11th February 2020

Symbol type

By specification, object property keys may be either of string type, or of symbol type. Not numbers, not booleans, only strings or symbols, these two types.

Till now we've been using only strings. Now let's see the benefits that symbols can give us.

Symbols

A "symbol" represents a unique identifier.

A value of this type can be created using `Symbol()`:

```
1  // id is a new symbol
2  let id = Symbol();
```

Upon creation, we can give symbol a description (also called a symbol name), mostly useful for debugging purposes:

```
1  // id is a symbol with the description "id"
2  let id = Symbol("id");
```

Symbols are guaranteed to be unique. Even if we create many symbols with the same description, they are different values. The description is just a label that doesn't affect anything.

For instance, here are two symbols with the same description – they are not equal:

```
1  let id1 = Symbol("id");
2  let id2 = Symbol("id");
3
4  alert(id1 == id2); // false
```

If you are familiar with Ruby or another language that also has some sort of "symbols" – please don't be misguided. JavaScript symbols are different.

Symbols don’t auto-convert to a string

Most values in JavaScript support implicit conversion to a string. For instance, we can `alert` almost any value, and it will work. Symbols are special. They don’t auto-convert.

For instance, this `alert` will show an error:

```
let id = Symbol("id");
alert(id); // TypeError: Cannot convert a Symbol value to a string
```

That’s a “language guard” against messing up, because strings and symbols are fundamentally different and should not accidentally convert one into another.

If we really want to show a symbol, we need to explicitly call `.toString()` on it, like here:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), now it works
```

Or get `symbol.description` property to show the description only:

```
let id = Symbol("id");
alert(id.description); // id
```

“Hidden” properties

Symbols allow us to create “hidden” properties of an object, that no other part of code can accidentally access or overwrite.

For instance, if we’re working with `user` objects, that belong to a third-party code. We’d like to add identifiers to them.

Let’s use a symbol key for it:

```
let user = { // belongs to another code
  name: "John"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // we can access the data using the symbol as the key
```

What’s the benefit of using `Symbol("id")` over a string `"id"` ?

As `user` objects belongs to another code, and that code also works with them, we shouldn’t just add any fields to it. That’s unsafe. But a symbol cannot be accessed accidentally, the third-party code probably won’t even see it, so it’s probably all right to do.

Also, imagine that another script wants to have its own identifier inside `user`, for its own purposes. That may be another JavaScript library, so that the scripts are completely unaware of each other.

Then that script can create its own `Symbol("id")`, like this:

```
// ...
let id = Symbol("id");

user[id] = "Their id value";
```

There will be no conflict between our and their identifiers, because symbols are always different, even if they have the same name.

…But if we used a string `"id"` instead of a symbol for the same purpose, then there *would* be a conflict:

```
let user = { name: "John" };

// Our script uses "id" property
user.id = "Our id value";

// ...Another script also wants "id" for its purposes...

user.id = "Their id value"
// Boom! overwritten by another script!
```

Symbols in a literal

If we want to use a symbol in an object literal `{...}`, we need square brackets around it.

Like this:

```
let id = Symbol("id");

let user = {
  name: "John",
  [id]: 123 // not "id: 123"
};
```

That's because we need the value from the variable `id` as the key, not the string "id".

Symbols are skipped by for…in

Symbolic properties do not participate in `for..in` loop.

For instance:

```
let id = Symbol("id");
let user = {
  name: "John",
  age: 30,
```

```
  [id]: 123
};

for (let key in user) alert(key); // name, age (no symbols)

// the direct access by the symbol works
alert( "Direct: " + user[id] );
```

`Object.keys(user)` also ignores them. That's a part of the general "hiding symbolic properties" principle. If another script or a library loops over our object, it won't unexpectedly access a symbolic property.

In contrast, Object.assign copies both string and symbol properties:

```
let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123
```

There's no paradox here. That's by design. The idea is that when we clone an object or merge objects, we usually want *all* properties to be copied (including symbols like `id`).

Global symbols

As we've seen, usually all symbols are different, even if they have the same name. But sometimes we want same-named symbols to be same entities. For instance, different parts of our application want to access symbol `"id"` meaning exactly the same property.

To achieve that, there exists a *global symbol registry*. We can create symbols in it and access them later, and it guarantees that repeated accesses by the same name return exactly the same symbol.

In order to read (create if absent) a symbol from the registry, use `Symbol.for(key)`.

That call checks the global registry, and if there's a symbol described as `key`, then returns it, otherwise creates a new symbol `Symbol(key)` and stores it in the registry by the given `key`.

For instance:

```
// read from the global registry
let id = Symbol.for("id"); // if the symbol did not exist, it is created

// read it again (maybe from another part of the code)
let idAgain = Symbol.for("id");

// the same symbol
alert( id === idAgain ); // true
```

Symbols inside the registry are called *global symbols*. If we want an application-wide symbol, accessible everywhere in the code – that's what they are for.

That sounds like Ruby

In some programming languages, like Ruby, there's a single symbol per name.

In JavaScript, as we can see, that's right for global symbols.

Symbol.keyFor

For global symbols, not only `Symbol.for(key)` returns a symbol by name, but there's a reverse call: `Symbol.keyFor(sym)`, that does the reverse: returns a name by a global symbol.

For instance:

```
1  // get symbol by name
2  let sym = Symbol.for("name");
3  let sym2 = Symbol.for("id");
4
5  // get name by symbol
6  alert( Symbol.keyFor(sym) ); // name
7  alert( Symbol.keyFor(sym2) ); // id
```

The `Symbol.keyFor` internally uses the global symbol registry to look up the key for the symbol. So it doesn't work for non-global symbols. If the symbol is not global, it won't be able to find it and returns `undefined`.

That said, any symbols have `description` property.

For instance:

```
1  let globalSymbol = Symbol.for("name");
2  let localSymbol = Symbol("name");
3
4  alert( Symbol.keyFor(globalSymbol) ); // name, global symbol
5  alert( Symbol.keyFor(localSymbol) ); // undefined, not global
6
7  alert( localSymbol.description ); // name
```

System symbols

There exist many "system" symbols that JavaScript uses internally, and we can use them to fine-tune various aspects of our objects.

They are listed in the specification in the Well-known symbols table:

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
- …and so on.

For instance, `Symbol.toPrimitive` allows us to describe object to primitive conversion. We'll see its use very soon.

Other symbols will also become familiar when we study the corresponding language features.

Summary

`Symbol` is a primitive type for unique identifiers.

Symbols are created with `Symbol()` call with an optional description (name).

Symbols are always different values, even if they have the same name. If we want same-named symbols to be equal, then we should use the global registry: `Symbol.for(key)` returns (creates if needed) a global symbol with `key` as the name. Multiple calls of `Symbol.for` with the same `key` return exactly the same symbol.

Symbols have two main use cases:

1. "Hidden" object properties. If we want to add a property into an object that "belongs" to another script or a library, we can create a symbol and use it as a property key. A symbolic property does not appear in `for..in`, so it won't be accidentally processed together with other properties. Also it won't be accessed directly, because another script does not have our symbol. So the property will be protected from accidental use or overwrite.

   So we can "covertly" hide something into objects that we need, but others should not see, using symbolic properties.

2. There are many system symbols used by JavaScript which are accessible as `Symbol.*`. We can use them to alter some built-in behaviors. For instance, later in the tutorial we'll use `Symbol.iterator` for iterables, `Symbol.toPrimitive` to setup object-to-primitive conversion and so on.

Technically, symbols are not 100% hidden. There is a built-in method Object.getOwnPropertySymbols(obj) that allows us to get all symbols. Also there is a method named Reflect.ownKeys(obj) that returns *all* keys of an object including symbolic ones. So they are not really hidden. But most libraries, built-in functions and syntax constructs don't use these methods.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)