**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](https://javascript.info/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](https://javascript.info/date)) 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](https://javascript.info/object-toprimitive" \l "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](https://tc39.github.io/ecma262/#sec-toprimitive)).

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:

// output

alert(obj);

// using object as a property key

anotherObj[obj] = 123;

**"number"**

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

// explicit conversion

let num = Number(obj);

// maths (except binary plus)

let n = +obj; // unary plus

let delta = date1 - date2;

// less/greater comparison

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.

// binary plus

let total = car1 + car2;

// obj == string/number/symbol

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](https://javascript.info/object-toprimitive" \l "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:

obj[Symbol.toPrimitive] = function(hint) {

// 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](https://javascript.info/object-toprimitive" \l "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:

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

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.

**[ToPrimitive and ToString/ToNumber](https://javascript.info/object-toprimitive" \l "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:
* let obj = {
* toString() { // toString handles all conversions in the absence of other methods
* return "2";
* }
* };

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:

let obj = {

toString() {

return "2";

}

};

alert(obj + 2); // 22 (ToPrimitive returned string => concatenation)

Number example:

let obj = {

toString() {

return true;

}

};

alert(obj + 2); // 3 (ToPrimitive returned boolean, not string => ToNumber)

**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.toPrimitive *must* return a primitive, otherwise, there will be an error.

**[Summary](https://javascript.info/object-toprimitive" \l "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.