JS Questions

Explain event delegation

Event delegation is a technique involving adding event listeners to a parent element instead of adding them to the descendant elements. The listener will fire whenever the event is triggered on the descendant elements due to event bubbling up the DOM. The benefits of this technique are:

- Memory footprint goes down because only one single handler is needed on the parent element, rather than having to attach event handlers on each descendant.
- There is no need to unbind the handler from elements that are removed and to bind the event for new elements.

References

- https://davidwalsh.name/event-delegate
- https://stackoverflow.com/questions/1687296/what-is-dom-event-delegation

[↑] Back to top

Explain how `this` works in JavaScript

There's no simple explanation for `this`; it is one of the most confusing concepts in JavaScript. A hand-wavey explanation is that the value of `this` depends on how the function is called. I have read many explanations on `this` online, and I found Arnav Aggrawal's explanation to be the clearest. The following rules are applied:

1. If the `new` keyword is used when calling the function, `this` inside the function is a brand new object.
2. If `apply`, `call`, or `bind` are used to call/create a function, `this` inside the function is the object that is passed in as the argument.
3. If a function is called as a method, such as `obj.method()` — `this` is the object that the function is a property of.
4. If a function is invoked as a free function invocation, meaning it was invoked without any of the conditions present above, `this` is the global object. In a browser, it is the `window` object. If in strict mode (`'use strict'`), `this` will be `undefined` instead of the global object.
5. If multiple of the above rules apply, the rule that is higher wins and will set the `this` value.
6. If the function is an ES2015 arrow function, it ignores all the rules above and receives the `this` value of its surrounding scope at the time it is created.

For an in-depth explanation, do check out his article on Medium.

Can you give an example of one of the ways that working with this has changed in ES6?

ES6 allows you to use arrow functions which uses the enclosing lexical scope. This is usually convenient, but does prevent the caller from controlling context via `.call` or `.apply`—the consequences being that a library such as `jQuery` will not properly bind `this` in your event handler functions. Thus, it's important to keep this in mind when refactoring large legacy applications.

References

- https://codeburst.io/the-simple-rules-to-this-in-javascript-35d97f31bde3
- https://stackoverflow.com/a/3127440/1751946

[↑] Back to top

Explain how prototypal inheritance works

This is an extremely common JavaScript interview question. All JavaScript objects have a `prototype` property, that is a reference to another object. When a property is accessed on an object and if the property is not found on that object, the JavaScript engine looks at the object's `prototype`, and the `prototype`'s `prototype` and so on, until it finds the property defined on one of the `prototype`s or until it reaches the end of the prototype chain. This behavior simulates classical inheritance, but it is really more of delegation than inheritance.

Example of Prototypal Inheritance

We already have a build-in `Object.create`, but if you were to provide a polyfill for it, that might look like:

```
if (typeof Object.create !== 'function') {
  Object.create = function (parent) {
    function Tmp() {}
    Tmp.prototype = parent;
    return new Tmp();
  };
}

const Parent = function() {
  this.name = "Parent";
}

Parent.prototype.greet = function() { console.log("hello from Parent"); }

const child = Object.create(Parent.prototype);

child.cry = function() {
  console.log("waaaaaahhhh!");
}

child.cry();
// Outputs: waaaaaahhhh!
```

```
child.greet();
// Outputs: hello from Parent
```

Things to note are:

- `.greet` is not defined on the *child*, so the engine goes up the prototype chain and finds `.greet` off the inherited from *Parent*.
- We need to call `Object.create` in one of following ways for the prototype methods to be inherited:
    - Object.create(Parent.prototype);
    - Object.create(new Parent(null));
    - Object.create(objLiteral);
    - Currently, `child.constructor` is pointing to the `Parent`:

```
child.constructor
ƒ () {
  this.name = "Parent";
}
child.constructor.name
"Parent"
```

- If we'd like to correct this, one option would be to do:

```
function Child() {
  Parent.call(this);
  this.name = 'child';
}

Child.prototype = Parent.prototype;
Child.prototype.constructor = Child;

const c = new Child();

c.cry();
// Outputs: waaaaaahhhh!

c.greet();
// Outputs: hello from Parent

c.constructor.name;
// Outputs: "Child"
```

References

- https://www.quora.com/What-is-prototypal-inheritance/answer/Kyle-Simpson
- https://davidwalsh.name/javascript-objects

- https://crockford.com/javascript/prototypal.html
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

[↑] Back to top

What do you think of AMD vs CommonJS?

Both are ways to implement a module system, which was not natively present in JavaScript until ES2015 came along. CommonJS is synchronous while AMD (Asynchronous Module Definition) is obviously asynchronous. CommonJS is designed with server-side development in mind while AMD, with its support for asynchronous loading of modules, is more intended for browsers.

I find AMD syntax to be quite verbose and CommonJS is closer to the style you would write import statements in other languages. Most of the time, I find AMD unnecessary, because if you served all your JavaScript into one concatenated bundle file, you wouldn't benefit from the async loading properties. Also, CommonJS syntax is closer to Node style of writing modules and there is less context-switching overhead when switching between client side and server side JavaScript development.

I'm glad that with ES2015 modules, that has support for both synchronous and asynchronous loading, we can finally just stick to one approach. Although it hasn't been fully rolled out in browsers and in Node, we can always use transpilers to convert our code.

References

- https://auth0.com/blog/javascript-module-systems-showdown/
- https://stackoverflow.com/questions/16521471/relation-between-commonjs-amd-and-requirejs

[↑] Back to top

Explain why the following doesn't work as an IIFE: `function foo(){ }();`. What needs to be changed to properly make it an IIFE?

IIFE stands for Immediately Invoked Function Expressions. The JavaScript parser reads `function foo(){ }();` as `function foo(){ }` and `();`, where the former is a *function declaration* and the latter (a pair of parentheses) is an attempt at calling a function but there is no name specified, hence it throws `Uncaught SyntaxError: Unexpected token )`.

Here are two ways to fix it that involves adding more parentheses: `(function foo(){ })()` and `(function foo(){ }())`. Statements that begin with `function` are considered to be *function declarations*; by wrapping this function within `()`, it becomes a *function expression* which can then be executed with the subsequent `()`. These functions are not exposed in the global scope and you can even omit its name if you do not need to reference itself within the body.

You might also use `void` operator: `void function foo(){ }();`. Unfortunately, there is one issue with such approach. The evaluation of given expression is always `undefined`, so if your IIFE function returns anything, you can't use it. An example:

```
// Don't add JS syntax to this code block to prevent Prettier from formatting
it.
const foo = void function bar() { return 'foo'; }();

console.log(foo); // undefined
```

References

- http://lucybain.com/blog/2014/immediately-invoked-function-expression/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void

[↑] Back to top

What's the difference between a variable that is: `null`, `undefined` or undeclared? How would you go about checking for any of these states?

Undeclared variables are created when you assign a value to an identifier that is not previously created using `var`, `let` or `const`. Undeclared variables will be defined globally, outside of the current scope. In strict mode, a `ReferenceError` will be thrown when you try to assign to an undeclared variable. Undeclared variables are bad just like how global variables are bad. Avoid them at all cost! To check for them, wrap its usage in a `try` / `catch` block.

```
function foo() {
  x = 1; // Throws a ReferenceError in strict mode
}

foo();
console.log(x); // 1
```

A variable that is `undefined` is a variable that has been declared, but not assigned a value. It is of type `undefined`. If a function does not return any value as the result of executing it is assigned to a variable, the variable also has the value of `undefined`. To check for it, compare using the strict equality (`===`) operator or `typeof` which will give the `'undefined'` string. Note that you should not be using the abstract equality operator to check, as it will also return `true` if the value is `null`.

```
var foo;
console.log(foo); // undefined
console.log(foo === undefined); // true
console.log(typeof foo === 'undefined'); // true

console.log(foo == null); // true. Wrong, don't use this to check!

function bar() {}
var baz = bar();
console.log(baz); // undefined
```

A variable that is `null` will have been explicitly assigned to the `null` value. It represents no value and is different from `undefined` in the sense that it has been explicitly assigned. To check for `null`, simply compare using the strict equality operator. Note that like the above, you should not be using the abstract equality operator (`==`) to check, as it will also return `true` if the value is `undefined`.

```
var foo = null;
console.log(foo === null); // true
console.log(typeof foo === 'object'); // true

console.log(foo == undefined); // true. Wrong, don't use this to check!
```

As a personal habit, I never leave my variables undeclared or unassigned. I will explicitly assign `null` to them after declaring if I don't intend to use it yet. If you use a linter in your workflow, it will usually also be able to check that you are not referencing undeclared variables.

References

- https://stackoverflow.com/questions/15985875/effect-of-declared-and-undeclared-variables
- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/undefined

[↑] Back to top

What is a closure, and how/why would you use one?

A closure is the combination of a function and the lexical environment within which that function was declared. The word "lexical" refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Closures are functions that have access to the outer (enclosing) function's variables—scope chain even after the outer function has returned.

Why would you use one?

- Data privacy / emulating private methods with closures. Commonly used in the module pattern.
- Partial applications or currying.

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures
- https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closure-b2f0d2152b36

[↑] Back to top

Can you describe the main difference between a `.forEach` loop and a `.map()` loop and why you would pick one versus the other?

To understand the differences between the two, let's look at what each function does.

`forEach`

- Iterates through the elements in an array.
- Executes a callback for each element.
- Does not return a value.

```
const a = [1, 2, 3];
const doubled = a.forEach((num, index) => {
  // Do something with num and/or index.
});

// doubled = undefined
```

`map`

- Iterates through the elements in an array.
- "Maps" each element to a new element by calling the function on each element, creating a new array as a result.

```
const a = [1, 2, 3];
const doubled = a.map(num => {
  return num * 2;
});

// doubled = [2, 4, 6]
```

The main difference between `.forEach` and `.map()` is that `.map()` returns a new array. If you need the result, but do not wish to mutate the original array, `.map()` is the clear choice. If you simply need to iterate over an array, `forEach` is a fine choice.

References

- https://codeburst.io/javascript-map-vs-foreach-f38111822c0f

[↑] Back to top

What's a typical use case for anonymous functions?

They can be used in IIFEs to encapsulate some code within a local scope so that variables declared in it do not leak to the global scope.

```
(function() {
  // Some code here.
})();
```

As a callback that is used once and does not need to be used anywhere else. The code will seem more self-contained and readable when handlers are defined right inside the code calling them, rather than having to search elsewhere to find the function body.

```
setTimeout(function() {
  console.log('Hello world!');
}, 1000);
```

Arguments to functional programming constructs or Lodash (similar to callbacks).

```
const arr = [1, 2, 3];
const double = arr.map(function(el) {
  return el * 2;
});
console.log(double); // [2, 4, 6]
```

References

- https://www.quora.com/What-is-a-typical-usecase-for-anonymous-functions
- https://stackoverflow.com/questions/10273185/what-are-the-benefits-to-using-anonymous-functions-instead-of-named-functions-fo

[↑] Back to top

How do you organize your code? (module pattern, classical inheritance?)

In the past, I've used Backbone for my models which encourages a more OOP approach, creating Backbone models and attaching methods to them.

The module pattern is still great, but these days, I use React/Redux which utilize a single-directional data flow based on Flux architecture. I would represent my app's models using plain objects and write utility pure functions to manipulate these objects. State is manipulated using actions and reducers like in any other Redux application.

I avoid using classical inheritance where possible. When and if I do, I stick to these rules.

[↑] Back to top

What's the difference between host objects and native objects?

Native objects are objects that are part of the JavaScript language defined by the ECMAScript specification, such as `String`, `Math`, `RegExp`, `Object`, `Function`, etc.

Host objects are provided by the runtime environment (browser or Node), such as `window`, `XMLHTTPRequest`, etc.

References

- https://stackoverflow.com/questions/7614317/what-is-the-difference-between-native-objects-and-host-objects

[↑] Back to top

Difference between: `function Person(){}`, `var person = Person()`, and `var person = new Person()`?

This question is pretty vague. My best guess at its intention is that it is asking about constructors in JavaScript. Technically speaking, `function Person(){}` is just a normal function declaration. The convention is to use PascalCase for functions that are intended to be used as constructors.

`var person = Person()` invokes the `Person` as a function, and not as a constructor. Invoking as such is a common mistake if the function is intended to be used as a constructor. Typically, the constructor does not return anything, hence invoking the constructor like a normal function will return `undefined` and that gets assigned to the variable intended as the instance.

`var person = new Person()` creates an instance of the `Person` object using the `new` operator, which inherits from `Person.prototype`. An alternative would be to use `Object.create`, such as: `Object.create(Person.prototype)`.

```
function Person(name) {
  this.name = name;
}

var person = Person('John');
console.log(person); // undefined
console.log(person.name); // Uncaught TypeError: Cannot read property 'name'
of undefined

var person = new Person('John');
console.log(person); // Person { name: "John" }
console.log(person.name); // "john"
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new

[↑] Back to top

What's the difference between `.call` and `.apply`?

Both `.call` and `.apply` are used to invoke functions and the first parameter will be used as the value of `this` within the function. However, `.call` takes in comma-separated arguments as the next arguments while `.apply` takes in an array of arguments as the next argument. An easy way to remember this is C for `call` and comma-separated and A for `apply` and an array of arguments.

```
function add(a, b) {
  return a + b;
}

console.log(add.call(null, 1, 2)); // 3
console.log(add.apply(null, [1, 2])); // 3
```

[↑] Back to top

Explain `Function.prototype.bind`.

Taken word-for-word from MDN:

> The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

In my experience, it is most useful for binding the value of `this` in methods of classes that you want to pass into other functions. This is frequently done in React components.

References

- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind

[↑] Back to top

When would you use `document.write()`?

`document.write()` writes a string of text to a document stream opened by `document.open()`. When `document.write()` is executed after the page has loaded, it will call `document.open` which clears the whole document (`<head>` and `<body>` removed!) and replaces the contents with the given parameter value. Hence it is usually considered dangerous and prone to misuse.

There are some answers online that explain `document.write()` is being used in analytics code or when you want to include styles that should only work if JavaScript is enabled. It is even being used in HTML5 boilerplate to load scripts in parallel and preserve execution order! However, I suspect those reasons might be outdated and in the modern day, they can be achieved without using `document.write()`. Please do correct me if I'm wrong about this.

References

- https://www.quirksmode.org/blog/archives/2005/06/three_javascrip_1.html
- https://github.com/h5bp/html5-boilerplate/wiki/Script-Loading-Techniques#documentwrite-script-tag

[↑] Back to top

What's the difference between feature detection, feature inference, and using the UA string?

Feature Detection

Feature detection involves working out whether a browser supports a certain block of code, and running different code depending on whether it does (or doesn't), so that the browser can always provide a working experience rather crashing/erroring in some browsers. For example:

```
if ('geolocation' in navigator) {
  // Can use navigator.geolocation
} else {
  // Handle lack of feature
}
```

Modernizr is a great library to handle feature detection.

Feature Inference

Feature inference checks for a feature just like feature detection, but uses another function because it assumes it will also exist, e.g.:

```
if (document.getElementsByTagName) {
  element = document.getElementById(id);
}
```

This is not really recommended. Feature detection is more foolproof.

UA String

This is a browser-reported string that allows the network protocol peers to identify the application type, operating system, software vendor or software version of the requesting software user agent. It can be accessed via `navigator.userAgent`. However, the string is tricky to parse and can be spoofed. For example, Chrome reports both as Chrome and Safari. So to detect Safari you have to check for the Safari string and the absence of the Chrome string. Avoid this method.

References

- https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Feature_detection
- https://stackoverflow.com/questions/20104930/whats-the-difference-between-feature-detection-feature-inference-and-using-th
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Browser_detection_using_the_user_age

nt

[↑] Back to top

Explain Ajax in as much detail as possible.

Ajax (asynchronous JavaScript and XML) is a set of web development techniques using many web technologies on the client side to create asynchronous web applications. With Ajax, web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows for web pages, and by extension web applications, to change content dynamically without the need to reload the entire page. In practice, modern implementations commonly substitute use JSON instead of XML, due to the advantages of JSON being native to JavaScript.

The `XMLHttpRequest` API is frequently used for the asynchronous communication or these days, the `fetch` API.

References

- https://en.wikipedia.org/wiki/Ajax_(programming)
- https://developer.mozilla.org/en-US/docs/AJAX

[↑] Back to top

What are the advantages and disadvantages of using Ajax?

Advantages

- Better interactivity. New content from the server can be changed dynamically without the need to reload the entire page.
- Reduce connections to the server since scripts and stylesheets only have to be requested once.
- State can be maintained on a page. JavaScript variables and DOM state will persist because the main container page was not reloaded.
- Basically most of the advantages of an SPA.

Disadvantages

- Dynamic webpages are harder to bookmark.
- Does not work if JavaScript has been disabled in the browser.
- Some webcrawlers do not execute JavaScript and would not see content that has been loaded by JavaScript.
- Basically most of the disadvantages of an SPA.

[↑] Back to top

Explain how JSONP works (and how it's not really Ajax).

JSONP (JSON with Padding) is a method commonly used to bypass the cross-domain policies in web browsers because Ajax requests from the current page to a cross-origin domain is not allowed.

JSONP works by making a request to a cross-origin domain via a `<script>` tag and usually with a `callback` query parameter, for example: `https://example.com?callback=printData`. The server will then wrap the data within a function called `printData` and return it to the client.

```
<!-- https://mydomain.com -->
<script>
function printData(data) {
  console.log(`My name is ${data.name}!`);
}
</script>

<script src="https://example.com?callback=printData"></script>
```

```
// File loaded from https://example.com?callback=printData
printData({ name: 'Yang Shun' });
```

The client has to have the `printData` function in its global scope and the function will be executed by the client when the response from the cross-origin domain is received.

JSONP can be unsafe and has some security implications. As JSONP is really JavaScript, it can do everything else JavaScript can do, so you need to trust the provider of the JSONP data.

These days, CORS is the recommended approach and JSONP is seen as a hack.

References

- https://stackoverflow.com/a/2067584/1751946

[↑] Back to top

Have you ever used JavaScript templating? If so, what libraries have you used?

Yes. Handlebars, Underscore, Lodash, AngularJS, and JSX. I disliked templating in AngularJS because it made heavy use of strings in the directives and typos would go uncaught. JSX is my new favorite as it is closer to JavaScript and there is barely any syntax to learn. Nowadays, you can even use ES2015 template string literals as a quick way for creating templates without relying on third-party code.

```
const template = `<div>My name is: ${name}</div>`;
```

However, do be aware of a potential XSS in the above approach as the contents are not escaped for you, unlike in templating libraries.

[↑] Back to top

Explain "hoisting".

Hoisting is a term used to explain the behavior of variable declarations in your code. Variables declared or initialized with the `var` keyword will have their declaration "moved" up to the top of the current scope, which we refer to as hoisting. However, only the declaration is hoisted, the assignment (if there is one), will stay where it is.

Note that the declaration is not actually moved - the JavaScript engine parses the declarations during compilation and becomes aware of declarations and their scopes. It is just easier to understand this behavior by visualizing the declarations as being hoisted to the top of their scope. Let's explain with a few examples.

```
// var declarations are hoisted.
console.log(foo); // undefined
var foo = 1;
console.log(foo); // 1

// let/const declarations are NOT hoisted.
console.log(bar); // ReferenceError: bar is not defined
let bar = 2;
console.log(bar); // 2
```

Function declarations have the body hoisted while the function expressions (written in the form of variable declarations) only has the variable declaration hoisted.

```
// Function Declaration
console.log(foo); // [Function: foo]
foo(); // 'FOOOOO'
function foo() {
  console.log('FOOOOO');
}
console.log(foo); // [Function: foo]

// Function Expression
console.log(bar); // undefined
bar(); // Uncaught TypeError: bar is not a function
var bar = function() {
  console.log('BARRRR');
};
console.log(bar); // [Function: bar]
```

[↑] Back to top

Describe event bubbling.

When an event triggers on a DOM element, it will attempt to handle the event if there is a listener attached, then the event is bubbled up to its parent and the same thing happens. This bubbling occurs up the element's ancestors all the way to the `document`. Event bubbling is the mechanism behind event delegation.

[↑] Back to top

What's the difference between an "attribute" and a "property"?

Attributes are defined on the HTML markup but properties are defined on the DOM. To illustrate the difference, imagine we have this text field in our HTML: `<input type="text" value="Hello">`.

```
const input = document.querySelector('input');
console.log(input.getAttribute('value')); // Hello
console.log(input.value); // Hello
```

But after you change the value of the text field by adding "World!" to it, this becomes:

```
console.log(input.getAttribute('value')); // Hello
console.log(input.value); // Hello World!
```

References

- https://stackoverflow.com/questions/6003819/properties-and-attributes-in-html

[↑] Back to top

Why is extending built-in JavaScript objects not a good idea?

Extending a built-in/native JavaScript object means adding properties/functions to its `prototype`. While this may seem like a good idea at first, it is dangerous in practice. Imagine your code uses a few libraries that both extend the `Array.prototype` by adding the same `contains` method, the implementations will overwrite each other and your code will break if the behavior of these two methods is not the same.

The only time you may want to extend a native object is when you want to create a polyfill, essentially providing your own implementation for a method that is part of the JavaScript specification but might not exist in the user's browser due to it being an older browser.

References

- http://lucybain.com/blog/2014/js-extending-built-in-objects/

[↑] Back to top

Difference between document `load` event and document `DOMContentLoaded` event?

The `DOMContentLoaded` event is fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading.

`window`'s `load` event is only fired after the DOM and all dependent resources and assets have loaded.

References

- https://developer.mozilla.org/en-US/docs/Web/Events/DOMContentLoaded
- https://developer.mozilla.org/en-US/docs/Web/Events/load

[↑] Back to top

What is the difference between `==` and `===`?

`==` is the abstract equality operator while `===` is the strict equality operator. The `==` operator will compare for equality after doing any necessary type conversions. The `===` operator will not do type conversion, so if two values are not the same type `===` will simply return `false`. When using `==`, funky things can happen, such as:

```
1 == '1'; // true
1 == [1]; // true
1 == true; // true
0 == ''; // true
0 == '0'; // true
0 == false; // true
```

My advice is never to use the `==` operator, except for convenience when comparing against `null` or `undefined`, where `a == null` will return `true` if `a` is `null` or `undefined`.

```
var a = null;
console.log(a == null); // true
console.log(a == undefined); // true
```

References

- https://stackoverflow.com/questions/359494/which-equals-operator-vs-should-be-used-in-javascript-comparisons

[↑] Back to top

Explain the same-origin policy with regards to JavaScript.

The same-origin policy prevents JavaScript from making requests across domain boundaries. An origin is defined as a combination of URI scheme, hostname, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model.

References

- https://en.wikipedia.org/wiki/Same-origin_policy

[↑] Back to top

Make this work:

```
duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]
```

```
function duplicate(arr) {
  return arr.concat(arr);
}

duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]
```

[↑] Back to top

Why is it called a Ternary expression, what does the word "Ternary" indicate?

"Ternary" indicates three, and a ternary expression accepts three operands, the test condition, the "then" expression and the "else" expression. Ternary expressions are not specific to JavaScript and I'm not sure why it is even in this list.

References

- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

[↑] Back to top

What is `"use strict";`? What are the advantages and disadvantages to using it?

'use strict' is a statement used to enable strict mode to entire scripts or individual functions. Strict mode is a way to opt into a restricted variant of JavaScript.

Advantages:

- Makes it impossible to accidentally create global variables.
- Makes assignments which would otherwise silently fail to throw an exception.
- Makes attempts to delete undeletable properties throw (where before the attempt would simply have no effect).
- Requires that function parameter names be unique.
- `this` is undefined in the global context.
- It catches some common coding bloopers, throwing exceptions.
- It disables features that are confusing or poorly thought out.

Disadvantages:

- Many missing features that some developers might be used to.
- No more access to `function.caller` and `function.arguments`.
- Concatenation of scripts written in different strict modes might cause issues.

Overall, I think the benefits outweigh the disadvantages, and I never had to rely on the features that strict mode blocks. I would recommend using strict mode.

References

- http://2ality.com/2011/10/strict-mode-hatred.html
- http://lucybain.com/blog/2014/js-use-strict/

[↑] Back to top

Create a for loop that iterates up to `100` while outputting "fizz" at multiples of `3`, "buzz" at multiples of `5` and "fizzbuzz" at multiples of `3` and `5`.

Check out this version of FizzBuzz by Paul Irish.

```
for (let i = 1; i <= 100; i++) {
  let f = i % 3 == 0,
    b = i % 5 == 0;
  console.log(f ? (b ? 'FizzBuzz' : 'Fizz') : b ? 'Buzz' : i);
}
```

I would not advise you to write the above during interviews though. Just stick with the long but clear approach. For more wacky versions of FizzBuzz, check out the reference link below.

References

- https://gist.github.com/jaysonrowe/1592432

[↑] Back to top

Why is it, in general, a good idea to leave the global scope of a website as-is and never touch it?

Every script has access to the global scope, and if everyone uses the global namespace to define their variables, collisions will likely occur. Use the module pattern (IIFEs) to encapsulate your variables within a local namespace.

[↑] Back to top

Why would you use something like the `load` event? Does this event have disadvantages? Do you know any alternatives, and why would you use those?

The `load` event fires at the end of the document loading process. At this point, all of the objects in the document are in the DOM, and all the images, scripts, links and sub-frames have finished loading.

The DOM event `DOMContentLoaded` will fire after the DOM for the page has been constructed, but do not wait for other resources to finish loading. This is preferred in certain cases when you do not need the full page to be loaded before initializing.

TODO.

References

- https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onload

[↑] Back to top

Explain what a single page app is and how to make one SEO-friendly.

The below is taken from the awesome Grab Front End Guide, which coincidentally, is written by me!

Web developers these days refer to the products they build as web apps, rather than websites. While there is no strict difference between the two terms, web apps tend to be highly interactive and dynamic, allowing the user to perform actions and receive a response to their action. Traditionally, the browser receives HTML from the server and renders it. When the user navigates to another URL, a full-page refresh is required and the server sends fresh new HTML to the new page. This is called server-side rendering.

However, in modern SPAs, client-side rendering is used instead. The browser loads the initial page from the server, along with the scripts (frameworks, libraries, app code) and stylesheets required for the whole app. When the user navigates to other pages, a page refresh is not triggered. The URL of the page is updated via the HTML5 History API. New data required for the new page, usually in JSON format, is retrieved by the browser via AJAX requests to the server. The SPA then dynamically updates the page with the data via JavaScript, which it has already downloaded in the initial page load. This model is similar to how native mobile apps work.

The benefits:

- The app feels more responsive and users do not see the flash between page navigations due to full-page refreshes.
- Fewer HTTP requests are made to the server, as the same assets do not have to be downloaded again for each page load.
- Clear separation of the concerns between the client and the server; you can easily build new clients for different platforms (e.g. mobile, chatbots, smart watches) without having to modify the server code. You can also modify the technology stack on the client and server independently, as long as the API contract is not broken.

The downsides:

- Heavier initial page load due to the loading of framework, app code, and assets required for multiple pages.
- There's an additional step to be done on your server which is to configure it to route all requests to a single entry point and allow client-side routing to take over from there.
- SPAs are reliant on JavaScript to render content, but not all search engines execute JavaScript

during crawling, and they may see empty content on your page. This inadvertently hurts the Search Engine Optimization (SEO) of your app. However, most of the time, when you are building apps, SEO is not the most important factor, as not all the content needs to be indexable by search engines. To overcome this, you can either server-side render your app or use services such as Prerender to "render your javascript in a browser, save the static HTML, and return that to the crawlers".

References

- https://github.com/grab/front-end-guide#single-page-apps-spas
- http://stackoverflow.com/questions/21862054/single-page-app-advantages-and-disadvantages
- http://blog.isquaredsoftware.com/presentations/2016-10-revolution-of-web-dev/
- https://medium.freecodecamp.com/heres-why-client-side-rendering-won-46a349fadb52

[↑] Back to top

What is the extent of your experience with Promises and/or their polyfills?

Possess working knowledge of it. A promise is an object that may produce a single value sometime in the future: either a resolved value or a reason that it's not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states: fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

Some common polyfills are `$.deferred`, Q and Bluebird but not all of them comply with the specification. ES2015 supports Promises out of the box and polyfills are typically not needed these days.

References

- https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-promise-27fc71e77261

[↑] Back to top

What are the pros and cons of using Promises instead of callbacks?

Pros

- Avoid callback hell which can be unreadable.
- Makes it easy to write sequential asynchronous code that is readable with `.then()`.
- Makes it easy to write parallel asynchronous code with `Promise.all()`.
- With promises, these scenarios which are present in callbacks-only coding, will not happen:
  - Call the callback too early
  - Call the callback too late (or never)
  - Call the callback too few or too many times

    - Fail to pass along any necessary environment/parameters
    - Swallow any errors/exceptions that may happen

Cons

- Slightly more complex code (debatable).
- In older browsers where ES2015 is not supported, you need to load a polyfill in order to use it.

References

- https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch3.md

[↑] Back to top

What are some of the advantages/disadvantages of writing JavaScript code in a language that compiles to JavaScript?

Some examples of languages that compile to JavaScript include CoffeeScript, Elm, ClojureScript, PureScript, and TypeScript.

Advantages:

- Fixes some of the longstanding problems in JavaScript and discourages JavaScript anti-patterns.
- Enables you to write shorter code, by providing some syntactic sugar on top of JavaScript, which I think ES5 lacks, but ES2015 is awesome.
- Static types are awesome (in the case of TypeScript) for large projects that need to be maintained over time.

Disadvantages:

- Require a build/compile process as browsers only run JavaScript and your code will need to be compiled into JavaScript before being served to browsers.
- Debugging can be a pain if your source maps do not map nicely to your pre-compiled source.
- Most developers are not familiar with these languages and will need to learn it. There's a ramp up cost involved for your team if you use it for your projects.
- Smaller community (depends on the language), which means resources, tutorials, libraries, and tooling would be harder to find.
- IDE/editor support might be lacking.
- These languages will always be behind the latest JavaScript standard.
- Developers should be cognizant of what their code is being compiled to—because that is what would actually be running, and that is what matters in the end.

Practically, ES2015 has vastly improved JavaScript and made it much nicer to write. I don't really see the need for CoffeeScript these days.

References

- https://softwareengineering.stackexchange.com/questions/72569/what-are-the-pros-and-co

ns-of-coffeescript

[↑] Back to top

What tools and techniques do you use for debugging JavaScript code?

- React and Redux
  - React Devtools
  - Redux Devtools
- Vue
  - Vue Devtools
- JavaScript
  - Chrome Devtools
  - `debugger` statement
  - Good old `console.log` debugging

References

- https://hackernoon.com/twelve-fancy-chrome-devtools-tips-dc1e39d10d9d
- https://raygun.com/blog/javascript-debugging/

[↑] Back to top

What language constructions do you use for iterating over object properties and array items?

For objects:

- `for-in` loops - `for (var property in obj) { console.log(property); }`. However, this will also iterate through its inherited properties, and you will add an `obj.hasOwnProperty(property)` check before using it.
- `Object.keys()` - `Object.keys(obj).forEach(function (property) { ... })`. `Object.keys()` is a static method that will lists all enumerable properties of the object that you pass it.
- `Object.getOwnPropertyNames()` - `Object.getOwnPropertyNames(obj).forEach(function (property) { ... })`. `Object.getOwnPropertyNames()` is a static method that will lists all enumerable and non-enumerable properties of the object that you pass it.

For arrays:

- `for` loops - `for (var i = 0; i < arr.length; i++)`. The common pitfall here is that `var` is in the function scope and not the block scope and most of the time you would want block scoped iterator variable. ES2015 introduces `let` which has block scope and it is recommended to use that instead. So this becomes: `for (let i = 0; i < arr.length; i++)`.

- `forEach` - `arr.forEach(function (el, index) { ... })`. This construct can be more convenient at times because you do not have to use the `index` if all you need is the array elements. There are also the `every` and `some` methods which will allow you to terminate the iteration early.
- `for-of` loops - `for (let elem of arr) { ... }`. ES6 introduces a new loop, the `for-of` loop, that allows you to loop over objects that conform to the iterable protocol such as `String`, `Array`, `Map`, `Set`, etc. It combines the advantages of the `for` loop and the `forEach()` method. The advantage of the `for` loop is that you can break from it, and the advantage of `forEach()` is that it is more concise than the `for` loop because you don't need a counter variable. With the `for-of` loop, you get both the ability to break from a loop and a more concise syntax.

Most of the time, I would prefer the `.forEach` method, but it really depends on what you are trying to do. Before ES6, we used `for` loops when we needed to prematurely terminate the loop using `break`. But now with ES6, we can do that with `for-of` loops. I would use `for` loops when I need even more flexibility, such as incrementing the iterator more than once per loop.

Also, when using the `for-of` loop, if you need to access both the index and value of each array element, you can do so with the ES6 Array `entries()` method and destructuring:

```
const arr = ['a', 'b', 'c'];

for (let [index, elem] of arr.entries()) {
  console.log(index, ': ', elem);
}
```

References

- http://2ality.com/2015/08/getting-started-es6.html#from-for-to-foreach-to-for-of
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/entries

[↑] Back to top

Explain the difference between mutable and immutable objects.

Immutability is a core principle in functional programming, and has lots to offer to object-oriented programs as well. A mutable object is an object whose state can be modified after it is created. An immutable object is an object whose state cannot be modified after it is created.

What is an example of an immutable object in JavaScript?

In JavaScript, some built-in types (numbers, strings) are immutable, but custom objects are generally mutable.

Some built-in immutable JavaScript objects are `Math`, `Date`.

Here are a few ways to add/simulate immutability on plain JavaScript objects.

Object Constant Properties

By combining `writable: false` and `configurable: false`, you can essentially create a constant (cannot be changed, redefined or deleted) as an object property, like:

```
let myObject = {};
Object.defineProperty(myObject, 'number', {
  value: 42,
  writable: false,
  configurable: false,
});
console.log(myObject.number); // 42
myObject.number = 43;
console.log(myObject.number); // 42
```

Prevent Extensions

If you want to prevent an object from having new properties added to it, but otherwise leave the rest of the object's properties alone, call `Object.preventExtensions(...)`:

```
var myObject = {
  a: 2
};

Object.preventExtensions(myObject);

myObject.b = 3;
myObject.b; // undefined
```

In non-strict mode, the creation of `b` fails silently. In strict mode, it throws a `TypeError`.

Seal

`Object.seal()` creates a "sealed" object, which means it takes an existing object and essentially calls `Object.preventExtensions()` on it, but also marks all its existing properties as `configurable: false`.

So, not only can you not add any more properties, but you also cannot reconfigure or delete any existing properties (though you can still modify their values).

Freeze

`Object.freeze()` creates a frozen object, which means it takes an existing object and essentially calls `Object.seal()` on it, but it also marks all "data accessor" properties as writable:false, so that their values cannot be changed.

This approach is the highest level of immutability that you can attain for an object itself, as it prevents any changes to the object or to any of its direct properties (though, as mentioned above, the contents of any referenced other objects are unaffected).

```
var immutable = Object.freeze({});
```

Freezing an object does not allow new properties to be added to an object and prevents from removing or altering the existing properties. `Object.freeze()` preserves the enumerability, configurability, writability and the prototype of the object. It returns the passed object and does not create a frozen copy.

What are the pros and cons of immutability?

Pros

TODO

Cons

TODO

How can you achieve immutability in your own code?

One way to achieve immutability is to use libraries like immutable.js, mori or immer.

The alternative is to use `const` declarations combined with the techniques mentioned above for creation. For "mutating" objects, use the spread operator, `Object.assign`, `Array.concat()`, etc., to create new objects instead of mutate the original object.

Examples:

```
// Array Example
const arr = [1, 2, 3];
const newArr = [...arr, 4]; // [1, 2, 3, 4]

// Object Example
const human = Object.freeze({race: 'human'});
const john = { ...human, name: 'John' }; // {race: "human", name: "John"}
const alienJohn = { ...john, race: 'alien' }; // {race: "alien", name: "John"}
```

References

- https://stackoverflow.com/questions/1863515/pros-cons-of-immutability-vs-mutability
- https://www.sitepoint.com/immutability-javascript/
- https://wecodetheweb.com/2016/02/12/immutable-javascript-using-es6-and-beyond/

[↑] Back to top

Explain the difference between synchronous and asynchronous functions.

Synchronous functions are blocking while asynchronous functions are not. In synchronous functions, statements complete before the next statement is run. In this case, the program is evaluated exactly in order of the statements and execution of the program is paused if one of the statements take a very long time.

Asynchronous functions usually accept a callback as a parameter and execution continue on the next line immediately after the asynchronous function is invoked. The callback is only invoked when the asynchronous operation is complete and the call stack is empty. Heavy duty operations such as loading data from a web server or querying a database should be done asynchronously so that the main thread can continue executing other operations instead of blocking until that long operation to complete (in the case of browsers, the UI will freeze).

[↑] Back to top

What is event loop? What is the difference between call stack and task queue?

The event loop is a single-threaded loop that monitors the call stack and checks if there is any work to be done in the task queue. If the call stack is empty and there are callback functions in the task queue, a function is dequeued and pushed onto the call stack to be executed.

If you haven't already checked out Philip Robert's talk on the Event Loop, you should. It is one of the most viewed videos on JavaScript.

References

- https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html
- http://theproactiveprogrammer.com/javascript/the-javascript-event-loop-a-stack-and-a-queue/

[↑] Back to top

Explain the differences on the usage of `foo` between `function foo() {}` and `var foo = function() {}`

The former is a function declaration while the latter is a function expression. The key difference is that function declarations have its body hoisted but the bodies of function expressions are not (they have the same hoisting behavior as variables). For more explanation on hoisting, refer to the question above on hoisting. If you try to invoke a function expression before it is defined, you will get an `Uncaught TypeError: XXX is not a function` error.

Function Declaration

```
foo(); // 'FOOOOO'
function foo() {
  console.log('FOOOOO');
}
```

Function Expression

```
foo(); // Uncaught TypeError: foo is not a function
var foo = function() {
  console.log('FOOOOO');
};
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function

[↑] Back to top

What are the differences between variables created using `let`, `var` or `const`?

Variables declared using the `var` keyword are scoped to the function in which they are created, or if created outside of any function, to the global object. `let` and `const` are *block scoped*, meaning they are only accessible within the nearest set of curly braces (function, if-else block, or for-loop).

```
function foo() {
  // All variables are accessible within functions.
  var bar = 'bar';
  let baz = 'baz';
  const qux = 'qux';

  console.log(bar); // bar
  console.log(baz); // baz
  console.log(qux); // qux
}

console.log(bar); // ReferenceError: bar is not defined
console.log(baz); // ReferenceError: baz is not defined
console.log(qux); // ReferenceError: qux is not defined
```

```
if (true) {
  var bar = 'bar';
  let baz = 'baz';
  const qux = 'qux';
}

// var declared variables are accessible anywhere in the function scope.
console.log(bar); // bar
// let and const defined variables are not accessible outside of the block
they were defined in.
console.log(baz); // ReferenceError: baz is not defined
console.log(qux); // ReferenceError: qux is not defined
```

`var` allows variables to be hoisted, meaning they can be referenced in code before they are declared. `let` and `const` will not allow this, instead throwing an error.

```
console.log(foo); // undefined

var foo = 'foo';

console.log(baz); // ReferenceError: can't access lexical declaration 'baz'
before initialization

let baz = 'baz';

console.log(bar); // ReferenceError: can't access lexical declaration 'bar'
before initialization

const bar = 'bar';
```

Redeclaring a variable with `var` will not throw an error, but 'let' and 'const' will.

```
var foo = 'foo';
var foo = 'bar';
console.log(foo); // "bar"

let baz = 'baz';
let baz = 'qux'; // Uncaught SyntaxError: Identifier 'baz' has already been
declared
```

`let` and `const` differ in that `let` allows reassigning the variable's value while `const` does not.

```
// This is fine.
let foo = 'foo';
foo = 'bar';

// This causes an exception.
const baz = 'baz';
baz = 'qux';
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

[↑] Back to top

What are the differences between ES6 class and ES5 function constructors?

Let's first look at example of each:

```
// ES5 Function Constructor
function Person(name) {
  this.name = name;
}

// ES6 Class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

For simple constructors, they look pretty similar.

The main difference in the constructor comes when using inheritance. If we want to create a `Student` class that subclasses `Person` and add a `studentId` field, this is what we have to do in addition to the above.

```
// ES5 Function Constructor
function Student(name, studentId) {
  // Call constructor of superclass to initialize superclass-derived members.
  Person.call(this, name);

  // Initialize subclass's own members.
  this.studentId = studentId;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

// ES6 Class
class Student extends Person {
  constructor(name, studentId) {
    super(name);
    this.studentId = studentId;
  }
}
```

It's much more verbose to use inheritance in ES5 and the ES6 version is easier to understand and remember.

References

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance
- https://eli.thegreenplace.net/2013/10/22/classical-inheritance-in-javascript-es5

[↑] Back to top

Can you offer a use case for the new arrow => function syntax? How does this new syntax differ from other functions?

One obvious benefit of arrow functions is to simplify the syntax needed to create functions, without a need for the `function` keyword. The `this` within arrow functions is also bound to the enclosing scope which is different compared to regular functions where the `this` is determined by the object calling it. Lexically-scoped `this` is useful when invoking callbacks especially in React components.

[↑] Back to top

What advantage is there for using the arrow syntax for a method in a constructor?

The main advantage of using an arrow function as a method inside a constructor is that the value of `this` gets set at the time of the function creation and can't change after that. So, when the constructor is used to create a new object, `this` will always refer to that object. For example, let's say we have a `Person` constructor that takes a first name as an argument has two methods to `console.log` that name, one as a regular function and one as an arrow function:

```
const Person = function(firstName) {
  this.firstName = firstName;
  this.sayName1 = function() { console.log(this.firstName); };
  this.sayName2 = () => { console.log(this.firstName); };
};

const john = new Person('John');
const dave = new Person('Dave');

john.sayName1(); // John
john.sayName2(); // John

// The regular function can have its 'this' value changed, but the arrow
function cannot
john.sayName1.call(dave); // Dave (because "this" is now the dave object)
john.sayName2.call(dave); // John

john.sayName1.apply(dave); // Dave (because 'this' is now the dave object)
john.sayName2.apply(dave); // John

john.sayName1.bind(dave)(); // Dave (because 'this' is now the dave object)
john.sayName2.bind(dave)(); // John

var sayNameFromWindow1 = john.sayName1;
sayNameFromWindow1(); // undefined (because 'this' is now the window object)
```

```
var sayNameFromWindow2 = john.sayName2;
sayNameFromWindow2(); // John
```

The main takeaway here is that `this` can be changed for a normal function, but the context always stays the same for an arrow function. So even if you are passing around your arrow function to different parts of your application, you wouldn't have to worry about the context changing.

This can be particularly helpful in React class components. If you define a class method for something such as a click handler using a normal function, and then you pass that click handler down into a child component as a prop, you will need to also bind `this` in the constructor of the parent component. If you instead use an arrow function, there is no need to also bind "this", as the method will automatically get its "this" value from its enclosing lexical context. (See this article for an excellent demonstration and sample code: https://medium.com/@machnicki/handle-events-in-react-with-arrow-functions-ede88184bbb)

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
- https://medium.com/@machnicki/handle-events-in-react-with-arrow-functions-ede88184bbb

[↑] Back to top

What is the definition of a higher-order function?

A higher-order function is any function that takes one or more functions as arguments, which it uses to operate on some data, and/or returns a function as a result. Higher-order functions are meant to abstract some operation that is performed repeatedly. The classic example of this is `map`, which takes an array and a function as arguments. `map` then uses this function to transform each item in the array, returning a new array with the transformed data. Other popular examples in JavaScript are `forEach`, `filter`, and `reduce`. A higher-order function doesn't just need to be manipulating arrays as there are many use cases for returning a function from another function. `Function.prototype.bind` is one such example in JavaScript.

Map

Let say we have an array of names which we need to transform each string to uppercase.

```
const names = ['irish', 'daisy', 'anna'];
```

The imperative way will be as such:

```
const transformNamesToUppercase = function(names) {
  const results = [];
  for (let i = 0; i < names.length; i++) {
    results.push(names[i].toUpperCase());
  }
  return results;
};
transformNamesToUppercase(names); // ['IRISH', 'DAISY', 'ANNA']
```

Use `.map(transformerFn)` makes the code shorter and more declarative.

```
const transformNamesToUppercase = function(names) {
  return names.map(name => name.toUpperCase());
};
transformNamesToUppercase(names); // ['IRISH', 'DAISY', 'ANNA']
```

References

- https://medium.com/javascript-scene/higher-order-functions-composing-software-5365cf2cbe99
- https://hackernoon.com/effective-functional-javascript-first-class-and-higher-order-functions-713fde8df50a
- https://eloquentjavascript.net/05_higher_order.html

[↑] Back to top

Can you give an example for destructuring an object or an array?

Destructuring is an expression available in ES6 which enables a succinct and convenient way to extract values of Objects or Arrays and place them into distinct variables.

Array destructuring

```
// Variable assignment.
const foo = ['one', 'two', 'three'];

const [one, two, three] = foo;
console.log(one); // "one"
console.log(two); // "two"
console.log(three); // "three"
```

```
// Swapping variables
let a = 1;
let b = 3;

[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1
```

Object destructuring

```
// Variable assignment.
const o = { p: 42, q: true };
const { p, q } = o;

console.log(p); // 42
console.log(q); // true
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
- https://ponyfoo.com/articles/es6-destructuring-in-depth

[↑] Back to top

ES6 Template Literals offer a lot of flexibility in generating strings, can you give an example?

Template literals help make it simple to do string interpolation, or to include variables in a string. Before ES2015, it was common to do something like this:

```
var person = { name: 'Tyler', age: 28 };
console.log('Hi, my name is ' + person.name + ' and I am ' + person.age + '
years old!');
// 'Hi, my name is Tyler and I am 28 years old!'
```

With template literals, you can now create that same output like this instead:

```
const person = { name: 'Tyler', age: 28 };
console.log(`Hi, my name is ${person.name} and I am ${person.age} years
old!`);
// 'Hi, my name is Tyler and I am 28 years old!'
```

Note that you use backticks, not quotes, to indicate that you are using a template literal and that you can insert expressions inside the `${}` placeholders.

A second helpful use case is in creating multi-line strings. Before ES2015, you could create a multi-line string like this:

```
console.log('This is line one.\nThis is line two.');
// This is line one.
// This is line two.
```

Or if you wanted to break it up into multiple lines in your code so you didn't have to scroll to the right in your text editor to read a long string, you could also write it like this:

```
console.log('This is line one.\n' +
  'This is line two.');
// This is line one.
// This is line two.
```

Template literals, however, preserve whatever spacing you add to them. For example, to create that same multi-line output that we created above, you can simply do:

```
console.log(`This is line one.
This is line two.`);
// This is line one.
// This is line two.
```

Another use case of template literals would be to use as a substitute for templating libraries for simple variable interpolations:

```
const person = { name: 'Tyler', age: 28 };
document.body.innerHTML = `
  <div>
    <p>Name: ${person.name}</p>
    <p>Name: ${person.age}</p>
  </div>
`
```

Note that your code may be susceptible to XSS by using `.innerHTML`. Sanitize your data before displaying it if it came from a user!

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

[↑] Back to top

Can you give an example of a curry function and why this syntax offers an advantage?

Currying is a pattern where a function with more than one parameter is broken into multiple functions that, when called in series, will accumulate all of the required parameters one at a time. This technique can be useful for making code written in a functional style easier to read and compose. It's important to note that for a function to be curried, it needs to start out as one

function, then broken out into a sequence of functions that each accepts one parameter.

```
function curry(fn) {
  if (fn.length === 0) {
    return fn;
  }

  function _curried(depth, args) {
    return function(newArgument) {
      if (depth - 1 === 0) {
        return fn(...args, newArgument);
      }
      return _curried(depth - 1, [...args, newArgument]);
    };
  }

  return _curried(fn.length, []);
}

function add(a, b) {
  return a + b;
}

var curriedAdd = curry(add);
var addFive = curriedAdd(5);

var result = [0, 1, 2, 3, 4, 5].map(addFive); // [5, 6, 7, 8, 9, 10]
```

References

- https://hackernoon.com/currying-in-js-d9ddc64f162e

[↑] Back to top

What are the benefits of using spread syntax and how is it different from rest syntax?

ES6's spread syntax is very useful when coding in a functional paradigm as we can easily create copies of arrays or objects without resorting to `Object.create`, `slice`, or a library function. This language feature is used often in Redux and rx.js projects.

```
function putDookieInAnyArray(arr) {
  return [...arr, 'dookie'];
}

const result = putDookieInAnyArray(['I', 'really', "don't", 'like']); // ["I",
"really", "don't", "like", "dookie"]

const person = {
  name: 'Todd',
  age: 29,
};

const copyOfTodd = { ...person };
```

ES6's rest syntax offers a shorthand for including an arbitrary number of arguments to be passed to a function. It is like an inverse of the spread syntax, taking data and stuffing it into an array rather than unpacking an array of data, and it works in function arguments, as well as in array and object destructuring assignments.

```
function addFiveToABunchOfNumbers(...numbers) {
  return numbers.map(x => x + 5);
}

const result = addFiveToABunchOfNumbers(4, 5, 6, 7, 8, 9, 10); // [9, 10, 11,
12, 13, 14, 15]

const [a, b, ...rest] = [1, 2, 3, 4]; // a: 1, b: 2, rest: [3, 4]

const { e, f, ...others } = {
  e: 1,
  f: 2,
  g: 3,
  h: 4,
}; // e: 1, f: 2, others: { g: 3, h: 4 }
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

[↑] Back to top

How can you share code between files?

This depends on the JavaScript environment.

On the client (browser environment), as long as the variables/functions are declared in the global scope (`window`), all scripts can refer to them. Alternatively, adopt the Asynchronous Module Definition (AMD) via RequireJS for a more modular approach.

On the server (Node.js), the common way has been to use CommonJS. Each file is treated as a module and it can export variables and functions by attaching them to the `module.exports` object.

ES2015 defines a module syntax which aims to replace both AMD and CommonJS. This will eventually be supported in both browser and Node environments.

[↑] Back to top

References

- http://requirejs.org/docs/whyamd.html
- https://nodejs.org/docs/latest/api/modules.html
- http://2ality.com/2014/09/es6-modules-final.html

Why you might want to create static class members?

Static class members (properties/methods) are not tied to a specific instance of a class and have the same value regardless of which instance is referring to it. Static properties are typically configuration variables and static methods are usually pure utility functions which do not depend on the state of the instance.

References

- https://stackoverflow.com/questions/21155438/when-to-use-static-variables-methods-and-when-to-use-instance-variables-methods

[↑] Back to top

Other Answers

- http://flowerszhong.github.io/2013/11/20/javascript-questions.html

---