Javascript Jabber

168 JSJ The Future of JavaScript with Jafar Husain

JOE:

Hey, before we start though Jafar, I just want to thank you for getting me added. I hope I wasn't making a pest of myself.

JAFAR:

Oh added, sorry, to?

JOE:

Oh, the Falcor.

JAFAR:

Oh yeah, absolutely. No problem.

JOE:

Yeah.

CHUCK:

Apparently it wasn't a big deal, because he didn't remember.

JOE:

[Laughs] Yeah, he totally can't remember.

[Laughter]

JOE:

I have an assistant who does all that crap.

[Laughter]

CHUCK:

Yeah. Whatever you're grateful to me for, it's cool.

[Laughter]

JOE:

Yeah. Whatever I did, I'm sure it was amazing. You're welcome.

CHUCK:

Yes, I know I'm awesome.

[This episode is sponsored by Frontend Masters. They have a terrific lineup of live courses you can attend either online or in person. They also have a terrific backlog of courses you can watch including JavaScript the Good Parts, Build Web Applications with Node.js, AngularJS In-Depth, and Advanced JavaScript. You can go check them out at FrontEndMasters.com.]

[This episode is sponsored by Hired.com. Every week on Hired, they run an auction where over a thousand tech companies in San Francisco, New York, and L.A. bid on JavaScript developers, providing them with salary and equity upfront. The average JavaScript developer gets an average of 5 to 15 introductory offers and an

average salary offer of $130,000 a year. Users can either accept an offer and go right into interviewing with the company or deny them without any continuing obligations. It's totally free for users. And when you're hired, they also give you a $2,000 bonus as a thank you for using them. But if you use the JavaScript Jabber link, you'll get a $4,000 bonus instead. Finally, if you're not looking for a job and know someone who is, you can refer them to Hired and get a $1,337 bonus if they accept a job. Go sign up at Hired.com/JavaScriptJabber.]

[This episode is sponsored by Wijmo 5, a brand new generation of JavaScript controls. A pretty amazing line of HTML5 and JavaScript products for enterprise application development in that Wijmo 5 leverages ECMAScript 5 and each control ships with AngularJS directives. Check out the faster, lighter, and more mobile Wijmo 5.]

[This episode is sponsored by DigitalOcean. DigitalOcean is the provider I use to host all of my creations. All the shows are hosted there along with any other projects I come up with. Their user interface is simple and easy to use. Their support is excellent and their VPS's are backed on Solid State Drives and are fast and responsive. Check them out at DigitalOcean.com. If you use the code JavaScriptJabber, you'll get a $10 credit.]

[This episode is brought to you by Braintree. If you're a developer or a manager of a mobile app and searching for the right payments API, check out Braintree. Braintree's new v.zero SDK makes it easy to support multiple mobile payment types with one simple integration. To learn more and to try out their sandbox, go to BraintreePayments.com/JavaScriptJabber.]

CHUCK:

Hey everybody and welcome to episode 168 of the JavaScript Jabber Show. This week on our panel we have Aimee Knight.

AIMEE:

Hello.

CHUCK:

Dave Smith.

DAVE:

Hey everyone.

CHUCK:

Joe Eames.

JOE:

Hey everybody.

CHUCK:

Jamison Dance.

JAMISON:

Hi friends.

CHUCK:

AJ O'Neal.

AJ:

Yo, yo, yo.

CHUCK:

I'm Charles Max Wood from DevChat.tv. And this week we have a special guest and that's Jafar Husain.

JAFAR:

Hi everybody.

CHUCK:

Do you want to introduce yourself?

JAFAR:

Yeah, sure. My name's Jafar Husain. I work at Netflix as a tech lead. I'm also a representative on the TC39, which is the JavaScript Standards Committee. And I'm working with the folks on the committee to help evolve the JavaScript language. My likes are reactive programming, which we might be talking about a little bit today, and of course all the new great stuff coming down the pipe in ES6 and ES7.

DAVE:

Wait, what did you just call them?

JAFAR:

Oh, no.

CHUCK:

That's right, ES6 and ES7. It's official.

AIMEE:

[Chuckles]

JAMISON:

Yeah, yeah. [Inaudible]

CHUCK:

From TC39.

[Laughter]

JOE:

We're recording it now.

JAMISON:

We got it.

JOE:

I just tweeted it.

JAFAR:

Yeah, I have to train myself out of it. I have to train myself out of it. I know. It's ES2015 and ES2016.

JOE:

[Developed] 2015?

CHUCK:

You did that just to confuse us, right?

AIMEE:

[Laughs]

JAFAR:

You guys passed the test. You got me.

[Laughter]

JOE:

He was checking us.

JAFAR:

[Chuckles]

AIMEE:

That question was on my list. I wanted to establish this once and for all.

JAMISON:

Is 'ES6' dead now? Is it actually ES2015?

JOE:

I can clarify this for you. This is how it works. If you're speaking to a member of TC39, it's called JavaScript 2015.

[Laughter]

JOE:

If you're not, it’s ES6 and ES7.

JAFAR:

[Laughs] I think that's the ad hoc system that's developed. It's not as important to me though what people it. I want people to understand why we made that name change. And I think it was really to signal the community that we're going to be iterating more quickly on JavaScript. The reason why we're putting a number on it, I worked at Microsoft by the way and I had nothing to do with this. This was not my decision. But the reason why we're putting a year on this is just really to signal the community that look, ES 6 was a huge release. It took forever to get out. That's not going to be the case hopefully with ES2016, ES2017, and into the future.

DAVE:

So, you're planning on annual releases?

JAFAR:

That's exactly the point of the name change, yes.

JOE:

Now, will this move to monthly releases?

[Laughter]

JAFAR:

God, I hope not.

[Laughter]

JAMISON:

Weekly sprints?

[Laughter]

JAFAR:

Year releases, it feels like breakneck pace to me, to be honest, because when you work backwards all the hoops you have to jump through in order to get a feature into JavaScript, it's really, you got to pretty much be done by six months. But you know what? That's how software works, right? A lot of software works on the train model. And JavaScript is software. The spec process, often we talk about the spec as software. And so, it's certainly formalized and expressed as formally as possible. So, I think it's possible to do this in the train model. And that's why we're doing it, because I think the committee got some criticism over how long it took to get ES6 out of there. I think some of that criticism is valid. But there are good reasons why it took so long. But I think this is our attempt to work more incrementally.

JOE:

So, can I ask the question that everybody who has ever done Agile, and everybody who has ever worked with IE is thinking, and that is…

CHUCK:

[Laughs]

JOE:

If the pace was so slow before, what's changing that makes you guys feel confident that a yearly release is going to be viable?

JAFAR:

Well, so there was a lot of stuff in ES6 not just because it was a bunch of disconnected features, but a lot of those features were built and layered on top of each other. Promises needed to have things like job scheduling for example, which is this new concept of, I don't know [if many] people have heard of microtasks, this idea that now JavaScript's taking control of the event loop. There were some really big pieces in there, foundational pieces that we wanted to get in place. And we're seeing now that we're actually developing new features in ES2016 on them. So, that's part of what it was.

It was about really taking a hard look at a language that we hadn't taken a hard look at in seven years, or we went through that whole long contentious ES4 process. And ES5 was this tiny, tiny release. And so, this language just hadn't had a big refresh and it had been a long time. So, I think now that we've established a base of features that we can build on top of, I think you're doing to start to see the features come more fast and furious and not have to be built from the ground up. Async/await is a great example of this. It's building on top of generators and promises. So, it's a very incremental feature. That's really the explanation for why we had to… it took a long time to do ES6. It's because JavaScript was real dusty.

CHUCK:

Now, while we're still on the topic of the release cycle and things like that, I'm curious what this means for browsers and their support cycles and things like that.

JOE:

Yeah, because to be honest, they suck.

JAFAR:

[Laughs] Well, I can tell you that I would certainly like to see more ES6 features in browsers. I think that along with other developers, are frustrated by the fact that even in IO.js I can't use arrow functions, which to me seem like a simple feature. That said, it's important to temper this frustration and enthusiasm for the new features with the knowledge that the browser vendors have a lot on their plate. I'm being somewhat diplomatic here. I work with them on the committee. That said, I really don't think there's a lot of excuses to not see new features being developed concurrently in browsers as they're being spec'd.

What we have in JavaScript is we have this ladder process of maturity for features, the ES feature maturity stages. And so, as features move through stages, as part of moving through those stages, we actually want to see browser implementations out there. So, you don't have to wait for ES6 to have a bow on it, if I'm a browser implementer, to go and start implementing some of these features. I can implement them. I can put them behind a compatibility flag, like they did in Chrome with Object.observe for example. There's no reason for browser vendors to sit on their hands. I'm not saying they are. But I'm saying I'd certainly like to see more ES6 features make it into browsers quicker.

JOE:

I can agree with that. Right now, ES6, the very, very, very best browser is at 61% compatibility.

JAFAR:

Now to be fair, yeah, remember what I said about ES6 being a massive release. It is a massive release. I like to think that we're going to start to see a pipeline moving a little more quickly as the features become more incremental in ES2016.

JAMISON:

So, if Babel exists, why do you care?

JAFAR:

That's a great question. So, I love that question. And for the most part, I don't. Personally, I can tell you that we use a transpiler at Netflix. We're using JSX which gives us access to a lot of the ES6 and even ES7 features. In general, most modern JavaScript programs have the build step. And so, it shouldn't be that hard, what with the state of the world around symbol, being able to debug in the original source, source maps that is, it shouldn't be that hard to work with something like Babel. So, for the most part, I think that's valid. I think it's fair to say, “Yes, we can go ahead and use transpilers.” And at the same time, there's also no reason why browsers shouldn't be moving as quickly as they can at this point on ES6 features.

JOE:

Well, we've also got the issue that there's a lot of features that you can't implement with Babel or any other transpiler because the browser… like WeakMaps as an example.

JAFAR:

That's a fair point. You're right. Not everything in ES6 can be transpiled. I think you're going to start to see a lot more features coming into JavaScript that are transpilation, easy to transpile. Some of the big features for ES7 are, like decorators and async/await, are very easy to transpile. But yeah, some of those foundational features like proxies or WeakMaps as you point out, collections, yeah they're not so easy.

CHUCK:

So, one other fork of this discussion is just the discussion around what is it, the WebAssembly language? I forget exactly what it was called. But does that play into this? Do you think some of these browsers might just wait until they can just implement that?

JAFAR:

Well, I don't know. So, WebAssembly is still pretty darn new. So, I don't necessarily feel very confident talking very much about it. I just haven't had very much time to delve deep, deep into it. It certainly wasn't announced at TC39. It's sort of an effort parallel to that. And so, it's possible. My expectation is that, and I could be wrong here, I'm talking… and I'm [on a limb] a little bit. But WebASM I think is going to be more useful for closer to the metal features. I don't think you're going to necessarily see JavaScript features implemented in terms of WebASM. Or are you saying that it would be instead of? Like people would stop pedaling new JavaScript features and maybe focus on WebASM?

CHUCK:

Yeah, I was just wondering. So the ES… I wanted to say ES7 but ES2016, ES2017…

JAFAR:

[Chuckles]

CHUCK:

As we go down the line, are we just going to be adding capabilities to WebAssembly and then the rest of it is okay? And here's the spec and it needs to compile or transpile to WebAssembly.

JAFAR:

Well, my feeling is that you're still going to see the JavaScript language evolve, right?

CHUCK:

Right.

JAFAR:

I think WebAssembly is more to make it a level playing field for other programming languages. And also to expand the capabilities of the web platform.

CHUCK:

Right.

JAFAR:

So, the idea is if I want to write the next great programming language for the web, I can compile down to WebAssembly and then I can compete on a level playing field with JavaScript, which as I think most of us know is not necessarily the world's best programming language ever created. And so, I think that is probably a valuable thing. But it's more about democratizing the field, leveling the playing field, I think that replacing JavaScript. I think it's about giving us other options.

CHUCK:

Okay.

JOE:

Okay. So, WebAssembly is a somewhat new topic. Can maybe somebody give a little explanation for people that may not be familiar with it?

JAFAR:

Yeah, sure. So, from what I gather, WebAssembly is, in the past or up until now, JavaScript has really been the Assembly language of the web.

CHUCK:

Yes.

JAFAR:

And so, if you wanted to build a programming language, you had to work in JavaScript. Now, that actually put JavaScript in kind of an odd position, because it had to be both a programming language that was good to program in for human beings but also a pretty darn good compilation target. And that's actually a tough line to walk. And although it's been claimed that JavaScript actually does a pretty darn good job, maybe even better than Java bytecodes, there is definitely some areas in which a lower level language would produce better results, particularly if you're trying to get close to the hardware. These are things that WebAssembly promised that we'll be able to do, hopefully give us the ability to get closer to hardware, maybe whether it be control over threads for example or control over SIMD.

And then people can build new languages on top of that. I don't know that most developers are going to find themselves programming in WebAssembly. I think what's much more common is that languages will compile to WebAssembly and then people will program perhaps different languages other than JavaScript that are compiled to WebAssembly.

AJ:

So, does this have any relation to Google's NaCl?

JAFAR:

I would say… They are two totally separate projects, as far as I know. And I think that there are also probably subtle and important differences between the two of them. I think that in this particular case, they're both driving at something similar though, which is how do we get better performance out of the web platform? But I think that's where the similarities end.

AIMEE:

I had one other quick question before we move on from talking about browser implementation. So, on JavaScript Weekly this week there was a good post on the performance of [all] the ES6 features. So, my question was, at what point is it, you said you're working with the different browser vendors so, at what point is it priority for them to focus on performance over getting out new features for ES7? Because I see that that could potentially slow adoption of people. Some people are already kind of happy with the way things are and resistant to change.

JAFAR:

Yeah, I think the interesting thing is performance is a bit of the chicken and egg problem. On one hand if you're a browser vendor you're looking at the features the people use and that's where you want to prioritize your optimization. Some of these features are very complicated and very hard to get right. Generators is one example of something that's just really hard to get correct behavior out of. And at the moment I don't think in any of the major browsers that's a feature that's very well-optimized. And if a feature is perceived as slow, people aren't going to use it. And then that causes browser vendors to say, “Well, nobody's using it so I'm not going to optimize it.”

So, we do have this bit of a chicken and egg problem. I don't know what the right answer is. I don't know that there is a right answer. I remember being very frustrated that ES5 features were slow when they first made their way into browsers. So, I sympathize with developers. And I think… long story short, I don't know what the right thing for browser vendors to prioritize is. It might be doing breadth and effectively trying to implement as many features as possible and then trying to optimize the ones that really take off and people start using. Or it might be to take longer and release ES6 and really just make sure that when they finish a feature, they also focus on performance.

My personal opinion would be to tend towards the former, actually because there's limited… one of the resources that we on the committee have to contend with is the fact that these companies don't have unlimited… although many of these companies have seemingly unlimited resources, that doesn't mean that, whether it's Microsoft or Google, certainly less so Mozilla, that doesn't mean that they're willing to dedicate all those resources to working on a browser. And even if they could, it doesn't really scale. And so, they've got a lot of features on their plate. And just one of them competing for other features, are JavaScript features. So, that to me is kind of an open question.

DAVE:

Those self-driving cars aren't going to build themselves, you know.

[Laughter]

JAFAR:

It's true.

AJ:

So, I have a related question. It seems like a lot of features are easy to implement as polyfills. And obviously not ES6 but with a lot of stuff we've had in JavaScript it's been easy to implement as a polyfill. And yet consistently when browsers release these new features, they're really slow. And everybody on all the blogs are like, “Well, don't even use the native one. Use the polyfill because it's faster.” And so, one of my questions is why do these browser vendors even bother implementing things slowly and releasing them rather than just including a JavaScript file in with every single page? And then when they've got it fast enough, then switch over?

JAFAR:

Well you know, so that's a totally valid question. And it might not be… here's what, I want to pose that it might not be that it's slow because it's implemented natively, right? I think it might be slow because it's correct. And this is an example, I'll give you a simple example which is bind. Bind, I think it's trivial to polyfill if you use bind in the 99, 98% case. There are often edge cases in the language that we on the committee have to spend time worrying about because of course everything's got to be formalized. When you have multiple implementations of a language you've got to make sure that everything is very explicit.

So, we end up spending a lot of time doing something not very fun, which is worrying about something that the vast majority of developers will never, ever do and putting features in the language basically to make sure that we get the same consistent behavior from all the implementations when that happens. But that necessarily makes the implementation of certain functions like bind for example, more complicated than they need to be. And here's the thing. When people polyfill, they don't worry about that 2% or that 3% and they cut corners. And sometimes, that's why they get better performance.

AJ:

So, then why are you worried about that 2 or 3%? Because if it's like in 99% of cases, we need it to do this, I'd say, well consider the other 2% the bad parts. Don't do that.

JAFAR:

Well…

DAVE:

[Chuckles] Because that would mean Twitter has only 2 million failed tweets a day.

JAFAR:

[Laughs] Well, I think the important thing to understand here is that, “Don't do that” isn't an option for the spec, right? What we can do is if we want to make sure, what's great about the spec process is that it makes sure that no one… there's many a benefit to the spec process, but one of the big ones is that no one company owns the web, right? We need multiple implementations of JavaScript that all conform and all work. And it's very healthy to have all these implementations. But that means you can't really just gloss over stuff. You've got to be very explicit about what happens when somebody passes this parameter, these parameters in this order, to this function, on Wednesday when it's raining. And then every single browser needs to do the exact same thing.

Otherwise, people start relying, it's totally possible, people will start relying on un-spec'd behavior and then we've really got a problem. Because maybe it's not… if we don't think about it, then whatever implementation happens to work a certain way, that becomes the way the web works. And that's not necessarily, especially because I can tell you that if spec developers are not thinking about it, it may also mean that implementers are not thinking about it. It may not necessarily produce the best outcomes.

AJ:

Okay. So to that point, I've got another question. So, I've tweeted this at you. One of my pet peeves right now is that everybody is implementing some way to convert between basic binary and string format, so base64, x, UTF8, ASCII, ArrayBuffer, [inaudible] array, and they're all doing it in different ways. But this is a standard part of Java Script. We have UTF8 as a standard part of JavaScript. We have ArrayBuffers as a standard part of JavaScript. In cases like this, why don't we have the TC39 stepping in and saying, “Okay, boom. We're implementing it this way and everybody's going to do it this way. You can stop copying the same function 600 times.”

JAFAR:

Well, historically JavaScript has not come with batteries, right? And that may not be a good thing. But one of the nice things about JavaScript not coming with batteries is that JavaScript has made its way into a lot of places where I don't think we ever expected JavaScript to be. Now, does that mean we shouldn't have functionality for text encoding? Not necessarily. Maybe that [rises] to the level of something that belongs in the standard library.

But it's interesting. Everybody I think that is using JavaScript, and people are using it for a huge number of different things, building robots and running them on tiny watches. And it's running in so many different places, it's amazing. What seems like standard library to you might not seem like standard library to somebody else. Now again, I'm playing devil's advocate here. I probably agree with you here that we want some better support for text encoding enter the standard library. I'm just telling you historically, it's tended towards caution. And I think that's part of the reason why.

DAVE:

So, we talked a lot about TC39 and I really only have one question.

JAFAR:

Yeah.

DAVE:

And that is, what happened to the first 38 TC's?

[Laughter]

JOE:

Nice.

JAFAR:

I think they're in a room in Switzerland somewhere in the ECMA. I'm not sure. I don't know. I jumped in at 39.

DAVE:

So, what is TC and what is 39?

JAFAR:

Oh no. I was so not prepared for this question. I absolutely should be.

AIMEE:

[Chuckles]

DAVE:

[Laughs] Oh, we'll strike this from the record. Never mind.

JAFAR:

[Laughs] Yeah, you can edit this out, right? The god's honest truth is that I think TC stands for Technical Committee. I could be wrong about that. But that's what I think. I can tell you at a high level about ECMA. But it's interesting. ECMA doesn't really impact very much on… there's probably when we're in a meeting, we take care of ECMA business. But to some degree I think, ECMA is something that the committee needs, because we need to follow the standards process and there's processes that are in place. And so, we want to co-op those.

But at the same time, you can participate in the JavaScript committee and not necessarily have to spend most of your time worrying about how ECMA works. Certainly it becomes very important near the end when you're putting a bow on the spec. But I don't think it's terribly important, certainly for me it hasn't been very important for me. I'm able to engage in technical conversations and I don't think the specific way that ECMA works necessarily impacts very much on the average meeting.

DAVE:

So, what are the TC39 meetings like? Is it like the Microsoft engineers arm wrestling with the Google engineers?

JAFAR:

[Laughs]

DAVE:

In a smoke-filled room?

JAFAR:

You know, I think that… this is my first committee. And the impression I had of committees was not a positive one. When you hear about committees, it's rarely surrounded by the words, “Boy, what a great committee,” right?

CHUCK:

[Laughs]

JAFAR:

It's just not necessarily…

JAMISON:

This was designed really well by [inaudible] committee.

AIMEE:

[Laughs]

JAFAR:

By a committee.

DAVE:

[Laughs]

JAFAR:

And you know, here's the thing. I think it's great. It's actually, it's had an incredible effect on me as an engineer. It's actually really pushed me to level up, because you're working on that committee with some giants, some absolute giants. People like Brendan Eich and people like Mark Miller for example. And you've got people who have been in this industry for 20 years. They've really distinguished themselves. And you've got…

JOE:

People like Jafar Husain?

JAFAR:

Oh, not in the same breath. No, certainly not.

JOE:

[Laughs]

JAFAR:

And people like me who are implementers and really practitioners who are there I think to kind of make sure that we ground the committee, that they're thinking about use cases, right? And I think that overall, I think the committee operates phenomenally well. I think you look at the way the different committees work. There's a certain measure of dysfunction in any committee situation. But personally, when I look around at the way the other committees work out there on the web, I prefer the way that TC39 works, which is to say that it's a consensus-based model.

And so, it's not that you necessarily have one person that owns a feature and that is the feature champion and they make all the decisions. Consensus is very important in TC39. It's a more deliberative body in my judgment that some of the other web standards bodies. And I think that's particularly important for programming languages, because damn it, you don't make big mistakes hopefully in programming languages, the web's programming language. And so, I prefer a committee that's more deliberative.

And it's a room with a lot of wattage and a lot of brain power. Although sometimes the meetings can be acrimonious. I think sometimes there's going to be some conflict. In general, what I come out with consistently when I leave this meeting, I really get the sense that everybody there wants something great on the web. We all want great things. Sometimes we might disagree about how to get there. But it's a committee with very well-established processes and culture. What I mean is, it's got a great culture.

And so, you can rely on these cultural touchstones. It makes it much easier to take decisions. So overall, it's been a really great experience. I think the first time I went I was a little bit surprised. I think I came at a particularly contentious meeting. And you know, there was some shouting even. But when we left I think people spoke to each other. There was no animos-… because people know that they're there for a purpose. They want something better for the web. And I don't think it's ever taken, not often anyway, I don't think people take it personally where there's acrimony.

DAVE:

Should we talk about some of the cool new features coming out, then?

JAFAR:

Yeah, sure. I'd love to. So, as you guys… some of the ones I'm actually really excited about at the moment are decorators. One of the things the committee really focuses on, and this is something that's been so healthy, it's been retrospective, really healthy, is primitives. If somebody comes along with a big, fat cool feature for JavaScript the first thing I think people in the committee are going to push back and say, "Look, let's decompose this." What is an immutable type library? Is it maybe a set of structural types that you can implement immutable types on top of? That type of thing. The committee is always focused on primitives, on building layered, on building up in layers.

And so, what's the simplest, simplest feature we can get into JavaScript in order to enable people out there in user land, what we call user land, actual web developers, to go ahead and actually experiment and try to build features of their own? Because I think one of the healthy things about the JavaScript is that we're looking not necessarily to innovate necessarily. We're looking to find what people are really doing out there on the web. We're looking to enable them to build features themselves, build in libraries, and then try and standardize on that. And so, I think that's a much healthier process than standards bodies that are very activist and that are designing big standards and then throwing it over the wall and hoping that it works in practice.

So, I think decorators are an exciting primitive because it's amazing the wide variety of things that they're useful for. Sorry, for perspective guys, decorators are functions that you can apply to most often functions but also I think eventually it'll be applicable to classes as well. So, you could apply them to classes and properties and functions. And all they are really is a function. There's nothing especially interesting or amazing about a decorator. It's mostly a syntactical affordance.

So, let me give you a simple example. Let's say that I wanted to log every single time a method was called. Well, I could crack open that method definition and I could add a line in there that logged. Or I could add a decorator to the declaration of that function. And that decorator is basically just the name of a function somewhere. And when it's applied to a member on a class or a function directly, that function is simply passed to the decorator function. And the decorator function emits a function, usually with the same signature. And so, what happens is it just becomes a function that calls the original function but first logs. And so, on the outside of a decorator what you pass into the decorator very often looks exactly the same and has the exact same API. But what we've done is effectively mixed behavior inside of there.

So that's one example of what we could use decorators for, basically composing functions. And the syntax is beautiful. It looks very nice. It's an @ sign and then the name of the function right on top of the function declaration inside of the class. Another really killer apt use for it is for properties inside of classes. Right now if you want to use the ES5 style setting read-only or setting all these other property descriptors on properties and classes, it's very painful. You have to do it outside of a class structure. Whereas a decorator, because it's just a function that accepts the actual property descriptor object, you can create like a read-only decorator. And all it's going to do is it's going to accept the property descriptor for that member. And then it can actually flip the read-only bit and then return that property descriptor before it gets set on the class. Does that make sense at a high level?

DAVE:

Yeah, super cool. Freaking love decorators in other languages.

AJ:

So, I have yet another question here. So, is there some particular performance benefit to doing this native in the language as opposed to just having it be a pre-compiled process?

JAFAR:

Well so, decorators happen at run-time. There's no performance benefit. This is really purely about a syntactic affordance. It's to make things look frankly more declarative, right? It's very useful I think to be able to glance, once you see decorators in action, it becomes very clear because what you're doing is, it's really… you can use them in two different ways. I think most folks out there are familiar, some folks out there might be familiar with annotations in Java. And decorators are different than annotations.

Annotations are just, you're just attaching metadata to functions and properties and that type of thing, whereas decorators are actually more powerful than that. You can use them to attach metadata to things. When the decorator runs, it can add a property to a function for example. But it can also, as I said earlier, create a new function that mixes behavior into that function. And so, part of the reasons why the committee chose decorators over annotations is that they're powerful enough to express both those concepts of simply attaching metadata and then deriving behavior off of that later on, but also being able to actually just mix behavior directly into functions.

AJ:

Okay. So, I don't think it's a secret that I'm a bit of a skeptic of ES6. And we've talked about this a little bit before.

JAFAR:

[Laughs] Yeah.

AJ:

But I'm going to bring up the point again. So to me, this is my fear, is that there are so many new features that are coming in and people that want really cool, weird features are already using transpilers. They've been using transpilers for the past two or three years, right? They were using CoffeeScript before it got renamed to ES6. And they'll be using TypeScript well before it gets renamed to ES8 or 7 or whatever it is, right? So, why so much focus on these things that can be done by the people? Like, the people that want to use them, they already have the capability, as opposed to fixing… earlier you were talking about more tiny, minuscule things. I feel like so many of those are fixed, but yet we're expounding into more and more and more, which means more training and more use cases and more difficulty explaining things and more varying opinions on how to do something that would otherwise be relatively simple. So, I guess it's more of a philosophical question, but…

DAVE:

Well, in the case of decorators, they enable, you said it was to do something relatively simple. But they enable you to do really powerful things very elegantly and simply. I think you're basically asking, why not just have transpilers do all the new language features? Is that basically what you're asking?

AJ:

Until we fix what's broken, yeah.

CHUCK:

That's what I heard as well. But the thing is that… I just want to jump in here because I have a few opinions on this. One is that for new features, the features in ES2015, 2016, whatever, if the browsers implement it then it's going to be more performant. And a lot of these things, or at least some of these things can't directly be transpiled. The other thing is that if you look at any company's overhead, their largest expense is almost always the people that are there. And so, if you can put in features that enable those people to do more work then the companies get more value out of the work that's being done on their applications. Finally, the other thing is that the developer experience also gets better. So, all of the things that you're talking about that are broken in ES5, sure time could be spent to go back and try and figure out those things and fix them. But if we have forward momentum and progress, then a lot of those issues are going to be handled in new versions anyway.

AJ:

I'm just concerned that we're never… the community of developers that are using transpilers now will never, ever touch JavaScript again for the rest of their lives, that they will always be using transpilers because they're always going to end… the transpiler-ness will… I'm just worried that it's going to explode into this huge thing where there are a million different versions of JavaScript. I guess that's my paranoid concern.

CHUCK:

So, I have a…

AJ:

Hopefully that doesn't happen.

CHUCK:

I have a different take on this question and it goes back to the life-cycle in browsers. And that is if you release a version of ECMAScript or a spec for ECMAScript every year, yeah are we going to be stuck in a world where we always have to have transpilers in order to support older browsers?

AJ:

Well, [inaudible] new browsers, because they're not going to be able to do everything.

CHUCK:

Yeah, or new browsers. It's the same, right? Because the new browsers are going to have the new version. The old browsers will have the old version. Those browsers may or may not have compatibility issues with older versions of JavaScript. So, what does that mean?

AJ:

And some will implement parts of ES7 and ES8 before they've finished implementing ES6.

JAFAR:

Yeah. It's important to understand why they're doing that. Part of the reason they're doing or implementing features from the newer specification is because that's actually an extremely valuable thing to the committee. Part of the reason why Chrome implemented Object.observe was so that we can get real-word experience with how Object.observe works. And we couldn't really polyfill it. So, it's not so much that we're perpetually excited by… the browser vendors are perpetually excited by new things and whatever shiny they're going to go and implement. There's a real practical reason why they should implement some new features as they move through the maturity processes in JavaScript.

Now, when it comes to the question about transpilers, are we going to have transpilers forever? I hope so. Transpilers have been an incredibly valuable thing for the committee. Right now, when the committee proposes a feature, previously to transpilers, the only mechanism we had was to get browser vendors to go and implement it. And for a while Firefox did that. It implemented generators a million years ago. And it turns out that developers have better things to do with their time than go ahead and try and use the latest JavaScript feature that works in one particular browser one place. They couldn't use it in their job. And so, in practice we didn't really get any practical feedback on that feature. Transpilers changed everything. It's been incredibly transformative for the committee because now people can actually use this thing tomorrow. As soon as it makes its way into Babel, as soon as it makes its way into a transpiler, they can try it out and they can start giving us feedback.

Now, nobody's forced to use them. We're not holding a gun to a developer's head and saying, “Go off and use destructuring. Go off and use object destructuring. Go off and use async/await.” These are the intrepid developers that are turning on flags, that are taking an explicit step and saying, “Yes, I know this feature is only at stage two but I'd like to use it.” And they're frankly donating, they're donating a certain amount to the committee in the sense that they're taking on some risk to help us make that feature really great before it really rolls out to all the developers. And I think that's an incredibly powerful process.

I don't see transpilers as any different than any other piece of software, to be honest. Now that we have things like, if you look at the number of things that come in when I do npm install on some of the software that I'm working on…

[Exhale sound]

JAFAR:

There are a huge number of dependencies, a huge number of dependencies. But does that actually keep… has that been a hindrance to me? No, not at all. Now that source maps are working, I don't mind necessarily debugging code that's produced by a transpiler. It doesn't bother me. I don't think languages are special in the sense that they're just another piece of your software stack.

AJ:

Right. And I think you make a really good point. I totally agree. I think that transpilers are good. I think that they… you know, and I can definitely see the value that they add to being able to prototype features quickly.

CHUCK:

I want to get into some more of the features though, of upcoming versions. You mentioned Object.observe.

JAFAR:

Yeah. It's interesting. I think it's fair to say that momentum on Object.observe has stalled.

DAVE:

Yeah.

JAFAR:

We haven't seen it move very quickly through the standardization process. It started out of the gate really quickly. And I don't see it, I don't… these things are advanced by whoever is championing them. And it's my perception personally that this feature is not being championed. And my current belief is that I don't think everybody on the committee is convinced, I know I'm certainly not, that this is a good model for doing model-view synchronization.

AJ:

Rip Angular digest.

JAFAR:

[Laughs] I mean, well I think it's things like there have been some incredible developments in the last two years in UI development. The way that I think about building user interfaces today is completely different, top to bottom, than the way I'd been programming them for the last 10, 11, 12 years. We've really, things like, programs like React for example, and even people who have been building on top of the ideas of React. Like in the ClojureScript community things like Ohm and this idea of building immutable user interfaces. That's truly exciting. And it seems to have tremendous advantages over object observation, which has a lot of weaknesses.

I think a lot of people when they dropped Object.observe in their program didn't magically find that most of their model-view synchronization problems have been solved. If you look at just the complexity of using Object.observe, well I can listen to one property. That's great. But in practice, I have big, deep object hierarchies that I'm binding views to. What if I want to listen to a property path? How does that work? I think certain implementations out there like, I know Google was working on some stuff that tried to use Object.observe. They provided specific methods for listening to paths. But that wasn't really necessarily proposed for standardization.

And the other question of course is, what's the right granularity for listening to notifications? Do I want to call back as soon as a property changes? Do I want to call back when a bunch of property changes happen on an individual object? Well okay, that's what Object.observe chose. But it's very hard to choose the right granularity for when you get notified if something changes. Maybe I have a big fat object graph and I want to get notified once if a change happens anywhere in there.

And interestingly enough, this new approach with immutable types of modeling the state of your user interface in an immutable pipe and then effectively creating a whole new copy of that type whenever you change it, and then going back and diffing, can actually have pretty good performance characteristics, at least predictable performance characteristics. In a way that Object.observe doesn't have because you have this notification storm problem of, “Well, what's the right level of granularity for listening to these notifications?” It can be very, very costly to set up all of these event listeners.

And so, I think that's partly, I'm going out on a limb here I think again, but I think that's partly why support and momentum behind this has fizzled. It's because we're… it's just such a fast moving community out there, the JavaScript community. We're figuring out exciting new ways and innovative new ways to handle model-view synchronization. And I think that we've kind of moved beyond this model. In a lot of quarters in the JavaScript words, they've moved beyond that model.

CHUCK:

So, is that what async/await is?

JAFAR:

No, no.

JOE:

No, no.

CHUCK:

[Laughs]

JAFAR:

So, async/await, so I don't… immutable types are interesting because they were recently proposed for JavaScript. And to let everybody know out there what immutable types are, imagine every single time you pushed an item into your JavaScript array, instead of changing the array it created a whole new array. Now, I love telling developers about that, because usually the first response to that is to recoil. They're kind of like, "Well, wait a minute. How the heck is that going to perform? That's going to be really slow. It just can't work." We've been trained for many, many years now to think that mutation is the only way that we can do things quickly.

And from changing things, changing the model, I end up with this whole big problem in model-view synchronization, right? How do we make sure that whenever we change our model, the view gets synchronized? Well, there's a very, very different approach to that, which is this immutable types that we talked about. Imagine if every single time I "changed" an array, I created a new array. Well, in that case what I can do is, it turns out that we've basically figured out, I think even in just the last six or seven years, how to do this efficiently. So that we can actually share most of the memory from the previous collection in the new collection. So, that's one of the things, the big things that have changed. We figured out how to do this efficiently.

And the other big thing that's changed is that it turns out that if I had one array that was created from another array, I can actually figure out, I can put together a diff over that array very, very quickly.

CHUCK:

I just want to chime in here because the immutable stuff is, it's really interesting. I just got into Elixir which is a language that's written on the Erlang VM. And it depends on a lot of the things that you're talking about with lists. And a list is basically a head and a tail. And the way that it deals with all that stuff, and all the immutable stuff. And then you get to the point where since it's immutable, the shared state isn't as scary, and threading and all of that stuff. So there are all kinds of implications for going the direction you're talking about, which is exciting.

JAFAR:

Yeah, I want you to imagine for a moment. Do you guys remember the 90s? Were you doing web development in the 90s where every single time we clicked the link, we had to go all the way back to the server and then process a page and then send it down again? You know, the funny thing about…

AIMEE:

I don't remember.

JAMISON:

Chuck, Chuck [inaudible].

[Laughter]

JAFAR:

And don't you feel old now?

[Laughter]

JOE:

Those were great days.

CHUCK:

I don't have to feel old. I am old.

[Laughter]

JAFAR:

Well it's funny. In a sense they were terrible days for users and they were actually pretty good days for web developers. Because let's face it, that wasn't a very compelling experience. But you know what it was? It was simple, right?

CHUCK:

Mmhmm.

JOE:

[Inaudible] yeah.

JAFAR:

You [inaudible] your PHP template and you were pretty much done. You hit the database, you merged it with a view, and you sent it over a wire. As soon as single-page web applications came along, we got really great compelling user experiences compared to what we had before. We also got this whole new set of complexity, right? The nice thing though, recreating the page every single time somebody clicked something is you don't need to worry about state, right? You're basically almost immutable in a sense. You have one piece of state. Every single time somebody clicked the link, you basically got to run a function and just take the state of your web page, which you might be employing from cookies and database, and run it through this simple function and generate the UI all over again. It turns out, that's actually pretty simple. That's a nice way of thinking about building a program.

And what's interesting is that we figured out now 10, 12 years later, that we can kind of move back to the same model and still get all of the usability benefits of single-page web applications. Imagine if every single time we clicked a button in a web page, we ended up regenerating the entire user interface. Netflix is actually moving very much in this direction. We're building our new user interfaces very much based on this model where every single time we change something, it's as if you were literally re-rendering the entire page. Now again, developers find it really hard to wrap their head around this, because it seems like it would be really inefficient just like it was back in the 90s because you had to see the whole page got refreshed. Your scroll position got lost. It was a big fat blink and then it was really, it was both slow and terrible usability.

And so, the way we make it work, I want you to imagine that you're building a mobile app for a second. And one of the things you have to do with mobile apps is that you have to suspend and resume. Well, that's actually a surprisingly hard problem in a lot of user interfaces. You got to go all over the user interface, collect up all the relevant pieces of state, and put them into one JSON object that's probably, think of it as a recipe for rebuilding the UI when you come back again. And then you're going to turn around and you're going to stick that in local storage.

Well, imagine if every single time somebody clicked a button or did something, you went through that same process. You actually, instead of just spreading the state out all over your user interface, you just kept it in one big hierarchical JSON object. And every single time somebody clicked the button, you made a change to that JSON object so that you have enough information to be able to recreate the entire view at any given moment. And then you re-render the whole thing. You ran it through a function just like you did back in the 90s, you ran it through a template, and just re-rendered it.

What immutable types allow you to do is they allow you to do this efficiently. If you did it with a JSON object, it'd be pretty expensive. You'd have to recreate all those DOM objects. So, what React does and what some people can use Angular 2 to do as well, I assume people are using Angular 2 to do as well, is this process of instead of creating the real DOM objects, you can create this lightweight JavaScript representation of the DOM the very first time. And then you create the real DOM objects. And then as soon as somebody clicks a button, you go back and you change your big fat JSON object, your recipe for your entire app state. And then you call render again and create a whole new application state.

And what React does is it takes the old visual representation and the new visual representation, the lightweight basically copy of what the DOM should look like, and diffs them and creates a patch and applies only that patch to the user interface. Now, that is still actually kind of time consuming. Even if you're not really creating real DOM objects and you're just creating a lightweight representation of the DOM, you end up creating a lot of these objects just to render, because you're re-rendering from the top every single time. It kind of seems silly. If I'm typing six or seven characters into a text box, imagine re-rendering the whole view six or seven times just to get six or seven characters in a text box. What's deceptive about this is that we're so trained form the last 10, 15 years of UI development to think that's inefficient that we don't even think it's a possibility. And so, we ignore it even though it's definitely the simplest thing to do, right? It's much simpler to just keep running this function over state and reproducing the new UI again and again and again.

Now, what's changed is with these immutable types what we do is first we diff the model. It turns out that if you have two, using this array that's sharing 90% of the same memory as the old array, and that's how immutable types work, if I add an item to an array or something, it's called a vector sometimes, it creates a whole new vector that's a copy with that extra item. But they share most of the same [inaudible]. So, it turns out that when we diff them, we can very, very quickly figure out what changed. That's a very different approach than Object.observe. Object.observe says, “Well okay, I guess you could listen to all the indexes on the array. And whenever something changes, we'll tell you what indexes changed.” Here, we're actually going over the entire array and diffing and figuring out the pieces that changed. And then we only regenerate the views for those pieces of the model that actually changed.

And so, instead of recreating the view representation for the entire UI, first we prune whole branches off of the view and we only recreate the view up from the root right up into the spot where something actually changed, that little text box. And think about it. User interfaces really aren't that deep. There may be two, three levels, five, six, 10 at the most. So, you're really creating 10, in the end you're really creating 10 objects and re-rendering them. And so, that's even beyond what people are doing with React. In some communities, user interfaces are being programmed this way, as if they're fully immutable. You basically have only one piece of state which is the JSON object you might use when you re-hydrated your app from local storage. You just keep updating this JSON object and re-rendering.

DAVE:

So, is immutable objects coming in a future version of the language?

JAFAR:

Well, so that's the interesting thing. They were proposed for JavaScript. And I was actually a very big fan because of course we use this technique at Netflix. But the committee members pushed back. And they pushed back and said, “Wait a second. We're also talking about introducing structural types into JavaScript.” So, when I say structural types, I mean real memory-mapped types like Struct that you'd have in C for example. And that turns out to be a primitive feature on which you can build immutable types. And so, I think the committee did the right thing and pushed back and said, “Look, we're currently focused on building efficient structural types. Once those are in place, you can build an immutable type library on top of that.” And I think that's probably the right way to go.

DAVE:

Can you tell us more about structural types? I haven't heard of those before. I mean, I've used C's struct. Is that something similar?

JAFAR:

Very similar. And typed arrays, it's technically something already in JavaScript because of course typed arrays are a good example of a structural type. They're not the JavaScript capital A Array. They are arrays that are contiguous blocks of memory and that you can perform certain operations on at a very quick pace. And so, typed arrays are currently used for all sorts of operations. I think you can use SIMD on typed arrays. I'm pretty sure that you can even pass typed arrays back and forth between Web Workers by reference, which is a little bit racy. But it does work and it actually allows you to use Web Workers for a lot of things that really aren't practical normally because of the high cost of serializing things over the wire when you use Web Workers. Because normally, Web Workers only communicate by strings. And so, any benefit you get from parallelizing stuff, you tend to lose in that expensive serialization step. But that's one of the advantages of typed arrays.

JAMISON:

So, I have a radical subject change. I've heard a lot about symbols in ES 20… 27, ES27.

[Laughter]

JAMISON:

Symbols. And I'm not quite sure what they are or what you use them for. Can you talk a little bit about those?

JAFAR:

Yeah, symbols are actually, I wonder how much the average developers are going to use symbols. Symbols might be one of these things that's a primitive that your library uses. And I doubt that a lot of end developers are going to use symbols. But a simple description of what they are, imagine an untypeable string. [Chuckles] Imagine a string that no human person could type. That's what a symbol is. And it basically allows you to attach data to an arbitrary JSON object, an arbitrary JavaScript object in such a way that it's not going to conflict with anything else that anybody else added. Does that make sense?

JAMISON:

It kind of does. That kind of sounds like symbols in Ruby. Is that similar?

DAVE:

Yeah, that's what I was going to ask.

JAFAR:

I believe yes, they're basically similar in terms of what they are, yes. You can take a symbol and you can set it on normally of course JavaScript objects. All the keys are strings, right? And so, technically a symbol is like a string but it's just a string that nobody else can type. And so, although it's not like a real map where I can have an object reference and as long as you don't have the same object reference, we're not going to collide on that map, in this case it effectively is like the same thing. Because if I instantiate a symbol of my own, even if you instantiate a symbol with the same name, it's not going to conflict when we both set properties on a JavaScript object.

JAMISON:

Okay, that makes sense. So, probably I don't care is the real answer.

JAFAR:

I can tell you that [chuckles] most of the use cases of symbols I've run across are definitely at the library level and you probably don't care. If you're doing…

JAMISON:

That's a good enough answer.

JAFAR:

If you're building web apps.

JAMISON:

Sure, sure. So, I know you've done a lot of work with observables. Can you talk a little bit about that? And does that fit at all in any of the new features in JavaScript?

JAFAR:

Well, so observable is currently proposed for ES2016. And it's currently moving through the standardization process. I think we're on stage one now. And of course, that's the second stage because of course we're base zero. And what an observable is, is think of it as a stream of information that's being pushed at you. And when I say pushed, what I mean is you hand this data source a callback and it keeps invoking that callback. So, if you're familiar with DOM events, if you're familiar with WebSockets, if you're familiar with streams, these are examples of push data sources where instead of you requesting the data and then blocking until you receive that data, you hand a callback to the data source and the data source pushed the data to you by invoking your callback when it arrives.

JAMISON:

This is… there are lots of libraries that do this already. So, is the difference… what's the difference of building it into JavaScript?

JAFAR:

Well, so what's really interesting about observable is that a lot of people are familiar with iterable. Or I'm just curious, because I know now everybody has had a chance to fully digest some of those ES6 features yet. But one of the interesting ones, if you've seen the for…of loop for example, is that it's based on this contract called iterable. And it's a really old concept in computer science.

The idea is you're a producer of data and I'm a consumer. And I want to get the items in this collection that you're producing one item at a time. And so, I just call next and you give me an item, and I call next and you give me an item, until I finally call next and you say, “I have no more data.” That's how an iterator works. And under the hood if you look at what for…of generates, it turns into a while loop where we just keep calling next until next basically indicates that the data source is dried up. There's nothing left. So, that's a really well-established concept in computer science. And it's surprising that JavaScript didn't start out with it. But it was developed in eight days, so you got to give Brendan Eich a little bit of credit.

But the interesting here with observable is it's kind of the same thing in reverse. So, instead of the consumer being in control and saying, “Hey producer, give me a new item,” and then blocking until they get that item, instead the consumer hands a callback to the producer. And the producer just keeps invoking that callback whenever it has an item to deliver. So, the producer is in control. So, there's actually an equivalence between iterable and observable. Now okay, why does that really matter to developers, right? Well, the reason why it matters is that anything that you can do with an iterable, and there's a whole series of operations. And it's usually the array operations that most developers are familiar with, map, filter, reduce, those types. Operations can all be implemented over iterable.

And if you can show an equivalence between iterable and observable as has been shown in other languages, it means that you can implement all those same operations, map, filter, reduce, over observable, and by extension anything that you can express as an observable. In other words, WebSockets. Imagine being able to program these streams of data just like you program arrays with map and filter and reduce. And it turns out that you can do that. It turns out that it really doesn't matter if data's being pushed at you or you're pulling data. It's like writing a SQL statement over data that hasn't arrived yet. Now, that's how I would best express it at a high level to somebody who hasn't seen this type of coding before.

It's really revelatory when you see it, because you realize we've been programming events and things like arrays or data from a file differently for the last 15, 20 years. And you can actually program in much the same way. And that's what's exciting about observable.

AIMEE:

I was also going to ask. We touched on async/await for a brief second. And it's one of the most popular topics out there right now. So, can you go over that briefly?

JAFAR:

Yeah, absolutely. So, async/await I think first cropped up although it's been in a lot of languages it looks, certainly cropped up in C# in the form that's most recognizable as its incarnation today in JavaScript. The way JavaScript's async/await works is that now that in ES6 we have promises which are basically an object that represents the result of an asynchronous operation, now that we have an object that we can hold onto as the result of an asynchronous operation, well now we can start to build features into the language that really take advantage of the fact that we know promises are there. And async/await is one of such features.

So, I can put the keyword async in front of a function. And with just that little modifier, any time that I want to pause until a promise has been resolved, and then continue on, instead of having to use callbacks and calling then on the promise, I can just put the await keyword in front of the promise inside of that async function. And under the hood, JavaScript will effectively stop control flow until that promise is resolved and then will resume. And what it means is that to the average developer, that you get to write code as if it's synchronous even though it might be asynchronous.

DAVE:

That's so cool.

AIMEE:

Do you have to wrap that in try/catch/correct? Which is the complaint I hear from a lot of people, is that they don't like using that.

JAFAR:

When you invoke an async function you have to wrap it in try/catch?

AIMEE:

Well, I'd have to go back and check the examples. I'm not positive.

JAFAR:

In all but a few use cases, that shouldn't be necessary. The goal here behind an async function is that it should never ever throw and it should never ever synchronously produce a result. And the only thing an async function should do is hand you a promise immediately, synchronously. And then when you then that promise, eventually it will give you the result.

Now there is a weird edge case unfortunately where I think personally the committee made a mistake. And that is there's a case where async functions can throw. And that is default parameters. So, if you have an expression inside your default parameter which just happens to throw, unfortunately it will throw synchronously. And I think in retrospect that was probably a mistake. But unfortunately I think we're probably going to be stuck with it because async functions are built on top of generators.

Under the hood, they're actually, as at least planned today, they're light syntactic sugar over generators. And so, that's one of the reasons why the feature's such a slam dunk. It's very, very easy to spec. It's an obvious win for developers. But unfortunately we didn't think about that case I think when we put in place generators, which are lazy. If you're familiar with generators they're lazy and they're not supposed to do anything until you call next the very first time. And that probably should have also applied to default parameters on the function. And so, I think async/await is going to get a cold from the mistake we made in generators.

DAVE:

Dude, you guys are so cool.

[Laughter]

DAVE:

TC39.

[Laughter]

DAVE:

It's no wonder you guys killed all those other TC's.

[Laughter]

JAFAR:

We'll never know what happened to them.

DAVE:

But seriously, is that harder and harder the more features you add to the language? In this case you had a collision between default parameters, new language feature, and async/await, new language feature. Do you have this n squared problem where for every new feature you have n features you have to go back and review for edge cases like that?

JAFAR:

Absolutely. And I can tell you personally for me, that's a lot of soul-searching. Because my bias is towards languages that are very small, that have a very small surface area. That's my own personal aesthetic, sense. One of my favorite programming languages is Scheme which up until the latest version was a very, very small spec document. And JavaScript is not small. JavaScript is a big language and it's getting bigger and bigger. And despite my personal feelings, or I wouldn't say feelings. I would say despite my initial impression, which is that JavaScript… and I'll tell you. I'll be very candid. I think JavaScript, my feeling about ES6 was that it was too big. There were too many features in there.

And then I actually got into the committee and I heard the rationales for each of these features. And you know what? For the most part, they were pretty bulletproof. There was really good reasons why putting those features in the platform was going to make real web developer's lives easier. And so, it's very hard to argue against that. JavaScript is, I like to call it a big tent language. It effectively has to support a large audience. And these languages inevitably grow in size, workhorse general purpose languages like JavaScript.

And I'll give you some examples. C# I felt the exact same way. C# was one of my programming languages that I spent a lot of time in, in my career. And over time it's grown and grown and grown. And sure enough, around the edges, it's very difficult to get all the interactions between the features right. I think they made a few small mistakes in C# which overall is still a very good language. But I think you've really got to say, “Look, any time you want to make a language useful, to some degree it's going to become a little bit of a kitchen sink language.” It can't be too ideological because we've got to be pragmatic, because we've just got so many different use cases to serve. And so, I've learned to love it I've learned to love and think that it's probably the right thing to add these features. It just means we've got to work harder to get them right.

DAVE:

Okay. I'd like to do something a little different. Let's call this a rapid fire question round where I would like to say a list of language features. And you as our resident TC39 expert will tell us what version of the language these features are available or will become available in.

JAFAR:

Okay.

DAVE:

Can we do that really quick?

JAFAR:

Alright.

DAVE:

Okay. Oh, and for bonus points, tell us which language you guys ripped them off from.

JAFAR:

[Laughs] Go ahead.

DAVE:

Okay.

JOE:

Awesome.

DAVE:

Let.

JAFAR:

ES2015. For now I'm going to abbreviate it to 15 and 16.

DAVE:

Okay.

JAFAR:

And 15 is what is currently called ES6.

DAVE:

Okay. Okay, for…of.

JAFAR:

For…of gives you a generic way of iterating over a variety of different data sources. It's available in 15.

DAVE:

Modules.

JAFAR:

Modules is a great way of bringing dependencies into a JavaScript program and being able to break your libraries up into smaller bits. Unlike things like CommonJS, it supports static analysis. There's a special syntax for modules. It's going to be available in 15.

DAVE:

Destructuring.

JAFAR:

So, that's one of my favorite features in JavaScript. One of the… can you imagine how hard it would be to write JSON if you didn't have JSON? How hard would it be to create JavaScript objects with a bunch of nested properties if all you could do was use sets, property sets? Well, the funny thing is that we've been in the exact same situation for taking data out of JSON. And we have to write line by line of code of declaring a variable, assigning it to some deep property inside of an object. Well, JSON's a good example of structuring where you have a special syntax for building JavaScript objects. Well, you can use that same syntax to take the data out of JavaScript objects.

And it's not just about writing a few smaller lines of code. What's so great about destructuring is it allows, it's the same thing that's great about structuring. When you're writing a program as a series of data transformations, structuring and destructuring provide a visual counterpoint that allows you to see the data literally flow through visually your data structures. Because you can see the JSON structure which is a visual structure that you can make out. And then within a function when you're pulling the data out, that uses the same JSON structure. So, you can see the data flow from this property to the next property, to this collection and throughout your program. And so, it becomes a visual programming medium instead of a list of instructions. And that's coming in 15.

DAVE:

Promises.

JAFAR:

Promises are coming in 15. Promises are an object that represents the eventual result of an asynchronous computation. What's exciting about promises is that unlike callbacks, promises when composed together automatically forward up asynchronous errors. So, just the type of behavior that you've come to expect from try/catch that you take for granted, you're now going to be able to take for granted in the asynchronous space. If an error happens, you don't have to catch it and forward it up every single time. You can count on being able to catch it anywhere in the chain and have promises forward those errors asynchronously up for you.

AJ:

And promises are available everywhere now, right? There's no browser that doesn't have them, or Node has them and everything has it, right?

JAFAR:

I'm not actually sure about the browser status. I think so. I'm not sure about Microsoft Edge. I don't mean to single them out. I'm not sure if they have it. But I can tell you…

AJ:

It's okay. They're used to it.

JAFAR:

[Laughs] I can tell you it's eminently polyfillable, promises. Even if you don't have them, there are lots of great libraries out there.

AJ:

Yes, there are.

DAVE:

Okay.

AJ:

And some of them are even better than the real ones.

[Chuckles]

DAVE:

Okay. Default function argument expressions.

JAFAR:

That's ES2015. And that's pretty straightforward. You can put parameter declarations inside of a function. You can use an equals and you can put in any kind of JavaScript expression there in the event that that parameter is omitted when the function is called. The parameter will be default to the value of that expression. And one of the things I really like about this over and above the same feature implemented in other languages is because JavaScript's a dynamic language. You can put pretty much any expression you want to in there. And so, that's much more flexible than some of the static languages where you can only put one of say, several static values. I'm not naming any names. C#.

DAVE:

And finally, async/await.

JAFAR:

Async/await is ES2016. As I mentioned before, it basically allows you to write your code as though it's synchronous but allow asynchronous control flow to actually happen. And it's based on top of promises. Think of it as syntactic sugar for the promises introduced in ES2015.

DAVE:

Ladies and gentlemen, I think that was the most beautiful explanation in rapid succession of the new language features I have ever heard. Bravo.

JAFAR:

[Laughs]

DAVE:

[Claps hands] Bravo.

JAFAR:

Thank you.

AJ:

I agree.

CHUCK:

I think this is also a good place to wrap up and go to picks. AJ, do you want to start us with picks?

AJ:

Oh, yes. Okay. So, first of all, I'm going to pick another podcast's episode of this episode. [Chuckles] This one was probably a little bit more information. [Inaudible] Jafar and I already [inaudible] a lot more of the other one.

[Laughter]

AJ:

It was good though. So, there's The Web Platform Podcast has an episode on ES6 and ES7 if you want to listen to that.

Also this guy here, he's got some really good videos on YouTube. If you take a look at his one on async JavaScript or on Falcor, those are both very interesting. And I enjoyed those.

And then Let's Encrypt which I've mentioned many, many times. They have their Python client… I hesitate to use the word complete. But there's… you can run it and you can get certificates. They're not valid from the real certificate authority yet. Those certificates arrive on July 24th. And then they're going to be doing security audits and stuff like that. And it'll go live-live on September 1st maybe. So, if you want to start playing with that, I got a little guide on how to do that on a Raspberry Pi which would work just as well as on an Ubuntu or whatever.

And I also discovered HAProxy, which you can do really cool things with, including multiplexing port 443 with SSH and OpenVPN and Let's Encrypt and your web server and whatever you want so that you can do some really fun things, like open up a VPN in a library where they're doing layer 7 firewall filtering. But they can't catch you. So, I'd totally say check out HAProxy if you want to do any weird stuff where you want to multiplex a port or have high availability servers or something like that, what it's meant for.

And I've got one more pick. So, this is actually, if you've listened to the show for a while, you've heard of Mandy maybe, and Mandy's fiancé. Now, we don't know much about them other than that Mandy's the best post podcast production editor. And that Mandy's fiancé loves video games. So, I started this Indiegogo campaign because Mandy had said something about how he was on video game hiatus. And they had just moved and had to cut back on expenditures. It turns out that he had some surgery. So, he really doesn't have much to do right now anyway. So, if you're one of our regular listeners and you want to do a little something nice for our podcast producer, go ahead to that Indiegogo campaign and put a little fiver in the pot. Consider it like a tip for all the times you didn't pay us for the hard work we do here. And that's it. That's what I've got.

CHUCK:

Alright. Dave.

DAVE:

Hey, a couple of picks for you today. I've been waxing a little academic and I read this really excellent paper called 'The Majority Illusion in Social Networks'. And the paper is about, it's a mathematical analysis of social networks and how sometimes we have these ideas that turn out to be paradoxical. Like for example, your friends probably have more friends than you do, mathematically provable and true. And the other thing is the main crux of the paper is that it only takes a couple of bad actors in a social network to give the impression that the majority of people in your circle are bad actors. And so, the example they gave is that if you have a population of 14 people, it only takes three. And then certain types of connections for you to now have this illusion that the majority of your connections are bad actors. It was very, very interesting. I'll link it in the show notes.

My second pick is another interesting effect called the Gell-Mann Amnesia Effect. This is coined by a person whose last name is Gell-Mann. So, it's not actually a man made of gel. And he notes how when you are an expert in an area and you hear the news media report on that area, and you find that the new media is actually quite wrong in their reporting, but then you hear the news media report on an area in which you are not an expert and you believe them, this is called the Gell-Mann Amnesia Effect. And anyway, it happens to me a lot every time I hear mainstream news report on something related to computering. And I realize that they have no idea what they're talking about. But then when they report on pretty much everything else, I'm like, “Oh my gosh. All these things are happening.” But you know, that's the amnesia effect. So, I'll link a paper about that as well in the show notes. Those are my picks.

CHUCK:

You realize that all experts have that effect with the news media, right? They're always wrong. It's just that you don't know it for the stuff that you're not an expert in.

DAVE:

That is the Gell-Mann Amnesia effect in a nutshell.

CHUCK:

Alright. Aimee, do you have some picks for us?

AIMEE:

Yep, I have two. So, the first one, I was going to plug one of Jafar's things. It's a course on Egghead. I initially watched a talk that he did with this thing called NomadJS which is an online meetup group. And then I wanted to dig in a little bit more so I did this Egghead course. It's just a short course. I think the first couple is free. You might have to pay for the last couple. But it's called 'End of the Loop'. And it goes into a little bit of what we talked about here with observables. So, I would recommend people check that out. It was fun.

And then my other one is an article called 'You Really Can Work Smarter, Not Harder'. And it's something that I, as I've taught myself programming and I've seen what other developers do, the good ones anyways, as you're learning it talks about reflecting on what you've learned, not just learning. So, ask yourself questions about what you just did or write down what you just did. Or just like, “What did I just learn from what I just did?” So, the article talks about that and studies they did, people's test scores who did this and didn't. And I thought it was pretty interesting. And that is it for me.

CHUCK:

Alright. Jamison, do you have some picks for us?

JAMISON:

I do. I have three. My first pick is the Elm programming language. It's like Haskell but you can get stuff done in it without being an expert in Haskell.

[Chuckles]

JAMISON:

It's strongly typed, purely functional, and it has some really cool ideas around reactive programming and also around using immutable data to build cool UIs, kind of similar to some of the stuff that Jafar was talking about.

My second pick is a YouTube show called The Katering Show. It's two women who have a fake food blog. And it's hilarious. I won't ruin the jokes by trying to describe them. But it's really funny.

And my last pick is a tweet that taught me that the term sharding that you use to distribute data among many copies of a database came apparently from Ultima Online, the game from the 90s. So, it's kind of a cool…

DAVE:

Oh.

JAMISON:

Computing history lesson. Those are my picks.

CHUCK:

Alright, Joe what are your picks?

JOE:

My first pick is something I'm disappointed that nobody else has already picked. You guys are not… you're all disappointing me. My first pick is the US Women's National Team, soccer team.

CHUCK:

Five to two, baby.

JOE:

[Laughs] Sunday was the finals games for the Women's World Cup in soccer, or football as the rest of the world calls it. And the US women dominated the competition scoring four goals in 16 minutes which is a record amongst all world cups ever, regardless of gender. Just to prove that our women are better than the rest of the world's men.

[Laughter]

CHUCK:

I'm telling you, I was screaming for 16 minutes straight.

JOE:

It was an amazing game to watch. Total combined score, five to two. So, seven aggregate goals. I kind of wish that every soccer game had seven total goals.

[Chuckles]

JOE:

But I just want to pick the US Women's National Soccer Team because they are awesome. And it was a great game to watch.

And one other pick. I wanted to pick this last week. A little tip that I heard is whenever you want to do a search on something JavaScript or browser related, HTML related, and of course you want to limit your search to MDN, the Mozilla Developer Network where all the great repository of information about these topics is and not any other websites that may or may not have useful and probably don't have useful information, you can…

DAVE:

[Fake coughs] W3Schools.

[Laughter]

JOE:

Exactly. So, if you know that it's something you don't want a blog article about, you actually want a real source, a valid source of, say you want to remember what the syntax is for the base tag in HTML. You can't remember. Is it href or the source attribute that I can remember? So, you got MDN.io. And if you just go to MDN.io/base then it will take you right to the article on the base tag at MDN. And if you can't remember and you're just going to do a phrase rather than just googling the phrase, do MDN.io/<then the phrase> and then it will basically run the same search for you but only on MDN. So, a nice little trick. And I've already used it and it's awesome. And everybody should. So, that's my second and final pick.

CHUCK:

Alright. I've got a couple of picks. The first one is a Bluez. It's called Bluez 2 Aftershokz. They are an over the ear bone conduction headphones. And so, they have the little band that goes behind your head. And then it hooks over your ear. And then the little pads rest on the sides of your face. And anyway, they're awesome. I listen to them pretty much when I'm doing anything except for mowing the lawn. And that's just because my lawnmower is too loud. I have to actually put something in my ears in order to hear it there. But I think they're awesome. I love them to death. They're about $100 so they're not cheap. But they rock. And I had somebody ask me if they bounce when I run and the answer is no. So, they stay right where I put them and they sound great. And I can still hear what's going on around me. So, if I'm running and somebody needs to pass me or anything like that, I'm driving in the car with my kids, I can have them in and I can still hear what's going on.

I was also going to pick the Women's National Team. But I think Joe pretty well covered that. It was such a fun game to watch.

And one other thing is I am opening up some spots for coaching and corporate training. So, keep an eye on my Twitter account for that. I'm @cmaxw on Twitter. And Jafar, what are your picks?

JAFAR:

I think it's funny. I'm actually giving a talk to a bunch of high school students at Netflix tomorrow who are in engineering. They're in an engineering program. So, I'm thinking about how to explain to them how much fun software engineering is as a discipline. And my mind immediately went to the book 'What Motivates Us', 'Autonomy, Mastery, and Purpose'. It's a book. I think it's actually called 'Drive, Autonomy, Master, and Purpose'. And it's basically talking about why employees do what they do, what basically goes into motivating employees and having them do well. And basically, you can tie each one of those things to the software industry. And so, I think, and in fact actually the book talks extensively about the software industry and why we see things like open source software which seems to contradict basic ideas about economics. Why are people spending all their time working and building such great software and then just giving it away? And I think it's really important, particularly for a manager, to go and read that book.

And I think it's one of the things actually, another pick of mine is Netflix. Because I think Netflix does that really, really well. I was just thinking about that as I was going through that book, all the things that I think Netflix does to make that, basically to motivate me. I'm not trying to plug Netflix as an employer, although they are a great employer. But I think software companies and any company can learn a lot from that book.

Let's see, what else? Quiescent which is a new ClojureScript framework that's built on top of React. I know we've got a lot of Angular fans out there too and I'm super excited about Angular 2. But this is a good explanation, a good example of how this kind of paradigm shift about thinking about user interfaces statelessly, it's so new that we're still figuring out ways of doing it. We're just iterating on top of this basic idea of just rebuilding the UI every single time. And Quiescent actually has a twist on at least what the first few frameworks did. And so, it's definitely something worth checking out.

Quiescent is a framework out there for ClojureScript which I'll also pick. ClojureScript is a great language that compiles into JavaScript. It's a very, very well designed language. And if you're looking around for something pretty darn different but compiles into JavaScript, ClojureScript has got a phenomenal stack. And so, I would definitely recommend doing that. And especially if you've never done a Lisp before, which ClojureScript is. It's a Lisp. So, it's always I think a very important thing in a software developer's career to at least learn Lisp once because it really challenges the way in which you think about programming. Those are my picks.

CHUCK:

Alright. Well, thank you all for coming and sharing your opinions. And thank you Jafar for being such a great expert on this stuff.

JAFAR:

Thanks so much for having me guys. I really appreciate it. It was a lot of fun.

CHUCK:

Alright, well we'll wrap up the show. We'll catch you all later.

JOE:

Thanks, everybody.

JAFAR:

Bye-bye.

[Once again, this episode was sponsored by Braintree. So, go check them out at Braintree.com. If you need any kind of credit card processing or payment processing in general, they are a great way to go. And we appreciate them sponsoring the show.]

[This episode is sponsored by MadGlory. You've been building software for a long time and sometimes it's get a little overwhelming. Work piles up, hiring sucks, and it's hard to get projects out the door. Check out MadGlory. They're a small shop with experience in shipping big products. They're smart, dedicated, will augment your team and work as hard as you do. Find them online at MadGlory.com or on Twitter @MadGlory.]

[Hosting and bandwidth provided by the Blue Box Group. Check them out at BlueBox.net.]

[Bandwidth for this segment is provided by CacheFly, the world’s fastest CDN. Deliver your content fast with CacheFly. Visit CacheFly.com to learn more.]

[Do you wish you could be part of the discussion on JavaScript Jabber? Do you have a burning question for one of our guests? Now you can join the action at our membership forum. You can sign up at JavaScriptJabber.com/jabber and there you can join discussions with the regular panelists and our guests.]