



[🏠](#) → [The JavaScript language](#) → [Objects: the basics](#)

Object to primitive conversion

What happens when objects are added `obj1 + obj2`, subtracted `obj1 - obj2` or printed using `alert(obj)` ?

There are special methods in objects that do the conversion.

In the chapter [Type Conversions](#) we've seen the rules for numeric, string and boolean conversions of primitives. But we left a gap for objects. Now, as we know about methods and symbols it becomes possible to close it.

For objects, there's no to-boolean conversion, because all objects are `true` in a boolean context. So there are only string and numeric conversions.

The numeric conversion happens when we subtract objects or apply mathematical functions. For instance, `Date` objects (to be covered in the chapter [Date and time](#)) can be subtracted, and the result of `date1 - date2` is the time difference between two dates.

As for the string conversion – it usually happens when we output an object like `alert(obj)` and in similar contexts.

ToPrimitive

When an object is used in the context where a primitive is required, for instance, in an `alert` or mathematical operations, it's converted to a primitive value using the `ToPrimitive` algorithm ([specification](#)).

That algorithm allows us to customize the conversion using a special object method.

Depending on the context, the conversion has a so-called "hint".

There are three variants:

"string"

When an operation expects a string, for object-to-string conversions, like `alert` :

```
1 // output
2 alert(obj);
3
4 // using object as a property key
5 anotherObj[obj] = 123;
```

"number"

When an operation expects a number, for object-to-number conversions, like maths:

```
1 // explicit conversion
2 let num = Number(obj);
3
4 // maths (except binary plus)
5 let n = +obj; // unary plus
6 let delta = date1 - date2;
7
8 // less/greater comparison
9 let greater = user1 > user2;
```

"default"

Occurs in rare cases when the operator is "not sure" what type to expect.

For instance, binary plus `+` can work both with strings (concatenates them) and numbers (adds them), so both strings and numbers would do. Or when an object is compared using `==` with a string, number or a symbol.

```
1 // binary plus
2 let total = car1 + car2;
3
4 // obj == string/number/symbol
5 if (user == 1) { ... };
```

The greater/less operator `<>` can work with both strings and numbers too. Still, it uses "number" hint, not "default". That's for historical reasons.

In practice, all built-in objects except for one case (`Date` object, we'll learn it later) implement "default" conversion the same way as "number" . And probably we should do the same.

Please note – there are only three hints. It's that simple. There is no "boolean" hint (all objects are `true` in boolean context) or anything else. And if we treat "default" and "number" the same, like most built-ins do, then there are only two conversions.

To do the conversion, JavaScript tries to find and call three object methods:

1. Call `obj[Symbol.toPrimitive](hint)` if the method exists,
2. Otherwise if hint is "string"
 - try `obj.toString()` and `obj.valueOf()`, whatever exists.
3. Otherwise if hint is "number" or "default"
 - try `obj.valueOf()` and `obj.toString()`, whatever exists.

Symbol.toPrimitive

Let's start from the first method. There's a built-in symbol named `Symbol.toPrimitive` that should be used to name the conversion method, like this:

```
1 obj[Symbol.toPrimitive] = function(hint) {
2   // return a primitive value
3   // hint = one of "string", "number", "default"
4 }
```

For instance, here `user` object implements it:

```
1 let user = {
2   name: "John",
3   money: 1000,
4
5   [Symbol.toPrimitive](hint) {
6     alert(`hint: ${hint}`);
7     return hint == "string" ? `{name: "${this.name}"}` : this.money;
8   }
9 };
10
11 // conversions demo:
12 alert(user); // hint: string -> {name: "John"}
13 alert(+user); // hint: number -> 1000
14 alert(user + 500); // hint: default -> 1500
```

As we can see from the code, `user` becomes a self-descriptive string or a money amount depending on the conversion. The single method `user[Symbol.toPrimitive]` handles all conversion cases.

toString/valueOf

Methods `toString` and `valueOf` come from ancient times. They are not symbols (symbols did not exist that long ago), but rather "regular" string-named methods. They provide an alternative "old-style" way to implement the conversion.

If there's no `Symbol.toPrimitive` then JavaScript tries to find them and try in the order:

- `toString` -> `valueOf` for "string" hint.
- `valueOf` -> `toString` otherwise.

For instance, here `user` does the same as above using a combination of `toString` and `valueOf`:

```
1 let user = {
2   name: "John",
3   money: 1000,
4
5   // for hint="string"
6   toString() {
7     return `{name: "${this.name}"}`;
8   },
9
10  // for hint="number" or "default"
11  valueOf() {
12    return this.money;
13  }
14
15 };
16
```

```

17 alert(user); // toString -> {name: "John"}
18 alert(+user); // valueOf -> 1000
19 alert(user + 500); // valueOf -> 1500

```

Often we want a single “catch-all” place to handle all primitive conversions. In this case we can implement `toString` only, like this:

```

1 let user = {
2   name: "John",
3
4   toString() {
5     return this.name;
6   }
7 };
8
9 alert(user); // toString -> John
10 alert(user + 500); // toString -> John500

```

In the absence of `Symbol.toPrimitive` and `valueOf`, `toString` will handle all primitive conversions.

ToPrimitive and ToString/ToNumber

The important thing to know about all primitive-conversion methods is that they do not necessarily return the “hinted” primitive.

There is no control whether `toString()` returns exactly a string, or whether `Symbol.toPrimitive` method returns a number for a hint “number”.

The only mandatory thing: these methods must return a primitive.

An operation that initiated the conversion gets that primitive, and then continues to work with it, applying further conversions if necessary.

For instance:

- Mathematical operations (except binary plus) perform `ToNumber` conversion:

```

1 let obj = {
2   toString() { // toString handles all conversions in the absence of other methods
3     return "2";
4   }
5 };
6
7 alert(obj * 2); // 4, ToPrimitive gives "2", then it becomes 2

```

- Binary plus checks the primitive – if it’s a string, then it does concatenation, otherwise it performs `ToNumber` and works with numbers.

String example:

```

1 let obj = {
2   toString() {
3     return "2";
4   }
5 };
6
7 alert(obj + 2); // 22 (ToPrimitive returned string => concatenation)

```

Number example:

```

1 let obj = {
2   toString() {
3     return true;
4   }
5 };
6
7 alert(obj + 2); // 3 (ToPrimitive returned boolean, not string => ToNumber)

```

i Historical notes

For historical reasons, methods `toString` or `valueOf` *should* return a primitive: if any of them returns an object, then there's no error, but that object is ignored (like if the method didn't exist).

In contrast, `Symbol.prototype.toPrimitive` *must* return a primitive, otherwise, there will be an error.

Summary

The object-to-primitive conversion is called automatically by many built-in functions and operators that expect a primitive as a value.

There are 3 types (hints) of it:

- "string" (for `alert` and other string conversions)
- "number" (for maths)
- "default" (few operators)

The specification describes explicitly which operator uses which hint. There are very few operators that "don't know what to expect" and use the "default" hint. Usually for built-in objects "default" hint is handled the same way as "number", so in practice the last two are often merged together.

The conversion algorithm is:

1. Call `obj[Symbol.toPrimitive](hint)` if the method exists,
2. Otherwise if hint is "string"
 - try `obj.toString()` and `obj.valueOf()`, whatever exists.
3. Otherwise if hint is "number" or "default"
 - try `obj.valueOf()` and `obj.toString()`, whatever exists.

In practice, it's often enough to implement only `obj.toString()` as a "catch-all" method for all conversions that return a "human-readable" representation of an object, for logging or debugging purposes.

[Previous lesson](#)[Next lesson](#)

Share    

[Tutorial map](#)

Comments

- You're welcome to post additions, questions to the articles and answers to them.
- 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](#)...)
- If you can't understand something in the article – please elaborate.