



## **A Comparative Study of Garbage Collection in JavaScript, Kotlin, and C#**

PROGRAMMING LANGUAGE MINI PROJECT1

**Instructor:** Dr. Kwankamol Nongpong

**Members:**

Phanthira Kositjaroenkul (6630003)

Kaung Khant Lin (6540131)

Thit Lwin Win Thant (6540122)

<b>Introduction</b>	<b>3</b>
<b>Comparative Analysis of Garbage Collection in JavaScript, Kotlin, and C#</b>	<b>3</b>
1. Garbage Collection Strategy	3
JavaScript	3
Kotlin (JVM)	4
C#	4
2. Manual vs. Automatic Garbage Collection Control	5
3. Runtime Environment	5
4. Performance Goals and Trade-offs	6
5. Special Features and Tools	6
JavaScript	6
Kotlin (JVM)	7
C#	7
6. Memory Leak Risks and Developer Responsibilities	8
<b>Summary Comparison</b>	<b>8</b>
<b>Insights of Garbage Collection in JavaScript, Kotlin, and C#</b>	<b>9</b>
1. Implications for Learning and Productivity	9
2. The Future: Smarter GC and Static Analysis	9
3. Trade-off Between Memory and Performance is Unavoidable	10
<b>Summary Table: Thoughts and Implications of Garbage Collection</b>	<b>10</b>
<b>Conclusion</b>	<b>11</b>
<b>References</b>	<b>12</b>
JavaScript	12
Kotlin	12
C#	12

# Introduction

This report analyzes garbage collection mechanisms in **JavaScript, Kotlin and C#**—three modern programming languages that represent diverse runtime environments and memory management models.

- **JavaScript** is a high-level, interpreted language widely used in web development. Its garbage collection is managed by browser engines like V8, making it an ideal example of automatic memory management in event-driven environments.
- **Kotlin**, a modern language that runs primarily on the **Java Virtual Machine (JVM)**, is designed to interoperate fully with Java while offering a safer and more concise syntax. Analyzing Kotlin allows us to explore GC behavior in the JVM ecosystem while highlighting newer language design choices.
- **C#** runs on the .NET runtime and features a robust and mature **Generational Garbage Collector**. As a statically typed, object-oriented language used for building desktop, web, and enterprise applications, it offers insights into managed memory in performance-critical systems.

## Comparative Analysis of Garbage Collection in JavaScript, Kotlin, and C#

### 1. Garbage Collection Strategy

Each language/runtime uses a different variant of garbage collection optimized for its environment, though all rely conceptually on **reachability-based collection**, not simple reference counting.

#### JavaScript

JavaScript engines (like V8 in Chrome/Node.js, SpiderMonkey in Firefox) primarily implement a **Mark-and-Sweep** algorithm with modern enhancements such as **generational and incremental collection**. The collector starts from known “roots”

such as global objects (`window` or `global`), traverses all reachable objects, and reclaims memory from those not reached.

This model handles **circular references** without issue, as reachability is the criterion—not reference count.

```
function createCycle() {  
    let a = {};  
    let b = {};  
    a.ref = b;  
    b.ref = a;  
    return null; // Both collectible once out of scope  
}
```

## Kotlin (JVM)

Kotlin relies on the Java Virtual Machine (JVM) for garbage collection. The JVM uses various **generational garbage collectors**—such as G1, ZGC, or Shenandoah—depending on configuration and version. These collectors divide the heap into **young and old generations**, performing frequent, fast collections on short-lived (young) objects while collecting long-lived ones less often. Internally, many still follow a **Mark-and-Sweep** or **Mark-Compact** approach.

Circular references pose no problem, as unreachable cycles are collected.

```
class Node(var next: Node?)  
fun main() {  
    val a = Node(null)  
    val b = Node(a)  
    a.next = b  
    // Both a and b are eligible for GC when unreachable  
}
```

## C#

C#'s .NET runtime also uses a **generational garbage collection** system built on Mark-and-Sweep with compaction. It categorizes memory into **three generations (0, 1, 2)**. Short-lived objects are typically collected in Gen 0, and long-lived ones are promoted to higher generations. The collector handles circular references efficiently using reachability.

```
class Node { public Node Next; }
void CreateCycle() {
    Node a = new Node();
    Node b = new Node();
    a.Next = b;
    b.Next = a;
    // Collected when no references remain
}
```

## 2. Manual vs. Automatic Garbage Collection Control

All three runtimes use **automatic garbage collection** by default. However, only C# offers a mechanism for **manual triggering** via `GC.Collect()`—though it is discouraged for most scenarios.

Language	Automatic GC	Manual Trigger
JavaScript	Yes	No
Kotlin (JVM)	Yes	No
C# (.NET)	Yes	Yes

In JavaScript and Kotlin, there is **no API to trigger garbage collection**, as the runtime determines optimal collection times. Forcing GC is seen as a misstep in these environments because it can disrupt performance assumptions.

```
// C# manual GC trigger
GC.Collect(); // Not recommended in most cases
```

Even in .NET, manual collection is generally reserved for specific edge cases, such as releasing large object graphs immediately before a known idle period.

## 3. Runtime Environment

Each language operates in a distinct runtime environment, which informs the design and behavior of its garbage collector.

- **JavaScript** executes in browser engines (e.g., V8, SpiderMonkey) or server-side environments like Node.js. These environments prioritize **responsiveness**, with GCs optimized for low pause times and frequent, incremental collection.
- **Kotlin** runs on the **JVM**, leveraging decades of GC research and support for enterprise-scale systems. JVM collectors are tunable, and different collectors optimize for **throughput, pause time, or footprint** depending on the application.
- **C#** runs on the **Common Language Runtime (CLR)** within .NET. It supports both client and server GC modes, with features like **concurrent GC, background sweeping, and heap compaction**. Its design balances **low latency and high throughput**, depending on configuration.

## 4. Performance Goals and Trade-offs

Each GC strategy reflects trade-offs tailored to the language's use cases.

- **JavaScript** prioritizes **UI responsiveness** and low pause times over raw throughput. This makes it suitable for interactive web applications but can lead to memory pressure under long-lived workloads if not managed carefully.
- **Kotlin/JVM** aims for **configurable GC performance**, allowing trade-offs between latency and throughput. G1 balances both; ZGC and Shenandoah prioritize **low pause times**, making them useful for real-time or latency-sensitive systems.
- **C#/.NET** is optimized for **scalable server applications**, offering tuning for desktop (low latency) or server (throughput) modes. It performs background collection to reduce GC pauses in multi-core systems.

## 5. Special Features and Tools

Each ecosystem provides tools for advanced memory control, weak references, and diagnostics—though differing in scope and accessibility.

### JavaScript

- **WeakMap, WeakSet**: Allow weakly-referenced keys; objects can be collected when no other references exist.

- **WeakRef and FinalizationRegistry**: Provide lower-level hooks into GC behavior for caching and cleanup logic.

```
let cache = new WeakMap();
(function() {
  let obj = { temp: true };
  cache.set(obj, "data");
  // 'obj' becomes collectible when out of scope
})();
```

## Kotlin (JVM)

- **WeakReference and SoftReference**: Weak refs allow collection when no strong references exist; soft refs stay longer and are collected under memory pressure.
- **GC tuning**: Developers can specify the GC strategy via JVM flags (e.g., `-XX:+UseG1GC`, `-XX:+UseZGC`) and profile performance using tools like VisualVM or JFR.

```
val data = MyData("important")
val weak = WeakReference(data)
// 'data' is collectible when no strong refs remain
```

## C#

- **WeakReference<T>**: Used in caches or observer patterns to prevent retention of objects.
- **GC API controls**: `GCSettings.LatencyMode`, `GC.Collect()`, `GC.AddMemoryPressure()` give developers granular control in high-performance scenarios.
- **Resource cleanup tools**: `IDisposable`, `using`, and finalizers support safe management of unmanaged resources (distinct from GC, but essential to memory hygiene).

```
var weak = new WeakReference<MyClass>(new MyClass());  
// Object collectible when no strong reference exists
```

## 6. Memory Leak Risks and Developer Responsibilities

Despite automatic garbage collection, memory leaks still occur when objects remain **reachable** due to lingering references—even if they're no longer logically in use.

- In **JavaScript**, leaks often stem from closures, DOM event listeners, unremoved timers, or accidental globals.
- In **Kotlin**, leaks arise from static/singleton references, unregistered observers, or long-lived coroutines.
- In **C#**, leaks are frequently caused by event handler subscriptions that aren't removed, static fields, or misused global collections. Failing to call `Dispose()` on classes managing unmanaged resources can also result in leaks.

In all three ecosystems, developers are responsible for **breaking unwanted references**, clearing collections, and unregistering callbacks when objects are no longer needed.

## Summary Comparison

Aspect	JavaScript	Kotlin (JVM)	C# (.NET)
GC Algorithm	Mark-and-Sweep (+ Gen)	Generational (G1, ZGC, etc.)	Generational (0/1/2, compacting)
Manual GC Trigger	✗ No	✗ No	✓ Yes (GC.Collect())
Runtime	Browser/Node.js engines	JVM	CLR (.NET)



Tuning Options	Limited	JVM flags, collector configs	GC settings, modes, APIs
Weak References	WeakMap, WeakRef	WeakReference, SoftReference	WeakReference<T>
Leak Causes	Closures, events, globals	Statics, coroutines, listeners	Events, statics, IDisposable

# Insights of Garbage Collection in JavaScript, Kotlin, and C#

## 1. Implications for Learning and Productivity

Languages with GC lower the barrier for newcomers and speed up prototyping and innovation, but they can also cultivate a false sense of “memory safety.” This could ironically increase the prevalence of leaks in the hands of less experienced programmers. There is a need for better developer education—tools and runtime warnings should actively nudge developers towards good patterns (e.g., warning on unremoved event handlers or uncanceled coroutines).

## 2. The Future: Smarter GC and Static Analysis

As applications grow in complexity and scale (e.g., single-page apps, large-scale backends, games), smarter, adaptive GC algorithms and deeper static analysis are needed. Improvements that connect static code analysis directly to memory usage patterns, surfacing potential leaks at compile or build time, would significantly raise application robustness.

### 3. Trade-off Between Memory and Performance is Unavoidable

Automatic GC often means trading off some raw performance for development productivity. In high-performance environments (like gaming in C# or mobile in Kotlin), developers increasingly mix memory-conscious patterns (like object pooling) with GC to meet demanding requirements.

## Summary Table: Thoughts and Implications of Garbage Collection

Aspect / Thought	Implications
Ease of Learning & Productivity	GC makes programming more accessible for beginners and accelerates development, but can create a false sense of memory safety and allow hidden leaks.
Developer Responsibility Remains	Despite GC, memory leaks can still occur; developers must manage object lifecycles, avoid lingering references, and follow good coding practices.
Platform-Specific Pitfalls	Memory leaks manifest differently on each platform (e.g., event listeners in JS, uncancelled coroutines in Kotlin, event handlers in C#); learning these is critical.
Advanced Tools (Weak References, etc.)	Advanced GC features (WeakMap/WeakSet, WeakReference) help with memory-sensitive scenarios, but can introduce subtle bugs if misused or misunderstood.
Performance vs. Productivity Trade-off	Automatic GC trades some raw performance for ease-of-use. High-performance applications may need supplementary techniques like object pooling or manual cleanup.
Need for Better Analysis & Tooling	Future improvements (e.g., better runtime diagnostics, static analysis) are needed to prevent leaks and help developers identify memory risks earlier.
GC Algorithm Adaptation & Evolution	Continued evolution of GC algorithms (smarter, low-latency, generational approaches) will be vital

	as applications become larger and more complex.
--	---

## Conclusion


In summary, garbage collection (GC) stands as a pivotal feature in modern programming languages, offering developers the ability to write safer, more maintainable code without the burden of manual memory management. Through this comparative analysis of JavaScript, Kotlin, and C#, it is evident that while all three languages rely on automatic GC to manage object lifecycles, their approaches and underlying runtime systems introduce unique characteristics, strengths, and challenges.

- **JavaScript** utilises incremental, low-pause garbage collectors designed to uphold seamless, interactive browsing experiences.
- **Kotlin**, through the JVM, employs modern generational collectors that efficiently manage memory for both quick-lived and persistent objects, supporting a wide span of applications from mobile to backend systems.
- **C#** relies on .NET's dynamic generational and concurrent collectors, enabling strong performance scaling and real-time responsiveness for everything from business software to gaming.

Recognising these distinctions is crucial for developers, as it empowers them to optimise memory management strategies and achieve reliable, high-performance applications tailored to their chosen platform.

# References

## JavaScript

-  Understand JS Garbage Collector in 4 mins
- <https://www.geeksforgeeks.org/javascript/garbage-collection-in-javascript/>
- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Memory\\_management#garbage\\_collection](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Memory_management#garbage_collection)

## Kotlin

- <https://newrelic.com/blog/best-practices/java-garbage-collection>
- <https://www.netdata.cloud/academy/java-garbage-collection/>
- <https://www.geeksforgeeks.org/java/garbage-collection-in-java/>
- <https://kotlinlang.org/docs/native-memory-manager.html>
- <https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.native.runtime/-g-c/>
- <https://moldstud.com/articles/p-a-comprehensive-comparison-of-kotlin-and-java-performance-in-real-world-native-applications>
- <https://appmaster.io/blog/kotlin-memory-management-and-garbage-collection>
- <https://dev.to/arsenikavalchuk/manual-memory-management-and-garbage-collection-in-kotlin-multiplatform-native-shared-libraries-11l3>
- <https://dev.to/arsenikavalchuk/memory-management-and-garbage-collection-in-kotlin-multiplatform-xcframework-15pa>

## C#

- <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview>
- <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/>
- <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals>
- <https://www.geeksforgeeks.org/c-sharp/garbage-collection-in-c-sharp-dot-net-framework/>
- <https://learn.microsoft.com/en-us/dotnet/api/system.weakreference?view=net-9.0>