Symbols and Symbol Properties

Symbols are a primitive type introduced in ECMAScript 6, joining the existing primitive types: strings, numbers, booleans, `null`, and `undefined`. Symbols began as a way to create private object members, a feature JavaScript developers wanted for a long time. Before symbols, any property with a string name was easy to access regardless of the obscurity of the name, and the "private names" feature was meant to let developers create non-string property names. That way, normal techniques for detecting these private names wouldn't work.

The private names proposal eventually evolved into ECMAScript 6 symbols, and this chapter will teach you how to use symbols effectively. While the implementation details remained the same (that is, they added non-string values for property names), the goal of privacy was dropped. Instead, symbol properties are categorized separately from other object properties.

Creating Symbols

Symbols are unique among JavaScript primitives in that they don't have a literal form, like `true` for booleans or `42` for numbers. You can create a symbol by using the global `Symbol` function, as in this example:

```
let firstName = Symbol();
let person = {};

person[firstName] = "Nicholas";
console.log(person[firstName]);     // "Nicholas"
```

Here, the symbol `firstName` is created and used to assign a new property on the `person` object. That symbol must be used each time you want to access that same property. Naming the symbol variable appropriately is a good idea, so you can easily tell what the symbol represents.

W> Because symbols are primitive values, calling `new Symbol()` throws an error when called. You can create an instance of `Symbol` via `new Object(yourSymbol)` as well, but it's unclear when this capability would be useful.

The `Symbol` function also accepts an optional argument that is the description of the symbol. The description itself cannot be used to access the property, but is used for debugging purposes. For example:

```
let firstName = Symbol("first name");
let person = {};

person[firstName] = "Nicholas";

console.log("first name" in person);         // false
console.log(person[firstName]);              // "Nicholas"
console.log(firstName);                      // "Symbol(first name)"
```

A symbol's description is stored internally in the `[[Description]]` property. This property is read whenever the symbol's `toString()` method is called either explicitly or implicitly. The `firstName` symbol's `toString()` method is called implictly by `console.log()` in this example, so the description gets printed to the log. It is not otherwise possible to access `[[Description]]` directly from code. I recommended always providing a description to make both reading and debugging symbols easier.

A> ### Identifying Symbols A> A>Since symbols are primitive values, you can use the `typeof` operator to determine if a variable contains a symbol. ECMAScript 6 extends `typeof` to return `"symbol"` when used on a symbol. For example: A> A>`js A>let symbol = Symbol("test symbol"); A>console.log(typeof symbol); // "symbol" A>` A> A>While there are other indirect ways of determining whether a variable is a symbol, the `typeof` operator is the most accurate and preferred technique.

Using Symbols

You can use symbols anywhere you'd use a computed property name. You've already seen bracket notation used with symbols in this chapter, but you can use symbols in computed object literal property names as well as with `Object.defineProperty()` and `Object.defineProperties()` calls, such as:

```
let firstName = Symbol("first name");

// use a computed object literal property
let person = {
    [firstName]: "Nicholas"
};

// make the property read only
Object.defineProperty(person, firstName, { writable: false });

let lastName = Symbol("last name");

Object.defineProperties(person, {
    [lastName]: {
        value: "Zakas",
        writable: false
    }
});

console.log(person[firstName]);     // "Nicholas"
console.log(person[lastName]);      // "Zakas"
```

This example first uses a computed object literal property to create the `firstName` symbol property. The following line then sets the property to be read-only. Later, a read-only `lastName` symbol property is created using the `Object.defineProperties()` method. A computed object literal property is used once again, but this time, it's part of the second argument to the `Object.defineProperties()` call.

While symbols can be used in any place that computed property names are allowed, you'll need to have a system for sharing these symbols between different pieces of code in order to use them effectively.

Sharing Symbols

You may find that you want different parts of your code to use the same symbols. For example, suppose you have two different object types in your application that should use the same symbol property to represent a unique identifier. Keeping track of symbols across files or large codebases can be difficult and error-prone. That's why ECMAScript 6 provides a global symbol registry that you can access at any point in time.

When you want to create a symbol to be shared, use the `Symbol.for()` method instead of calling the `Symbol()` method. The `Symbol.for()` method accepts a single parameter, which is a string identifier for the symbol you want to create. That parameter is also used as the symbol's description. For example:

```
let uid = Symbol.for("uid");
let object = {};

object[uid] = "12345";

console.log(object[uid]);       // "12345"
console.log(uid);               // "Symbol(uid)"
```

The `Symbol.for()` method first searches the global symbol registry to see if a symbol with the key `"uid"` exists. If so, the method returns the existing symbol. If no such symbol exists, then a new symbol is created and registered to the global symbol registry using the specified key. The new symbol is then returned. That means subsequent calls to `Symbol.for()` using the same key will return the same symbol, as follows:

```
let uid = Symbol.for("uid");
let object = {
    [uid]: "12345"
};

console.log(object[uid]);       // "12345"
console.log(uid);               // "Symbol(uid)"

let uid2 = Symbol.for("uid");

console.log(uid === uid2);      // true
console.log(object[uid2]);      // "12345"
console.log(uid2);              // "Symbol(uid)"
```

In this example, `uid` and `uid2` contain the same symbol and so they can be used interchangeably. The first call to `Symbol.for()` creates the symbol, and the second call retrieves the symbol from the global symbol registry.

Another unique aspect of shared symbols is that you can retrieve the key associated with a symbol in the global symbol registry by calling the `Symbol.keyFor()` method. For example:

```
let uid = Symbol.for("uid");
console.log(Symbol.keyFor(uid));    // "uid"

let uid2 = Symbol.for("uid");
```

```
console.log(Symbol.keyFor(uid2));   // "uid"

let uid3 = Symbol("uid");
console.log(Symbol.keyFor(uid3));   // undefined
```

Notice that both `uid` and `uid2` return the `"uid"` key. The symbol `uid3` doesn't exist in the global symbol registry, so it has no key associated with it and `Symbol.keyFor()` returns `undefined`.

W> The global symbol registry is a shared environment, just like the global scope. That means you can't make assumptions about what is or is not already present in that environment. Use namespacing of symbol keys to reduce the likelihood of naming collisions when using third-party components. For example, jQuery code might use `"jquery."` to prefix all keys, for keys like `"jquery.element"` or similar.

Symbol Coercion

Type coercion is a significant part of JavaScript, and there's a lot of flexibility in the language's ability to coerce one data type into another. Symbols, however, are quite inflexible when it comes to coercion because other types lack a logical equivalent to a symbol. Specifically, symbols cannot be coerced into strings or numbers so that they cannot accidentally be used as properties that would otherwise be expected to behave as symbols.

The examples in this chapter have used `console.log()` to indicate the output for symbols, and that works because `console.log()` calls `String()` on symbols to create useful output. You can use `String()` directly to get the same result. For instance:

```
let uid = Symbol.for("uid"),
    desc = String(uid);

console.log(desc);              // "Symbol(uid)"
```

The `String()` function calls `uid.toString()` and the symbol's string description is returned. If you try to concatenate the symbol directly with a string, however, an error will be thrown:

```
let uid = Symbol.for("uid"),
    desc = uid + "";            // error!
```

Concatenating `uid` with an empty string requires that `uid` first be coerced into a string. An error is thrown when the coercion is detected, preventing its use in this manner.

Similarly, you cannot coerce a symbol to a number. All mathematical operators cause an error when applied to a symbol. For example:

```
let uid = Symbol.for("uid"),
    sum = uid / 1;            // error!
```

This example attempts to divide the symbol by 1, which causes an error. Errors are thrown regardless of the mathematical operator used (logical operators do not throw an error because all symbols are considered equivalent to `true`, just like any other non-empty value in JavaScript).

Retrieving Symbol Properties

The `Object.keys()` and `Object.getOwnPropertyNames()` methods can retrieve all property names in an object. The former method returns all enumerable property names, and the latter returns all properties regardless of enumerability. Neither method returns symbol properties, however, to preserve their ECMAScript 5 functionality. Instead, the `Object.getOwnPropertySymbols()` method was added in ECMAScript 6 to allow you to retrieve property symbols from an object.

The return value of `Object.getOwnPropertySymbols()` is an array of own property symbols. For example:

```
let uid = Symbol.for("uid");
let object = {
    [uid]: "12345"
};

let symbols = Object.getOwnPropertySymbols(object);

console.log(symbols.length);        // 1
console.log(symbols[0]);            // "Symbol(uid)"
console.log(object[symbols[0]]);    // "12345"
```

In this code, `object` has a single symbol property called `uid`. The array returned from `Object.getOwnPropertySymbols()` is an array containing just that symbol.

All objects start with zero own symbol properties, but objects can inherit symbol properties from their prototypes. ECMAScript 6 predefines several such properties, implemented using what are called well-known symbols.

Exposing Internal Operations with Well-Known Symbols

A central theme for ECMAScript 5 was exposing and defining some of the "magic" parts of JavaScript, the parts that developers couldn't emulate at the time. ECMAScript 6 carries on that tradition by exposing even more of the previously internal logic of the language, primarily by using symbol prototype properties to define the basic behavior of certain objects.

ECMAScript 6 has predefined symbols called *well-known symbols* that represent common behaviors in JavaScript that were previously considered internal-only operations. Each well-known symbol is represented by a property on the `Symbol` object, such as `Symbol.create`.

The well-known symbols are:

- `Symbol.hasInstance` - A method used by `instanceof` to determine an object's inheritance.
- `Symbol.isConcatSpreadable` - A Boolean value indicating that `Array.prototype.concat()` should flatten the collection's elements if the collection is passed as a parameter to `Array.prototype.concat()`.

- `Symbol.iterator` - A method that returns an iterator. (Iterators are covered in Chapter 8.)
- `Symbol.match` - A method used by `String.prototype.match()` to compare strings.
- `Symbol.replace` - A method used by `String.prototype.replace()` to replace substrings.
- `Symbol.search` - A method used by `String.prototype.search()` to locate substrings.
- `Symbol.species` - The constructor for making derived objects. (Derived objects are covered in Chapter 9.)
- `Symbol.split` - A method used by `String.prototype.split()` to split up strings.
- `Symbol.toPrimitive` - A method that returns a primitive value representation of an object.
- `Symbol.toStringTag` - A string used by `Object.prototype.toString()` to create an object description.
- `Symbol.unscopables` - An object whose properties are the names of object properties that should not be included in a `with` statement.

Some commonly used well-known symbols are discussed in the following sections, while others are discussed throughout the rest of the book to keep them in the correct context.

I> Overwriting a method defined with a well-known symbol changes an ordinary object to an exotic object because this changes some internal default behavior. There is no practical impact to your code as a result, it just changes the way the specification describes the object.

The Symbol.hasInstance Property

Every function has a `Symbol.hasInstance` method that determines whether or not a given object is an instance of that function. The method is defined on `Function.prototype` so that all functions inherit the default behavior for the `instanceof` property and the method is nonwritable and nonconfigurable as well as nonenumerable, to ensure it doesn't get overwritten by mistake.

The `Symbol.hasInstance` method accepts a single argument: the value to check. It returns true if the value passed is an instance of the function. To understand how `Symbol.hasInstance` works, consider the following code:

```
obj instanceof Array;
```

This code is equivalent to:

```
Array[Symbol.hasInstance](obj);
```

ECMAScript 6 essentially redefined the `instanceof` operator as shorthand syntax for this method call. And now that there's a method call involved, you can actually change how `instanceof` works.

For instance, suppose you want to define a function that claims no object as an instance. You can do so by hardcoding the return value of `Symbol.hasInstance` to `false`, such as:

```
function MyObject() {
    // ...
```

```
}

Object.defineProperty(MyObject, Symbol.hasInstance, {
    value: function(v) {
        return false;
    }
});

let obj = new MyObject();

console.log(obj instanceof MyObject);       // false
```

You must use `Object.defineProperty()` to overwrite a nonwritable property, so this example uses that method to overwrite the `Symbol.hasInstance` method with a new function. The new function always returns `false`, so even though `obj` is actually an instance of the `MyObject` class, the `instanceof` operator returns `false` after the `Object.defineProperty()` call.

Of course, you can also inspect the value and decide whether or not a value should be considered an instance based on any arbitrary condition. For instance, maybe numbers with values between 1 and 100 are to be considered instances of a special number type. To achieve that behavior, you might write code like this:

```
function SpecialNumber() {
    // empty
}

Object.defineProperty(SpecialNumber, Symbol.hasInstance, {
    value: function(v) {
        return (v instanceof Number) && (v >=1 && v <= 100);
    }
});

let two = new Number(2),
    zero = new Number(0);

console.log(two instanceof SpecialNumber);    // true
console.log(zero instanceof SpecialNumber);   // false
```

This code defines a `Symbol.hasInstance` method that returns `true` if the value is an instance of `Number` and also has a value between 1 and 100. Thus, `SpecialNumber` will claim `two` as an instance even though there is no directly defined relationship between the `SpecialNumber` function and the `two` variable. Note that the left operand to `instanceof` must be an object to trigger the `Symbol.hasInstance` call, as nonobjects cause `instanceof` to simply return `false` all the time.

W> You can also overwrite the default `Symbol.hasInstance` property for all builtin functions such as the `Date` and `Error` functions. This isn't recommended, however, as the effects on your code can be unexpected and confusing. It's a good idea to only overwrite `Symbol.hasInstance` on your own functions and only when necessary.

The Symbol.isConcatSpreadable Symbol

JavaScript arrays have a `concat()` method designed to concatenate two arrays together. Here's how that method is used:

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ]);

console.log(colors2.length);    // 4
console.log(colors2);           // ["red","green","blue","black"]
```

This code concatenates a new array to the end of `colors1` and creates `colors2`, a single array with all items from both arrays. However, the `concat()` method can also accept nonarray arguments and, in that case, those arguments are simply added to the end of the array. For example:

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ], "brown");

console.log(colors2.length);    // 5
console.log(colors2);           // ["red","green","blue","black","brown"]
```

Here, the extra argument `"brown"` is passed to `concat()` and it becomes the fifth item in the `colors2` array. Why is an array argument treated differently than a string argument? The JavaScript specification says that arrays are automatically split into their individual items and all other types are not. Prior to ECMAScript 6, there was no way to adjust this behavior.

The `Symbol.isConcatSpreadable` property is a boolean value indicating that an object has a `length` property and numeric keys, and that its numeric property values should be added individually to the result of a `concat()` call. Unlike other well-known symbols, this symbol property doesn't appear on any standard objects by default. Instead, the symbol is available as a way to augment how `concat()` works on certain types of objects, effectively short-circuiting the default behavior. You can define any type to behave like arrays do in a `concat()` call, like this:

```
let collection = {
    0: "Hello",
    1: "world",
    length: 2,
    [Symbol.isConcatSpreadable]: true
};

let messages = [ "Hi" ].concat(collection);

console.log(messages.length);    // 3
console.log(messages);           // ["Hi","Hello","world"]
```

The `collection` object in this example is set up to look like an array: it has a `length` property and two numeric keys. The `Symbol.isConcatSpreadable` property is set to `true` to indicate that the property

values should be added as individual items to an array. When `collection` is passed to the `concat()` method, the resulting array has `"Hello"` and `"world"` as separate items after the `"Hi"` element.

I> You can also set `Symbol.isConcatSpreadable` to `false` on array subclasses to prevent items from being separated by `concat()` calls. Subclassing is discussed in Chapter 8.

The Symbol.match, Symbol.replace, Symbol.search, and Symbol.split Symbols

Strings and regular expressions have always had a close relationship in JavaScript. The string type, in particular, has several methods that accept regular expressions as arguments:

- `match(regex)` - Determines whether the given string matches a regular expression
- `replace(regex, replacement)` - Replaces regular expression matches with a `replacement`
- `search(regex)` - Locates a regular expression match inside the string
- `split(regex)` - Splits a string into an array on a regular expression match

Prior to ECMAScript 6, the way these methods interacted with regular expressions was hidden from developers, leaving no way to mimic regular expressions using developer-defined objects. ECMAScript 6 defines four symbols that correspond to these four methods, effectively outsourcing the native behavior to the `RegExp` builtin object.

The `Symbol.match`, `Symbol.replace`, `Symbol.search`, and `Symbol.split` symbols represent methods on the regular expression argument that should be called on the first argument to the `match()` method, the `replace()` method, the `search()` method, and the `split()` method, respectively. The four symbol properties are defined on `RegExp.prototype` as the default implementation that the string methods should use.

Knowing this, you can create an object to use with the string methods in a way that is similar to regular expressions. To do, you can use the following symbol functions in code:

- `Symbol.match` - A function that accepts a string argument and returns an array of matches, or `null` if no match is found.
- `Symbol.replace` - A function that accepts a string argument and a replacement string, and returns a string.
- `Symbol.search` - A function that accepts a string argument and returns the numeric index of the match, or -1 if no match is found.
- `Symbol.split` - A function that accepts a string argument and returns an array containing pieces of the string split on the match.

The ability to define these properties on an object allows you to create objects that implement pattern matching without regular expressions and use them in methods that expect regular expressions. Here's an example that shows these symbols in action:

```
// effectively equivalent to /^.{10}$/
let hasLengthOf10 = {
    [Symbol.match]: function(value) {
        return value.length === 10 ? [value] : null;
    },
    [Symbol.replace]: function(value, replacement) {
        return value.length === 10 ? replacement : value;
```

```
    },
    [Symbol.search]: function(value) {
        return value.length === 10 ? 0 : -1;
    },
    [Symbol.split]: function(value) {
        return value.length === 10 ? ["", ""] : [value];
    }
};

let message1 = "Hello world",   // 11 characters
    message2 = "Hello John";    // 10 characters


let match1 = message1.match(hasLengthOf10),
    match2 = message2.match(hasLengthOf10);

console.log(match1);            // null
console.log(match2);            // ["Hello John"]

let replace1 = message1.replace(hasLengthOf10, "Howdy!"),
    replace2 = message2.replace(hasLengthOf10, "Howdy!");

console.log(replace1);          // "Hello world"
console.log(replace2);          // "Howdy!"

let search1 = message1.search(hasLengthOf10),
    search2 = message2.search(hasLengthOf10);

console.log(search1);           // -1
console.log(search2);           // 0

let split1 = message1.split(hasLengthOf10),
    split2 = message2.split(hasLengthOf10);

console.log(split1);            // ["Hello world"]
console.log(split2);            // ["", ""]
```

The `hasLengthOf10` object is intended to work like a regular expression that matches whenever the string length is exactly 10. Each of the four methods on `hasLengthOf10` is implemented using the appropriate symbol, and then the corresponding methods on two strings are called. The first string, `message1`, has 11 characters and so it will not match; the second string, `message2`, has 10 characters and so it will match. Despite not being a regular expression, `hasLengthOf10` is passed to each string method and used correctly due to the additional methods.

While this is a simple example, the ability to perform more complex matches than are currently possible with regular expressions opens up a lot of possibilities for custom pattern matchers.

The Symbol.toPrimitive Method

JavaScript frequently attempts to convert objects into primitive values implicitly when certain operations are applied. For instance, when you compare a string to an object using the double equals (`==`) operator, the object

is converted into a primitive value before comparing. Exactly what primitive value should be used was previously an internal operation, but ECMAScript 6 exposes that value (making it changeable) through the `Symbol.toPrimitive` method.

The `Symbol.toPrimitive` method is defined on the prototype of each standard type and prescribes what should happen when the object is converted into a primitive. When a primitive conversion is needed, `Symbol.toPrimitive` is called with a single argument, referred to as `hint` in the specification. The `hint` argument is one of three string values. If `hint` is `"number"` then `Symbol.toPrimitive` should return a number. If `hint` is `"string"` then a string should be returned, and if it's `"default"` then the operation has no preference as to the type.

For most standard objects, number mode has the following behaviors, in order by priority:

1. Call the `valueOf()` method, and if the result is a primitive value, return it.
2. Otherwise, call the `toString()` method, and if the result is a primitive value, return it.
3. Otherwise, throw an error.

Similarly, for most standard objects, the behaviors of string mode have the following priority:

1. Call the `toString()` method, and if the result is a primitive value, return it.
2. Otherwise, call the `valueOf()` method, and if the result is a primitive value, return it.
3. Otherwise, throw an error.

In many cases, standard objects treat default mode as equivalent to number mode (except for `Date`, which treats default mode as equivalent to string mode). By defining an `Symbol.toPrimitive` method, you can override these default coercion behaviors.

I> Default mode is only used for the `==` operator, the `+` operator, and when passing a single argument to the `Date` constructor. Most operations use string or number mode.

To override the default conversion behaviors, use `Symbol.toPrimitive` and assign a function as its value. For example:

```
function Temperature(degrees) {
    this.degrees = degrees;
}

Temperature.prototype[Symbol.toPrimitive] = function(hint) {

    switch (hint) {
        case "string":
            return this.degrees + "\u00b0"; // degrees symbol

        case "number":
            return this.degrees;

        case "default":
            return this.degrees + " degrees";
    }
};
```

```
let freezing = new Temperature(32);

console.log(freezing + "!");            // "32 degrees!"
console.log(freezing / 2);              // 16
console.log(String(freezing));          // "32°"
```

This script defines a `Temperature` constructor and overrides the default `Symbol.toPrimitive` method on the prototype. A different value is returned depending on whether the `hint` argument indicates string, number, or default mode (the `hint` argument is filled in by the JavaScript engine). In string mode, the `Symbol.toPrimitive` method returns the temperature with the Unicode degrees symbol. In number mode, it returns just the numeric value, and in default mode, it appends the word "degrees" after the number.

Each of the log statements triggers a different `hint` argument value. The `+` operator triggers default mode by setting `hint` to `"default"`, the `/` operator triggers number mode by setting `hint` to `"number"`, and the `String()` function triggers string mode by setting `hint` to `"string"`. Returning different values for all three modes is possible, it's much more common to set the default mode to be the same as string or number mode.

The Symbol.toStringTag Symbol

One of the most interesting problems in JavaScript has been the availability of multiple global execution environments. This occurs in web browsers when a page includes an iframe, as the page and the iframe each have their own execution environments. In most cases, this isn't a problem, as data can be passed back and forth between the environments with little cause for concern. The problem arises when trying to identify what type of object you're dealing with after the object has been passed between different environments.

The canonical example of this issue is passing an array from an iframe into the containing page or vice-versa. In ECMAScript 6 terminology, the iframe and the containing page each represent a different *realm* which is an execution environment for JavaScript. Each realm has its own global scope with its own copy of global objects. In whichever realm the array is created, it is definitely an array. When it's passed to a different realm, however, an `instanceof Array` call returns `false` because the array was created with a constructor from a different realm and `Array` represents the constructor in the current realm.

A Workaround for the Identification Problem

Faced with this problem, developers soon found a good way to identify arrays. They discovered that by calling the standard `toString()` method on the object, a predictable string was always returned. Thus, many JavaScript libraries began including a function like this:

```
function isArray(value) {
    return Object.prototype.toString.call(value) === "[object Array]";
}

console.log(isArray([]));   // true
```

This may look a bit roundabout, but it worked quite well for identifying arrays in all browsers. The `toString()` method on arrays isn't useful for identifying an object because it returns a string representation of the items the object contains. But the `toString()` method on `Object.prototype` had a quirk: it included internally-

defined name called `[[Class]]` in the returned result. Developers could use this method on an object to retrieve what the JavaScript environment thought the object's data type was.

Developers quickly realized that since there was no way to change this behavior, it was possible to use the same approach to distinguish between native objects and those created by developers. The most important case of this was the ECMAScript 5 `JSON` object.

Prior to ECMAScript 5, many developers used Douglas Crockford's *json2.js*, which creates a global `JSON` object. As browsers started to implement the `JSON` global object, figuring out whether the global `JSON` was provided by the JavaScript environment itself or through some other library became necessary. Using the same technique I showed with the `isArray()` function, many developers created functions like this:

```
function supportsNativeJSON() {
    return typeof JSON !== "undefined" &&
        Object.prototype.toString.call(JSON) === "[object JSON]";
}
```

The same characteristic of `Object.prototype` that allowed developers to identify arrays across iframe boundaries also provided a way to tell if `JSON` was the native `JSON` object or not. A non-native `JSON` object would return `[object Object]` while the native version returned `[object JSON]` instead. This approach became the de facto standard for identifying native objects.

The ECMAScript 6 Answer

ECMAScript 6 redefines this behavior through the `Symbol.toStringTag` symbol. This symbol represents a property on each object that defines what value should be produced when `Object.prototype.toString.call()` is called on it. For an array, the value that function returns is explained by storing `"Array"` in the `Symbol.toStringTag` property.

Likewise, you can define the `Symbol.toStringTag` value for your own objects:

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

Person.prototype[Symbol.toStringTag] = "Person";

let me = new Person("Nicholas");

console.log(me.toString());                         // "[object Person]"
console.log(Object.prototype.toString.call(me));    // "[object Person]"
```

In this example, a `Symbol.toStringTag` property is defined on `Person.prototype` to provide the default behavior for creating a string representation. Since `Person.prototype` inherits the `Object.prototype.toString()` method, the value returned from `Symbol.toStringTag` is also used when calling the `me.toString()` method. However, you can still define your own `toString()` method that

provides a different behavior without affecting the use of the `Object.prototype.toString.call()` method. Here's how that might look:

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

Person.prototype[Symbol.toStringTag] = "Person";

Person.prototype.toString = function() {
    return this.name;
};

let me = new Person("Nicholas");

console.log(me.toString());                         // "Nicholas"
console.log(Object.prototype.toString.call(me));    // "[object Person]"
```

This code defines `Person.prototype.toString()` to return the value of the `name` property. Since `Person` instances no longer inherit the `Object.prototype.toString()` method, calling `me.toString()` exhibits a different behavior.

I> All objects inherit `Symbol.toStringTag` from `Object.prototype` unless otherwise specified. The string `"Object"` is the default property value.

There is no restriction on which values can be used for `Symbol.toStringTag` on developer-defined objects. For example, nothing prevents you from using `"Array"` as the value of the `Symbol.toStringTag` property, such as:

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

Person.prototype[Symbol.toStringTag] = "Array";

Person.prototype.toString = function() {
    return this.name;
};

let me = new Person("Nicholas");

console.log(me.toString());                         // "Nicholas"
console.log(Object.prototype.toString.call(me));    // "[object Array]"
```

The result of calling `Object.prototype.toString()` is `"[object Array]"` in this code, which is the same result you'd get from an actual array. This highlights the fact that `Object.prototype.toString()` is no longer a completely reliable way of identifying an object's type.

Changing the string tag for native objects is also possible. Just assign to `Symbol.toStringTag` on the object's prototype, like this:

```
Array.prototype[Symbol.toStringTag] = "Magic";

let values = [];

console.log(Object.prototype.toString.call(values));    // "[object
Magic]"
```

`Symbol.toStringTag` is overwritten for arrays in this example, meaning the call to `Object.prototype.toString()` results in `"[object Magic]"` instead of `"[object Array]"`. While I recommended not changing built-in objects in this way, there's nothing in the language that forbids doing so.

The Symbol.unscopables Symbol

The `with` statement is one of the most controversial parts of JavaScript. Originally designed to avoid repetitive typing, the `with` statement later became roundly criticized for making code harder to understand and for negative performance implications as well as being error-prone.

As a result, the `with` statement is not allowed in strict mode; that restriction also affects classes and modules, which are strict mode by default and have no opt-out.

While future code will undoubtedly not use the `with` statement, ECMAScript 6 still supports `with` in nonstrict mode for backwards compatibility and, as such, had to find ways to allow code that does use `with` to continue to work properly.

To understand the complexity of this task, consider the following code:

```
let values = [1, 2, 3],
    colors = ["red", "green", "blue"],
    color = "black";

with(colors) {
    push(color);
    push(...values);
}

console.log(colors);    // ["red", "green", "blue", "black", 1, 2, 3]
```

In this example, the two calls to `push()` inside the `with` statement are equivalent to `colors.push()` because the `with` statement added `push` as a local binding. The `color` reference refers to the variable created outside the `with` statement, as does the `values` reference.

But ECMAScript 6 added a `values` method to arrays. (The `values` method is discussed in detail in Chapter 7, "Iterators and Generators.") That would mean in an ECMAScript 6 environment, the `values` reference inside the `with` statement should refer not to the local variable `values`, but to the array's `values` method, which would break the code. This is why the `Symbol.unscopables` symbol exists.

The `Symbol.unscopables` symbol is used on `Array.prototype` to indicate which properties shouldn't create bindings inside of a `with` statement. When present, `Symbol.unscopables` is an object whose keys are the identifiers to omit from `with` statement bindings and whose values are `true` to enforce the block. Here's the default `Symbol.unscopables` property for arrays:

```
// built into ECMAScript 6 by default
Array.prototype[Symbol.unscopables] = Object.assign(Object.create(null), {
    copyWithin: true,
    entries: true,
    fill: true,
    find: true,
    findIndex: true,
    keys: true,
    values: true
});
```

The `Symbol.unscopables` object has a `null` prototype, which is created by the `Object.create(null)` call, and contains all of the new array methods in ECMAScript 6. (These methods are covered in detail in Chapter 7, "Iterators and Generators," and Chapter 9, "Arrays.") Bindings for these methods are not created inside a `with` statement, allowing old code to continue working without any problem.

In general, you shouldn't need to define `Symbol.unscopables` for your objects unless you use the `with` statement and are making changes to an existing object in your code base.

Summary

Symbols are a new type of primitive value in JavaScript and are used to create properties that can't be accessed without referencing the symbol.

While not truly private, these properties are harder to accidentally change or overwrite and are therefore suitable for functionality that needs a level of protection from developers.

You can provide descriptions for symbols that allow for easier identification of symbol values. There is a global symbol registry that allows you to use shared symbols in different parts of code by using the same description. In this way, the same symbol can be used for the same reason in multiple places.

Methods like `Object.keys()` or `Object.getOwnPropertyNames()` don't return symbols, so a new method called `Object.getOwnPropertySymbols()` was added in ECMAScript 6 to allow retrieval of symbol properties. You can still make changes to symbol properties by calling the `Object.defineProperty()` and `Object.defineProperties()` methods.

Well-known symbols define previously internal-only functionality for standard objects and use globally-available symbol constants, such as the `Symbol.hasInstance` property. These symbols use the prefix `Symbol.` in the specification and allow developers to modify standard object behavior in a variety of ways.