You Don't Know JS: Types & Grammar

Chapter 3: Natives

Several times in Chapters 1 and 2, we alluded to various built-ins, usually called "natives," like `String` and `Number`. Let's examine those in detail now.

Here's a list of the most commonly used natives:

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()` -- added in ES6!

As you can see, these natives are actually built-in functions.

If you're coming to JS from a language like Java, JavaScript's `String()` will look like the `String(..)` constructor you're used to for creating string values. So, you'll quickly observe that you can do things like:

```
var s = new String( "Hello World!" );

console.log( s.toString() ); // "Hello World!"
```

It *is* true that each of these natives can be used as a native constructor. But what's being constructed may be different than you think.

```
var a = new String( "abc" );

typeof a; // "object" ... not "String"

a instanceof String; // true

Object.prototype.toString.call( a ); // "[object String]"
```

The result of the constructor form of value creation (`new String("abc")`) is an object wrapper around the primitive (`"abc"`) value.

Importantly, `typeof` shows that these objects are not their own special *types*, but more appropriately they are subtypes of the `object` type.

This object wrapper can further be observed with:

```
console.log( a );
```

The output of that statement varies depending on your browser, as developer consoles are free to choose however they feel it's appropriate to serialize the object for developer inspection.

Note: At the time of writing, the latest Chrome prints something like this: `String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`. But older versions of Chrome used to just print this: `String {0: "a", 1: "b", 2: "c"}`. The latest Firefox currently prints `String ["a","b","c"]`, but used to print `"abc"` in italics, which was clickable to open the object inspector. Of course, these results are subject to rapid change and your experience may vary.

The point is, `new String("abc")` creates a string wrapper object around `"abc"`, not just the primitive `"abc"` value itself.

Internal `[[Class]]`

Values that are `typeof "object"` (such as an array) are additionally tagged with an internal `[[Class]]` property (think of this more as an internal *class*ification rather than related to classes from traditional class-oriented coding). This property cannot be accessed directly, but can generally be revealed indirectly by borrowing the default `Object.prototype.toString(..)` method called against the value. For example:

```
Object.prototype.toString.call( [1,2,3] );                    // "[object Array]"

Object.prototype.toString.call( /regex-literal/i );     // "[object RegExp]"
```

So, for the array in this example, the internal `[[Class]]` value is `"Array"`, and for the regular expression, it's `"RegExp"`. In most cases, this internal `[[Class]]` value corresponds to the built-in native constructor (see below) that's related to the value, but that's not always the case.

What about primitive values? First, `null` and `undefined`:

```
Object.prototype.toString.call( null );                // "[object Null]"
Object.prototype.toString.call( undefined );    // "[object Undefined]"
```

You'll note that there are no `Null()` or `Undefined()` native constructors, but nevertheless the `"Null"` and `"Undefined"` are the internal `[[Class]]` values exposed.

But for the other simple primitives like `string`, `number`, and `boolean`, another behavior actually kicks in, which is usually called "boxing" (see "Boxing Wrappers" section next):

```
Object.prototype.toString.call( "abc" );        // "[object String]"
Object.prototype.toString.call( 42 );           // "[object Number]"
Object.prototype.toString.call( true );         // "[object Boolean]"
```

In this snippet, each of the simple primitives are automatically boxed by their respective object wrappers, which is why `"String"`, `"Number"`, and `"Boolean"` are revealed as the respective internal `[[Class]]` values.

Note: The behavior of `toString()` and `[[Class]]` as illustrated here has changed a bit from ES5 to ES6, but we cover those details in the *ES6 & Beyond* title of this series.

Boxing Wrappers

These object wrappers serve a very important purpose. Primitive values don't have properties or methods, so to access `.length` or `.toString()` you need an object wrapper around the value. Thankfully, JS will automatically *box* (aka wrap) the primitive value to fulfill such accesses.

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

So, if you're going to be accessing these properties/methods on your string values regularly, like a `i < a.length` condition in a `for` loop for instance, it might seem to make sense to just have the object form of the value from the start, so the JS engine doesn't need to implicitly create it for you.

But it turns out that's a bad idea. Browsers long ago performance-optimized the common cases like `.length`, which means your program will *actually go slower* if you try to "preoptimize" by directly using the object form (which isn't on the optimized path).

In general, there's basically no reason to use the object form directly. It's better to just let the boxing happen implicitly where necessary. In other words, never do things like `new String("abc")`, `new Number(42)`, etc -- always prefer using the literal primitive values `"abc"` and `42`.

Object Wrapper Gotchas

There are some gotchas with using the object wrappers directly that you should be aware of if you *do* choose to ever use them.

For example, consider `Boolean` wrapped values:

```
var a = new Boolean( false );

if (!a) {
        console.log( "Oops" ); // never runs
}
```

The problem is that you've created an object wrapper around the `false` value, but objects themselves are "truthy" (see Chapter 4), so using the object behaves oppositely to using the underlying `false` value itself, which is quite contrary to normal expectation.

If you want to manually box a primitive value, you can use the `Object(..)` function (no `new` keyword):

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

Again, using the boxed object wrapper directly (like `b` and `c` above) is usually discouraged, but there may be some rare occasions you'll run into where they may be useful.

Unboxing

If you have an object wrapper and you want to get the underlying primitive value out, you can use the `valueOf()` method:

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

Unboxing can also happen implicitly, when using an object wrapper value in a way that requires the primitive value. This process (coercion) will be covered in more detail in Chapter 4, but briefly:

```
var a = new String( "abc" );
var b = a + ""; // `b` has the unboxed primitive value "abc"

typeof a; // "object"
typeof b; // "string"
```

Natives as Constructors

For `array`, `object`, `function`, and regular-expression values, it's almost universally preferred that you use the literal form for creating the values, but the literal form creates the same sort of object as the constructor form does (that is, there is no nonwrapped value).

Just as we've seen above with the other natives, these constructor forms should generally be avoided, unless you really know you need them, mostly because they introduce exceptions and gotchas that you probably don't really *want* to deal with.

`Array(..)`

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

Note: The `Array(..)` constructor does not require the `new` keyword in front of it. If you omit it, it will behave as if you have used it anyway. So `Array(1,2,3)` is the same outcome as `new Array(1,2,3)`.

The `Array` constructor has a special form where if only one `number` argument is passed, instead of providing that value as *contents* of the array, it's taken as a length to "presize the array" (well, sorta).

This is a terrible idea. Firstly, you can trip over that form accidentally, as it's easy to forget.

But more importantly, there's no such thing as actually presizing the array. Instead, what you're creating is an otherwise empty array, but setting the `length` property of the array to the numeric value specified.

An array that has no explicit values in its slots, but has a `length` property that *implies* the slots exist, is a weird exotic type of data structure in JS with some very strange and confusing behavior. The capability to create such a value comes purely from old, deprecated, historical functionalities ("array-like objects" like the `arguments` object).

Note: An array with at least one "empty slot" in it is often called a "sparse array."

It doesn't help matters that this is yet another example where browser developer consoles vary on how they represent such an object, which breeds more confusion.

For example:

```
var a = new Array( 3 );

a.length; // 3
a;
```

The serialization of `a` in Chrome is (at the time of writing): `[ undefined x 3 ]`. **This is really unfortunate.** It implies that there are three `undefined` values in the slots of this array, when in fact the slots do not exist (so-called "empty slots" -- also a bad name!).

To visualize the difference, try this:

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```

Note: As you can see with `c` in this example, empty slots in an array can happen after creation of the array. Changing the `length` of an array to go beyond its number of actually-defined slot values, you implicitly introduce empty slots. In fact, you could even call `delete b[1]` in the above snippet, and it would introduce an empty slot into the middle of `b`.

For `b` (in Chrome, currently), you'll find `[ undefined, undefined, undefined ]` as the serialization, as opposed to `[ undefined x 3 ]` for `a` and `c`. Confused? Yeah, so is everyone else.

Worse than that, at the time of writing, Firefox reports `[ , , , ]` for `a` and `c`. Did you catch why that's so confusing? Look closely. Three commas implies four slots, not three slots like we'd expect.

What!? Firefox puts an extra `,` on the end of their serialization here because as of ES5, trailing commas in lists (array values, property lists, etc.) are allowed (and thus dropped and ignored). So if you were to type in a `[ , , , ]` value into your program or the console, you'd actually get the underlying value that's like `[ , , ]` (that is, an array with three empty slots). This choice, while confusing if reading the developer console, is defended as instead making copy-n-paste behavior accurate.

If you're shaking your head or rolling your eyes about now, you're not alone! Shrugs.

Unfortunately, it gets worse. More than just confusing console output, `a` and `b` from the above code snippet actually behave the same in some cases **but differently in others**:

```
a.join( "-" ); // "--"
b.join( "-" ); // "--"

a.map(function(v,i){ return i; }); // [ undefined x 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

Ugh.

The `a.map(..)` call *fails* because the slots don't actually exist, so `map(..)` has nothing to iterate over. `join(..)` works differently. Basically, we can think of it implemented sort of like this:

```
function fakeJoin(arr,connector) {
        var str = "";
        for (var i = 0; i < arr.length; i++) {
                if (i > 0) {
                        str += connector;
                }
                if (arr[i] !== undefined) {
                        str += arr[i];
                }
        }
        return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"
```

As you can see, `join(..)` works by just *assuming* the slots exist and looping up to the `length` value. Whatever `map(..)` does internally, it (apparently) doesn't make such an assumption, so the result from the strange "empty slots" array is unexpected and likely to cause failure.

So, if you wanted to *actually* create an array of actual `undefined` values (not just "empty slots"), how could you do it (besides manually)?

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

Confused? Yeah. Here's roughly how it works.

`apply(..)` is a utility available to all functions, which calls the function it's used with but in a special way.

The first argument is a `this` object binding (covered in the *this & Object Prototypes* title of this series), which we don't care about here, so we set it to `null`. The second argument is supposed to be an array (or something *like* an array -- aka an "array-like object"). The contents of this "array" are "spread" out as arguments to the function in question.

So, `Array.apply(..)` is calling the `Array(..)` function and spreading out the values (of the `{ length: 3 }` object value) as its arguments.

Inside of `apply(..)`, we can envision there's another `for` loop (kinda like `join(..)` from above) that goes from `0` up to, but not including, `length` (`3` in our case).

For each index, it retrieves that key from the object. So if the array-object parameter was named `arr` internally inside of the `apply(..)` function, the property access would effectively be `arr[0]`, `arr[1]`, and `arr[2]`. Of course, none of those properties exist on the `{ length: 3 }` object value, so all three of those property accesses would return the value `undefined`.

In other words, it ends up calling `Array(..)` basically like this: `Array(undefined,undefined,undefined)`, which is how we end up with an array filled with `undefined` values, and not just those (crazy) empty slots.

While `Array.apply( null, { length: 3 } )` is a strange and verbose way to create an array filled with `undefined` values, it's **vastly** better and more reliable than what you get with the footgun'ish `Array(3)` empty slots.

Bottom line: **never ever, under any circumstances**, should you intentionally create and use these exotic empty-slot arrays. Just don't do it. They're nuts.

`Object(..)`, `Function(..)`, and `RegExp(..)`

The `Object(..)` / `Function(..)` / `RegExp(..)` constructors are also generally optional (and thus should usually be avoided unless specifically called for):

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; };
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

There's practically no reason to ever use the `new Object()` constructor form, especially since it forces you to add properties one-by-one instead of many at once in the object literal form.

The `Function` constructor is helpful only in the rarest of cases, where you need to dynamically define a function's parameters and/or its function body. **Do not just treat `Function(..)` as an alternate form of `eval(..)`.** You will almost never need to dynamically define a function in this way.

Regular expressions defined in the literal form (`/^a*b+/g`) are strongly preferred, not just for ease of syntax but for performance reasons -- the JS engine precompiles and caches them before code execution. Unlike the other constructor forms we've seen so far, `RegExp(..)` has some reasonable utility: to dynamically define the pattern for a regular expression.

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?:" + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

This kind of scenario legitimately occurs in JS programs from time to time, so you'd need to use the `new RegExp("pattern","flags")` form.

`Date(..)` and `Error(..)`

The `Date(..)` and `Error(..)` native constructors are much more useful than the other natives, because there is no literal form for either.

To create a date object value, you must use `new Date()`. The `Date(..)` constructor accepts optional arguments to specify the date/time to use, but if omitted, the current date/time is assumed.

By far the most common reason you construct a date object is to get the current timestamp value (a signed integer number of milliseconds since Jan 1, 1970). You can do this by calling `getTime()` on a date object instance.

But an even easier way is to just call the static helper function defined as of ES5: `Date.now()`. And to polyfill that for pre-ES5 is pretty easy:

```
if (!Date.now) {
        Date.now = function(){
                return (new Date()).getTime();
        };
}
```

Note: If you call `Date()` without `new`, you'll get back a string representation of the date/time at that moment. The exact form of this representation is not specified in the language spec, though browsers tend to agree on something close to: `"Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)"`.

The `Error(..)` constructor (much like `Array()` above) behaves the same with the `new` keyword present or omitted.

The main reason you'd want to create an error object is that it captures the current execution stack context into the object (in most JS engines, revealed as a read-only `.stack` property once constructed). This stack context includes the function call-stack and the line-number where the error object was created, which makes debugging that error much easier.

You would typically use such an error object with the `throw` operator:

```
function foo(x) {
        if (!x) {
                throw new Error( "x wasn't provided" );
        }
        // ..
}
```

Error object instances generally have at least a `message` property, and sometimes other properties (which you should treat as read-only), like `type`. However, other than inspecting the above-mentioned `stack` property, it's usually best to just call `toString()` on the error object (either explicitly, or implicitly through coercion -- see Chapter 4) to get a friendly-formatted error message.

Tip: Technically, in addition to the general `Error(..)` native, there are several other specific-error-type natives: `EvalError(..)`, `RangeError(..)`, `ReferenceError(..)`, `SyntaxError(..)`, `TypeError(..)`, and `URIError(..)`. But it's very rare to manually use these specific error natives. They are automatically used if your program actually suffers from a real exception (such as referencing an undeclared variable and getting a `ReferenceError` error).

`Symbol(..)`

New as of ES6, an additional primitive value type has been added, called "Symbol". Symbols are special "unique" (not strictly guaranteed!) values that can be used as properties on objects with little fear of any collision. They're primarily designed for special built-in behaviors of ES6 constructs, but you can also define your own symbols.

Symbols can be used as property names, but you cannot see or access the actual value of a symbol from your program, nor from the developer console. If you evaluate a symbol in the developer console, what's shown looks like `Symbol(Symbol.create)`, for example.

There are several predefined symbols in ES6, accessed as static properties of the `Symbol` function object, like `Symbol.create`, `Symbol.iterator`, etc. To use them, do something like:

```
obj[Symbol.iterator] = function(){ /*..*/ };
```

To define your own custom symbols, use the `Symbol(..)` native. The `Symbol(..)` native "constructor" is unique in that you're not allowed to use `new` with it, as doing so will throw an error.

```
var mysym = Symbol( "my own symbol" );
mysym;                          // Symbol(my own symbol)
mysym.toString();       // "Symbol(my own symbol)"
typeof mysym;           // "symbol"
```

```
var a = { };
a[mysym] = "foobar";

Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]
```

While symbols are not actually private (`Object.getOwnPropertySymbols(..)` reflects on the object and reveals the symbols quite publicly), using them for private or special properties is likely their primary use-case. For most developers, they may take the place of property names with `_` underscore prefixes, which are almost always by convention signals to say, "hey, this is a private/special/internal property, so leave it alone!"

Note: `Symbol`s are *not* `object`s, they are simple scalar primitives.

Native Prototypes

Each of the built-in native constructors has its own `.prototype` object -- `Array.prototype`, `String.prototype`, etc.

These objects contain behavior unique to their particular object subtype.

For example, all string objects, and by extension (via boxing) `string` primitives, have access to default behavior as methods defined on the `String.prototype` object.

Note: By documentation convention, `String.prototype.XYZ` is shortened to `String#XYZ`, and likewise for all the other `.prototype`s.

- `String#indexOf(..)`: find the position in the string of another substring
- `String#charAt(..)`: access the character at a position in the string
- `String#substr(..)`, `String#substring(..)`, and `String#slice(..)`: extract a portion of the string as a new string
- `String#toUpperCase()` and `String#toLowerCase()`: create a new string that's converted to either uppercase or lowercase
- `String#trim()`: create a new string that's stripped of any trailing or leading whitespace

None of the methods modify the string *in place*. Modifications (like case conversion or trimming) create a new value from the existing value.

By virtue of prototype delegation (see the *this & Object Prototypes* title in this series), any string value can access these methods:

```
var a = " abc ";

a.indexOf( "c" ); // 3
a.toUpperCase(); // " ABC "
a.trim(); // "abc"
```

The other constructor prototypes contain behaviors appropriate to their types, such as `Number#toFixed(..)` (stringifying a number with a fixed number of decimal digits) and `Array#concat(..)` (merging arrays). All functions have access to `apply(..)`, `call(..)`, and `bind(..)` because `Function.prototype` defines them.

But, some of the native prototypes aren't *just* plain objects:

```
typeof Function.prototype;                      // "function"
Function.prototype();                           // it's an empty function!

RegExp.prototype.toString();            // "/(?:)/" -- empty regex
"abc".match( RegExp.prototype );        // [""]
```

A particularly bad idea, you can even modify these native prototypes (not just adding properties as you're probably familiar with):

```
Array.isArray( Array.prototype );       // true
Array.prototype.push( 1, 2, 3 );        // 3
Array.prototype;                                        // [1,2,3]

// don't leave it that way, though, or expect weirdness!
```

```
// reset the `Array.prototype` to empty
Array.prototype.length = 0;
```

As you can see, `Function.prototype` is a function, `RegExp.prototype` is a regular expression, and `Array.prototype` is an array. Interesting and cool, huh?

Prototypes As Defaults

`Function.prototype` being an empty function, `RegExp.prototype` being an "empty" (e.g., non-matching) regex, and `Array.prototype` being an empty array, make them all nice "default" values to assign to variables if those variables wouldn't already have had a value of the proper type.

For example:

```
function isThisCool(vals,fn,rx) {
        vals = vals || Array.prototype;
        fn = fn || Function.prototype;
        rx = rx || RegExp.prototype;

        return rx.test(
                vals.map( fn ).join( "" )
        );
}

isThisCool();           // true

isThisCool(
        ["a","b","c"],
        function(v){ return v.toUpperCase(); },
        /D/
);                                      // false
```

Note: As of ES6, we don't need to use the `vals = vals || ..` default value syntax trick (see Chapter 4) anymore, because default values can be set for parameters via native syntax in the function declaration (see Chapter 5).

One minor side-benefit of this approach is that the `.prototype`s are already created and built-in, thus created *only once*. By contrast, using `[]`, `function(){}`, and `/(?:)/` values themselves for those defaults would (likely, depending on engine implementations) be recreating those values (and probably garbage-collecting them later) for *each call* of `isThisCool(..)`. That could be memory/CPU wasteful.

Also, be very careful not to use `Array.prototype` as a default value **that will subsequently be modified**. In this example, `vals` is used read-only, but if you were to instead make in-place changes to `vals`, you would actually be modifying `Array.prototype` itself, which would lead to the gotchas mentioned earlier!

Note: While we're pointing out these native prototypes and some usefulness, be cautious of relying on them and even more wary of modifying them in any way. See Appendix A "Native Prototypes" for more discussion.

Review

JavaScript provides object wrappers around primitive values, known as natives (`String`, `Number`, `Boolean`, etc). These object wrappers give the values access to behaviors appropriate for each object subtype (`String#trim()` and `Array#concat(..)`).

If you have a simple scalar primitive value like `"abc"` and you access its `length` property or some `String.prototype` method, JS automatically "boxes" the value (wraps it in its respective object wrapper) so that the property/method accesses can be fulfilled.