

**Функции, замыкания, функциональное программирование**

# Замыкания

- JS использует лексическую область видимости
- При вызове функции действует scope, который имелся на момент ее создания, а не вызова
- Замыкание - это функция, хранящая ссылку на область видимости, в которой она создана
- Все функции в JS являются замыканиями

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

    function inner() {
        return scope;
    }
    return inner();
}
checkScope(); // => ? "local"
```

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

    function inner() {
        return scope;
    }
    return inner;
}
checkScope()(); // => ? "local"
```

## Замыкания: инкапсуляция данных

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

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

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

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

## Замыкания: паттерн "модуль"

- Паттерн "модуль" - анонимная немедленно вызываемая функция
- Локальные переменные становятся private переменными модуля
- Публичные методы передаются наружу и работают с данными в замыкании

```
let basketModule = (function() {  
  var basket = []; // приватная переменная  
  return { // методы доступные извне  
    add: function(values) {  
      basket.push(values);  
    },  
    getCount: function() {  
      return basket.length;  
    },  
    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.getCount(); // 2  
basketModule.getTotal(); // 0.8  
  
basketModule.basket; // undefined  
basket; // ReferenceError
```

# Замыкания: (исчезающе) тонкие моменты

```
function arrayOfFunctions() {
  let result = [], i;

  for (i = 0; i < 10; i++) {
    result.push(
      function() { return i; }
    );
  }
  return result; // Массив из 10 функций
}
arrayOfFunctions()[5](); // => ? 10
```

---

```
function arrayOfFunctions() {
  let result = [], i
  for (i = 0; i < 10; i++) {
    result.push(
      function(x){return x;}.bind(null,i)
    );
  }
  return result;
}
arrayOfFunctions()[5](); // 5
```

```
function arrayOfFunctions() {
  let result = [], i;
  for (i = 0; i < 10; i++) {
    result.push(
      (function (x) {
        return function() { return x; };
      })(i)
    );
  }
  return result;
}
arrayOfFunctions()[5](); // 5
```

```
function arrayOfFunctions() {
  let result = [];
  for (let i = 0; i < 10; i++) {
    result.push(
      function() { return i; }
    );
  }
  return result;
}
arrayOfFunctions()[5](); // 5
```

## Функциональное программирование

- Программа не содержит изменяемых переменных (!)
  - Облегчается тестирование и отладка
  - Вычисления очень легко сделать многопоточными
  - Уменьшается число потенциальных ошибок
- Функции являются first-class citizen

# Функциональное программирование

- JS выполняет главное условие: функции - это тип данных/замыкания
- Дополнительные возможности в ES5/6: `bind`, `tail call optimization`
- Все остальное можно имитировать/"завелосипедить"







# reduce

- Что общего у двух кусков кода?

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

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

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

# reduce

- Мы можем параметризовать алгоритм, добавив два параметра: начальное значение `accumulator` и функцию `callback`
- `reduce` - алгоритм получения значения в результате прохода по массиву

```
function reduce(arr, callback, initAccumulator) {  
  let acc = initAccumulator, item;  
  for (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 большее значение  
  -Infinity  
); // => 27
```

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

## Array.prototype.reduce

- `Array.prototype.reduce(callback[, initAccumulator]);`
- `callback(accumulator, item, index, array)`
- 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
```

# Array.prototype.map

- `map` - создает новый массив, элементы которого получены в результате применения функции в каждом элементу исходного массива
- `Array.prototype.map(callback)`
- `callback(currentValue, index, array)`

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

```
arr.map(x => x * 4); // => [4, 8, 12, 16]
```

```
arr.reduce((acc, item) => acc.push(item * 4), []); // map через reduce
```

```
[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]
```

## Array.prototype.filter/every/some

- `Array.prototype.filter(callback)`- создать новый массив, элементы которого отвечают условию
- `Array.prototype.some(callback)`- аналог `||`
- `Array.prototype.every(callback)`- аналог `&&`
- `callback(value, index, array)`
- `some` выполняется до первого truthy value, `every` - до первого falsy

```
[4, 19, 8, -3, 5, 6]
  .filter(x => !(x % 2)) // отбросить все нечетные числа
  .some(x => x > 7);     // проверить есть ли среди оставшихся числа больше 7
```

```
[4, 19, 8, -3, 5, 6].reduce(function(acc, x) {
  return x % 2 ? acc : [...acc, x];
}, []); // filter через reduce
```

## Функции высшего порядка

- принимают одну или несколько функций в качестве аргументов
- возвращают новую функцию в результате своего выполнения

```
let not = function(f) {  
  return function(...args) {  
    return !f(...args);  
  };  
}  
let even = n => n % 2 === 0;  
let odd = not(even);  
  
[1, 1, 3, 5, 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));
  }
};
// или коротко: 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 и каррирование

- Частичное применение - создание новой функции, с предустановленным значением одного или нескольких аргументов
- Каррирование - преобразование функции с несколькими параметрами в набор функций с одним параметром

```
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!'
```

```
// Каррирование
const curry2 = f => // для функции двух аргументов
  x =>
    y => f(x, y);
```

```
const curriedGreet = curry2(greet);
curriedGreet('Hello')('world'); // => 'Hello, world!'
```

```
const greetWithHello2 = curriedGreet('Hello');
greetWithHello2('world'); // => 'Hello, world!'
greetWithHello2('moto'); // => 'Hello, moto!'
```



# Каррирование

```
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'}  
];
```

```
const curry2 = f => x => y => f(x, y);  
const compose = (f, g) => x => g(f(x));
```

```
const regularPluck = (propName, arr) => arr.map(x => x[propName]);  
const pluck = curry2(regularPluck);  
const sum = (arr) => arr.reduce((a, b) => a + b);
```

```
regularPluck('name', expenses); // ['Rent', 'Netflix', 'Gym', 'Bills']  
pluck('price')(expenses); // => [500, 5.99, 15, 100]
```

```
const sumPrices = compose(pluck('price'), sum);  
const getCategories = pluck('type');
```

```
sumPrices(expenses); // 620.99  
getCategories(expenses); // ["Household", "Services", "Health", "Household"]
```

## То есть как, совсем нет переменных?

- Состояние хранится в стеке вызовов и замыканиях

```
const reverse = function reverse(arr) {  
  return arr.length === 1  
    ? arr  
    : [...reverse(arr.slice(1)), arr[0]]  
};  
reverse([1, 2, 3]); // => [3, 2, 1]
```

```
const forEach = function forEach(arr, callback, n = 0) {  
  if (n === arr.length) return;  
  callback(arr[n]);  
  return forEach(arr, callback, n + 1)  
}  
forEach([1, 2, 3], console.log); // 1, 2, 3
```

```
const f = function f(n) {  
  return n === 0  
    ? 1  
    : n * f(n - 1);  
};  
f(5); // => 120  
f(1000); // RuntimeError
```

## Оптимизация хвостовой рекурсии

- Хвостовая рекурсия - рекурсия, в которой рекурсивный вызов является последней операцией перед возвратом из текущей функции
- Хвостовые вызовы могут быть оптимизированы так, что не будут использовать стек
- tail call optimization включена в экспериментальных режимах движков JS

```
const factorial1 = function factorial(n) {  
  return n === 0 ? 1 : n * factorial(n - 1);  
  // результат рекурсивного вызова умножается на n, нельзя оптимизировать  
}
```

```
const factorial2 = function factorial(n, acc = 1) {  
  return n === 0 ? acc : factorial(n - 1, n * acc);  
  // результат рекурсивного вызова сразу же возвращается из функции: это tail call  
}
```

```
factorial1(50000); // RangeError  
factorial2(50000); // Infinity
```