

Fundamentals of JavaScript 2

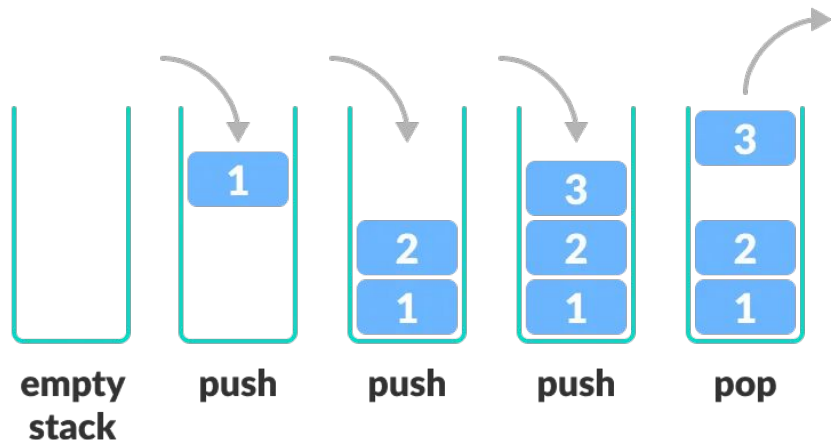
Stack and Queue

Stack

Стек – коллекция, реализованная по принципу LIFO (Last In, First Out)

Стек можно реализовать с помощью однонаправленного связного списка или массива.

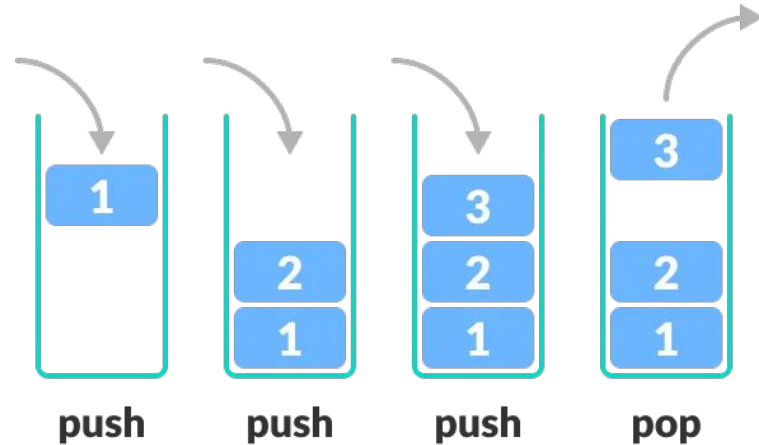
Пространственная сложность: $O(n)$



Stack. Methods and attributes

- **stack.push()** – добавление элемента в стек
- **stack.pop()** – удаление элемента с вершины стека
- **stack.peek()** – возвращает значение элемента на вершине стека
- **stack.size** – количество элементов

Сложность всех операций: $O(1)$



Stack. Implementation

- **Стек легко реализуется с помощью Array:**
 - **Array.prototype.push()** – добавляет элемент в конец массива и возвращает текущую длину
 - **Array.prototype.pop()** – удаляет элемент из конца массива и возвращает его значение
 - **Array.length** – число элементов массива

```
> let stack = []  
  
    stack.push(1)  
    stack  
◀ ▶ [1]  
  
    > stack.push(2)  
    stack  
◀ ▶ (2) [1, 2]  
  
    > stack.push(3)  
    stack  
◀ ▶ (3) [1, 2, 3]
```

```
> stack.pop()  
    stack  
◀ ▶ (2) [1, 2]  
  
    > stack.pop()  
    stack  
◀ ▶ [1]  
  
    > stack.pop()  
    stack  
◀ ▶ []
```

Stack. Use cases

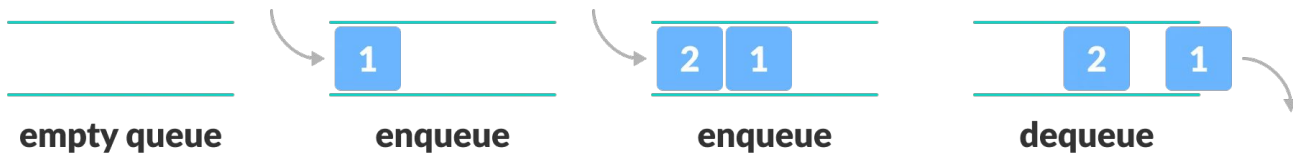
- Реализация операции “отмена” в текстовых редакторах и веб-браузерах
- Обход деревьев и графов
- Синтаксический анализ исходного кода
- Стек вызовов (хранит адрес возврата из подпрограммы)

Queue

Очередь – коллекция, реализованная по принципу FIFO (First In, First Out)

Очередь можно реализовать с помощью однонаправленного связного списка или массива

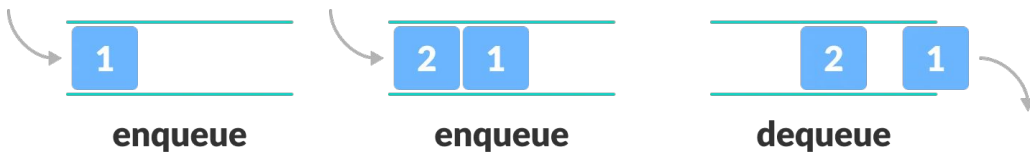
Пространственная сложность: $O(n)$



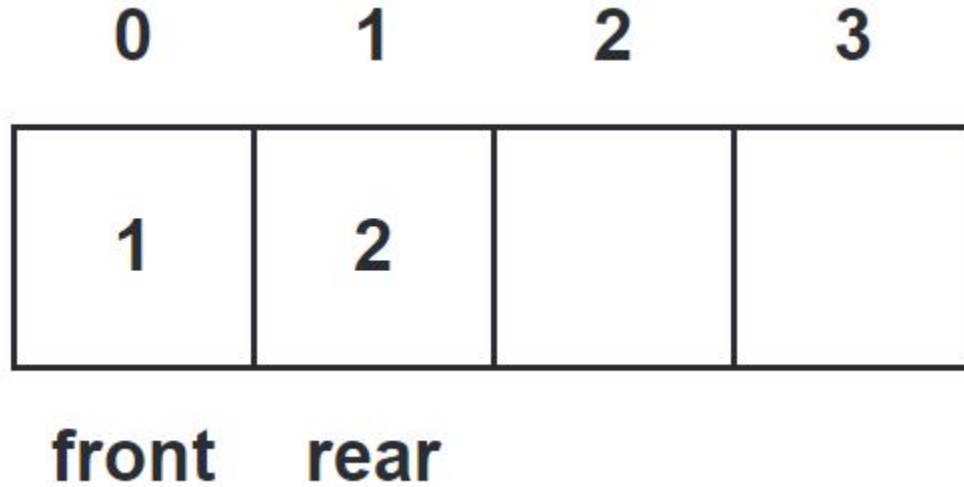
Queue. Methods

- **queue.enqueue()** – добавление элемента в конец очереди
- **queue.dequeue()** – удаление элемента из начала очереди
- **queue.peek()** – возвращает значение элемента из начала очереди
- **queue.size** – количество элементов

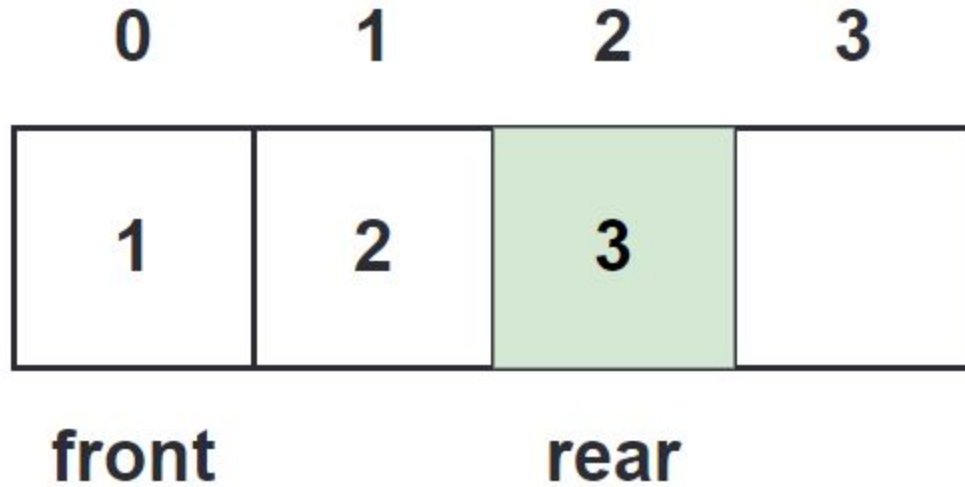
Сложность операций: $O(1)$



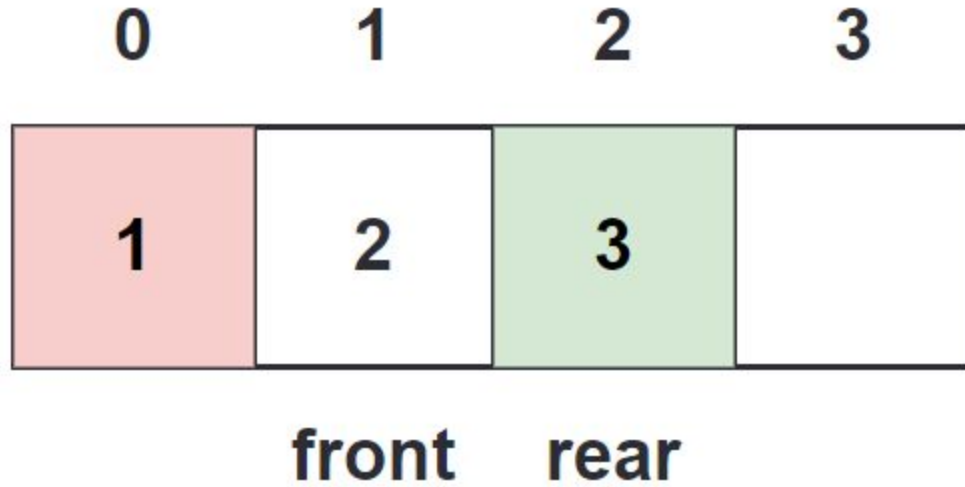
Queue with array



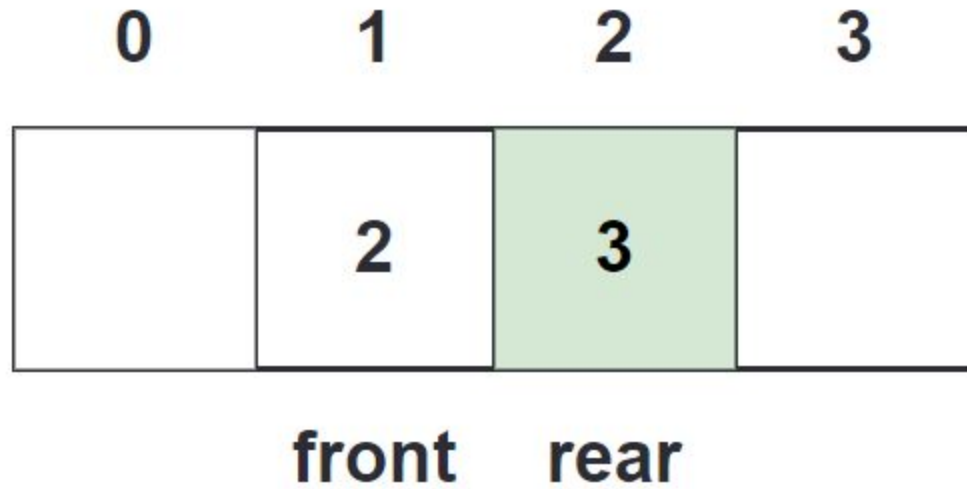
Queue with array



Queue with array



Queue with array



Deque, double-ended queue

Дек – структура данных, в которой добавление и удаление элементов происходит с обоих концов.

Пространственная сложность: $O(n)$.

Временная сложность вставки в начало/конец: $O(1)$.



Queue. Implementation

- Очередь также можно реализовать с помощью Array:
 - **Array.prototype.push()** – добавляет элемент в конец массива и возвращает текущую длину
 - **Array.prototype.shift()** – удаляет элемент из начала массива и возвращает его значение
 - **Array.length** – число элементов массива

```
> let queue = []
```

```
queue.push(1)  
queue
```

```
< ▶ [1]
```

```
> queue.push(2)  
queue
```

```
< ▶ (2) [1, 2]
```

```
> queue.push(3)  
queue
```

```
< ▶ (3) [1, 2, 3]
```

```
> queue.shift()  
queue
```

```
< ▶ (2) [2, 3]
```

```
> queue.shift()  
queue
```

```
< ▶ [3]
```

```
> queue.shift()  
queue
```

```
< ▶ []
```


Rest and destructurization

Rest operator

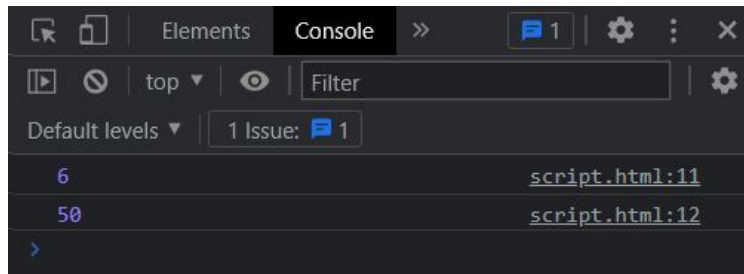
(converting input arguments into an array)

Как реализовать функцию с произвольным числом аргументов?

Помещаем все переданные аргументы в массив



```
function sum(...args) {  
  let sum = 0  
  for(let i=0; i<args.length; i++) {  
    sum += args[i] ?? 0  
  }  
  return sum  
}  
  
console.log( sum(1, 2, 3) )  
console.log( sum(10, 10, 10, 10, 10) )
```



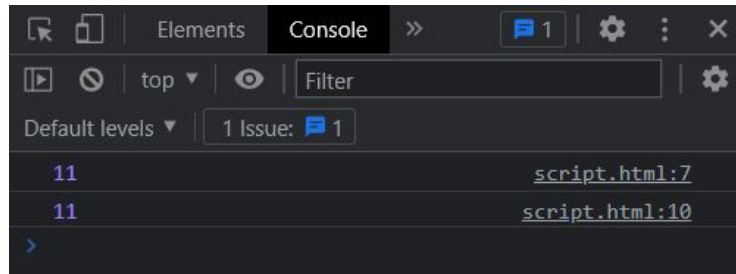
Spread operator

(converting an array into input arguments)

Как передать массив аргументов в функцию?

```
function linear(k, x, b) {  
    return k * x + b  
}  
  
//обычная передача аргументов  
console.log( linear(2, 3, 5) )  
  
//с использованием spread  
let args = [2, 3, 5]  
console.log( linear(...args) )
```

Распаковываем переменные из массива
и передаем в функцию

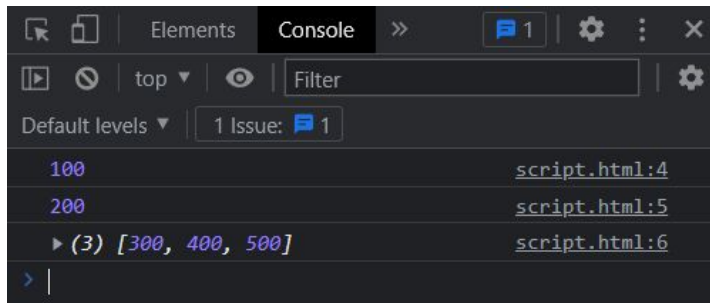
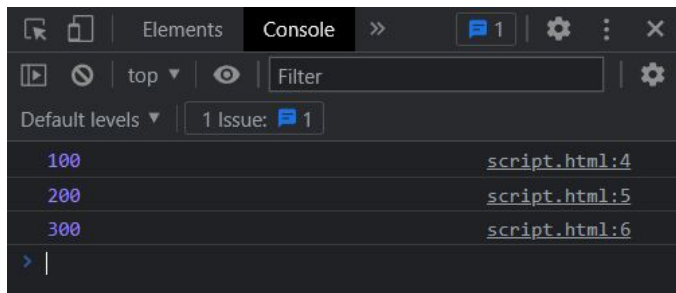


Destructurization

Деструктуризация – синтаксис присваивания, который позволяет разбить массив или объект на части и присвоить нескольким переменным.

```
let [a, b, c] = [100, 200, 300]
console.log(a)
console.log(b)
console.log(c)
```

```
let [a, b, ...other] = [100, 200, 300, 400, 500]
console.log(a)
console.log(b)
console.log(other)
```

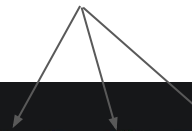


Destructurization

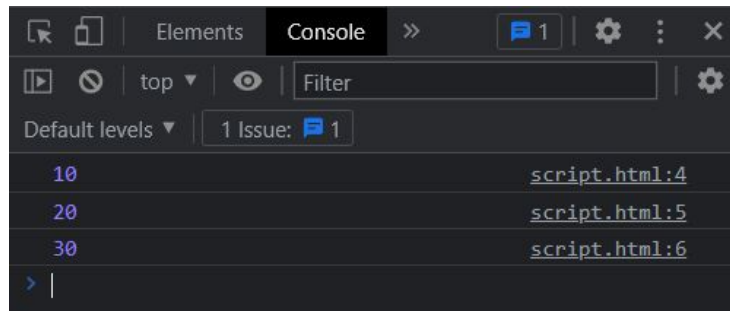
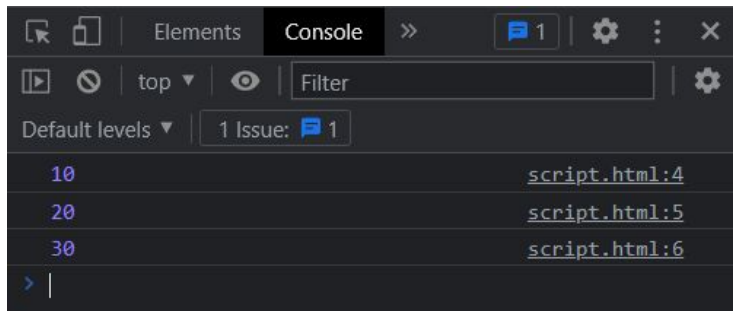
Деструктуризация объектов

```
let {a, b, c} = {a: 10, b: 20, c: 30}  
console.log(a)  
console.log(b)  
console.log(c)
```

(Применяется, когда вы хотите задать переменным, полученным из свойств, новые имена)



```
let {a: n1, b: n2, c: n3} = {a: 10, b: 20, c: 30}  
console.log(n1)  
console.log(n2)  
console.log(n3)
```



Working with js arrays

immutability

Методы *push*, *pop*, *shift*, *unshift*, *sort*, *reverse*, *splice* и некоторые др. **меняют (мутируют)** исходный массив.

Если мы не хотим мутировать исходный массив, то мы можем:

- Заморозить массив (или объект) с помощью метода **Object.freeze**, сделав его иммутабельным.
- Перед мутацией создать копию массива
- Не использовать мутирующие методы

Заморозка

Заморозка
массива

```
> let arr = [1, 2, 3, 4, 5]
Object.freeze(arr)
arr.push(100)
✖ ▶ Uncaught TypeError: Cannot add property 5,
  object is not extensible
    at Array.push (<anonymous>)
    at <anonymous>:5:5
```

Неглубокая копия

```
> let arr = [1, 2, 3, 4, 5]
let arr2 = [...arr]
arr2.push(100)
arr
< ▶ (5) [1, 2, 3, 4, 5]
> arr2
< ▶ (6) [1, 2, 3, 4, 5, 100]
```

Распаковываем все
элементы арг в новый
массив

forEach method

Метод **forEach** принимает в качестве аргумента функцию и запускает её для каждого элемента массива

Переданная
функция будет
запущена для
каждого элемента
массива

текущий
элемент

текущий
индекс

ссылка на сам
массив

```
arr.forEach(function(item, index, array) {  
  // считывать значение можно через item  
  // менять - через array[index]  
});
```

```
> let arr = ["Artem", "Dana", "Mike", "Rachel"];  
arr.forEach(function(item, index, array) {  
  console.log(item)  
});
```

Artem

Dana

Mike

Rachel

map method

Метод **map** вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

```
let resultArr = arr.map(function(item, index, array) {  
  // то, что мы вернём будет вставлено в новый массив на месте текущего элемента  
});
```

(map не меняет исходный массив)

(Получаем массив остатков от деления на 3)

```
> let arr = [17, 12, 15, 16, 14, 19];  
  arr.map(function(item, index, array) {  
    return item % 3  
  });  
↵ ▶ (6) [2, 0, 0, 1, 2, 1]
```

(Получаем массив элементов умноженных на 10)

```
> let arr = [1, 2, 3, 4, 5];  
  arr.map(function(item, index, array) {  
    return item * 10  
  });  
↵ ▶ (5) [10, 20, 30, 40, 50]
```

filter method

Метод **filter** запускает функцию для каждого элемента массива. Все элементы, для которых функция вернула true, помещаются в новый массив и **filter** возвращает его.

```
let filteredArray = arr.filter(function(item, index, array) {  
  // если return true - элемент добавляется к результату, и перебор продолжается  
  // возвращается пустой массив в случае, если ничего не найдено  
});
```

(filter не меняет исходный массив)

```
> let arr = [17, 12, 15, 16, 14, 19];  
arr.filter(function(item, index, array) {  
  if(item % 2 == 0) ←  
    return true  
  else  
    return false  
});  
↵ ▶ (3) [12, 16, 14] ←
```

Хотим выбрать все четные элементы

Новый массив из отобранных элементов

reduce method

Метод **reduce** позволяет вычислить какое-нибудь единое значение на основании всего массива.

Значение аккумулятора после последней итерации возвращается в качестве результата

Переданная функция будет запущена для каждого элемента массива

“Аккумулятор” - переменная, в которой накапливается результат

Начальное значение аккумулятора

```
let result = arr.reduce(function(accumulator, item, index, array) {  
  //в accumulator содержится значение, которое функция  
  //вернула на предыдущей итерации  
}, initialAccumulator);
```

Вычисление суммы элементов массива с помощью reduce

```
> let arr = [1, 2, 3, 4, 5]  
  
arr.reduce(function(accumulator, item, index, array) {  
  return accumulator + item  
}, 0);  
  
◀ 15
```

find and findIndex methods

Метод **findIndex** запускает функцию для каждого элемента массива, чтобы найти определенный элемент. Возвращает индекс найденного элемента (для которого переданная функция вернула true).

Метод **find** аналогичен **findIndex**, только возвращает не индекс найденного элемента, а сам элемент (и undefined, если элемент не найден)

```
let resultIndex = arr.findIndex(function(item, index, array) {  
  // если return true - возвращается индекс найденного элемента и перебор прерывается  
  // если все итерации оказались ложными, возвращается -1  
});
```

```
> let arr = [17, 12, 15, 16, 11];  
   arr.findIndex(function(item, index, array) {  
     if(item % 5 == 0)   
       return true  
     else  
       return false  
   });  
< 2
```

Ищем элемент кратный 5

Индекс найденного
элемента

sort method

Метод **sort** сортирует массив на месте, меняя в нём порядок элементов. Для сравнения двух элементов используется переданная функция.

а и b могут быть любыми объектами

```
arr.sort(function compare(a, b) {  
  if (a > b) return 1; // если первое значение больше второго  
  if (a == b) return 0; // если равны  
  if (a < b) return -1; // если первое значение меньше второго  
})
```

Сравниваем два объекта и возвращаем положительное или отрицательное число в зависимости от того, какой объект мы посчитали БОльшим. Если объекты равны, то возвращается ноль

```
> let arr = [7, 4, 1, 2, 3, 6, 5];  
  arr.sort(function(a, b) {  
    if(a > b) return 1;  
    if(a == b) return 0;  
    if(a < b) return -1;  
  });  
◀ ▶ (7) [1, 2, 3, 4, 5, 6, 7]
```

(sort меняет исходный массив и возвращает его по ссылке)

Template strings

Template Literals

Иногда бывает удобно формировать строки не конкатенацией, а с помощью шаблонов строк.

Нужно использовать наклонные одинарные кавычки и подставлять значения выражений с помощью конструкции `${expression}`

```
> let name = "Adam";  
let age = 29  
  
"Hello, my name is " + name + "! I am " + age + " years old."  
⏏ 'Hello, my name is Adam! I am 29 years old.'
```

← Конкатенация

Шаблоны строк →

```
> let name = "Adam";  
let age = 29;  
  
`Hello, my name is ${name}! I am ${age} years old.`  
⏏ 'Hello, my name is Adam! I am 29 years old.'
```

```
> let value = ["Adam"];  
  
`Hello, my name is ${value[0]}! I am ${25+4} years old.`  
⏏ 'Hello, my name is Adam! I am 29 years old.'
```

← (Подставлять можно и более сложные выражения)

Tagged Template Literals

Перед строковым шаблоном можно указать имя функции, которая будет его обрабатывать.

массив подстрок, между вставными конструкциями \${}

массив подставляемых выражений

```
> function processTemplate(strings, ...params) {  
  return ""  
}  
  
let name = "Adam";  
let age = 29;  
  
processTemplate`Hello, my name is ${name}! I am ${age} years old.`  
⏪ ''
```

Пример функции, обрамляющей переданные параметры в скобки*

```
> function processTemplate(strings, ...params) {  
  return strings[0] + "(" + params[0] + ")" + strings[1] + "(" + params[1] + ")" + strings[2]  
}  
  
let name = "Adam";  
let age = 29;  
  
processTemplate`Hello, my name is ${name}! I am ${age} years old.`  
⏪ 'Hello, my name is (Adam)! I am (29) years old.'
```

*Понятно, что если число параметров в шаблоне не равно двум, то эта функция работать не будет.

Нужно писать универсальную с использованием циклов.

Global methods

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects

Type conversion

С помощью глобальных объектов **Number**, **String**, **Boolean** мы можем выполнять приведение данных к определенному типу.

Приведение к number

```
> Number('115')
< 115

> Number([])
< 0

> Number({})
< NaN
```

Приведение к string

```
> String(115)
< '115'

> String([])
< ''

> String({})
< '[object Object]'
```

Приведение к boolean

```
> Boolean(1)
< true

> Boolean(0)
< false

> Boolean([])
< true

> Boolean('')
< false
```


Object.create method

Метод **Object.create** создает новый объект с указанным прототипом.

```
Object.create(proto)
```

Прототип нашего нового объекта

Создаём пустой
объект с
указанным
прототипом

```
> let protoObj = {name: "Mike", hello() {console.log("Hello!")}}  
  
let myObj;  
myObj = Object.create(protoObj)  
  
< { }  
  [[Prototype]]: Object  
    ▶ hello: f hello()  
      name: "Mike"  
    ▶ [[Prototype]]: Object
```

Object.keys and Object.values methods

Метод **Object.keys** создает массив ключей объекта.

Метод **Object.values** создает массив значений объекта.

В отличие от цикла **for in**, данные методы будут возвращать **только собственные свойства и методы объекта** (те поля, которые он наследует от прототипа, не будут входить в возвращаемые массивы)

`Object.keys(obj)`

`Object.values(obj)`

Ключи объекта

Значения объекта

```
> Object.keys( {name: "Ban", age: 34, hello() {console.log()}} )  
< ▶ (3) ['name', 'age', 'hello']  
  
> Object.values( {name: "Ban", age: 34, hello() {console.log()}} )  
< ▶ (3) ['Ban', 34, f]
```

Object.entries method

Метод **Object.entries** создает вложенный массив пар “ключ-значение” объекта.

Этот метод также возвращает **только свойства экземпляра объекта**, а не унаследованные свойства прототипа.

```
Object.entries(obj)
```

массив пар
“ключ-значение”

```
> Object.entries( {name: "Ban", age: 34, hello() {console.log()}} )  
◀ ▼ (3) [Array(2), Array(2), Array(2)] ⓘ  
  ▶ 0: (2) ['name', 'Ban']  
  ▶ 1: (2) ['age', 34]  
  ▶ 2: (2) ['hello', f]  
    length: 3  
  ▶ [[Prototype]]: Array(0)
```

Object.assign method

Метод **Object.assign** копирует значения из одного объекта в другой.

```
Object.assign(target, ...sources)
```

Исходный объект
(будет возвращен после
модификации)

Объекты, откуда мы
хотим скопировать
свойства и методы в
основной объект

```
> let obj1 = {name: "Adam"}  
   let obj2 = {}  
   Object.assign(obj2, obj1)  
   obj1.name = "Smitto"  
   obj2.name  
< 'Adam'
```

Копируем первый объект
во второй

Так что, если первый объект
изменится, на втором это не
отразится

Object.freeze and Object.isFrozen methods

Метод **Object.freeze** предотвращает модификацию свойств и значений объекта и добавление или удаление свойств объекта.

Метод **Object.isFrozen** позволяет определить, был ли объект заморожен или нет (возвращает true/false)

`Object.freeze(obj)`

`Object.isFrozen(obj)`

Замораживаем
объект

Пытаемся
изменить/удалить/до-
бавить свойство

Но объект остался
неизменным

```
> let myObj = {a: 10, b: 20, c: 30}
Object.freeze(myObj)
< ▶ {a: 10, b: 20, c: 30}

> myObj.a = 100
< 100

> delete myObj.b
< false

> myObj.d = 40
< 40

> myObj
< ▶ {a: 10, b: 20, c: 30}
```

Object.seal and Object.isSealed methods

Метод **Object.seal** предотвращает добавление новых свойств объекта, но позволяет изменять существующие свойства.

Метод **Object.isSealed** позволяет определить, был ли объект запечатан или нет (возвращает true/false)

`Object.seal(obj)`

`Object.isSealed(obj)`

Запечатываем
объект

Пытаемся
изменить/удалить/доб
авить свойство

Сработало только
изменение значения
существующего
свойства

```
> let myObj = {a: 10, b: 20, c: 30}
Object.seal(myObj)
< ▶ {a: 10, b: 20, c: 30}

> myObj.a = 100
< 100

> delete myObj.b
< false

> myObj.d = 40
< 40

> myObj
< ▶ {a: 100, b: 20, c: 30}
```

Array.isArray method

Статический метод **Array.isArray** помогает определить, является ли переданный объект массивом.

```
Array.isArray(obj)
```

Объект для проверки

Для `typeof` массивы и объекты неотличимы (возвращает и для тех и для других `'object'`)

```
> typeof []  
< 'object'  
  
> typeof {}  
< 'object'
```

`Array.isArray` позволяет различать массивы и объекты

```
> Array.isArray([])  
< true  
  
> Array.isArray({})  
< false
```

Array.from and Array.of methods

Метод **Array.from** создает новый экземпляр Array из массивоподобного или итерируемого объекта.

Метод **Array.of** создает новый массива из произвольного числа переданных аргументов

```
> Array.from("Hello!")  
◀ ▶ (6) ['H', 'e', 'l', 'l', 'o', '!']
```

```
> Array.of(1, 2, 3, "Hello")  
◀ ▶ (4) [1, 2, 3, 'Hello']
```


JSON format

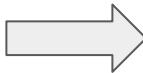
JSON (JavaScript Object Notation) - это популярный формат текстовых данных, который используется для обмена данными в современных приложениях.

Синтаксис JSON очень похож на синтаксис объектов (это по сути он и есть) в JS.

Данные в формате JSON выглядят более наглядно, чем в XML.

XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```



JSON

```
{ "empinfo" :
  {
    "employees" : [
      {
        "name" : "James Kirk",
        "age" : 40,
      },
      {
        "name" : "Jean-Luc Picard"
        "age" : 45,
      },
      {
        "name" : "Wesley Crusher"
        "age" : 27,
      }
    ]
  }
}
```

Working with JSON

Для работы с JSON в JS есть глобальный объект **JSON**

Метод **JSON.stringify** преобразует JS объект в JSON строку.

Метод **JSON.parse** преобразует строку в формате JSON в JS объект.

JS объект -> строка JSON

```
> let obj = {
  name: "Mike",
  age: 33,
  books: [
    {title: "Anastasiya"},
    {title: "War and peace"},
    {title: "Demidovich"}
  ]
}

JSON.stringify(obj)
< ' {"name": "Mike", "age": 33, "books": [{"title": "Anastasiya"}, {"title": "War and peace"}, {"title": "Demidovich"}] } '
```

строка JSON -> JS объект

```
> JSON.parse(' {"name": "Mike", "age": 33, "books": [{"title": "Anastasiya"}, {"title": "War and peace"}, {"title": "Demidovich"}] } ')
< {name: 'Mike', age: 33, books: Array(3)} ⓘ
  age: 33
  books: Array(3)
    ▶ 0: {title: 'Anastasiya'}
    ▶ 1: {title: 'War and peace'}
    ▶ 2: {title: 'Demidovich'}
```

Глубокое копирование объекта с помощью JSON

```
> let obj1 = {a: 1}

let obj2 = JSON.parse(JSON.stringify( obj1 ))
obj2.a = 100

obj1
< ▶ {a: 1}
```

Темы для докладов

1) Генераторы.

<https://javascript.info/generators>

2) Garbage collector.

<https://javascript.info/garbage-collection>