Symbols

Learn the new data type introduced by ES2015 - symbols. Learn how to use its creating function and its values.

The symbol is an entirely new data type introduced in ES2015. It's primitive, like numbers and strings. Its behavior is quite odd and it serves a small use case. Let's dive right in.

Introduction

Every symbol is unique. Running the `Symbol()` function produces dynamically unique values every time. Both equality operators, `==` and `===`, will always return `false` when comparing two symbols. This is vital to its behavior and its purpose.

Symbols are anonymous. That is, we can't ever see the value that they contain. It's hidden in the JavaScript engine. This makes their uses quite limited.

In fact, the only practical purpose a symbol has is serving as a key for a property in an object. Before ES2015, the only data type acceptable for an object's key was a string. Now, object properties can be strings or symbols.

Creating a Symbol

```
const sym = Symbol();
```

The quirks begin here. Although the function begins with a capital letter, it's not a constructor; `new Symbol()` will throw an error.

```
const sym = Symbol();
console.log(sym); // -> Symbol()

const sym2 = new Symbol();
// -> TypeError: Symbol is not a constructor
```

Note that logging a symbol produces `Symbol()`. We can't see what the symbol actually looks like. This is all we'll ever get.

Symbol Usage

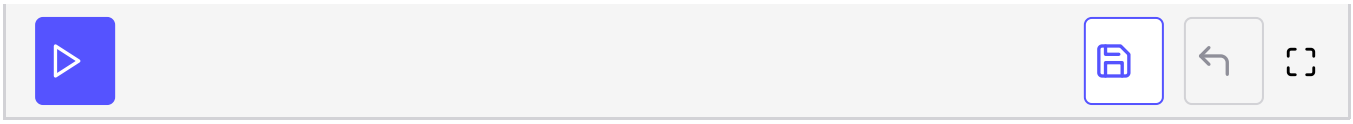
As mentioned, the only purpose a symbol has is to be used as a key for an object property. The symbol can later be used to access the property.

When a symbol is used as a key, it doesn't show up when iterating through the object.

```
const sym = Symbol();
const obj = {
    [sym]: 'I\'m a symbol.'
};

console.log(obj); // -> { [Symbol()]: 'I\'m a symbol.' }

for(const key in obj) {
  console.log(key, obj[key]);
} // -> Ø
```



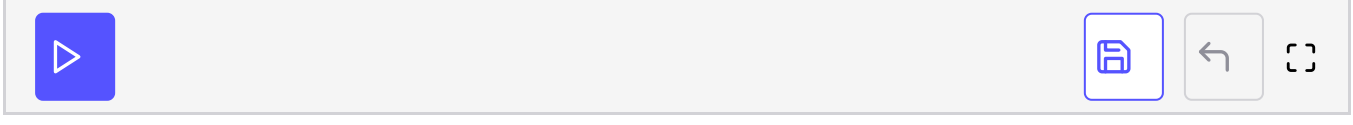
`Object.getOwnPropertyNames` is a function that takes in an object and returns its own properties. This function won't reveal symbol property names. `Object.keys` and `JSON.stringify` won't work either. To reveal the symbols present in an object, we must use `Object.getOwnPropertySymbols()`.

```
const sym = Symbol();
const obj = {
    prop: 'some property',
    [sym]: 'I\'m a symbol.'
};

console.log(Object.keys(obj)); // -> [ 'prop' ]
console.log(Object.getOwnPropertyNames(obj)); // -> [ 'prop' ]

console.log(JSON.stringify(obj)); // -> {"prop":"some property"}

console.log(Object.getOwnPropertySymbols(obj)); // -> [ Symbol() ]
```



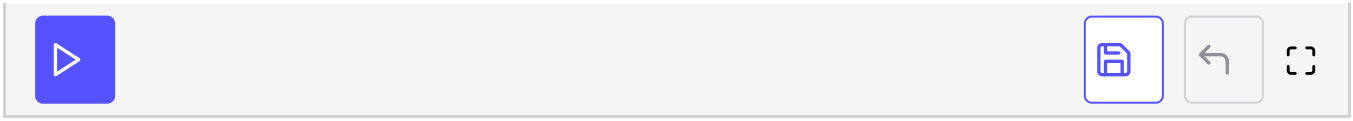
Global Symbols

There's a way to create global symbols that can be accessed anywhere across a program. We can do this without polluting the global scope with variables. We can use `Symbol.for()`.

This function takes in a string and generates a symbol for that string. Every time we pass in that string, it'll return the samy symbol.

```
const sym = Symbol.for('some string');
const sym2 = Symbol.for('some string');

console.log(sym, sym2); // -> Symbol(some string) Symbol(some string)
console.log(sym === sym2); // -> true
```



Functionality

We pass in a string to `Symbol.for`. If it's the first time we're passing in that string, the function will create a new symbol, tie it to that string, and save it in the global symbol registry. It'll return that symbol.

Every time we use `Symbol.for`, the engine performs a lookup to determine if it's already created a symbol for the string. If so, it'll simply return that symbol instead of creating one.

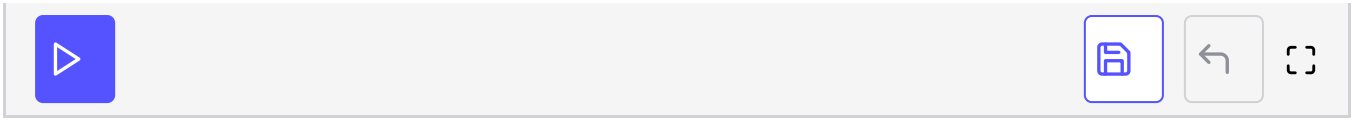
Tagged Symbols

This allows us to keep track of symbols with strings. We can't view a symbol directly, but we can essentially give our symbols a label to identify them.

We can also see which key created a symbol using the function `Symbol.keyFor`. If we previously created a symbol using `Symbol.for`, passing that symbol into `Symbol.keyFor` will return the string we used.

The method `Symbol.toString()`, called directly on a symbol, does something similar.

```
const sym = Symbol.for('some string');
console.log(Symbol.keyFor(sym)); // -> some string
console.log(sym.toString()); // -> Symbol(some string)
```



Use Case

A decent use case for symbols is to extend the functionality of an object. By using symbols, we can do so in a way that doesn't interfere with other aspects of the object.

For example, if we want to add utility methods on a class without disrupting the class's enumerable properties, a symbol is a good way to do so. Symbols won't show up when we iterate over the object and are a fairly quiet and minimal way to modify an object.

`Symbol.iterator`

There's a global symbol available to us placed directly on the global symbol function: `Symbol.iterator`. This special symbol is used to specify an iterator, something that we'll cover in the next lesson.

That's it for symbols.