "Restricted stock"
        + "401K benefits"
        + "Health and medical benefits"
        + "PTO and other allowances"
        + "Other benefits";
```

Therefore, it's virtually impossible for someone to deduce your compensation starting from your salary and vice versa.

Salary is often discussed when you are extended an offer, or just before you are officially extended an offer.
Never mention salary unless you are asked about it.

> Focus on what you are going to give, not what you are going to get. Otherwise you will be perceived as a selfish guy whose only focus is money. Nobody likes to work with selfish people.

Before Anything, Do Your Market Research

Before even applying for the job, research **the median rate** for the job.

> Do not be emotional; manage your expectations based on the market conditions.

So, what if you don't know what the job will pay? Well, then you're clearly not prepared for an interview.

Here are some things you can do to **be** prepared:

Research Salaries on the Internet

This is so easy today as to be considered a **requirement** by most interviewers.

There are many sites that list salary ranges:
glassdoors.com, salary.com, salaryexpert.com, careerjournal.com... are just a few.

Ask the Recruiters In Your Network

Yes, it's **perfectly legitimate** to ask the recruiters you are in touch with for a salary range for a given position, in a given company; and if you don't know any recruiters, you are not managing your career effectively; you should.

The recruiter will most probably give you a **precise** answer. Any vague answer is **not** a good sign. Yes, the recruiter is paid by the company, not you; but the salary range is a reasonable question to ask for; and they'll answer you truthfully.

> If they ask for **your** compensation, than the recruiter has not done her homework.

Consult an Insider

Ask someone at the company you'll work at. Some companies even post salary ranges, so make sure you go to their websites too. Ask a good friend in the company about the salary ranges there.

Don't have any friends in the company? Get some.

Ask Someone at Your Company

Find someone in your company doing an equivalent job that you are going to apply for.

Don't ask what they make. It's rude, and ineffective, since they won't tell you the truth.

Instead, ask what the salary range is for their job. You'll be much better off.

Ask Your Friends

Ask your contacts who work in similar positions to learn more about the job role's requirements and the salary range.

...

Put together all the numbers you find and take the mean of it. Remember, when you are asked for how much money you want to make, you want to **give them a number**. If you give them a range, don't make it more than the **10%** of the mean value you've calculated.

Yes, you might be wrong. That's why the more sources you have, the better.

What's Your Salary Expectation?

After doing your research, it's easy to answer this question. Just don't try to be smart. **Be honest and candid**.

Don't play games, know your average range, and **state it**.
Also state that salary is only a part of your compensation, and **you are open to further discussing the opportunity and other benefits**.

Remember that you will not be obliged to stay with any number you state.
And if you have a realistic view of the market value, you will never be pricing yourselves out of the job.

Don't be emotional. Don't be embarrassed, don't appear unsure, don't act absurdly.
Just state a range.

Here's how this question should be replied:

"My research shows that the roles I'm moving to in this industry pay around ${your range}K."

"Of course, I'm flexible on that, depending on the opportunity and other benefits"

Contrary to popular belief, trying to get your potential employer to name a price first, is one of the top ten stupidest advice you might have. That's not negotiation; it'll make you seem dumb instead.

Remember, you are not in control of the interview.
And any cunningness you think you are doing will never change who's in charge of the interview.

An interviewer who thinks you don't know the range, will not tell you their range.

Consider the cost of living while specifying a salary range. For instance, living in San Francisco is almost twice as expensive as living in Texas. Your compensation should reflect that.

Don't Let the Salary Negotiation Turn You into a Werewolf

So your hard work paid off, and an offer is on the table. Thinking "*now the negotiation begins, and it's time to get more money.*" will completely change your personality. Even if you don't notice the change, the other party **will**.

When you negotiate for a car, the thing which you are negotiating doesn't change even if you become a jerk. However, in salary negotiations, it's **you** who's being negotiated.
If you change your behavior (*and you will*) it's possible that the company will lose faith in you.

Believe me or not, even if you are the toughest negotiator, your gain out of the salary "negotiation" will not exceed **$10K** a year (**without** *taxes and deductions*). The interviewer you are doing the salary negotiations with, generally cannot give you more than that without taking the offer back and discussing with higher management for further approval; and if the top management rejects the change, you will be in an awkward situation to either agree on something that's lower than you wanted, or decline the offer. Even if you get the job, your self-image is damaged.

Believe me, it's not worth taking the risk.

This may sound counterintuitive. My humble recommendation is **not** to negotiate.

If you still believe you are a good negotiator; your friends and family always say that you are a good negotiator; if you badly want to replace the goodwill that's put on the table with a few thousand bucks; and you know that you can **convince** the interviewer that you are worth every dollar that the job has to make... than good luck, go ahead and negotiate. Otherwise, if the salary offered is within your expectations, then there is no point in further negotiating.

If you think an additional $10K (*before deductions*) will make you the happiest person in the known universe, then I'm sorry and you are not adult enough for this job market.

Moreover, you are convincing the interviewer that you deserve the job; and you are **not** convincing her that you are worth every penny that the job has to make.

Remember, when an offer is extended to you, there's not only offer, but a considerable amount of **goodwill** on the table. The harsher you negotiate, the more you take from that goodwill off of the table; and it'll bite you later in the long run.

You **might** be able to get some additional compensation from the process; but so what? Or at a cost of what?

Do you think it will be worth it when everyone else around you gets a pay raise, but you don't because you've used your chances at salary negotiation? Worse, what if your manager's manager looks at your salary history and sees that everyone has gotten a raise but you.

And do you think you will be calm and professional when negotiating your salary?

First impressions count; and your manager will never forget the **brutal** and **selfish** image of you when asking for more.

Have some candor. If you have talked about your salary expectation before, and the offer given to you is within your preferred range, there's no point in being overly eager.

Remember? **Candor always prevails against cunningness**. Always.

Chapter 12
How to Handle Offers

That's the most delicate part of the overall process.

There are two parts in this: Getting an offer; and accepting or rejecting an offer.

You cannot accept an offer you don't have; and an offer is a "**very**" specific entity with **compensation**, **benefits**, **location**, and everything. As we've seen before, a promise of an offer, is not an offer.

In this section we will see what we should do while...

- ★ **Receiving** an offer;
- ★ **Accepting** an offer;
- ★ And **declining** an offer

respectively.

Receiving an Offer

An offer is a very specific thing. It has four components: **position**, **location**, **compensation**, and a **decision date**.
Anything that has one or more of those components missing, is **not** an offer.

The first thing you do when you receive an offer is to **thank** the offeror.
When you've been extended an offer the first thing you should do is to say "thank you".

Do not even receive an offer without being appropriately thankful.

Then make sure that all the key components of the offer (*position, location, compensation, decision date*) are available to you. If any of those components are not available, then **ask for it**. If the person cannot tell you everything, that's okay. Just find where you can get the missing details.

The company may be thinking that you will be accepting on the spot. **You have no obligation to do so**. Give yourself some time to think about it, and discuss it with your family. If the company is not professional enough to give you a couple of days to think about the offer, then it's not a company worth working for anyway.

It's a professional courtesy to inform within 24 hours anyone you've been interviewing that you have accepted an offer. It's not bragging or whining. It's just **adult**, **professional behavior**.

Accepting an Offer

If you are doing several interviews in parallel (*you should*) and you are good enough (*you should be*), you will be getting two or more offers within the same time frame; and you can accept only one offer. So **after** accepting one offer, you'll be rejecting others' offers.

When accepting an offer; be excited, appreciative, and **express these emotions**.
Even it's not the first company on your list, hey! you've accepted the offer; **show your appreciation**.

Not showing your appreciation after receiving an offer
is like saying "well... I guess so" to a marriage proposal.

If you have made your decision, there's no reason to wait. Accept immediately; and **send thank you notes** to those who have been involved in the process. It's unprofessional to know that you are going to accept, and then delay your response.

There's no shame in quick acceptance.
Waiting is a stressful situation for both parties.
Your prospective employer will appreciate your promptness.

Be Specific

Another common mistake is that you accept, and the other party does not understand because you use **vague** terms like...

"Bob, I will be an excellent contributor to your team."

Instead, **explicitly** use the words **"accept"** and **"offer"** in your response:

"Bob, I **accept** your **offer**. I'm truly excited that I will be starting to work with you guys."

Always Accept Before You Decline

Always accept one offer before declining others.

I know it sounds trivial, and you should be mad to do otherwise, but interviewing with multiple companies simultaneously is such a hassle, that this scenario can happen to the best of us.

What if you decline an offer first, and then the other party decides to freeze the job position (*happened to me*) then you will be without offers wondering what you did wrong.

If you are having dozens of interviews in parallel, things may get out of your control; and you may have forgotten that you haven't accepted an offer yet before declining others. Keep in mind that until you've accepted an offer, the job is not yours. **Anything can happen**.

Always **accept before you decline**. If you decline first and then find out that there is a change with your first offer, then you're left with nothing. This **does** happen.

Also it's possible that you get an offer from a company that's not your first choice, and even if you ask them for more time (*you should*), the offer will expire before your first choice offer comes in. That's keeping a rabbit in hand, versus waiting for one in the bush; and it's a complex situation to address here. It depends on the likelihood of the realization of the second offer (*are you sure you will get an offer, or is it just you being optimistic?*) and how happy you would be with the current offer at hand. I'd suggest you to **think very carefully** if you fall into a similar situation.

Declining an Offer

If you have accepted another offer, **follow up with a thank you note to the person you declined**. That's an important part that's mostly overlooked during the joy of the acceptance of your offer. Also send thank you notes to everyone who have played a crucial role in the process. You'll never know when you'll meet again.

Similar to accepting an offer, make sure you use the words **decline** and **offer**.

Only decline offers after you've accepted one; and say that you've accepted another offer to the company. This will keep you from having to turn down better and revised offers; and it will save the company time because they would not need to revise their offer, and utilize their time by doing something better. They will remember that, and appreciate that.

Empire "Counter-Strikes" Back

And when you say you have an offer to your current company; they may prepare a counter-offer. There are a billion reasons accepting that counter offer would be equivalent to committing suicide.

No matter how appealing the counter offer is, I strongly suggest you kindly reject it.
You will be emotionally, and strategically safer that way.

Chapter 13
The Postlude

That's all the important things you need to pay attention to during behavioral interviews. To wrap up, we've seen:

★ The importance of **honesty**, **candor** and **telling the truth**;

★ How to use every opportunity;

★ How to prepare an effective **cover letter** that will get your resumé read;

★ How to prepare an effective **resumé** that will get you an **interview**;

★ What you should pay attention to **during the interview**;

★ How to answer **the most important two interview questions**;

★ How to answer **questions about your salary**;

★ What **questions to ask** when the time comes;

★ **How to close** at the end of the interview;

★ **How to follow up** after the interview;

★ And how to **receive**, **accept** and **decline** offers.

As final remarks, I'll just add a few additional key points in the following page.

These are as important as the interview itself, so **read and apply them carefully**.

Take Notes After the Interview While Your Mind Is Fresh

Most of the questions come from two sources: what you said on your resumé, and your answers to questions in an **earlier interview**, and issues that the interviewer mentioned in an **earlier interview**.

This not only stresses the importance that you should not bend the fact in your resumé, and be truthful at all times; but it also means that **listening** is an important skill that you **must** have. **Listen** to the interview very **carefully**; and take notes **immediately after** the interview. **The sooner, the better**.

The interview you just had, **will** include clues about what will be asked in the next interview. So taking notes is **crucial**.

For instance, if the interviewer mentions "*loose coupling*" in the interview, and how they implement it in their recent project, your next interview may be a hands on coding session asking you to cram some "*loose coupling*" examples; or if the interviewer asks your opinion about a specific library or framework, you may be requested to create an application using that library in the following interview.

Don't Hesitate to Ask for Feedback

If after a telephone interview you don't get called to the next stage, do not be afraid to contact the interviewer again and **ask for feedback** on your **performance**. This will help you to improve your own skills and hopefully produce better outcome in the future interviews.

Always Be Thankful

It does not matter whether you accept or decline an offer, or whether you've just been through an interview.
Whenever you **can**, follow up with a **thank you note**, or pick up your phone, call their number and thank them.

Don't worry; you are not flattering anyone. You are **expected** to show your appreciation.
And if you do not, it will show that you are not professional enough to show your gratitude.

...

That's all about the behavioral part of the interview process. **Study** and **practice** them well, and you will see that you will become gradually better. And feel free to send your comments to me, I'd **love** to hear about your experiences.

Chapter 14
Conclusion

Dear Ahmed,

This was a long and enjoyable journey, wasn't it?

We've covered many things with a comprehensive set of interview questions, and touched many technical topics such as **algorithmic complexity**, **Big-O** analysis, **Closures**, **Modules**, and dissected an important subset of JavaScript Application **Design Patterns**. We have also mentioned the key technologies JavaScript interacts with, like **CSS**, **HTML**, **DOM**, and **Node.JS**.

What is more important is that we have learned the subtle art and the exact science of preparing for a **behavioral interview**, starting from how to prepare a killer resumé, to how to handle, accept, and reject offers.

If I was to summarize this approximately 50,000 word book with a single sentence, I would have written:

> **Always be Prepared!**

Because the most important thing is improving one's self forever:

> Suspicion, doubt, scientific inquiry, approaching things with a grain of salt, always questioning everything, and having an unfulfillable appetite to learn... are hard characteristic traits to grow; and once you have those traits, you will see that being prepared will feel so natural.

I hope you enjoyed this book as much as I enjoyed writing it.

Now, go ace that interview!

Respectfully,

V.Özçelik

Bibliography & References

- ".net Design Patterns" http://www.dofactory.com/Patterns/Patterns.aspx
- "A Beginner's Guide to Big-O Notation" http://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- "Advanced JavaScript: Objects, Arrays, and Array-like Objects" http://nfriedly.com/techblog/2009/06/advanced-javascript-objects-arrays-and-array-like-objects/
- Aeter, Suleman; "Why Big-O Needs and Update" http://www.futurechips.org/thoughts-for-researchers/big-o-notation-useless-today.html
- "A Fistful of Monads" http://learnyouahaskell.com/a-fistful-of-monads
- "Ajax Patterns" http://ajaxpatterns.org/Patterns
- Allen, David; "Understanding the Matrix of Self Management" http://www.cbsnews.com/8301-505125_162-51254243/understanding-the-matrix-of-self-management/
- "Allegory of Cave" http://en.wikipedia.org/wiki/Allegory_of_the_Cave
- Alman, Ben; "Immediately Invoked Function Expressions" http://benalman.com/news/2010/11/immediately-invoked-function-expression/
- Alsup, Mike; "A Plugin Development Pattern" http://www.learningjquery.com/2007/10/a-plugin-development-pattern
- "Analysis of Algorithms" http://en.wikipedia.org/wiki/Analysis_of_algorithms
- "Antipattern" http://en.wikipedia.org/wiki/Anti-pattern
- "AMD API" https://github.com/amdjs/amdjs-api/wiki/AMD
- "Amortized Analysis" http://en.wikipedia.org/wiki/Amortized_analysis
- "A Re-Introduction to JavaScript" https://developer.mozilla.org/en-US/docs/JavaScript/A_re-introduction_to_JavaScript

- ★ "Array Data Structure" http://en.wikipedia.org/wiki/Array_data_structure
- ★ "Bachmann, Paul" http://en.wikipedia.org/wiki/Paul_Gustav_Heinrich_Bachmann
- ★ Baden, Scott B.; "Latency Hiding" http://www.pdc.kth.se/education/historical/previous-years-summer-schools/2009/handouts/PDC_Su09.pdf
- ★ "Behavior-Based Interviews: How the Past can Predict the Future" http://www.trainingindustry.com/articles/behavior-based-interviews.aspx
- ★ "Best, Worst, and Average Case" http://en.wikipedia.org/wiki/Best,_worst_and_average_case
- ★ "Brainteaser Interview Questions" http://jobs.aol.com/articles/2010/09/22/brain-teaser-interview-questions/
- ★ "Big-O Notation" http://en.wikipedia.org/wiki/Big_O_notation
- ★ "Big-O Notation" http://xlinux.nist.gov/dads/HTML/bigOnotation.html
- ★ "Big-O, How Do you Calculate and Approximate it?" http://stackoverflow.com/questions/3255/big-o-how-do-you-calculate-approximate-it/4852666#4852666
- ★ "Blocking IO" http://blog.pebblecode.com/post/11607779340/blocking-and-non-blocking-i-o
- ★ "Body Language" http://en.wikipedia.org/wiki/Body_language
- ★ "Bubble Sort" http://en.wikipedia.org/wiki/Bubble_sort
- ★ Bushcman, Meunier, Rohnert, Stal "Pattern Oriented Software Architecture" http://www.amazon.com/Pattern-Oriented-Software-Architecture-Volume-Patterns/dp/0471958697
- ★ "By Value versus By Reference" http://docstore.mik.ua/orelly/webprog/jscript/ch11_02.htm
- ★ "Cache" http://en.wikipedia.org/wiki/Cache_(computing)
- ★ "Callback Hell" http://callbackhell.com/
- ★ "Call Stack" http://en.wikipedia.org/wiki/Call_stack
- ★ "Chicken or the Egg Problem" http://en.wikipedia.org/wiki/Chicken_or_the_egg
- ★ Chuan Shi, "JavaScript Patterns" http://shichuan.github.com/javascript-patterns/
- ★ "Circular Memory Leak Mitigation" http://msdn.microsoft.com/en-us/library/dd361842(v=vs.85).aspx

- ★ "Circular Reference" http://en.wikipedia.org/wiki/Circular_reference#In_computer_programming
- ★ "Closures for Dummies (or why IIFE !== closure)" http://unscriptable.com/2011/10/02/closures-for-dummies-or-why-iife-closure/
- ★ "CommonJS Modules" http://wiki.commonjs.org/wiki/Modules/1.1
- ★ "Complexity Theory" http://en.wikipedia.org/wiki/Computational_complexity_theory
- ★ "CommonJS" http://www.commonjs.org/
- ★ "Concurrent Computing" http://en.wikipedia.org/wiki/Concurrent_computing
- ★ "Continuations" http://en.wikipedia.org/wiki/Continuations
- ★ "Continuation Passing Style" http://en.wikipedia.org/wiki/Continuation-passing_style
- ★ Cornford, Richard; "Browser Detection (and What to Do Instead)" http://jibbering.com/faq/notes/detect-browser/
- ★ "Comma Operator" https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator
- ★ "CommonJS" http://en.wikipedia.org/wiki/CommonJS
- ★ Crockford, Douglas; "JavaScript the Good Parts" http://shop.oreilly.com/product/9780596517748.do
- ★ Crockford, Douglas; "Private Members in JavaScript" http://javascript.crockford.com/private.html
- ★ Croll, Angus; "Function Declaration vs Function Expressions" http://javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions/
- ★ Croll, Angus; "Understanding JavaScript Closures" http://javascriptweblog.wordpress.com/2010/10/25/understanding-javascript-closures/
- ★ "Curl" https://github.com/unscriptable/curl
- ★ Darell, Tore; "Scope in JavaScript" http://tore.darell.no/pages/scope_in_javascript
- ★ "Decision Tree" http://en.wikipedia.org/wiki/Decision_tree
- ★ "Design by Contract" http://en.wikipedia.org/wiki/Design_by_contract

- "Design Patterns (Computer Science)" http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29
- "Design Patterns by Gang of Four" http://www.amazon.com/Design-Patterns-Elements-Object-Oriented-ebook/dp/B000SEIBB8
- "Design Patterns" http://sourcemaking.com/design_patterns
- Diaz, Dustin; "JavaScript Curry" http://www.dustindiaz.com/javascript-curry/
- "DISC Assesment" http://en.wikipedia.org/wiki/DISC_assessment
- "Distributed Computing" http://en.wikipedia.org/wiki/Distributed_computing
- "DOMContentLoaded" https://developer.mozilla.org/en-US/docs/DOM/DOM_event_reference/DOMContentLoaded
- "DOM Level 0 Events" http://en.wikipedia.org/wiki/DOM_events#DOM_Level_0
- "DOM Level 2 Specification" http://www.w3.org/TR/DOM-Level-2-Core/
- Denicola, Domenic, "You are Missing the Point of Promises" http://domenic.me/2012/10/14/youre-missing-the-point-of-promises/
- Doyle, Alison; "Behavioral Interviews" http://jobsearch.about.com/cs/interviews/a/behavioral.htm
- Ebert, Ralf; "Git in Action" http://www.ralfebert.de/blog/tools/git_screencast/
- "EcmaScript Programming Language" http://www.ecmascript.org/
- "EcmaScript Language Specification" http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf
- "EcmaScript 6 Private Names" http://wiki.ecmascript.org/doku.php?id=strawman:private_names
- "Elevator Pitch" http://en.wikipedia.org/wiki/Elevator_pitch
- "Facial Expressions" http://en.wikipedia.org/wiki/Facial_expression
- "Factory Method Pattern" http://en.wikipedia.org/wiki/Factory_method_pattern
- "Façade Pattern" http://en.wikipedia.org/wiki/Facade_pattern
- "First Class Functions" http://en.wikipedia.org/wiki/First-class_function

- "First Steps in Removing Promises" https://groups.google.com/forum/?fromgroups=#!msg/nodejs/RvNoQtoWyZA/a8Hu83Ewb0IJ
- "Fluent Interface" http://en.wikipedia.org/wiki/Fluent_interface
- Fowler, Martin; "Passive View" http://martinfowler.com/eaaDev/PassiveScreen.html
- Fowler, Martin; "GUI Architectures" http://martinfowler.com/eaaDev/uiArchs.html
- Fowler, Martin; et. al.; "Refactoring, Improving the Design of Existing Code" http://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672
- "Function Object" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function
- "Functions and Function Scope" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope
- "Function / bind" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function/bind
- "Functional Programming" http://en.wikipedia.org/wiki/Functional_programming
- "Functions and Function Scope" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope
- "Github: a web-based hosting service for projects that use the Git revision control system" http://github.com/
- "Git Immersion" http://gitimmersion.com/
- "Git Book" http://git-scm.com/book
- "Git Magic" http://www-cs-students.stanford.edu/~blynn/gitmagic/index.html
- "Global Variables are Bad" http://c2.com/cgi/wiki?GlobalVariablesAreBad
- Godin, Seth; "Authenticity" http://sethgodin.typepad.com/seths_blog/2009/02/authenticity.html
- Greer, Derek; "JavaScript Closures Explained" http://lostechies.com/derekgreer/2012/02/17/javascript-closures-explained/
- Griggs, vs. Duke Power Co., 401 US 424 - Supreme Court 1971; http://scholar.google.com/scholar_case?case=8655598674229196978&q=Griggs+Duke+Power&hl=en&as_sdt=2,24

- ★ Gutman, Ron; "The Hidden Power of Smiling" http://www.ted.com/talks/ron_gutman_the_hidden_power_of_smiling.html
- ★ "Harmony Modules" http://wiki.ecmascript.org/doku.php?id=harmony:modules
- ★ "Hash Table" http://en.wikipedia.org/wiki/Hash_table
- ★ Haverbeke, Marjin; "Eloquent JavaScript" http://eloquentjavascript.net/
- ★ Haverbeke, Marjin; "Eloquent JavaScript, Functional Programming" http://eloquentjavascript.net/chapter6.html
- ★ Heilmann, Christian; "Show Love to the Module Pattern" http://christianheilmann.com/2007/07/24/show-love-to-the-module-pattern/
- ★ "High Dominance Only" http://www.discusonline.com/udisc/highdo.html
- ★ Houghton, Alastair; "Optimization without Measurement - A Seductive Trap" http://alastairs-place.net/blog/2007/08/12/seductive-trap/
- ★ "Hollywood Principle" http://en.wikipedia.org/wiki/Hollywood_principle
- ★ "How to Ace a Google Interview" http://online.wsj.com/article/SB10001424052970204552304577112522982505222.html
- ★ "HTML5 Rocks" http://www.html5rocks.com/en/
- ★ "HTML5 Hub" http://html5hub.com/
- ★ "Human Behavior" http://en.wikipedia.org/wiki/Human_behavior
- ★ Irish, Paul; "Developers We Admire" http://www.paulirish.com/2012/developers-we-admire/
- ★ "Iterator" http://en.wikipedia.org/wiki/Iterator
- ★ "JavaScript Anonymous Functions" http://helephant.com/2008/08/23/javascript-anonymous-functions/
- ★ "JavaScript Coercion Demystified" http://webreflection.blogspot.com/2010/10/javascript-coercion-demystified.html
- ★ "JavaScript Statements" https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements

- "JavaScript Timers with setTimeout and setInterval" http://www.elated.com/articles/javascript-timers-with-settimeout-and-setinterval/
- Janmyr, Anders; "A Not Very Short Introduction to Node.JS" http://anders.janmyr.com/2011/05/not-very-short-introduction-to-nodejs.html
- "Jedis Feel Doin It" https://gist.github.com/2694911
- "jQuery: the write less, do more library" http://jquery.com/
- Koch, Peter, Paul; "The this keyword" http://www.quirksmode.org/js/this.html
- Konicek, Martin; "Non-blocking IO Demystified" http://coding-time.blogspot.com/2012/06/non-blocking-io-demystified.html
- "KISS Principle" http://en.wikipedia.org/wiki/KISS_principle
- "Lambda Calculus" http://en.wikipedia.org/wiki/Lambda_calculus
- "Lazy Loading" http://en.wikipedia.org/wiki/Lazy_loading
- Ley, Jim; "JavaScript Closures" http://jibbering.com/faq/notes/closures/
- Lippert, Eric; "How Do the Script Garbage Collectors Work" http://blogs.msdn.com/b/ericlippert/archive/2003/09/17/53038.aspx
- "List of Complexity Classes" http://en.wikipedia.org/wiki/List_of_complexity_classes
- "Linked List" http://en.wikipedia.org/wiki/Linked_list
- "Long-Term Memory" http://en.wikipedia.org/wiki/Long-term_memory
- "Lookup Table" http://en.wikipedia.org/wiki/Lookup_table
- Lotado, Mark; "A Visual Git Reference" http://marklodato.github.com/visual-git-guide/index-en.html
- "Maximally Minimal Classes" http://wiki.ecmascript.org/doku.php?id=strawman:maximally_minimal_classes
- McCoy, Nicholas Firth; "Backbone.js in Practice Part 1 – Preventing Memory Leaks" https://paydirtapp.com/blog/backbone-in-practice-memory-management-and-event-bindings/

- ★ McDowell, Gayle Laakmann; "Debunking the Google Interview Myth" http://www.technologywoman.com/2010/05/17/debunking-the-google-interview-myth/
- ★ "Memory Management" https://developer.mozilla.org/en-US/docs/JavaScript/Memory_Management
- ★ "Memory Management and Recycling" http://www.memorymanagement.org/articles/recycle.html
- ★ "Merge Sort" http://en.wikipedia.org/wiki/Merge_sort
- ★ Michaux, Peter; "Hyper-Private Variables in JavaScript" http://michaux.ca/articles/hyper-private-variables-in-javascript
- ★ Michaux, Peter; "Module Pattern Provides no Privacy at All" http://peter.michaux.ca/articles/module-pattern-provides-no-privacy-at-least-not-in-javascript-tm
- ★ "Model-Driven Architecture" http://en.wikipedia.org/wiki/Model-driven_architecture
- ★ "Model-View-Adapter" http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93adapter
- ★ "Model-View-Controller" http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
- ★ "Model-View-Presenter" http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter
- ★ "Monads (Functional Programming)" http://en.wikipedia.org/wiki/Monad_(functional_programming)
- ★ "Monads are Burritos?" http://chrisdone.com/posts/monads-are-burritos
- ★ "Moore's Law" http://en.wikipedia.org/wiki/Moore's_law
- ★ "Mozilla Developer Network" https://developer.mozilla.org/en-US/
- ★ Meyer, Ulrich; Sanders, Peter; Sibeyn, Jop; "Algorithms for Memory Hierarchies" http://www.amazon.com/Algorithms-Memory-Hierarchies-Advanced-Lectures/dp/3540008837
- ★ "MVC vs MVP vs MVVM" http://nirajrules.wordpress.com/2009/07/18/mvc-vs-mvp-vs-mvvm/
- ★ Nadel, Ben; "A Graphical Explanation of JavaScript Closures in a jQuery Context" http://www.bennadel.com/blog/1482-A-Graphical-Explanation-Of-Javascript-Closures-In-A-jQuery-Context.htm
- ★ "Named Function Expressions" http://kangax.github.com/nfe/
- ★ "NaN" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/NaN

- "new Operator" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/new
- "Node.JS" http://nodejs.org/
- "Node.JS - A Giant Step Backwards?" https://news.ycombinator.com/item?id=3510758
- "Node.JS API" http://nodejs.org/api/
- "Node.JS Changelog" http://nodejs.org/changelog.html
- "Node.JS Modules" http://nodejs.org/docs/latest/api/modules.html
- "Node.JS Modules" https://github.com/joyent/node/wiki/modules
- "Object" https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object
- "Objects and Arrays in JavaScript" http://msdn.microsoft.com/en-us/library/89t1khd2(v=vs.94).aspx
- "Object.create" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/create
- "Object.freeze" https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze
- "Object Oriented JavaScript" https://developer.mozilla.org/en-US/docs/JavaScript/Introduction_to_Object-Oriented_JavaScript
- "Occam's Razor" http://en.wikipedia.org/wiki/Occam's_razor
- "o2.js JavaScript Framework" https://github.com/volkan/o2.js/blob/master/o2.js
- "o2.js DOM Readiness Check" https://github.com/volkan/o2.js/blob/master/o2.js/o2.dom.ready.js
- "o2.js o2.method.repeat Module" https://github.com/volkan/o2.js/blob/master/o2.js/o2.method.repeat.js
- "o2.js o2.unit.core Module" https://github.com/volkan/o2.js/blob/master/o2.js/o2.unit.core.js
- "Opportunity Cost" http://en.wikipedia.org/wiki/Opportunity_cost
- Osmani, Addy; "Essential JavaScript Design Patterns" http://addyosmani.com/resources/essentialjsdesignpatterns/book/
- Özçelik, Volkan; "JavaScript Function Kung-Fu" http://o2js.com/2011/05/03/javascript-function-kung-fu/

- ★ Özçelik, Volkan; "JavaScript Module Pattern" http://02js.com/2011/04/24/the-module-pattern/
- ★ Özçelik, Volkan; "JavaScript Widget Development Best Practices" http://02js.com/2012/07/05/javascript-widgets-overview/
- ★ Özçelik, Volkan; "Memory Leakage in Internet Explorer - Revisited" http://www.codeproject.com/Articles/12231/Memory-Leakage-in-Internet-Explorer-revisited
- ★ Özçelik, Volkan; "A Little Known Encapsulation Technique in JavaScript: Isolated Objects" http://02js.com/2011/10/16/javascript-isolated-objects/
- ★ Özçelik, Volkan: "Avoid Yoda Conditions" https://github.com/volkan/02.js/blob/master/CONVENTIONS.md#avoid-yoda-conditions
- ★ Özçelik, Volkan: "02.js JavaScript Conventions and Best Practices" https://github.com/volkan/02.js/blob/master/CONVENTIONS.md
- ★ "Parallel Computing" http://en.wikipedia.org/wiki/Parallel_computing
- ★ "Pipeline (Unix)" http://en.wikipedia.org/wiki/Pipeline_(Unix)
- ★ "Polymorphism" http://en.wikipedia.org/wiki/Polymorphism_(computer_science)
- ★ "Posture" http://en.wikipedia.org/wiki/Posture_(psychology)
- ★ "Promises/A+ Spec" http://promises-aplus.github.io/promises-spec/
- ★ "Promises/A+ Tests" https://npmjs.org/package/promises-aplus-tests
- ★ "Proper Tail Calls" http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls
- ★ "Prototype=Based Programming" http://en.wikipedia.org/wiki/Prototype-based_programming
- ★ "Psychology" http://en.wikipedia.org/wiki/Psychology
- ★ "Q" https://github.com/kriskowal/q
- ★ "Refactorings in Alphabetic Order" http://www.refactoring.com/catalog/
- ★ "Reflection" http://en.wikipedia.org/wiki/Reflection_(computer_programming)
- ★ "require.js" http://requirejs.org/

- "require.js API" http://requirejs.org/docs/api.html
- "Requirements Analysis" http://en.wikipedia.org/wiki/Requirements_analysis
- Resig, John; "Accuracy of JavaScript Time" http://ejohn.org/blog/accuracy-of-javascript-time/
- Resig, John; "DOM Document Fragments" http://ejohn.org/blog/dom-documentfragments/
- Resig, John; "How JavaScript Timers Work" http://ejohn.org/blog/how-javascript-timers-work/
- "Reserved Words / JavaScript" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Reserved_Words
- Rsywck, Jan, Van; "Taking Baby Steps With Node.JS – CommonJS and Creating Custom Modules" http://elegantcode.com/2011/02/04/taking-baby-steps-with-node-js-commonjs-and-creating-custom-modules/
- Sandmann, Søren; "Big-O Misconceptions" http://ssp.impulsetrain.com/2012-10-15_Big-O_Misconceptions.html
- "Scope" http://en.wikipedia.org/wiki/Scope_(computer_science)
- "Scientific Skepticism" http://en.wikipedia.org/wiki/Scientific_skepticism
- Schmidt , Frank L.; Hunter, John E.; "The Validity and Utility of Selection Methods in Personnel Psychology: Practical and Theoretical Implications of 85 Years of Research Findings" http://mavweb.mnsu.edu/howard/Schmidt%20and%20Hunter%201998%20Validity%20and%20Utility%20Psychological%20Bulletin.pdf
- "Set (Abstract Datatype)" http://en.wikipedia.org/wiki/Set_(abstract_data_type)
- Sexton, Alex; "How, When, and Why Script Loaders are Appropriate" http://msdn.microsoft.com/en-us/magazine/hh227261.aspx
- "Serif" http://en.wikipedia.org/wiki/Serif
- Smithson, Michael; "Psychology's Ambivalent View of Uncertainty" http://psychology3.anu.edu.au/people/smithson/details/Pubs/Ch18_Smithson.pdf
- "Social Learning Theory" http://en.wikipedia.org/wiki/Social_learning_theory
- "Software Design Pattern" http://en.wikipedia.org/wiki/Software_design_pattern
- "Software Project Management" http://en.wikipedia.org/wiki/Software_project_management

- "Someday/Maybe = Unlikely/Never 3 Tips to Fix and Avoid This" http://gtd.marvelz.com/blog/2007/08/14/somedaymaybe-unlikelynever-3-tips-to-fix-and-avoid-this/
- Soshnikov Dmitry; "ECMA-262-3 in detail. Chapter 5. Functions" http://dmitrysoshnikov.com/ecmascript/chapter-5-functions/
- Stefanov, Stoyan; "Rendering: repaint, reflow/relayout, restyle" http://calendar.perfplanet.com/2009/rendering-repaint-reflow-relayout-restyle/
- Stefanov, Stoyan; "JavaScript Patterns" http://shop.oreilly.com/product/9780596806767.do
- Steele, Oliver; "Web MVC" http://osteele.com/archives/2004/08/web-mvc
- "Strict Mode" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/Strict_mode
- Souders, Steve; "Loading Scripts Without Blocking" http://www.stevesouders.com/blog/2009/04/27/loading-scripts-without-blocking/
- "Space/Time Tradeoff" http://en.wikipedia.org/wiki/Space%E2%80%93time_tradeoff
- "Substitute Algorithm" http://www.refactoring.com/catalog/substituteAlgorithm.html
- "SuperHero.JS" http://superherojs.com/
- "SWOT Analysis" http://en.wikipedia.org/wiki/SWOT_analysis
- "Talks to Help You Become a Better Front-End Engineer" http://www.smashingmagazine.com/2012/12/22/talks-to-help-you-become-a-better-front-end-engineer-in-2013/
- "Tail Recursion" http://c2.com/cgi/wiki?TailRecursion
- "The Elements of a Design Pattern" http://www.uie.com/articles/elements_of_a_design_pattern/
- "Time Complexity" http://en.wikipedia.org/wiki/Time_complexity
- "Time Complexity, Space Complexity, and the O-notation" http://www.leda-tutorial.org/en/official/ch02s02s03.html
- "Timers, Timer Resolution, and Development of Efficient Code" http://msdn.microsoft.com/en-us/library/windows/hardware/gg463266.aspx

- "TraceMonkey, Mozilla:JavaScript" https://wiki.mozilla.org/JavaScript:TraceMonkey
- "Traveling Salesman Problem" http://en.wikipedia.org/wiki/Travelling_salesman_problem
- "Undefined" https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/undefined
- "Understanding and Solving Internet Explorer Leak Patterns" http://msdn.microsoft.com/en-us/library/bb250448(v=vs.85).aspx
- "Understanding Events" http://www.quirksmode.org/js/introevents.html
- "Understanding the Node.JS Event Loop" http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/
- "Using Pull Requests" https://help.github.com/articles/using-pull-requests
- "UNIX Philosophy" http://en.wikipedia.org/wiki/Unix_philosophy
- "W3Fools – A W3Sceools Interventions" http://www.w3fools.com/
- Wells, Martin; "High Performance Garbage-Collector Friendly Code" http://buildnewgames.com/garbage-collector-friendly-code/
- "What is the Point of Promises" http://www.kendoui.com/blogs/teamblog/posts/13-03-28/what-is-the-point-of-promises.aspx
- "Working With Objects" https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Working_with_Objects
- "What are Design Patterns and Why Do I Need Them?" http://www.developer.com/design/article.php/1474561/What-Are-Design-Patterns-and-Do-I-Need-Them.htm
- "Why eval is evil" http://xkr.us/js/eval
- "V8 JavaScript Engine" http://code.google.com/p/v8/
- York, Dan; "Node.JS, Doctor's Office and Fast-Foot Restaurants – Understanding Event-Driven Programming" http://code.danyork.com/2011/01/25/node-js-doctors-offices-and-fast-food-restaurants-understanding-event-driven-programming/
- "YUI onavailable" http://developer.yahoo.com/yui/event/#onavailable

★ Zakas, Nicholas; "What is a non-blocking script" http://www.nczonline.net/blog/2010/08/10/what-is-a-non-blocking-script/

★ Zaytsev, Juri; "Use Cases for JavaScript Closures" http://msdn.microsoft.com/en-us/magazine/ff696765.aspx

★ "Zen" http://en.wikipedia.org/wiki/Zen