JavaScript - продвинутый

Рекурсия и стек;
область видимости и замыкание;
каррирование;
функции IIFE и функции NFE;
синтаксис new Function и eval;

Стек исполнения

Стек исполнения (или стек вызовов) — это структура данных, используемая в программировании для отслеживания активных подпрограмм (или функций) в процессе их выполнения. Каждый раз, когда вызывается функция, она добавляется на вершину стека. Когда выполнение функции завершается, она удаляется из стека.

Основные моменты о стеке исполнения в JavaScript:

- LIFO (Last In, First Out): Последний вызов функции добавляется на вершину стека и первым завершает выполнение.
- Однопоточность: JavaScript выполняется в одном потоке, поэтому он имеет только один стек исполнения.
- Обработка событий: JavaScript использует цикл событий (event loop) для управления асинхронными операциями, которые обрабатываются в очереди задач, когда стек исполнения пуст.

Контекст выполнения

Информация о процессе выполнения запущенной функции хранится в её контексте выполнения (execution context).

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение this (мы не используем его в данном примере) и прочую служебную информацию.

Один вызов функции имеет ровно один контекст выполнения, связанный с ним.

Когда функция производит вложенный вызов, происходит следующее:

Выполнение текущей функции приостанавливается.

Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – стеке контекстов выполнения.

Выполняются вложенные вызовы, для каждого из которых создается свой контекст выполнения.

После их завершения старый контекст достается из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

Рекурсия

Рекурсия – это термин в программировании, означающий вызов функцией самой себя. Рекурсивные функции могут быть использованы для элегантного решения определённых задач.

Когда функция вызывает саму себя, это называется шагом рекурсии. База рекурсии – это такие аргументы функции, которые делают задачу настолько простой, что решение не требует дальнейших вложенных вызовов.

1. Итеративный способ: цикл `for`:

```
function pow(x, n) {
  let result = 1;

  // умножаем result на x n раз в цикле
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

alert( pow(2, 3) ); // 8
```

2. Рекурсивный способ: упрощение задачи и вызов функцией самой себя:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); // 8
```

Область видимости

Глобальная область видимости (Global Scope)

Переменные и функции, объявленные вне любых функций или блоков, находятся в глобальной области видимости. Они доступны везде в коде.

Функциональная область видимости (Function Scope)

Переменные, объявленные внутри функции с помощью var, let или const, находятся в функциональной области видимости. Они доступны только внутри этой функции.

Область видимости блока (Block Scope)

Переменные, объявленные внутри блока (например, внутри {}) с помощью let или const, находятся в области видимости блока. Они доступны только внутри этого блока.

Область видимости модуля (Module Scope)

В контексте ES6 модулей, каждая переменная или функция, объявленная внутри модуля, имеет область видимости модуля. Эти переменные и функции не доступны в глобальной области видимости, а только внутри модуля, если они не экспортированы.

Лексическая область видимости (Lexical Scope)

JavaScript использует лексическую область видимости, что означает, что область видимости определяется по месту объявления переменных в исходном коде, а не по месту их вызова. Вложенные функции могут получать доступ к переменным своих родительских функций.

Цепочка областей видимости (Scope Chain)

Когда интерпретатор JavaScript встречает переменную, он начинает поиск этой переменной в текущей области видимости. Если переменная не найдена, поиск продолжается в следующей (родительской) области видимости и так далее, пока не будет достигнута глобальная область видимости. Если переменная не найдена нигде, возникает ошибка.

```
var globalVar = "I am global";

function outerFunction() {
    var outerVar = "I am outer";

    function innerFunction() {
        var innerVar = "I am inner";

        console.log(innerVar); // Найдена в текущей области видимости (innerFunction)
        console.log(outerVar); // Найдена в родительской области видимости (outerFunction)
        console.log(globalVar); // Найдена в глобальной области видимости
    }

    innerFunction();
}

outerFunction();
```

Особенность var в блочной области видимости

Область видимости

var: Переменные, объявленные с помощью var, имеют функциональную область видимости. Они доступны во всей функции, в которой они объявлены, независимо от того, где именно внутри функции они были объявлены. Если var используется вне функции, переменная имеет глобальную область видимости.

let и const: Переменные, объявленные с помощью let и const, имеют блочную область видимости. Они доступны только внутри блока {}, в котором они объявлены.

```
function exampleVar() {
    if (true) {
        var x = "var variable";
    }
    console.log(x); // "var variable" – доступно, так как x имеет функциональную область видимости
}

exampleVar();

function exampleLetConst() {
    if (true) {
        let y = "let variable";
        const z = "const variable";
    }
    console.log(y); // Ошибка: y не определена
    console.log(z); // Ошибка: z не определена
}

exampleLetConst();
```

Влияние на глобальный объект

var: Переменные, объявленные с помощью var в глобальной области видимости, становятся свойствами глобального объекта (например, window в браузере).

let и const: Переменные, объявленные с помощью let и const в глобальной области видимости, не становятся свойствами глобального объекта.

```
var globalVar = "global var";
console.log(window.globalVar); // "global var"

let globalLet = "global let";
console.log(window.globalLet); // undefined

const globalConst = "global const";
console.log(window.globalConst); // undefined
```

Поведение при поднятии (Hoisting)

var: Переменные, объявленные с помощью var, поднимаются в начало своей области видимости (функции или глобальной). При этом они инициализируются значением undefined.

let и const: Переменные, объявленные с помощью let и const, также поднимаются в начало их блока, но они остаются неинициализированными до фактического выполнения объявления. Доступ к таким переменным до их объявления приводит к ошибке (находятся в "временной мертвой зоне").

```
console.log(a); // undefined — поднятие переменной var
var a = "var variable";

console.log(b); // Ошибка: Cannot access 'b' before initialization
let b = "let variable";

console.log(c); // Ошибка: Cannot access 'c' before initialization
const c = "const variable";
```

Временная мёртвая зона (Temporal Dead Zone, TDZ) — это термин, описывающий поведение переменных, объявленных с помощью let и const в JavaScript. Временная мёртвая зона начинается с момента входа в область видимости блока, в котором объявлена переменная, и заканчивается в точке ее фактического объявления и инициализации.

Обновляемость и переопределение

var: Переменные, объявленные с помощью var, могут быть переобъявлены в той же области видимости без ошибок.

let и const: Переменные, объявленные с помощью let и const, не могут быть переобъявлены в той же области видимости. Переменные, объявленные с помощью const, также не могут быть переназначены.

```
var d = "first var";
var d = "second var"; // Не вызывает ошибку

let e = "first let";
// let e = "second let"; // Ошибка: Identifier 'e' has already been declared

const f = "first const";
// const f = "second const"; // Ошибка: Identifier 'f' has already been declared
// f = "second const"; // Ошибка: Assignment to constant variable
```

Замыкание

Замыкание (closure) в JavaScript — это функция, которая имеет доступ к своему собственному областью видимости, к области видимости внешней функции и к глобальной области видимости. Замыкание создается, когда функция объявляется внутри другой функции, и внутренняя функция получает доступ к переменным внешней функции.

Замыкания позволяют функции "запоминать" её окружение, даже после того, как внешняя функция завершила выполнение. Это возможно благодаря тому, что в JavaScript функции являются объектами первого класса и могут сохранять ссылки на свои области видимости.

Замыкания — мощный инструмент в JavaScript, позволяющий создавать функции с приватным состоянием, запоминать контекст и обеспечивать доступ к переменным внешних функций. Они широко используются для создания функций-обработчиков, модулей и других конструкций, требующих сохранения состояния.

Пример

1. Функция outerFunction объявляет переменную outerVariable и внутреннюю функцию innerFunction.
2. Функция innerFunction имеет доступ к переменной outerVariable из внешней функции.
3. outerFunction возвращает innerFunction.
4. Переменная myInnerFunction сохраняет ссылку на возвращенную внутреннюю функцию.
5. Когда вызывается myInnerFunction, она по-прежнему имеет доступ к переменной outerVariable, даже после того как outerFunction завершила выполнение.

```
function outerFunction() {
    let outerVariable = "I am from outer function";

    function innerFunction() {
        console.log(outerVariable);
    }

    return innerFunction;
}

const myInnerFunction = outerFunction();
myInnerFunction(); // "I am from outer function"
```

Каррирование

Каррирование (currying) в JavaScript — это техника трансформации функции, которая принимает несколько аргументов, в последовательность функций, каждая из которых принимает один аргумент. Каррирование позволяет разбить процесс вызова функции на несколько этапов, что может быть полезно для создания более гибких и переиспользуемых функций.

Преимущества каррирования

- Переиспользуемость: Каррирование позволяет создать частично применённые функции, которые можно переиспользовать с разными аргументами.
- Функциональное программирование: Каррирование является важной техникой в функциональном программировании, делая код более декларативным и читабельным.
- Улучшение композиции: Каррированные функции легче комбинировать и использовать в цепочках вызовов функций.

Пример

```
function add(x, y, z) {
    return x + y + z;
}

console.log(add(1, 2, 3)); // 6
```

```
const curriedAdd = x => y => z => x + y + z;

console.log(curriedAdd(1)(2)(3)); // 6
```

```
function curriedAdd(x) {
    return function(y) {
        return function(z) {
            return x + y + z;
        };
    };
}

console.log(curriedAdd(1)(2)(3)); // 6
```

Частичное применение

Каррирование позволяет частично применять функции, что может быть полезно в различных сценариях.

```
const add = x => y => x + y;

const add5 = add(5);
console.log(add5(3)); // 8
console.log(add5(10)); // 15
```

Универсальная функция каррирования

Можно создать универсальную функцию для каррирования любой функции:

Каррирование в JavaScript — это мощная техника, которая позволяет создавать функции, принимающие один аргумент за раз, из функций, принимающих несколько аргументов. Это делает функции более гибкими и переиспользуемыми, что особенно полезно в контексте функционального программирования.

```
function curry(func) {
    return function curried(...args) {
        if (args.length >= func.length) {
            return func.apply(this, args);
        } else {
            return function(...args2) {
                return curried.apply(this, args.concat(args2));
            };
        }
    };
}

function multiply(a, b, c) {
    return a * b * c;
}

const curriedMultiply = curry(multiply);

console.log(curriedMultiply(2)(3)(4)); // 24
console.log(curriedMultiply(2, 3)(4)); // 24
console.log(curriedMultiply(2)(3, 4)); // 24
```

Функции IIFE

В прошлом, поскольку существовал только var, а он не имел блочной области видимости, программисты придумали способ ее эмулировать. Этот способ получил название «Immediately-invoked function expressions» (сокращенно IIFE).

Это не то, что мы должны использовать сегодня, но, так как вы можете встретить это в старых скриптах, полезно понимать принцип работы.

```
(function() {

  var message = "Привет";

  alert(message); // Привет

})();
```

```
// Способы создания IIFE

(function() {
  alert("Круглые скобки вокруг функции");
})();

(function() {
  alert("Круглые скобки вокруг всего выражения");
}());

!function() {
  alert("Выражение начинается с логического оператора НЕ");
}();

+function() {
  alert("Выражение начинается с унарного плюса");
}();
```

Функция это объект

Каждое значение в JavaScript имеет свой тип. А функция – это какой тип?

В JavaScript функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но и использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т. д.

Свойство «name» - имя функции

Свойство «length» - содержит количество параметров функции в её объявлении.

Пользовательские свойства

Мы также можем добавить свои собственные свойства.

Свойство не есть переменная

Свойство функции, назначенное как sayHi.counter = 0, не объявляет локальную переменную counter внутри неё. Другими словами, свойство counter и переменная let counter – это две независимые вещи.

Мы можем использовать функцию как объект, хранить в ней свойства, но они никак не влияют на ее выполнение. Переменные – это не свойства функции и наоборот. Это два параллельных мира.

```
function sayHi() {
  alert("Hi");

  // давайте посчитаем, сколько вызовов мы сделали
  sayHi.counter++;
}
sayHi.counter = 0; // начальное значение

sayHi(); // Hi
sayHi(); // Hi

alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2 раза
```

Функции NFE

В контексте JavaScript, NFE расшифровывается как "Named Function Expression" (Именованное Функциональное Выражение). Это концепция, когда функция в JavaScript объявляется внутри выражения и имеет имя, которое может использоваться внутри самой функции или вне ее.

Особенности именованных функциональных выражений:

- Имя доступно внутри функции: Использование имени функции внутри самой функции может быть полезно для самовызова или рекурсивных вызовов.
- Имя доступно снаружи выражения: Имя функции также доступно в области видимости, где было объявлено функциональное выражение, но обычно это имя не доступно в глобальной области видимости.
- Отладка: Использование имен в функциональных выражениях улучшает отладку кода, потому что имена функций отображаются в стеке вызовов и в инструментах разработчика браузера.

Пример

```
let factorial = function fact(n) {
    if (n === 0) {
        return 1;
    }
    return n * fact(n - 1);
};

console.log(factorial(5)); // Выводит 120
```

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // Теперь всё в порядке
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (вложенный вызов работает)
```

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Ошибка: sayHi не является функцией
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Ошибка, вложенный вызов sayHi больше не работает!
```

Функция NFE не работает с Function Declaration

Трюк с «внутренним» именем, описанный выше, работает только для Function Expression и не работает для Function Declaration. Для Function Declaration синтаксис не предусматривает возможность объявить дополнительное «внутреннее» имя.

Зачастую, когда нам нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

Синтаксис new Function

В JavaScript оператор new Function позволяет создавать новые функции динамически во время выполнения программы на основе переданных строк кода. Этот подход полезен в ситуациях, когда необходимо создать функцию на лету, например, в ответ на данные, полученные от пользователя или из внешнего источника данных.

Синтаксис оператора new Function выглядит следующим образом:

- **arg1, arg2, ...argN:** Это строки, представляющие аргументы функции. Они являются именами параметров, которые будут доступны внутри созданной функции.
- **functionBody:** Это строка, содержащая тело функции, которую нужно создать. Она содержит JavaScript-код, который будет выполнен при вызове этой функции.

```
new Function([arg1, arg2, ...argN], functionBody)
```

```
const sum = new Function('a', 'b', 'return a + b;');
console.log(sum(2, 3)); // Выводит 5
```

Особенности

Область видимости: Функции, созданные с помощью new Function, не наследуют лексическую область видимости места их создания. Они имеют доступ только к глобальной области видимости и области видимости window (в браузере).

Безопасность: Использование new Function с необработанными внешними данными может представлять угрозу безопасности приложения (например, инъекции кода). Поэтому следует аккуратно обрабатывать и проверять входные данные перед использованием их в new Function.

Производительность: Создание функций с помощью new Function может быть менее эффективным, чем обычные объявления функций в коде, из-за необходимости компиляции и интерпретации строки кода.

eval

eval в JavaScript — это встроенная функция, которая принимает строку в качестве аргумента и выполняет её как JavaScript код на текущей области видимости. Она позволяет динамически выполнять код, создавать новые функции или переменные, и интерпретировать строки как JavaScript выражения.

eval(expressionString)

expressionString: Это строка, содержащая JavaScript код, который нужно выполнить.

Важные аспекты использования eval

- **Область видимости:** Код, выполненный через eval, выполняется в текущей лексической области видимости и может создавать или изменять переменные в этой области.
- **Безопасность:** Использование eval с непроверенными или внешними данными может быть опасным, так как это может привести к выполнению вредоносного кода (инъекции кода). Поэтому рекомендуется избегать использования eval для выполнения кода, полученного от пользователя или из ненадежных источников.
- **Производительность:** eval может замедлить выполнение кода из-за необходимости интерпретации строки на лету. В большинстве случаев использование других методов, таких как функции и обработка данных напрямую, может быть эффективнее и безопаснее.

Выполнение выражения:

javascript

```javascript
let x = 10;
let y = 20;
let result = eval('x + y');
console.log(result); // Выводит 30
```

Создание функции динамически:

javascript

```javascript
let funcStr = 'function add(a, b) { return a + b; }';
eval(funcStr);

console.log(add(3, 4)); // Выводит 7
```

Использование eval для выполнения JSON:

javascript

```javascript
let jsonData = '{"name": "John", "age": 30}';
let obj = eval('(' + jsonData + ')');

console.log(obj.name); // Выводит "John"
```

Ресурсы

Рекурсия и стек - ТЫК

Область видимости и замыкание - ТЫК

Каррирование - ТЫК

Объект функции, NFE - ТЫК

функции IIFE - ТЫК

синтаксис new Function - ТЫК

eval - ТЫК