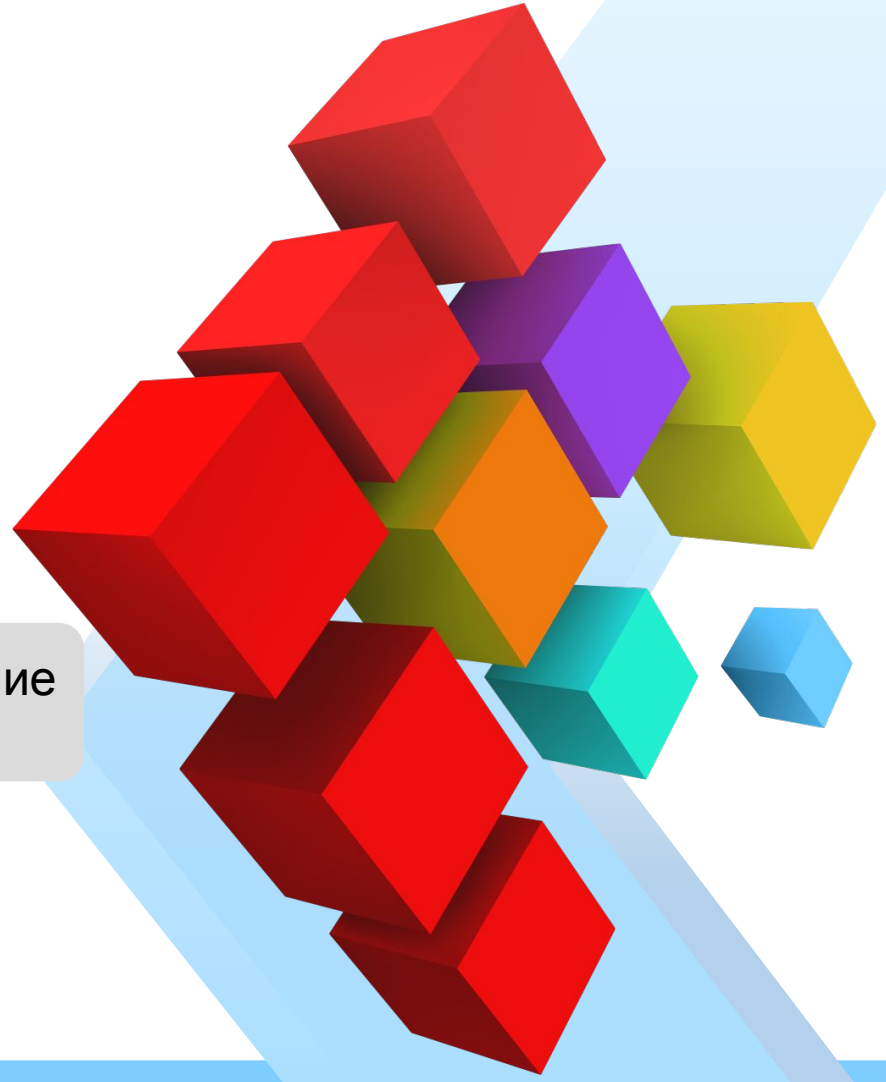
СИНЕРГИЯ

Основы программирования на языке **JavaScript** **2** часть

"Верстка CSS и программирование на языке JavaScript"



Колобова Луиза Владимировна

СИНЕРГИЯ

1

ООП и Объекты

Основы ООП

Объектно-ориентированное программирование на сегодняшний день является одной из господствующих парадигм в разработке приложений, и в JavaScript мы также можем использовать все преимущества ООП. В то же время применительно к JavaScript объектно-ориентированное программирование имеет некоторые особенности.

Объекты

В прошлых темах мы работали с примитивными данными - числами, строками, но данные не всегда представляют примитивные типы. Например, если в нашей программе нам надо описать сущность человека, у которого есть имя, возраст, пол и так далее, то естественно мы не сможем представить сущность человека в виде числа или строки. Нам потребуется несколько строк или чисел, чтобы должным образом описать человека. В этом плане человек будет выступать как сложная комплексная структура, у которого будут отдельные свойства - возраст, рост, имя, фамилия и т.д.

Для работы с подобными структурами в JavaScript используются объекты. Каждый объект может хранить свойства, которые описывают его состояние, и методы, которые описывают его поведение.

В примерах мы будем использовать **console.log()**

Создание нового объекта

Есть несколько способов создания нового объекта.
Первый способ заключается в использовании конструктора Object:

```
let user = new Object();
```

В данном случае объект называется user.
Выражение new Object() представляет вызов конструктора - функции, создающей новый объект. Для вызова конструктора применяется оператор new. Вызов конструктора фактически напоминает вызов обычной функции.
Второй способ создания объекта представляет использование фигурных скобок:

```
let user = {};
```

На сегодняшний день более распространенным является второй способ.

Свойства объекта

После создания объекта мы можем определить в нем свойства. Чтобы определить свойство, надо после названия объекта через точку указать имя свойства и присвоить ему значение:

```
let user = {};
user.name = "Tom";
user.age = 26;
```

В данном случае объявляются два свойства name и age, которым присваиваются соответствующие значения. После этого мы можем использовать эти свойства, например, вывести их значения в консоли:

```
console.log(user.name);
console.log(user.age);
```

Также можно определить свойства при определении объекта:
В этом случае для присвоения значения свойству используется символ двоеточия, а после определения свойства ставится запятая (а не точка с запятой).

```
let user = {
name: "Tom",
age: 26
};
```

Свойства объекта

Кроме того, доступен сокращенный способ определения свойств:

```
let name = "Tom";
let age = 34;
let user = {name, age};
console.log(user.name);     // Tom
console.log(user.age);      // 34
```

В данном случае названия переменных также являются и названиями свойств объекта. И таким образом можно создавать более сложные конструкции:

```
let name = "Tom";
let age = 34;
let user = {name, age};

let teacher = {user, course: "JavaScript"};
console.log(teacher.user);      // {name: "Tom", age: 34}
console.log(teacher.course);    // JavaScript
```

Методы объекта

Методы объекта определяют его поведение или действия, которые он производит. Методы представляют собой функции. Например, определим метод, который бы выводил имя и возраст человека:

```
let user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){

    console.log(user.name);
    console.log(user.age);
};

// вызов метода
user.display();
```

Как и в случае с функциями методы сначала определяются, а потом уже вызываются.

Методы объекта

Также методы могут определяться непосредственно при определении объекта:

```
let user = {
    name: "Tom",
    age: 26,
    display: function(){

        console.log(this.name);
        console.log(this.age);
    }
};
```

Как и в случае со свойствами, методу присваивается ссылка на функцию с помощью знака двоеточия.

Метод this

Чтобы обратиться к свойствам или методам объекта внутри этого объекта, используется ключевое слово this. Оно означает ссылку на текущий объект.
Также можно использовать сокращенный способ определения методов

```
let user = {

    name: "Tom",
    age: 26,
    display(){

        console.log(this.name, this.age);
    },
    move(place){
        console.log(this.name, "goes to", place);
    }
};
user.display(); // Tom 26
user.move("the shop");  // Tom goes to the shop
```

Синтаксис массивов

Существует также альтернативный способ определения свойств и методов с помощью синтаксиса массивов:

```
let user = {};
user["name"] = "Tom";
user["age"] = 26;
user["display"] = function(){

    console.log(user.name);
    console.log(user.age);
};

// вызов метода
user["display"]();
```

Название каждого свойства или метода заключается в кавычки и в квадратные скобки, затем им также присваивается значение. Например, user["age"] = 26.

Синтаксис массивов

При обращении к этим свойствам и методам можно использовать либо нотацию точки (user.name), либо обращаться так: user["name"]
Также можно определить свойства и методы через синтаксис массивов напрямую при создании объекта:

```
let user = {
    ["name"]: "Tom",
    ["age"]: 26,
    ["display"]: function(){

        console.log(user.name);
        console.log(user.age);
    }
};
user["display"]();
```

Строки в качестве свойств и методов

Также следует отметить, что названия свойств и методов объекта всегда представляют строки. То есть мы могли предыдущее определение объекта переписать так:

```
let user = {
    "name": "Tom",
    "age": 26,
    "display": function(){

        console.log(user.name);
        console.log(user.age);
    }
};
// вызов метода
user.display();
```

Строки в качестве свойств и методов

С одной стороны, разницы никакой нет между двумя определениями. С другой стороны, бывают случаи, где заключение названия в строку могут помочь. Например, если название свойства состоит из двух слов, разделенных пробелом:

```
let user = {
    name: "Tom",
    age: 26,
    "full name": "Tom Johns",
    "display info": function(){

        console.log(user.name);
        console.log(user.age);
    }
};

console.log(user["full name"]);
user["display info"]();
```

Только в этом случае для обращения к подобным свойствам и методам мы должны использовать синтаксис массивов.

Динамическое определение имен свойств и методов

Синтаксис массивов открывает нам другую возможность - определение имени свойства вне объекта:

```
const prop1  = "name";
const prop2  = "age";
let tom = {
    [prop1]: "Tom",
    [prop2]: 37
};
console.log(tom);           // {name: "Tom", age: 37}
console.log(tom.name);      // Tom
console.log(tom["age"]);    // 37
```

Динамическое определение имен свойств и методов

Благодаря этому, например, можно динамически создавать объекты с произвольными названиями свойств:

```
function createObject(propName, propValue){
    return {
            [propName]: propValue,
            print(){
                console.log(`${propName}: ${propValue}`);
            }
    };
}
let person = createObject("name", "Tom");
person.print();     // name: Tom

let book = createObject("title", "JavaScript Reference");
book.print();   // title: JavaScript Reference
```

Удаление свойств

Выше мы посмотрели, как можно динамически добавлять новые свойства к объекту. Однако также мы можем удалять свойства и методы с помощью оператора delete. И как и в случае с добавлением мы можем удалять свойства двумя способами. Первый способ - использование нотации точки: *delete объект.свойство*. Либо использовать синтаксис массивов:

```
let user = {};
user.name = "Tom";
user.age = 26;
user.display = function(){

    console.log(user.name);
    console.log(user.age);
};
console.log(user.name); // Tom
delete user.name; // удаляем свойство // альтернативный вариант // delete user["name"];
console.log(user.name); // undefined
```

После удаления свойство будет не определено, поэтому при попытке обращения к нему,

Константные объекты

Возможно, нам потребуется, чтобы объект нельзя было изменить, то есть сделать константным. Однако просто определить его как обычную константу с помощью оператора const недостаточно. Например:

```
const person = {name: "Tom", age: 37};
person.name= "Bob";
console.log(person.name);   // Bob
```

Здесь мы видим, что свойство объекта изменило свое значение, хотя объект определен как константа.
Оператор const лишь влияет на то, что мы не можем присвоить константе новое значение, например, как в следующем случае:

```
const person = {name: "Tom", age: 37};
person = {name: "Sam", age: 56};    // Ошибка - нельзя константе присвоить значение второй раз
```

Константные объекты

Тем не менее значения свойств объекта мы можем изменять.
Чтобы сделать объект действительно константным, необходимо применить специальный метод Object.freeze(). В этот метод в качестве параметра передается объект, который надо сделать константным:

```
const person = {name: "Tom", age: 37};
Object.freeze(person);
person.name= "Bob";
console.log(person.name);   // Tom - значение свойства не
изменилось
```

Создание объекта из переменных и констант
При создании объекта его свойствам могут передаваться значения переменных, констант или динамически вычисляемые результаты функций:

```
function getSalary(status){
    if(status==="senior") return 1500;
    else return 500;
}
const name = "Tom";
const age = 37;
const person = { name: name, age: age, salary: getSalary()};

console.log(person);    // {name: "Tom", age: 37, salary:
500}
```

Константные объекты

Но если названия констант/переменных совпадает с названиями свойств, то можно сократить передачу значений:

```
const name = "Tom";
const age = 37;
const salary = 500;
const person = { name, age, salary};
 console.log(person); // {name: "Tom", age: 37, salary: 500}
```

В данном случае объект person автоматически получит свойства, названия которых будут соответствовать названиям констант, а в качестве значений иметь значения этих констант.

Константные объекты

То же самое относится к передаче функций методам объекта:

```
function display(){
    console.log(this.name, this.age);
}
const move = function(place){ console.log(this.name, "goes
to", place)};
const name = "Tom";
const age = 37;
const salary = 500;
const person = { name, age, salary, display, move};
person.display();         // Tom 37
person.move("cinema");  // Tom goes to cinema
```

Константные объекты

То же самое относится к передаче функций методам объекта:

```
function display(){
    console.log(this.name, this.age);
}
const move = function(place){ console.log(this.name, "goes
to", place)};
const name = "Tom";
const age = 37;
const salary = 500;
const person = { name, age, salary, display, move};
person.display();       // Tom 37
person.move("cinema");  // Tom goes to cinema
```

В данном случае объект person имеет два метода, которые соответствуют переданным в объект функциям - display() и move(). Стоит отметить, что при такой передаче функций методам объекта, мы по-прежнему можем использовать в этих функциях ключевое слово this для обращения к функциональности объекта. Однако стоит быть осторожным при передаче лямбд-выражений, поскольку для глобальных лямбд-выражений this будет представлять объект окна браузера:

```
const move = (place)=>{ console.log(this.name, "goes to",
place); console.log(this);};
const name = "Tom";
const person = { name, move};
person.move("cinema");  //  goes to cinema
```

Определение класса

Для определения класса используется ключевое слово class:

```
class Person{ };
```

После слова class идет название класса (в данном случае класс называется Person), и затем в фигурных скобках определяется тело класса.
Это наиболее распространённый способ определения класса. Но есть и другие способы. Так, также можно определить анонимный класс и присвоить его переменной или константе:

```
const Person = class{};
```

В принципе мы можем создать и не анонимный класс и присвоить его переменной или константе:

```
const User = class Person{};
```

Создание объектов

Класс – это общее представление некоторых сущностей или объектов. Конкретным воплощением этого представления, класса является объект. И после определения класса мы можем создать объекты класса с помощью конструктора:

```
class Person{};
const tom = new Person();
const bob = new Person();
```

Для создания объекта с помощью конструктора сначала ставится ключевое слово new. Затем, собственно, идет вызов конструктора - по сути вызов функции по имени класса. По умолчанию классы имеют один конструктор без параметров. Поэтому в данном случае при вызове конструктора в него не передается никаких аргументов.

Создание объектов

Стоит отметить, что в отличие от функций, чтобы использовать класс, его надо сначала определить. Например, в следующем коде мы получим ошибку, так как пытаемся использовать класс до его определения:

```
const tom = new Person();   // ! Ошибка - Uncaught ReferenceError: Cannot access 'Person' before initialization
class Person{};
```

Если определение класса присвоено переменной или константе, то мы можем использовать имя этой переменной/константы для создания объектов класса:

```
const User = class Person{}
const tom = new User();
console.log(tom);
```

Выше в коде несмотря на то, что мы используем вызов new User(), в реальности создаваемый объект будет представлять класс Person.

Пример

Пример создания объекта анонимного класса:

```
const Person = class{}
const tom = new Person();
console.log(tom);
```

Поля и свойства класса
Для хранения данных или состояния объекта в классе используются поля и свойства.
Итак, выше был определен класс Person, который представлял человека. У человека есть отличительные признаки, например, имя и возраст. Определим в классе Person поля для хранения этих данных.

Определение поля фактически просто представляет его название:здесь определено поле name для хранения имени человека, и поле age для хранения возраста человека.

```
class Person{
    name;
    age;
}
const tom = new Person();
tom.name = "Tom";
tom.age = 37;
console.log(tom.name);  // Tom
console.log(tom.age);   // 37
```

Пример

После создания объекта класса мы можем обратиться к этим полям. Для этого после имени объекта через точку указывается имя поля:

```
tom.name = "Tom";       // установим значение поля
console.log(tom.name);  // получим значение свойства
```

В примере выше поля класса также можно назвать свойствами. По сути свойства представляют доступные извне или публичные поля класса. Дальше мы подробно разберем, когда поля бывают непубличные, то есть недоступными извне. Но пока стоит понять, что свойства и публичные поля – это одно и то же. И в примере выше поля name и age также можно назвать свойствами.

При необходимости мы можем присвоить полям некоторые начальные значения:

```
class Person{
    name = "Unknown";
    age= 18;
}
const tom = new Person();
console.log(tom.name);  // Unknown
tom.name = "Tom";
console.log(tom.name);  // Tom
```

Обращение к полям и методам внутри класса. Слово this

Что если мы хотим в методах класса обратиться к полям класса или к другим его методам? В этом случае перед именем поля/свойства или метода указывается ключевое слово this, которое в данном случае указывает на текущий объект.
Например, определим метод, который выводит информацию об объекте:

```
class Person{
    name;
    age;
    print(){
        console.log(`Name: ${this.name}  Age: ${this.age}`);
    }
}
const tom = new Person();
tom.name = "Tom";
tom.age = 37;
tom.print();    // Name: Tom  Age: 37

const bob = new Person();
bob.name = "Bob";
bob.age = 41;
bob.print();    // Name: Bob  Age: 41
```

Определение конструктора

Для создания объекта класса используется конструктор:

```
const bob = new Person();
```

Вызов конструктора по умолчанию, который есть в классах, фактически представляет вызов метода, который имеет то же название, что и класс, и возвращает объект этого класса.
Но также мы можем определить в классах свои конструкторы:

```
class Person{
    name;
    age;
    constructor(){
        console.log("Вызов конструктора");
    }
    print(){
        console.log(`Name: ${this.name}  Age: ${this.age}`);
    }
}
const tom = new Person();   // Вызов конструктора
const bob = new Person();   // Вызов конструктора
```

Определение конструктора

Конструктор определяется с помощью метода с именем constructor. По сути это обычный метод, который может принимать параметры. В данном случае конструктор просто выводит на консоль некоторое сообщение. Соответственно при выполнении строки

```
const tom = new Person();
```

Мы увидим в консоли браузера соответствующее сообщение.
Как правило, цель конструктора - инициализация объекта некоторыми начальными данными:

```
class Person{
    name;
    age;
    constructor(personName, personAge){
        this.name = personName;
        this.age = personAge;
    }
    print(){
        console.log(`Name: ${this.name}  Age: ${this.age}`);
    }
}
const tom = new Person("Tom", 37);
tom.print();    // Name: Tom  Age: 37
const bob = new Person("Bob", 41);
bob.print()     // Name: Bob  Age: 41
```

Здесь конструктор принимает два параметра и передает их значения полям класса.

Определение конструктора

Соответственно при создании объекта мы можем передать в конструктор соответствующие значения для этих параметров:

```
const tom = new Person("Tom", 37);
```

Стоит отметить, что в примере выше определения полей класса избыточно. Обращение в конструкторе к полям через this фактически будет аналогично их определению, и в данном случае мы можем убрать определение полей:

```
class Person{

    constructor(personName, personAge){
        this.name = personName;
        this.age = personAge;
    }
    print(){
        console.log(`Name: ${this.name}  Age: ${this.age}`);
    }
}
const tom = new Person("Tom", 37);
tom.print();    // Name: Tom  Age: 37
const bob = new Person("Bob", 41);
bob.print()     // Name: Bob  Age: 41
```

Принцип KISS and SOLID

Принцип программирования KISS — делайте вещи проще

Большая часть программных систем необоснованно перегружена практически ненужными функциями, что ухудшает удобство их использования конечными пользователями, а также усложняет их поддержку и развитие разработчиками. Следование принципу «KISS» позволяет разрабатывать решения, которые просты в использовании и в сопровождении.

KISS — это принцип проектирования и программирования, при котором простота системы декларируется в качестве основной цели или ценности. Есть два варианта расшифровки аббревиатуры: «keep it simple, stupid» и более корректный «keep it short and simple».

В проектировании следование принципу KISS выражается в том, что:

- не имеет смысла реализовывать дополнительные функции, которые не будут использоваться вовсе или их использование крайне маловероятно, как правило, большинству пользователей достаточно базового функционала, а усложнение только вредит удобству приложения;
- не стоит перегружать интерфейс теми опциями, которые не будут нужны большинству пользователей, гораздо проще предусмотреть для них отдельный «расширенный» интерфейс (или вовсе отказаться от данного функционала);
- бессмысленно делать реализацию сложной бизнес-логики, которая учитывает абсолютно все возможные варианты поведения системы, пользователя и окружающей среды, — во-первых, это просто невозможно, а во-вторых, такая фанатичность заставляет собирать «звездолёт», что чаще всего иррационально с коммерческой точки зрения.

Принцип KISS and SOLID

В программировании следование принципу KISS можно описать так:

- не имеет смысла беспредельно увеличивать уровень абстракции, надо уметь вовремя остановиться;
- бессмысленно закладывать в проект избыточные функции «про запас», которые может быть когда-нибудь кому-либо понадобятся (тут скорее правильнее подход согласно принципу YAGNI);
- не стоит подключать огромную библиотеку, если вам от неё нужна лишь пара функций;
- декомпозиция чего-то сложного на простые составляющие — это архитектурно верный подход (тут KISS перекликается с DRY);
- абсолютная математическая точность или предельная детализация нужны не всегда — большинство систем создаются не для запуска космических шаттлов, данные можно и нужно обрабатывать с той точностью, которая достаточна для качественного решения задачи, а детализацию выдавать в нужном пользователю объёме, а не в максимально возможном объёме.

SOLID – принципы

SOLID — это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании — Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion. В переводе на русский: принципы единственной ответственности, открытости / закрытости, подстановки Барбары Лисков, разделения интерфейса и инверсии зависимостей)

Аббревиатура SOLID была предложена Робертом Мартином, автором нескольких книг, широко известных в сообществе разработчиков. Эти принципы позволяют строить на базе ООП масштабируемые программные продукты с понятной бизнес-логикой.

Расшифровка:

- Single responsibility — принцип единственной ответственности
- Open-closed — принцип открытости / закрытости
- Liskov substitution — принцип подстановки Барбары Лисков
- Interface segregation — принцип разделения интерфейса
- Dependency inversion — принцип инверсии зависимостей

SOLID – принципы

- **Принцип единственной обязанности / ответственности (single responsibility principle / SRP)** означает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.
- **Принцип открытости / закрытости (open-closed principle / OCP)** декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.
- **Принцип подстановки Барбары Лисков (Liskov substitution principle / LSP)** в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом».
- П**ринцип разделения интерфейса (interface segregation principle / ISP)** в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.
- **Принцип инверсии зависимостей(dependency inversion)** гласит: сущности должны зависеть от абстракций, а не от чего-то конкретного. Это означает, что модуль высокого уровня не должен зависеть от модуля низкого уровня, но они оба должны зависеть от абстракций.

СИНЕРГИЯ

2

Функции

Что такое функция?

Функция – это фрагмент кода, который можно выполнить многократно в разных частях программы. Т.е. одни и те же действия много раз с разными исходными значениями.

В следующем примере имеются повторяющиеся блоки кода, которые можно вынести отдельно в JavaScript функцию:

```
let a = 5;
let b = 7;
let sum;

sum = a + b;
console.log(sum); // 12

a = 10;
b = 4;

sum = a + b;
console.log(sum); // 14
```

Что такое функция?

Функция sum, в которую вынесен повторяющийся блок кода. После этого в местах программы, в которых его нужно выполнять, просто помещён вызов этой функции:

```
let a = 5;
let b = 7;

function sum(a, b) {
  sum = a + b;
  console.log(sum);
}

sum(a, b); // 12

a = 10;
b = 4;

sum(a, b); // 14
```

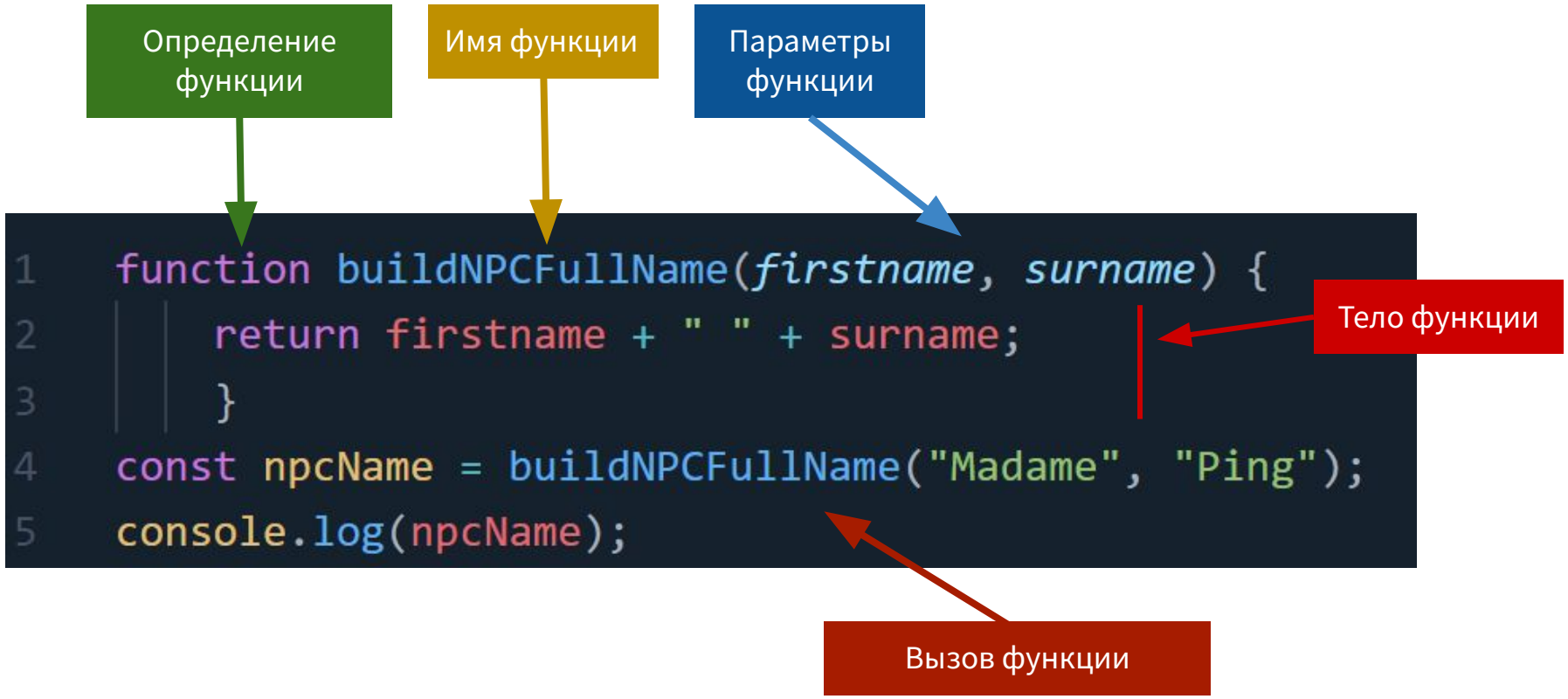
Это один из классических сценариев использования функций, который позволяет значительно упростить написание программ на JavaScript.

Что такое функция?

Кроме этого, функции позволяют структурировать код. Например, если перед вами стоит какая-то задача, то чтобы её проще написать, её можно разбить на подзадачи и оформить каждую из них в виде функции. А затем уже используя их написать финальный код. Вдобавок к этому в такой код будет проще вносить различные изменения и добавлять новые возможности.

```
1  function buildNPCFullName(firstname, surname) {
2      return firstname + " " + surname;
3      }
4  const npcName = buildNPCFullName("Madame", "Ping");
5  console.log(npcName);
```

Функция


Определение функции
Имя функции
Параметры функции
function buildNPCFullName(firstname, surname) {
return firstname + " " + surname;
}
const npcName = buildNPCFullName("Madame", "Ping");
console.log(npcName);
Тело функции
Вызов функции

Объявление и вызов функции

Операции с функцией в JavaScript можно разделить на 2 этапа:

- объявление (создание) функции;
- вызов (выполнение) этой функции.

Объявление функции

Написание функции посредством Function Declaration начинается с ключевого слова function. После чего указывается имя, круглые скобки, внутрь которых при необходимости помещаются параметры, и тело, заключённое в фигурные скобки. Имя функции ещё очень часто называют названием. В теле пишутся те действия, которые вы хотите выполнить с помощью этой функции.

```
// nameFn - имя функции, params - параметры
function nameFn (params) {
  // тело функции
}
```

```
// объявляем функцию someName
function someName() {
  console.log('Вы вызвали функцию someName!');
}
// function - ключевое слово, которое означает, что мы создаём функцию
// someName - имя функции
// () - круглые скобки, внутри которых при необходимости описываются параметры
// { ... } - тело функции
```

Объявление функции

При составлении имени функции необходимо руководствоваться такими же правилами, что для переменных. Т.е. можно использовать буквы, цифры (0 – 9), знаки «$» и «_». В качестве букв рекомендуется использовать английский алфавит (a-z, A-Z). Имя функции, также как и имя переменной не может начинаться с цифры.

При этом функции рекомендуется давать осмысленное название, т.е. такое, что было понятно, что она делает только исходя из её имени. Кроме этого, не нужно делать так, чтобы одна функция выполняла разные задачи. Лучше создать много различных функций, каждая из которых будет выполнять одну строго определённую задачу.

Вызов функции

Объявленная функция сама по себе не выполняется. Запуск функции выполняется посредством её вызова.
При этом когда мы объявляем функцию с именем, мы тем самым по сути создаём новую переменную с этим названием. Эта переменная будет функцией. Для вызова функции необходимо указать её имя и две круглые скобки, в которых при необходимости ей можно передать аргументы. Отделение одного аргумента от другого выполняется с помощью запятой.

```
// объявляем функцию someName
function someName() {
  console.log('Вы вызвали функцию someName!');
}
// вызываем функцию someName
someName();
```

Параметры функции

Параметры предназначены для получения значений, переданных в функцию, по имени. Их именование осуществляется также как переменных:

```
// объявляем функцию с двумя параметрами
  function fullname(firstname, lastname) {
  console.log(`${firstname} ${lastname}`);
}
```

Параметры ведут себя как переменные и в теле функции мы имеем доступ к ним. Значения этих переменных (в данном случае firstname и lastname) определяются в момент вызова функции. Обратиться к ним вне функции нельзя.

Если параметры не нужны, то круглые скобки всё равно указываются.

Параметры и аргументы

Параметры, как мы уже отметили выше – это по сути переменные, которые описываются в круглых скобках на этапе объявления функции. Параметры доступны только внутри функции, получить доступ к ним снаружи нельзя. Значения параметры получают в момент вызова функции, т.е. посредством аргументов.

Аргументы – это значения, которые мы передаём в функцию в момент её вызова.

```
// userFirstName и userLastName – параметры (userFirstName будет иметь значение первого аргумента,
function sayWelcome (userFirstName, userLastName) {
  console.log(`Добро пожаловать, ${userLastName} ${userFirstName}`);
}
// 'Иван' и 'Иванов' – аргументы
sayWelcome('Иван', 'Иванов'); // Добро пожаловать, Иванов Иван
// 'Петр' и 'Петров' – аргументы
sayWelcome('Петр', 'Петров'); // Добро пожаловать, Петров Петр
```

Параметры и аргументы

При вызове функции в JavaScript количество аргументов не обязательно должно совпадать с количеством параметров. Если аргумент не передан, а мы хотим его получить с помощью параметра, то он будет иметь значение undefined.

```
function sayWelcome (userFirstName, userLastName) {
  console.log( `Добро пожаловать, ${userLastName} ${userFirstName} `);
}
// с одним аргументом
sayWelcome('Иван'); // Добро пожаловать, undefined Иван
// без передачи аргументов
sayWelcome(); // Добро пожаловать, undefined undefined
```

Параметры и аргументы

Передача аргументов примитивных типов осуществляется по значению. Т.е. значение переменной не изменится снаружи, если мы изменим её значение внутри функции.

```
let a = 7;
let b = 5;

function sum (a, b) {
  a *= 2; // 14
  console.log(a + b);
}
sum(a, b); // 19
console.log(a); // 7
```

Передача значения по ссылке

Очень важно отметить при работе с функциями, что когда в качестве аргумента мы указываем ссылочный тип, то его изменение внутри функции изменит его и снаружи:

```
// объявим переменную someUser и присвоим ей объект, состоящий из двух свойств
const someUser = {
  firstname: 'Петр',
  lastname: 'Петров'
}
// объявим функцию changeUserName
function changeUserName(user) {
  // изменим значение свойства firstname на новое
  user.firstname = 'Александр';
}
// вызовем функцию changeUserName
changeUserName(someUser);
// выводим значение свойства firstname в консоль
console.log(someUser.firstname); // Александр
```

Передача значения по ссылке

Чтобы избежать изменение внешнего объекта, который мы передаем в функцию через аргумент, необходимо создать копию этого объекта, например, посредством Object.assign:

```
const someUser = {
  firstname: 'Петр',
  lastname: 'Петров'
}
function changeUserName(user) {
  // создадим копию объекта user
  const copyUser = Object.assign({}, user);
  // изменим значение свойства firstname на новое
  copyUser.firstname = 'Александр';
  // вернём копию объекта в качестве результата
  return copyUser;
}

// вызовем функцию и сохраним результат вызова функции в переменную updatedUser
const updatedUser = changeUserName(someUser);
// выводим значение свойства firstname в консоль для объекта someUser
console.log(someUser.firstname); // Петр
// выводим значение свойства firstname в консоль для объекта updatedUser
console.log(updatedUser.firstname); // Александр
```

Передача значения по ссылке

В примере мы внутри функции создали новый объект copyUser, который является копией объекта, переданного функции в качестве аргумента в момент её вызова. Т. е. someUser и copyUser - это разные объекты, хоть и содержащие на этапе копирования одинаковые свойства. После копирования, мы уже меняем свойство нового объекта и возвращаем его в качестве результата выполнения функции.

На практике бывают ситуации, когда внутри функции происходит изменение внешних объектов. Но в большинстве случаев изменять оригинальный объект или какие-то другие внешние переменные внутри функции не рекомендуется.

Локальные и внешние переменные

Переменные, объявленные внутри функции, называются локальными. Они не доступны вне функции. По сути это переменные, которые действуют только внутри функции.

```
let a = 7;
// объявление функции sum
function sum(a) {
  // локальная переменная функции
  let b = 8;
  console.log(a + b);
}
// вызов функции sum
sum(a);
console.log(b); // Error: b is not defined
```

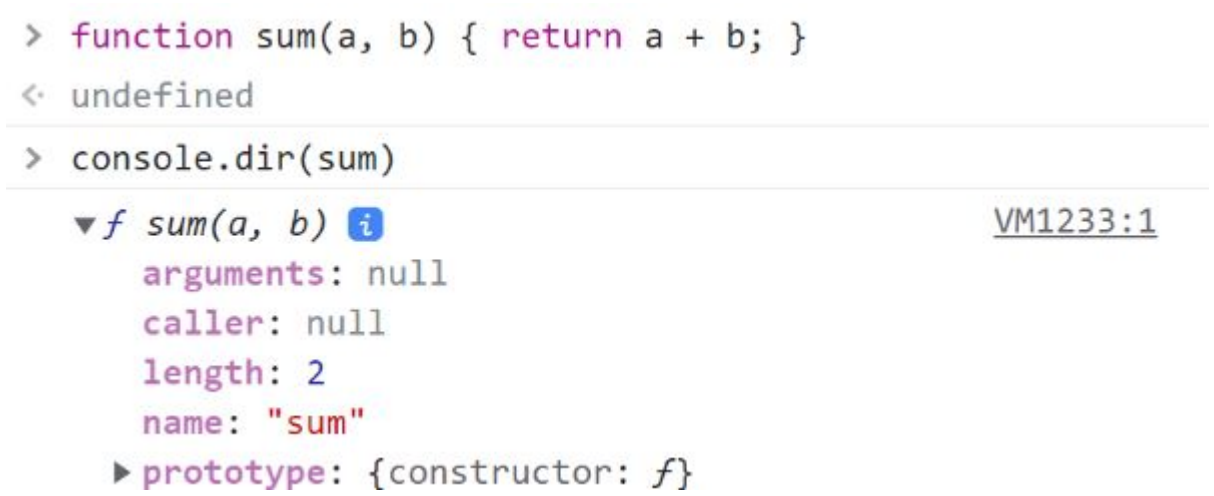
Локальные и внешние переменные

При этом, когда мы обращаемся к переменной и её нет внутри функции, она берётся снаружи. Переменные объявленные вне функции являются по отношению к ней внешними.

```
// внешние переменные
let a = 7;
let b = 3;
function sum() {
  // локальная переменная
  let a = 8;
  // изменение значения внешней переменной (т.к. b нет внутри функции)
  b = 4;
  console.log(a + b);
}
sum(); // 12
```

Функция – это объект

Функция в JavaScript, как уже было отмечено выше – это определённый тип объектов, которые можно вызывать. А если функция является объектом, то у неё как у любого объекта имеются свойства. Убедиться в этом очень просто. Для этого можно воспользоваться методом console.dir() и передать ему в качестве аргумента функцию. На изображении показана структура функции sum. Используя точечную запись, мы например, можем получить название функции (свойство name) и количество параметров(length):



Функция – это объект

На изображении показана структура функции sum. Используя точечную запись, мы например, можем получить название функции (свойство name) и количество параметров(length):

```
console.log(sum.name); // sum
console.log(sum.length); // 2
```

Узнать является ли переменная функцией можно с помощью typeof:

```
function myFunc() {};

console.log(typeof myFunc); // function
```

Возврат значения

Функция всегда возвращает значение, даже если мы не указываем это явно. По умолчанию она возвращает значение undefined.
Для явного указания значения, которое должна возвращать функция используется инструкция return. При этом значение или выражение, результат которого должна вернуть функция задаётся после этого ключевого слова.

```
// expression – выражение, результат которого будет возвращен функцией myFn
function myFn() {
  return expression;
}
```

Возврат значения

Если return не указать, то функция всё равно возвратит значение, в данном случае undefined.

```
function sum(a, b) {
  console.log(a + b);
}

// вызовем функцию и сохраним её результат в константу result
const result = sum(4, 3);
// выведем значение переменной result в консоль
console.log(result); // undefined
```

Возврат значения

С использованием инструкции return:

```
function sum(a, b) {
  // вернём в качестве результата a + b
  return a + b;
}

// вызовем функцию и сохраним её результат в константу result
const result = sum(4, 3);
// выведем значение переменной result в консоль
console.log(result); // 7
```

Возврат значения

Инструкции, расположенные после return никогда не выполняются:

```
function sum(a, b) {
  // вернём в качестве результата a + b
  return a + b;
  // код, расположенный после return никогда не выполнится
  console.log('Это сообщение не будет выведено в консоль');
}

sum(4, 90);
```

В этом примере, функция sum возвращает число 94 и прекращает выполнение дальнейших инструкций после return. А так как работа функции закончилась, то сообщение в консоль выведено не будет.

Что такое встроенные (стандартные) функции

В JavaScript имеется огромный набор встроенных (стандартных) функций. Данные функции уже описаны в самом движке браузера. Практически все они являются методами того или иного объекта.

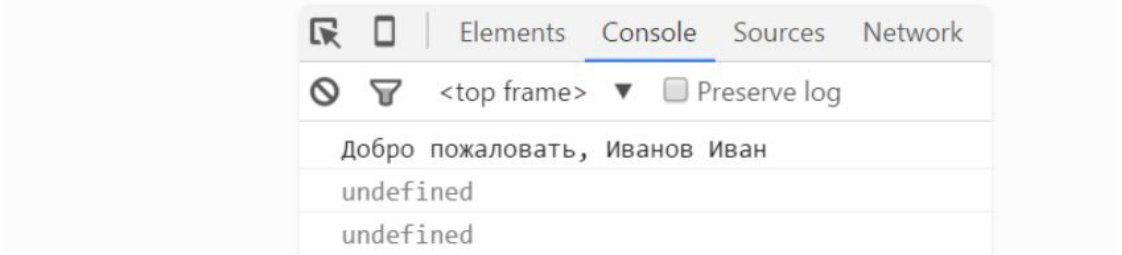
Например, для того чтобы вызвать встроенную функцию (метод) alert, её не надо предварительно объявлять. Она уже описана в браузере. Вызов метода alert осуществляется посредством указания имени, круглых скобок и аргумента внутри них. Данный метод предназначен для вывода сообщения на экран в форме диалогового окна. Текстовое сообщение берётся из значения параметра данной функции.

```
// вызов функции alert
alert("Некоторый текст");
```

Что такое встроенные (стандартные) функции

Функция в JavaScript в результате своего выполнения всегда возвращает результат, даже если он явно не определён с помощью оператора return. Этот результат значение undefined

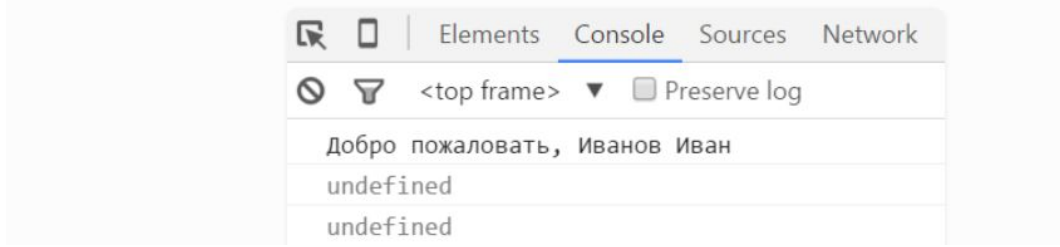
```
// 1. функция, не возвращающая никакого результата
function sayWelcome (userFirstName, userLastName) {
  console.log("Добро пожаловать, " + userLastName + " " + userFirstName);
}
// попробуем получить результат у функции, которая ничего не возвращает
console.log(sayWelcome ("Иван", "Иванов"));
// 2. функция, содержащая оператор return без значения
function sayDay (day) {
  day = "Сегодня, " + day;
  return;
  //эта инструкция не выполнится, т.к. она идёт после оператора return
  console.log(day);
}
// попробуем получить результат у функции, которая содержит оператор return без значения
console.log(sayDay("21 февраля 2016г."));
```



Что такое встроенные (стандартные) функции

Функция в JavaScript в результате своего выполнения всегда возвращает результат, даже если он явно не определён с помощью оператора return. Этот результат значение undefined.

```
// 1. функция, не возвращающая никакого результата
function sayWelcome (userFirstName, userLastName) {
  console.log("Добро пожаловать, " + userLastName + " " + userFirstName);
}
// попробуем получить результат у функции, которая ничего не возвращает
console.log(sayWelcome ("Иван", "Иванов"));
// 2. функция, содержащая оператор return без значения
function sayDay (day) {
  day = "Сегодня, " + day;
  return;
  //эта инструкция не выполнится, т.к. она идёт после оператора return
  console.log(day);
}
// попробуем получить результат у функции, которая содержит оператор return без значения
console.log(sayDay("21 февраля 2016г."));
```



Такой же результат будет получен, если для оператора return не указать возвращаемое значение.