Тип данных Symbol

По спецификации, в качестве ключей для свойств объекта могут использоваться только строки или символы. Ни числа, ни логические значения не подходят, разрешены только эти два типа данных.

До сих пор мы видели только строки. Теперь давайте разберём символы, увидим, что хорошего они нам дают.

Символы

«Символ» представляет собой уникальный идентификатор.

Создаются новые символы с помощью функции `Symbol()`:

```
1  // Создаём новый символ - id
2  let id = Symbol();
```

При создании символу можно дать описание (также называемое имя), в основном использующееся для отладки кода:

```
1  // Создаём символ id с описанием (именем) "id"
2  let id = Symbol("id");
```

Символы гарантированно уникальны. Даже если мы создадим множество символов с одинаковым описанием, это всё равно будут разные символы. Описание – это просто метка, которая ни на что не влияет.

Например, вот два символа с одинаковым описанием – но они не равны:

```
1  let id1 = Symbol("id");
2  let id2 = Symbol("id");
3
4  alert(id1 == id2); // false
```

Если вы знаете Ruby или какой-то другой язык программирования, в котором есть своего рода «символы» – пожалуйста, будьте внимательны. Символы в JavaScript имеют свои особенности, и не стоит думать о них, как о символах в Ruby или в других языках.

Символы не преобразуются автоматически в строки

Большинство типов данных в JavaScript могут быть неявно преобразованы в строку. Например, функция `alert` принимает практически любое значение, автоматически преобразовывает его в строку, а затем выводит это значение, не сообщая об ошибке. Символы же особенные и не преобразуются автоматически.

К примеру, `alert` ниже выдаст ошибку:

```
1  let id = Symbol("id");
2  alert(id); // TypeError: Cannot convert a Symbol \
```

Это – языковая «защита» от путаницы, ведь строки и символы – принципиально разные типы данных и не должны неконтролируемо преобразовываться друг в друга.

Если же мы действительно хотим вывести символ с помощью `alert`, то необходимо явно преобразовать его с помощью метода `.toString()`, вот так:

```
1  let id = Symbol("id");
2  alert(id.toString()); // Symbol(id), теперь работа
```

Или мы можем обратиться к свойству `symbol.description`, чтобы вывести только описание:

```
1  let id = Symbol("id");
2  alert(id.description); // id
```

«Скрытые» свойства

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы.

Например, мы работаем с объектами `user`, которые принадлежат стороннему коду. Мы хотим добавить к ним идентификаторы.

Используем для этого символьный ключ:

```
1  let user = {
2    name: "Вася"
3  };
4
5  let id = Symbol("id");
6
7  user[id] = 1;
8
9  alert( user[id] ); // мы можем получить доступ к данным
```

Почему же лучше использовать `Symbol("id")`, а не строку `"id"`?

Так как объект `user` принадлежит стороннему коду, и этот код также работает с ним, то нам не следует добавлять к нему какие-либо поля. Это небезопасно. Но к символу сложно нечаянно обратиться, сторонний код вряд ли его вообще увидит, и, скорее всего, добавление поля к объекту не вызовет никаких проблем.

Кроме того, предположим, что другой скрипт для каких-то своих целей хочет записать собственный идентификатор в объект `user`. Этот скрипт может быть какой-то JavaScript-библиотекой, абсолютно не связанной с нашим скриптом.

Сторонний код может создать для этого свой символ `Symbol("id")`:

```
// ...
let id = Symbol("id");

user[id] = "Их идентификатор";
```

Конфликта между их и нашим идентификатором не будет, так как символы всегда уникальны, даже если их имена совпадают.

А вот если бы мы использовали строку `"id"` вместо символа, то тогда *был бы* конфликт:

```
let user = { name: "Вася" };

// Объявляем в нашем скрипте свойство "id"
user.id = "Наш идентификатор";

// ...другой скрипт тоже хочет свой идентификатор...

user.id = "Их идентификатор"
// Ой! Свойство перезаписано сторонней библиотекой!
```

Символы в литеральном объекте

Если мы хотим использовать символ при литеральном объявлении объекта `{...}`, его необходимо заключить в квадратные скобки.

Вот так:

```
let id = Symbol("id");

let user = {
  name: "Вася",
  [id]: 123 // просто "id: 123" не сработает
};
```

Это вызвано тем, что нам нужно использовать значение переменной `id` в качестве ключа, а не строку «id».

Символы игнорируются циклом for...in

Свойства, чьи ключи – символы, не перебираются циклом `for..in`.

Например:

```
let id = Symbol("id");
let user = {
  name: "Вася",
  age: 30,
  [id]: 123
};

for (let key in user) alert(key); // name, age (свойств

// хотя прямой доступ по символу работает
alert( "Напрямую: " + user[id] );
```

Это – часть общего принципа «сокрытия символьных свойств». Если другая библиотека или скрипт будут работать с нашим объектом, то при переборе они не получат ненароком наше символьное свойство.
`Object.keys(user)` также игнорирует символы.

А вот Object.assign, в отличие от цикла `for..in`, копирует и строковые, и символьные свойства:

```
let id = Symbol("id");
let user = {
```

```
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123
```

Здесь нет никакого парадокса или противоречия. Так и задумано. Идея заключается в том, что, когда мы клонируем или объединяем объекты, мы обычно хотим скопировать *все* свойства (включая такие свойства с ключами-символами, как, например, `id` в примере выше).

Глобальные символы

Итак, как мы видели, обычно все символы уникальны, даже если их имена совпадают. Но иногда мы наоборот хотим, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части нашего приложения хотят получить доступ к символу `"id"`, подразумевая именно одно и то же свойство.

Для этого существует *глобальный реестр символов*. Мы можем создавать в нём символы и обращаться к ним позже, и при каждом обращении нам гарантированно будет возвращаться один и тот же символ.

Для чтения (или, при отсутствии, создания) символа из реестра используется вызов `Symbol.for(key)`.

Он проверяет глобальный реестр и, при наличии в нём символа с именем `key`, возвращает его, иначе же создаётся новый символ `Symbol(key)` и записывается в реестр под ключом `key`.

Например:

```
// читаем символ из глобального реестра и записываем ег
let id = Symbol.for("id"); // если символа не существуе

// читаем его снова в другую переменную (возможно, из д
let idAgain = Symbol.for("id");

// проверяем -- это один и тот же символ
alert( id === idAgain ); // true
```

Символы, содержащиеся в реестре, называются *глобальными символами*. Если вам нужен символ, доступный везде в коде – используйте глобальные символы.

> **Похоже на Ruby**
>
> В некоторых языках программирования, например, Ruby, на одно имя (описание) приходится один символ, и не могут существовать разные символы с одинаковым именем.
>
> В JavaScript, как мы видим, это утверждение верно только для глобальных символов.

Symbol.keyFor

Для глобальных символов, кроме `Symbol.for(key)`, который ищет символ по имени, существует обратный метод: `Symbol.keyFor(sym)`, который, наоборот, принимает глобальный символ и возвращает его имя.

К примеру:

```
// получаем символ по имени
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// получаем имя по символу
```

```
6  alert( Symbol.keyFor(sym) ); // name
7  alert( Symbol.keyFor(sym2) ); // id
```


Внутри метода `Symbol.keyFor` используется глобальный реестр символов для нахождения имени символа. Так что этот метод не будет работать для неглобальных символов. Если символ неглобальный, метод не сможет его найти и вернёт `undefined`.

Впрочем, для любых символов доступно свойство `description`.

Например:

```
1  let globalSymbol = Symbol.for("name");
2  let localSymbol = Symbol("name");
3
4  alert( Symbol.keyFor(globalSymbol) ); // name, глобальны
5  alert( Symbol.keyFor(localSymbol) ); // undefined для н
6
7  alert( localSymbol.description ); // name
```

Системные символы

Существует множество «системных» символов, использующихся внутри самого JavaScript, и мы можем использовать их, чтобы настраивать различные аспекты поведения объектов.

Эти символы перечислены в спецификации в таблице Well-known symbols:

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
- …и так далее.

В частности, `Symbol.toPrimitive` позволяет описать правила для объекта, согласно которым он будет преобразовываться к примитиву. Мы скоро увидим его применение.

С другими системными символами мы тоже скоро познакомимся, когда будем изучать соответствующие возможности языка.

Итого

Символ (symbol) – примитивный тип данных, использующийся для создания уникальных идентификаторов.

Символы создаются вызовом функции `Symbol()`, в которую можно передать описание (имя) символа.

Даже если символы имеют одно и то же имя, это – разные символы. Если мы хотим, чтобы одноимённые символы были равны, то следует использовать глобальный реестр: вызов `Symbol.for(key)` возвращает (или создаёт) глобальный символ с `key` в качестве имени. Многократные вызовы команды `Symbol.for` с одним и тем же аргументом возвращают один и тот же символ.

Символы имеют два основных варианта использования:

1. «Скрытые» свойства объектов. Если мы хотим добавить свойство в объект, который «принадлежит» другому скрипту или библиотеке, мы можем создать символ и использовать его в качестве ключа. Символьное свойство не появится в `for..in`, так что оно не будет нечаянно обработано вместе с другими. Также оно не будет модифицировано прямым обращением, так как другой скрипт не знает о нашем символе. Таким образом, свойство будет защищено от случайной перезаписи или использования.

   Так что, используя символьные свойства, мы можем спрятать что-то нужное нам, но что другие видеть не должны.

2. Существует множество системных символов, используемых внутри JavaScript, доступных как `Symbol.*`. Мы можем использовать их, чтобы

изменять встроенное поведение ряда объектов. Например, в дальнейших главах мы будем использовать `Symbol.iterator` для итераторов, `Symbol.toPrimitive` для настройки преобразования объектов в примитивы и так далее.

Технически символы скрыты не на 100%. Существует встроенный метод Object.getOwnPropertySymbols(obj) – с его помощью можно получить все свойства объекта с ключами-символами. Также существует метод Reflect.ownKeys(obj), который возвращает *все* ключи объекта, включая символьные. Так что они не совсем спрятаны. Но большинство библиотек, встроенных методов и синтаксических конструкций не используют эти методы.

Комментарии

перед тем как писать...