7th November 2019

Object to primitive conversion

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

In that case, objects are auto-converted to primitives, and then the operation is carried out.

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 fill it.

1. All objects are `true` in a boolean context. There are only numeric and string conversions.
2. 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.
3. As for the string conversion – it usually happens when we output an object like `alert(obj)` and in similar contexts.

ToPrimitive

We can fine-tune string and numeric conversion, using special object methods.

There are three variants of type conversion, so-called "hints", described in the specification:

`"string"`

For an object-to-string conversion, when we're doing an operation on an object that expects a string, like `alert`:

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

`"number"`

For an object-to-number conversion, like when we're doing 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. So if the a binary plus gets an object as an argument, it uses the `"default"` hint to convert it.

Also, if an object is compared using `==` with a string, number or a symbol, it’s also unclear which conversion should be done, so the `"default"` hint is used.

```
1  // binary plus uses the "default" hint
2  let total = obj1 + obj2;
3
4  // obj == number uses the "default" hint
5  if (user == 1) { ... };
```

The greater and less comparison operators, such as `<` `>`, can work with both strings and numbers too. Still, they use the `"number"` hint, not `"default"`. That’s for historical reasons.

In practice though, we don’t need to remember these peculiar details, because all built-in objects except for one case (`Date` object, we’ll learn it later) implement `"default"` conversion the same way as `"number"`. And we can do the same.

No `"boolean"` hint

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)` – the method with the symbolic key `Symbol.toPrimitive` (system symbol), if such 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    // must return a primitive value
```

```
  // hint = one of "string", "number", "default"
};
```

For instance, here `user` object implements it:

```
let user = {
  name: "John",
  money: 1000,

  [Symbol.toPrimitive](hint) {
    alert(`hint: ${hint}`);
    return hint == "string" ? `{name: "${this.name}"}` : this.money;
  }
};

// conversions demo:
alert(user); // hint: string -> {name: "John"}
alert(+user); // hint: number -> 1000
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.

These methods must return a primitive value. If `toString` or `valueOf` returns an object, then it's ignored (same as if there were no method).

By default, a plain object has following `toString` and `valueOf` methods:

- The `toString` method returns a string `"[object Object]"`.
- The `valueOf` method returns the object itself.

Here's the demo:

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

alert(user); // [object Object]
alert(user.valueOf() === user); // true
```

So if we try to use an object as a string, like in an `alert` or so, then by default we see `[object Object]`.

And the default `valueOf` is mentioned here only for the sake of completeness, to avoid any confusion. As you can see, it returns the object itself, and so is ignored. Don't ask me why, that's for historical reasons. So we can assume it doesn't exist.

Let's implement these methods.

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

```
let user = {
  name: "John",
  money: 1000,

  // for hint="string"
  toString() {
    return `{name: "${this.name}"}`;
  },

  // for hint="number" or "default"
  valueOf() {
    return this.money;
  }

};

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

As we can see, the behavior is the same as the previous example with `Symbol.toPrimitive`.

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

```
let user = {
  name: "John",

  toString() {
    return this.name;
  }
};

alert(user); // toString -> John
alert(user + 500); // toString -> John500
```

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

Return types

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, not an object.

> **Historical notes**
>
> For historical reasons, if `toString` or `valueOf` returns an object, there's no error, but such value is ignored (like if the method didn't exist). That's because in ancient times there was no good "error" concept in JavaScript.
>
> In contrast, `Symbol.toPrimitive` *must* return a primitive, otherwise there will be an error.

Further conversions

As we know already, many operators and functions perform type conversions, e.g. multiplication `*` converts operands to numbers.

If we pass an object as an argument, then there are two stages:

1. The object is converted to a primitive (using the rules described above).
2. If the resulting primitive isn't of the right type, it's converted.

For instance:

```
let obj = {
  // toString handles all conversions in the absence of other methods
  toString() {
    return "2";
  }
};

alert(obj * 2); // 4, object converted to primitive "2", then multiplication
```

1. The multiplication `obj * 2` first converts the object to primitive (that's a string `"2"`).
2. Then `"2" * 2` becomes `2 * 2` (the string is converted to number).

Binary plus will concatenate strings in the same situation, as it gladly accepts a string:

```
let obj = {
  toString() {
    return "2";
  }
};

alert(obj + 2); // 22 ("2" + 2), conversion to primitive returned a string =>
```

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 operations that need a string)
- `"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

- 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…)