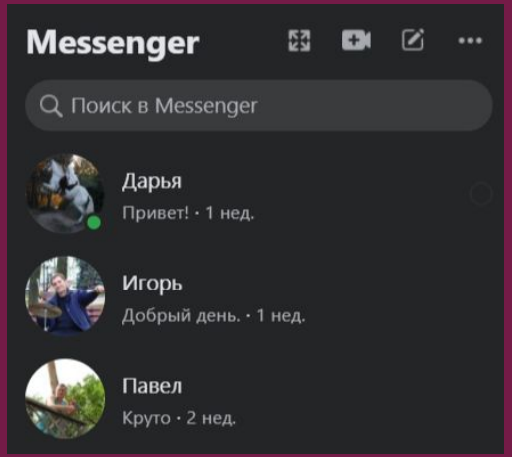
JS Fundamentals
this
•••

Автор презентации: Макухина Марина

Контекст выполнения

Контекст выполнения (*execution context*) — это концепция, описывающая окружение, в котором производится выполнение кода на JavaScript. Код всегда выполняется внутри некоего контекста.

Рассмотрим контекст выполнения на примере сообщений в Facebook. Все сообщения имеют одинаковую структуру — получателя, текст сообщения и дату отправки. Можно рассмотреть переписку с определенным пользователем как контекст выполнения. При определения получателя вся дальнейшая переписка будет проходить с конкретным человеком. То есть выполняться в определенном контексте.



Область видимости

Область видимости (scope) устанавливает, где и когда определяются переменные, константы и аргументы. Переменные, которые еще не были объявлены, или переменные, которые прекратили существование после выхода из функции, не находятся в области видимости.

Подобно тому как блок или функция может вкладываться внутрь другого блока или функции, области видимости могут вкладываться в другие области видимости. Если переменную не удается найти в текущей области видимости, движок обращается к следующей внешней области видимости. Это продолжается до тех пор, пока не будет найдена искомая переменная или не будет достигнута глобальная область видимости.

Глобальная область видимости

Вновь запущенная программа JavaScript (прежде, чем будут вызваны любые функции) выполняется в глобальной области видимости. Всё, что было объявлено в глобальной области видимости, будет доступно для всех областей видимости в программе.

```
let age = 10;
function grow() {
  age = 100;
  return age;
}
console.log(grow()); // 100
```

Проблема этого подхода в том, что функции жестко зависят от контекста (или области видимости), из которого они вызываются. Любая функция в любой части программы может изменить значение глобальной переменной.

Лексическое окружение

Когда в программе вызывается функция, внутри интерпретатора создается специальный (скрытый) объект *LexicalEnvironment* (лексическое окружение), привязанный к этому вызову. Все определения констант, переменных и прочего внутри функции автоматически записываются в словарь.

Объект лексического окружения состоит из двух частей:

- *Environment record* — это место, в котором хранятся все локальные переменные;
- *outer* — ссылка на внешнее лексическое окружение: то есть то, которое соответствует коду снаружи.

Глобальное лексическое окружение

У каждой выполняемой функции, блока кода и скрипта есть определенное лексическое окружение. Внутри блоков и функций можно создавать другие блоки и функции, образуя цепочки лексических окружений. На вершине иерархии всегда будет располагаться глобальное лексическое окружение.

```
let day = "Monday";
function showDay() {
  console.log(day);
}
day = "Tuesday";
showDay();
```

В функции *showDay()* запрашивается доступ к переменной *day*. Интерпретатор попытается найти переменную в текущей записи окружения. Так как она не будет найдена, интерпретатор пойдет по цепочке дальше. Благодаря существующему в каждом лексическом окружении полю *outer*, поиск продолжается до тех пор, пока переменная не обнаружится в каком-то из внешних окружений.

Ключевое слово this

Во время создания лексического окружения, например если была вызвана функция, будет сформирована *Environment record*, содержащая информацию о переменных текущей области видимости, а также ключевое слово *this*. Значение *this* динамически устанавливается JavaScript-движком на этапе создания контекста выполнения и указывает на объект, связанный с этим контекстом.

Если использовать *this* в глобальной области, оно будет просто ссылаться на глобальный объект. В нестрогом режиме значением *this* будет объект window.

```
console.log(this);
// Window {parent: Window, opener: null, top: Window, length: 4, frames: Window, …}
```

Применение this

Для доступа к информации внутри объекта метод может использовать ключевое слово *this*.

В JavaScript значение *this* вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

```
let user = {
  name: "Kevin",
  sayHi() {
    console.log(this.name);
  },
};
user.sayHi(); // Kevin
```

Функция может быть повторно использована в качестве метода у различных объектов. Ключевое слово this относиться к тому объекту, в методе которого оно используется.

Использование this внутри функции

Одна и та же функция, которая была назначена двум разным объектам, имеет различное значение *this* при вызовах.

```
function sendMessage(message) {
  return `To: ${this.name}, message: ${message}`;
}
let user1 = { name: "Darya" };
let user2 = { name: "Leo" };
user1.sendMessage = sendMessage;
user2.sendMessage = sendMessage;
console.log(user1.sendMessage("Hello!")); // To: Darya, message: Hello!
console.log(user2.sendMessage("Hello!")); // To: Leo, message: Hello!
```

Вызов такой функции без объекта в строгом режиме приведет к ошибке.

Функции-конструкторы

Если перед вызовом функции присутствует ключевое слово *new*, то данная функция будет действовать как конструктор. Такой вызов создаёт пустой *this* в начале выполнения и возвращает заполненный в конце.

Функции-конструкторы являются обычными функциями, но их принято называть с большой буквы. Основная цель конструкторов – удобное повторное создание однотипных объектов.

```
function Triangle(a, b, c) {
  this.a = a;
  this.b = b;
  this.c = c;
}

let tr = new Triangle(1, 2, 3);

console.log("P = " +
 (tr.a + tr.b + tr.c));
// P = 6
```

Явная передача контекста в функцию

При вызове функции можно явно передать ей значение *this* с помощью специальных методов:

- *apply* вызывает функцию с указанным значением *this* и аргументами в виде массива;
- *call* вызывает функцию с указанным значением *this* и индивидуально предоставленными аргументами.

```
function sendMessage(txt) {
 return `To: ${this.name}, message:
${txt}`;
}

let user = { name: "Darya" };
let res1 = sendMessage.apply(user, ["hi"]);
let res2 = sendMessage.call(user, "hi");
console.log(res1);
// To: Darya, message: hi
console.log(res2);
// To: Darya, message: hi
```

Преобразования массивоподобных объектов к массиву

Метод *slice()* может использоваться для преобразования массивоподобных объектов в новый массив *Array*. Методу *slice()* для работы требует только нумерованные свойства и length.

Массивоподобный объект *arguments* как раз имеет числовый имена свойств. Преобразование *arguments* к массиву позволит использовать любые массивные методы (map, filter, reduce и т.д.) для работы с аргументами.

```
function printNumbers() {
  let args = [].slice.call(arguments);
  console.log(args.filter((v) => typeof v == "number"));
}
printNumbers(10, "portal", 8, 23, true); // [10, 8, 23]
```

Стрелочные функции и this

Стрелочные функции не содержат собственный контекст *this*, а используют значение *this* окружающего контекста.

Так как значение this определяется лексикой, вызов стрелочных функций с помощью методов *call* | *apply*, даже если передать аргументы в эти методы, не влияет на значение *this*.

```
let person = {
  name: "Luke",
  sayHi() {
    let arrow = () =>
console.log("Hi, " + this.name);
    arrow();
  },
};
person.sayHi();
```

Отсутствие *this* естественным образом ведёт к другому ограничению: стрелочные функции не могут быть использованы как конструкторы. Они не могут быть вызваны с *new*.

Метод bind

Метод *bind* позволяет создать функцию, которая, вне зависимости от способа её вызова, вызывается с определённым значением *this*.

Возвращаемый при первом вызове *bind* “необычный функциональный объект” запоминает контекст только во время создания. То есть невозможно изменить существующую привязку с помощью повторного использования *bind* или *call* и *apply*.

```
let item = { name: "drone" };
function getItemData(id) {
  console.log(id + " " + this.name);
}
let func = getItemData.bind(item);
func(1); // 1 drone
func.call({ name: "hat" }, 2); // 2 drone
getItemData.apply({ name: "tv" }, [3]);
// 3 tv
```

Полезные ссылки

https://learn.javascript.ru/arrow-functions

https://learn.javascript.ru/bind

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

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice

На этом всё!

•••

Спасибо за внимание