You Don't Know JS: *this* & Object Prototypes

Chapter 3: Objects

In Chapters 1 and 2, we explained how the `this` binding points to various objects depending on the call-site of the function invocation. But what exactly are objects, and why do we need to point to them? We will explore objects in detail in this chapter.

Syntax

Objects come in two forms: the declarative (literal) form, and the constructed form.

The literal syntax for an object looks like this:

```
var myObj = {
    key: value
    // ...
};
```

The constructed form looks like this:

```
var myObj = new Object();
myObj.key = value;
```

The constructed form and the literal form result in exactly the same sort of object. The only difference really is that you can add one or more key/value pairs to the literal declaration, whereas with constructed-form objects, you must add the properties one-by-one.

Note: It's extremely uncommon to use the "constructed form" for creating objects as just shown. You would pretty much always want to use the literal syntax form. The same will be true of most of the built-in objects (see below).

Type

Objects are the general building block upon which much of JS is built. They are one of the 6 primary types (called "language types" in the specification) in JS:

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `object`

Note that the *simple primitives* (`string` , `number` , `boolean` , `null` , and `undefined`) are **not** themselves `objects` . `null` is sometimes referred to as an object type, but this misconception stems from a bug in the language which causes `typeof null` to return the string `"object"` incorrectly (and confusingly). In fact, `null` is its own primitive type.

It's a common mis-statement that "everything in JavaScript is an object". This is clearly not true.

By contrast, there *are* a few special object sub-types, which we can refer to as *complex primitives*.

`function` is a sub-type of object (technically, a "callable object"). Functions in JS are said to be "first class" in that they are basically just normal objects (with callable behavior semantics bolted on), and so they can be handled like any other plain object.

Arrays are also a form of objects, with extra behavior. The organization of contents in arrays is slightly more structured than for general objects.

Built-in Objects

There are several other object sub-types, usually referred to as built-in objects. For some of them, their names seem to imply they are directly related to their simple primitives counter-parts, but in fact, their relationship is more complicated, which we'll explore shortly.

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

These built-ins have the appearance of being actual types, even classes, if you rely on the similarity to other languages such as Java's `String` class.

But in JS, these are actually just built-in functions. Each of these built-in functions can be used as a constructor (that is, a function call with the `new` operator -- see Chapter 2), with the result being a newly *constructed* object of the sub-type in question. For instance:

```
var strPrimitive = "I am a string";
typeof strPrimitive;                            // "string"
strPrimitive instanceof String;                 // false

var strObject = new String( "I am a string" );
typeof strObject;                               // "object"
strObject instanceof String;                    // true

// inspect the object sub-type
Object.prototype.toString.call( strObject );    // [object String]
```

We'll see in detail in a later chapter exactly how the `Object.prototype.toString...` bit works, but briefly, we can inspect the internal sub-type by borrowing the base default `toString()` method, and you can see it reveals that `strObject` is an object that was in fact created by the `String` constructor.

The primitive value `"I am a string"` is not an object, it's a primitive literal and immutable value. To perform operations on it, such as checking its length, accessing its individual character contents, etc, a `String` object is required.

Luckily, the language automatically coerces a `"string"` primitive to a `String` object when necessary, which means you almost never need to explicitly create the Object form. It is **strongly preferred** by the majority of the JS community to use the literal form for a value, where possible, rather than the constructed object form.

Consider:

```
var strPrimitive = "I am a string";

console.log( strPrimitive.length );         // 13

console.log( strPrimitive.charAt( 3 ) );    // "m"
```

In both cases, we call a property or method on a string primitive, and the engine automatically coerces it to a `String` object, so that the property/method access works.

The same sort of coercion happens between the number literal primitive `42` and the `new Number(42)` object wrapper, when using methods like `42.359.toFixed(2)`. Likewise for `Boolean` objects from `"boolean"` primitives.

`null` and `undefined` have no object wrapper form, only their primitive values. By contrast, `Date` values can *only* be created with their constructed object form, as they have no literal form counter-part.

`Object`s, `Array`s, `Function`s, and `RegExp`s (regular expressions) are all objects regardless of whether the literal or constructed form is used. The constructed form does offer, in some cases, more options in creation than the literal form counterpart. Since objects are created either way, the simpler literal form is almost universally preferred. **Only use the constructed form if you need the extra options.**

`Error` objects are rarely created explicitly in code, but usually created automatically when exceptions are thrown. They can be created with the constructed form `new Error(..)`, but it's often unnecessary.

Contents

As mentioned earlier, the contents of an object consist of values (any type) stored at specifically named *locations*, which we call properties.

It's important to note that while we say "contents" which implies that these values are *actually* stored inside the object, that's merely an appearance. The engine stores values in implementation-dependent ways, and may very well not store them *in* some object container. What *is* stored in the container are these property names, which act as pointers (technically, *references*) to where the values are stored.

Consider:

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

myObject.a;     // 2

myObject["a"];  // 2
```

To access the value at the *location* `a` in `myObject`, we need to use either the `.` operator or the `[ ]` operator. The `.a` syntax is usually referred to as "property" access, whereas the `["a"]` syntax is usually referred to as "key" access. In reality, they both access the same *location*, and will pull out the same value, `2`, so the terms can be used interchangeably. We will use the most common term, "property access" from here on.

The main difference between the two syntaxes is that the `.` operator requires an `Identifier` compatible property name after it, whereas the `[".."]` syntax can take basically any UTF-8/unicode compatible string as the name for the property. To reference a property of the name "Super-Fun!", for instance, you would have to use the `["Super-Fun!"]` access syntax, as `Super-Fun!` is not a valid `Identifier` property

name.

Also, since the `["..."]` syntax uses a string's **value** to specify the location, this means the program can programmatically build up the value of the string, such as:

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

var idx;

if (wantA) {
    idx = "a";
}

// later

console.log( myObject[idx] ); // 2
```

In objects, property names are **always** strings. If you use any other value besides a `string` (primitive) as the property, it will first be converted to a string. This even includes numbers, which are commonly used as array indexes, so be careful not to confuse the use of numbers between objects and arrays.

```
var myObject = { };

myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"];               // "foo"
myObject["3"];                  // "bar"
myObject["[object Object]"];    // "baz"
```

Computed Property Names

The `myObject[..]` property access syntax we just described is useful if you need to use a computed expression value *as* the key name, like `myObject[prefix + name]`. But that's not really helpful when declaring objects using the object-literal syntax.

ES6 adds *computed property names*, where you can specify an expression, surrounded by a `[ ]` pair, in the key-name position of an object-literal declaration:

```
var prefix = "foo";

var myObject = {
    [prefix + "bar"]: "hello",
    [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

The most common usage of *computed property names* will probably be for ES6 `Symbol`s, which we will not be covering in detail in this book. In short, they're a new primitive data type which has an opaque unguessable value (technically a `string` value). You will be strongly discouraged from working with the *actual value* of a `Symbol` (which can theoretically be different between different JS engines), so the name of the `Symbol`, like `Symbol.Something` (just a made up name!), will be what you use:

```
var myObject = {
    [Symbol.Something]: "hello world"
};
```

Property vs. Method

Some developers like to make a distinction when talking about a property access on an object, if the value being accessed happens to be a function. Because it's tempting to think of the function as *belonging* to the object, and in other languages, functions which belong to objects (aka, "classes") are referred to as "methods", it's not uncommon to hear, "method access" as opposed to "property access".

The specification makes this same distinction, interestingly.

Technically, functions never "belong" to objects, so saying that a function that just happens to be accessed on an object reference is automatically a "method" seems a bit of a stretch of semantics.

It *is* true that some functions have `this` references in them, and that *sometimes* these `this` references refer to the object reference at the call-site. But this usage really does not make that function any more a "method" than any other function, as `this` is dynamically bound at run-time, at the call-site, and thus its relationship to the object is indirect, at best.

Every time you access a property on an object, that is a **property access**, regardless of the type of value you get back. If you *happen* to get a function from that property access, it's not magically a "method" at that point. There's nothing special (outside of possible implicit `this` binding as explained earlier) about a function that comes from a property access.

For instance:

```
function foo() {
    console.log( "foo" );
}

var someFoo = foo;  // variable reference to `foo`

var myObject = {
    someFoo: foo
};

foo;                // function foo(){..}

someFoo;            // function foo(){..}

myObject.someFoo;   // function foo(){..}
```

`someFoo` and `myObject.someFoo` are just two separate references to the same function, and neither implies anything about the function being special or "owned" by any other object. If `foo()` above was defined to have a `this` reference inside it, that `myObject.someFoo` *implicit binding* would be the **only** observable difference between the two references. Neither reference really makes sense to be called a "method".

Perhaps one could argue that a function *becomes a method*, not at definition time, but during run-time just for that invocation, depending on how it's called at its call-site (with an object reference context or not -- see Chapter 2 for more details). Even this interpretation is a bit of a stretch.

The safest conclusion is probably that "function" and "method" are interchangeable in JavaScript.

Note: ES6 adds a `super` reference, which is typically going to be used with `class` (see Appendix A). The way `super` behaves (static binding rather than late binding as `this`) gives further weight to the idea that a function which is `super` bound somewhere is more a "method" than "function". But again, these are just subtle semantic (and mechanical) nuances.

Even when you declare a function expression as part of the object-literal, that function doesn't magically *belong* more to the object -- still just multiple references to the same function object:

```
var myObject = {
    foo: function foo() {
        console.log( "foo" );
    }
};

var someFoo = myObject.foo;

someFoo;        // function foo(){..}

myObject.foo;   // function foo(){..}
```

Note: In Chapter 6, we will cover an ES6 short-hand for that `foo: function foo(){ .. }` declaration syntax in our object-literal.

Arrays

Arrays also use the `[ ]` access form, but as mentioned above, they have slightly more structured organization for how and where values are stored (though still no restriction on what *type* of values are stored). Arrays assume *numeric indexing*, which means that values are stored in locations, usually called *indices*, at non-negative integers, such as `0` and `42`.

```
var myArray = [ "foo", 42, "bar" ];

myArray.length;     // 3

myArray[0];         // "foo"

myArray[2];         // "bar"
```

Arrays *are* objects, so even though each index is a positive integer, you can *also* add properties onto the array:

```
var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";

myArray.length; // 3

myArray.baz;    // "baz"
```

Notice that adding named properties (regardless of `.` or `[ ]` operator syntax) does not change the reported `length` of the array.

You *could* use an array as a plain key/value object, and never add any numeric indices, but this is a bad idea because arrays have behavior and optimizations specific to their intended use, and likewise with plain objects. Use objects to store key/value pairs, and arrays to store values at numeric indices.

Be careful: If you try to add a property to an array, but the property name *looks* like a number, it will end up instead as a numeric index (thus modifying the array contents):

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length; // 4

myArray[3];     // "baz"
```

Duplicating Objects

One of the most commonly requested features when developers newly take up the JavaScript language is how to duplicate an object. It would seem like there should just be a built-in `copy()` method, right? It turns out that it's a little more complicated than that, because it's not fully clear what, by default, should be the algorithm for the duplication.

For example, consider this object:

```
function anotherFunction() { /*..*/ }

var anotherObject = {
    c: true
};

var anotherArray = [];

var myObject = {
    a: 2,
    b: anotherObject,   // reference, not a copy!
    c: anotherArray,    // another reference!
    d: anotherFunction
};

anotherArray.push( anotherObject, myObject );
```

What exactly should be the representation of a *copy* of `myObject` ?

Firstly, we should answer if it should be a *shallow* or *deep* copy? A *shallow copy* would end up with `a` on the new object as a copy of the value `2`, but `b`, `c`, and `d` properties as just references to the same places as the references in the original object. A *deep copy* would duplicate not only `myObject`, but `anotherObject` and `anotherArray`. But then we have issues that `anotherArray` has references to `anotherObject` and `myObject` in it, so *those* should also be duplicated rather than reference-preserved. Now we have an infinite circular duplication problem because of the circular reference.

Should we detect a circular reference and just break the circular traversal (leaving the deep element not fully duplicated)? Should we error out completely? Something in between?

Moreover, it's not really clear what "duplicating" a function would mean? There are some hacks like pulling out the `toString()` serialization of a function's source code (which varies across implementations and is not even reliable in all engines depending on the type of function being inspected).

So how do we resolve all these tricky questions? Various JS frameworks have each picked their own interpretations and made their own decisions. But which of these (if any) should JS adopt as *the* standard? For a long time, there was no clear answer.

One subset solution is that objects which are JSON-safe (that is, can be serialized to a JSON string and then re-parsed to an object with the same structure and values) can easily be *duplicated* with:

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

Of course, that requires you to ensure your object is JSON safe. For some situations, that's trivial. For others, it's insufficient.

At the same time, a shallow copy is fairly understandable and has far less issues, so ES6 has now defined `Object.assign(..)` for this task. `Object.assign(..)` takes a *target* object as its first parameter, and one or more *source* objects as its subsequent parameters. It iterates over all the *enumerable* (see below), *owned keys* (**immediately present**) on the *source* object(s) and copies them (via `=` assignment only) to *target*. It also, helpfully, returns *target*, as you can see below:

```
var newObj = Object.assign( {}, myObject );

newObj.a;                       // 2
newObj.b === anotherObject;     // true
newObj.c === anotherArray;      // true
newObj.d === anotherFunction;   // true
```

Note: In the next section, we describe "property descriptors" (property characteristics) and show the use of `Object.defineProperty(..)`. The duplication that occurs for `Object.assign(..)` however is

purely `=` style assignment, so any special characteristics of a property (like `writable`) on a source object **are not preserved** on the target object.

Property Descriptors

Prior to ES5, the JavaScript language gave no direct way for your code to inspect or draw any distinction between the characteristics of properties, such as whether the property was read-only or not.

But as of ES5, all properties are described in terms of a **property descriptor**.

Consider this code:

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

Object.getOwnPropertyDescriptor( myObject, "a" );
// {
//    value: 2,
//    writable: true,
//    enumerable: true,
//    configurable: true
// }
```

As you can see, the property descriptor (called a "data descriptor" since it's only for holding a data value) for our normal object property `a` is much more than just its `value` of `2`. It includes 3 other characteristics: `writable`, `enumerable`, and `configurable`.

While we can see what the default values for the property descriptor characteristics are when we create a normal property, we can use `Object.defineProperty(..)` to add a new property, or modify an existing one (if it's `configurable`!), with the desired characteristics.

For example:

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );

myObject.a; // 2
```

Using `defineProperty(..)`, we added the plain, normal `a` property to `myObject` in a manually explicit way. However, you generally wouldn't use this manual approach unless you wanted to modify one of the descriptor characteristics from its normal behavior.

Writable

The ability for you to change the value of a property is controlled by `writable`.

Consider:

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // not writable!
    configurable: true,
    enumerable: true
} );

myObject.a = 3;

myObject.a; // 2
```

As you can see, our modification of the `value` silently failed. If we try in `strict mode`, we get an error:

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // not writable!
    configurable: true,
    enumerable: true
} );

myObject.a = 3; // TypeError
```

The `TypeError` tells us we cannot change a non-writable property.

Note: We will discuss getters/setters shortly, but briefly, you can observe that `writable:false` means a value cannot be changed, which is somewhat equivalent to if you defined a no-op setter. Actually, your no-op setter would need to throw a `TypeError` when called, to be truly conformant to `writable:false`.

Configurable

As long as a property is currently configurable, we can modify its descriptor definition, using the same `defineProperty(..)` utility.

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

myObject.a = 3;
myObject.a;                 // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false,    // not configurable!
    enumerable: true
} );

myObject.a;                 // 4
myObject.a = 5;
myObject.a;                 // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

The final `defineProperty(..)` call results in a TypeError, regardless of `strict mode`, if you attempt to change the descriptor definition of a non-configurable property. Be careful: as you can see, changing `configurable` to `false` is a **one-way action, and cannot be undone!**

Note: There's a nuanced exception to be aware of: even if the property is already `configurable:false`, `writable` can always be changed from `true` to `false` without error, but not back to `true` if already `false`.

Another thing `configurable:false` prevents is the ability to use the `delete` operator to remove an existing property.

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

myObject.a;             // 2
delete myObject.a;
myObject.a;             // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a;             // 2
delete myObject.a;
myObject.a;             // 2
```

As you can see, the last `delete` call failed (silently) because we made the `a` property non-configurable.

`delete` is only used to remove object properties (which can be removed) directly from the object in question. If an object property is the last remaining *reference* to some object/function, and you `delete` it, that removes the reference and now that unreferenced object/function can be garbage collected. But, it is **not** proper to think of `delete` as a tool to free up allocated memory as it does in other languages (like C/C++). `delete` is just an object property removal operation -- nothing more.

Enumerable

The final descriptor characteristic we will mention here (there are two others, which we deal with shortly when we discuss getter/setters) is `enumerable`.

The name probably makes it obvious, but this characteristic controls if a property will show up in certain object-property enumerations, such as the `for..in` loop. Set to `false` to keep it from showing up in such enumerations, even though it's still completely accessible. Set to `true` to keep it present.

All normal user-defined properties are defaulted to `enumerable`, as this is most commonly what you want. But if you have a special property you want to hide from enumeration, set it to `enumerable:false`.

We'll demonstrate enumerability in much more detail shortly, so keep a mental bookmark on this topic.

Immutability

It is sometimes desired to make properties or objects that cannot be changed (either by accident or intentionally). ES5 adds support for handling that in a variety of different nuanced ways.

It's important to note that **all** of these approaches create shallow immutability. That is, they affect only the object and its direct property characteristics. If an object has a reference to another object (array, object, function, etc), the *contents* of that object are not affected, and remain mutable.

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

We assume in this snippet that `myImmutableObject` is already created and protected as immutable. But, to also protect the contents of `myImmutableObject.foo` (which is its own object -- array), you would also need to make `foo` immutable, using one or more of the following functionalities.

Note: It is not terribly common to create deeply entrenched immutable objects in JS programs. Special cases can certainly call for it, but as a general design pattern, if you find yourself wanting to *seal* or *freeze* all your objects, you may want to take a step back and reconsider your program design to be more robust to potential changes in objects' values.

Object Constant

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

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );
```

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).

You could "deep freeze" an object by calling `Object.freeze(..)` on the object, and then recursively iterating over all objects it references (which would have been unaffected thus far), and calling `Object.freeze(..)` on them as well. Be careful, though, as that could affect other (shared) objects you're not intending to affect.

`[[Get]]`

There's a subtle, but important, detail about how property accesses are performed.

Consider:

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

myObject.a; // 2
```

The `myObject.a` is a property access, but it doesn't *just* look in `myObject` for a property of the name `a`, as it might seem.

According to the spec, the code above actually performs a `[[Get]]` operation (kinda like a function call: `[[Get]]()`) on the `myObject`. The default built-in `[[Get]]` operation for an object *first* inspects the object for a property of the requested name, and if it finds it, it will return the value accordingly.

However, the `[[Get]]` algorithm defines other important behavior if it does *not* find a property of the requested name. We will examine in Chapter 5 what happens *next* (traversal of the `[[Prototype]]` chain, if any).

But one important result of this `[[Get]]` operation is that if it cannot through any means come up with a value for the requested property, it instead returns the value `undefined`.

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

myObject.b; // undefined
```

This behavior is different from when you reference *variables* by their identifier names. If you reference a variable that cannot be resolved within the applicable lexical scope look-up, the result is not `undefined` as it is for object properties, but instead a `ReferenceError` is thrown.

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

myObject.a; // undefined

myObject.b; // undefined
```

From a *value* perspective, there is no difference between these two references -- they both result in `undefined`. However, the `[[Get]]` operation underneath, though subtle at a glance, potentially performed a bit more "work" for the reference `myObject.b` than for the reference `myObject.a`.

Inspecting only the value results, you cannot distinguish whether a property exists and holds the explicit value `undefined`, or whether the property does *not* exist and `undefined` was the default return value after `[[Get]]` failed to return something explicitly. However, we will see shortly how you *can* distinguish these two scenarios.

`[[Put]]`

Since there's an internally defined `[[Get]]` operation for getting a value from a property, it should be obvious there's also a default `[[Put]]` operation.

It may be tempting to think that an assignment to a property on an object would just invoke `[[Put]]` to set or create that property on the object in question. But the situation is more nuanced than that.

When invoking `[[Put]]`, how it behaves differs based on a number of factors, including (most impactfully) whether the property is already present on the object or not.

If the property is present, the `[[Put]]` algorithm will roughly check:

1. Is the property an accessor descriptor (see "Getters & Setters" section below)? **If so, call the setter, if any.**
2. Is the property a data descriptor with `writable` of `false`? **If so, silently fail in `non-strict mode`, or throw `TypeError` in `strict mode`.**
3. Otherwise, set the value to the existing property as normal.

If the property is not yet present on the object in question, the `[[Put]]` operation is even more nuanced and complex. We will revisit this scenario in Chapter 5 when we discuss `[[Prototype]]` to give it more clarity.

Getters & Setters

The default `[[Put]]` and `[[Get]]` operations for objects completely control how values are set to existing or new properties, or retrieved from existing properties, respectively.

Note: Using future/advanced capabilities of the language, it may be possible to override the default `[[Get]]` or `[[Put]]` operations for an entire object (not just per property). This is beyond the scope of our discussion in this book, but will be covered later in the "You Don't Know JS" series.

ES5 introduced a way to override part of these default operations, not on an object level but a per-property level, through the use of getters and setters. Getters are properties which actually call a hidden function to retrieve a value. Setters are properties which actually call a hidden function to set a value.

When you define a property to have either a getter or a setter or both, its definition becomes an "accessor

descriptor" (as opposed to a "data descriptor"). For accessor-descriptors, the `value` and `writable` characteristics of the descriptor are moot and ignored, and instead JS considers the `set` and `get` characteristics of the property (as well as `configurable` and `enumerable`).

Consider:

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,   // target
    "b",        // property name
    {           // descriptor
        // define a getter for `b`
        get: function(){ return this.a * 2 },

        // make sure `b` shows up as an object property
        enumerable: true
    }
);

myObject.a; // 2

myObject.b; // 4
```

Either through object-literal syntax with `get a() { .. }` or through explicit definition with `defineProperty(..)`, in both cases we created a property on the object that actually doesn't hold a value, but whose access automatically results in a hidden function call to the getter function, with whatever value it returns being the result of the property access.

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2
```

Since we only defined a getter for `a`, if we try to set the value of `a` later, the set operation won't throw an error but will just silently throw the assignment away. Even if there was a valid setter, our custom getter is hard-coded to return only `2`, so the set operation would be moot.

To make this scenario more sensible, properties should also be defined with setters, which override the default `[[Put]]` operation (aka, assignment), per-property, just as you'd expect. You will almost certainly want to always declare both getter and setter (having only one or the other often leads to unexpected/surprising behavior):

```
var myObject = {
    // define a getter for `a`
    get a() {
        return this._a_;
    },

    // define a setter for `a`
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

Note: In this example, we actually store the specified value `2` of the assignment (`[[Put]]` operation) into another variable `_a_`. The `_a_` name is purely by convention for this example and implies nothing special about its behavior -- it's a normal property like any other.

Existence

We showed earlier that a property access like `myObject.a` may result in an `undefined` value if either the explicit `undefined` is stored there or the `a` property doesn't exist at all. So, if the value is the same in both cases, how else do we distinguish them?

We can ask an object if it has a certain property *without* asking to get that property's value:

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

("a" in myObject);              // true
("b" in myObject);              // false

myObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "b" ); // false
```

The `in` operator will check to see if the property is *in* the object, or if it exists at any higher level of the `[[Prototype]]` chain object traversal (see Chapter 5). By contrast, `hasOwnProperty(..)` checks to see if *only* `myObject` has the property or not, and will *not* consult the `[[Prototype]]` chain. We'll come back to the important differences between these two operations in Chapter 5 when we explore `[[Prototype]]` s in detail.

`hasOwnProperty(..)` is accessible for all normal objects via delegation to `Object.prototype` (see Chapter 5). But it's possible to create an object that does not link to `Object.prototype` (via `Object.create(null)` -- see Chapter 5). In this case, a method call like `myObject.hasOwnProperty(..)` would fail.

In that scenario, a more robust way of performing such a check is `Object.prototype.hasOwnProperty.call(myObject,"a")`, which borrows the base `hasOwnProperty(..)` method and uses *explicit* `this` *binding* (see Chapter 2) to apply it against our `myObject`.

Note: The `in` operator has the appearance that it will check for the existence of a *value* inside a container, but it actually checks for the existence of a property name. This difference is important to note with respect to arrays, as the temptation to try a check like `4 in [2, 4, 6]` is strong, but this will not behave as expected.

Enumeration

Previously, we explained briefly the idea of "enumerability" when we looked at the `enumerable` property descriptor characteristic. Let's revisit that and examine it in more close detail.

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` NON-enumerable
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

// .......

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

You'll notice that `myObject.b` in fact **exists** and has an accessible value, but it doesn't show up in a `for..in` loop (though, surprisingly, it **is** revealed by the `in` operator existence check). That's because "enumerable" basically means "will be included if the object's properties are iterated through".

Note: `for..in` loops applied to arrays can give somewhat unexpected results, in that the enumeration of an array will include not only all the numeric indices, but also any enumerable properties. It's a good idea to use `for..in` loops *only* on objects, and traditional `for` loops with numeric index iteration for the values stored in arrays.

Another way that enumerable and non-enumerable properties can be distinguished:

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` non-enumerable
    { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

`propertyIsEnumerable(..)` tests whether the given property name exists *directly* on the object and is also `enumerable:true`.

`Object.keys(..)` returns an array of all enumerable properties, whereas `Object.getOwnPropertyNames(..)` returns an array of *all* properties, enumerable or not.

Whereas `in` vs. `hasOwnProperty(..)` differ in whether they consult the `[[Prototype]]` chain or not, `Object.keys(..)` and `Object.getOwnPropertyNames(..)` both inspect *only* the direct object specified.

There's (currently) no built-in way to get a list of **all properties** which is equivalent to what the `in` operator test would consult (traversing all properties on the entire `[[Prototype]]` chain, as explained in Chapter 5). You could approximate such a utility by recursively traversing the `[[Prototype]]` chain of an object, and for each level, capturing the list from `Object.keys(..)` -- only enumerable properties.

Iteration

The `for..in` loop iterates over the list of enumerable properties on an object (including its `[[Prototype]]` chain). But what if you instead want to iterate over the values?

With numerically-indexed arrays, iterating over the values is typically done with a standard `for` loop, like:

```
var myArray = [1, 2, 3];

for (var i = 0; i < myArray.length; i++) {
    console.log( myArray[i] );
}
// 1 2 3
```

This isn't iterating over the values, though, but iterating over the indices, where you then use the index to reference the value, as `myArray[i]`.

ES5 also added several iteration helpers for arrays, including `forEach(..)`, `every(..)`, and `some(..)`. Each of these helpers accepts a function callback to apply to each element in the array, differing only in they respectively respond to a return value from the callback.

`forEach(..)` will iterate over all values in the array, and ignores any callback return values. `every(..)` keeps going until the end *or* the callback returns a `false` (or "falsy") value, whereas `some(..)` keeps going until the end *or* the callback returns a `true` (or "truthy") value.

These special return values inside `every(..)` and `some(..)` act somewhat like a `break` statement inside a normal `for` loop, in that they stop the iteration early before it reaches the end.

If you iterate on an object with a `for..in` loop, you're also only getting at the values indirectly, because it's actually iterating only over the enumerable properties of the object, leaving you to access the properties manually to get the values.

Note: As contrasted with iterating over an array's indices in a numerically ordered way (`for` loop or other iterators), the order of iteration over an object's properties is **not guaranteed** and may vary between different JS engines. **Do not rely** on any observed ordering for anything that requires consistency among environments, as any observed agreement is unreliable.

But what if you want to iterate over the values directly instead of the array indices (or object properties)? Helpfully, ES6 adds a `for..of` loop syntax for iterating over arrays (and objects, if the object defines its own custom iterator):

```
var myArray = [ 1, 2, 3 ];

for (var v of myArray) {
    console.log( v );
}
// 1
// 2
// 3
```

The `for..of` loop asks for an iterator object (from a default internal function known as `@@iterator` in spec-speak) of the *thing* to be iterated, and the loop then iterates over the successive return values from calling that iterator object's `next()` method, once for each loop iteration.

Arrays have a built-in `@@iterator`, so `for..of` works easily on them, as shown. But let's manually iterate the array, using the built-in `@@iterator`, to see how it works:

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

Note: We get at the `@@iterator` *internal property* of an object using an ES6 `Symbol`: `Symbol.iterator`. We briefly mentioned `Symbol` semantics earlier in the chapter (see "Computed Property Names"), so the same reasoning applies here. You'll always want to reference such special properties by `Symbol` name reference instead of by the special value it may hold. Also, despite the name's implications, `@@iterator` is **not the iterator object** itself, but a **function that returns** the iterator object -- a subtle but important detail!

As the above snippet reveals, the return value from an iterator's `next()` call is an object of the form `{ value: .. , done: .. }`, where `value` is the current iteration value, and `done` is a `boolean` that indicates if there's more to iterate.

Notice the value `3` was returned with a `done:false`, which seems strange at first glance. You have to call the `next()` a fourth time (which the `for..of` loop in the previous snippet automatically does) to get `done:true` and know you're truly done iterating. The reason for this quirk is beyond the scope of what we'll discuss here, but it comes from the semantics of ES6 generator functions.

While arrays do automatically iterate in `for..of` loops, regular objects **do not have a built-in `@@iterator`**. The reasons for this intentional omission are more complex than we will examine here, but in

general it was better to not include some implementation that could prove troublesome for future types of objects.

It *is* possible to define your own default `@@iterator` for any object that you care to iterate over. For example:

```
var myObject = {
    a: 2,
    b: 3
};

Object.defineProperty( myObject, Symbol.iterator, {
    enumerable: false,
    writable: false,
    configurable: true,
    value: function() {
        var o = this;
        var idx = 0;
        var ks = Object.keys( o );
        return {
            next: function() {
                return {
                    value: o[ks[idx++]],
                    done: (idx > ks.length)
                };
            }
        };
    }
} );

// iterate `myObject` manually
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// iterate `myObject` with `for..of`
for (var v of myObject) {
    console.log( v );
}
// 2
// 3
```

Note: We used `Object.defineProperty(..)` to define our custom `@@iterator` (mostly so we could

make it non-enumerable), but using the `Symbol` as a *computed property name* (covered earlier in this chapter), we could have declared it directly, like `var myObject = { a:2, b:3, [Symbol.iterator]: function(){ /* .. */ } }`.

Each time the `for..of` loop calls `next()` on `myObject`'s iterator object, the internal pointer will advance and return back the next value from the object's properties list (see a previous note about iteration ordering on object properties/values).

The iteration we just demonstrated is a simple value-by-value iteration, but you can of course define arbitrarily complex iterations for your custom data structures, as you see fit. Custom iterators combined with ES6's `for..of` loop are a powerful new syntactic tool for manipulating user-defined objects.

For example, a list of `Pixel` objects (with `x` and `y` coordinate values) could decide to order its iteration based on the linear distance from the `(0,0)` origin, or filter out points that are "too far away", etc. As long as your iterator returns the expected `{ value: .. }` return values from `next()` calls, and a `{ done: true }` after the iteration is complete, ES6's `for..of` can iterate over it.

In fact, you can even generate "infinite" iterators which never "finish" and always return a new value (such as a random number, an incremented value, a unique identifier, etc), though you probably will not use such iterators with an unbounded `for..of` loop, as it would never end and would hang your program.

```
var randoms = {
    [Symbol.iterator]: function() {
        return {
            next: function() {
                return { value: Math.random() };
            }
        };
    }
};

var randoms_pool = [];
for (var n of randoms) {
    randoms_pool.push( n );

    // don't proceed unbounded!
    if (randoms_pool.length === 100) break;
}
```

This iterator will generate random numbers "forever", so we're careful to only pull out 100 values so our program doesn't hang.

Review (TL;DR)

Objects in JS have both a literal form (such as `var a = { .. }`) and a constructed form (such as `var a = new Array(..)`). The literal form is almost always preferred, but the constructed form offers, in some cases, more creation options.

Many people mistakingly claim "everything in JavaScript is an object", but this is incorrect. Objects are one of the 6 (or 7, depending on your perspective) primitive types. Objects have sub-types, including `function`, and also can be behavior-specialized, like `[object Array]` as the internal label representing the array object sub-type.

Objects are collections of key/value pairs. The values can be accessed as properties, via `.propName` or `["propName"]` syntax. Whenever a property is accessed, the engine actually invokes the internal default `[[Get]]` operation (and `[[Put]]` for setting values), which not only looks for the property directly on the object, but which will traverse the `[[Prototype]]` chain (see Chapter 5) if not found.

Properties have certain characteristics that can be controlled through property descriptors, such as `writable` and `configurable`. In addition, objects can have their mutability (and that of their properties) controlled to various levels of immutability using `Object.preventExtensions(..)`, `Object.seal(..)`, and `Object.freeze(..)`.

Properties don't have to contain values -- they can be "accessor properties" as well, with getters/setters. They can also be either *enumerable* or not, which controls if they show up in `for..in` loop iterations, for instance.

You can also iterate over **the values** in data structures (arrays, objects, etc) using the ES6 `for..of` syntax, which looks for either a built-in or custom `@@iterator` object consisting of a `next()` method to advance through the data values one at a time.