You Don't Know JS Yet: Get Started - 2nd Edition

Appendix A: Exploring Further

In this appendix, we're going to explore some topics from the main chapter text in a bit more detail. Think of this content as an optional preview of some of the more nuanced details covered throughout the rest of the book series.

Values vs. References

In Chapter 2, we introduced the two main types of values: primitives and objects. But we didn't discuss yet one key difference between the two: how these values are assigned and passed around.

In many languages, the developer can choose between assigning/passing a value as the value itself, or as a reference to the value. In JS, however, this decision is entirely determined by the kind of value. That surprises a lot of developers from other languages when they start using JS.

If you assign/pass a value itself, the value is copied. For example:

```
var myName = "Kyle";

var yourName = myName;
```

Here, the `yourName` variable has a separate copy of the `"Kyle"` string from the value that's stored in `myName`. That's because the value is a primitive, and primitive values are always assigned/passed as **value copies**.

Here's how you can prove there's two separate values involved:

```
var myName = "Kyle";

var yourName = myName;

myName = "Frank";

console.log(myName);
// Frank

console.log(yourName);
// Kyle
```

See how `yourName` wasn't affected by the re-assignment of `myName` to `"Frank"`? That's because each variable holds its own copy of the value.

By contrast, references are the idea that two or more variables are pointing at the same value, such that modifying this shared value would be reflected by an access via any of those references. In JS, only object values (arrays, objects, functions, etc.) are treated as references.

Consider:

```
var myAddress = {
    street: "123 JS Blvd",
    city: "Austin",
    state: "TX"
};

var yourAddress = myAddress;

// I've got to move to a new house!
myAddress.street = "456 TS Ave";

console.log(yourAddress.street);
// 456 TS Ave
```

Because the value assigned to `myAddress` is an object, it's held/assigned by reference, and thus the assignment to the `yourAddress` variable is a copy of the reference, not the object value itself. That's why the updated value assigned to the `myAddress.street` is reflected when we access `yourAddress.street`. `myAddress` and `yourAddress` have copies of the reference to the single shared object, so an update to one is an update to both.

Again, JS chooses the value-copy vs. reference-copy behavior based on the value type. Primitives are held by value, objects are held by reference. There's no way to override this in JS, in either direction.

So Many Function Forms

Recall this snippet from the "Functions" section in Chapter 2:

```
var awesomeFunction = function(coolThings) {
    // ..
    return amazingStuff;
};
```

The function expression here is referred to as an *anonymous function expression*, since it has no name identifier between the `function` keyword and the `(..)` parameter list. This point confuses many JS developers because as of ES6, JS performs a "name inference" on an anonymous function:

```
awesomeFunction.name;
// "awesomeFunction"
```

The `name` property of a function will reveal either its directly given name (in the case of a declaration) or its inferred name in the case of an anonymous function expression. That value is generally used by developer tools when inspecting a function value or when reporting an error stack trace.

So even an anonymous function expression *might* get a name. However, name inference only happens in limited cases such as when the function expression is assigned (with `=`). If you pass a function expression as an argument to a function call, for example, no name inference occurs; the `name` property will be an empty string, and the developer console will usually report "(anonymous function)".

Even if a name is inferred, **it's still an anonymous function.** Why? Because the inferred name is a metadata string value, not an available identifier to refer to the function. An anonymous function doesn't have an identifier to use to refer to itself from inside itself—for recursion, event unbinding, etc.

Compare the anonymous function expression form to:

```
// let awesomeFunction = ..
// const awesomeFunction = ..
var awesomeFunction = function someName(coolThings) {
    // ..
    return amazingStuff;
};

awesomeFunction.name;
// "someName"
```

This function expression is a *named function expression*, since the identifier `someName` is directly associated with the function expression at compile time; the association with the identifier `awesomeFunction` still doesn't happen until runtime at the time of that statement. Those two identifiers don't have to match; sometimes it makes sense to have them be different, other times it's better to have them be the same.

Notice also that the explicit function name, the identifier `someName`, takes precedence when assigning a *name* for the `name` property.

Should function expressions be named or anonymous? Opinions vary widely on this. Most developers tend to be unconcerned with using anonymous functions. They're shorter, and unquestionably more common in the broad sphere of JS code out there.

In my opinion, if a function exists in your program, it has a purpose; otherwise, take it out! And if it has a purpose, it has a natural name that describes that purpose.

If a function has a name, you the code author should include that name in the code, so that the reader does not have to infer that name from reading and mentally executing that function's source code. Even a trivial function body like `x * 2` has to be read to infer a name like "double" or "multBy2"; that brief extra mental work is unnecessary when you could just take a second to name the function "double" or "multBy2" *once*, saving the reader that repeated mental work every time it's read in the future.

There are, regrettably in some respects, many other function definition forms in JS as of early 2020 (maybe more in the future!).

Here are some more declaration forms:

```
// generator function declaration
function *two() { .. }

// async function declaration
async function three() { .. }

// async generator function declaration
async function *four() { .. }

// named function export declaration (ES6 modules)
export function five() { .. }
```

And here are some more of the (many!) function expression forms:

```
// IIFE
(function(){ .. })();
(function namedIIFE(){ .. })();

// asynchronous IIFE
(async function(){ .. })();
(async function namedAIIFE(){ .. })();

// arrow function expressions
var f;
f = () => 42;
f = x => x * 2;
f = (x) => x * 2;
f = (x,y) => x * y;
f = x => ({ x: x * 2 });
f = x => { return x * 2; };
f = async x => {
    var y = await doSomethingAsync(x);
    return y * 2;
};
someOperation( x => x * 2 );
// ..
```

Keep in mind that arrow function expressions are **syntactically anonymous**, meaning the syntax doesn't provide a way to provide a direct name identifier for the function. The function expression may get an inferred name, but only if it's one of the assignment forms, not in the (more common!) form of being passed as a function call argument (as in the last line of the snippet).

Since I don't think anonymous functions are a good idea to use frequently in your programs, I'm not a fan of using the `=>` arrow function form. This kind of function actually has a specific purpose (i.e., handling the `this` keyword lexically), but that doesn't mean we should use it for every function we write. Use the most appropriate tool for each job.

Functions can also be specified in class definitions and object literal definitions. They're typically referred to as "methods" when in these forms, though in JS this term doesn't have much observable difference over "function":

```
class SomethingKindaGreat {
    // class methods
    coolMethod() { .. }   // no commas!
    boringMethod() { .. }
}

var EntirelyDifferent = {
    // object methods
    coolMethod() { .. },   // commas!
    boringMethod() { .. },

    // (anonymous) function expression property
    oldSchool: function() { .. }
};
```

Phew! That's a lot of different ways to define functions.

There's no simple shortcut path here; you just have to build familiarity with all the function forms so you can recognize them in existing code and use them appropriately in the code you write. Study them closely and practice!

Coercive Conditional Comparison

Yes, that section name is quite a mouthful. But what are we talking about? We're talking about conditional expressions needing to perform coercion-oriented comparisons to make their decisions.

`if` and `? :`-ternary statements, as well as the test clauses in `while` and `for` loops, all perform an implicit value comparison. But what sort? Is it "strict" or "coercive"? Both, actually.

Consider:

```
var x = 1;

if (x) {
    // will run!
}

while (x) {
    // will run, once!
    x = false;
}
```

You might think of these `(x)` conditional expressions like this:

```
var x = 1;

if (x == true) {
    // will run!
}

while (x == true) {
    // will run, once!
    x = false;
}
```

In this specific case -- the value of `x` being `1` -- that mental model works, but it's not accurate more broadly. Consider:

```
var x = "hello";

if (x) {
    // will run!
}

if (x == true) {
    // won't run :(
}
```

Oops. So what is the `if` statement actually doing? This is the more accurate mental model:

```
var x = "hello";

if (Boolean(x) == true) {
    // will run
}

// which is the same as:

if (Boolean(x) === true) {
    // will run
}
```

Since the `Boolean(..)` function always returns a value of type boolean, the `==` vs `===` in this snippet is irrelevant; they'll both do the same thing. But the important part is to see that before the comparison, a coercion occurs, from whatever type `x` currently is, to boolean.

You just can't get away from coercions in JS comparisons. Buckle down and learn them.

Prototypal "Classes"

In Chapter 3, we introduced prototypes and showed how we can link objects through a prototype chain.

Another way of wiring up such prototype linkages served as the (honestly, ugly) predecessor to the elegance of the ES6 `class` system (see Chapter 2, "Classes"), and is referred to as prototypal classes.

TIP:

TIP:

While this style of code is quite uncommon in JS these days, it's still perplexingly rather common to be asked about it in job interviews!

Let's first recall the `Object.create(..)` style of coding:

```
var Classroom = {
    welcome() {
        console.log("Welcome, students!");
    }
};

var mathClass = Object.create(Classroom);

mathClass.welcome();
// Welcome, students!
```

Here, a `mathClass` object is linked via its prototype to a `Classroom` object. Through this linkage, the function call `mathClass.welcome()` is delegated to the method defined on `Classroom`.

The prototypal class pattern would have labeled this delegation behavior "inheritance," and alternatively have defined it (with the same behavior) as:

```
function Classroom() {
    // ..
}

Classroom.prototype.welcome = function hello() {
    console.log("Welcome, students!");
};

var mathClass = new Classroom();

mathClass.welcome();
// Welcome, students!
```

All functions by default reference an empty object at a property named `prototype`. Despite the confusing naming, this is **not** the function's *prototype* (where the function is prototype linked to), but rather the prototype object to *link to* when other objects are created by calling the function with `new`.

We add a `welcome` property on that empty object (called `Classroom.prototype`), pointing at the `hello()` function.

Then `new Classroom()` creates a new object (assigned to `mathClass`), and prototype links it to the existing `Classroom.prototype` object.

Though `mathClass` does not have a `welcome()` property/function, it successfully delegates to the function `Classroom.prototype.welcome()`.

This "prototypal class" pattern is now strongly discouraged, in favor of using ES6's `class` mechanism:

```
class Classroom {
    constructor() {
        // ..
    }

    welcome() {
        console.log("Welcome, students!");
    }
}

var mathClass = new Classroom();

mathClass.welcome();
// Welcome, students!
```

Under the covers, the same prototype linkage is wired up, but this `class` syntax fits the class-oriented design pattern much more cleanly than "prototypal classes".