10 Interview Questions Every JavaScript Developer Should Know

AKA: The Keys to JavaScript Mastery

Bruce Lee on the Avenue — Leevin3 (CC BY-NC 2.0)

At most companies, management must trust the developers to give technical interviews in order to assess candidate skills. If you do well as a candidate, you'll eventually need to interview. Here's how.

. . .

It Starts With People

In "How to Build a High Velocity Development Team", I made a couple points worth repeating:

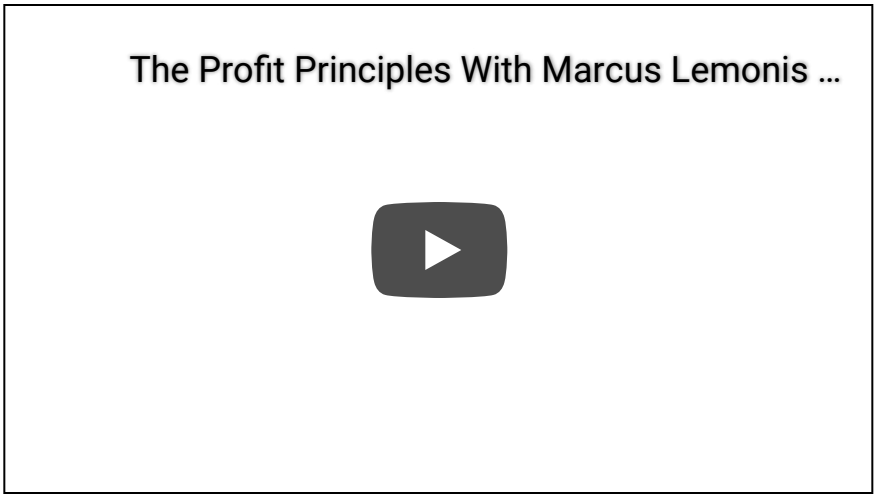
> *"Nothing predicts business outcomes better than an exceptional team. If you're going to beat the odds, you need to invest here, first."*

As Marcus Lemonis says, focus on the 3 P's:

. . .

"People, Process, Product"

. . .



Your early hires should be very strong, senior-level candidates. People who can hire and mentor other developers, and help the mid-level and junior developers you'll eventually want to hire down the road.

Read "Why Hiring is So Hard in Tech" for a good breakdown of the general do's and don'ts of candidate evaluation.

. . .

The best way to evaluate a candidate is a pair programming exercise.

. . .

Pair program with the candidate. Let the candidate drive. Watch and listen more than you talk. A good project might be to pull tweets from the Twitter API and display them on a timeline.

That said, no single exercise will tell you everything you need to know. An interview can be a very useful tool as well, but don't waste time asking about syntax or language quirks. You need to see the big picture. Ask about architecture and paradigms—the big decisions that can have a major impact on the whole project.

Syntax and features are easy to Google. It's much harder to Google for software engineering wisdom or the common paradigms and idioms JavaScript developers pick up with experience.

JavaScript is special, and it plays a critical role in almost every large application. What is it about JavaScript that makes it meaningfully different from other languages?

Here are some questions that will help you explore the stuff that really matters:

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

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

Good to hear:

- Prototypal inheritance (also: prototypes, OLOO).
- Functional programming (also: closures, first class functions, lambdas).

Red flags:

- No clue what a paradigm is, no mention of prototypal oo or functional programming.

Learn More:

- The Two Pillars of JavaScript Part 1—Prototypal OO.
- The Two Pillars of JavaScript Part 2—Functional Programming.

2. What is functional programming?

Functional programming produces programs by composing mathematical functions and avoids shared state & mutable data. Lisp (specified in 1958) was among the first languages to support functional programming, and was heavily inspired by lambda calculus. Lisp and many Lisp family languages are still in common use today.

Functional programming is an essential concept in JavaScript (one of the two pillars of JavaScript). Several common functional utilities were added to JavaScript in ES5.

Good to hear:

- Pure functions / function purity.
- Avoid side-effects.
- Simple function composition.
- Examples of functional languages: Lisp, ML, Haskell, Erlang, Clojure, Elm, F Sharp, OCaml, etc…
- Mention of features that support FP: first-class functions, higher order functions, functions as arguments/values.

Red flags:

- No mention of pure functions / avoiding side-effects.
- Unable to provide examples of functional programming languages.
- Unable to identify the features of JavaScript that enable FP.

Learn More:

- The Two Pillars of JavaScript Part 2.
- The Dao of Immutability.
- Composing Software.
- The Haskell School of Music.

3. What is the difference between classical inheritance and prototypal inheritance?

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

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

. . .

> *In JavaScript, prototypal inheritance is simpler & more flexible than class inheritance.*

. . .

Good to hear:

- Classes: create tight coupling or hierarchies/taxonomies.
- Prototypes: mentions of concatenative inheritance, prototype delegation, functional inheritance, object composition.

Red Flags:

- No preference for prototypal inheritance & composition over class inheritance.

Learn More:

- The Two Pillars of JavaScript Part 1—Prototypal OO.
- Common Misconceptions About Inheritance in JavaScript.

Classical Inheritance is Obsolete - How to Think in Prototypal OO

from **Eric Elliott**

43:18

4. What are the pros and cons of functional programming vs object-oriented programming?

OOP Pros: It's easy to understand the basic concept of objects and easy to interpret the meaning of method calls. OOP tends to use an imperative style rather than a declarative style, which reads like a straight-forward set of instructions for the computer to follow.

OOP Cons: OOP Typically depends on shared state. Objects and behaviors are typically tacked together on the same entity, which may be accessed at random by any number of functions with non-deterministic order, which may lead to undesirable behavior such as race conditions.

FP Pros: Using the functional paradigm, programmers avoid any shared state or side-effects, which eliminates bugs caused by multiple functions competing for the same resources. With features such as the availability of point-free style (aka tacit programming), functions tend to be radically simplified and easily recomposed for more generally reusable code compared to OOP.

FP also tends to favor declarative and denotational styles, which do not spell out step-by-step instructions for operations, but instead concentrate on **what** to do, letting the underlying functions take care of the **how**. This leaves tremendous latitude for refactoring and performance optimization, even allowing you to replace entire algorithms with more efficient ones with very little code change. (e.g., memoize, or use lazy evaluation in place of eager evaluation.)

Computation that makes use of pure functions is also easy to scale across multiple processors, or across distributed computing clusters without fear of threading resource conflicts, race conditions, etc…

FP Cons: Over exploitation of FP features such as point-free style and large compositions can potentially reduce readability because the resulting code is often more abstractly specified, more terse, and less concrete.

More people are familiar with *OO* and imperative programming than functional programming, so even common idioms in functional programming can be confusing to new team members.

FP has a much steeper learning curve than OOP because the broad popularity of OOP has allowed the language and learning materials of OOP to become more conversational, whereas the language of FP tends to be much more academic and formal. FP concepts are frequently written about using idioms and notations from lambda calculus, algebras, and category theory, all of which requires a prior knowledge foundation in those domains to be understood.

Good to hear:

- Mentions of trouble with shared state, different things competing for the same resources, etc…
- Awareness of FP's capability to radically simplify many applications.
- Awareness of the differences in learning curves.
- Articulation of side-effects and how they impact program maintainability.
- Awareness that a highly functional codebase can have a steep learning curve.
- Awareness that a highly OOP codebase can be extremely resistant to change and very brittle compared to an equivalent FP codebase.
- Awareness that immutability gives rise to an extremely accessible and malleable program state history, allowing for the easy addition of features like infinite undo/redo, rewind/replay, time-travel debugging, and so on. Immutability can be achieved in

either paradigm, but a proliferation of shared stateful objects complicates the implementation in OOP.

Red flags:

- Unable to list disadvantages of one style or another—Anybody experienced with either style should have bumped up against some of the limitations.

Learn More:

- The Two Pillars of JavaScript Part 1—Prototypal OO.
- The Two Pillars of JavaScript Part 2—Functional Programming.

5. When is classical inheritance an appropriate choice?

The answer is never, or almost never. Certainly never more than one level. Multi-level class hierarchies are an anti-pattern. I've been issuing this challenge for years, and the only answers I've ever heard fall into one of several common misconceptions. More frequently, the challenge is met with silence.

> *"If a feature is sometimes useful*
> *and sometimes dangerous*
> *and if there is a better option*
> *then **always use the better option**."*
> *~ Douglas Crockford*

Good to hear:

- Rarely, almost never, or never.
- A single level is sometimes OK, from a framework base-class such as `React.Component`.
- "Favor object composition over class inheritance."

Learn More:

- The Two Pillars of JavaScript Part 1—Prototypal OO.
- JS Objects—Inherited a Mess.

6. When is prototypal inheritance an appropriate choice?

There is more than one type of prototypal inheritance:

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

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

Good to hear:

- In situations where modules or functional programming don't provide an obvious solution.
- When you need to compose objects from multiple sources.
- Any time you need inheritance.

Red flags:

- No knowledge of when to use prototypes.
- No awareness of mixins or *`Object.assign()`*.

Learn More:

- "Programming JavaScript Applications": Prototypes section.

7. What does "favor object composition over class inheritance" mean?

This is a quote from "Design Patterns: Elements of Reusable Object-Oriented Software". It means that code reuse should be achieved by assembling smaller units of functionality into new objects instead of inheriting from classes and creating object taxonomies.

In other words, use **can-do, has-a,** or **uses-a** relationships instead of **is-a** relationships.

Good to hear:

- Avoid class hierarchies.
- Avoid brittle base class problem.
- Avoid tight coupling.
- Avoid rigid taxonomy (forced is-a relationships that are eventually wrong for new use cases).
- Avoid the gorilla banana problem ("what you wanted was a banana, what you got was a gorilla holding the banana, and the entire jungle").
- Make code more flexible.

Red Flags:

- Fail to mention any of the problems above.
- Fail to articulate the difference between composition and class inheritance, or the advantages of composition.

Learn More:

the Stamp Specification

Move Over, `class`:
Composable Factory Functions Are Here

medium.com

8. What are two-way data binding and one-way data flow, and how are they different?

Two way data binding means that UI fields are bound to model data dynamically such that when a UI field changes, the model data changes with it and vice-versa.

One way data flow means that the model is the single source of truth. Changes in the UI trigger messages that signal user intent to the model (or "store" in React). Only the model has the access to change the app's state. The effect is that data always flows in a single direction, which makes it easier to understand.

One way data flows are deterministic, whereas two-way binding can cause side-effects which are harder to follow and understand.

Good to hear:

- React is the new canonical example of one-way data flow, so mentions of React are a good signal. Cycle.js is another popular implementation of uni-directional data flow.
- Angular is a popular framework which uses two-way binding.

Red flags:

- No understanding of what either one means. Unable to articulate the difference.

Learn more:

9. What are the pros and cons of monolithic vs microservice architectures?

A monolithic architecture means that your app is written as one cohesive unit of code whose components are designed to work together, sharing the same memory space and resources.

A microservice architecture means that your app is made up of lots of smaller, independent applications capable of running in their own memory space and scaling independently from each other across potentially many separate machines.

Monolithic Pros: The major advantage of the monolithic architecture is that most apps typically have a large number of cross-cutting concerns, such as logging, rate limiting, and security features such audit trails and DOS protection.

When everything is running through the same app, it's easy to hook up components to those cross-cutting concerns.

There can also be performance advantages, since shared-memory access is faster than inter-process communication (IPC).

Monolithic cons: Monolithic app services tend to get tightly coupled and entangled as the application evolves, making it difficult to isolate services for purposes such as independent scaling or code maintainability.

Monolithic architectures are also much harder to understand, because there may be dependencies, side-effects, and magic which are not

obvious when you're looking at a particular service or controller.

Microservice pros: Microservice architectures are typically better organized, since each microservice has a very specific job, and is not concerned with the jobs of other components. Decoupled services are also easier to recompose and reconfigure to serve the purposes of different apps (for example, serving both the web clients and public API).

They can also have performance advantages depending on how they're organized because it's possible to isolate hot services and scale them independent of the rest of the app.

Microservice cons: As you're building a new microservice architecture, you're likely to discover lots of cross-cutting concerns that you did not anticipate at design time. A monolithic app could establish shared magic helpers or middleware to handle such cross-cutting concerns without much effort.

In a microservice architecture, you'll either need to incur the overhead of separate modules for each cross-cutting concern, or encapsulate cross-cutting concerns in another service layer that all traffic gets routed through.

Eventually, even monolthic architectures tend to route traffic through an outer service layer for cross-cutting concerns, but with a monolithic architecture, it's possible to delay the cost of that work until the project is much more mature.

Microservices are frequently deployed on their own virtual machines or containers, causing a proliferation of VM wrangling work. These tasks are frequently automated with container fleet management tools.

Good to hear:

- Positive attitudes toward microservices, despite the higher initial cost vs monolthic apps. Aware that microservices tend to perform and scale better in the long run.
- Practical about microservices vs monolithic apps. Structure the app so that services are independent from each other at the code level, but easy to bundle together as a monolithic app in the

beginning. Microservice overhead costs can be delayed until it becomes more practical to pay the price.

Red flags:

- Unaware of the differences between monolithic and microservice architectures.
- Unaware or impractical about the additional overhead of microservices.
- Unaware of the additional performance overhead caused by IPC and network communication for microservices.
- Too negative about the drawbacks of microservices. Unable to articulate ways in which to decouple monolithic apps such that they're easy to split into microservices when the time comes.
- Underestimates the advantage of independently scalable microservices.

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

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

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

User interfaces are asynchronous by nature, and spend most of their time waiting for user input to interrupt the event loop and trigger event handlers.

Node is asynchronous by default, meaning that the server works in much the same way, waiting in a loop for a network request, and accepting more incoming requests while the first one is being handled.

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

Good to hear:

- An understanding of what blocking means, and the performance implications.
- An understanding of event handling, and why its important for UI code.

Red flags:

- Unfamiliar with the terms asynchronous or synchronous.
- Unable to articulate performance implications or the relationship between asynchronous code and UI code.

. . .

Conclusion

Stick to high-level topics. If they can answer these questions, that typically means that they have enough programming experience to pick up language quirks & syntax in a few weeks, even if they don't have a lot of JavaScript experience.

Don't disqualify candidates based on stuff that's easy to learn (including classic CS-101 algorithms, or any type of puzzle problem).

What you really need to know is, "does this candidate understand how to put an application together?"

That's it for the spoken interview.

In real interviews, I place a much stronger emphasis on coding challenges and ***watching candidates code.*** Those topics are covered in depth in my "Master the JavaScript Interview" series.

. . .

. . .

Eric Elliott *is the author of "Programming JavaScript Applications" (O'Reilly), and cofounder of DevAnywhere.io. He has contributed to software experiences for* ***Adobe Systems****,* ***Zumba Fitness****,* ***The Wall Street Journal****,* ***ESPN****,* ***BBC****, and top recording artists including* ***Usher****,* ***Frank Ocean****,* ***Metallica****, and many more.*

He works anywhere he wants with the most beautiful woman in the world.