The Ultimate Guide to JavaScript Symbol

Summary: in this tutorial, you will learn about the JavaScript symbol primitive type and how to use the symbol effectively.

Creating symbols

ES6 added `Symbol` as a new primitive type. Unlike other primitive types such as number, boolean, null, undefined, and string, the symbol type doesn't have a literal form.

To create a new symbol, you use the global `Symbol()` function as shown in this example:

```
let s = Symbol('foo');
```

The `Symbol()` function creates a new *unique* value each time you call it:

```
console.log(Symbol() === Symbol()); // false
```

The `Symbol()` function accepts a `description` as an optional argument. The `description` argument will make your symbol more descriptive.

The following example creates two symbols: `firstName` and `lastName`.

```
let firstName = Symbol('first name'),
    lastName = Symbol('last name');
```

You can access the symbol's description property using the `toString()` method. The `console.log()` method calls the `toString()` method of the symbol implicitly as shown in the following example:

```
console.log(firstName); // Symbol(first name)
console.log(lastName); // Symbol(last name)
```

Since symbols are primitive values, you can use the `typeof` operator to check whether a variable is a symbol. ES6 extended `typeof` to return the `symbol` string when you pass in a symbol variable:

```
console.log(typeof firstName); // symbol
```

Since a symbol is a primitive value, if you attempt to create a symbol using the `new` operator, you will get an error:

```
let s = new Symbol(); // error
```

Sharing symbols

ES6 provides you with a global symbol registry that allows you to share symbols globally. If you want to create a symbol that will be shared, you use the `Symbol.for()` method instead of calling the `Symbol()` function.

The `Symbol.for()` method accepts a single parameter that can be used for symbol's description, as shown in the following example:

```
let ssn = Symbol.for('ssn');
```

The `Symbol.for()` method first searches for the symbol with the `ssn` key in the global symbol registry. It returns the existing symbol if there is one. Otherwise, the `Symbol.for()` method creates a new symbol, registers it to the global symbol registry with the specified key, and returns the symbol.

Later, if you call the `Symbol.for()` method using the same key, the `Symbol.for()` method will return the existing symbol.

```
let citizenID = Symbol.for('ssn');
```

```
console.log(ssn === citizenID); // true
```

In this example, we used the `Symbol.for()` method to look up the symbol with the `ssn` key. Since the global symbol registry already contained it, the `Symbol.for()` method returned the existing symbol.

To get the key associated with a symbol, you use the `Symbol.keyFor()` method as shown in the following example:

```
console.log(Symbol.keyFor(citizenID)); // 'ssn'
```

If a symbol does not exist in the global symbol registry, the `System.keyFor()` method returns `undefined`.

```
let systemID = Symbol('sys');
console.log(Symbol.keyFor(systemID)); // undefined
```

Symbol usages

A) Using symbols as unique values

Whenever you use a string or a number in your code, you should use symbols instead. For example, you have to manage the status in the task management application.

Before ES6, you would use strings such as `open`, `in progress`, `completed`, `canceled`, and `on hold` to represent different statuses of a task. In ES6, you can use symbols as follows:

```
let statuses = {
    OPEN: Symbol('Open'),
    IN_PROGRESS: Symbol('In progress'),
    COMPLETED: Symbol('Completed'),
    HOLD: Symbol('On hold'),
    CANCELED: Symbol('Canceled')
};
// complete a task
task.setStatus(statuses.COMPLETED);
```

B) Using a symbol as the computed property name of an object

You can use symbols as computed property names. See the following example:

```
let status = Symbol('status');
let task = {
    [status]: statuses.OPEN,
    description: 'Learn ES6 Symbol'
};
console.log(task);
```

To get all the enumerable properties of an object, you use the `Object.keys()` method.

```
console.log(Object.keys(task)); // ["description"]
```

To get all properties of an object whether the properties are enumerable or not, you use the `Object.getOwnPropertyNames()` method.

```
console.log(Object.getOwnPropertyNames(task)); // ["description"]
```

To get all property symbols of an object, you use the `Object.getOwnPropertySymbols()` method, which has been added in ES6.

```
console.log(Object.getOwnPropertySymbols(task)); //[Symbol(status)]
```

The `Object.getOwnPropertySymbols()` method returns an array of own property symbols from an object.

Well-known symbols

ES6 provides predefined symbols which are called well-known symbols. The well-known symbols represent the common behaviors in JavaScript. Each well-known symbol is a static property of the `Symbol` object.

Symbol.hasInstance

The `Symbol.hasInstance` is a symbol that changes the behavior of the `instanceof` operator. Typically, when you use the `instanceof` operator:

```
obj instanceof type;
```

JavaScript will call the `Symbol.hasIntance` method as follows:

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

It then depends on the method to determine if `obj` is an instance of the `type` object. See the following example.

```
class Stack {
}
console.log([] instanceof Stack); // false
```

The `[]` array is not an instance of the `Stack` class, therefore, the `instanceof` operator returns `false` in this example.

Assuming that you want the `[]` array is an instance of the `Stack` class, you can add the `Symbol.hasInstance` method as follows:

```
class Stack {
    static [Symbol.hasInstance](obj) {
        return Array.isArray(obj);
    }
}
console.log([] instanceof Stack); // true
```

Symbol.iterator

The `Symbol.iterator` specifies whether a function will return an iterator for an object.

The objects that have `Symbol.iterator` property are called iterable objects.

In ES6, all collection objects (Array, Set and Map) and strings are iterable objects.

ES6 provides the for...of loop that works with the iterable object as in the following example.

```
var numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}

// 1
// 2
// 3
```

Internally, the JavaScript engine first calls the `Symbol.iterator` method of the numbers array to get the iterator object.

Then, it invokes the `iterator.next()` method and copies the value property of the iterator object into the `num` variable.

After three iterations, the `done` property of the result object is `true`, the loop exits.

You can access the default iterator object via `System.iterator` symbol as follows:

```
var iterator = numbers[Symbol.iterator]();

console.log(iterator.next()); // Object {value: 1, done: false}
console.log(iterator.next()); // Object {value: 2, done: false}
console.log(iterator.next()); // Object {value: 3, done: false}
console.log(iterator.next()); // Object {value: undefined, done: true}
```

By default, a collection is not iterable. However, you can make it iterable by using the `Symbol.iterator` as shown in the following example:

```
class List {
    constructor() {
        this.elements = [];
    }
```

```
    add(element) {
        this.elements.push(element);
        return this;
    }

    *[Symbol.iterator]() {
        for (let element of this.elements) {
            yield  element;
        }
    }
}

let chars = new List();
chars.add('A')
     .add('B')
     .add('C');

// because of the Symbol.iterator
for (let c of chars) {
    console.log(c);
}

// A
// B
// C
```

Symbol.isConcatSpreadable

To concatenate two arrays, you use the `concat()` method as shown in the following example:

```
let odd  = [1, 3],
    even = [2, 4];
let all = odd.concat(even);
console.log(all); // [1, 3, 2, 4]
```

In this example, the resulting array contains the single elements of both arrays. In addition, the `concat()` method also accepts a non-array argument as illustrated below.

```
let extras = all.concat(5);
console.log(extras); // [1, 3, 2, 4, 5]
```

The number 5 becomes the fifth element of the array.

As you can see in the above example when we pass an array to the `concat()` method, the `concat()` method spreads the array into individual elements. However, it treats a single primitive argument differently. Prior to ES6, you could not change this behavior.

This is why the `Symbol.isConcatSpreadable` symbol comes into play.

The `Symbol.isConcatSpreadable` property is a Boolean value that determines whether an object is added individually to the result of the `concat()` function.

Consider the following example:

```
let list = {
    0: 'JavaScript',
    1: 'Symbol',
    length: 2
};
let message = ['Learning'].concat(list);
console.log(message); // ["Learning", Object]
```

The list object is concatenated to the `['Learning']` array. However, its individual elements are not spreaded.

To enable the elements of the `list` object added to the array individually when passing to the `concat()` method, you need to add the `Symbol.isConcatSpreadable` property to the `list` object as follows:

```
let list = {
    0: 'JavaScript',
    1: 'Symbol',
    length: 2,
    [Symbol.isConcatSpreadable]: true
};
```

```
let message = ['Learning'].concat(list);
console.log(message); // ["Learning", "JavaScript", "Symbol"]
```

Note that if you set the value of the `Symbol.isConcatSpreadable` to `false` and pass the `list` object to the `concat()` method, it will be concatenated to the array as the whole object.

Symbol.toPrimitive

The `Symbol.toPrimitive` method determines what should happen when an object is converted into a primitive value.

The JavaScript engine defines the `Symbol.toPrimitive` method on the prototype of each standard type.

The `Symbol.toPrimitive` method takes a `hint` argument that has one of three values: “number”, “string”, and “default”. The `hint` argument specifies the type of the return value. The `hint` parameter is filled by the JavaScript engine based on the context in which the object is used.

Here is an example of using the `Symbol.toPrimitive` method.

```
function Money(amount, currency) {
    this.amount = amount;
    this.currency = currency;
}
Money.prototype[Symbol.toPrimitive] = function(hint) {
    var result;
    switch (hint) {
        case 'string':
            result = this.amount + this.currency;
            break;
        case 'number':
            result = this.amount;
            break;
        case 'default':
            result = this.amount + this.currency;
            break;
    }
    return result;
```

```
}

var price = new Money(799, 'USD');

console.log('Price is ' + price); // Price is 799USD
console.log(+price + 1); // 800
console.log(String(price)); // 799USD
```

In this tutorial, you have learned about JavaScript symbols and how to use symbols for unique values and object properties. Also, you learned how to use well-known symbols to modify object behaviors.