# Отчет об архитектурном анализе: Управление памятью, Замыкания и Контекст исполнения (День 4)

## 1. Введение: Эпистемологический сдвиг в понимании состояния

Четвертый день программы профессиональной переподготовки знаменует собой критический рубеж в трансформации ментальной модели Senior Unity-разработчика при переходе к архитектуре веб-ориентированных систем. Если предыдущие этапы были посвящены базовому синтаксису и асинхронности, то текущий модуль требует фундаментального пересмотра концепции **состояния (state)** и **времени жизни данных (lifetime)**. В экосистеме.NET CLR (Common Language Runtime) разработчик привык к строгому, детерминированному миру: стек (stack) предназначен для быстрых, короткоживущих типов значений, куча (heap) — для долгоживущих ссылочных типов, а компилятор Roslyn выполняет роль строгого контролера, преобразующего высокоуровневые абстракции в предсказуемый Intermediate Language (IL).1

В среде выполнения JavaScript, в частности в движке V8 (Google Chrome, Node.js), эти границы размываются. То, что в C# выглядит как локальная переменная на стеке, в JavaScript может мгновенно и неявно быть "повышено" (promoted) до объекта в куче через механизм замыкания (closure). Для архитектора, проектирующего высоконагруженные системы, это различие не является семантическим нюансом; это вопрос производительности, управления памятью и предсказуемости поведения приложения. Непонимание того, как V8 аллоцирует контексты исполнения (Execution Contexts), приводит к двум классическим проблемам: "утечкам памяти через замыкания" (memory leaks via closures) и деоптимизации кода (deoptimization), когда движок вынужден отказываться от сгенерированного машинного кода в пользу медленной интерпретации.

Данный аналитический отчет представляет собой исчерпывающее теоретическое руководство, деконструирующее механику замыканий, управление областью видимости и функциональные паттерны в JavaScript через призму сравнения с архитектурой.NET. Мы спустимся на уровень байт-кода V8 (Ignition) и IL-кода CLR, чтобы увидеть, как идентичные синтаксические конструкции порождают фундаментально разные структуры данных в памяти. Анализ охватывает не только "как" работает код, но и "почему" архитекторы V8 приняли те или иные решения, и какие последствия это несет для Enterprise-разработки.1

## 2. Архитектура лексического окружения и механика захвата переменных

### 2.1. Дихотомия стека и кучи: V8 против CLR

В традиционном программировании на C/C++ или C# стек — это непрерывная область памяти, где хранятся локальные переменные функции. Доступ к ним осуществляется через смещение относительно указателя стека (stack pointer), что является чрезвычайно быстрой операцией. Когда функция завершается, указатель стека сдвигается обратно, и память считается освобожденной. В C# Value Types (struct, int, enum) живут на стеке, если они не являются частью класса или не захвачены лямбда-выражением.

В JavaScript, исполняемом движком V8, концепция "локальной переменной" сложнее. Хотя V8 использует системный стек для управления потоком исполнения (frames), сами переменные (Variable Binding) могут существовать в двух состояниях:

1. **Stack-allocated:** Если движок (Parser и Scope Analysis) может доказать, что переменная никогда не используется внутренней функцией (замыканием) и не покидает область видимости, она может быть аллоцирована на стеке или в регистре виртуальной машины.
2. **Context-allocated (Heap-allocated):** Если переменная "захватывается" (captured) любой внутренней функцией, она принудительно переносится в специальный объект кучи, называемый **Context**.

Это фундаментальное различие. В C# разработчик явно видит разницу между struct (стек) и class (куча). В JS это решение принимает движок неявно. Архитектор должен понимать, что создание любой функции внутри другой функции потенциально переводит локальные переменные родителя из категории "быстрый доступ через регистр" в категорию "медленный доступ через указатель в куче".2

### 2.2. Механизм замыкания в C#: Анатомия DisplayClass

Чтобы понять отличие, рассмотрим сначала, как C# реализует замыкания. Компилятор C# (Roslyn) использует технику, называемую "closure lowering" (понижение замыкания). Замыкание в.NET — это не магия времени выполнения, а синтаксический сахар, раскрываемый на этапе компиляции.

Рассмотрим код:

C#

public Func<int> CreateCounter() {  
 int count = 0;  
 return () => count++;  
}

При компиляции в IL этот код преобразуется в структуру классов. Компилятор создает скрытый класс, который в терминологии разработчиков компилятора часто называют DisplayClass (хотя точное имя генерируется, например, <>c\_\_DisplayClass0\_0).4

Процесс трансформации выглядит следующим образом:

1. **Создание класса:** Генерируется класс <>c\_\_DisplayClass0\_0.
2. **Повышение поля:** Локальная переменная count удаляется из метода CreateCounter и становится публичным полем count внутри сгенерированного класса.
3. **Перенос метода:** Лямбда-выражение () => count++ становится методом экземпляра этого класса.
4. **Инстанцирование:** Внутри CreateCounter вместо выделения места на стеке под int, создается экземпляр класса new <>c\_\_DisplayClass0\_0(). Это размещение в куче (heap allocation).

Таким образом, "замыкание" в C# — это просто объект, который удерживает состояние. Ссылка на этот объект передается делегату. Это объясняет, почему переменные живут дольше, чем вызвавший их метод: пока существует делегат, существует и ссылка на экземпляр DisplayClass, и Garbage Collector (GC) не может его собрать.7

### 2.3. Механизм замыкания в V8: Цепочка контекстов (Scope Chain)

В JavaScript (V8) нет этапа предварительной компиляции классов в том же смысле. Вместо генерации статических классов V8 использует динамическую структуру данных, называемую **Context** (реализуется в C++ как класс Context, наследующий от FixedArray).2

Когда функция вызывается в JS, V8 создает:

1. **Execution Context:** Содержит состояние текущего исполнения (this, pointer to code).
2. **Lexical Environment (Scope):** Это и есть механизм хранения переменных.

Если функция содержит внутренние функции, V8 создает объект Context в куче. Этот объект похож на массив, где каждый "слот" соответствует переменной.

Важнейшее отличие от C# заключается в цепочке (Chaining).

В C# DisplayClass обычно плоский (или содержит ссылку на другой DisplayClass при сложных вложениях). В V8 каждый Context имеет ссылку на Previous Context (родительский).

#### 2.3.1. Структура Context в памяти

Объект Context в V8 содержит:

* Ссылку на родительский контекст (для реализации Scope Chain).
* Ссылку на глобальный объект (в корне цепочки).
* Слоты для переменных (Context Slots).

Когда внутренняя функция обращается к переменной x, движок V8 не просто ищет её "где-то". Он выполняет поиск по цепочке контекстов. Оптимизирующий компилятор TurboFan пытается заменить этот поиск прямым доступом по индексу (LdaContextSlot), если глубина вложенности и индекс слота известны (Static Scope Analysis).9

| **Характеристика** | **C# Closure (DisplayClass)** | **JS Closure (Context Object)** |
| --- | --- | --- |
| **Время создания** | Compile-time (Class definition), Runtime (Instantiation) | Runtime (Context allocation) |
| **Структура памяти** | Объект класса с полями | FixedArray (похож на массив) со слотами |
| **Доступ к переменным** | ldfld (Load Field) IL инструкция | LdaContextSlot байт-код (индекс в массиве) |
| **Оверхед** | Аллокация одного объекта на scope | Аллокация Context объекта + Closure объекта |
| **Влияние на GC** | Стандартное управление поколением | Участие в Mark-Sweep (Old Gen) или Scavenger (New Gen) |

### 2.4. Проблема "общего" замыкания и утечки памяти

Специфическая проблема V8, которую должен понимать.NET-разработчик, — это гранулярность захвата. В старых версиях V8 (и многих других JS-движках) замыкание удерживало ссылку на *весь* лексический контекст, даже если использовало только одну переменную.

Представим ситуацию:

JavaScript

function heavyComputation() {  
 const hugeData = new Array(1000000).fill('data');  
 const smallCounter = 0;  
  
 return function increment() {  
 // Мы используем только smallCounter  
 return smallCounter + 1;  
 };  
}

В идеальном мире (и в C# благодаря анализу потока данных) hugeData должен быть уничтожен GC, так как он не используется возвращаемой функцией. Однако, если hugeData и smallCounter попадают в один и тот же объект Context, то ссылка на функцию increment будет удерживать весь Context, а значит, и hugeData останется в памяти.8

Современный V8 (TurboFan) использует агрессивный анализ (Scope Analysis), чтобы исключить неиспользуемые переменные из контекста. Если переменная hugeData не упоминается во внутренних функциях, она останется на стеке (или будет оптимизирована) и не попадет в кучу Context. Однако, стоит добавить хотя бы одну *другую* неиспользуемую функцию, которая ссылается на hugeData, и весь контекст будет "заражен", удерживая память. Это поведение существенно отличается от детерминированного анализа "используемых полей" при генерации DisplayClass в C#.12

## 3. Анализ проблемы цикла: var против let и эволюция стандартов

Для C#-разработчика, пережившего переход с.NET 2.0 на 4.0/5.0, проблема "захвата переменной цикла" является болезненно знакомой. В JavaScript эта проблема существовала десятилетиями и была решена только с внедрением ES6 (ECMAScript 2015).

### 3.1. Архитектура ошибки "3, 3, 3"

Классический пример, демонстрирующий разницу в управлении памятью:

JavaScript

// Legacy JS (var)  
for (var i = 0; i < 3; i++) {  
 setTimeout(() => console.log(i), 100);  
}  
// Output: 3, 3, 3

В C# до версии 5.0 поведение foreach было аналогичным. Причина кроется в области видимости переменной. Ключевое слово var в JS (как и переменная цикла в старом C# for) имеет **функциональную область видимости (Function Scope)**. Это означает, что переменная i создается *один раз* при входе в функцию. Физически в памяти (в стеке или контексте) существует только один слот для i. Каждая итерация цикла обновляет значение в *этом же* слоте.

Все три созданные лямбды (замыкания) захватывают ссылку на *один и тот же* Context, в котором лежит i. К моменту выполнения setTimeout, цикл завершился, и значение в слоте i равно 3.

### 3.2. Блочная область видимости let: V8 implementation

С введением let (Block Scope) V8 меняет семантику кардинально. Спецификация ECMAScript требует, чтобы для каждой итерации цикла создавалась *новая* лексическая привязка (binding).

JavaScript

// Modern JS (let)  
for (let i = 0; i < 3; i++) {  
 setTimeout(() => console.log(i), 100);  
}  
// Output: 0, 1, 2

Для архитектора важно понимать цену этого "сахара". Внутри движка V8 это реализуется следующим образом: при каждой итерации создается новый объект Context (или новый блок в текущем контексте, в зависимости от оптимизации), и значение переменной копируется из предыдущей итерации в новую.

Это означает, что цикл на 10,000 итераций с замыканием внутри при использовании let создает 10,000 независимых контекстов в куче. В C# 5.0 изменение поведения foreach также привело к созданию новых экземпляров переменных, но благодаря оптимизациям CLR и структуре стека это часто дешевле, чем создание полноценных объектов Context в V8.4

**Рекомендация архитектора:** Если вы работаете с высоконагруженными циклами (например, обработка пикселей или физика в браузере), бездумное создание замыканий внутри цикла с let может вызвать "GC thrashing" (чрезмерную работу сборщика мусора). В таких случаях, паттерны без замыканий или переиспользование объектов могут быть предпочтительнее.

## 4. Ключевое слово this: Потеря контекста и методы привязки

В C#, this — это неизменяемая ссылка на текущий экземпляр класса. Это указатель, неявно передаваемый первым аргументом в любой нестатический метод. Он надежен и предсказуем. В JavaScript this — это параметр вызова, определяемый **динамически** в момент исполнения.

### 4.1. Четыре правила привязки V8

Понимание this требует знания того, как V8 определяет "Receiver" (получателя вызова).

1. **Default Binding:** Вызов функции func(). В строгом режиме ("use strict") this будет undefined. В "небрежном" (sloppy) режиме — глобальный объект (window или global).
2. **Implicit Binding:** Вызов метода объекта obj.func(). V8 видит, что функция вызвана в контексте объекта, и устанавливает this = obj.
3. **Explicit Binding:** Использование .call(ctx), .apply(ctx) или .bind(ctx). Программист явно указывает V8, какой объект использовать.
4. **new Binding:** Вызов с оператором new. V8 создает новый пустой объект, привязывает к нему this и выполняет функцию как конструктор.11

### 4.2. Архитектурное сравнение: Delegates vs. Function Objects

Для C#-разработчика делегат Action action = myObj.Method; — это безопасная капсула. Внутри MulticastDelegate хранятся два ключевых поля: \_target (ссылка на объект myObj) и \_methodPtr (указатель на метод).18 При вызове делегата CLR знает, на каком объекте вызывать метод.

В JavaScript:

JavaScript

let action = myObj.method;  
action(); // Потеря контекста!

Здесь action получает ссылку только на *функцию* (код), но не на объект myObj. Связь разрывается. При вызове action() срабатывает правило Default Binding, и this становится undefined (или global).

### 4.3. Механика .bind() и BoundFunction

Чтобы эмулировать поведение C# делегата, используется myObj.method.bind(myObj).

Внутренне V8 реализует это через создание специального экзотического объекта — BoundFunction.

Структура BoundFunction в куче V8 содержит:

* ]: Ссылка на оригинальную функцию.
* ]: Ссылка на объект, который будет this.
* ]: Список частично примененных аргументов (для каррирования).

Анализ производительности:

Вызов bind в JavaScript дороже, чем создание делегата в C# (относительно), и, что важнее, вызов "привязанной" функции исторически был значительно медленнее обычного вызова из-за дополнительной индирекции (прослойки). Современный TurboFan (оптимизирующий компилятор V8) научился встраивать (inline) и оптимизировать BoundFunction, но это все еще требует "разогрева" кода. Стрелочные функции (() => this.method()) часто предоставляют более производительную альтернативу, так как они захватывают this лексически (как переменную в замыкании), избегая механики динамического this.21

## 5. Утечки памяти: Специфика DOM и Замыканий

C#-разработчик привык к тому, что утечки памяти чаще всего связаны с подписками на события (event +=), которые удерживают объекты. В веб-разработке добавляется новый слой сложности: **DOM (Document Object Model)**.

### 5.1. Detached DOM Tree (Отсоединенное дерево DOM)

Это уникальный класс утечек, не имеющий прямого аналога в бэкенд-разработке. DOM-дерево — это двусвязный граф. Родитель ссылается на детей, дети — на родителя и соседей.

Если JS-код (например, замыкание) удерживает ссылку на один маленький элемент <span> (например, для обновления текста), а этот <span> является частью огромной таблицы, которая была удалена из документа (body.removeChild(table)), то вся таблица останется в памяти.

Сборщик мусора V8 (Orinoco) видит, что замыкание ссылается на span. span через parentNode ссылается на td, тот на tr, тот на tbody, и так далее до корня удаленной таблицы. Весь граф объектов остается "живым" (reachable), хотя пользователь его не видит. Это называется "Detached DOM Tree" leak.13

В Unity аналогом можно считать удержание ссылки на Transform уничтоженного GameObject, что не дает выгрузить ресурсы, связанные с иерархией, но в браузере масштаб проблемы часто больше из-за динамической природы UI.

### 5.2. Практика профилирования

Для диагностики таких проблем используется **Chrome DevTools Memory Profiler**. Ключевой навык архитектора — умение снимать **Heap Snapshots** и использовать режим "Comparison" (Сравнение) или "Retainers" (Удерживающие пути).

* **Distance:** Расстояние от корня GC (GC Root) до объекта.
* **Shallow Size:** Размер самого объекта.
* **Retained Size:** Размер памяти, который будет освобожден, если этот объект будет удален.

Если вы видите HTMLDivElement с огромным Retained Size в статусе "Detached", и в Retainers указан Context (замыкание) — это ваш клиент. Решением часто является использование WeakMap или WeakRef для хранения связей с DOM-элементами, или тщательная зачистка (nullifying) ссылок при уничтожении компонентов.26

## 6. Функциональные паттерны и производительность

Переход с C# LINQ на функциональные методы массивов JS (map, filter, reduce) требует понимания разницы между **ленивыми (lazy)** и **энергичными (eager)** вычислениями.

### 6.1. LINQ: Deferred Execution

В C# выражение:

C#

var query = collection.Where(x => x > 10).Select(x => x \* 2);

Не выполняет никакой работы. Оно создает объект-запрос (State Machine). Итерация происходит только при вызове foreach или .ToList(). Данные проходят через конвейер по одному элементу (pull-model). Память под промежуточные коллекции не выделяется (кроме итераторов).28

### 6.2. JS Array Methods: Eager Execution & Intermediate Arrays

В стандартном JavaScript (до появления Iterator Helpers) методы массивов энергичны:

JavaScript

const result = array  
 .filter(x => x > 10) // Проход 1: Создает НОВЫЙ массив в памяти  
 .map(x => x \* 2); // Проход 2: Создает ЕЩЕ ОДИН массив

Для массива из 100,000 объектов это катастрофа производительности. V8 вынужден аллоцировать память под промежуточный массив после filter, заполнять его, затем читать из него для map и сразу же выбрасывать. Это создает огромное давление на Garbage Collector (Minor GC / Scavenger).29

### 6.3. Elements Kinds: Скрытые оптимизации V8

В C# массив int всегда содержит int. В JS массив `` может содержать что угодно. V8 пытается оптимизировать хранение, отслеживая "Elements Kind" (Тип элементов).

* PACKED\_SMI\_ELEMENTS: Массив плотный (без дыр), содержит только малые целые числа (Smi). Самый быстрый доступ.
* PACKED\_DOUBLE\_ELEMENTS: Содержит числа с плавающей точкой.
* PACKED\_ELEMENTS: Содержит объекты или разнородные типы.
* HOLEY\_\*: Массив с "дырами" (пропущенными индексами). Значительно медленнее.31

**Анти-паттерн:** Инициализация массива через new Array(100) создает "дырявый" массив (HOLEY\_SMI\_ELEMENTS). Добавление null или строки в массив чисел вызывает необратимую деградацию типа массива до более общего (PACKED\_ELEMENTS), что замедляет все операции с ним. Для Unity-разработчика это сравнимо с Boxed-массивом object против int.

### 6.4. Оптимизация: Трансдюсеры и Итераторы

Чтобы достичь производительности LINQ в JS, архитектор должен использовать:

1. **Цепочку через reduce:** Выполнение фильтрации и маппинга за один проход.
2. **Генераторы (function\*)**: Они возвращают итераторы, которые ленивы по своей природе. Это прямой аналог IEnumerable в C# (yield return).
3. **Iterator Helpers (Proposal):** Новые методы .map(), .filter() у итераторов, которые внедряются в стандарты и V8, позволяют писать ленивый код нативно.32

## 7. Каррирование (Currying) и Частичное применение

В функциональном программировании эти техники позволяют создавать специализированные функции из общих.

### 7.1. C# vs JS Реализация

В C# каррирование реализуется через вложенные Func<>, что создает цепочку делегатов.

C#

Func<int, Func<int, int>> add = x => y => x + y;

В JS это вложенные стрелочные функции:

JavaScript

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

### 7.2. Влияние на память в V8

Каждый вызов каррированной функции add(5) создает новое замыкание (Context + Closure object), удерживающее переменную x. Если вы используете глубокое каррирование в "горячем коде" (например, в цикле рендеринга requestAnimationFrame), вы генерируете огромное количество мусора.

Для V8 более дружелюбным является частичное применение через замыкание или bind (если он оптимизирован), но идеальным для производительности остается "мономорфный" вызов функции с полным набором аргументов. Каррирование создает полиморфизм и усложняет работу Inline Caches (IC) — механизма кэширования типов в V8.34

## 8. Заключение

Переход на клиентскую сторону требует от C#-архитектора не просто изучения нового синтаксиса, а глубокого понимания того, как V8 управляет памятью и исполнением.

Ключевые выводы 4-го дня:

1. **Замыкание — это объект.** Любая функция, покидающая свой scope, тянет за собой Context в куче.
2. **Scope Chain физичен.** Это связанный список объектов в памяти. Утечка памяти — это часто ссылка на Context, который удерживает ненужный DOM.
3. **this динамичен.** Это требует дисциплины или использования стрелочных функций/bind для сохранения контекста.
4. **Массивы не типизированы, но оптимизированы.** Поддержание PACKED\_SMI и избегание промежуточных аллокаций (map -> reduce) критично для производительности.

Понимание этих механизмов позволяет писать JavaScript-код, который не просто "работает", а работает эффективно, предсказуемо и масштабируемо, приближаясь по характеристикам к оптимизированному коду на C#.

### Приложение: Таблица сравнительного анализа (.NET vs V8)

| **Концепция** | **.NET CLR (C#)** | **V8 Engine (JavaScript)** |
| --- | --- | --- |
| **Реализация замыкания** | Компилятор генерирует класс DisplayClass. Поля класса хранят захваченные переменные. | Движок создает объект Context в куче. Слоты контекста хранят переменные. Цепочка контекстов. |
| **Аллокация переменных** | Явное разделение: Value Types (Stack), Reference Types (Heap). | Эвристическое (Escape Analysis): Стек, если возможно; Контекст (Куча), если захвачено замыканием. |
| **Захват в циклах** | До C# 5.0: одна переменная. C# 5.0+: новая переменная на итерацию (для foreach). | var: одна переменная (function scope). let/const: новая привязка на итерацию (block scope). |
| **Привязка метода (this)** | Делегат жестко хранит ссылку на Target (объект) и MethodPtr. | this вычисляется при вызове. bind() создает обертку BoundFunction. |
| **Коллекции** | IEnumerable (LINQ): Ленивое выполнение, Pull-model. | Array.prototype: Энергичное выполнение, создание промежуточных массивов. Генераторы — ленивые. |
| **Управление памятью** | Generational GC (Gen0, Gen1, Gen2, LOH). Детерминированные финализаторы. | Generational GC (Scavenger, Mark-Sweep-Compact). Слабые ссылки (WeakRef/WeakMap) критичны для DOM. |

### Цитируемые материалы

1 Uploaded Document: "Общий неделя 9"

2 Chromium Source: contexts.h (V8 internals)

4 Blog: Magic Behind Closures (C# DisplayClass)

3 Blog: JS Scoping and V8 Contexts

31 V8 Blog: Elements Kinds

9 Chromium Source: js-native-context-specialization.cc

25 StackOverflow: Detached DOM Tree Memory Leak

36 NodeBook: V8 Engine Intro (Deoptimization)

#### Источники

1. Общий неделя 9
2. src/contexts.h - v8/v8 - Git at Google, дата последнего обращения: декабря 6, 2025, <https://chromium.googlesource.com/v8/v8/+/1.3.13.1/src/contexts.h>
3. Everything I Wish I Knew About Javascript Scoping A Week Ago - Jesse Duffield, дата последнего обращения: декабря 6, 2025, <https://jesseduffield.com/Everything-I-Wish-I-Knew-About-JS-Scoping-A-Week-Ago/>
4. Magic behind closures - Tearth's homepage, дата последнего обращения: декабря 6, 2025, <https://tearth.dev/posts/magic-behind-closures/>
5. How a micro-optimisation in the .NET compiler broke our web application at runtime | by Matt Thompson | Spektrix Engineering | Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/spektrix-engineering/how-a-micro-optimisation-in-the-net-compiler-broke-our-web-application-at-runtime-7f418e5ded90>
6. does C# and VB lambdas have \*\*scope chain\*\* issues similar to javascript?, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/5923772/does-c-sharp-and-vb-lambdas-have-scope-chain-issues-similar-to-javascript>
7. A Simple Explanation of C# Closures | Closures, lambdas, and first-class functions don't have to be so hard., дата последнего обращения: декабря 6, 2025, <https://www.simplethread.com/c-closures-explained/>
8. How do closures work behind the scenes? (C#) - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/1928636/how-do-closures-work-behind-the-scenes-c>
9. js-native-context-specialization.cc - Chromium Code Search, дата последнего обращения: декабря 6, 2025, <https://source.chromium.org/chromium/chromium/src/+/main:v8/src/compiler/js-native-context-specialization.cc;l=1482;drc=22629fc9a7e45cf5e4c691db371f69f176318f11>
10. V8 Bytecode Reference - Kuter Dinel's blog, дата последнего обращения: декабря 6, 2025, <https://kuterdinel.com/v8-bytecode-reference.html>
11. Closures, Objects, and the Fauna of the Heap | Many But Finite, дата последнего обращения: декабря 6, 2025, <https://manybutfinite.com/post/closures-objects-heap/>
12. JavaScript Closures Concerning Unreferenced Variables - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/24468713/javascript-closures-concerning-unreferenced-variables>
13. Memory Leaks in DOM Elements and Closures | by Rahul Jindal - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@rahul.jindal57/memory-leaks-in-dom-elements-and-closures-b3452f129dac>
14. Closing over the loop variable considered harmful, part two | Fabulous adventures in coding, дата последнего обращения: декабря 6, 2025, <https://ericlippert.com/2009/11/16/closing-over-the-loop-variable-considered-harmful-part-two/>
15. C# 5.0 Breaking Changes - Didactic Code, дата последнего обращения: декабря 6, 2025, <https://davefancher.com/2012/11/03/c-5-0-breaking-changes/>
16. Grokking V8 closures for fun (and profit?), дата последнего обращения: декабря 6, 2025, <https://mrale.ph/blog/2012/09/23/grokking-v8-closures-for-fun.html>
17. this - JavaScript | MDN, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>
18. Delegate.Target Property (System) - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.delegate.target?view=net-9.0>
19. C# — Delegate - by Serhat Zor - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@serhat21zor/c-delegate-fc74d2bf6596>
20. Why one delegate is faster than the other? - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/71920613/why-one-delegate-is-faster-than-the-other>
21. Understand JavaScript more deeply by writing a bind function | by adamisom - Medium, дата последнего обращения: декабря 6, 2025, <https://medium.com/@adambomb/understand-javascript-more-deeply-by-writing-a-bind-function-8b619b242dcc>
22. V8: Behind the Scenes (February Edition feat. A tale of TurboFan) - Benedikt Meurer, дата последнего обращения: декабря 6, 2025, <https://benediktmeurer.de/2017/03/01/v8-behind-the-scenes-february-edition/>
23. Function.prototype.bind() - JavaScript - MDN Web Docs - Mozilla, дата последнего обращения: декабря 6, 2025, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind>
24. Record heap snapshots | Chrome DevTools, дата последнего обращения: декабря 6, 2025, <https://developer.chrome.com/docs/devtools/memory-problems/heap-snapshots>
25. Locating detached DOM tree memory leak - javascript - Stack Overflow, дата последнего обращения: декабря 6, 2025, <https://stackoverflow.com/questions/21226646/locating-detached-dom-tree-memory-leak>
26. How to Avoid Memory Leaks in JavaScript Event Listeners - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/alex_aslam/how-to-avoid-memory-leaks-in-javascript-event-listeners-4hna>
27. Fix memory problems | Chrome DevTools, дата последнего обращения: декабря 6, 2025, <https://developer.chrome.com/docs/devtools/memory-problems>
28. Deferred execution and lazy evaluation - LINQ to XML - .NET - Microsoft Learn, дата последнего обращения: декабря 6, 2025, <https://learn.microsoft.com/en-us/dotnet/standard/linq/deferred-execution-lazy-evaluation>
29. Avoid intermediate arrays (filter/map) to make Javascript fast - DEV Community, дата последнего обращения: декабря 6, 2025, <https://dev.to/alexander-nenashev/avoid-intermediate-arrays-in-js-5bkg>
30. Filter/map/reduce are too slow, any tool to replace them with loops? : r/javascript - Reddit, дата последнего обращения: декабря 6, 2025, <https://www.reddit.com/r/javascript/comments/7dk699/filtermapreduce_are_too_slow_any_tool_to_replace/>
31. Elements kinds in V8, дата последнего обращения: декабря 6, 2025, <https://v8.dev/blog/elements-kinds>
32. tc39/proposal-iterator-helpers: Methods for working with iterators in ECMAScript - GitHub, дата последнего обращения: декабря 6, 2025, <https://github.com/tc39/proposal-iterator-helpers>
33. Getting Started with JavaScript Iterator Helpers - OpenReplay Blog, дата последнего обращения: декабря 6, 2025, <https://blog.openreplay.com/getting-started-javascript-iterator-helpers/>
34. What is the advantage of currying? - Software Engineering Stack Exchange, дата последнего обращения: декабря 6, 2025, <https://softwareengineering.stackexchange.com/questions/185585/what-is-the-advantage-of-currying>
35. Currying vs partial function application | Jon Skeet's coding blog, дата последнего обращения: декабря 6, 2025, <https://codeblog.jonskeet.uk/2012/01/30/currying-vs-partial-function-application/>
36. Inside the V8 JavaScript Engine - NodeBook, дата последнего обращения: декабря 6, 2025, <https://www.thenodebook.com/node-arch/v8-engine-intro>