You Don't Know JS: ES6 & Beyond

Chapter 6: API Additions

From conversions of values to mathematic calculations, ES6 adds many static properties and methods to various built-in natives and objects to help with common tasks. In addition, instances of some of the natives have new capabilities via various new prototype methods.

Note: Most of these features can be faithfully polyfilled. We will not dive into such details here, but check out "ES6 Shim" (https://github.com/paulmillr/es6-shim/) for standards-compliant shims/polyfills.

`Array`

One of the most commonly extended features in JS by various user libraries is the Array type. It should be no surprise that ES6 adds a number of helpers to Array, both static and prototype (instance).

`Array.of(..)` Static Function

There's a well known gotcha with the `Array(..)` constructor, which is that if there's only one argument passed, and that argument is a number, instead of making an array of one element with that number value in it, it constructs an empty array with a `length` property equal to the number. This action produces the unfortunate and quirky "empty slots" behavior that's reviled about JS arrays.

`Array.of(..)` replaces `Array(..)` as the preferred function-form constructor for arrays, because `Array.of(..)` does not have that special single-number-argument case. Consider:

```
var a = Array( 3 );
a.length;						// 3
a[0];							// undefined

var b = Array.of( 3 );
b.length;						// 1
b[0];							// 3

var c = Array.of( 1, 2, 3 );
c.length;						// 3
c;								// [1,2,3]
```

Under what circumstances would you want to use `Array.of(..)` instead of just creating an array with literal syntax, like `c = [1,2,3]`? There's two possible cases.

If you have a callback that's supposed to wrap argument(s) passed to it in an array, `Array.of(..)` fits the bill perfectly. That's probably not terribly common, but it may scratch an itch for you.

The other scenario is if you subclass `Array` (see "Classes" in Chapter 3) and want to be able to create and initialize elements in an instance of your subclass, such as:

```
class MyCoolArray extends Array {
	sum() {
		return this.reduce( function reducer(acc,curr){
			return acc + curr;
		}, 0 );
	}
}

var x = new MyCoolArray( 3 );
```

```
x.length;                                               // 3 -- oops!
x.sum();                                                // 0 -- oops!

var y = [3];                                    // Array, not MyCoolArray
y.length;                                               // 1
y.sum();                                                // `sum` is not a function

var z = MyCoolArray.of( 3 );
z.length;                                               // 1
z.sum();                                                // 3
```

You can't just (easily) create a constructor for `MyCoolArray` that overrides the behavior of the `Array` parent constructor, because that constructor is necessary to actually create a well-behaving array value (initializing the `this`). The "inherited" static `of(..)` method on the `MyCoolArray` subclass provides a nice solution.

`Array.from(..)` Static Function

An "array-like object" in JavaScript is an object that has a `length` property on it, specifically with an integer value of zero or higher.

These values have been notoriously frustrating to work with in JS; it's been quite common to need to transform them into an actual array, so that the various `Array.prototype` methods (`map(..)`, `indexOf(..)` etc.) are available to use with it. That process usually looks like:

```
// array-like object
var arrLike = {
        length: 3,
        0: "foo",
        1: "bar"
};

var arr = Array.prototype.slice.call( arrLike );
```

Another common task where `slice(..)` is often used is in duplicating a real array:

```
var arr2 = arr.slice();
```

In both cases, the new ES6 `Array.from(..)` method can be a more understandable and graceful -- if also less verbose -- approach:

```
var arr = Array.from( arrLike );

var arrCopy = Array.from( arr );
```

`Array.from(..)` looks to see if the first argument is an iterable (see "Iterators" in Chapter 3), and if so, it uses the iterator to produce values to "copy" into the returned array. Because real arrays have an iterator for those values, that iterator is automatically used.

But if you pass an array-like object as the first argument to `Array.from(..)`, it behaves basically the same as `slice()` (no arguments!) or `apply(..)` does, which is that it simply loops over the value, accessing numerically named properties from `0` up to whatever the value of `length` is.

Consider:

```
var arrLike = {
        length: 4,
        2: "foo"
};
```

```
Array.from( arrLike );
// [ undefined, undefined, "foo", undefined ]
```

Because positions `0` , `1` , and `3` didn't exist on `arrLike` , the result was the `undefined` value for each of those slots.

You could produce a similar outcome like this:

```
var emptySlotsArr = [];
emptySlotsArr.length = 4;
emptySlotsArr[2] = "foo";

Array.from( emptySlotsArr );
// [ undefined, undefined, "foo", undefined ]
```

Avoiding Empty Slots

There's a subtle but important difference in the previous snippet between the `emptySlotsArr` and the result of the `Array.from(..)` call. `Array.from(..)` never produces empty slots.

Prior to ES6, if you wanted to produce an array initialized to a certain length with actual `undefined` values in each slot (no empty slots!), you had to do extra work:

```
var a = Array( 4 );                                                  // four empty slots!

var b = Array.apply( null, { length: 4 } );            // four `undefined` values
```

But `Array.from(..)` now makes this easier:

```
var c = Array.from( { length: 4 } );                   // four `undefined` values
```

Warning: Using an empty slot array like `a` in the previous snippets would work with some array functions, but others ignore empty slots (like `map(..)` , etc.). You should never intentionally work with empty slots, as it will almost certainly lead to strange/unpredictable behavior in your programs.

Mapping

The `Array.from(..)` utility has another helpful trick up its sleeve. The second argument, if provided, is a mapping callback (almost the same as the regular `Array#map(..)` expects) which is called to map/transform each value from the source to the returned target. Consider:

```
var arrLike = {
        length: 4,
        2: "foo"
};

Array.from( arrLike, function mapper(val,idx){
        if (typeof val == "string") {
                return val.toUpperCase();
        }
        else {
                return idx;
        }
} );
// [ 0, 1, "FOO", 3 ]
```

Note: As with other array methods that take callbacks, `Array.from(..)` takes an optional third argument that if set will specify the `this` binding for the callback passed as the second argument. Otherwise, `this` will be `undefined` .

See "TypedArrays" in Chapter 5 for an example of using `Array.from(..)` in translating values from an array of 8-bit values to an array of 16-bit values.

Creating Arrays and Subtypes

In the last couple of sections, we've discussed `Array.of(..)` and `Array.from(..)`, both of which create a new array in a similar way to a constructor. But what do they do in subclasses? Do they create instances of the base `Array` or the derived subclass?

```
class MyCoolArray extends Array {
        ..
}

MyCoolArray.from( [1, 2] ) instanceof MyCoolArray;      // true

Array.from(
        MyCoolArray.from( [1, 2] )
) instanceof MyCoolArray;                                                    // false
```

Both `of(..)` and `from(..)` use the constructor that they're accessed from to construct the array. So if you use the base `Array.of(..)` you'll get an `Array` instance, but if you use `MyCoolArray.of(..)`, you'll get a `MyCoolArray` instance.

In "Classes" in Chapter 3, we covered the `@@species` setting which all the built-in classes (like `Array`) have defined, which is used by any prototype methods if they create a new instance. `slice(..)` is a great example:

```
var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;                            // true
```

Generally, that default behavior will probably be desired, but as we discussed in Chapter 3, you *can* override if you want:

```
class MyCoolArray extends Array {
        // force `species` to be parent constructor
        static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;                            // false
x.slice( 1 ) instanceof Array;                                          // true
```

It's important to note that the `@@species` setting is only used for the prototype methods, like `slice(..)`. It's not used by `of(..)` and `from(..)`; they both just use the `this` binding (whatever constructor is used to make the reference). Consider:

```
class MyCoolArray extends Array {
        // force `species` to be parent constructor
        static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

MyCoolArray.from( x ) instanceof MyCoolArray;           // true
MyCoolArray.of( [2, 3] ) instanceof MyCoolArray;        // true
```

`copyWithin(..)` Prototype Method

`Array#copyWithin(..)` is a new mutator method available to all arrays (including Typed Arrays; see Chapter 5). `copyWithin(..)` copies a portion of an array to another location in the same array, overwriting whatever was there before.

The arguments are *target* (the index to copy to), *start* (the inclusive index to start the copying from), and optionally *end* (the exclusive index to stop copying). If any of the arguments are negative, they're taken to be relative from the end of the array.

Consider:

```
[1,2,3,4,5].copyWithin( 3, 0 );                // [1,2,3,1,2]

[1,2,3,4,5].copyWithin( 3, 0, 1 );             // [1,2,3,1,5]

[1,2,3,4,5].copyWithin( 0, -2 );               // [4,5,3,4,5]

[1,2,3,4,5].copyWithin( 0, -2, -1 );    // [4,2,3,4,5]
```

The `copyWithin(..)` method does not extend the array's length, as the first example in the previous snippet shows. Copying simply stops when the end of the array is reached.

Contrary to what you might think, the copying doesn't always go in left-to-right (ascending index) order. It's possible this would result in repeatedly copying an already copied value if the from and target ranges overlap, which is presumably not desired behavior.

So internally, the algorithm avoids this case by copying in reverse order to avoid that gotcha. Consider:

```
[1,2,3,4,5].copyWithin( 2, 1 );         // ???
```

If the algorithm was strictly moving left to right, then the `2` should be copied to overwrite the `3`, then *that* copied `2` should be copied to overwrite `4`, then *that* copied `2` should be copied to overwrite `5`, and you'd end up with `[1,2,2,2,2]`.

Instead, the copying algorithm reverses direction and copies `4` to overwrite `5`, then copies `3` to overwrite `4`, then copies `2` to overwrite `3`, and the final result is `[1,2,2,3,4]`. That's probably more "correct" in terms of expectation, but it can be confusing if you're only thinking about the copying algorithm in a naive left-to-right fashion.

`fill(..)` Prototype Method

Filling an existing array entirely (or partially) with a specified value is natively supported as of ES6 with the `Array#fill(..)` method:

```
var a = Array( 4 ).fill( undefined );
a;
// [undefined,undefined,undefined,undefined]
```

`fill(..)` optionally takes *start* and *end* parameters, which indicate a subset portion of the array to fill, such as:

```
var a = [ null, null, null, null ].fill( 42, 1, 3 );

a;                                                                      // [null,42,42,null]
```

`find(..)` Prototype Method

The most common way to search for a value in an array has generally been the `indexOf(..)` method, which returns the index the value is found at or `-1` if not found:

```
var a = [1,2,3,4,5];

(a.indexOf( 3 ) != -1);                 // true
(a.indexOf( 7 ) != -1);                 // false

(a.indexOf( "2" ) != -1);               // false
```

The `indexOf(..)` comparison requires a strict `===` match, so a search for `"2"` fails to find a value of `2`, and vice versa. There's no way to override the matching algorithm for `indexOf(..)`. It's also unfortunate/ungraceful to have to make the manual comparison to the `-1` value.

Tip: See the *Types & Grammar* title of this series for an interesting (and controversially confusing) technique to work around the `-1` ugliness with the `~` operator.

Since ES5, the most common workaround to have control over the matching logic has been the `some(..)` method. It works by calling a function callback for each element, until one of those calls returns a `true` /truthy value, and then it stops. Because you get to define the callback function, you have full control over how a match is made:

```
var a = [1,2,3,4,5];

a.some( function matcher(v){
        return v == "2";
} );                                                            // true

a.some( function matcher(v){
        return v == 7;
} );                                                            // false
```

But the downside to this approach is that you only get the `true` / `false` indicating if a suitably matched value was found, but not what the actual matched value was.

ES6's `find(..)` addresses this. It works basically the same as `some(..)` , except that once the callback returns a `true` /truthy value, the actual array value is returned:

```
var a = [1,2,3,4,5];

a.find( function matcher(v){
        return v == "2";
} );                                                            // 2

a.find( function matcher(v){
        return v == 7;                                  // undefined
});
```

Using a custom `matcher(..)` function also lets you match against complex values like objects:

```
var points = [
        { x: 10, y: 20 },
        { x: 20, y: 30 },
        { x: 30, y: 40 },
        { x: 40, y: 50 },
        { x: 50, y: 60 }
];

points.find( function matcher(point) {
        return (
                point.x % 3 == 0 &&
                point.y % 4 == 0
        );
} );                                                            // { x: 30, y: 40 }
```

Note: As with other array methods that take callbacks, `find(..)` takes an optional second argument that if set will specify the `this` binding for the callback passed as the first argument. Otherwise, `this` will be `undefined` .

`findIndex(..)` Prototype Method

While the previous section illustrates how `some(..)` yields a boolean result for a search of an array, and `find(..)` yields the matched value itself from the array search, there's also a need for finding the positional index of the matched value.

`indexOf(..)` does that, but there's no control over its matching logic; it always uses `===` strict equality. So ES6's `findIndex(..)` is the answer:

```
var points = [
	{ x: 10, y: 20 },
	{ x: 20, y: 30 },
	{ x: 30, y: 40 },
	{ x: 40, y: 50 },
	{ x: 50, y: 60 }
];

points.findIndex( function matcher(point) {
	return (
		point.x % 3 == 0 &&
		point.y % 4 == 0
	);
} );								// 2

points.findIndex( function matcher(point) {
	return (
		point.x % 6 == 0 &&
		point.y % 7 == 0
	);
} );								// -1
```

Don't use `findIndex(..) != -1` (the way it's always been done with `indexOf(..)`) to get a boolean from the search, because `some(..)` already yields the `true` / `false` you want. And don't do `a[ a.findIndex(..) ]` to get the matched value, because that's what `find(..)` accomplishes. And finally, use `indexOf(..)` if you need the index of a strict match, or `findIndex(..)` if you need the index of a more customized match.

Note: As with other array methods that take callbacks, `findIndex(..)` takes an optional second argument that if set will specify the `this` binding for the callback passed as the first argument. Otherwise, `this` will be `undefined`.

`entries()`, `values()`, `keys()` Prototype Methods

In Chapter 3, we illustrated how data structures can provide a patterned item-by-item enumeration of their values, via an iterator. We then expounded on this approach in Chapter 5, as we explored how the new ES6 collections (Map, Set, etc.) provide several methods for producing different kinds of iterations.

Because it's not new to ES6, `Array` might not be thought of traditionally as a "collection," but it is one in the sense that it provides these same iterator methods: `entries()`, `values()`, and `keys()`. Consider:

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

[...a.values()];					// [1,2,3]
[...a.keys()];						// [0,1,2]
[...a.entries()];					// [ [0,1], [1,2], [2,3] ]

[...a[Symbol.iterator]()];			// [1,2,3]
```

Just like with `Set`, the default `Array` iterator is the same as what `values()` returns.

In "Avoiding Empty Slots" earlier in this chapter, we illustrated how `Array.from(..)` treats empty slots in an array as just being present slots with `undefined` in them. That's actually because under the covers, the array iterators behave that way:

```
var a = [];
a.length = 3;
a[1] = 2;

[...a.values()];		// [undefined,2,undefined]
[...a.keys()];			// [0,1,2]
[...a.entries()];		// [ [0,undefined], [1,2], [2,undefined] ]
```

`Object`

A few additional static helpers have been added to `Object`. Traditionally, functions of this sort have been seen as focused on the behaviors/capabilities of object values.

However, starting with ES6, `Object` static functions will also be for general-purpose global APIs of any sort that don't already belong more naturally in some other location (i.e., `Array.from(..)`).

`Object.is(..)` Static Function

The `Object.is(..)` static function makes value comparisons in an even more strict fashion than the `===` comparison.

`Object.is(..)` invokes the underlying `SameValue` algorithm (ES6 spec, section 7.2.9). The `SameValue` algorithm is basically the same as the `===` Strict Equality Comparison Algorithm (ES6 spec, section 7.2.13), with two important exceptions.

Consider:

```
var x = NaN, y = 0, z = -0;

x === x;                                                        // false
y === z;                                                        // true

Object.is( x, x );                                      // true
Object.is( y, z );                                      // false
```

You should continue to use `===` for strict equality comparisons; `Object.is(..)` shouldn't be thought of as a replacement for the operator. However, in cases where you're trying to strictly identify a `NaN` or `-0` value, `Object.is(..)` is now the preferred option.

Note: ES6 also adds a `Number.isNaN(..)` utility (discussed later in this chapter) which may be a slightly more convenient test; you may prefer `Number.isNaN(x)` over `Object.is(x,NaN)`. You *can* accurately test for `-0` with a clumsy `x == 0 && 1 / x === -Infinity`, but in this case `Object.is(x,-0)` is much better.

`Object.getOwnPropertySymbols(..)` Static Function

The "Symbols" section in Chapter 2 discusses the new Symbol primitive value type in ES6.

Symbols are likely going to be mostly used as special (meta) properties on objects. So the `Object.getOwnPropertySymbols(..)` utility was introduced, which retrieves only the symbol properties directly on an object:

```
var o = {
        foo: 42,
        [ Symbol( "bar" ) ]: "hello world",
        baz: true
};

Object.getOwnPropertySymbols( o );      // [ Symbol(bar) ]
```

`Object.setPrototypeOf(..)` Static Function

Also in Chapter 2, we mentioned the `Object.setPrototypeOf(..)` utility, which (unsurprisingly) sets the `[[Prototype]]` of an object for the purposes of *behavior delegation* (see the *this & Object Prototypes* title of this series). Consider:

```
var o1 = {
        foo() { console.log( "foo" ); }
};
var o2 = {
        // .. o2's definition ..
};

Object.setPrototypeOf( o2, o1 );
```

```
// delegates to `o1.foo()`
o2.foo();                                                    // foo
```

Alternatively:

```
var o1 = {
        foo() { console.log( "foo" ); }
};

var o2 = Object.setPrototypeOf( {
        // .. o2's definition ..
}, o1 );

// delegates to `o1.foo()`
o2.foo();                                                    // foo
```

In both previous snippets, the relationship between `o2` and `o1` appears at the end of the `o2` definition. More commonly, the relationship between an `o2` and `o1` is specified at the top of the `o2` definition, as it is with classes, and also with `__proto__` in object literals (see "Setting `[[Prototype]]` " in Chapter 2).

Warning: Setting a `[[Prototype]]` right after object creation is reasonable, as shown. But changing it much later is generally not a good idea and will usually lead to more confusion than clarity.

`Object.assign(..)` Static Function

Many JavaScript libraries/frameworks provide utilities for copying/mixing one object's properties into another (e.g., jQuery's `extend(..)`). There are various nuanced differences between these different utilities, such as whether a property with value `undefined` is ignored or not.

ES6 adds `Object.assign(..)` , which is a simplified version of these algorithms. The first argument is the *target*, and any other arguments passed are the *sources*, which will be processed in listed order. For each source, its enumerable and own (e.g., not "inherited") keys, including symbols, are copied as if by plain `=` assignment. `Object.assign(..)` returns the target object.

Consider this object setup:

```
var target = {},
        o1 = { a: 1 }, o2 = { b: 2 },
        o3 = { c: 3 }, o4 = { d: 4 };

// setup read-only property
Object.defineProperty( o3, "e", {
        value: 5,
        enumerable: true,
        writable: false,
        configurable: false
} );

// setup non-enumerable property
Object.defineProperty( o3, "f", {
        value: 6,
        enumerable: false
} );

o3[ Symbol( "g" ) ] = 7;

// setup non-enumerable symbol
Object.defineProperty( o3, Symbol( "h" ), {
        value: 8,
        enumerable: false
} );

Object.setPrototypeOf( o3, o4 );
```

Only the properties `a`, `b`, `c`, `e`, and `Symbol("g")` will be copied to `target`:

```
Object.assign( target, o1, o2, o3 );

target.a;                                                       // 1
target.b;                                                       // 2
target.c;                                                       // 3

Object.getOwnPropertyDescriptor( target, "e" );
// { value: 5, writable: true, enumerable: true,
//   configurable: true }

Object.getOwnPropertySymbols( target );
// [Symbol("g")]
```

The `d`, `f`, and `Symbol("h")` properties are omitted from copying; non-enumerable properties and non-owned properties are all excluded from the assignment. Also, `e` is copied as a normal property assignment, not duplicated as a read-only property.

In an earlier section, we showed using `setPrototypeOf(..)` to set up a `[[Prototype]]` relationship between an `o2` and `o1` object. There's another form that leverages `Object.assign(..)`:

```
var o1 = {
        foo() { console.log( "foo" ); }
};

var o2 = Object.assign(
        Object.create( o1 ),
        {
                // .. o2's definition ..
        }
);

// delegates to `o1.foo()`
o2.foo();                                                       // foo
```

Note: `Object.create(..)` is the ES5 standard utility that creates an empty object that is `[[Prototype]]`-linked. See the *this & Object Prototypes* title of this series for more information.

`Math`

ES6 adds several new mathematic utilities that fill in holes or aid with common operations. All of these can be manually calculated, but most of them are now defined natively so that in some cases the JS engine can either more optimally perform the calculations, or perform them with better decimal precision than their manual counterparts.

It's likely that asm.js/transpiled JS code (see the *Async & Performance* title of this series) is the more likely consumer of many of these utilities rather than direct developers.

Trigonometry:

- `cosh(..)` - Hyperbolic cosine
- `acosh(..)` - Hyperbolic arccosine
- `sinh(..)` - Hyperbolic sine
- `asinh(..)` - Hyperbolic arcsine
- `tanh(..)` - Hyperbolic tangent
- `atanh(..)` - Hyperbolic arctangent
- `hypot(..)` - The squareroot of the sum of the squares (i.e., the generalized Pythagorean theorem)

Arithmetic:

- `cbrt(..)` - Cube root

- `clz32(..)` - Count leading zeros in 32-bit binary representation
- `expm1(..)` - The same as `exp(x) - 1`
- `log2(..)` - Binary logarithm (log base 2)
- `log10(..)` - Log base 10
- `log1p(..)` - The same as `log(x + 1)`
- `imul(..)` - 32-bit integer multiplication of two numbers

Meta:

- `sign(..)` - Returns the sign of the number
- `trunc(..)` - Returns only the integer part of a number
- `fround(..)` - Rounds to nearest 32-bit (single precision) floating-point value

`Number`

Importantly, for your program to properly work, it must accurately handle numbers. ES6 adds some additional properties and functions to assist with common numeric operations.

Two additions to `Number` are just references to the preexisting globals: `Number.parseInt(..)` and `Number.parseFloat(..)`.

Static Properties

ES6 adds some helpful numeric constants as static properties:

- `Number.EPSILON` - The minimum value between any two numbers: `2^-52` (see Chapter 2 of the *Types & Grammar* title of this series regarding using this value as a tolerance for imprecision in floating-point arithmetic)
- `Number.MAX_SAFE_INTEGER` - The highest integer that can "safely" be represented unambiguously in a JS number value: `2^53 - 1`
- `Number.MIN_SAFE_INTEGER` - The lowest integer that can "safely" be represented unambiguously in a JS number value: `-(2^53 - 1)` or `(-2)^53 + 1`.

Note: See Chapter 2 of the *Types & Grammar* title of this series for more information about "safe" integers.

`Number.isNaN(..)` Static Function

The standard global `isNaN(..)` utility has been broken since its inception, in that it returns `true` for things that are not numbers, not just for the actual `NaN` value, because it coerces the argument to a number type (which can falsely result in a NaN). ES6 adds a fixed utility `Number.isNaN(..)` that works as it should:

```
var a = NaN, b = "NaN", c = 42;

isNaN( a );                                             // true
isNaN( b );                                             // true -- oops!
isNaN( c );                                             // false

Number.isNaN( a );                                 // true
Number.isNaN( b );                                 // false -- fixed!
Number.isNaN( c );                                 // false
```

`Number.isFinite(..)` Static Function

There's a temptation to look at a function name like `isFinite(..)` and assume it's simply "not infinite". That's not quite correct, though. There's more nuance to this new ES6 utility. Consider:

```
var a = NaN, b = Infinity, c = 42;

Number.isFinite( a );                   // false
Number.isFinite( b );                   // false
Number.isFinite( c );                           // true
```

The standard global `isFinite(..)` coerces its argument, but `Number.isFinite(..)` omits the coercive behavior:

```
var a = "42";

isFinite( a );                                          // true
Number.isFinite( a );                           // false
```

You may still prefer the coercion, in which case using the global `isFinite(..)` is a valid choice. Alternatively, and perhaps more sensibly, you can use `Number.isFinite(+x)`, which explicitly coerces `x` to a number before passing it in (see Chapter 4 of the *Types & Grammar* title of this series).

Integer-Related Static Functions

JavaScript number values are always floating point (IEEE-754). So the notion of determining if a number is an "integer" is not about checking its type, because JS makes no such distinction.

Instead, you need to check if there's any non-zero decimal portion of the value. The easiest way to do that has commonly been:

```
x === Math.floor( x );
```

ES6 adds a `Number.isInteger(..)` helper utility that potentially can determine this quality slightly more efficiently:

```
Number.isInteger( 4 );                          // true
Number.isInteger( 4.2 );                        // false
```

Note: In JavaScript, there's no difference between `4`, `4.`, `4.0`, or `4.0000`. All of these would be considered an "integer", and would thus yield `true` from `Number.isInteger(..)`.

In addition, `Number.isInteger(..)` filters out some clearly not-integer values that `x === Math.floor(x)` could potentially mix up:

```
Number.isInteger( NaN );                        // false
Number.isInteger( Infinity );           // false
```

Working with "integers" is sometimes an important bit of information, as it can simplify certain kinds of algorithms. JS code by itself will not run faster just from filtering for only integers, but there are optimization techniques the engine can take (e.g., asm.js) when only integers are being used.

Because of `Number.isInteger(..)`'s handling of `NaN` and `Infinity` values, defining a `isFloat(..)` utility would not be just as simple as `!Number.isInteger(..)`. You'd need to do something like:

```
function isFloat(x) {
        return Number.isFinite( x ) && !Number.isInteger( x );
}

isFloat( 4.2 );                                         // true
isFloat( 4 );                                           // false

isFloat( NaN );                                         // false
isFloat( Infinity );                            // false
```

Note: It may seem strange, but Infinity should neither be considered an integer nor a float.

ES6 also defines a `Number.isSafeInteger(..)` utility, which checks to make sure the value is both an integer and within the range of `Number.MIN_SAFE_INTEGER` - `Number.MAX_SAFE_INTEGER` (inclusive).

```
var x = Math.pow( 2, 53 ),
	y = Math.pow( -2, 53 );

Number.isSafeInteger( x - 1 );		// true
Number.isSafeInteger( y + 1 );		// true

Number.isSafeInteger( x );			// false
Number.isSafeInteger( y );			// false
```

String

Strings already have quite a few helpers prior to ES6, but even more have been added to the mix.

Unicode Functions

"Unicode-Aware String Operations" in Chapter 2 discusses `String.fromCodePoint(..)`, `String#codePointAt(..)`, and `String#normalize(..)` in detail. They have been added to improve Unicode support in JS string values.

```
String.fromCodePoint( 0x1d49e );			// "𝒞"

"ab𝒞d".codePointAt( 2 ).toString( 16 );		// "1d49e"
```

The `normalize(..)` string prototype method is used to perform Unicode normalizations that either combine characters with adjacent "combining marks" or decompose combined characters.

Generally, the normalization won't create a visible effect on the contents of the string, but will change the contents of the string, which can affect how things like the `length` property are reported, as well as how character access by position behave:

```
var s1 = "e\u0301";
s1.length;							// 2

var s2 = s1.normalize();
s2.length;							// 1
s2 === "\xE9";						// true
```

`normalize(..)` takes an optional argument that specifies the normalization form to use. This argument must be one of the following four values: `"NFC"` (default), `"NFD"`, `"NFKC"`, or `"NFKD"`.

Note: Normalization forms and their effects on strings is well beyond the scope of what we'll discuss here. See "Unicode Normalization Forms" (http://www.unicode.org/reports/tr15/) for more information.

`String.raw(..)` Static Function

The `String.raw(..)` utility is provided as a built-in tag function to use with template string literals (see Chapter 2) for obtaining the raw string value without any processing of escape sequences.

This function will almost never be called manually, but will be used with tagged template literals:

```
var str = "bc";

String.raw`\ta${str}d\xE9`;
// "\tabcd\xE9", not "	abcdé"
```

In the resultant string, `\` and `t` are separate raw characters, not the one escape sequence character `\t`. The same is true with the Unicode escape sequence.

`repeat(..)` Prototype Function

In languages like Python and Ruby, you can repeat a string as:

```
"foo" * 3;                                                  // "foofoofoo"
```

That doesn't work in JS, because `*` multiplication is only defined for numbers, and thus `"foo"` coerces to the `NaN` number.

However, ES6 defines a string prototype method `repeat(..)` to accomplish the task:

```
"foo".repeat( 3 );                                  // "foofoofoo"
```

String Inspection Functions

In addition to `String#indexOf(..)` and `String#lastIndexOf(..)` from prior to ES6, three new methods for searching/inspection have been added: `startsWith(..)`, `endsWith(..)`, and `includes(..)`.

```
var palindrome = "step on no pets";

palindrome.startsWith( "step on" );     // true
palindrome.startsWith( "on", 5 );       // true

palindrome.endsWith( "no pets" );       // true
palindrome.endsWith( "no", 10 );        // true

palindrome.includes( "on" );            // true
palindrome.includes( "on", 6 );         // false
```

For all the string search/inspection methods, if you look for an empty string `""`, it will either be found at the beginning or the end of the string.

Warning: These methods will not by default accept a regular expression for the search string. See "Regular Expression Symbols" in Chapter 7 for information about disabling the `isRegExp` check that is performed on this first argument.

Review

ES6 adds many extra API helpers on the various built-in native objects:

- `Array` adds `of(..)` and `from(..)` static functions, as well as prototype functions like `copyWithin(..)` and `fill(..)`.
- `Object` adds static functions like `is(..)` and `assign(..)`.
- `Math` adds static functions like `acosh(..)` and `clz32(..)`.
- `Number` adds static properties like `Number.EPSILON`, as well as static functions like `Number.isFinite(..)`.
- `String` adds static functions like `String.fromCodePoint(..)` and `String.raw(..)`, as well as prototype functions like `repeat(..)` and `includes(..)`.

Most of these additions can be polyfilled (see ES6 Shim), and were inspired by utilities in common JS libraries/frameworks.