# Преобразование объектов в примитивы

Что произойдёт, если сложить два объекта `obj1 + obj2`, вычесть один из другого `obj1 - obj2` или вывести их на экран, воспользовавшись `alert(obj)`?

В этом случае объекты сначала автоматически преобразуются в примитивы, а затем выполняется операция.

В главе Преобразование типов мы видели правила для численных, строковых и логических преобразований. Но обделили вниманием объекты. Теперь, поскольку мы уже знаем о методах объектов и символах, можно исправить это упущение.

1. Все объекты в логическом контексте являются `true`. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда мы вычитаем объекты или выполняем математические операции. Например, объекты `Date` (мы рассмотрим их в статье Дата и время) могут вычитаться, и результатом `date1 - date2` будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, когда мы выводим объект `alert(obj)`, а также в других случаях, когда объект используется как строка.

## Преобразование к примитивам

Мы можем тонко настраивать строковые и численные преобразования, используя специальные методы объекта.

Существуют три варианта преобразований («три хинта»), описанные в спецификации:

**`"string"`**

Для преобразования объекта к строке, когда операция ожидает получить строку, например `alert`:

```
// вывод
alert(obj);

// используем объект в качестве имени свойства
anotherObj[obj] = 123;
```

**`"number"`**

Для преобразования объекта к числу, в случае математических операций:

```
// явное преобразование
let num = Number(obj);

// математическое (исключая бинарный оператор "+")
let n = +obj; // унарный плюс
let delta = date1 - date2;

// сравнения больше/меньше
let greater = user1 > user2;
```

**`"default"`**

Происходит редко, когда оператор «не уверен», какой тип ожидать.



Например, бинарный плюс `+` может работать с обоими типами: строками (объединять их) и числами (складывать). Таким образом, и те, и другие будут вычисляться. Или когда происходит сравнение объектов с помощью нестрогого равенства `==` со строкой, числом или символом, и неясно, какое преобразование должно быть выполнено.

```
// бинарный плюс
let total = car1 + car2;

// obj == string/number/symbol
if (user == 1) { ... };
```

Оператор больше/меньше `<>` также может работать как со строками, так и с числами. Однако, по историческим причинам он использует хинт «number», а не «default».

На практике все встроенные объекты, исключая `Date` (мы познакомимся с ним чуть позже), реализуют `"default"` преобразования тем же способом, что и `"number"`. И нам следует поступать так же.

Обратите внимание, что существуют лишь три варианта хинтов. Всё настолько просто. Не существует хинта со значением «boolean» (все объекты являются `true` в логическом контексте) или каких-либо ещё. И если мы считаем `"default"` и `"number"` одинаковыми, как большинство встроенных объектов, то остаются всего два варианта преобразований.

**В процессе преобразования движок JavaScript пытается найти и вызвать три следующих метода объекта:**

1. Вызывает `obj[Symbol.toPrimitive](hint)` – метод с символьным ключом `Symbol.toPrimitive` (системный символ), если такой метод существует, и передаёт ему хинт.
2. Иначе, если хинт равен `"string"`
   - пытается вызвать `obj.toString()`, а если его нет, то `obj.valueOf()`, если он существует.
3. В случае, если хинт равен `"number"` или `"default"`
   - пытается вызвать `obj.valueOf()`, а если его нет, то `obj.toString()`, если он существует.

## Symbol.toPrimitive

Начнём с универсального подхода – символа `Symbol.toPrimitive`: метод с таким названием (если есть) используется для всех преобразований:

```
obj[Symbol.toPrimitive] = function(hint) {
  // должен вернуть примитивное значение
  // hint равно чему-то одному из: "string", "number" и
};
```

Для примера используем его в реализации объекта `user`:

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

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

// демонстрация результатов преобразований:
alert(user); // hint: string -> {name: "John"}
alert(+user); // hint: number -> 1000
alert(user + 500); // hint: default -> 1500
```



Как мы видим из кода, `user` преобразовывается либо в информативную читаемую строку, либо в денежный счёт в зависимости от значения хинта. Единственный метод `user[Symbol.toPrimitive]` смог обработать все случаи преобразований.

## Методы toString/valueOf

Методы `toString` и `valueOf` берут своё начало с древних времён. Они не символы, так как в то время символов ещё не существовало, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода `Symbol.toPrimitive`, движок JavaScript пытается найти эти методы и вызвать их следующим образом:

- `toString -> valueOf` для хинта со значением «string».
- `valueOf -> toString` – в ином случае.

Для примера, используем их в реализации всё того же объекта `user`. Воспроизведём его поведение комбинацией методов `toString` и `valueOf`:

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

  // для хинта равного "string"
  toString() {
    return `{name: "${this.name}"}`;
  },

  // для хинта равного "number" или "default"
  valueOf() {
    return this.money;
  }

};

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

Как видим, получилось то же поведение, что и в предыдущем примере с `Symbol.toPrimitive`.

Довольно часто мы хотим описать одно «универсальное» преобразование объекта к примитиву для всех ситуаций. Для этого достаточно создать один `toString`:

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

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

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

В отсутствие `Symbol.toPrimitive` и `valueOf`, `toString` обработает все случаи преобразований к примитивам.

## Возвращаемые типы

Важно понимать, что все описанные методы для преобразований объектов не обязаны возвращать именно требуемый «хинтом» тип примитива.

Нет обязательного требования, чтобы `toString()` возвращал именно строку, или чтобы метод `Symbol.toPrimitive` возвращал именно число для хинта «number».

**Единственное обязательное требование: методы должны возвращать примитив, а не объект.**

> **Историческая справка**
>
> По историческим причинам, если `toString` или `valueOf` вернёт объект, то ошибки не будет, но такое значение будет проигнорировано (как если бы метода вообще не существовало).
>
> Метод `Symbol.toPrimitive`, напротив, *обязан* возвращать примитив, иначе будет ошибка.

## Последующие операции

Операция, инициировавшая преобразование, получает примитив и затем продолжает работу с ним, производя дальнейшие преобразования, если это необходимо.

Например:

- Математические операции, исключая бинарный плюс, преобразуют примитив к числу:

```
let obj = {
  // toString обрабатывает все преобразования в случа
  toString() {
    return "2";
  }
};

alert(obj * 2); // 4, объект был преобразован к прими
```

- Бинарный плюс `+` в аналогичном случае сложит строки:

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

alert(obj + 2); // 22 (преобразование к примитиву вер
```

## Итого

Преобразование объектов в примитивы вызывается автоматически многими встроенными функциями и операторами, которые ожидают примитив в качестве аргумента.

Существует всего 3 типа преобразований (хинтов):

- `"string"` (для `alert` и других операций, которым нужна строка)
- `"number"` (для математических операций)
- `"default"` (для некоторых операций)

В спецификации явно указано, какой хинт должен использовать каждый оператор. И существует совсем немного операторов, которые не знают, что ожидать, и используют хинт со значением `"default"`. Обычно для встроенных объектов хинт `"default"` обрабатывается так же, как `"number"`. Таким образом, последние два очень часто объединяют вместе.

Алгоритм преобразований к примитивам следующий:

1. Сначала вызывается метод `obj[Symbol.toPrimitive](hint)`, если он существует.
2. Иначе, если хинт равен `"string"`

- происходит попытка вызвать `obj.toString()`, затем `obj.valueOf()`, смотря что есть.

3. Иначе, если хинт равен `"number"` или `"default"`
   - происходит попытка вызвать `obj.valueOf()`, затем `obj.toString()`, смотря что есть.

На практике довольно часто достаточно реализовать только `obj.toString()` как «универсальный» метод для всех типов преобразований, возвращающий «читаемое» представление объекта, достаточное для логирования или отладки.





## Комментарии