

Язык JavaScript (ООП в  
функциональном стиле)

**JS**

# Создание объектов

```
let storage = {  
  store: [],  
  set: function (key, value) {  
    this.store[key] = value;  
  },  
  get: function(key) {  
    if(!key) return;  
    return this.store[key];  
  }  
}  
  
storage.set('name', 'Peter');  
storage.get('name');
```

Как задать еще один объект хранилище ?

# Функция как шаблон для новых объектов

**Конструктором** становится любая функция, вызванная через **new**.

Функция, запущенная через **new**, делает следующее:

- ❖ Создаётся новый пустой объект;
- ❖ Ключевое слово **this** получает ссылку на этот пустой объект;
- ❖ Функция выполняется. Как правило, она модифицирует **this**, добавляет методы, свойства;
- ❖ Возвращается **this**.

# Функция как шаблон для новых объектов

```
function Storage() {  
    this.store = [];  
    this.set = function (key, value) {  
        this.store[key] = value;  
    };  
    this.get = function (key) {  
        return this.store[key];  
    };  
}
```

С помощью ключевого слова `new` и функции шаблона, мы можем создать объект с одними и теми же свойствами и методами столько раз, сколько нам необходимо:

```
let store1 = new Storage();  
store1.set('name', 'Peter');  
let store2 = new Storage();
```

# Добавление свойств к шаблону

Так как функция является по сути объектом то мы смело можем добавить к ней свойства и методы (их называют статические):

```
Storage.type = 'digit';  
Storage.maxSize = 100;  
Storage.setMaxSize = function (value) {  
    this.maxSize = value;  
};
```

В данном примере эти свойства и методы не будут частью шаблона если мы выполним `new Storage();`

# Создание объектов в конструкторах

Как правило, **конструкторы ничего не возвращают**. Их задача – записать всё, что нужно, в `this`, который автоматически станет результатом.

Но если явный вызов `return` всё же есть, то применяется простое правило:

- ❖ при вызове `return` с объектом, будет возвращён он, а не `this`;
- ❖ при вызове `return` с примитивным значением, оно будет отброшено.

# Создание объектов в конструкторах

```
function Storage(max) {  
    this.store = [];  
    this.set = function (key, value) {  
        this.store[key] = value;  
    };  
    this.get = function (key) {  
        return this.store[key];  
    };  
  
    return { maxSize: max };  
}
```

```
let hub = new Storage(20); // создаем объект  
hub.set('name', 'Peter'); // undefined!  
hub.maxSize; // 20
```

# Приватные переменные, доступные только внутри нашего объекта

Локальные переменные, включая параметры конструктора, можно считать приватными свойствами.

```
function Storage(max) {  
    let store = [];  
    this.set = function (key, value) {  
        store[key] = value;  
    };  
    this.get = function (key) {  
        return store[key];  
    };  
}
```

```
let db = new Storage();  
db.store; // undefined  
db.set('name', 'Peter');
```



# Преобразование объектов: toString и valueOf

---

Бывают операции, при которых объект должен быть преобразован в примитив (число, строку, логический тип).

Любой объект в логическом контексте – true, даже если это пустой массив [] или объект {}.

Если в объекте присутствует метод toString, который возвращает примитив, то он используется для преобразования объекта к строке.

Для численного преобразования объекта используется метод valueOf, а если его нет – то toString.

# Преобразование объектов: toString и valueOf

```
function User(firstName, lastName, age) {  
  
    this.toString = function() {  
        return firstName + ' ' + lastName;  
    };  
    this.valueOf = function(){  
        return age;  
    };  
}  
  
let user = new User("Иван", "Иванов", 18);  
console.log(user);  
console.log("Пользователь совершенно летний: " +  
    (user >= 18?"Да":"Нет"));
```

# Ссылочный тип

```
function User() {  
  
    this.firstName = "Иван";  
    this.lastName = "Иванов";  
  
    this.fullName = function() {  
        return this.firstName + ' ' + this.lastName;  
    };  
}
```

```
let user = new User();  
let fullName = user.fullName;  
console.log(fullName()); //Что будет выведено??
```

# Явное указание this

```
function User(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
};
```

```
let fullName = function() {  
    return this.firstName + ' ' + this.lastName;  
};
```

```
let user = new User("Иван", "ИВАНОВИЧ");  
console.log(fullName.call(user));  
console.log(fullName.apply(user));
```

# Привязка контекста

```
function User() {  
  
    this.firstName = "Иван";  
    this.lastName = "Иванов";  
  
    this.fullName = function() {  
        return this.firstName + ' ' + this.lastName;  
    }.bind(this);  
}
```

```
let user = new User();  
let fullName = user.fullName;  
console.log(fullName());
```

# Наследование в функциональном стиле

---

Наследование – это создание новых «классов» на основе существующих.

Цель наследования – это наследование общего функционала и тем самым не дублировать код.

В JavaScript его можно реализовать несколькими путями, один из которых – с использованием наложения конструкторов.

# Наследование в функциональном стиле

```
function Shape(centerX, centerY){
  this.centerX = centerX;
  this.centerY = centerY;
  this.toString = function(){
    return "Координаты центра " + this.centerX +
      ":" + this.centerY;
  }
}

function Circle(centerX, centerY, radius){
  Shape.call(this, centerX, centerY);
  this.radius = radius;
  this.toString = function(){
    return "Координаты центра " + this.centerX +
      ":" + this.centerY + " Радиус " + this.radius;
  }
}
```

# Полиморфный конструктор

---

Когда непонятно, какого типа аргумент передадут, и хочется в одном конструкторе охватить все варианты **лучше использовать полиморфный конструктор.**

А в остальных случаях отличная альтернатива – фабричные методы.



# Полиморфный конструктор

```
function Shape(centerX, centerY, radius){
  if (arguments.length == 3) {
    this.centerX = centerX;
    this.centerY = centerY;
    this.radius = radius;
    this.toString = function(){
      return "Круг: Координаты центра " + this.centerX
        + ":" + this.centerY + " Радиус " + this.radius;
    }
  } else if (arguments.length == 2) {
    this.centerX = centerX;
    this.centerY = centerY;
    this.toString = function(){
      return "Точка: Координаты " + this.centerX + ":" +
        this.centerY;
    }
  } else {
    this.toString = function(){ return "Абстрактная фигура"; }
  }
}
```

# Фабричные методы

---

Фабричный метод - называется статический метод, который служит для создания новых объектов (поэтому и называется «фабричным»).

Использование фабричных методов обеспечивает: лучшую читаемость кода и удобную расширяемость.

# Фабричные методы

```
function Shape(){  
    this.toString = function(){  
        return "Абстрактная фигура";  
    }  
}
```

```
Shape.point = function(data){  
    let shape = new Shape();  
    shape.centerX = data.centerX;  
    shape.centerY = data.centerY;  
    shape.toString = function(){  
        return "Точка: Координаты " + this.centerX + ":" +  
            this.centerY;  
    }  
    return shape;  
}
```