**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 only seen strings. Now let’s see the advantages that symbols can give us.

**[Symbols](https://javascript.info/symbol" \l "symbols)**

“Symbol” value represents a unique identifier.

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

// id is a new symbol

let id = Symbol();

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

// id is a symbol with the description "id"

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:

let id1 = Symbol("id");

let id2 = Symbol("id");

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 occasionally convert one into another.

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

let id = Symbol("id");

alert(id.toString()); // Symbol(id), now it works

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

let id = Symbol("id");

alert(id.description); // id

**[“Hidden” properties](https://javascript.info/symbol" \l "hidden-properties)**

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

For instance, if we want to store an “identifier” for the object user, we can use a symbol as a key for it:

let user = { name: "John" };

let id = Symbol("id");

user[id] = "ID Value";

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"?

Let’s make the example a bit deeper to see that.

Imagine that another script wants to have its own “id” property inside user, for its own purposes. That may be another JavaScript library, so 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, because symbols are always different, even if they have the same name.

Now note that 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 = "ID Value";

// ...if later another script the uses "id" for its purposes...

user.id = "Their id value"

// boom! overwritten! it did not mean to harm the colleague, but did it!

**[Symbols in a literal](https://javascript.info/symbol" \l "symbols-in-a-literal)**

If we want to use a symbol in an object literal, we need square brackets.

Like this:

let id = Symbol("id");

let user = {

name: "John",

[id]: 123 // not just "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](https://javascript.info/symbol" \l "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] );

That’s a part of the general “hiding” concept. If another script or a library loops over our object, it won’t unexpectedly access a symbolic property.

In contrast, [Object.assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/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).

**Property keys of other types are coerced to strings**

We can only use strings or symbols as keys in objects. Other types are converted to strings.

For instance, a number 0 becomes a string "0" when used as a property key:

let obj = {

0: "test" // same as "0": "test"

};

// both alerts access the same property (the number 0 is converted to string "0")

alert( obj["0"] ); // test

alert( obj[0] ); // test (same property)

**[Global symbols](https://javascript.info/symbol" \l "global-symbols)**

As we’ve seen, usually all symbols are different, even if they have the same names. 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 create or read a symbol in 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

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](https://javascript.info/symbol" \l "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:

let sym = Symbol.for("name");

let sym2 = Symbol.for("id");

// get name from symbol

alert( Symbol.keyFor(sym) ); // name

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 return undefined.

For instance:

alert( Symbol.keyFor(Symbol.for("name")) ); // name, global symbol

alert( Symbol.keyFor(Symbol("name2")) ); // undefined, the argument isn't a global symbol

**[System symbols](https://javascript.info/symbol" \l "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](https://tc39.github.io/ecma262/#sec-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](https://javascript.info/symbol" \l "summary)**

Symbol is a primitive type for unique identifiers.

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

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 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 occasionally listed. Also it won’t be accessed directly, because another script does not have our symbol, so it will not occasionally intervene into its actions.

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

1. 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](https://javascript.info/iterable), Symbol.toPrimitive to setup [object-to-primitive conversion](https://javascript.info/object-toprimitive) and so on.

Technically, symbols are not 100% hidden. There is a built-in method [Object.getOwnPropertySymbols(obj)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertySymbols) that allows us to get all symbols. Also there is a method named [Reflect.ownKeys(obj)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect/ownKeys) that returns *all* keys of an object including symbolic ones. So they are not really hidden. But most libraries, built-in methods and syntax constructs adhere to a common agreement that they are. And the one who explicitly calls the aforementioned methods probably understands well what he’s doing.