# Функции, функциональное программирование

# Замыкания

— JS использует *лексическую область видимости*

— *Замыкание* - функция, хранящая связь с областью видимости, в которой она создана

— Все функции в JS являются замыканиями

```
let scope = "global";
function checkScope() {
    let scope = "local";

    return {
        getScopeVar()  { return scope; },
        setScopeVar(s) { scope = s; }
    };
}

let obj = checkScope();
obj.getScopeVar();             // ? "local"
obj.setScopeVar("new scope");
obj.getScopeVar();             // ? "new scope"

let obj2 = checkScope();
obj2.getScopeVar();            // ? "local"!
```

# Замыкания: паттерн "модуль"

— Паттерн "модуль" - анонимная немедленно вызываемая функция

— Локальные переменные становятся приватными для модуля

— Публичные методы передаются наружу модуля

```
let basketModule = (function() {
  let basket = []; // приватная переменная

  return {         // публичные методы
    add: function(value) {
      basket.push(value);
    },

    getTotal: function() {
      let sum = 0;
      for (item of basket) {
        sum += item.price;
      };
      return sum;
    }
  };
})();
```

```
basketModule.add(
    { item: "bread", price: 0.5 });

basketModule.add(
    { item: "butter", price: 0.3 });

basketModule.getTotal(); // 0.8

basketModule.basket; // undefined
basket; // ReferenceError
```

# Функциональное программирование

— Функции являются first-class citizen

— Программа не содержит изменяемых переменных (!)

Облегчается тестирование и отладка

Вычисления очень легко сделать многопоточными

Уменьшается число потенциальных ошибок

```
;;; Lisp factorial example

(defun factorial (N)
  "Compute the factorial of N."
  (if (= N 1)
    1
    (* N (factorial (- N 1)))))
```

# Функциональное программирование в JS

— В JS функции - это тип данных/замыкания

— Дополнительно ES5/6: `bind`, `tail call optimization`

— Все остальное из мира ФП можно имитировать вручную

— В JS обычно используются только *отдельные элементы* ФП

# forEach

— `[...].forEach(callback)` - циклический перебор элементов массива

— Параметр: `callback(element, index, array)`

```
const items = ["item1", "item2", "item3"];

for (let i = 0; i < items.length; i++) {
    console.log(items[i], i)
}
```

```
const items = ["item1", "item2", "item3"];

items.forEach(function(item, index) {
    console.log(item, index);
});
```

```
const items = ["item1", "item2", "item3"];
// callback как отдельная именованная функция
const logger = (item, index) => console.log(item, index);

items.forEach(logger);
// или даже!
items.forEach(console.log);
```

# map

— `[...].map(callback)` - создает новый массив путем применения функции к элементам исходного массива

— Параметр: `callback(element, index, array)`

```
let arr = [1, 2, 3, 4],
    result = [];

for (const x of arr) {
    result.push(x * 4);
}
```

```
let arr = [1, 2, 3, 4];

let result = arr.map(x => x * 4);
```

```
[1, 4, 9].map(Math.sqrt); // => [1, 2, 3]

// За что все мы любим JavaScript?
["1", "2", "3"].map(Number.parseInt);
// => ? [1, NaN, NaN] за то, что с ним не соскучишься!

function parseItRight(item) {
    return Number.parseInt(item, 10);
}
["1", "2", "3"].map(parseItRight); // => [1, 2, 3]
```

# filter

— `[...].filter(callback)` - создает новый массив, в котором содержатся только те элементы исходного массива, для которых выполняется предикат

```
let arr = [4, 19, 8, -3, 5, 6],
    result = [];

for (const x of arr) {
    if (!(x % 2)) {
        result.push(v);
    }
}
```

```
let arr = [4, 19, 8, -3, 5, 6];

let result = arr.filter(x => !(x % 2));
```

# some

— `[...].some(callback)` - проверяет выполняется ли предикат хотя бы для одного элемента массива

— "Ленивое поведение" - вернет `true` после первого же truthy результата

— В некотором смысле `some` - аналог оператора `||`

```
// Есть ли в массиве хотя бы один
// отрицательный элемент?
let arr = [4, 19, 8, -3, 5, 6],
    result = false;

for (const x of arr) {
    if (x < 0) {
        result = true;
        break;
    }
};

result; // -> true
```

```
// Есть ли в массиве хотя бы один
// отрицательный элемент?
let arr = [4, 19, 8, -3, 5, 6];

const result = arr.some(x => x < 0); // -> true
```

# every

— `[...].every(callback)` - проверяет выполняется ли предикат для всех элементов массива

— "Ленивое поведение" - вернет `false` после первого же falsy результата

— В некотором смысле `every` - аналог оператора `&&`

```
// Все ли элементы массива положительны?
let arr = [4, 19, 8, -3, 5, 6],
    result = true;

for (const x of arr) {
    if (x < 0) {
        result = false;
        break;
    }
};

result; // -> false
```

```
// Все ли элементы массива положительны?
let arr = [4, 19, 8, -3, 5, 6];

const result = arr.every(x => x > 0);
// -> false
```

# reduce

— Найдите два отличия на картинке

```
function sumOfArray(arr) {
    let sum = 0;
    for (const item of arr) {
        sum = sum + item;
    }
    return sum;
}
sumOfArray([1, 2, 3, 4]); // 10
```

```
function maxOfArray(arr) {
    let max = -Infinity;
    for (const item of arr) {
        max = (max > item) ? max : item;
    }
    return max;
}
maxOfArray([1, 2, 3, 27, 4]); // 27
```

```
function reduce(arr) {
    let accumulator = [initValue];
    for (const item of arr) {
        accumulator = [Do something with item and accumulator]
    }
    return accumulator;
}
```

# reduce

— Мы можем параметризовать алгоритм начальным значением `accumulator` и функцией `callback`

— `reduce` получает значение в результате прохода по массиву

```
function reduce(arr, callback, initAccumulator) {
    let acc = initAccumulator;
    for (const item of arr) {
        acc = callback(acc, item);
    }
    return acc;
}

reduce([1, 2, 3, 4], (acc, x) => acc + x, 0); // -> 10
reduce(
    [1, 2, 3, 27, 4],
    (acc, x) => Math.max(acc, x), // на каждой итерации возвращаем бóльшее значение
    -Infinity
); // -> 27

reduce([1, 2, 27, 15, 30, 0], Math.max, -Infinity); // -> 30, или даже так!
```

# reduce

— `[...].reduce(callback[, initAccumulator]);`

— `callback(accumulator, item, index, array)`

— `[...].reduceRight` проходит по массиву в обратном направлении

— Если `initAccumulator` не задан, то первый вызов `callback` будет со значениями первых двух элементов массива

```
[1, 2, 3, 4].reduce( (sum, item) => sum + item ); // => 10

let arr = [false, true, false, 15, undefined];
arr.reduce((acc, item) => acc || item); // => true
arr.reduceRight((acc, item) => acc || item); // => 15

[[1, 2, 3], [4, 5, 6], [7, 8, 9]].reduce(
    (acc, arr) => [...acc, ...arr]
); // => [1, 2, 3, 4, 5, 6, 7, 8, 9], конкатенация массивов
```

# Декларативное программирование

— Императивный код описывает действия которые нужно произвести, отвечая на вопрос "КАК"

— Декларативный код описывает "ЧТО" нужно получить, не регламентируя последовательность действий

```
let arr = [1, 2, 3],
    result = [];
for (const x in arr) {
    result.push(x + 1);
}
```

```
let arr = [1, 2, 3],
    result = arr.map(x => x + 1);
```

```
const btn = document.createElement('button');
btn.onclick = function(event) {
 if (this.classList.contains("red")) {
   this.classList.remove("red");
   this.classList.add("blue");
 } else {
   this.classList.remove("blue");
   this.classList.add("red");
 }
};
```

```
<Btn
  onClick={this.toggleButton}
  className={this.state.buttonStyle}>
</Btn>
```

# Чистые функции

— Чистая (pure) функция:

— при одинаковых параметрах всегда дает один и тот же результат

— не имеет других наблюдаемых побочных эффектов

```
Math.random(); // 0.1869512743664441
Math.random(); // 0.3383707346070188
/*----------------------------------*/
let counter = 0,
    count = () => counter++;
count();   // -> 0
count();   // -> 1
/*----------------------------------*/
let multiplier = 2,
    mul2 = x => x * multiplier;
mul2(2);   // -> 4
mul2(2);   // -> 4
multiplier = 3; // как вдруг!
mul2(2);   // -> 6
/*----------------------------------*/
function save() {
    window.fetch("post.html");
}
```

```
Math.max(2, 4); // -> 4
Math.max(2, 4); // -> 4

/*----------------------------------*/

let mul2 = function(x) {
    const multiplier = 2;
    return x * multiplier;
}
mul2(2);   // -> 4
mul2(2);   // -> 4

/*----------------------------------*/

let arr = [1, 2, 3, 4];
    arr.map(x => x + 1);
arr;       // -> [1, 2, 3, 4]
```

# Функции высшего порядка

— Принимают одну или несколько функций в качестве аргументов

— Возвращают новую функцию в результате выполнения

```
const not = function(f) {
    return function(...args) {
        return !f(...args);
    };
}

const even = n => n % 2 === 0,
    odd = not(even);

[1, 1, 5, 3, 9].every(odd); // -> true
```

# Функции высшего порядка

```
const expenses = [
  {name: "Rent",    price: 500,  type: "Household"},
  {name: "Netflix", price: 5.99, type: "Services"},
  {name: "Gym",     price: 15,   type: "Health"},
  {name: "Bills",   price: 100,  type: "Household"}
];

// Подсчет общих затрат
const getSum = (exp) =>
    exp.reduce(
        (sum, item) => sum + item.price, 0);
getSum(expenses); // 620.99

// Получить только Household записи
const getHousehold = (exp) =>
    exp.filter(
        item => item.type === "Household");
getHousehold(expenses);
// -> [{name: "Rent" ...}, {name: "Bills" ...}]

// Композиция функций
getSum(getHousehold(expences)); // => 600
```

```
const compose = (f, g) => {
    return (x) => {
        return g(f(x));
    }
};
// кратко: compose = (f, g) => (x) => g(f(x));
const odd = compose(x => x % 2 === 0, x => !x);

// получить сумму всех Household
const getHouseholdSum = compose(
    getHousehold, getSum);
getHouseholdSum(expences);  // => 600

const getCategories = (exp) =>
    exp.map(item => item.type);
getCategories(expences);
// ["Household", "Services", "Health", "Household"]

const getUnique = list => [...new Set(list)];
const getUniqueCategories = compose(
    getCategories, getUnique);
getUniqueCategories(expences);
// => ["Household", "Services", "Health"]
```

# Частичное применение функции

— Частичное применение - создание новой функции, с предустановленным значением одного или нескольких аргументов

```
const greet = (greet, name) => `${greet}, ${name}!`;
greet("Hello", "world");  // -> "Hello, world!"

// Частичное применение
const greetWithHello = greet.bind(null, "Hello");
greetWithHello("world");  // => "Hello, world!"
greetWithHello("moto");   // => "Hello, moto!"
```