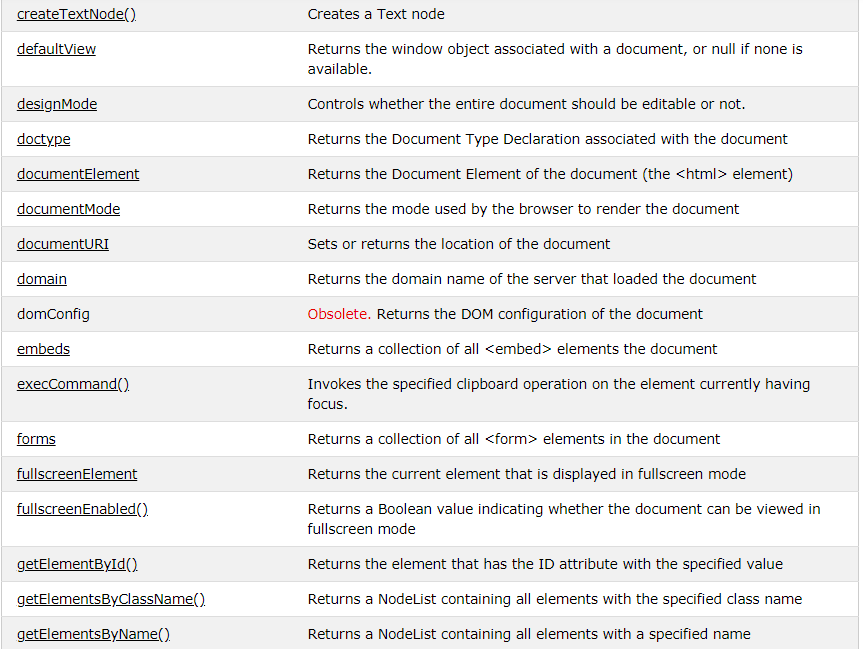
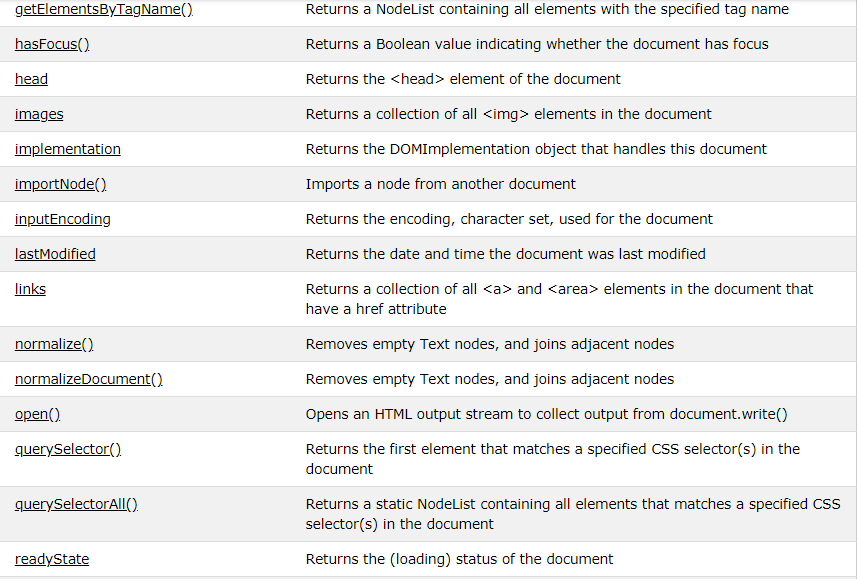
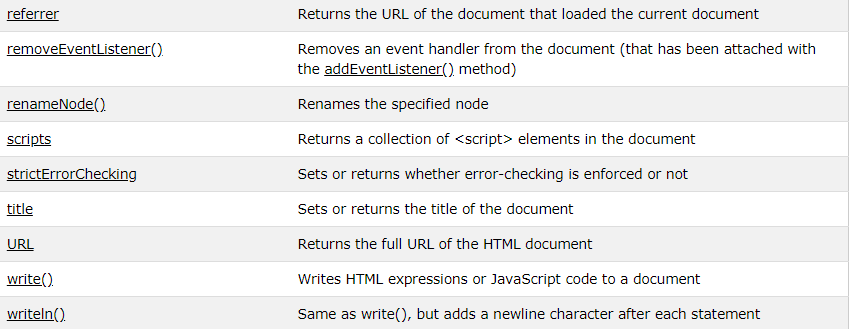
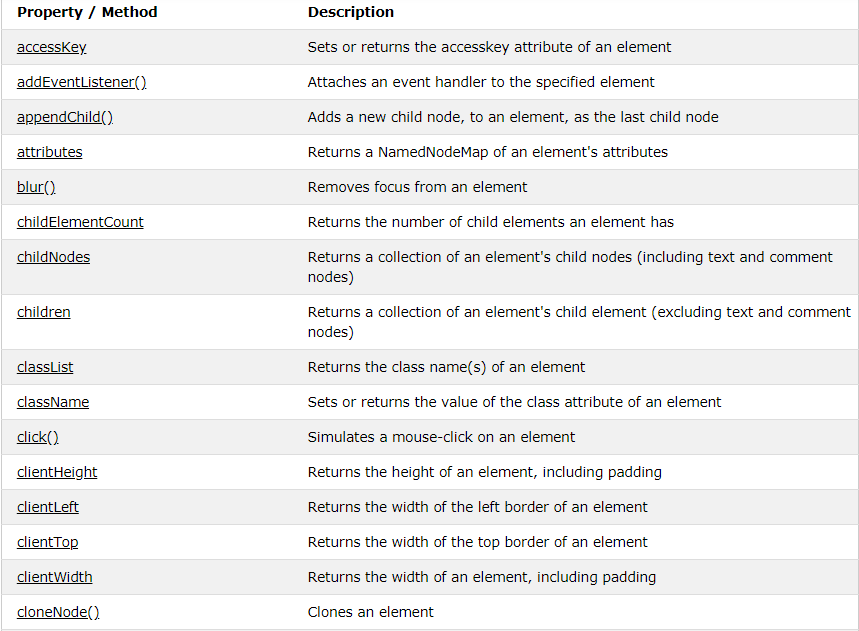


****

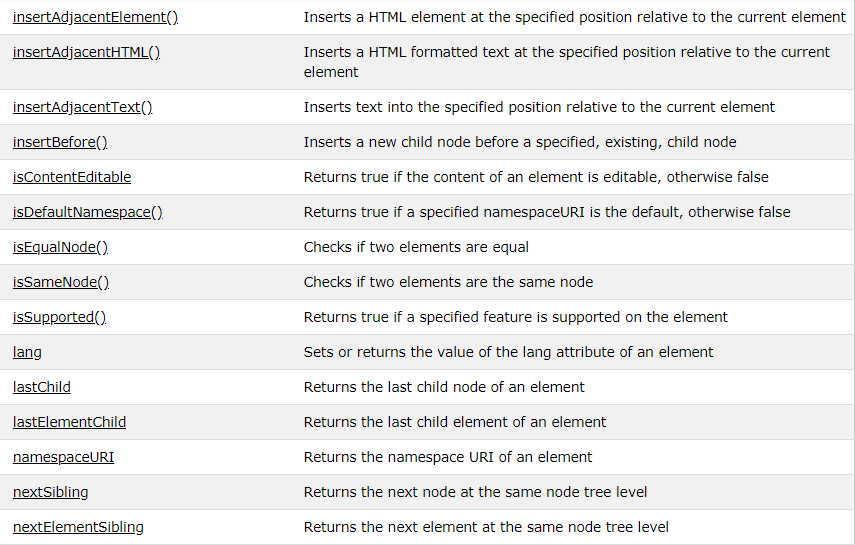
****

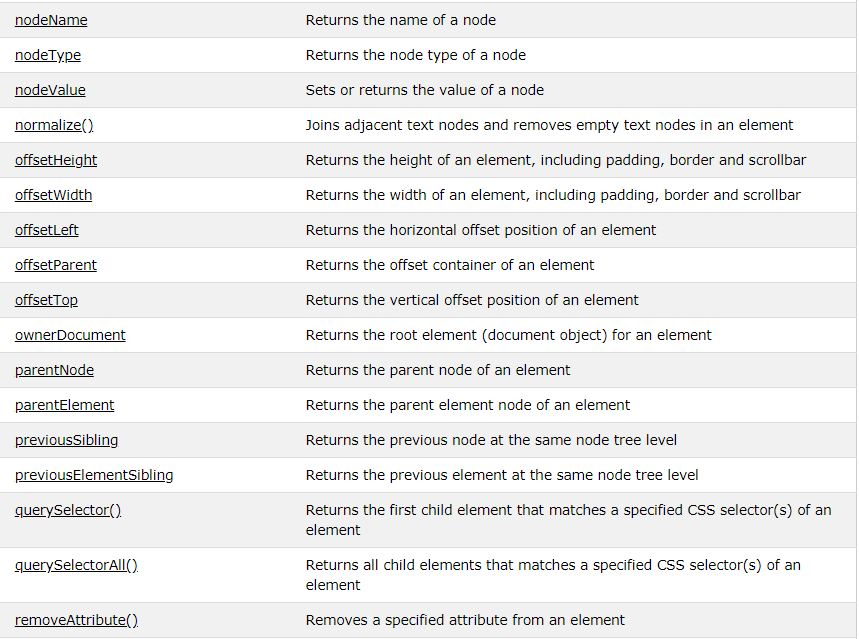
****

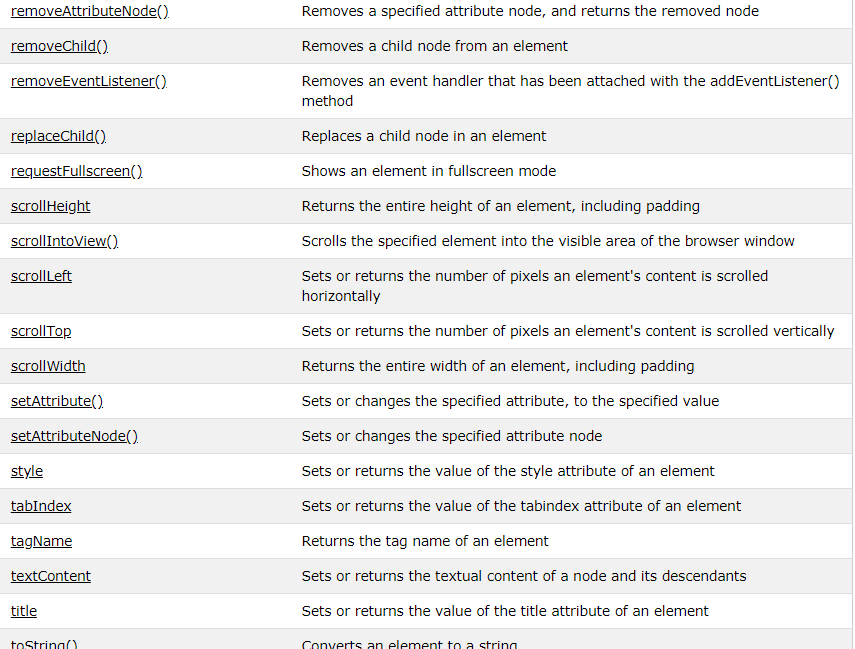
**The Element Object Properties and Methods**



****

****

****

****

**The Attr Object**

In the HTML DOM, the Attr object represents an HTML attribute.

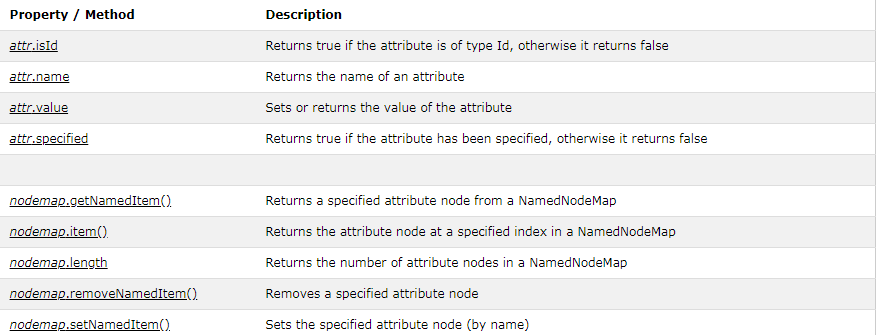
An HTML attribute always belongs to an HTML element.

**The NamedNodeMap Object**

In the HTML DOM, the NamedNodeMap object represents an unordered collection of an elements attribute nodes.

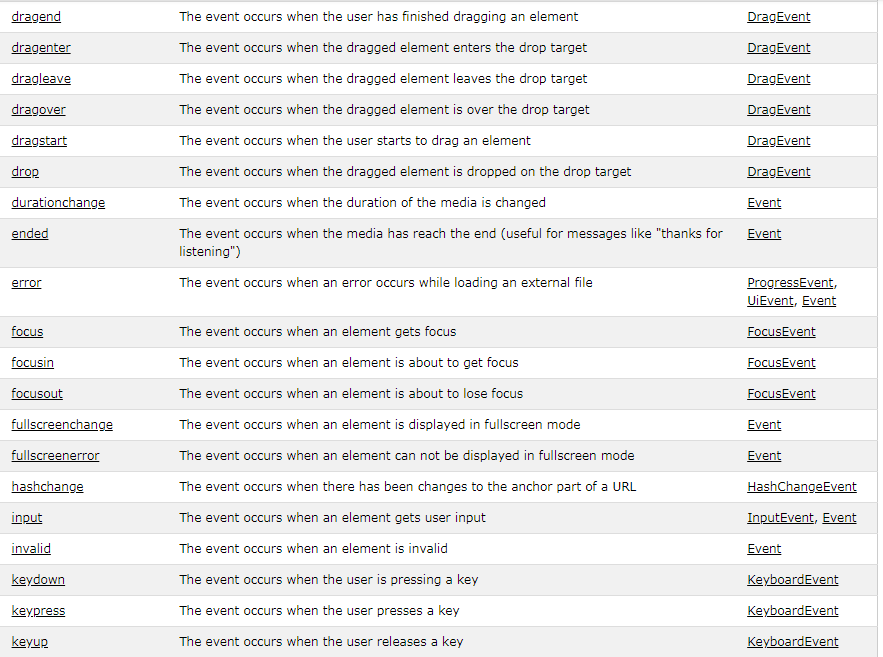
Nodes in a NamedNodeMap can be accessed by name or by index (number).

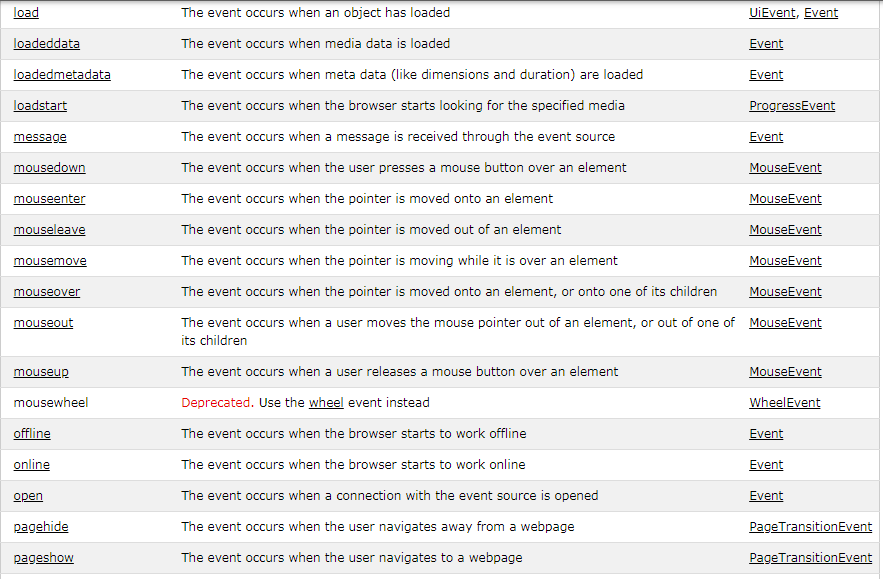
**Properties and Methods**

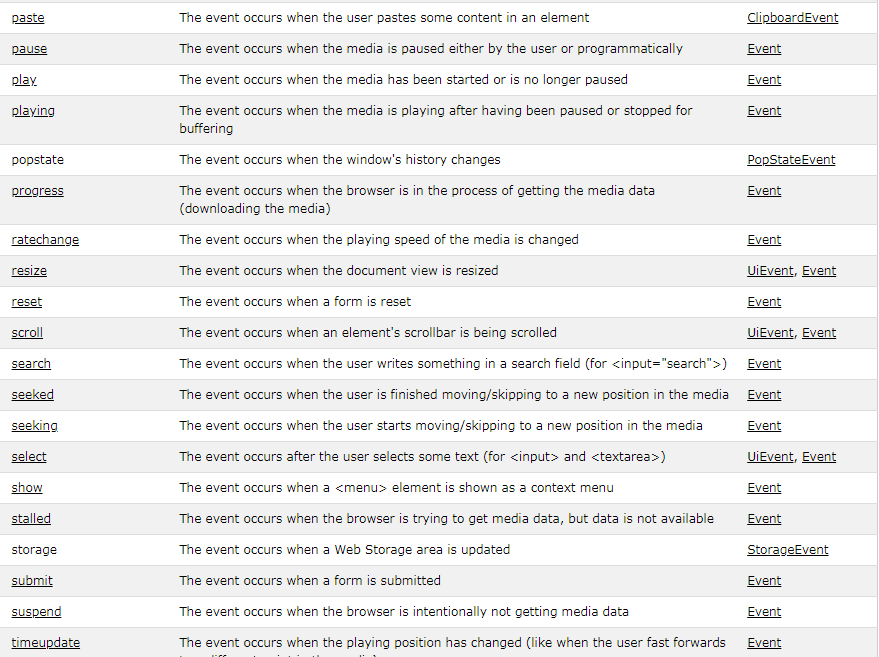
****

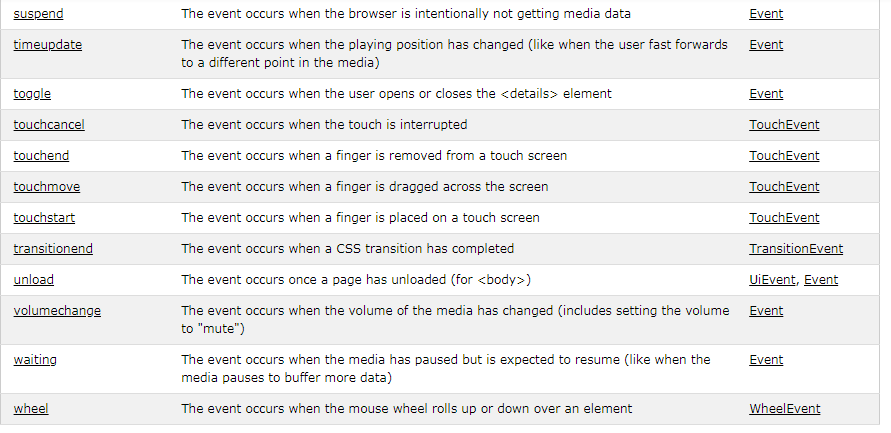
**HTML DOM Events**

****

****

****

****

****

**Here are some of the common JavaScript events:**

**change::** An HTML element has been changed

**click::** The user clicks an HTML element

**mouseover::** The user moves the mouse over an HTML element

**mouseout::** The user moves the mouse away from an HTML element

**keydown::** The user pushes a keyboard key

**load::** The browser has finished loading the page

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”](https://medium.com/javascript-scene/how-to-build-a-high-velocity-development-team-4b2360d34021), 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.”*

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”](https://medium.com/javascript-scene/why-hiring-is-so-hard-in-tech-c462c3230017) for a good breakdown of the general do’s and don’ts of candidate evaluation.

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](https://medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3) — Prototypal OO.
* [The Two Pillars of JavaScript Part 2](https://medium.com/javascript-scene/the-two-pillars-of-javascript-pt-2-functional-programming-a63aa53a41a4) — 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](https://medium.com/javascript-scene/the-two-pillars-of-javascript-pt-2-functional-programming-a63aa53a41a4).
* [The Dao of Immutability](https://medium.com/javascript-scene/the-dao-of-immutability-9f91a70c88cd).
* [Composing Software](https://medium.com/javascript-scene/composing-software-an-introduction-27b72500d6ea).
* [The Haskell School of Music](http://haskell.cs.yale.edu/wp-content/uploads/2015/03/HSoM.pdf).

#### 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](https://medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3) — Prototypal OO.
* [Common Misconceptions About Inheritance in JavaScript](https://medium.com/javascript-scene/common-misconceptions-about-inheritance-in-javascript-d5d9bab29b0a).

#### 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](https://medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3) — Prototypal OO.
* [The Two Pillars of JavaScript Part 2](https://medium.com/javascript-scene/the-two-pillars-of-javascript-pt-2-functional-programming-a63aa53a41a4) — 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](https://medium.com/javascript-scene/common-misconceptions-about-inheritance-in-javascript-d5d9bab29b0a). 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](https://medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3) — Prototypal OO.
* [JS Objects — Inherited a Mess](http://davidwalsh.name/javascript-objects).

#### 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](http://chimera.labs.oreilly.com/books/1234000000262/ch03.html#chcsrdou100015eilvj6l9inj).

#### 7. What does “favor object composition over class inheritance” mean?

This is a quote from [“Design Patterns: Elements of Reusable Object-Oriented Software”](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612). 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:**

[**Introducing**  
**the Stamp Specification**  
Move Over, `class`:  
Composable Factory Functions Are Heremedium.com](https://medium.com/p/77f8911c2fee)

#### 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”](https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closure-b2f0d2152b36) series.

### ****What is a Closure?****

A closure is an **inner** function that has access to the **outer** function’s variables.

var outerNumber = 30

function outer() {  
 var innerNumber = 20

function inner() {  
 return outerNumber + innerNumber  
}

return inner()  
}

outer() //prints 50

Here the function **inner** is considered a closure. That’s it.

Here is a closure written in ES6

const outer = () => {  
 let i = 0;

return () => {  
 return i++;  
 }  
}

let increment = outer();  
increment() //prints 0  
increment() //prints 1

### Why is this Important?

var outerNumber = 30

function outer() {  
 var innerNumber = 20

function inner() {  
 return outerNumber + innerNumber  
}

return inner()  
}

outer() //prints 50

**As you can see in this example, a closure is simply a function inside of another function.**

What is unique is that the inner function **inner** has access to variables in outer function, **outer's** scope, which in our case is the variable **outerNumber.**

Having said that, an important facet of a closure is that the inner function still has access to the outer function variables even after the outer function has returned.

### What the hell does that mean?

Let’s look at this example below:

function fullName(firstName) {  
 const intro = 'Hello'  
   
 function combineNames(lastName) {  
 return `${intro} ${firstName} ${lastName}!`  
 }  
 return combineNames  
}

const bretFullName = fullName('Bret')

bretFullName(); //prints `Hello Bret undefined!

bretFullName('Doucette'); //prints `Hello Bret Doucette!`

When I write const bretFullName = fullName('Bret') the outer function fullName is called and returned.

Remember I said above that the **inner function still has access to the outer function variables even after the function has been returned.**

In our case the inner function is combineNames. It still has access to the outer function variables, firstName === 'Bret’, despite the fact that fullName, the outer function has been called and returned.

So when I wrote in my console const bretFullName = fullName('Bret'), the bretFullName variable nows contains the inner function combineNames, and a reference to the variables of the outer function, firstName.

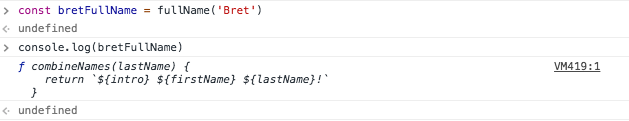
Therefore when I call bretFullName() I see back the following text:

Hello Bret undefined!

**RECAP: The VARIABLE bretFullName now contains a reference to the following 3 things:**

* Variable of the outer function intro === 'Hello'
* Variable of the outer function firstName === 'Bret'
* The inner function combineNames

As proof that the variable bretFullName contains the inner function, look at what happens when I console.log the variable bretFullName.



The variable fullName contains a function

Continuing on… now when we call bretFullName('Doucette') the following will print out:

Hello Bret Doucette.

### Why?

Like I said above bretFullName now contains reference to the inner function. I proved this in the console.log above.

function fullName(firstName) {  
 const intro = 'Hello'  
   
 function combineNames(lastName) {  
 return `${intro} ${firstName} ${lastName}!`  
 }  
 return combineNames  
}

const bretFullName = fullName('Bret')

bretFullName(); //prints `Hello Bret undefined!

bretFullName('Doucette'); //prints `Hello Bret Doucette!`

So we call bretFullName with the string Doucette, it passes the the string Doucette as a parameter to the inner function.

As stated above, the inner function **ALSO** has access to the variables of the outer function intro == 'Hello'& firstName === 'Bret' despite the fact that the outer function fullName has already been called and returned.

Having everything the inner function needs, the inner function executes its code:

return `${intro} ${firstName} ${lastName}!`;

#### ****AND the result is:****

Hello Bret Doucette!

Boom!

### Conclusion

I know that was a lot. But let’s look at what just happened.

When you have a closure, the inner function, **the closure**, still has access to the outer function’s variables. It does this by storing references to the outer function’s variables. It does not store the actual value.

The result is that you can call the inner function later in your program.

#### Another tid bit on closures…

**Closures ensure data privacy.** The variables enclosed by the closure are only in scope within the containing (outer) function. For example…

var add = function () {  
 var counter = 0  
 return function () {  
 return counter += 1  
 }  
}()

\*note I immediately invoke the function here

add(); //prints 1  
add(); //prints 2  
add(); //prints 3

Here the variable counter in the anonymous closure function can access the counter variable in the outer function.**Even cooler, the variable counter in the anonymous function is protected by the scope of the anonymous function and can only be changed using the add function.**

Closures provide a great way to more or less create private variables.

### When Can Closures Go Wrong?

Memory leaks = memory that isn’t being used and that also isn’t being picked up by garbage collection.

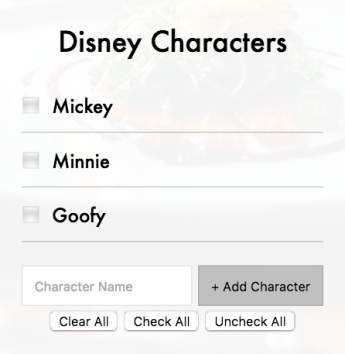
Simply speaking with JavaScript, garbage collection works by keeping variables that have a reference and cleaning up those that do not.

As I stated above, closures work by storing references to the outer function’s variables. They do not store the actual value. So if you are working with rather large datasets and have lots closures that are storing references to these rather large datasets, garbage collection will not clean up these references.

This references will take up memory and overtime you could expect your app’s performance to slow down or downgrade.

### Event Delegation

Event Delegation solves this problem. To understand Event Delegation, we need to look below at our list of Disney Characters.



Thank you Wes Bos for the CSS styles!

This list has some basic functionality. For our purposes, you can add characters to the list, and you can check the boxes next to the characters name.

This list is also dynamic. The inputs (Mickey, Minnie, Goofy) were added **AFTER the initial page load,**and subsequently didn’t have event listeners attached to them.

Let’s take a look at the code:

const checkBoxes = document.querySelectorAll(‘input’)

checkBoxes.forEach(input => input.addEventListener(‘click’, ()=> alert(‘hi!’)))

**//an alert should fire when I click on the inputs (Mickey, Minnie, or Goofy)**

But lets’ take a look at the HTML **AT** PAGE LOAD**:**

<ul class=”characters”>  
</ul>

Now let’s take a look at the HTML **AFTER** PAGE LOAD (from local storage, API call, etc)**:**

<ul class=”characters”>  
 <li>  
 <input type=”checkbox” data-index=”0" id=”item0">  
 <label for=”item0">Mickey</label>  
 </li>  
   
 <li>  
 <input type=”checkbox” data-index=”1" id=”item1">  
 <label for=”item1">Minnie</label>  
 </li>  
   
 <li>  
 <input type=”checkbox” data-index=”2" id=”item2">  
 <label for=”item2">Goofy</label>  
 </li>  
</ul>

**\*\* The inputs were placed on the DOM after page load and DID NOT have event listeners bound to them.**

If you try to click on the inputs (the characters — Mickey, Minnie, or Goody), you would expect an alert to pop up that says ‘hi!’, but because they weren’t present at page load, the event listeners WERE NOT bound to them, and subsequently nothing happens.



The alert ‘hi’ does not appear!

### So … how do we fix this problem?

Event Delegation.

The whole idea behind event delegation is that instead of listening for a change on the inputs directly,**we should look for an HTML element that is going to be on the page when the page initially loads.**

In our example — the unordered list with the class name characters is on the page at page load. We can attached the event listener there!

**<ul class=”characters”> // PARENT - ALWAYS ON THE PAGE**  
 <li>  
 **<input type=”checkbox” data-index=”0" id=”char0"> //CHILD 1**  
 <label for=”char0">Mickey</label>  
 </li>  
   
 <li>  
 **<input type=”checkbox” data-index=”1" id=”char1"> //CHILD 2**  
 <label for=”char1">Minnie</label>  
 </li>  
   
 <li>  
 **<input type=”checkbox” data-index=”2" id=”char2"> //CHILD 3**  
 <label for=”char2">Goofy</label>  
 </li>  
</ul>

It is best to think of event delegation as responsible parents and negligent children. The parents are basically gods, and the children have to listen to whatever the parents say. The beauty is if we add more children (more inputs), the parents stay the same — they were there from the beginning or, in other words, on page load.

Let’s attach the event listener.

**<ul class=”characters”>  
</ul>**

<script>  
 function toggleDone (event) {  
 console.log(event.target)  
 }

const characterList = **document.querySelector('.characters')**  
  **characterList.addEventListener('click', toggleDone)**  
</script>

So now that we have an event listener set on the unordered list, charactersand not the individual children (the actual characters), what happens if we click an input (Mickey, Minnie, or Goofy) after page load and **console.log**that**event.target?**

#### Console.log(event.target)

The following event target is returned.



event.target

The event.target is a reference to the object that dispatched the event. **Or in other words, it identifies the HTML element on which the event occurred.**

The **event in our case is the click!** The **object on which the event occurred is the <input/>**.

\*\* A label is considered part of the input object — that is why we see both. \*\*

#### Console.log(event.currentTarget)

If we **console.log(event.currentTarget)**— we see something different.



event.currentTarget

The **event.currentTarget** identifies the current target for the event, as the event traverses the DOM. **It always refers to the element to which the event listener has been attached.**In our case the event listener was attached to the unordered list, characters, so that is what we see in our console.

### Writing Event Delegation in JavaScript

Because we now know that the **EVENT.TARGET** **identifies the HTML elements on which the event occurred, and we also know what element we want to listen for (the input element),**solving this in JavaScript is relatively easy.

//Event Delegation

function toggleDone (event) {  
 if (!event.target.matches(‘input’)) return  
 console.log(event.target)  
 //We now have the correct input - we can manipulate the node here  
}

Basically the code above states, if the event target that was clicked **DOES NOT**match the input element, exit the function.

If the event target that was clicked **DOES**match the input element, console.log the event.target and execute subsequent JavaScript code on that child node.

**This is important**, now we can be confident that a user clicked the correct child node, even though the inputs were added to the DOM **after** the initial page load.

### Event Bubbling

If you want to stop reading here — by all means please do! We just covered the basics of event delegation. But for a deeper understanding of why event delegation works, we need to understand **Event Bubbling.**

#### What REALLY happens when you make a click?

Whenever a user makes a click it ripples up all the way up to the top of the DOM and triggers**clicks events on all the parent elements of the element you clicked**. You don’t always see these clicks, because you aren’t always listening (with an event listener) for a click on these elements, but this bubbling up does happen.

**This is called event bubbling or event propagation.**

Because of its bubbling nature, event propagation basically means that anytime you click one of our inputs on the DOM, you are effectively clicking the entire document body.

Here is an example in action:

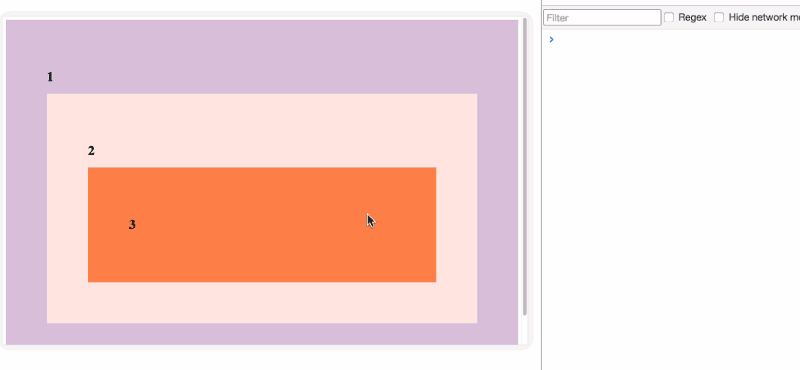
<div class=”one”>  
 <div class=”two”>  
 <div class=”three”>  
 </div>  
 </div>  
</div>

<script>  
 const divs = document.querySelectorAll('div');

function logClassName(event) {  
 console.log(this.classList.value);  
 }

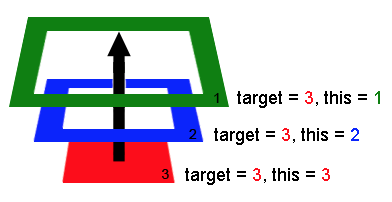
divs.forEach(div => div.addEventListener('click', logClassName));  
</script>

Above we have three divs: DIV #1, DIV #2, DIV #3. Each DIV has it’s own event listener, and when we click on a DIV in browser, we ask it to console.log its class name by executing the function, logClassName().



Thanks Wes Bos again!

Above is what we see in the browser. **Notice how my mouse is clicking DIV #3**. As one would expect, when I click DIV #3, I see the class name logged in the console (to the right). **But when I click DIV #3, I also DIV #2 and DIV #1 being logged to the console. This is event bubbling!**We see the class name logged, because we added event listeners to each parent **div**.



Source: <https://javascript.info/> — clicking on DIV #3 bubbles up to 2, and 1.

#### ****Back to our Event Delegation Example****

**<ul class=”characters”> // PARENT -- This is where the listener is!**  
 <li>  
 **<input type=”checkbox” data-index=”0" id=”char0"> //CHILD 1**  
 <label for=”char0">Mickey</label>  
 </li>  
 <li>  
 **<input type=”checkbox” data-index=”1" id=”char1"> //CHILD 2**  
 <label for=”char1">Minnie</label>  
 </li>  
 <li>  
 **<input type=”checkbox” data-index=”2" id=”char2"> //CHILD 3**  
 <label for=”char2">Goofy</label>  
 </li>  
</ul>

<script>  
 const characterList = document.querySelector('.characters');  
 characterList.addEventListener('click', toggleDone);  
</script>

So back to our example in the event delegation section — we had only one event listener, and it was set on our unordered list characters. **YET** when we clicked a child of that parent HTML element, the HTML element input, it fired the event listener we set that was bound to the unordered list.

Because of event bubbling you can place an event listener on a single parent HTML element that lives above a HTML child, and that event listener will get executed whenever an event occurs on any of its child nodes — even if these node children are added to the page after the initial load!

### In Conclusion — Why Use Event Delegation?

Without event delegation you would have to rebind the click event listener to each new input loaded to the page. Coding this is complicated and burdensome. For one, it would drastically increase the amount of event listeners on your page, and more event listeners would increase the total memory footprint of your page. Having a larger memory footprint decreases performance… and poor performance is a bad thing. Second, there can be memory leak issues associated with binding and unbinding event listeners and removing elements from the dom. But that is beyond the scope of this article!  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**first class functions**

A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**pure functions / function purity**

A pure function is a function where the return value is only determined by its input values, without observable side effects. ... When a function performs any other “action”, apart from calculating its return value, the function is impure. It follows that a function which calls an impure function is impure as well.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Function composition (computer science)**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **function composition** (not to be confused with [object composition](https://en.wikipedia.org/wiki/Object_composition)) is an act or mechanism to combine simple [functions](https://en.wikipedia.org/wiki/Subroutine) to build more complicated ones. Like the usual [composition of functions](https://en.wikipedia.org/wiki/Function_composition) in [mathematics](https://en.wikipedia.org/wiki/Mathematics), the result of each function is passed as the argument of the next, and the result of the last one is the result of the whole.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Object Composition in Javascript

You’ve just landed your dream job as a game developer, building a role-playing game with mages and fighters. You’re comfortable with object oriented programming from your experience with Java and Ruby, but the new project is in Javascript. Fortunately you read last week’s post on [Javascript ‘Classes’ and Prototypal Inheritance](https://medium.com/code-monkey/javascript-classes-and-prototypal-inheritance-2a53ed7343d8) and you’re ready to dive in. You design your class structure and get started.

class Character {

constructor(name) {

this.name = name;

this.health = 100;

}

}

class Fighter extends Character {

constructor(name) {

super(name);

this.stamina = 100;

}

fight() {

console.log(`${this.name} takes a mighty swing!`);

this.stamina--;

}

}

class Mage extends Character {

constructor(name) {

super(name);

this.mana = 100;

}

cast() {

console.log(`${this.name} casts a fireball!`);

this.mana--;

}

}

zapper = new Mage('Zapper');

thumper = new Fighter('Thumper');

zapper.cast(); // Zapper casts a fireball!

console.log(zapper.mana); // 99

thumper.fight(); // Thumper takes a mighty swing!

console.log(thumper.stamina); // 99

Our RPG characters using Classes

A few months later you release your game and it’s an overnight success! Players love it, but after a while they become bored with just mages and fighters and they demand a new player type, ‘paladin’, that can cast spells **and**fight.

#### The Problems of Classical Inheritance

You look at your class structure and ponder how to implement the paladin. You consider a couple of options:

1. You could write a new Paladin class that extends from Character and then copy the fight() and cast() code from Fighter and Mage, but then you’d be duplicating code and you know that’s not a good solution.
2. You could move fight() and cast() code up into the Character class so that all three character types could use them. This is perhaps a better solution, but you’ll also need to override the fight() function in the Mage class and the cast() function in Fighter class.

Neither looks like a very good solution… and why are you having to muck about in code that’s working perfectly fine to add the Paladin?

You’ve just discovered some of the problems with classical inheritance.

* You have to define your class taxonomy in advance. The is just about impossible to get right the first time, except for trivial projects.
* If you don’t get it right, you’ll be forced to change it later. And unfortunately, the parents and the children are tightly coupled which requires you to make changes in many places, and probably add bugs.

These problems are so common that there are names for them. Option 1 is known as the duplication by necessity problem. Option 2 is called the [Gorilla / Banana problem](https://medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53).

The problem isn’t really with object-oriented languages. It’s with classical inheritance, which makes you think in terms of what things **are**rather than what they **do**.

#### Object Composition in Javascript

There are three primary methods for inheritance in Javascript. The first, prototype delegation, is the method we used to mimic class inheritance in the example above. The other two are concatenative inheritance and functional inheritance. Let’s use these to rewrite our characters without a concept of class.

const canCast = (state) => ({

cast: (spell) => {

console.log(`${state.name} casts ${spell}!`);

state.mana--;

}

})

const canFight = (state) => ({

fight: () => {

console.log(`${state.name} slashes at the foe!`);

state.stamina--;

}

})

const fighter = (name) => {

let state = {

name,

health: 100,

stamina: 100

}

return Object.assign(state, canFight(state));

}

const mage = (name) => {

let state = {

name,

health: 100,

mana: 100

}

return Object.assign(state, canCast(state));

}

scorcher = mage('Scorcher')

scorcher.cast('fireball'); // Scorcher casts fireball!

console.log(scorcher.mana) // 99

slasher = fighter('Slasher')

slasher.fight(); // Slasher slashes at the foe!

console.log(slasher.stamina) // 99

RPG Characters using Object Composition

In this example, we focus on what an object can **do**rather than trying to define what it **is**. We define behaviors using [factory functions](https://medium.com/javascript-scene/javascript-factory-functions-with-es6-4d224591a8b1) that accept state and return an object that acts upon it.

( *A factory function is any function which is not a class or constructor that returns a (presumably new) object. In JavaScript, any function can return an object. When it does so without the new keyword, it’s a factory function.* )

Our canCast() function accepts state and returns an object with the cast() function. When we execute cast(‘fireball’) the function logs to the console and reduces the mana by 1. Note that we could have many functions on the object returned by canCast(), perhaps one for each spell, but it’s more future-safe to have only one.

We create our mage with another factory function. In this case, we define our state with some initial values, and then we use [Object.assign()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) to create and return a new object that includes the properties from state and also the cast() function. This copying of properties from one or more objects into a new one is known as concatenative inheritance.

Our brand new paladin can be created like this:

Roland, our new Paladin

const paladin = (name) => {

let state = {

name,

health: 100,

mana: 100,

stamina: 100

}

return Object.assign(state, canCast(state), canFight(state));

}

roland = paladin('Roland');

roland.fight(); // Roland slashes at the foe!

roland.cast('Holy Light'); // Roland casts Holy Light!

#### `new` and `this`

You may have noticed one difference between object composition and classical inheritance that I haven’t mentioned. In classical inheritance, you create a new object using the new keyword, but you don’t use new with factory functions. This is another advantage that object composition has. When using classical inheritance, it is easy to forget the new when instantiating an object and Javascript doesn’t warn you if you do. Instead it will execute your constructor function as a regular function. This can lead to difficult to diagnose bugs.

Additionally, you probably noticed that the object composition approach completely avoided the use of this. The this keyword is confusing to many, and doesn’t behave quite the same way as it does in other languages, so avoiding it helps in code comprehension.

I should point out here that avoiding this comes at a cost. Because Object.assign() copies the properties (including functions) from one object to another, you are increasing the memory burden. When you use prototype delegation and the this keyword you don’t duplicate the functions and properties, you delegate instead. That’s the value of prototypal inheritance, but it doesn’t mean you have to emulate classes. I’ll be exploring this in a future article.

#### Summary

* In classical inheritance, we tend to think of our objects in terms of what they **are**, but when using object composition we think about what they can **do**.
* Classical inheritance is difficult to do correctly, and difficult to change later.
* Javascript provides simple tools for object composition that avoid the pitfalls of classical inheritance.

#### Parting words

It may seem as though I’ve been bashing pretty hard on classical languages in my last couple of posts. If you’ve done much OOP you already know you should prefer object composition over inheritance, and many classical languages address this need through interfaces or modules. Unfortunately, we are nearly all taught about classes first and interfaces and object composition later. Why?

And to be fair, creating objects using constructor functions and prototype delegation is faster than factory functions, but the difference is so small that unless you are creating 10s of thousands of objects per second, it won’t have a significant impact.

I encourage you to play with the code above by copying it to [repl.it](https://repl.it/)

See if you can add an Archer to the party, or modify the behaviors so that they can’t be used too often!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<style type="text/css">

a { text-decoration: none; text-transform: uppercase; cursor: help;}

a::before { content: ">" }

a::after { content: "<" }

</style>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<section>

<h1>Lorem Ipsum</h1>

<figure class="image">

<img src="https://goo.gl/zF9eky" alt="Lorem Ipsum">

<figcaption class="caption">Lorem Ipsum</figcaption>

</figure>

<details class="lorem-ipsum">

<summary class="description">Lorem ipsum dolor sit amet, consectetur adipiscing elit...</summary>

<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Nam fermentum posuere mauris, quis aliquam nibh dictum sed.</p>

</details>

</section>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

function appendChildren(decorateDivFunction) {

var allDivs = document.getElementsByTagName("div");

let test = allDivs.length;

for (var i = 0; i < test; i++) {

var newDiv = document.createElement("div");

decorateDivFunction(newDiv);

allDivs[i].appendChild(newDiv);

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

@supports (display: grid) {

.gridExample {

display: grid;

/\* grid-template-columns: repeat(5, 20%); \*/

grid-template-columns: repeat(5, minmax(60px, 1fr));

border: 1px dotted grey;

background-color: #eee;

}

.gridExample > \* { border: 1px dotted red; }

.gridExample h3 {

grid-area: 2/2/3/5;

background: rgba(255, 255, 0, 0.75);

align-self: center;

}

.gridExample li { list-style-type: none; background: rgba(100, 255, 0, 0.75) }

.gridExample .rule1 { grid-area: 4/2/5/3; }

.gridExample .rule2 { grid-area: 5/3/6/4; }