JS6 и прочее

Введение

Стандарт ES-2015 был принят в июне 2015. Пока что большинство браузеров реализуют его частично.

V8 большинство возможностей ES-2015 поддерживаются только при включенном use strict.

Необходимо использовать Babel.JS – это транспайлер, переписывающий код на ES-2015 в код на предыдущем стандарте ES5.
Он состоит из двух частей:

1. Собственно транспайлер, который переписывает код.
2. Полифилл, который добавляет методы `Array.from`, `String.prototype.repeat` и другие.

```
<script src="https://bla-
bla.bla/browser.min.js"></script>
<script type="text/babel">
  let arr = ["hello", 2]; // let
  let [str, times] = arr; //
деструктуризация
  alert( str.repeat(times) ); //
hellohello, метод repeat
</script>
```

Переменные: let и const

`let`

У объявлений переменной через `let` есть три основных отличия от `var`:

1. Область видимости переменной `let` – блок `{...}`. Переменная, объявленная через var, видна везде в функции. Переменная, объявленная через `let`, видна только в рамках блока `{...}`, в котором объявлена. Это, в частности, влияет на объявления внутри `if`, `while` или `for`.

2. Переменная `let` видна только после объявления. Переменные `var` существуют и до объявления. Они равны `undefined`. Также, переменные `let` нельзя повторно объявлять. То есть, такой код выведет ошибку.

3. При использовании в цикле, для каждой итерации создаётся своя переменная. Каждому повторению цикла соответствует своя независимая переменная `let`. Если внутри цикла есть вложенные объявления функций, то в замыкании каждой будет та переменная, которая была при соответствующей итерации. Переменная `var` – одна на все итерации цикла и видна даже после цикла.

const

Объявление `const` задаёт константу, то есть переменную, которую нельзя менять. В остальном объявление const полностью аналогично `let`.

Если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё. То же самое верно, если константе присвоен массив или другое объектное значение.

Это объявление создаёт константу, чья область действия может быть как глобальной, так и локальной внутри блока, в котором она объявлена.

Глобальные константы не становятся свойствами объекта window, в отличие от `var`-переменных.

Инициализация константы обязательна; необходимо указать значение одновременно с объявлением .

Имена констант не могут совпадать с именами функций или переменных той же области видимости.

Деструктуризация

Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

Массив

Пример деструктуризации массива:

```
'use                                    strict';
let [first, second] = ["Кафедра", "ИС"];
```

Первое значение массива пойдёт в переменную `first`, второе – в `last`, а последующие (если есть) – будут отброшены.

Ненужные элементы массива также можно отбросить, поставив лишнюю запятую:

```
let [, , title] = "Слава выпускникам кафедры ИС".split(" ");
```

В коде выше первый и второй элементы массива никуда не записались, они были отброшены. Также и все элементы после третьего.

Значения по умолчанию

Если значений в массиве меньше, чем переменных – ошибки не будет, просто присваивается `undefined`:

```
let [first, second] = [];
```

В таких случаях задают значение по умолчанию.

После переменной исп. символ = со значением:

```
let [first="Гость", second="Анонимный"] = [];
```

В качестве значений по умолчанию можно использовать не только примитивы, но и выражения, даже включающие в себя вызовы функций.

Деструктуризация объекта

Деструктуризацию можно использовать и с объектами. При этом указывается, какие свойства в какие переменные должны «идти».

Базовый синтаксис:

```
let {var1, var2} = {var1: …, var2: …};
```

Объект справа – уже существующий, который разбивают на переменные. А слева – список переменных, в которые нужно соответствующие свойства записать.

```
'use strict';
let options = {
  title: "Меню",
  width: 100,
  height: 200
};
let {title, width, height} = options;
```

Если необходимо присвоить свойство объекта в переменную с другим именем, то можно указать соответствие через двоеточие, вот так: `let {width: w, height: h, title} = options;`

Если каких-то свойств в объекте нет, можно указать значение по умолчанию через знак равенства =.

```
let {width=100, height=200, title} = options;
```

Можно и сочетать одновременно двоеточие и равенство.

В примерах выше переменные объявлялись прямо перед присваиванием: `let {…} = {…}`. Конечно, можно и без `let`, использовать уже существующие переменные.

В JavaScript, если в основном потоке кода (не внутри другого выражения) встречается `{...}`, то это воспринимается как блок. Можно использовать такой блок для ограничения видимости переменных.

Чтобы избежать интерпретации `{a, b}` как блока, нужно обернуть всё присваивание в скобки:

```
let a, b;
({a, b} = {a:5, b:6}); // внутри выражения это уже не блок
```

Оператор «spread/rest»

Новый оператор ... называется spread (распростанение, расширение) или *rest* (остаток) в зависимости от того, где и как он используется. *Spread* используется для разделения коллекций на отдельные элементы, а *rest*, наоборот, для соединения отдельных значений в массив.

Spread в литералах массива

Клонирование свойств массивов:

```
var arr = ['will', 'love'];
var data = ['You', ...arr, 'spread',
'operator'];
console.log(data); // ['You', 'will',
'love', 'spread', 'operator']
```

Конкатенация массивов

```
var arr1 = [0, 1, 2]; var arr2 = [3, 4,
5]; arr1 = [...arr1, ...arr2];
```

Копирование всего массива целиком

```
var arr = [1, 2, 3, 4, 5];
var data = [...arr];
console.log(data); // [1, 2, 3, 4, 5]
```

Важно понимать, что при подобном использовании оператора ... происходит именно копирование всех свойств, а не ссылки на массив.

```
var arr = [1, 2, 3, 4, 5];
var data = [...arr];
var copy = arr;
arr === data; // false
```
– ссылки отличаются – два разных массива

```
arr === copy; // true
```
– две переменные ссылаются на один массив

Spread syntax переходит лишь на один уровень глубже при копировании массива. Он может не подходить для копир-ния многоразмерных массивов.

Оператор ...rest

Для того чтобы получить все значения массива, но не уверены в их числе – можно добавить параметр, который получит «всё остальное», при помощи оператора "...":

```
let [first, second, ...rest] = "Пролетарии всех стран объединяйтесь".split(" ");
```

Значением `rest` будет массив из оставшихся элементов массива. Имя должно стоять только последним элементом в списке слева.

Литерал объекта (новое в ECMAScript 2018):

```
let objClone = { ...obj };
```

Функции

Оператор spread вместо arguments

Чтобы получить массив аргументов, можно использовать оператор …, например:

```
function show(first, second, ...rest) { }
```

В rest попадёт массив всех аргументов, начиная с третьего.

`rest` – настоящий массив, с методами `map`, `forEach` и другими, в отличие от arguments.

Этот оператор можно использовать и при вызове функции, для передачи массива параметров как списка.

Обычно используют Function.prototype.apply в случаях, когда хотят использовать элементы массива в качестве аргументов функции.

```
function sum(x, y, z) {
  return x + y + z;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers));  //  expected
output: 6
console.log(sum.apply(null,  numbers));  //
expected output: 6
```

Параметры по умолчанию

Можно указывать параметры по умолчанию через равенство `=`.

Параметр по умолчанию используется при отсутствующем аргументе или равном `undefined`.

При передаче любого значения, кроме `undefined`, включая пустую строку (`""`), ноль (`0`) или `null`, параметр считается переданным, и значение по умолчанию не используется.

Параметры по умолчанию могут быть выражениями.

Деструктуризация в параметрах

Если функция получает объект, то она может его тут же разбить в переменные:

```
let options = {
  title: "JS",
  width: 100,
  height: 200
};
function show({title, width, height}) {}
show(options);
```

Можно использовать и более сложную деструктуризацию, с соответствиями и значениями по умолчанию

```
function show({title="JS", width:w=100, height:h=200}){ }
```

Если хочется, чтобы функция могла быть вызвана вообще без аргументов – нужно добавить ей параметр по умолчанию – уже не внутрь деструктуризации, а в самом списке аргументов:

```
function show({title="JS", width:w=100, height:h=200} = {}) {  }
```

Имя «name»

В свойстве `name` у функции находится её имя.

```
function f() {} // f.name == "f"
let g = function g() {}; // g.name == "g"
alert(f.name + ' ' + g.name) // f g
```

В примере выше показаны Function Declaration и Named Function Expression.

В синтаксисе выше довольно очевидно, что у этих функций есть имя `name`, оно указано в объявлении.

JavaScript идёт дальше, он старается даже анонимным функциям дать разумные имена. Например, при создании анонимной функции с одновременной записью в переменную или свойство – её имя равно названию переменной (или свойства).

```
// свойство g.name = "g"
let g = function() {};
let user = {
sayHi: function() {}
```

Функции в блоке

Объявление функции Function Declaration, сделанное в блоке, видно только в этом блоке.

СТРЕЛОЧНЫЕ ФУНКЦИИ

Появился новый синтаксис для задания функций через «стрелку» =>.

Его простейший вариант выглядит так:

```
'use                                strict';
let     inc     =     x     =>     x+1;
alert( inc(1) ); // 2
```

Все стрелочные функции создаются с помощью function expression.

Перед использованием стрелочной функции её всегда необходимо создать заранее.

Как видно, "`x => x+1`" – это уже готовая функция. Слева от `=>` находится аргумент, а справа – выражение, которое нужно вернуть.

Если аргументов несколько, то нужно обернуть их в скобки, вот так:

```
let sum = (a,b) => a + b;
```

Если нужно задать функцию без аргументов, то также используются скобки, в этом случае – пустые:

```
let getTime = () => new Date().getHours()
+ ':' + new Date().getMinutes();
```

Когда тело функции достаточно большое, то можно его обернуть в фигурные скобки `{…}`.

Заметим, что как только тело функции оборачивается в `{…}`, то её результат уже не возвращается автоматически.

Такая функция должна делать явный `return`, как в примере выше, если необходимо что-либо возвратить.

Функции-стрелки не имеют своего this

Внутри их – тот же `this`, что и снаружи.

```
let group = {
  title: "Наш курс",
  students: ["Вася", "Петя", "Даша"],
  showList: function() {
    this.students.forEach(
      student => alert(this.title + ': ' +
                       student)
    ) }}
```

```
group.showList();
//          Наш          курс:          Вася
//          Наш          курс:          Петя
// Наш курс: Даша
```

Если бы в `forEach` вместо функции-стрелки была обычная функция, то была бы ошибка:

```
let group = {
  title: "Наш курс",
  students: ["Вася", "Петя", "Даша"],
  showList: function() {
  this.students.forEach(function(student)
{     alert(this.title + ': ' + student);
//  будет ошибка
                    })
                  }}
group.showList();
```

При запуске будет "попытка прочитать свойство `title` у `undefined`", так как `.forEach(f)` при запуске `f` не ставит `this`.

То есть, `this` внутри `forEach` будет `undefined`.

Чтобы выполнить какой-либо метод из объекта до появления стрелочных функций, необходимо было воспользоваться одним из методов замены ключевого слова `this`: записать его в переменную `self`, или воспользоваться методом функций `bind`

Функции стрелки нельзя запускать с new

Отсутствие у функции-стрелки "своего `this`" влечёт за собой естественное ограничение: такие функции нельзя использовать в качестве конструктора, то есть нельзя вызывать через `new`.

Функции-стрелки не имеют своего arguments

В качестве `arguments` используются аргументы внешней «обычной» функции.

```
function f() {
 let showArg = () => alert(arguments[0]);
  showArg();
}
f(1); // 1
```

Вызов `showArg()` выведет 1, получив его из аргументов функции `f`. Функция-стрелка здесь вызвана без параметров, но это не важно: `arguments` всегда берутся из внешней «обычной» функции.

Строки

Интерполяция

Интерполяция в JavaScript работает следующим образом. В строке создается конструкция `${...}`, внутри которой вы можете поместить любую переменную или выражение:

```
var age = 35;
console.log(`I am ${age} years old`); //
I am 35 years old
```

Строки, созданные с помощью обычных кавычек (' и ") не поддерживают интерполяцию. Для поддержки интерполяции следует использовать обратную кавычку ` (клавиша ё на клавиатуре).

Интерполяция выражений

С помощью интерполяции в строку можно поместить результат выполнения любого выражения, например, вызов функции:

```
const up = (str) => str.toUpperCase();
let str = `this is ${ up('sting') } in uppercase`;
console.log(str); // this is STRING in uppercase
```

Вложенная интерполяция

Скорее всего, будут возникать ситуации, когда одного уровня интерполяции будет недостаточно. В подобных случаях удобно пользоваться вложенностью (интерполяция внутри интерполяции). Следует помнить, что весь код, находящийся внутри `${...}` интерпретируется, как отдельное выражение, то есть может содержать обратные кавычки, которые не будут восприняты, как конец строки:

```
const up = (str) => str.toUpperCase();
let user = 'Drozin';
let str = `Mr ${up(`${user}s`)} is
great`;
console.log(str); // these DROZIN is
great
```

Функции шаблонизации

Можно использовать свою функцию шаблонизации для строк.

Название этой функции ставится перед первой обратной кавычкой:

```
let str = func`моя строка`;
```

Эта функция будет автоматически вызвана и получит в качестве аргументов строку, разбитую по вхождениям параметров `${...}` и сами эти параметры.

Полезные методы

Добавлен ряд полезных методов общего назначения:

- str.includes(s) – проверяет, включает ли одна строка в себя другую, возвращает true/false.
- str.endsWith(s) – возвращает true, если строка str заканчивается подстрокой s.
- str.startsWith(s) – возвращает true, если строка str начинается со строки s.
- str.repeat(times) – повторяет строку str times раз.

Улучшена поддержка юникода

Тип данных Symbol

В JavaScript появился новый примитивный тип данных: symbol.

В отличие от остальных примитивных типов он не может быть представлен в форме литерала.

```
var sym = Symbol( "необязательное описание" );

typeof sym;                    // "symbol"
```

Следует обратить внимание на три момента.

• Использовать оператор new с функцией Symbol(..) нельзя. Это не конструктор, и вы создаете не объект.

• Передавать функции Symbol(..) параметр необязательно. Передаваемый параметр представляет собой строку с описанием назначения символа.

• Оператор typeof выводит новое значение ("symbol"), которое является основным способом идентификации символа.

Описание, если оно дается, используется исключительно для перевода представления символа в строку.

Аналогично тому, как примитивные строковые значения не являются экземплярами класса String, символы — не экземпляры класса Symbol.

Внутреннее значение самого символа — которое еще называют его именем — скрыто из кода, и получить его нельзя. Его можно представить как автоматически генерируемое, уникальное (внутри вашего приложения) строковое значение.

Главным образом он применяется для создания напоминающего строку значения, которое не конфликтует ни с каким другим. Его можно использовать, к примеру, как константу, представляющую имя события:

```
const EVT_LOGIN = Symbol( "event.login" );
```

После этого `EVT_LOGIN` подставляется вместо строкового литерала "event.login":

```
evthub.listen( EVT_LOGIN, function(data){
    // ..
} );
```

Реестр символов

Использование символов имеет свои особенности. Скажем, в последних фрагментах кода переменные `EVT_LOGIN` и `INSTANCE` пришлось сохранить во внешней области видимости (и даже в глобальной), потому что они должны были находиться в открытом доступе для всех частей кода, которым они могли потребоваться.

Чтобы дать коду доступ к символам, их значения следует создать в глобальном реестре (global symbol registry).

Метод `Symbol.for(..)` проверяет в глобальном реестре, хранится ли там символ с заданным описанием, и, обнаружив, возвращает его, а в противном случае — создает. Другими словами, глобальный реестр интерпретирует значения символов по тексту описания.

Одновременно это означает, что любая часть приложения может затребовать символ из реестра с помощью метода `Symbol.for(..)` при условии, что имя описания совпадает.

Восстановить текст описания (ключ) зарегистрированного символа позволяет метод `Symbol.keyFor(..)`

`Symbol.keyFor` работает только для глобальных символов, для остальных будет возвращено undefined

Символы как свойства объектов

Символ, который используется как свойство/ключ объекта, сохраняется особым образом и не отображается при обычном перечислении свойств:

```
var    o = {
    foo: 42,
    [ Symbol( "bar" ) ]: "hello world",
    baz: true
};

Object.getOwnPropertyNames(   o   );          //   [
"foo","baz" ]
```

Свойство-символ недоступно, если обратиться к его названию.

Вот способ, позволяющий получить символьное свойство объекта:

```
Object.getOwnPropertySymbols( o ); // [ Symbol(bar) ]
```

Очевидно, что символьное свойство на самом деле не является скрытым или недоступным, так как его всегда можно увидеть с помощью метода `Object.getOwnPropertySymbols(..)`.

Встроенные символы

В ES6 существует ряд предопределенных, встроенных символов, обеспечивающих различные поведения значениям объектов JavaScript.

Объекты и прототипы

Короткое свойство

Пусть переменные name и isAdmin, и необходимо использовать в объекте.

При объявлении объекта в этом случае достаточно указать только имя свойства, а значение будет взято из переменной с аналогичным именем.

```
let name = "Вася";
let isAdmin = true;
let user = {
  name,
  isAdmin
};
```

Вычисляемые свойства

В качестве имени свойства можно использовать выражение, например:

```
'use strict';
let propName = "firstName";
let user = {
  [propName]: "Вася"
};
alert( user.firstName ); // Вася
```

Геттер-сеттер для прототипа

В ES5 для прототипа был метод-геттер:

- `Object.getPrototypeOf(obj)`

В ES-2015 также добавился сеттер:

- `Object.setPrototypeOf(obj, newProto)`

Функция `Object.setPrototypeOf`, принимает два объекта. Первому переданному объекту будет присвоен второй в качестве прототипа:

```
const proto = { /* properties and methods */ };
const obj = { /* properties and methods */ };
Object.setPrototypeOf(obj, proto);
```

Свойство __proto__, которое даёт прямой доступ к прототипу добавлено в стандарт.

Object.assign

Функция `Object.assign(target, src1, src2...)` получает список объектов и копирует в первый `target` свойства из остальных.
При этом последующие свойства перезаписывают предыдущие.

```
'use strict';
let user = { name: "Вася" };
let visitor = { isAdmin: false, visits: true };
let admin = { isAdmin: true };
Object.assign(user, visitor, admin);
// user <- visitor <- admin
alert( JSON.stringify(user) ); // name: Вася,
visits: true, isAdmin: true
```

Его также можно использовать для 1-уровневого клонирования объекта:

```
'use strict';
let user = { name: "Вася", isAdmin: false };
// clone = пустой объект + все свойства user
let clone = Object.assign({}, user);
```

Object.is(value1, value2)

Новая функция для проверки равенства значений. Возвращает `true`, если значения `value1` и `value2` равны, иначе `false`. Она похожа на обычное строгое равенство `===`, но есть отличия:

```
// Сравнение +0 и -0
alert( Object.is(+0, -0)); // false
alert( +0 === -0 );        // true
// Сравнение с NaN
alert( Object.is(NaN, NaN) ); // true
alert( NaN === NaN );         // false
```

Отличия эти в большинстве ситуаций некритичны, так что не похоже, чтобы эта функция вытеснила обычную проверку `===`.
Что интересно – этот алгоритм сравнения, который называется SameValue, применяется во внутренних реализациях различных методов современного стандарта.

Методы объекта

Добавлены «методы объекта», которые, по сути, являются свойствами-функциями, привязанными к объекту.

Их особенности:

1. Более короткий синтаксис объявления.
2. Наличие в методах специального внутреннего свойства [[HomeObject]] («домашний объект»), ссылающегося на объект, которому метод принадлежит.

Для объявления метода вместо записи "`prop: function() {…}`" нужно написать просто "`prop() { … }`".

```
'use strict';
let name = "Вася";
let user = {
  name,
  saySome() {
    alert(this.name);
  }
};
user.saySome(); // Вася
```

Также методами станут объявления геттеров `get prop()` и сеттеров `set prop()`:

```
'use strict';
let name = "Вася", surname="Петров";
let user = {
  name,
  surname,
  get fullName() {
    return `${name} ${surname}`;
  }
};
alert( user.fullName ); // Вася Петров
```

Можно задать и метод с вычисляемым названием:

```
'use strict';
let methodName = "getFirstName";
let user = {
  // в квадратных скобках может быть любое
выражение, которое должно вернуть название метода
  [methodName]() {  // вместо [methodName]:
function() {
    return "Вася";
  }
};
alert( user.getFirstName() ); // Вася
```

super

В ES-2015 появилось новое ключевое слово `super`. Оно предназначено только для использования в методах объекта.

Вызов `super.parentProperty` позволяет из метода объекта получить свойство его прототипа.

Например, в коде ниже `rabbit` наследует от `animal`.

Вызов `super.walk()` из метода объекта `rabbit` обращается к `animal.walk()`:

```
let animal = {
  walk() {       alert("I'm walking");     }
};
let rabbit = {
  __proto__: animal,
  walk() {
    alert(super.walk); // walk() { … }
    super.walk(); // I'm walking
  }
};
rabbit.walk();
```

Как правило, это используется в классах, но важно понимать, что «классы» здесь на самом деле ни при чём.

Свойство super работает через прототип, на уровне методов объекта.

При обращении через super используется [[HomeObject]] текущего метода, и от него берётся __proto__.

Поэтому super работает только внутри методов.

В частности, если переписать этот код, оформив rabbit.walk как обычное свойство-функцию, то будет ошибка.

Исключением из этого правила являются функции-стрелки. В них используется `super` внешней функции.

Свойство [[HomeObject]] – не изменяемое

При создании метода – он привязан к своему объекту навсегда. Технически можно даже скопировать его и запустить отдельно, и `super` продолжит работать:

```
let walk = rabbit.walk; // скопируем метод в переменную
walk(); // вызовет animal.walk()
// I'm walking
```

Правила для `this` в методах – те же, что и для обычных функций. В примере выше при вызове `walk()` без объекта `this` будет `undefined`.

Классы

Новая конструкция class – удобный «синтаксический сахар» для задания конструктора вместе с прототипом.

Class

Синтаксис для классов выглядит так:

```
class Название [extends Родитель]  {
  constructor
  методы
}
```

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    alert(this.name);
  }
}
let user = new User("Вася");
user.sayHi(); // Вася
```

Функция `constructor` запускается при создании `new User`, остальные методы записываются в `User.prototype`.

Это объявление примерно аналогично такому:

```
function User(name) {
  this.name = name;
}

User.prototype.sayHi = function() {
  alert(this.name);
};
```

При объявлении через `class` есть и ряд отличий:

- `User` нельзя вызывать без `new`, будет ошибка.
- Объявление класса с точки зрения области видимости ведёт себя как `let`. В частности, оно видно только в текущем блоке и только в коде, который находится ниже объявления (Function Declaration видно и до объявления).

Методы, объявленные внутри `class`, также имеют ряд особенностей:

- Метод `sayHi` является именно методом, то есть имеет доступ к `super`.
- Все методы класса работают в строгом режиме `use strict`, даже если он не указан.
- Все методы класса не перечислимы. То есть в цикле `for..in` по объекту их не будет.

В отличие от объектных литералов, в теле класса отсутствуют запятые, отделяющие члены друг от друга. Более того, запятые там вообще недопустимы.

Class Expression

Также, как и Function Expression, классы можно задавать «инлайн», в любом выражении и внутри вызова функции. Это называется Class Expression:

```
'use strict';
let User = class {
  sayHi() { alert('Привет!'); }
};

new User().sayHi();
```

В примере выше у класса нет имени, что один-в-один соответствует синтаксису функций. Но имя можно дать. Тогда оно, как и в Named Function Expression, будет доступно только внутри класса:

```
let SiteGuest = class User {
  sayHi() { alert('Привет!'); }
};

new SiteGuest().sayHi(); // Привет
new User(); // ошибка
```

Геттеры, сеттеры и вычисляемые свойства

В классах, как и в обычных объектах, можно объявлять геттеры и сеттеры через get/set, а также использовать […] для свойств с вычисляемыми именами.

```
class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
```

```
// геттер
get fullName() {
  return `${this.firstName} ${this.lastName}`;
}

// сеттер
set fullName(newValue) {
  [this.firstName, this.lastName] = newValue.split(' ');
}
```

```
// вычисляемое название метода
  ["test".toUpperCase()]() {
    alert("PASSED!");
  }

};
```

```
let user = new User("Вася", "Пупков");
alert( user.fullName ); // Вася Пупков
user.fullName = "Иван Петров";
alert( user.fullName ); // Иван Петров
user.TEST(); // PASSED!
```

При чтении fullName будет вызван метод `get fullName()`, при присвоении – метод set fullName с новым значением.

class не позволяет задавать свойства-значения

В синтаксисе классов, как мы видели выше, можно создавать методы. Они будут записаны в прототип.

Однако, нет возможности задать в прототипе обычное значение (не функцию), такое как User.prototype.key = "value".

Конечно, никто не мешает после объявления класса в прототип дописать подобные свойства, однако предполагается, что в прототипе должны быть только методы.

Если свойство-значение, всё же, необходимо, то можно создать геттер, который будет нужное значение возвращать.

Статические свойства

Класс, как и функция, является объектом. Статические свойства класса User – это свойства непосредственно User, то есть доступные из него «через точку».
Для их объявления используется ключевое слово `static`.

```
class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  static createGuest() {
    return new User("Гость", "Сайта");
  }
};
```

```
let user = User.createGuest();

alert( user.firstName ); // Гость

alert( User.createGuest ); // createGuest ... (функция)
```

Как правило, они используются для операций, не требующих наличия объекта, например – для фабричных, как в примере выше, то есть как альтернативные варианты конструктора. Или же, можно добавить метод `User.compare`, который будет сравнивать двух пользователей для целей сортировки.

Также статическими удобно делать константы:

```
'use strict';
class Menu {
  static get elemClass() {
    return "menu"
  }
}
alert( Menu.elemClass ); // menu
```

Наследование

Синтаксис:

```
class Child extends Parent {
   ...
}
```

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  walk() {
    alert("I walk: " + this.name);
  }
}
```

```
class Rabbit extends Animal {
  walk() {
    super.walk();
    alert("...and jump!");
  }
}

new Rabbit("Вася").walk();
// I walk: Вася
// and jump!
```

Как видим, в `new Rabbit` доступны как свои методы, так и (через `super`) методы родителя.

Это потому, что при наследовании через `extends` формируется стандартная цепочка прототипов: методы `Rabbit` находятся в `Rabbit.prototype`, методы `Animal` – в `Animal.prototype`, и они связаны через `__proto__`

Конструктор `constructor` родителя наследуется автоматически. То есть, если в потомке не указан свой `constructor`, то используется родительский. В примере выше `Rabbit`, таким образом, использует constructor от `Animal`.

Если же у потомка свой `constructor`, то, чтобы в нём вызвать конструктор родителя – используется синтаксис `super()` с аргументами для родителя.

```
class Rabbit extends Animal {
  constructor() {
// вызвать конструктор Animal с аргументом "Кроль"
    super("Кроль");
// то же, что и Animal.call(this, "Кроль")
  }
}

new Rabbit().walk(); // I walk: Кроль
```

Для такого вызова есть небольшие ограничения:

- Вызвать конструктор родителя можно только изнутри конструктора потомка. В частности, `super()` нельзя вызвать из произвольного метода.
- В конструкторе потомка мы обязаны вызвать `super()` до обращения к `this`. До вызова `super` не существует `this`, так как по спецификации в этом случае именно super инициализирует `this`.

Ключевое слово class можно рассматривать таким способом. Это своего рода макрос, который автоматически заполняет прототип объекта. По вашему желанию он также связывает соотношение [[Prototype]], когда используется с ключевым словом extends.

Класс ES6 не является реальной программной единицей, это метаконцепция, которая охватывает существующие программные единицы, такие как функции и свойства, и связывает их друг с другом.

Promise

Введение

Promise (промис, англ. "обещание") - это объект, представляющий результат успешного или неудачного завершения асинхронной операции.

Объект **Promise** (обещание) используется для отложенных и асинхронных вычислений.

Promise может находиться в трёх состояниях:

- *ожидание (pending)*: начальное состояние, не выполнено и не отклонено.
- *выполнено (fulfilled)*: операция завершена успешно.
- *отклонено (rejected)*: операция завершена с ошибкой.

Другой термин, описывающий состояние *заданный (settled)*: обещание выполнено или отклонено, но не находится в состоянии ожидания.

Объект promise—это данные, возвращаемые асинхронной функцией. Это может быть resolve, если функция прошла успешно или reject, если функция вернула ошибку.

Промис—это объект, представляющий окончательное завершение или сбой асинхронной операции. По сути, промис—это возвращаемый объект, к которому прикрепляется колбэк, вместо его передачи в функцию.

JavaScript является однопоточным. Всё происходит в той последовательности, в которой написано, но асинхронные операции происходят в порядке их завершения.

```
console.log('1');

setTimeout(function(){ console.log('2'); },
3000);

console.log('3');

setTimeout(function(){ console.log('4'); },
1000);
```

Результатом будет 1 3 4 2.

Почему 4 встречается раньше чем 2.

Причина в том, что, несмотря на то, что строка с 2 описана раньше, она начала выполняться только после 3000 мс, поэтому 4 выводится до 2.

В типичном веб-приложении может выполняться множество асинхронных операций, таких как загрузка изображений, получение данных из *JSON*, обращение к *API* и других.

Создание промисов

Для того чтобы понять промисы, надо разобраться с двумя основными вещами. Первая — это создание промисов. Вторая — обработка результатов, возвращаемых промисами.

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

Конструктор принимает функцию, выполняющую некие действия, мы назвали её здесь `executor`. Эта функция принимает два параметра — `resolve` и `reject`, которые, в свою очередь, также являются функциями.

Промисы обычно используются для выполнения асинхронных операций или кода, который может заблокировать главный поток, например — такого, который работает с файлами, выполняет вызовы неких API, делает запросы к базам данных, занимается операциями ввода-вывода, и так далее.

Запуск подобных асинхронных операций выполняется в функции `executor`.

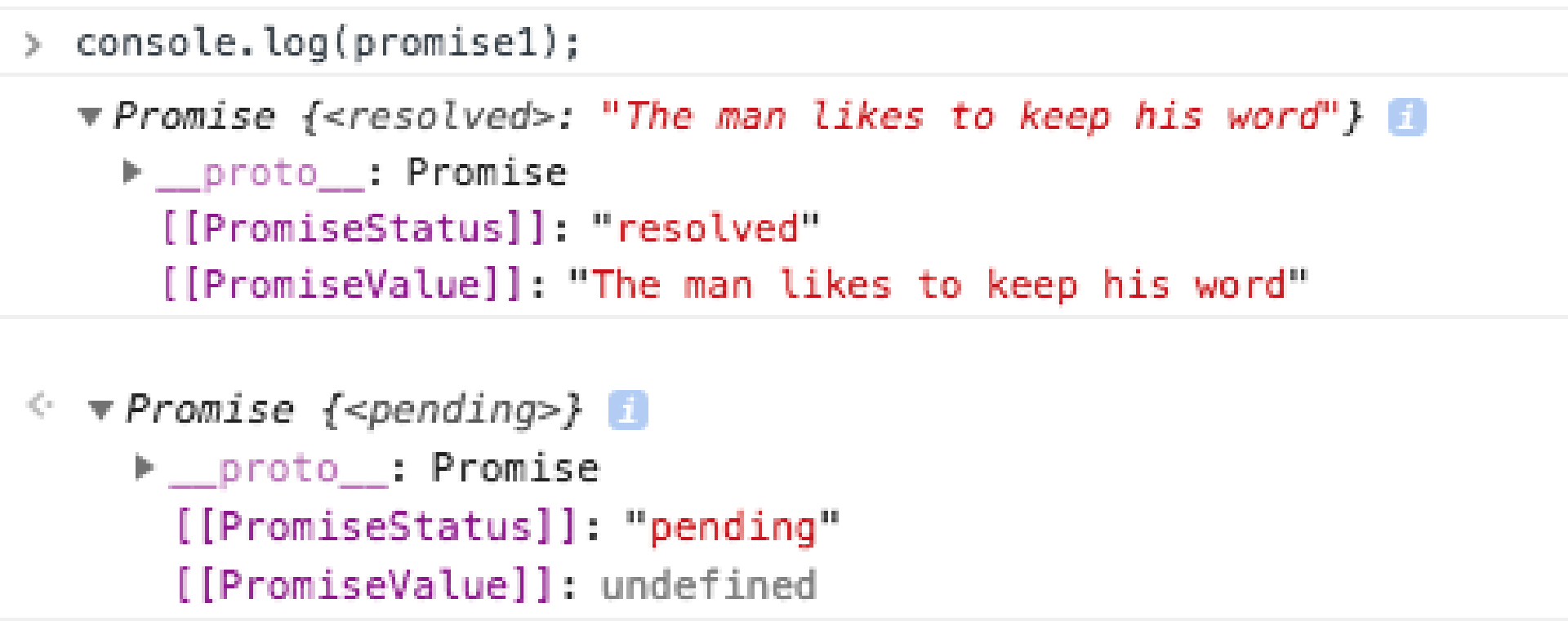
Если асинхронная операция будет завершена успешно, тогда результат, ожидаемый от промиса, будет возвращён путём вызова функции `resolve`.

Ситуация, в которой вызывается эта функция, определяется создателем промиса.

Аналогично, при возникновении ошибки, сведения о том, что случилось, возвращают, вызывая функцию `reject`.

```
var keepsHisWord;
keepsHisWord = true;
promise1 = new Promise(function(resolve, reject)
{
  if (keepsHisWord) {
    resolve("The man likes to keep his word");
  } else {
    reject("The man doesnt want to keep his
word");
  }
});
console.log(promise1);
```

Может возвращает не обычную строку, а объект.



У промиса есть состояние (PromiseStatus) и значение (PromiseValue)

В `PromiseStatus` могут появляться три разных значения: pending (ожидание), `resolved` (успешное разрешение) и `rejected` (отклонение).

Когда промис создаётся, в `PromiseStatus` будет значение `pending`, а в `PromiseValue` будет undefined.

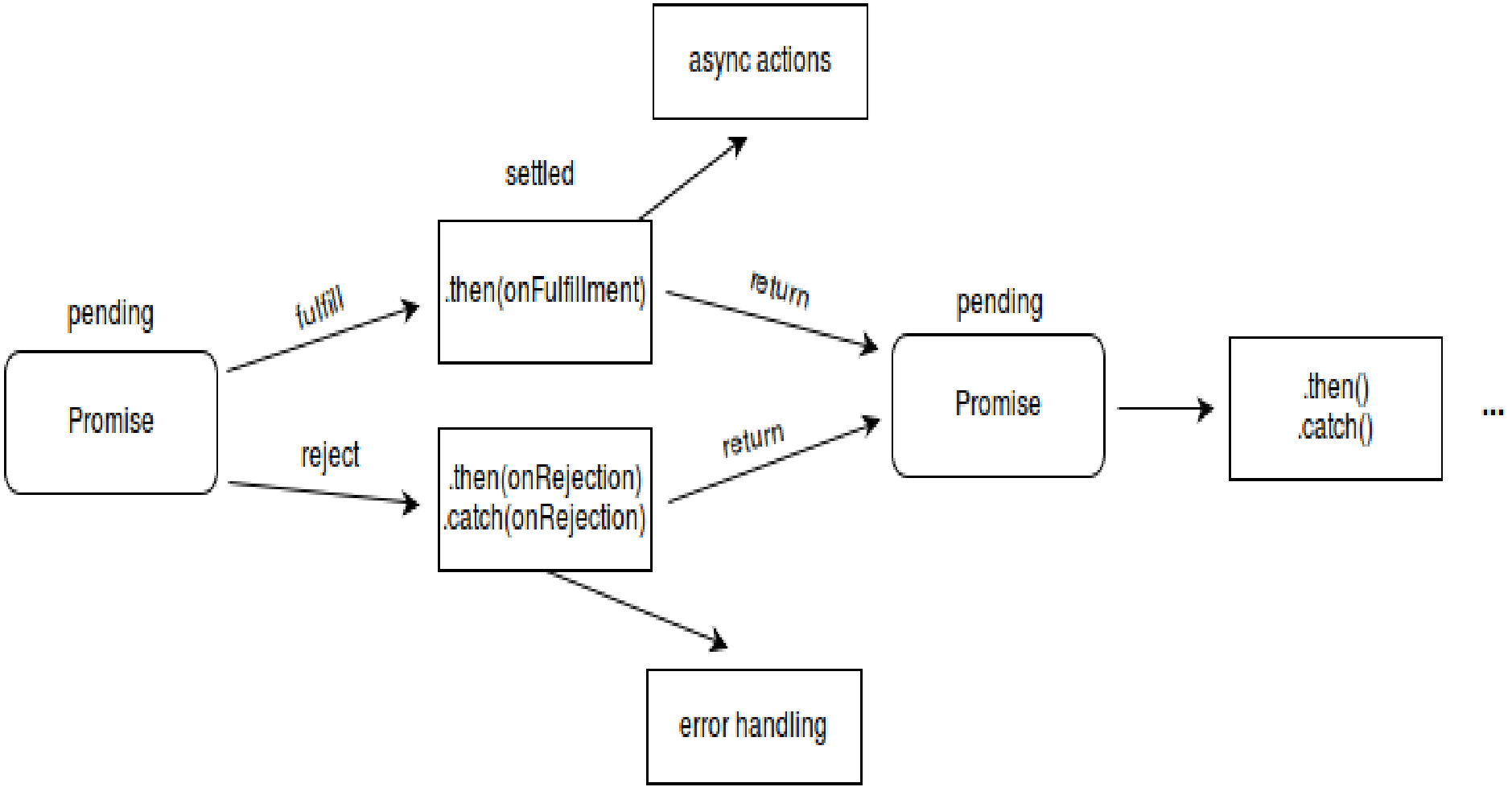
Эти значения будут сохраняться до разрешения или отклонения промиса. Когда промис находится в состоянии `resolved` или `rejected`, его называют заданным (`settled`) промисом.

Такой промис перешёл из состояния ожидания в состояние, в котором он имеет либо состояние `resolved`, либо состояние `rejected`.

В любом из этих случаев вызывается обработчик, прикрепленный к обещанию методом `then` .

Если в момент прикрепления обработчика обещание уже сдержано или нарушено, он все равно будет выполнен, т.е. между выполнением обещания и прикреплением обработчика нет «состояния гонки», как, например, в случае с событиями в DOM.

Методы `Promise.prototype.then` и `Promise.prototype.catch` сами возвращают обещания, их можно вызывать цепочкой, создавая *соединения.*


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
reject
.then(onRejection)
.catch(onRejection)
return
Promise
.then()
.catch()
...
error handling

Прототип Promise. Методы.

При переходе промиса в состояние `resolved` или `rejected` будет вызван, как минимум, один из следующих методов:

```
Promise.prototype.catch(onRejected)
```

```
Promise.prototype.then(onFulfilled, onRejected)
```

```
Promise.prototype.finally(onFinally)
```

`Promise.prototype.catch(onRejected)`

Добавляет функцию обратного вызова, для обработки отклонения обещания, которая возвращает новое обещание выполненное с переданным значением, если она вызвана, или оригинальное значение `resolve`, если обещание выполнено.

`Promise.prototype.then(onFulfilled, onRejected)`

Добавляет обработчик выполнения и отклонения обещания, и возвращает новое обещание выполненное со значением вызванного обработчика, или оригинальное значение, если обещание не было обработано (т.е. если соответствующий обработчик `onFulfilled` или `onRejected` не является функцией)

```
var myPromise = new Promise(function(resolve,
reject) {
  money = 20000;
  price = 60000;
  if (money  > price ) {
    resolve({
      brand: "iphone",
      model: "10X"
    });
  } else {
    reject("We donot have enough money.");
  }
});
```

```
myPromise.then(
  function(value) {
 console.log("I got if ", JSON.stringify(value));
  },
  function(reason) {
  console.log("I coudn't ", reason);
  }
);
```

```
myPromise.then(function(value) {
console.log("I got if ", JSON.stringify(value));
});

myPromise.catch(function(reason) {
  console.log("I coudn't ", reason);
});

myPromise.finally(function() {
  console.log(
    "I still happy"
  );
});
```

Чейнинг или цепочки из промисов

Одно из основных преимуществ промисов заключается в том, что мы можем строить цепочки или «чейны» из асинхронных операций.

Это значит, что мы можем указать, что выполнение последующих операций будет начато только после успешного выполнения предыдущих операций.

Такой метод называется чейнинг (от англ. chaining) промисов.

```
function get(url) {
  // Возвращает новый промис
  return new Promise(function(resolve, reject) {
… }
```

```
get('story.json').then(function(response) {
  return JSON.parse(response);
}).then(function(response) {
  console.log("Ух ты, JSON!", response);
});
```

```
get('story.json').then(function(response) {
  console.log("Успешное выполнение!", response);
}).catch(function(error) {
  console.log("Ошибка!", error);
});
```

`catch` это синтаксический сахар для `then(undefined, func)` — такой код немного читабельнее.

Разница почти незаметна, но, на практике, очень полезна. Если промису не передан аргумент `reject`, то выполнение передается следующему `then` с заданным `reject` (или `catch`, так как это одно и то же).

В коде `then(func1, func2)` будет вызвано только либо `func1`, либо `func2`. Однако, если промис будет вызван, как `then(func1).catch(func2)`, то возможен вариант, что выполнится и `func1`, и `func2`, если во время выполнения `func1` произошла ошибка.

Это происходит потому, что `then` и `catch` — два разных шага в цепочке выполнения.

Можно продолжить цепочку вызовов *после* ошибки, т. е. после `catch`, что полезно для выполнения новых действий даже после того, как действие вернет ошибку в цепочке вызовов.

```
new Promise((resolve, reject) => {
    console.log('Начало');
    resolve();
})
```

```
.then(() => {
    throw new Error('Где-то произошла ошибка');
    console.log('Выведи это');
})
.catch(() => {
    console.log('Выведи то');
})
.then(() => {
    console.log('Выведи это, несмотря ни на
что');
});
```

В результате выведется данный текст:

Начало

Выведи то

Выведи это, несмотря ни на что

Заметьте, что текст "*Выведи это*" не вывелся, потому что "*Где то произошла ошибка*" привела к отказу

Очередь асинхронных действий

Когда вы возвращаете что-то из функции обратного вызова у `then`, выглядит это как настоящая магия.

Если вы возвращаете значение, то следующий `then` будет вызван с этим значением.

Если же вы возвращаете промис или промисоподобный объект, то следующий `then` дождется выполнения этого промиса, и только потом будет вызван.

Статические методы объекта Promise

Существует четыре статических методы объекта `Promise`.

Первые два метода — `Promise.reject(reason)` и `Promise.resolve(value)`, которые позволяют создавать, соответственно, отклонённые и разрешённые промисы.

```
var promise4 = Promise.resolve(1);

var promise3 = Promise.reject("Not interested");
```

Следующие два метода, `Promise.all` и `Promise.race`, предназначены для работы с наборами промисов.

Если, для решения некоей задачи, надо обрабатывать несколько промисов, удобнее всего поместить эти промисы в массив, а затем выполнять с ними необходимые действия.

Эти два метода позволяют запустить асинхронные операции параллельно.

`Promise.all(iterable)`

Она возвращает обещание, которое успешно, если все аргументы успешны, и отклонен, когда любой из его аргументов отклонен.

В случае успеха результирующее обещание содержит массив результатов каждого обещания (в том же порядке, в котором их передали), а в случае неудачи — ошибку первого неуспешного обещания.

Случаи для использования Promise.all — загрузка нескольких HTTP-запросов одновременно, запуск нескольких процессов одновременно, или несколько одновременных запросов к базе данных.

Метод `Promise.race(iterable)` возвращает разрешённый или отклонённый промис со значением или причиной отклонения, после того как один из переданных промисов будет, соответственно, разрешён или отклонён.

Итераторы

Итератором (iterator) называется структурированный шаблон, предназначенный для последовательного извлечения информации из источника.

В ES6 для итераторов появился неявный стандартизованный интерфейс.

Многие встроенные структуры JavaScript предоставляют итератор, реализующий этот стандарт.

Можно сконструировать и собственные версии итераторов, если придерживаться стандарта для обеспечения максимальной совместимости.

Итераторы позволяют упорядоченно, последовательно извлекать данные.

Например, можно реализовать сервис, который при каждом запросе будет генерировать новый уникальный идентификатор; или формировать бесконечный ряд значений, прокручиваемых в фиксированном списке циклическим способом; или присоединить итератор к результатам запроса к базе данных, чтобы по одной извлекать новые строки.

Спецификация ES6 описывает интерфейс `Iterator` как отвечающий следующим требованиям:

`Iterator [обязательные параметры]`

`next() {метод}`: загружает следующий IteratorResult

Два дополнительных параметра для расширения некоторых итераторов:

`Iterator [необязательные параметры]`

`return() {метод}`: останавливает итератор и возвращает `IteratorResult`

`throw() {метод}`: сообщает об ошибке и возвращает `IteratorResult`

Интерфейс `IteratorResult` определен следующим образом:

`IteratorResult  value {свойство}:`

значение на текущей итерации или окончательное возвращаемое значение(не обязательно, если это 'undefined')

`done {свойство}:`

тип boolean, показывает состояние выполнения

Существует также интерфейс `Iterable`, описывающий объект, который должен уметь генерировать итераторы:

`Iterable` `@@iterator()` `{метод}`: генерирует `Iterator`

Интерфейс `IteratorResult` определяет, что возвращаемое значение любой из операций итератора будет представлять собой объект следующего вида:

```
{ value: .. , done: true / false }
```

Встроенные итераторы всегда возвращают значения в таком виде, но если нужно, ничто не мешает добавить и другие свойства.

Метод next()

Рассмотрим итерируемый массив и итератор, который он создает для работы со своими значениями:

```
var arr = [1,2,3];

var it = arr[Symbol.iterator]();

it.next();   // { value: 1, done: false }
it.next();   // { value: 2, done: false }
it.next();   // { value: 3, done: false }

it.next();   // { value: undefined, done: true }
```

Итератор `it` из последнего фрагмента кода не сообщает о завершении своей работы: когда вы получаете значение 3, `false` меняется на `true`.

Но чтобы узнать об этом, вам приходится еще раз вызывать метод `next()`, фактически выходя за границу значений массива.

Такое проектное решение, как правило, считается наилучшим вариантом.

Примитивные строковые значения также по умолчанию итерируемы:

```
var greeting = "hello world";

var it = greeting[Symbol.iterator]();

it.next();   // { value: "h", done: false }

it.next();   // { value: "e", done: false }

..
```

Метод итератора `next(..)` в зависимости от вашего желания принимает один или несколько аргументов.

Встроенные итераторы эту возможность в основном не используют, чего нельзя сказать об итераторе генератора.

Необязательные методы: return(..) и throw(..)

Необязательные методы интерфейса итератора — `return(..)` и `throw(..)` — у большинства встроенных итераторов не реализуются.

Тем не менее они, безусловно, значимы в контексте генераторов.

Метод `return(..)` определен как отправляющий итератору сигнал о том, что код извлечения информации завершил работу и больше не будет извлекать значения.

Этот сигнал уведомляет итератор, отвечающий на вызовы метода `next(..)`, что пришло время приступить к очистке, например к освобождению/закрытию сетевого соединения, базы данных или дескриптора файла.

Если у итератора есть метод `return(..)` и при этом возникает любое условие, которое можно интерпретировать как аномальное или раннее прекращение использования итератора, этот метод вызывается автоматически.

Впрочем, это легко сделать и вручную.

Метод `return(..)`, как и `next(..)`, возвращает объект IteratorResult.

В общем случае необязательное значение, отправленное в метод `return(..)`, будет возвращено как значение в объекте `IteratorResult`, хотя возможны и другие варианты развития событий.

Метод `throw(..)` сообщает итератору об исключении или ошибке, причем эту информацию тот может использовать иначе, нежели подаваемый методом `return(..)` сигнал завершения.

Ведь, в отличие от последнего, исключение или ошибка вовсе не подразумевает обязательного прекращения работы итератора.

Например, в случае с итераторами генератора метод `throw(..)`, по сути, вставляет в приостановленный контекст выполнения генератора порожденное им исключение, которое может быть перехвачено оператором `try..catch`.

Необработанное исключение в конечном счете прерывает работу итератора.

По общему соглашению, после вызова методов `return(..)` или `throw(..)` итератор не должен больше генерировать никаких результатов .

Цикл итератора

Появившийся в ES6 цикл `for..of` работает непосредственно с итерируемыми объектами.

Если итератор сам является итерируемым, его можно напрямую использовать с циклом `for..of`.

Чтобы сделать итератор таковым, его следует передать методу `Symbol.iterator`, который вернет нужный результат.

```
var    it = {

// делаем итератор 'it' итерируемым
[Symbol.iterator]() { return this; },

    next() { .. },     .. };

it[Symbol.iterator]() === it; // true

for (var v of it)

{

   console.log( v );

}
```

Напомню, что ранее мы уже говорилось о принятой для итераторов тенденции не возвращать вместе с предполагаемым окончательным значением `done: true`.

Я НАДЕЮСЬ ТЕПЕРЬ ПОНЯТНО ПОЧЕМУ???

Пользовательские итераторы

В дополнение к стандартным встроенным итераторам теперь можно создать собственный.

Для обеспечения взаимодействия с элементами ES6, использующими итераторы (например, с циклом `for..of` или оператором `...`), достаточно корректного интерфейса (или интерфейсов).

```
'use strict';
let range = {
  from: 1,
  to: 5
}
// сделаем объект range итерируемым
range[Symbol.iterator] = function() {

  let current = this.from;
  let last = this.to;

 // метод должен вернуть объект с методом next()
```

```
return {
    next() {
      if (current <= last) {
        return {
          done: false,
          value: current++
        };
      } else {
        return {
          done: true
        };
      }
    }  }  };
```

```
for (let num of range) {
  alert(num); // 1, затем 2, 3, 4, 5
}
```

Генераторы

В ES6 появилась новая, несколько необычная форма функции, названная генератором.

Такая функция может остановиться во время выполнения, а затем продолжить работу с прерванного места.

Более того, каждый цикл остановки и возобновления работы позволяет передавать сообщения в обе стороны.

Как генератор может вернуть значение, так и восстанавливающий его работу управляющий код может переслать туда что-нибудь.

Синтаксис

Функция-генератор объявляется следующим образом:

```
function *foo() {     //
  ..
}
```

Положение звездочки * функционально несущественно.

Кроме того, существует краткая форма генератора в объектных литералах

```
var     a = {
   *foo() { .. }
};
```

Выполнение генератора

Вызывается он как обычная функция: `foo();`

В него можно передавать аргументы:

```
function *foo(x,y) {     // .. }

foo( 5, 10 );
```

Основное отличие состоит в том, что запуск генератора, не приводит к исполнению его кода. Вместо этого создается итератор, контролирующий то, как генератор исполняет свой код.

```
function *foo() {     // .. }

var it = foo();

// чтобы начать/продолжить выполнение '*foo()',

// вызываем 'it.next(..)'
```

Ключевое слово yield

Внутри генераторов используется ключевое слово, сигнализирующее о прерывании работы: `yield`.

```
function *foo() {

var x = 10;

var y = 20;

yield;

var z = x + y;

}
```

В этом генераторе `*foo()` сначала запускаются операции из первых двух строк, а затем ключевое слово `yield` останавливает работу.

После ее возобновления запускается последняя строчка генератора `*foo()`.

Ключевое слово `yield` может появляться в генераторе произвольное количество раз (или вообще отсутствовать).

Ключевое слово `yield` не просто прерывает работу генератора.

В момент остановки оно посылает наружу значение.

```
function *foo() {
   while (true) {
      yield Math.random();
      }
}
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
let generator = generateSequence();
let one = generator.next();
alert(JSON.stringify(one)); // {value: 1, done: false}
let two = generator.next();
alert(JSON.stringify(two)); // {value: 2, done: false}
let three = generator.next();
alert(JSON.stringify(three)); // {value: 3, done: true
```

Генератор можно перебирать и через for..of

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, затем 2 !!!!!!
   }
```

Выражение `yield ..` позволяет не только передать значение (ключевое слово `yield`, не сопровождающееся ничем, означает yield `undefined`), но и получить — то есть заместить — конечное значение при возобновлении работы генератора.

```
function *foo() {
   var x = yield 10;
   console.log( x );
}
```

Этот генератор в момент остановки сначала получит значение `10`.

После возобновления работы — методом `it.next(..)` — любое восстановленное значение (если таковое вообще существует) полностью заместит выражение `yield 10`, то есть именно оно будет присвоено переменной `x`.

Вызов `let result = yield value` делает следующее:

- Возвращает `value` во внешний код, приостанавливая выполнение генератора.
- Внешний код может обработать значение, и затем вызвать `next` с аргументом: `generator.next(arg)`.
- Генератор продолжит выполнение, аргумент `next` будет возвращён как результат `yield` (и записан в `result`).

Если нужно вставить `yield ..` в место, где недопустимы такие выражения, как `a = 3`, его помещают в скобки `( )`.

Из-за низкого приоритета ключевого слова `yield` практически все выражения, следующие за `yield ..`, будут вычисляться раньше.

Более низким приоритетом обладают только оператор распределения `...` и запятая `,`, до них дело доходит после вычисления выражения с `yield`.

При комбинации ключевого слова `yield` с какими-либо операторами или с другими ключевыми словами `yield` разумно прибегнуть к группировке с помощью скобок `( .. )`, чтобы четко обозначить свои намерения

Выражение yield *

Аналогично тому, как символ `*` превращает объявление функции в объявление генератора, в случае использования `*` с ключевым словом `yield` образуется совершенно другой механизм, называемый `yield`-делегированием.

Грамматически выражение `yield *..` будет вести себя так же, как и рассмотренное в предыдущем разделе `yield ...`

Выражению `yield *..` требуется итерируемый объект; оно вызывает его итератор и передает ему управление собственным генератором. Рассмотрим пример:

```
function *foo() {

   yield *[1,2,3];

}
```

Применение генераторов

Создание набора значений

Этот вариант применения может быть как простым (например, случайные строки или возрастающие числа), так и дающим более структурированный доступ к данным (например, итерации по строкам, которые вернул запрос к базе данных).

В любом случае мы контролируем генератор с помощью итератора, что дает возможность реализовывать некую логическую схему при каждом вызове метода `next(..)`.

Обычные итераторы, работающие со структурами данных, просто извлекают оттуда значения, не добавляя никакой управляющей логической схемы.

Очередь задач для последовательного выполнения

Этот вариант применения часто представляет собой управление шагами какого-нибудь алгоритма, причем на каждом из них требуется извлекать данные из некого внешнего источника — немедленно или с асинхронной задержкой.

С точки зрения кода внутри генератора детали реализации синхронного или асинхронного извлечения данных в точке оператора `yield` совершенно непрозрачны.

Более того, они намеренно сделаны абстрактными, чтобы не прятать естественное последовательное выражение шагов за сложностями реализации.

Кроме того, абстрагирование дает возможность часто менять или перерабатывать реализацию, никак не затрагивая код внутри генератора.

Если смотреть на генераторы с точки зрения их применения, они перестают быть всего лишь красивым синтаксисом для конечного автомата с ручным управлением.

Генераторы — это мощный инструмент абстрагирования для систематизации и контроля за упорядоченным созданием и использованием данных.

Модули

Два основных ключевых слова, активирующие модули ES6, — `import` и `export`. Этот синтаксис имеет множество нюансов, потому рассмотрим его более подробно.

Ключевые слова `import` и `export` всегда должны появляться на верхнем уровне области видимости, в которой будут применяться .

Например, их нельзя вставлять в условный оператор `if`; они должны располагаться вне блоков и функций .

Экспорт членов API

Ключевое слово `export` или помещается перед объявлением, или используется в качестве своего рода оператора со специальным списком привязок, предназначенных для экспорта. Например:

```
export function foo() {     // .. }

export var awesome = 42;

var bar = [1,2,3];

export { bar };
```

Другой способ реализации этого же экспорта:

```
function foo() {    // .. }

var awesome = 42;

var bar = [1,2,3];

export { foo, awesome, bar };
```

Все это называется **именованным экспортом** (**named exports**), так как вы, по сути, экспортируются привязки имени переменных, функций и пр. Все, что не помечатся ключевым словом `export`, остается закрытым внутри области видимости модуля.

В процессе именованного экспорта можно переименовать член модуля (это явление известно как псевдоним):

```
function foo() { .. }

export { foo as bar };
```

Здесь при импорте доступным будет только член `bar`, а член `foo` останется скрытым внутри модуля.

Экспорт модулей отличается от обычного присваивания значений и ссылок, которое вы привыкли выполнять при помощи оператора =.

На самом деле при экспорте какого-либо элемента вы передаете привязку (своего рода указатель).

Если поменяется внутри модуля значение переменной, привязку к которой уже экспортировали, то, даже если ее импорт уже завершился, импортированная привязка даст текущее (обновленное) значение.

```
var awesome = 42;

export { awesome };

// позднее awesome = 100;
```

Когда модуль уже импортирован, не имеет значения, до или после этого было выполнено присваивание `awesome = 100`. По завершении импортированная привязка начнет давать нам значение `100`, а не `42`.

Хотя внутри определения модуля ничто не запрещает несколько раз использовать ключевое слово export, в ES6 более предпочтителен подход, при котором оно существует в единственном экземпляре.

Это так называемый **экспорт по умолчанию** (**default export**).

Экспорт по умолчанию задает конкретную экспортированную привязку как вариант по умолчанию, выбираемый при импорте модуля.

Возможна только одна привязка default на одно определение модуля.

Здесь есть нюанс, на который следует обратить внимание. Давайте сравним два фрагмента.

Первый:

```
function foo(..) {     // .. }

export default foo;
```

Второй:

```
function foo(..) {     // .. }

export { foo as default };
```

В первом случае вы экспортируете привязку в значение функционального выражения, а не в идентификатор `foo`.

Другими словами, `export default ..` преобразовывает выражение.

Если позднее присвоить функцию `foo` другому значению внутри модуля, процедура импорта все равно будет показывать изначально экспортированную функцию.

Второй случай. Здесь привязка экспорта по умолчанию на самом деле выполняется к идентификатору `foo`, а не к его значению, то есть вы получаете ранее описанное поведение (в частности, если позднее вы поменяете значение `foo`, обновится и значение на стороне импорта).

Первый фрагмент можно переписать еще и таким образом:

```
export default function foo(..) {    // .. }
```

Вернемся ко второму фрагменту:

```
function foo(..) {     // .. }

export { foo as default };
```

Здесь привязка экспорта по умолчанию на самом деле выполняется к идентификатору `foo`, а не к его значению, то есть вы получаете ранее описанное поведение (в частности, если позднее вы поменяете значение `foo`, обновится и значение на стороне импорта).

Будьте крайне осторожны с этим небольшим подводным камнем в синтаксисе экспорта по умолчанию, особенно если логическая схема требует обновлять экспортируемые значения.

Если вы не планируете этого делать, прекрасно подойдет вариант `export default ...`

В противном случае следует пользоваться вариантом `export { .. as default }`.

Импорт членов API

Для импорта модулей очевидным образом используется ключевое слово `import`.

Для импорта конкретных именованных членов API модуля в область видимости верхнего уровня применяется следующий синтаксис: `import { foo, bar, baz } from "foo";`

Строка "`foo`" называется спецификатором модуля (module specifier). Так как нам нужен статически анализируемый синтаксис, роль спецификатора играет строковый литерал; здесь нельзя использовать переменную, содержащую строковое значение

Перечисленные здесь идентификаторы foo, bar и baz должны совпадать с элементами именованного экспорта API модуля (применяется статический анализ и утверждение ошибки).

В текущей области видимости они связаны как идентификаторы верхнего уровня:

```
import { foo } from "foo";

foo();
```

Можно переименовать импортированные связанные идентификаторы:

```
import { foo as theFooFunc } from "foo";

theFooFunc();
```

Если модуль обладает только результатами экспорта по умолчанию, которые вы хотите импортировать и привязать к идентификатору, можно по вашему выбору опустить `{ .. }` для этого связывания.

```
import foo from "foo"; // или:

import { default as foo } from "foo";
```

Как объяснялось в предыдущем разделе, ключевое слово `default` в процедуре экспорта модуля задает именованный экспорт, в котором фигурирует имя `default`, как это показано во втором, более многословном варианте .

Изменение имени с `default` на другое (в данном случае на `foo`) в явном виде выполняется в последнем фрагменте и неявно в первом .

Кроме того, при наличии у модуля соответствующего определения можно импортировать результаты экспорта по умолчанию вместе с результатами именованного экспорта.

```
export default function foo() { .. }

export function bar() { .. } export function
baz() { .. }
```

Импортируем результаты экспорта по умолчанию этого модуля и два результата именованного экспорта.

```
import FOOFN, { bar, baz as BAZ } from "foo";

FOOFN(); bar(); BAZ();
```

Крайне рекомендуется придерживаться подхода из философии модулей в ES6, который гласит, что импортировать следует только конкретные привязки.

Если модуль предоставляет 10 методов API, а вам требуются только два из них, считается, что перетаскивать весь набор привязок API — пустая трата ресурсов.

Однако может понадобится вариант, когда вы импортируете все содержимое модуля в одно пространство имен вместо импорта отдельных членов в область видимости.

К счастью, у оператора `import` есть синтаксическая вариация, поддерживающая такой стиль работы с модулем. Она называется импортом пространства имен (`namespace import`).

Вы можете импортировать API целиком в единую привязку к пространству имен модуля:

```
import * as foo from "foo";

foo.bar();

foo.x;                                  // 42

foo.baz();
```

Если модуль, который импортируется с помощью выражения `* as ..`, обладает результатом экспорта по умолчанию, этот результат в указанном пространстве имен называется `default`.

Можно дополнительно именовать импорт по умолчанию извне привязки пространства имен как идентификатор верхнего уровня.

```
import foofn, * as hello from "world";

foofn();

hello.default();

hello.bar();

hello.baz();
```

Локальные импортированные привязки неизменяемы и/или предназначены только для чтения, и при попытке что-нибудь с ними сделать вы получаете ошибку TypeErrors.

Это очень важно, так как без подобной защиты ваши изменения в конце концов повлияли бы на других пользователей модуля (напоминаю: он представляет собой синглтон), что потенциально могло бы привести к непредсказуемым результатам.

Коллекции

TypedArrays

Типизированные массивы связаны с предоставлением структурированного доступа к бинарным данным с помощью напоминающей массив семантики (индексный доступ и т. п.).

Слово «тип» в названии относится к «представлению», наложенному на тип набора битов, которое, по сути, представляет собой отображение того, как именно будут рассматриваться эти биты: как массив 8-битных целых со знаком, 16-битных целых со знаком и т. п.

Совокупность битов называется буфером и чаще всего создается непосредственно конструктором `ArrayBuffer(..)`:

```
var buf = new ArrayBuffer( 32 );

buf.byteLength;                    // 32
```

Здесь `buf` — бинарный буфер длиной в 32 байта (256 бит), который заранее инициализирован значениями 0. Единственный доступный способ взаимодействия с этим буфером — проверка его свойства `byteLength`.

В этот буфер массива можно сверху поместить «представление» в форме типизированного массива.

```
var arr = new Uint16Array( buf );

arr.length;                 // 16
```

Здесь arr — это типизированный массив 16-битных целых без знака, отображенный на 256-битный буфер `buf`, то есть вы получаете 16 элементов.

Порядок байтов

Важно понимать, что типизированный массив `arr` отображается с учетом порядка байтов (от младшего к старшему или от старшего к младшему) платформы, на которой работает JS.

Если бинарные данные созданы на платформе с одним, а интерпретируются на платформе с противоположным порядком байтов, это может стать проблемой.

Порядок определяется тем, где именно, справа или слева, в многобайтовом числе — например, в 16-битном целом без знака, которое мы создали в предыдущем фрагменте кода, — находится младший байт (набор из 8 бит).

К примеру, возьмем десятичное число 3085, для представления которого требуется 16 бит.

При наличии всего одного 16-битного числового контейнера оно будет представлено в двоичной форме как 0000110000001101 (шестнадцатеричная форма 0c0d) вне зависимости от порядка байтов.

Но если представить 3085 в виде двух 8-битных чисел, порядок байтов сильно повлияет на способ сохранения в памяти:

- 0000110000001101 / 0c0d (от старшего к младшему);
- 0000110100001100 / 0d0c (от младшего к старшему).

Если вы получили представление числа 3085 в системе, где принят порядок от младшего к старшему, то есть 0000110100001100, а затем наложили поверх представление в системе с обратным порядком байтов, вы увидите значение 3340 (в случае основания 10) и 0d0c (в случае основания 16).

Существует быстрый способ от MDN, позволяющий проверить порядок байтов ваших сценариев JavaScript

Множественные представления

К одному буферу можно присоединить несколько представлений, например:

```
var buf = new ArrayBuffer( 2 );

var view8 = new Uint8Array( buf );

var view16 = new Uint16Array( buf );
```

```
view16[0] = 3085;

view8[0];                  // 13

view8[1];                  // 12

view8[0].toString( 16 );          // "d"

view8[1].toString( 16 );          // "c"
```

Конструкторы типизированных массивов имеют множество версий сигнатур.

У этой формы есть два дополнительных параметра: `byteOffset` и `length`.

Другими словами, представление типизированного массива можно начать в месте, положение которого отлично от 0, и он вовсе не должен заполнять буфер целиком.

Конструкторы типизированных массивов

Кроме формы (`buffer,[offset, [length]]`) конструкторы типизированных массивов могут выглядеть следующим образом:

- `[constructor\](length)` — создает новое представление в новом буфере с числом байт `length`;
- `[constructor\](typedArr)` — создает новое представление и новый буфер и копирует содержимое из представления `typedArr`;

`[constructor\](obj)` — создает новое представление и буфер и в цикле просматривает напоминающий массив объект `obj` с целью копирования его содержимого.

На момент появления ES6 доступны следующие конструкторы типизированных массивов:

- `Int8Array` (8-битные целые со знаком), `Uint8Array` (8-битные целые без знака); —`Uint8ClampedArray` (8-битные целые без знака, со значениями в диапазоне 0-255);

- `Int16Array` (16-битные целые со знаком), `Uint16Array` (16-битные целые без знака);

- `Int32Array` (32-битные целые со знаком), `Uint32Array` (32-битные целые без знака);

- `Float32Array` (32-битные с плавающей точкой, IEEE-754);
- `Float64Array` (64-битные с плавающей точкой, IEEE-754).

Экземпляры конструкторов типизированных массивов практически совпадают с обычными встроенными массивами.

Отличие состоит в фиксированной длине и значениях одного и того же типа.

Однако те и другие пользуются практически одинаковыми методами прототипов. Это означает, что с большой вероятностью можно иметь дело с типизированными массивами как с обычными, не прибегая к преобразованию.

Некоторые методы `Array.prototype` нельзя использовать с объектами `TypedArray`, например модификаторы (`splice(..)`, `push(..)` и т. п.) или `concat(..)`.

Элементы типизированных массивов и в самом деле ограничены объявленным размером битов.

Если элементу массива `Uint8Array` присвоить значение, размер которого превышает 8 бит, значение будет обернуто таким образом, чтобы остаться в указанных пределах длины.

Обойти это ограничение позволяет функция

Объект `TypedArray` обладает методом `sort(..)` как обычные массивы, но по умолчанию тот выполняет сравнение с помощью численной сортировки, а не преобразует значения в строки для лексикографического сравнения.

Метод `TypedArray.sort(..)` принимает в качестве необязательного аргумента функцию сравнения, аналогично работающему таким же способом методу `Array.sort(..)`.

Карты

В JS объекты — это основной механизм для создания структур данных, представляющих собой неупорядоченные пары «ключ/значение», также известных как карты.

При этом основной недостаток применения объектов в качестве карт — невозможность взять в качестве ключа нестроковое значение.

```
var m = new Map();

var x = { id: 1 }, y = { id: 2 };

m.set( x, "foo" );

m.set( y, "bar" );

m.get( x );                    // "foo"

m.get( y );                    // "bar"
```

Для удаления элемента из карты применяется не оператор `delete`, а метод `delete(..)`.

```
m.set( x, "foo" );

m.set( y, "bar" );

m.delete( y );
```

Содержимое всей карты удаляется методом `clear()`. Для получения информации о длине карты (то есть о количестве ключей) используется свойство `size` (а не `length`).

```
m.set( x, "foo" );

m.set( y, "bar" );

m.size;                          // 2

m.clear();

m.size;                          // 0
```

Конструктор `Map(..)` также может получать итерируемый объект, который должен сгенерировать список массивов, причем первый элемент каждого будет ключом, а второй — значением.

Такой формат итераций идентичен генерируемому методом `entries()`.

В конструкторе `Map(..)` можно просто вручную задать список элементов (массив ключа / массивы значений):

```
var x = { id: 1 }, y = { id: 2 };

var m = new Map( [ [ x, "foo" ], [ y, "bar" ] ]
);

m.get( x );                 // "foo"

m.get( y );                 // "bar"
```

Значения карт

Для получения списка значений карты применяется метод values(..), возвращающий итератор.

Существуют разные способы последовательной обработки итератора (напоминающей обработку массива), например, оператор распределения ... и цикл for..of. Кроме того, существует метод Array. from(..).

```
var    vals = [ ...m.values() ];

vals;                                                    // ["foo","bar"]

Array.from( m.values() );        // ["foo","bar"]
```

Элементы карты можно циклически просматривать с помощью метода entries() (или встроенного итератора карты).

```
var    vals = [ ...m.entries() ];

vals[0][0] === x;                // true

vals[0][1];                      // "foo"
```

Ключи карт

Для получения списка ключей карты используется метод keys(), возвращающий итератор.

```
var keys = [ ...m.keys() ];

keys[0] === x;                  // true

keys[1] === y;                  // true
```

Чтобы узнать, есть ли в карте конкретный ключ, используйте метод has(..).

По сути, карты позволяют ассоциировать дополнительную информацию (значение) с объектом (ключом), не добавляя ее к самому объекту.

В качестве ключа карты может использоваться любое значение, но, как правило, это объекты, так как строки и другие примитивы уже зарезервированы в качестве ключей обычных объектов.

Другими словами, вы, скорее всего, станете использовать для карт объекты, кроме случаев, когда некоторые или все ключи не должны быть объектами и поэтому больше подходят карты.

Если в качестве ключа карты используется объект, который позднее оказывается удаленным (исчезают все ссылки на него), для освобождения памяти при проходе сборщика мусора, карта все равно будет возвращать эту запись .

Чтобы сделать запись карты доступной для сборщика мусора, потребуется ее удаление .

Объекты WeakMap

Существует «слабая» вариация карты — объект WeakMap, обладающий таким же внешним поведением, но отличающийся тем, как под него выделяется память (и, в частности, как работает механизм сборки мусора).

Слабые карты принимают в качестве ключей только объекты, и те удерживаются слабо: если такой объект удаляется сборщиком мусора, то удаляется и соответствующая запись в коллекции WeakMap.

Но это не внешнее поведение, поскольку сборщик мусора подбирает только те объекты, на которые нет ни одной ссылки, — то есть вы не можете проверить, существует ли такой объект в коллекции WeakMap.

У коллекций WeakMap отсутствует свойство size и метод clear(), а еще нет итераторов для ключей, значений или элементов.

Так что даже если вы сбросите ссылку x, что приведет к удалению соответствующей записи из коллекции m посредством сборщика мусора, определить, что именно происходит, будет невозможно.

Важно заметить, что коллекция WeakMap слабо удерживает только ключи, но не значения.

Объекты Set

Объекты set представляют собой коллекции уникальных значений (дубликаты здесь игнорируются).

API для них примерно такой же, как для карт.

Вместо метода set(..) используется метод add(..), а метод get(..) вообще отсутствует.

Конструктор Set(..) напоминает конструктор Map(..), так как может принимать итерируемый объект, например другой объект set или массив значений.

Но если конструктор Map(..) ожидает список элементов (массив ключа / массивы значений), то конструктору Set(..) нужен только список значений (массив значений):

```
var    x = { id: 1 },    y = { id: 2 };

var    s = new Set( [x,y] );
```

Метод get(..) этой коллекции не требуется, так как элементы оттуда не извлекаются, а всего лишь проверяется наличие или отсутствие какого-либо из них с помощью метода has(..).

В методе has(..) используется почти такой же алгоритм сравнения, как в методе Object.is(..), но –0 и 0 интерпретируются как одно значение, а не как разные .

Итераторы коллекций Set

Коллекции set обладают теми же методами итератора, что и карты. Поведение этих методов отличается, хотя его можно до некоторой степени назвать симметричным.

В коллекциях set итераторы keys() и values() дают список уникальных значений. Итератор entries() предоставляет список массивов записей, в котором оба элемента массива представляют собой уникальное значение коллекции. По умолчанию для коллекции set используется итератор values().

WeakSets

Если коллекции WeakMap слабо держат свои ключи (цепляясь при этом за значения), то коллекции WeakSet слабо держат значения (ключей у них просто нет).

Значениями коллекции WeakSet могут быть только объекты, но ни в коем случае не примитивы, допустимые в коллекциях set.

Дополнения к API

Массив

- Статическая функция `Array.of(..)`
- У конструктора `Array(..)` есть всем известная странность.
- Если передать ему только один числовой аргумент, будет создан не массив из одного элемента, а пустой массив со свойством `length`, равным переданному числу.
- Это приводит к возникновению проблемного поведения «пустых слотов», которое так ругают, говоря о массивах в JS

```
var a = Array( 3 );
a.length; // 3
a[0]; // undefined
var b = Array.of( 3 );
b.length; // 1
b[0]; // 3
var c = Array.of( 1, 2, 3 );
c.length; // 3
c; // [1,2,3]
```

- **Статическая функция `Array.from(..)`**
- Напоминающим массив объектом в JavaScript называется объект, обладающий свойством `length`, значения которого представлены целым числом от нуля и выше.

```
// объект, напоминающий массив
var arrLike =
{
length: 3,
0: "foo",
1: "bar"
};
```

- Метод `Array.from(..)` проверяет, является ли первый аргумент итерируемым.
- Если это так, итератор используется для генерации значений, которые «копируются» в возвращаемый массив.
- Реальные массивы имеют для таких случаев встроенный итератор, применяющийся автоматически.
- Но если передать в качестве первого аргумента методу Array. from(..) объект, напоминающий массив, он поведет себя как метод slice() (не имеющий аргументов) или метод apply(..), то есть просто начнет перебирать значения в цикле, обращаясь к именованным свойствам, имена которых начинаются с 0 и заканчиваются значением свойства length.

```
var arrLike = {
length: 4, 2: "foo“
 };
Array.from( arrLike );
// [ undefined, undefined, "foo", undefined ]
```

Так как позиций 0, 1 и 3 в объекте `arrLike` не существует, для каждого из этих слотов результатом становится значение `undefined`.

Метод `Array.from(..)` никогда не создает пустых слотов.

- Отображение
- Метод `Array.from(..)` умеет делать еще одну полезную вещь.
- Второй аргумент, если он присутствует, представляет собой отображающий обратный вызов (почти то же самое, чего ожидает обычный метод `Array#map(..)`), который активируется, когда нужно отобразить/преобразовать каждое значение из источника в возвращаемый целевой объект.

```
var arrLike = { length: 4, 2: "foo" };
 Array.from( arrLike, function mapper(val,idx){
if (typeof val == "string") {
return val.toUpperCase();
}
else {
return idx;
}
} );
// [ 0, 1, "FOO", 3 ]
```

- Оба метода, `of(..)` и `from(..)`, используют для создания массива конструктор, из которого они были вызваны.
- Соответственно, если в качестве основы использовать метод `Array.of(..)`, получится экземпляр класса `Array`, в то время как метод `MyCoolArray.of(..)` даст вам экземпляр `MyCoolArray`.

- **Метод прототипа `copyWithin(..)`**
- Новый модифицирующий метод `Array#copyWithin(..)` доступен для всех массивов (в том числе для типизированных).
- Метод `copyWithin(..)` копирует фрагмент массива в другую позицию, переписывая бывшие там ранее значения.
- Его аргументы — `target` (индекс позиции, в которую осуществляется копирование), `start` (начальный индекс позиции источника копируемых элементов) и по желанию `end` (конечный индекс позиции источника).
- В случае отрицательного значения любого элемента он начинает отсчитываться от конца массива.

```
[1,2,3,4,5].copyWithin(3, 0); //[1,2,3,1,2]

[1,2,3,4,5].copyWithin(3, 0, 1); //[1,2,3,1,5]

[1,2,3,4,5].copyWithin(0, -2); //[4,5,3,4,5]

[1,2,3,4,5].copyWithin(0, -2, -1); //[4,2,3,4,5]
```

Метод `copyWithin(..)` не увеличивает длину массива. При достижении конца массива копирование просто останавливается.

- Вопреки распространенному мнению, копирование далеко не всегда выполняется слева направо (с возрастающим индексом).
- В случае перекрытия диапазонов источника элементов и цели возможно многократное копирование уже скопированных значений, что, очевидно, нельзя назвать желательным поведением.
- Поэтому алгоритм избегает подобной ситуации путем копирования в обратном порядке.

```
[1,2,3,4,5].copyWithin( 2, 1 ); // ???
```

- При смещении слева направо 2 будет скопировано, чтобы переписать 3, затем это скопированное значение 2 снова будет скопировано, чтобы переписать значение 4, затем таким же образом будет переписано значение 5, и в результате вы получим массив [1,2,2,2,2].
- Если же поменять направление работы алгоритма, то сначала будет скопировано 4, чтобы переписать значение 5, затем — 3, чтобы переписать 4, затем — 2, чтобы переписать 3, и в итоге получим [1,2,2,3,4].
- Такой рез-т является более корректным с точки зрения ожиданий, но нельзя понять, как он был получен, ограничившись представлением о направлении коп-я слева направо.

- **Метод прототипа `fill(..)`**
- Заполнение существующего массива полностью (или частично) одним значением поддерживается в ES6 при помощи метода `Array#fill(..)`:

```
var a = Array( 4 ).fill( undefined );
a; // [undefined,undefined,undefined,undefined]
```

- В качестве необязательных параметров метод `fill(..)` принимает значения начального и конечного индексов, определяющих заполняемый фрагмент массива:

```
var a = [null, null, null, null].fill(42, 1, 3);
a; // [null,42,42,null]
```

- **Метод прототипа `find(..)`**
- Самым распространенным способом поиска значения в массиве в общем случае был метод `indexOf(..)`, возвращающий индекс обнаруженного значения или `-1`, если оно в массиве отсутствует.
- Поиск при помощи метода `indexOf(..)` требует строгого соответствия `===`, поэтому по значению `"2"` нельзя найти `2` и наоборот.
- Переопределить алгоритм поиска соответствия для метода `indexOf(..)` невозможно.
- Еще одна проблема заключается в необходимости вручную проверять равенство значению `-1`.

- С ES5 самым распространенным способом получения контроля над логической схемой поиска соответствий был метод `some(..)`.
- Он создает для каждого элемента обратный вызов функции, пока не получит в результате такого вызова значение `true`/истинное значение, после чего работа останавливается.
- Так как функцию обратного вызова определяете вы сами, у вас есть полный контроль над способом сопоставлений.
- Но, к сожалению, в таком случае вы получаете только значение `true/false`, указывающее на обнаружение соответствия, однако само совпавшее значение не выводится.

- Эту проблему решает появившийся в ES6 метод `find(..)`. В своей основе он функционирует аналогично методу `some(..)`, но после возвращения обратным вызовом значения `true` передается еще и значение массива.
- Кроме того, существует пользовательская функция `matcher(..)`, позволяющая сопоставлять такие сложные значения, как объекты.

- **Метод прототипа `findIndex(..)`**
- ES6 появился новый метод `findIndex(..)` позволяющей управлять логической схемой поиска соответствий.

```
points.findIndex(
    function matcher(point) { return (
point.x % 3 == 0 && point.y % 4 == 0
); }
); // 2
```

- Также сужествует метод поиска в массивах, возвращающий логическое значение, которое называется `includes(..)`:

```
var vals = [ "foo", "bar", 42, "baz" ];
if (vals.includes( 42 )) {
 // найдено!
}
```

НЕ ВХОДИТ В СТАНДАРТ ES6

- **Методы прототипа `entries()`, `values()`, `keys()`**
- Так как объекты `Array` были стандартизированы еще до ES6, традиционно их не рассматривают в качестве коллекций, но они могут называться таковыми благодаря наличию все тех же методов итератора: `entries()`, `values()` и `keys()`.

```
var a = [1,2,3];
[...a.values()]; // [1,2,3]
[...a.keys()]; // [0,1,2]
[...a.entries()]; // [ [0,1], [1,2], [2,3] ]
[...a[Symbol.iterator]()]; // [1,2,3]
```

- **`reduce()`**
- Этот метод принимает функцию, которая имеет в качестве аргумента аккумулятор и значение. Он применяет функцию к аккумулятору и каждому значению массива, чтобы в результате вернуть только одно значение.

```
const myArray = [1, 2, 3, 4, 5]

myArray.reduce((total, value) => total * value)
// 1 * 2 * 3 * 4 * 5
//-------> Output = 120
```

- **`every()`**
- Этот метод проверяет, удовлетворяют ли все элементы массива условию, заданному в передаваемой функции. Он вернет значение `true`, если каждый элемент совпадет с проверяемой функцией, и значение `false` — если нет.

```
const myArray = ["a", "b", "c", "d", "e"]
myArray.every(test => test === "d")
//-------> Output : false
const myArray2 = ["a", "a", "a", "a", "a"]
myArray2.every(test => test === "a")
//-------> Output : true
```

- **`map()`**
- Этот метод принимает функцию в качестве параметра и создает новый массив с результатом вызова указанной функции для каждого элемента массива. Он всегда будет возвращать одинаковое количество элементов.

```
const myArray = [5, 4, 3, 2, 1]
myArray.map(x => x * x)
//-------> Output : 25
//                  16
//                  9
//                  4
//                  1
```

- **forEach()**
- Этот метод применяет функцию к каждому элементу массива.

```
const myArray = [
   { id: 1, name: "john" },
   { id: 2, name: "Ali" },
   { id: 3, name: "Mass" },
]
myArray.forEach(el => console.log(el.name))
//-------> Output : john
//                   Ali
//                   Mass
```

- **`flat()`**
- Этот метод принимает в качестве аргумента массив массивов и сглаживает вложенные массивы в массив верхнего уровня. Обратите внимание, что этот метод работает только для одного уровня.

```
const myArray = [[1, 2], [3, 4], 5]

myArray.flat()
//-------> Output : [1, 2, 3, 4, 5]
```

- **`flatMap()`**
- Этот метод применяет функцию к каждому элементу массива, а затем сглаживает результат в новый массив. Он объединяет метод `flat()` и метод `map()` в одну функцию.

```
const myArray = [[1], [2], [3], [4], [5]]
myArray.flatMap(arr => arr * 10)
//-------> Output : [10, 20, 30, 40, 50]
// With .flat() and .map()
myArray.flat().map(arr => arr * 10)
//-------> Output : [10, 20, 30, 40, 50]
```

- `filter()`

- Этот метод принимает функцию в качестве параметра и возвращает новый массив, содержащий все элементы массива, для которого функция фильтрации передавалась в качестве аргумента, и возвращает ее со значением true.

```
const myArray = [
  { id: 1, name: "john" },
  { id: 2, name: "Ali" },
  { id: 3, name: "Mass" },
  { id: 4, name: "Mass" },
]

myAArray.filter(element => element.name ===
"Mass")
//-------> Output : 0:{id: 3, name: "Mass"},
//                  1:{id: 4, name: "Mass"}
```

Объект

- **Статическая функция `Object.is(..)`**
- Статическая функция `Object.is(..)` сравнивает значения еще более строгим образом, чем оператор ===. Метод `Object.is(..)` активизирует лежащий в его основе алгоритм `SameValue` (спецификацияES6, раздел 7.2.9).
- По сути он аналогичен алгоритму строгой операции сравнения === (спецификация ES6, раздел 7.2.13), но с двумя важными исключениями.

```
var x = NaN, y = 0, z = -0;
x === x; // false
y === z; // true
Object.is( x, x ); // true
Object.is( y, z ); // false
```

- Для строгих сравнений нужно пользоваться оператором `===`; не следует рассматривать метод `Object.is(..)` как его замену.
- Но в случаях, когда требуется строго идентифицировать значение `NaN` или `-0`, поможет `Object.is(..)`.

- **Статическая функция `Object.getOwnPropertySymbols(..)`**
- Скорее всего, символы будут использоваться в основном как специальные свойства (метасвойства) объектов. Поэтому в ES6 появился метод `Object.getOwnPropertySymbols(..)`, извлекающий из объектов исключительно их символьные свойства:

```
var o = { foo: 42, [ Symbol( "bar" ) ]: "hello
world", baz: true };
Object.getOwnPropertySymbols( o );
// [ Symbol(bar) ]
```

- **Статическая функция `Object.setPrototypeOf(..)`**
- Метод `Object.setPrototypeOf(..)`, задает прототип объекта [[Prototype]] для делегирования поведения

```
var o1 = { foo() { console.log( "foo" ); } };
var o2 = { // .. определение o2 .. };
Object.setPrototypeOf( o2, o1 );
// делегирует в 'o1.foo()'

o2.foo(); // foo
```

```
var o1 = { foo() { console.log( "foo" ); } };
var o2 = Object.setPrototypeOf( {
 // .. определение o2 ..
}, o1 );
// делегирует в 'o1.foo()'

o2.foo(); // foo
```

- Статическая функция `Object.assign(..)`
- В ES6 появился метод `Object.assign(..)`, представляющий упрощенную версию этих алгоритмов. Его первый аргумент — целевой объект (`target`), а все остальные передаваемые аргументы — объекты-источники (`sources`), которые обрабатываются в порядке перечисления.
- Перечисляемые и собственные (то есть не «унаследованные») ключи каждого источника, в том числе и символы, копируются так же, как и в случае оператора присваивания =.
- Метод `Object.assign(..)` возвращает целевой объект.

- Рассмотрим пример настройки объекта:

```
var target = {},
o1 = { a: 1 }, o2 = { b: 2 }, o3 = { c: 3 },
o4 = { d: 4 };
// устанавливаем свойство только для чтения
Object.defineProperty( o3, "e", {
 value: 5,
enumerable: true,
writable: false,
configurable: false
} );
```

```
// устанавливаем неперечисляемое свойство
Object.defineProperty( o3, "f", {
 value: 6,
 enumerable: false
} );
o3[ Symbol( "g" ) ] = 7;
// устанавливаем неперечисляемый символ
Object.defineProperty( o3, Symbol( "h" ), {
 value: 8,
 enumerable: false
} );
Object.setPrototypeOf( o3, o4 );
```

- В целевой объект будут скопированы только свойства `a`, `b`, `c`, `e` и `Symbol("g")`.

```
Object.assign( target, o1, o2, o3 );
target.a; // 1
target.b; // 2
target.c; // 3
```

```
Object.getOwnPropertyDescriptor( target, "e" );
// { value: 5,
// writable: true,
// enumerable: true,
// configurable: true }

Object.getOwnPropertySymbols( target );
// [Symbol("g")]
```

- Свойства `d`, `f` и `Symbol("h")` при копировании опускаются; из операции присваивания исключаются все свойства, которые не являются перечисляемыми и собственными.
- Кроме того, e копируется как обычное свойство присваивания, а не как предназначенное только для чтения.

- С помощью метода `setPrototypeOf(..)` можно настроить соотношение вида `[[Prototype]]` между объектами `o2` и `o1`.

- Рассмотрим другой вариант установки этого взаимодействия, в котором используется метод `Object.assign(..)`:

```
var o1 = {
 foo() { console.log( "foo" ); }
};
var o2 = Object.assign(
 Object.create( o1 ),
 {
 // .. определение o2 ..
 }
);
// делегирует в 'o1.foo()'
o2.foo(); // foo
```