**Лабораторная работа №23**

**Тема работы:** Разработка программ с применением типа данных Symbol.

**Цель работы:** закрепить навыки по работе с типом данных Symbol.

**Теоретические сведения**

**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](https://learn.javascript.ru/symbol" \l "simvoly)**

«Символ» представляет собой уникальный идентификатор.

Создаются новые символы с помощью функции Symbol():

// Создаём новый символ - id

let id = Symbol();

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

// Создаём символ id с описанием (именем) "id"

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:

let id1 = Symbol("id");

let id2 = Symbol("id");

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:

let id = Symbol("id");

alert(id); // TypeError: Cannot convert a Symbol value to a string

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:

let id = Symbol("id");

alert(id.toString()); // Symbol(id), теперь работает

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

let id = Symbol("id");

alert(id.description); // id

**[«Скрытые» свойства](https://learn.javascript.ru/symbol" \l "skrytye-svoystva)**

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:

let user = {

name: "Вася"

};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // мы можем получить доступ к данным по ключу-символу

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](https://learn.javascript.ru/symbol" \l "simvoly-v-literalnom-obekte)**

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](https://learn.javascript.ru/symbol" \l "simvoly-ignoriruyutsya-tsiklom-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 (свойства с ключом-символом нет среди перечисленных)

// 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](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/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](https://learn.javascript.ru/symbol" \l "globalnye-simvoly)**

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 его в переменную

let id = Symbol.for("id"); // если символа не существует, он будет создан

// читаем его снова в другую переменную (возможно, из другого места кода)

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](https://learn.javascript.ru/symbol" \l "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

alert( Symbol.keyFor(sym) ); // name

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:

let globalSymbol = Symbol.for("name");

let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, глобальный символ

alert( Symbol.keyFor(localSymbol) ); // undefined для неглобального символа

alert( localSymbol.description ); // name

**[Системные символы](https://learn.javascript.ru/symbol" \l "sistemnye-simvoly)**

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](https://tc39.github.io/ecma262/#sec-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.

**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](https://learn.javascript.ru/type-conversions) 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](https://learn.javascript.ru/date)) 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.

**[Symbol.iterator](https://learn.javascript.ru/iterable" \l "symbol-iterator)**

Мы легко поймём принцип устройства перебираемых объектов, создав один из них.

Например, у нас есть объект. Это не массив, но он выглядит подходящим для for..of.

Например, объект range, который представляет собой диапазон чисел:

let range = {

from: 1,

to: 5

};

// Мы хотим, чтобы работал for..of:

// for(let num of range) ... num=1,2,3,4,5

Чтобы сделать range итерируемым (и позволить for..of работать с ним), нам нужно добавить в объект метод с именем Symbol.iterator (специальный встроенный Symbol, созданный как раз для этого).

1. Когда цикл for..of запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть итератор – объект с методом next.
2. Дальше for..of работает только с этим возвращённым объектом.
3. Когда for..of хочет получить следующее значение, он вызывает метод next() этого объекта.
4. Результат вызова next() должен иметь вид {done: Boolean, value: any}, где done=true означает, что итерация закончена, в противном случае value содержит очередное значение.

Вот полная реализация range с пояснениями:

let range = {

from: 1,

to: 5

};

// 1. вызов for..of сначала вызывает эту функцию

range[Symbol.iterator] = function() {

// ...она возвращает объект итератора:

// 2. Далее, for..of работает только с этим итератором, запрашивая у него новые значения

return {

current: this.from,

last: this.to,

// 3. next() вызывается на каждой итерации цикла for..of

next() {

// 4. он должен вернуть значение в виде объекта {done:.., value :...}

if (this.current <= this.last) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

};

// теперь работает!

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Обратите внимание на ключевую особенность итераторов: разделение ответственности.

* У самого range нет метода next().
* Вместо этого другой объект, так называемый «итератор», создаётся вызовом range[Symbol.iterator](), и именно его next() генерирует значения.

Таким образом, итератор отделён от самого итерируемого объекта.

Технически мы можем объединить их и использовать сам range как итератор, чтобы упростить код.

Например, вот так:

let range = {

from: 1,

to: 5,

[Symbol.iterator]() {

this.current = this.from;

return this;

},

next() {

if (this.current <= this.to) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Теперь range[Symbol.iterator]() возвращает сам объект range: у него есть необходимый метод next(), и он запоминает текущее состояние итерации в this.current. Короче? Да. И иногда такой способ тоже хорош.

Недостаток такого подхода в том, что теперь мы не можем использовать этот объект в двух параллельных циклах for..of: у них будет общее текущее состояние итерации, потому что теперь существует лишь один итератор – сам объект. Но необходимость в двух циклах for..of, выполняемых одновременно, возникает редко, даже при наличии асинхронных операций.

**Бесконечные итераторы**

Можно сделать бесконечный итератор. Например, range будет бесконечным при range.to = Infinity. Или мы можем создать итерируемый объект, который генерирует бесконечную последовательность псевдослучайных чисел. Это бывает полезно.

Метод next не имеет ограничений, он может возвращать всё новые и новые значения, это нормально.

Конечно же, цикл for..of с таким итерируемым объектом будет бесконечным. Но мы всегда можем прервать его, используя break.

**[Строка – перебираемый объект](https://learn.javascript.ru/iterable" \l "stroka-perebiraemyy-obekt)**

Среди встроенных перебираемых объектов наиболее широко используются массивы и строки.

Для строки for..of перебирает символы:

for (let char of "test") {

// срабатывает 4 раза: по одному для каждого символа

alert( char ); // t, затем e, затем s, затем t

}

И он работает корректно даже с суррогатными парами!

let str = '𝒳😂';

for (let char of str) {

alert( char ); // 𝒳, а затем 😂

}

**[Явный вызов итератора](https://learn.javascript.ru/iterable" \l "yavnyy-vyzov-iteratora)**

Чтобы понять устройство итераторов чуть глубже, давайте посмотрим, как их использовать явно.

Мы будем перебирать строку точно так же, как цикл for..of, но вручную, прямыми вызовами. Нижеприведённый код получает строковый итератор и берёт из него значения:

let str = "Hello";

// делает то же самое, что и

// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {

let result = iterator.next();

if (result.done) break;

alert(result.value); // выводит символы один за другим

}

Такое редко бывает необходимо, но это даёт нам больше контроля над процессом, чем for..of. Например, мы можем разбить процесс итерации на части: перебрать немного элементов, затем остановиться, сделать что-то ещё и потом продолжить.

**[Итерируемые объекты и псевдомассивы](https://learn.javascript.ru/iterable" \l "array-like)**

Есть два официальных термина, которые очень похожи, но в то же время сильно различаются. Поэтому убедитесь, что вы как следует поняли их, чтобы избежать путаницы.

* Итерируемые объекты – это объекты, которые реализуют метод Symbol.iterator, как было описано выше.
* Псевдомассивы – это объекты, у которых есть индексы и свойство length, то есть, они выглядят как массивы.

При использовании JavaScript в браузере или других окружениях мы можем встретить объекты, которые являются итерируемыми или псевдомассивами, или и тем, и другим.

Например, строки итерируемы (для них работает for..of) и являются псевдомассивами (они индексированы и есть length).

Но итерируемый объект может не быть псевдомассивом. И наоборот: псевдомассив может не быть итерируемым.

Например, объект range из примера выше – итерируемый, но не является псевдомассивом, потому что у него нет индексированных свойств и length.

А вот объект, который является псевдомассивом, но его нельзя итерировать:

let arrayLike = { // есть индексы и свойство length => псевдомассив

0: "Hello",

1: "World",

length: 2

};

// Ошибка (отсутствует Symbol.iterator)

for (let item of arrayLike) {}

Что у них общего? И итерируемые объекты, и псевдомассивы – это обычно не массивы, у них нет методов push, pop и т.д. Довольно неудобно, если у нас есть такой объект и мы хотим работать с ним как с массивом. Например, мы хотели бы работать с range, используя методы массивов. Как этого достичь?

**Порядок выполнения работы**

Задача 1

Для некоторого объекта добавить ключ, которы будет являтся символом.

Задача 2

С помощью конструктора создать объект, описывающий студента вашей группы, в конструкторе должно быть минимум 7 свойств и методов. Написать функцию, которая будет преобразовывать объект в итератор, чтобы он мог перебираться с помощью цикла for..of.