Яндекс

Типы данных (часть 2)

Евгений Марков, разработчик интерфейсов

В предыдущих сериях

- Примитивы
- Числа
- Строки
- Массивы
- Объекты
- Функции

Сегодня в программе

- Объекты и их методы
- JSON
- Symbol
- for...of
- Map, Set
- Даты
- Регулярные выражения

Что мы знаем об объектах?

- Всё что не number, string, boolean, null, undefined, symbol* – Объект
- Функции – тоже объекты
- Это набор свойств, свойство состоит из ключа и значения
- Все ключи приводятся к строке
- Для итерации по ключам есть оператор for … in
- Для проверки наличия свойства есть оператор in
- Есть 2 способа обращаться к свойствам: obj['key'] и obj.key
- При передаче объектов в функции копируются ссылки на них
- typeof null возвращает 'object'
- typeof someFunction возвращает 'function'
- Boolean({}) возвращает true

Пример объекта

```
const tweet = {
  id: 10,
  text: 'Я и IoT, доклад на WSD Вадима Макеева #wstdays',
  user: {
    id: 20,
    name: 'Веб-стандарты',
    screenName: 'webstandards_ru',
    followersCount: 6443
  },
  hashtags: ['wstdays']
};
```


Если кто-то из вас уже обменивался данными по сети, то вы знаете, что для этого часто используется формат данных JSON. Видно, что на скрине практически JSON объект. Это не случайно, JSON является подмножеством JavaScript.

JSON

```
type Json =
     | string
     | number
     | boolean
     | null
     | { [property: string]: Json }
     | Json[];
```

JSON ⊂ ECMAScript

В JavaScript объектах можно хранить что угодно, в JSON только то, что можно безболезненно преобразовать в строку: string, number, boolean, null, массив тех же типов данных, объект с ключами-строками и теми же типами данных в качестве значений.

Работа с JSON

JSON.stringify(any) – преобразование JS сущностей в строку (сериализация)

JSON.parse(str) – преобразование строки в JS сущности (десериализация)

```
JSON.stringify(tweet);

/*
"{
  "id": 782188596690350100,
  "text": "Я и IoT, доклад на WSD Вадима Макеева #wstdays",
  "user": {
    "id": 42081171,
    "name": "Веб-стандарты",
    "screenName": "webstandards_ru",
    "followersCount": 6443
  },
  "hashtags": [
    "wstdays"
  ]
}"
*/
```


Для работы с JSON есть 2 статических метода у класса JSON: stringify и parse. stringify преобразовывает JS сущность в строку, этот процесс называется "сериализация", parse делает обратную операцию ("десериализацию"). С JSON вам определённо нужно будет работать, независимо от того будете вы фронтендерами или нет. Но мы отвлеклись, давайте вернёмся к JS объектам.

Аккуратнее с ключами!

```
const server = { url: 'https://example.com' };
const proxy = { url: 'https://anonymous-proxy.com' };

server[proxy] = { enable: true };

{
  "url":"https://example.com",
  "[object Object]": { "enable": true }
}
```


Как мы уже сказали, в объектах ключами могут быть только строки, а всё что не строки – преобразуется к строке. Посмотрим на примере как это происходит.

Свойства-функции

```
const tweet = {
  likes: 16,

  getLikes: function () {
    return this.likes;
  },

  setLikes: function (value) {
    this.likes = parseInt(value) || 0;
    return this;
  }
};

tweet.getLikes(); // 16
```

Сравнение объектов

```
const obj1 = { a: 1 };
const obj2 = { a: 1 };

obj1 === obj2 // ?
```

Сравнение объектов

```
const obj1 = { a: 1 };
const obj2 = obj1;

obj1 === obj2 // true
```

Операция сравнения двух сложных типов вернёт истину только если ссылки обоих объектов указывают на один и тот же объект в памяти

Deep Equal

- Универсального встроенного алгоритма нет
- Есть костыль: `JSON.stringify(a) === JSON.stringify(b)`

- Сравнивайте нужные поля вручную
- Используйте сторонние реализации

Lo | Lodash
A modern JavaScript utility library delivering modularity, performance & extras.

```
_.isEqual(value, other)
```

Методы и свойства экземпляров Object

- hasOwnProperty() – почти как `key in obj`
- toString()
- toLocaleString()
- valueOf()

- propertyIsEnumerable()
- isPrototypeOf()
- __proto__

toString()

```
const bear = {
  face: '🐻'
};

panda.toString(); // '[object Object]'
```

toString()

```
const bear = {
  face: '🐻',

  toString: function() {
    return this.face;
  }
};

bear.toString(); // '🐻'
```

toLocaleString()

```
const bear = {
  face: '🐻',

  toString: function() {
    return this.face;
  },

  toLocaleString: function() {
    if (navigator.language === 'zh') {
      return '🐼';
    }

    return this.toString();
  }
};

bear.toString(); // '🐻'
bear.toLocaleString(); // '🐼'
```

Приведение к строке

```
const panda = {
  face: '🐼',

  toString: function () {
    return this.face;
  }
};

panda.toString(); // '🐼'

String(panda); // '🐼'

'' + panda; // '🐼'
```

```
const arr = [true, 1, 'hello', null, undefined, { key: 'value' }];

arr.toString(); // "true,1,hello,,,[object Object]"
```

valueOf()

- Используется для численного преобразования объекта
- Если не определён, то вместо него вызывается toString()

```
const room = {
  numberStr: '777',
  numberNum: 777,

  toString: function() {
    return this.numberStr;
  },

  valueOf: function() {
    return this.numberNum;
  }
};

+room; // 777, вызвался valueOf

delete room.valueOf; // valueOf удалён

+room; // 777, вызвался toString
```

Объявление свойств объекта (Basic)

```
const bear = {
  'color': 'brown'
}

const bear = {
  color: 'brown' // '' Можно опускать
}

const color = 'brown';

const bear = {
  color // Короткая форма записи для color: color
}

bear.name = 'Пух';
bear['name'] = 'Пух';
```

```
const field = 'name';

const bear = {
  [field]: 'brown'
};

bear[field] = 'white';
```

Объявление свойств объекта (Advanced)

```
const panda = {};

Object.defineProperty(panda, 'color', {
  value: 'black',
  writable: true,
  enumerable: true,
  configurable: true
});
```

Объект

Свойство

Дескриптор

Атрибут writable

```
const panda = {};

Object.defineProperty(panda, 'color', {
  value: 'black',
  writable: false // запрет редактирования
});

panda.color = 'white';

// TypeError: Cannot assign to read only property 'color'
```

Атрибут configurable

```
const panda = {};

Object.defineProperty(panda, 'color', {
  value: 'black',
  configurable: false // запрет удаления
});

delete panda.color;

// TypeError: Cannot delete property 'color'
```

Атрибут enumerable

```
const panda = {
  color: 'black',
  toString: () => '🐼'
};

Object.keys(panda); // ['color', 'toString']
```

Атрибут enumerable

```
const panda = {
  color: 'black'
};

Object.defineProperty(panda, 'toString', {
  value: () => '🐼',
  enumerable: false // исключение toString из перечисления
});

Object.keys(panda); // ['color']
```

Значения по умолчанию

Значения параметров writable, enumerable и configurable по умолчанию — true.

```
const panda = {
  color: 'black'
};

Object.getOwnPropertyDescriptors(panda);

/*
{
  color: {
     value: 'black',
     writable: true,
     enumerable: true,
     configurable: true
   }
}
*/
```

Заморозка

```
const panda = {
  color: 'black'
};

Object.freeze(panda);

Object.isFrozen(panda); // true

Object.getOwnPropertyDescriptor(panda, 'color');

/*
{
  value: 'black',
  writable: false,
  enumerable: true,
  configurable: false
}
*/
```

Заморозка

- Действует нерекурсивно
- Изменяет объект, не создаёт «замороженную» копию

```
const panda = {
  classification: {
    kingdom: 'Animalia',
    class: 'Mammalia'
  }
};

Object.freeze(panda);
panda.classification.kingdom = 'Fungi';

panda
/*
{
  classification: {
    kingdom: 'Fungi',
    class: 'Mammalia'
  }
}
*/
```

Заморозка

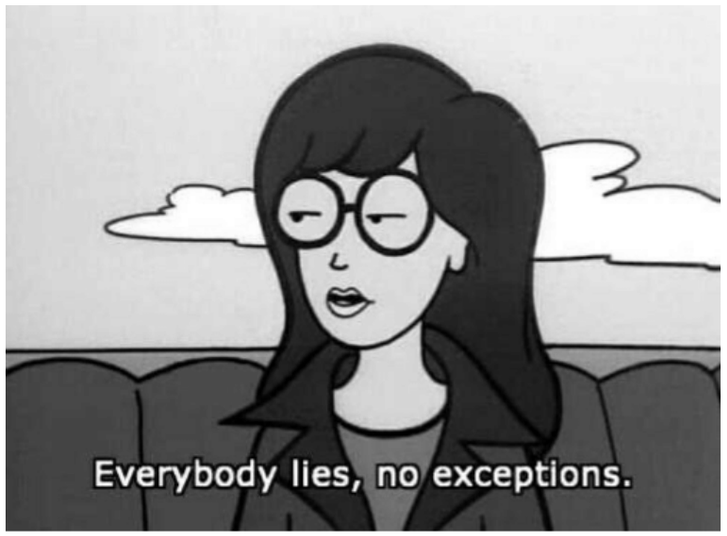
- Действует нерекурсивно
- Изменяет объект, не создаёт «замороженную» копию

```
const panda = {
  classification: {
    kingdom: 'Animalia',
    class: 'Mammalia'
  }
};

Object.freeze(panda);
panda.classification.kingdom = 'Fungi';

panda
/*
{
  classification: {
    kingdom: 'Fungi',
    class: 'Mammalia'
  }
}
*/
```

Примитивов больше чем 5



Symbol

Symbol – уникальный идентификатор

```
// Создание символа
const id = Symbol();

// Создание символа с описанием
const id = Symbol('id');

// Сравнение символов с одинаковым описанием
const id1 = Symbol('id');
const id2 = Symbol('id');

id1 === id2; // false
```


Мы долго от вас скрывали, но на самом деле помимо string, number, boolean, null и undefined есть ещё шестой примитивный тип данных Symbol.

Символы создаются вызовом функции Symbol, без оператора new.
Им можно задать описание, это просто метка и она ни на что не влияет.

Особенности Symbol

- Могут быть ключами в объекте
- Не преобразуются автоматически к строке

```
const user = {
  name: 'Вася'
};

const id = Symbol('id');

user[id] = 1;

console.log(user[id]);
// 1
```


Мы раньше говорили, что ключом объекта может быть только string, а всё что не string преобразуется в string через toString. С символом это не так. Он никак не преобразуется и символы считаются вторым допустимым типом данных для ключей объектов.

Предположим, у нас есть некий пользователь Вася. Этот объект настолько стар, к нему уже так много обращений в коде, что добавлять новые поля страшно и это сломает программу во многих местах. А мы хотим добавить Васе какой-нибудь id'шник чтобы с ним работать из нового кода. Для этого создадим символ, по символу в объект запишем значение и теперь имея доступ к символу и объекту мы всегда можем получить этот id'шник, а в старом коде ничего не сломается. Почему?

Особенности Symbol

- Символьные ключи нельзя перебрать

```
const id = Symbol('id');

const user = {
  name: 'Вася',
  age: 30,
  [id]: 123
};

// Перебор ключей объекта user
for (const key in user) {
  console.log(key);
}
// name, age

// Обращение к id напрямую
console.log(user[id]);
// 123
```


Символ позволяет добавлять "скрытые" поля. К ним можно обратиться напрямую, можно достать их специальными методами, но обычными форами или часто используемыми методами объектов символьные свойства не получить.

В примере попробуем перебрать все обычные свойства циклом for...in. Конечно же у нас это получится. Но цикл не выведет скрытое поле.

Я вас заранее разочарую, но Symbol хоть и кажется супер полезным, но применяется нечасто в типовом промышленном коде. Но этот тип данных играет большую роль в сохранении самим языком обратной совместимости, а также в добавлении новых фич. Давайте смотреть его тонкости дальше.

Глобальный реестр символов

- *Symbol.key('id')* – Чтение/запись в глобальный реестр
- *Symbol.keyFor(symbol)* – Получение описания по символу в глобальном реестре

```
// Читаем символ из глобального реестра и записываем его в переменную
const id = Symbol.for('id'); // если символа не существует, он будет создан

// Читаем его снова в другую переменную (возможно, из другого места кода)
const idAgain = Symbol.for('id');

// Проверяем, что id и idAgain — тот же символ
id === idAgain; // true
```

```
const globalSymbol = Symbol.for('name');
const localSymbol = Symbol('name');

Symbol.keyFor(globalSymbol); // 'name'
Symbol.keyFor(localSymbol); // undefined
```


Иногда мы хотим, чтобы символы с одинаковыми описаниями были одной сущностью. Например, мы хотим инициализировать все символы в точке входа в программу, а получить их в произвольном месте.
Для этого существует глобальный реестр.

Для работы с ним есть 2 статических метода Symbol.key и Symbol.keyFor. for с описанием позволяет записать в реестр символ или, если он уже есть, то получить его. keyFor наверное наименее часто используемый метод, он нужен для получения описаний символов в глобальном реестре.

Зачем такое может пригодиться? Ну например, чтобы подменять реализации стандартных методов в JavaScript объектах, также символы могут быть применены при реализации паттерна "Inversion Of Control", с помощью которого можно по интерфейсу, какому-то идентификатору зарегистрировать реализацию, а потом получить эту реализацию, таким образом уменьшив связность кода.

Системные символы

- Symbol.iterator
- Symbol.hasInstance
- Symbol.toPrimitive
- ...

Well-known symbols

В стандарте есть так называемые "хорошо известные" символы. Они используются внутри самого JavaScript и позволяют настраивать различные аспекты поведения объектов.

Вот мы уже почти до самого интересного.

Сколько способов написать цикл вы знаете?

Но перед этим ответьте на вопрос. Сколько различных вариантов написания цикла вы знаете?

for...of

```
const iterable = {
  [Symbol.iterator]() {
    let step = 0

    const iterator = {
      next() {
        step++

        switch (step) {
          case 1:
            return { value: 'This', done: false }
          case 2:
            return { value: 'is', done: false }
          case 3:
            return { value: 'iterable', done: false }
          default:
            return { value: undefined, done: true }
        }
      }
    }

    return iterator
  }
}

for (const step of iterable) {
  console.log(step)
}
```


Теперь узнаете ещё один цикл – for...of.

Классический for можно использовать со всем у чего есть индексы.
С помощью for...in пробежаться по ключам объекта.

A for...of может быть применён к любому "перебираемому объекту". Что значит перебираемый? Значит, что он реализует Iteration Protocol.

Давайте на примере. Чтобы сделать объект перебираемым, мы должны добавить скрытый метод доступный по символу Symbol.iterator. Метод должен знать как итерироваться по объекту и вернуть при вызове объект у которого есть один единственный метод next(). next() должен возвращать объект с полями value и done. value – следующее значение из перебираемого объекта, a done – флаг, который говорит о том, что пора завершить перебор.

Iterator Protocol реализован для строк, для массивов. Многие авторы сторонних библиотек со структурами данных для JS реализуют его.

С символами и for..of мы всё. Совет напоследок: используйте for...of всегда когда реализован Iterator Protocol у объекта. Если вам нужна работа с индексами – берите классический for. Если нужно что-то близкое к классическому for, но со сложными условиями итерации – юзайте while. Забудьте про for...in. Если хотите пройтись по ключам или значениям объекта – возьмите их через Object.keys или Object.values и пройдитесь for..of'ом.

Map и Set

Появились в ECMAScript 2015

Map – коллекция для хранения записей вида ключ:значение.
Ключом, в отличие от объектов, может быть *произвольное* значение.
Область применения – ситуации, когда строковых ключей не хватает.

Set – коллекция для хранения множества значений, причём каждое может встречаться лишь один раз.

Map. Инициализация

```
// Создание Map
const map = new Map();

// Сохранение значений
map.set('1', 'str1');
map.set(1, 'num1');
map.set(true, 'bool1');

// Чейнинг
map
  .set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

```
// Инициализация при создании
const map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, 'bool1']
]);
```

Map. Методы и свойства

size – количество сохранённых пар ключ:значение
set(key, value) – сохранение новой пары ключ:значение или изменение старой
get(key) – получение пары по ключу
has(key) – проверка наличия пары с заданным ключом
delete(key) – удаление пары по ключу
clear() – удаление всех пар из коллекции
keys() – получение всех ключей пар
values() – получение всех значений пар
entries() – получение всех пар ([[key1, value1], [key2, value2]])
forEach(callbackFn) – итерация по коллекции

MDN

Map. Нюансы

- Сравнение ключей аналогично строгому равенству (===)
- Перебор идёт в том же порядке, что и вставка

Set. Инициализация

```
const visits = new Set()

const vasya = { name: 'Вася' }
const petya = { name: 'Петя' }
const dasha = { name: 'Даша' }

// посещения, некоторые пользователи заходят много раз
visits.add(vasya)
visits.add(petya)
visits.add(dasha)
visits.add(vasya)
visits.add(petya)

// set сохраняет только уникальные значения
visits.size; // 3

visits.forEach(user => user.name); // Вася, Петя, Даша
```

Set. Методы и свойства

size – количество сохранённых значений
add(value) – добавление значения
has(value) – проверка наличия значения
delete(value) – удаление значения
clear() – удаление всех значений
values() – получение всех значений
forEach(callbackFn) – итерация по коллекции

Внутренне Set похож на Map. Не удивляйтесь наличию методов keys() и entries(), которые возвращают тоже самое, что и values().

MDN

Даты

Date – встроенный тип данных для работы с датами и временем

```
// Создание объекта Date с текущими датой и временем
new Date();

// Создание даты из UNIX Timestamp
// e.g. timestamp = 1540300025000
new Date(timestamp);

// Создание объекта из строки в формате ISO 8601
// e.g. dateString = '2019-10-04T17:36:57.534Z'
new Date(dateString);

// Создание даты из набора параметров
// Параметры можно опускать
new Date(year, month, date, hours, minutes, seconds, ms);
```


In JavaScript, dates are internally represented as an integer number of milliseconds since the Unix epoch date (Jan 1, 1970, midnight UTC).

Статические методы Date

- Date.now()
- Date.UTC(year, month, day, hours, minutes, seconds, ms)

```
Date.now();

// UNIX timestamp
// 1570216418206
```

```
Date.UTC(2019, 9, 4, 10, 0, 0, 0);

// timestamp, e.g. 1570183200000
```

Методы и свойства экземпляров Date

```
const date = new Date(2018, 9, 23, 17, 50, 05);
```

```
date.getFullYear(); // 2018
date.getMonth(); // 9
date.getDate(); // 23
date.getHours(); // 17
date.getMinutes(); // 50
date.getSeconds(); // 5
```

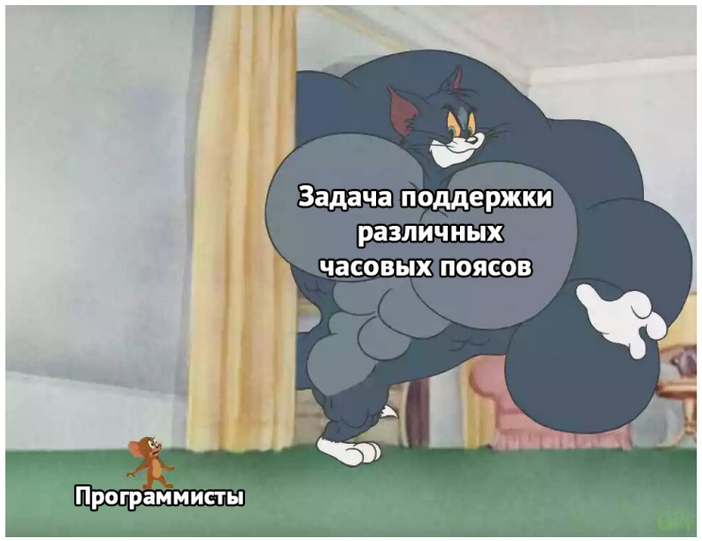
```
date.setFullYear(2020);
date.setMonth(1);
date.setDate(25);
date.setHours(4);
date.setMinutes(7);
date.setSeconds(0);
```

MDN

Странности

- date.getMonth() возвращает месяц от 0 до 11
- date.getDay() возвращает день недели от 0 до 6. Неделя в JavaScript начинается с воскресенья.
- Можно создать new Date(2019, 0, 32), что эквивалентно 1 февраля.
- Даты вычитаются, результат в миллисекундах
- Стандарт ISO8601 поддерживается не полностью
- Есть мелкие расхождения в имплементации стандарта браузерами
- Браузеры неявно приводят даты к временной зоне пользователя

Post-production проблемы



Универсальная таблица оценки задач

изян – 1ч
изи – 2ч
просто – 4ч
вроде просто – 6ч
норм – 8ч
норм так – 12ч
хз – 16ч
хз как-то – 20ч
как-то сложно – 24ч
сложно – 30ч
очень сложно – 40ч
б** – 48ч
п****ц – 60ч
п****ц какой-то – 80ч
вроде изян – 100ч
поддержать таймзоны – ∞

Библиотеки

- date-fns
- luxon
- moment
- ...

Регулярные выражения

Это мощный инструмент поиска и замены в тексте

- В JS имеют стандартный PCRE синтаксис
- Реализованы отдельным типом данных RegExp
- Интегрированы в методы экземпляров String

Есть замечательный сервис для тестирования регулярок – regex101

Способы создания

```
const regexp = new RegExp(pattern, flags);

const regexp = /pattern/; // без флагов
const regexp = /pattern/gi; // с флагами
```

Флаги

i – делает поиск регистронезависимым (A === a)
g – с этим флагом происходит поиск всех совпадений, не только первого
u – включает полную поддержку Unicode (🙈)

m – включает многострочный режим
s – когда ищем любой символ (точка, .) учитывает и переносы строки
y – включает режим поиска на конкретной позиции

Методы строк

str.search(regexp) – возвращает позицию первого совпадения
str.match(regexp) – возвращает совпадение(я) с регулярным выражением
str.replace(regexp, replacement) – заменяет совпадение(я) на replacement
str.split(regexp) – разбивает строку на массив строк по заданной регулярке

```
const str = 'anananas'

str.match(/an/)
[ 'an', index: 0, input: 'anananas', groups: undefined ]

str.match(/an/g)
[ 'an', 'an', 'an' ]
```

Методы экземпляров RegExp

regexp.test(str) – выполняет сопоставление регулярки указанной строке
regexp.exec(str) – выполняет поиск сопоставлений регулярки в строке

```
const str = "Я ЛюБлЮ JavaScript";
const regexp = /люблю/i;

regexp.test(str); // true
```

Полезные ссылки

- Объекты:основы на learn.javascript.ru
- Туториал и документация по объектам на MDN
- Про Map и Set на learn.javascript.ru
- Про Map и Set на MDN
- Работа с датами на learn.javascript.ru
- Документация по датам на MDN
- Регулярные выражения на learn.javascript.ru
- Туториал и документация по регулярным выражениям на MDN
- Устройство цикла for...of на MDN

Яндекс

Спасибо

Евгений Марков

Разработчик интерфейсов

evgenymarkov@yandex-team.ru