Ramdal Events Documentation

Contents

Introduction

Ramdal Events is a high-performance serialized event system for Unity that allows you to assign and invoke methods directly from the Inspector while delivering near delegate-level, GC-free runtime performance.

Designed with flexibility, performance, and productivity in mind, Ramdal Events supports virtually any method signature and includes powerful tools such as refactor-safe Method IDs, project-wide event tracking, automatic optimization, built-in diagnostics, and much more to help you build and maintain event-driven systems with confidence.

Key Features

- Invoke virtually any method, including public, private, protected, internal, static, return-value, and multi-parameter methods.
- Near delegate-level, GC-free runtime performance through delegate-based invocation and automatic caching.
- Refactor-safe Method IDs that allow you to rename methods without breaking persistent references.
- Project-wide Events Tracker to search, filter, validate, and repair events.
- Automatic Background Compilation & Caching for maximum runtime performance.
- IL2CPP AOT Optimization to generate specialized code paths for IL2CPP builds.
- Dynamic Parameter Generator for seamless support of custom parameter types.

Getting Started

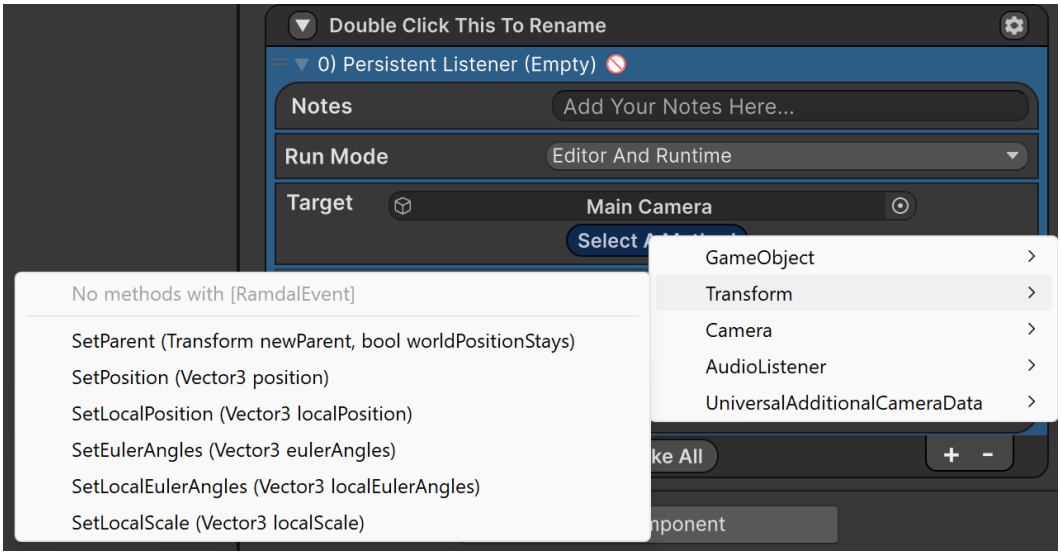
- Import the package into your Unity project.
- When Unity finishes compiling, the Get Started window will open automatically, providing quick access to the documentation, demos, and key features.
- If the window is closed, you can reopen it at any time from **Tools > Ramdal Games > Ramdal Events > Get Started**.

Accessing the Tools

All Ramdal Events tools and settings are accessible from **Tools > Ramdal Games > Ramdal Events**, or directly from the **Settings button** available on every Ramdal Event component.

Create Your First Ramdal Event

```
using RamdalGames.Events;
using UnityEngine;

Unity Script | 0 references
public class GetStarted : MonoBehaviour {
    [SerializeField] private RamdalEvent ramdalEvent;

}
```



- **Create a Ramdal Event:** Add a ***RamdalEvent*** field to your script. Don't forget to include the ***RamdalGames.Events*** namespace.
- **Rename the Event:** Double-click the event name to rename it or add custom notes.
- **Listener Notes:** Each persistent listener supports its own notes, making it easier to document complex setups.
- **Add a Listener:** Click the + button to create a new persistent listener.
- **Select a Target:** Choose the object that contains the method you want to invoke. Ramdal Events supports **MonoBehaviours, Components, ScriptableObjects, GameObjects,** and **Prefabs** as targets.
- **Choose a Method:** Open the method dropdown to select the method you want to invoke. By default, Unity's built-in actions (such as Set Active, Enable, etc.) are always available.
- **Expose Your Own Methods:** Your custom methods do not appear automatically. To make a method available in the dropdown, simply decorate it with the **[RamdalEvent]** attribute. The next section explains how to use this attribute and why it is required.

Exposing Methods with [RamdalEvent]

To expose one of your methods to Ramdal Events, simply decorate it with the **RamdalEvent** attribute, this tells the tool that the method can be exposed and serialized.

```
[RamdalEvent("Jump_ID")]
0 references
public void Jump() {

}
```

Why use the extra step?

Going with the attribute is a deliberate design choice and comes with several benefits:

- It acts as a visual indicator that the method is being used outside of your C# code, making you much less likely to accidentally remove it. This happened to me countless times with traditional event systems.
- It removes the noise by only showing the methods you want to expose.
- It's faster and safer because the system only needs to scan methods marked with the attribute.
- It also enables project-wide event tracking and validation.
- Most importantly, it lets you assign a **unique Method ID**. Ramdal Events serializes this ID together with the method signature, allowing you to safely rename your methods later without breaking existing event references.

Method IDs

The **Method ID** is a unique identifier used by Ramdal Events to identify and serialize a method. Unlike the C# method name, the Method ID is designed to remain stable, allowing you to freely rename your methods later without breaking existing event bindings.

Important to know:

- Each method should have a **unique Method ID** within the same class. **IDs do not need to be unique across your entire project**.
- The Method ID can be anything you like. It doesn't have to match the C# method name. You can use a descriptive name, a random string, numbers, a GUID, or any naming convention that fits your project. Think of it as a stable identifier rather than the method's actual name.
- You can choose **not to provide a Method ID** and the method will still work normally. However, you'll lose features such as **rename safety** and **Background Compilation & Caching**, since the system relies on stable IDs to identify methods.

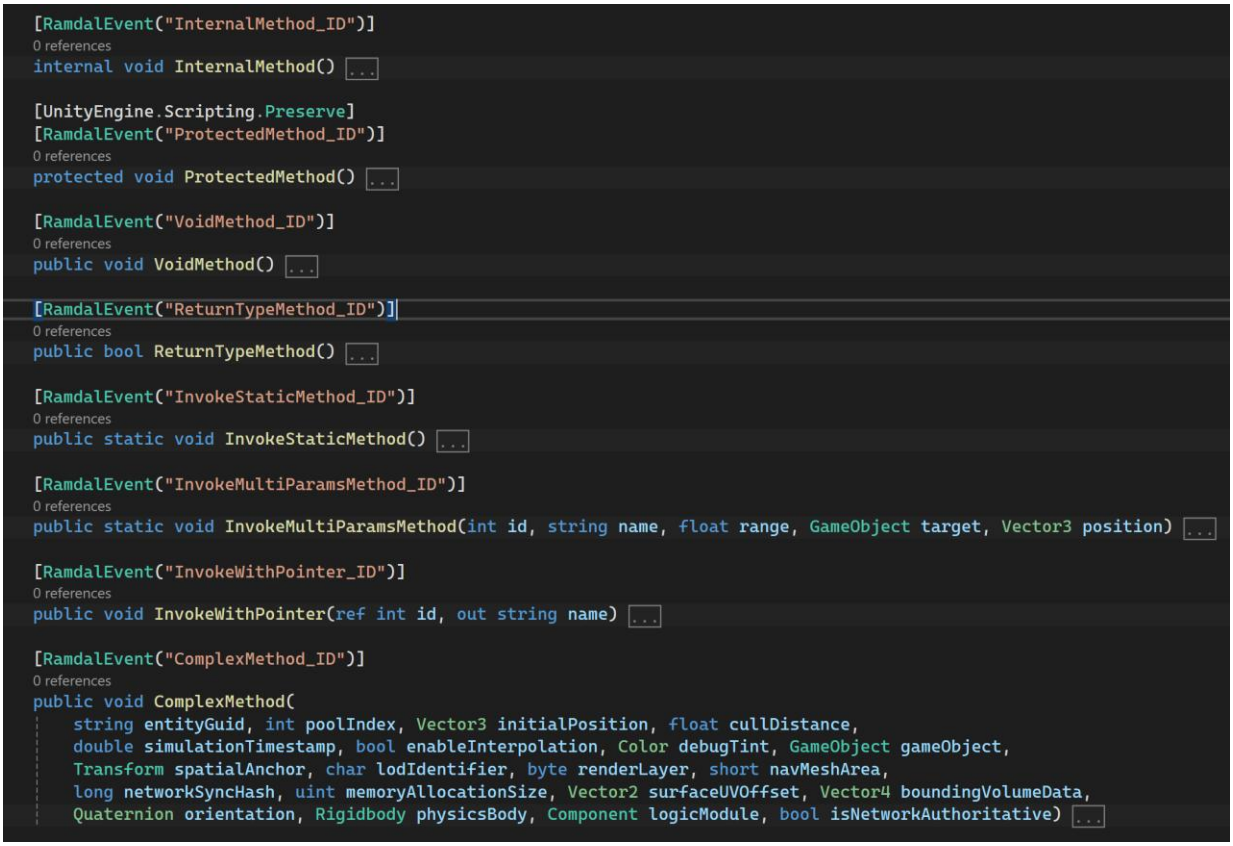
```
[RamdalEvent("PlayerJump")]
0 references
public void Jump0() { }

[RamdalEvent("Jump_Ability")]
0 references
public void Jump1() { }

[RamdalEvent("f7c9c8d1")]
0 references
public void Jump2() { }

[RamdalEvent("123456")]
0 references
public void Jump3() { }

[RamdalEvent("This method handles the player's jump.")]
0 references
public void Jump4() { }
```

- If you accidentally create duplicate Method IDs (for example by duplicating a method), Ramdal Events will immediately warn you so the issue doesn't go unnoticed.

- Once a Method ID is in use, avoid changing it. If an event still references the old ID and you change both the ID and the method name, the system can no longer identify the method and the binding will be lost.
- If you changed a Method ID, **don't worry**. Ramdal Events will warn you, and the Events Tracker provides a safe way to update every reference across your project in just a few clicks.

Supported Method Signatures

One of the biggest advantages of Ramdal Events is that it isn't limited to simple public void methods. It supports virtually any method signature you can write in C#, allowing you to expose the methods you actually want instead of writing wrapper methods.



What is supported?

- Access levels: **public, private, protected, internal, static.**
- **Any** serialized or custom type.
- **Multiple** parameters per method.
- Up to **16 parameters (Fast Invoker),** beyond that **slower fallback invoker**.
- **ref**, **out**, and **pointer-like parameters** (with slower fallback invoker)

Important Notes:

- Methods must belong to a **MonoBehaviour** or **ScriptableObject**. Scripts themselves are serialized as Editor-only **MonoScript** assets, not runtime object instances.
- **Generic methods are not supported.** You cannot bind methods like **MyMethod<T>(*T value*).** If you need to expose a generic method to Ramdal Events, you must create a non-generic wrapper method instead.
- Exceeding 16 parameters or using **ref/out** will switch the call to a slower fallback invoker automatically. This is fully supported, but should be avoided in hot paths like Update.

Runtime Usage

Just like Unity Events, Ramdal Events lets you invoke your events directly from C# using a simple runtime API. You can add and remove listeners, invoke events, initialize them ahead of time, and use generic event variants for parameterized callbacks.

Invoke()

```
Unity Script | 0 references
public class GetStarted : MonoBehaviour {
    [SerializeField] private RamdalEvent ramdalEvent;

    Unity Message | 0 references
    private void Update() {
        ramdalEvent.Invoke();
    }
```

Invoking a Ramdal Event works just like invoking a Unity Event.

When an event is invoked, Ramdal Events executes **two separate listener groups**:

1. **Persistent Listeners -** The listeners you assign through the Inspector.
2. **Runtime Listeners -** The listeners you register through C# using the runtime API.

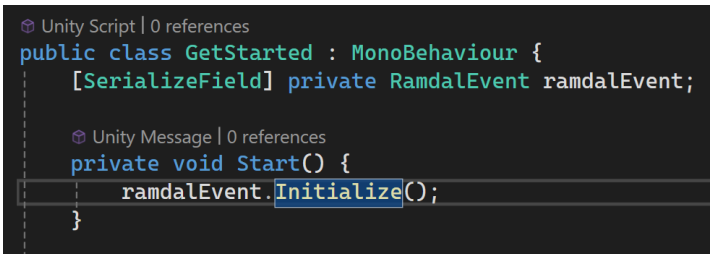
These two systems are completely independent from each other. You can use either one on its own, or combine both on the same event.

The **invocation order** is always the same:

1. **All Persistent Listeners** are invoked first.
2. **All Runtime Listeners** are invoked immediately afterwards.

The following sections cover the Runtime API in more detail, including how to add, remove, and manage runtime listeners.

Initialize()

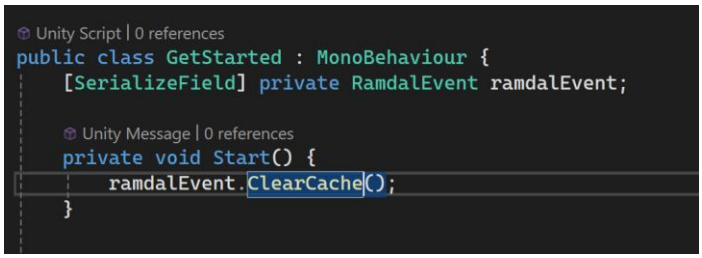


By default, Ramdal Events initializes itself automatically the first time it is invoked. This means you never have to call **Initialize()** manually.

However, if the first invocation happens during gameplay and you want to eliminate the small one-time initialization cost, you can initialize the event ahead of time.

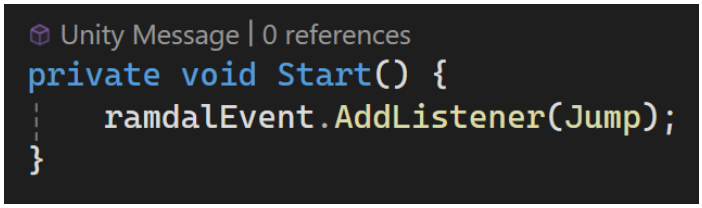
Calling **Initialize()** is completely optional. Once initialized, all subsequent invocations are ready to execute immediately.

ClearCache()



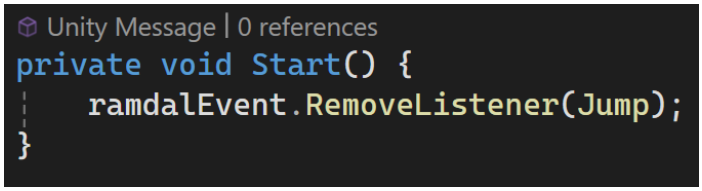
Clears all cached data generated during initialization. The event will automatically initialize again the next time it is invoked.

AddListener()



The runtime listener must match the event's signature. For example, a ***RamdalEvent<int>*** expects an **Action<int>,** while a non-generic ***RamdalEvent*** expects an **Action**. The different event variants are covered in the Generic Ramdal Events.

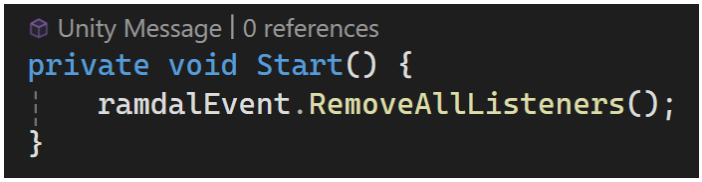
RemoveListener()



Removes a previously registered runtime listener.

Just like ***AddListener(),*** the runtime listener must match the event's signature ***(Action, Action<T>, Action<T1, T2>, etc.)***. The different event variants are covered in the Generic Ramdal Events.

RemoveAllListeners()



Removes all runtime listeners. Persistent Inspector listeners are **not** affected.

AddPersistentListener()

Editor Only. Requires the ***RamdalGames.Events.Editor*** namespace.

Adds a new persistent listener to a ***Ramdal Event*** from code.

```
using RamdalGames.Events;
#if UNITY_EDITOR
using RamdalGames.Events.Editor;
#endif

Unity Script | 0 references
public class GetStarted : MonoBehaviour {
    [SerializeField] private RamdalEvent ramdalEvent;
    0 references
    public void AddPersistentListeners() {
#if UNITY_EDITOR
        RamdalEventEditorTools.AddPersistentListener(ramdalEvent, this, (Action)Jump);
#endif
    }
```

Parameters

- **ramdalEvent** *(Required)* - The Ramdal Event to add the listener to.
- **owner** *(Required)* - The Unity object that owns the event. It will be marked dirty so the changes are saved.
- **methodDelegate** *(Required)* - The method to bind as a persistent listener.
- **runMode** *(Optional)* - Controls when the listener is allowed to execute.
- **eventNotes** *(Optional)* - Custom notes stored with the listener for documentation and organization.
- **defaultParameterValues** *(Optional)* - The default parameter values assigned to the listener. These are the same values you would normally enter in the Inspector.

RemovePersistentListener()

Editor Only. Requires the ***RamdalGames.Events.Editor*** namespace.

```
    0 references
    public void RemovePersistentListener() {
        int index = ramdalEvent.GetPersistentListenerIndex(methodDelegate: (Action)Jump);
#if UNITY_EDITOR
        RamdalEventEditorTools.RemovePersistentListener(ramdalEvent, this, index);
#endif
    }
```

Removes a single persistent listener from a ***Ramdal Event***.

Parameters

- **ramdalEvent** *(Required)* - The Ramdal Event containing the listener.
- **owner** *(Required)* - The Unity object that owns the event. It will be marked dirty so the changes are saved.
- **index** *(Required)* - The index of the persistent listener to remove. You can obtain it using GetPersistentListenerIndex().

RemovePersistentListeners()

Editor Only. Requires the ***RamdalGames.Events.Editor*** namespace.

```
    0 references
    public void RemovePersistentListener() {
        List<int> indices = ramdalEvent.GetPersistentListenerIndices(methodDelegate: (Action)Jump);
#if UNITY_EDITOR
        RamdalEventEditorTools.RemovePersistentListeners(ramdalEvent, this, indices);
#endif
    }
```

Removes multiple persistent listeners from a ***Ramdal Event***.

Parameters

- **ramdalEvent** *(Required)* - The Ramdal Event containing the listeners.

- **owner** *(Required)* - The Unity object that owns the event. It will be marked dirty so the changes are saved.
- **indices** *(Required)* - A list of listener indices to remove. You can obtain them using ***GetPersistentListenerIndices()*** or ***GetPersistentListenerIndicesNonAlloc().***

Get Persistent Listener Index

```
private List<int> indicesNonAlloc;
Unity Message | 0 references
private void Start() {
    int index = ramdalEvent.GetPersistentListenerIndex(methodDelegate: (Action)Jump);
    List<int> indices = ramdalEvent.GetPersistentListenerIndices(methodDelegate: (Action)Jump);
    ramdalEvent.GetPersistentListenerIndicesNonAlloc(indicesNonAlloc, methodDelegate: (Action)Jump);
}
```

GetPersistentListenerIndex()

Returns the index of the **first** persistent listener matching the supplied filters. If no matching listener is found, -1 is returned.

GetPersistentListenerIndices()

Returns the indices of **all** persistent listeners matching the supplied filters.

GetPersistentListenerIndicesNonAlloc()

Works the same as ***GetPersistentListenerIndices(),*** but writes the results into a ***List<int>*** you provide, avoiding a new allocation.

Available Filters (all optional)

- **methodName** - Match by the C# method name.
- **methodID** - Match by the Ramdal Event Method ID.
- **target** - Match listeners targeting a specific Unity object.
- **paramCount** - Match methods with a specific number of parameters.
- **runMode** - Match listeners using a specific RunMode.
- **methodDelegate** - Match an exact delegate. This is the safest and recommended filter when you already have access to the method.

You can combine multiple filters to narrow down your search. At least one filter should be provided; calling these methods without any filters will get skipped.

Properties

Initialized

Returns whether the event has already been initialized.

```
Unity Message | 0 references
private void Start() {
    bool initialized = ramdalEvent.Initialized;
    int count = ramdalEvent.PersistentListenersCount;
}
```

PersistentListenersCount

Returns the number of persistent listeners.

Generic Ramdal Events

Generic ***RamdalEvent<T...>*** variants are only required for runtime usage when working with **runtime listeners** that have parameters.

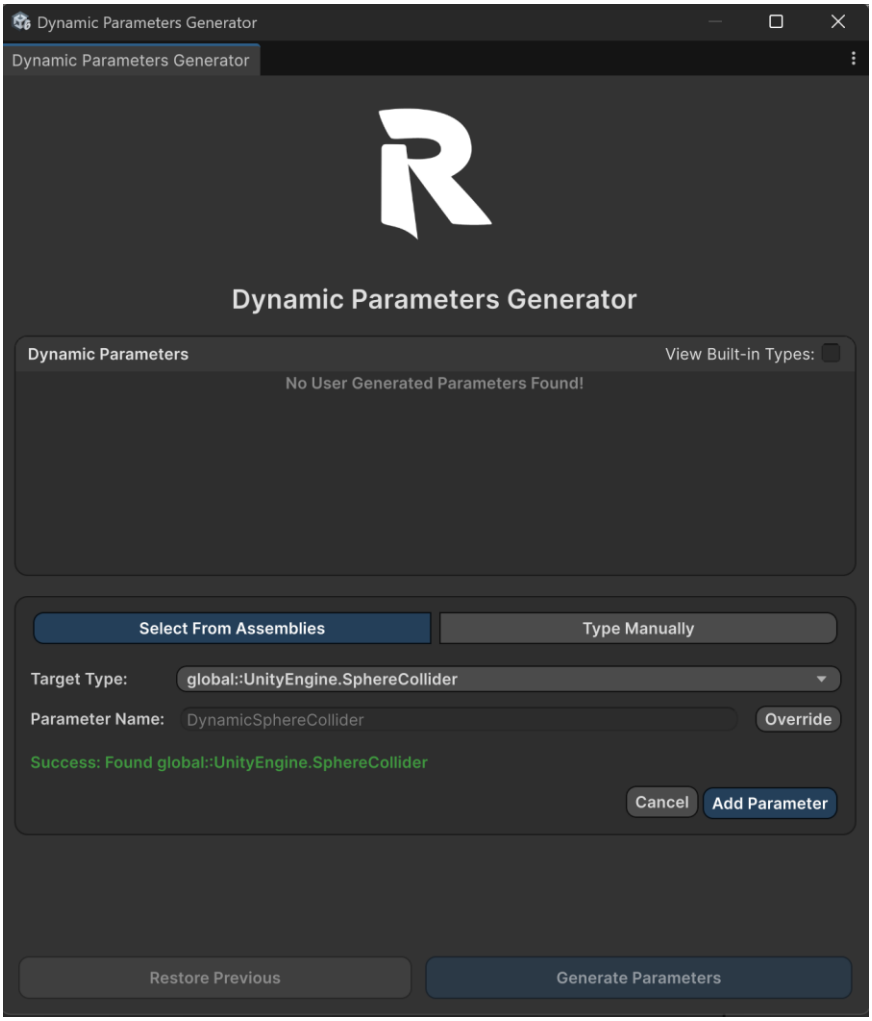
Persistent Inspector listeners do not require generic event variants. They can bind parameterized methods directly without any generic declaration.

```
[SerializeField] private RamdalEvent ramdalEvent;
[SerializeField] private RamdalEvent<float> onJump;
[SerializeField] private RamdalEvent<int, bool> onDeath;
Unity Message | 0 references
private void Start() {
    ramdalEvent.AddListener(Detect);
    onJump.AddListener(Jump);
    onDeath.AddListener(OnDeath);
}
Unity Message | 0 references
private void Update() {
    ramdalEvent.Invoke();
    onJump.Invoke(33.33f);
    onDeath.Invoke(15, false);
}
```

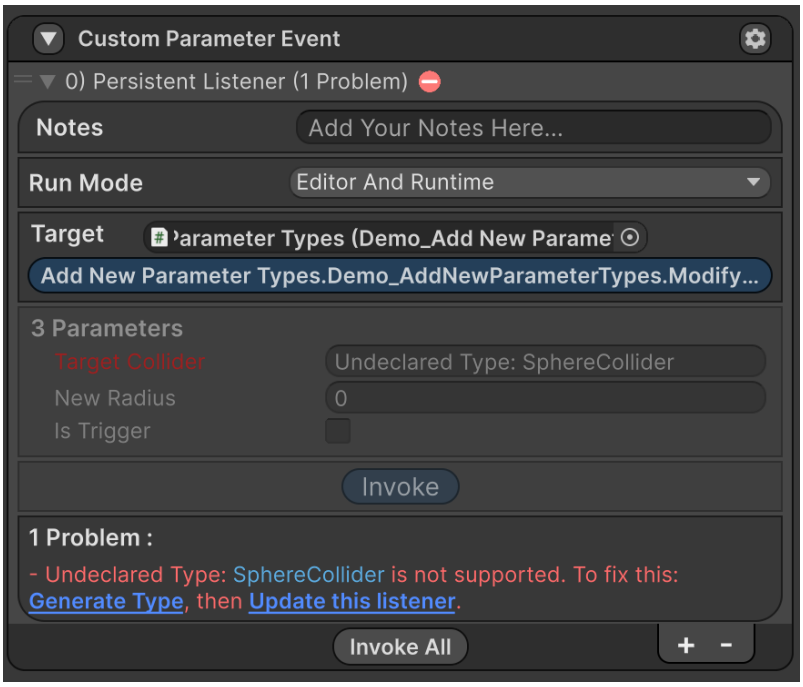
Generic variants are only used when the runtime event itself needs to carry parameters.

Dynamic Parameter Generator

Ramdal Events comes with support for many of the most commonly used parameter types out of the box. If you need additional custom types, you can easily extend the supported parameter list without writing any code. Generated parameters automatically use Unity's serialization and any custom property drawers you already have.



Generating a Dynamic Parameter



1. Open **Tools > Ramdal Games > Ramdal Events > Parameters Generator**, or click the **Settings** button on any Ramdal Event component and select **Parameters Generator**.
2. If you received an **Unsupported/Undeclared Type** error while binding a method, click the **Generate Type** link to automatically open the generator.
3. Click **Add New Dynamic Parameter**.
4. Search for your type from all loaded assemblies, or manually enter its fully qualified type name.
5. Once the type is validated, click **Add** to include it in the list of supported parameters.
6. Optionally change the generated display name.
7. Click **Generate Parameters**.
8. Return to your persistent listener and click **Update This Listener**. Your parameter is now fully supported.

Important Notes

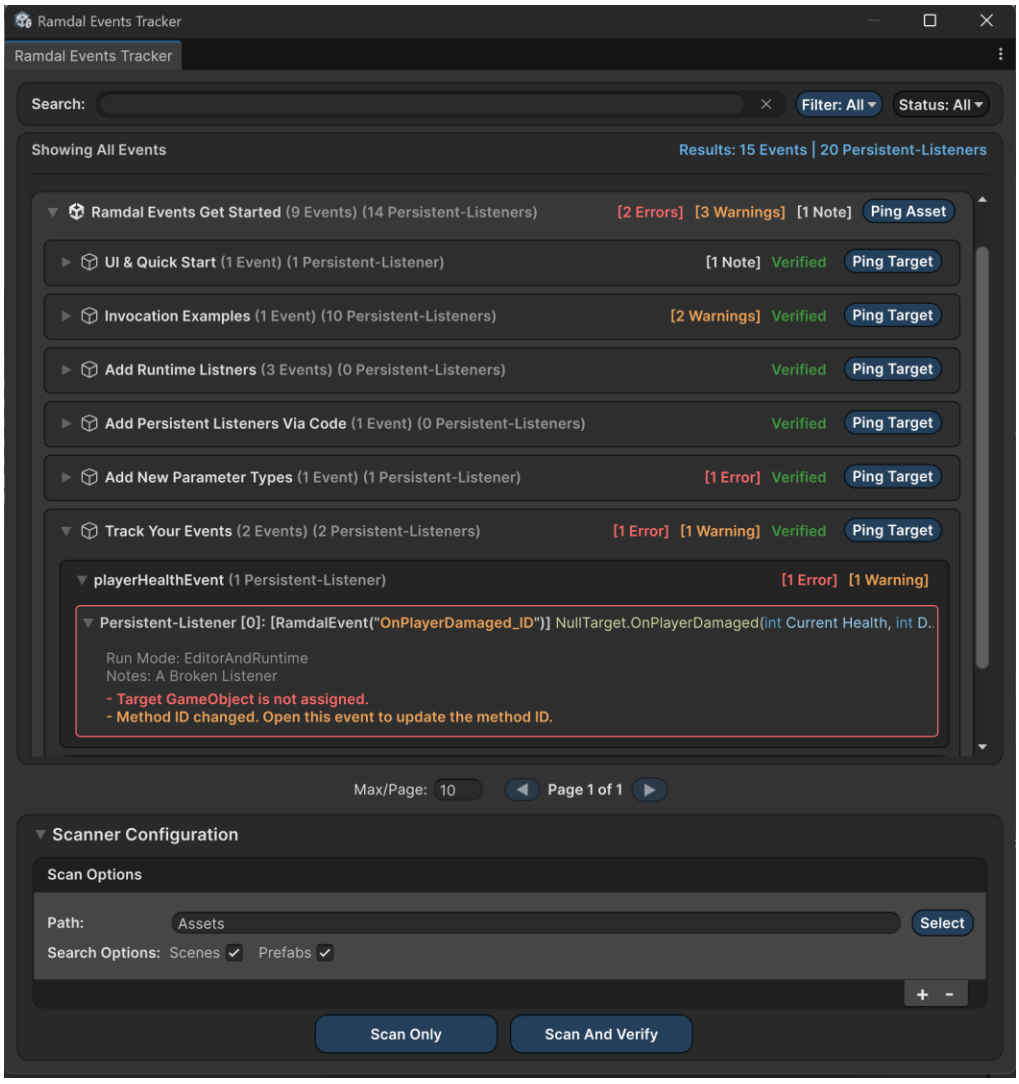
- When using Ramdal Events inside a List, **Unity's default list drawer** creates new elements by duplicating the last item. This also **duplicates the Dynamic Parameter references**, causing both listeners to share the **same parameter instance**. After adding a new element, **clear** the duplicated persistent **listener and reassign its values to create a unique parameter instance.**
- **Do not delete generated parameters** after they have been used by listeners. Existing listeners will lose their parameter data and the value will become null.
- **You can't and shouldn't rename** the generated Dynamic Parameter name after it has been used by listeners. Existing listeners reference it directly, so renaming it will cause them to become missing. However Renaming the original C# type is completely safe.
- **Avoid manually editing the generated file** unless you know exactly what you're doing. Regenerating new parameters will overwrite any manual changes.
- **Do not rename the generated C# file**, as the generator expects the original filename.
- **Editor-only types are currently not supported.** Adding them will compile in the Editor but can cause build errors. Support for these types is planned for a future update.

- If your type cannot be found, try entering its **fully qualified name**, including its namespace.
- Only add **public, accessible runtime types**. Private, internal, or otherwise inaccessible types may produce compiler errors.
- If something goes wrong, use the **Restore Previous** button to revert to the last generated version.

Events Tracker

The Events Tracker is there to make working with Ramdal Events easier. You can open it from **Tools > Ramdal Games > Ramdal Events > Events Tracker**. It lets you search, inspect, and validate every event and persistent listener in your project from a single window.

Each result shows detailed information about the event and its listeners, along with any warnings or errors. You can quickly **Ping** the target object and use the built-in search and filters to find exactly what you need.



Scanning Your Project

The tracker provides two scanning modes:

Scan Only

- Scans your project and lists every Ramdal Event without performing validation.
- Recommended when you simply want to locate events.

Scan & Verify

- Scans the project and verifies every event, checking for issues such as missing methods, duplicate Method IDs, invalid listeners, and other common problems.

Note: On large projects, the first scan may take a little longer since every scene and prefab must be processed. If you only need to locate events, **Scan Only** is usually faster.

Results Hierarchy

Results are organized in the following hierarchy:

1. **Scene / Prefab**
2. **GameObject**
3. **Ramdal Event**
4. **Persistent Listener**

This makes it easy to navigate directly to the exact listener you're looking for.

Searching & Filtering

The tracker supports searching and filtering to quickly narrow down your results.

Multiple filters can be combined to quickly locate specific listeners.

Common Workflows

Fixing a Renamed Method ID

If you renamed a Method ID, Ramdal Events will warn you and display the previous ID.

To update existing listeners:

1. Copy the old Method ID from the warning.
2. Open the Events Tracker.
3. Scan the project.
4. Search for the old ID using the **Method ID** filter.

5. Ping each result.
6. Opening the event automatically updates the listener to the new Method ID.
7. Once the listener has been updated, the warning can be safely dismissed.
8. Repeat for all remaining results.

Resolving Duplicate Method IDs

If you have duplicate Method IDs and you're no longer sure which method was duplicated last or which one should receive a new ID, simply search for that Method ID in the tracker.

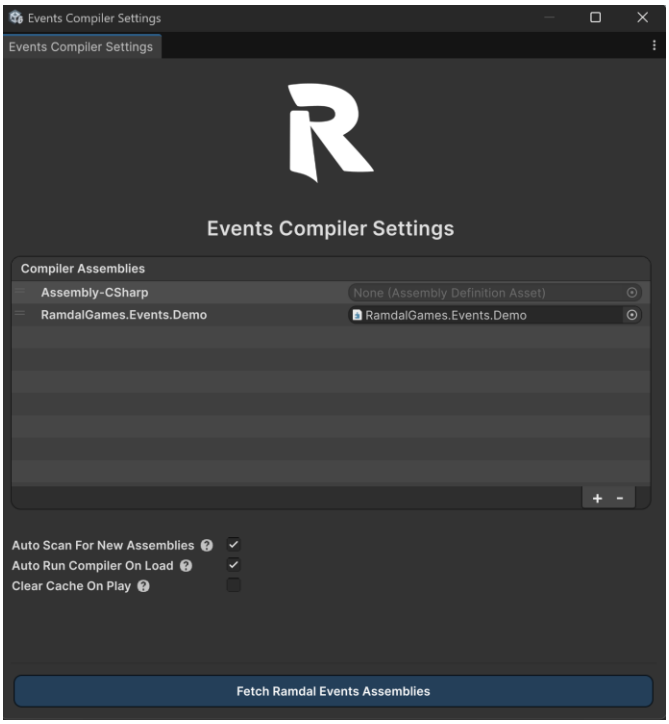
The results will show all methods using it. If all results belong to the same method on the same script, that ID is the original. Then search for the other method name in the project. If it returns no results, that method is the duplicate and should receive a new Method ID.

Checking Whether a Method Is Used

Before deleting a method, search for its **Method Name** and **Method ID** in the tracker.

If no listeners are found, the method is not referenced by any Ramdal Event and can be safely removed.

Background Compilation & Caching



Ramdal Events achieves its performance by caching event data so it can be reused efficiently instead of being rebuilt each time. This makes the first call path extremely fast, especially for events that may only be invoked once. Caching happens automatically at multiple stages.

To make this even faster, Ramdal Events includes a **Background Compiler** that runs automatically before the first scene loads. It scans the configured assemblies for methods exposed with the ***[RamdalEvent]*** attribute and caches all the information required for fast event initialization later.

The compiler runs **asynchronously on a background thread**, is designed to be extremely fast even in large projects, and should have no impact on startup time.

The compiler is fully configurable. You can choose which assemblies to scan, run it manually, or disable it completely if you prefer. Even when disabled, Ramdal Events continues to work normally. The only difference is that methods are discovered the first time they're needed before being cached, resulting in a slightly slower first initialization.

Compiler API

The compiler can be accessed through **RamdalEventRuntimeTools.EventsCompiler**.

RunAsync()

Starts the background compiler using the assemblies configured in the compiler settings.

RunAsync(string[] assemblies)

Starts the background compiler and scans only the assemblies you provide.

CancelBackgroundCompilation()

Requests cancellation of the current background compilation.

ClearAllCache()

Clears all cached compiler data. The cache will be rebuilt the next time the compiler runs or when events initialize normally.

Properties

IsCompiling

Returns whether the background compiler is currently running.

HasFinished

Returns whether the latest background compilation has completed.

Notes

- Background Compilation is **enabled by default**.
- Only assemblies containing Ramdal Events need to be scanned.
- Disabling the compiler does **not** disable Ramdal Events or its runtime caching. It only removes the pre-compilation step before events are first used.
- Most projects can simply leave the default settings unchanged.

IL2CPP AOT Optimization

IL2CPP builds introduce a specific performance consideration related to generic code paths. When Unity encounters generic combinations that were not explicitly generated during compilation, it falls back to a shared generic implementation. This fallback is safe, but less optimized because the compiler cannot generate specialized code paths for those cases.

In Ramdal Events, this can happen when new parameter combinations are used that were not previously seen during compilation.

What AOT Optimization Does

The AOT optimization step generates explicit C# references for every parameter combination used in your project. This forces IL2CPP to generate fully specialized native code paths instead of relying on shared generic implementations.

The result is more efficient event invocation and reduced overhead in IL2CPP builds, especially for multi-parameter events.

Performance Impact

This step is optional, but recommended if you want maximum performance in IL2CPP builds. In some cases, it can improve invocation performance by up to **3x or more**, depending on parameter complexity.

This optimization has **no effect in the Editor or Mono builds**, since those use JIT compilation.

When to Regenerate

You only need to regenerate the AOT file when you introduce new parameter combination in your events

Automation (Recommended)

You can automate this step in your build pipeline:

```
RamdalEventEditorTools.GenerateAOTRuntimeOptimization();
```

This ensures the AOT file is always up to date before every build.

Manual Generation

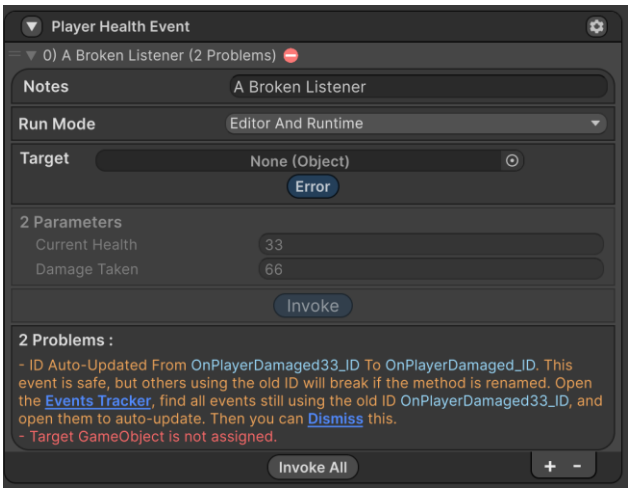
If you are not automating the process, you can generate the optimization file manually:

- From any Ramdal Event inspector header **> Generate IL2CPP AOT Optimization**
- Or via menu:
  Tools > Ramdal Games > Ramdal Events > Generate IL2CPP AOT Optimization

External Reference

For more details on IL2CPP limitations, see Unity documentation.

Logs & Diagnostics



Every persistent listener includes a built-in logging and diagnostics system designed to help you identify and resolve problems quickly.

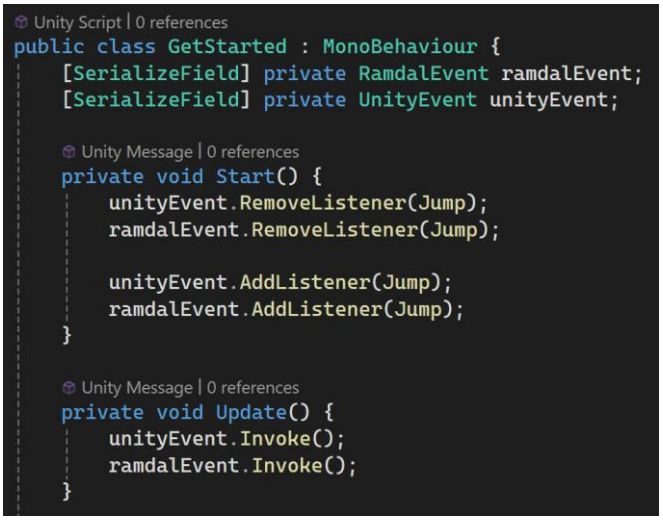
Rather than silently failing, Ramdal Events reports warnings, errors, and useful information directly where the problem occurs. Many messages also provide contextual guidance or one-click actions to help you fix the issue immediately.

The diagnostics system can help detect issues such as:

- Missing or renamed methods.
- Changed or duplicate Method IDs.
- Unsupported parameter types.
- Invalid or missing targets.
- Build and serialization issues.
- Other common configuration problems.

Whenever possible, Ramdal Events explains what went wrong, why it happened, and how to resolve it, making it much easier to diagnose issues without searching through your project manually.

Migrating from Unity Events



```
Unity Script | 0 references
public class GetStarted : MonoBehaviour {
    [SerializeField] private RamdalEvent ramdalEvent;
    [SerializeField] private UnityEvent unityEvent;

    Unity Message | 0 references
    private void Start() {
        unityEvent.RemoveListener(Jump);
        ramdalEvent.RemoveListener(Jump);

        unityEvent.AddListener(Jump);
        ramdalEvent.AddListener(Jump);
    }

    Unity Message | 0 references
    private void Update() {
        unityEvent.Invoke();
        ramdalEvent.Invoke();
    }
```

Migrating from Unity Events is straightforward.

- Replace your **UnityEvent** fields with the equivalent **RamdalEvent**.
- Existing runtime code (Invoke(), adding/removing runtime listeners, etc.) works the same, so in most cases no additional code changes are required.
- If you want to add **persistent listeners from code**, see the Runtime Usage section.

The only manual step is reassigning your existing Inspector listeners, as Unity and Ramdal Events use different serialization formats. You can either reassign them manually or create your own migration tool.

Note: An automatic Unity Events migration tool is planned for a future update.

Common Issues / Troubleshooting

This section covers the most common problems you may encounter when working with Ramdal Events and how to fix them quickly.

Method Does Not Appear in the Dropdown

- Make sure the method is marked with the **[RamdalEvent]** attribute.
- Ensure the script is compiled without errors.
- Verify the method belongs to a **MonoBehaviour** or **ScriptableObject**.

Method Becomes Missing After Renaming

- This happens when the Method ID and Method Name is changed, or when no Method ID was assigned. It can also occur if the parameter types or count were modified, or if the method itself was deleted.
- Use **Events Tracker** to locate all references to the old Method ID.
- Update affected listeners using the tracker workflow.
- Read Common Workflows for more info.

Duplicate Method ID Error

- Each Method ID must be unique within the same class.
- If duplicates are detected, Ramdal Events will show a warning.
- Use the **Events Tracker** to locate all usages and assign a new ID where needed.
- Read Common Workflows for more info.

Event Not Invoking

- Check if the event has any persistent listeners assigned.
- Ensure runtime listeners are registered correctly.

Undeclared Type

- Open the Dynamic Parameter Generator.
- Add the missing type and regenerate parameters.
- Make sure the type is accessible and not editor-only.
- Read Generating a Dynamic Parameter for more info.

Null Parameter

- You might renamed the **DynamicParameterContainer** Name or the **User Generated Parameters** cs.
- Try regenerating the parameter type and refresh the persistent listeners and use the events tracker to find the rest of the corruption.

Limitations & Things to Know

Destroying Persistent Listener Targets

If you destroy an object referenced by a persistent listener, call **yourEvent.ClearCache()** afterwards. Otherwise, the cached delegate will continue referencing the destroyed object, preventing it from being **garbage collected** and potentially causing a **memory leak**.

Runtime Listeners Must Be Unsubscribed

Just like normal **C# delegates**, if you subscribe to a runtime listener using **AddListener(),** you should call **RemoveListener()** before the target object is destroyed. Failing to unsubscribe can lead to **memory leaks** or **unexpected callbacks**.

Unity List Drawer Bug

Unity has a bug where repeatedly selecting and deselecting the rearrange handle can create duplicate item panels. Although Ramdal Events uses a heavily modified list drawer, it still inherits this Unity Editor bug. This is a rare issue that was only encountered during extensive testing. If it happens, simply select another object to exit the Inspector, then select the object again. It is purely visual and does not affect event functionality.

Unity List Duplication Bug

When a Ramdal Event is stored inside a List or array drawn by Unity's default list drawer, adding a new element to that list duplicates the previous one, including its Dynamic Parameter references. As a result, both events will initially share the same parameter instances.

After creating a new element, clear the copied persistent listener and assign it again to generate new parameter instances.

Note: This only affects Unity's default List/array drawer. Duplicating GameObjects, Prefabs, or Components does **not** have this issue.

Unity Prefab Override Bug

Unity currently has a known issue where Prefab overrides are not cleaned up correctly when the managed instance type of a **SerializeReference** changes. This is a Unity bug and is unrelated to Ramdal Events.

See Unity's Issue Tracker for more information.

Code Stripping / Missing Methods in Builds

Avoid aggressive stripping of private methods or methods not directly referenced in code but used by Ramdal Events through **[RamdalEvent].** These methods may be removed during build optimization and cause missing bindings at runtime.
Use the **[Preserve]** attribute to ensure such methods are retained.

Background Compiler Threading Note

The Background Compilation system runs asynchronously and uses multi-threading for performance. This is fully safe on supported platforms. However, on **WebGL** builds, multi-threading support is limited or unavailable depending on Unity settings and browser constraints, so background compilation may be disabled or will throw warnings.

Even if disabled or unavailable, Ramdal Events will still function normally. The compiler is purely an optimization layer that improves first-call initialization speed and is not required for correct behavior.

Dynamic Parameters & Editor Types Limitation

Avoid adding Editor-only types when generating Dynamic Parameters. They may compile in the Editor but will fail during build and prevent your project from building correctly.

Also avoid using types that contain ***#if UNITY_EDITOR*** wrapped serialized data or editor-only fields inside otherwise runtime types.
These can cause serialization mismatches and build-time failures when used with Ramdal Events.

This may result in errors such as:

"A scripted object (probably YourNamespace.YourClass?) has a different serialization layout when loading. (Read 32 bytes but expected 64 bytes)"

Best Practices

- Use **unique Method IDs** per method inside the same class.
- Keep Method IDs **stable** once used in persistent listeners.
- Use **[RamdalEvent]** only on methods you actually want to expose.
- Call **Initialize()** before the first invocation for events used in performance-critical paths.
- Enable Background Compilation.
- Generate **IL2CPP AOT Optimization** before building for **IL2CPP**.
- Use the **Events Tracker** to validate and debug event usage.
- Avoid **ref/out** parameters and more than **16 parameters** in hot paths.
- Use Dynamic Parameters only for the types you need. Avoid generating too many, not because of performance, but to keep your project clean and organized.
- Do not manually edit generated parameter or compiler files.
- Use the **Events Tracker** before deleting or renaming methods.
- Use **notes** to stay organized and quickly understand the purpose of your events and persistent listeners.
- Use **[UnityEngine.Scripting.Preserve]** on methods used only by **Ramdal Events** to prevent them from being stripped during builds. This is especially important for private or protected methods that are not directly referenced in code.

FAQ

Do I need a Method ID?

No, but you should use one. Without a Method ID, you lose safe method renaming and Background Compilation & Caching optimizations.

What happens if I rename a Method ID?

If you change the **[RamdalEvent]** attribute after a method has already been assigned (for example, changing from no Method ID to a Method ID, or changing the Method ID itself), existing listeners are not updated automatically. They continue referencing the old serialized information. If you later rename the method as well, those listeners will no longer be able to find it. Use the **Events Tracker** to locate and update all affected listeners before renaming the method.

Read Common Workflows for more info.

Do I need to use generic RamdalEvent<T>?

Only if you are using runtime listeners with parameters in **C#**. **Persistent inspector listeners do not require generic events**.

Can I use private or protected methods?

Yes. Methods do not need to be public as long as they are marked with **[RamdalEvent]**.

Why didn't my method show up in the dropdown?

Make sure the method is marked with the **[RamdalEvent]** attribute, belongs to a **MonoBehaviour** or **ScriptableObject**, and that the project compiled without errors.

Can I delete generated parameters

You can delete generated parameters, but only if you are certain they are not used in any persistent listener.

Why is my custom parameter type not supported?

Generate a Dynamic Parameter for the type using the **Dynamic Parameters Generator**.

Why can't I find my type in the Dynamic Parameter Generator?

If your type does not appear in the generator, it is usually not serialized or not accessible at runtime.Avoid using ***editor-only*** types or types wrapped in ***#if UNITY_EDITOR***.

Make sure to type the class name correctly (for example SphereCollider, not spherecollider), or use the full namespace if needed. You can also switch to the assembly view and manually locate the type in its hierarchy.

Can I change a Method ID after assigning listeners?

Yes, but existing listeners won't update automatically. Use the **Events Tracker** to locate and update them.

Read Common Workflows for more info.

Why are my parameters null?

This usually means the generated Dynamic Parameter renamed, or deleted. Regenerate the parameter and refresh the affected listeners.

Can I expose static methods?

Yes. Static methods are fully supported, as long as they belong to a **MonoBehaviour** or **ScriptableObject**.

Can I expose methods with return values?

Yes. Methods with return values are fully supported and do not require wrapper methods. Any return value is simply ignored during event invocation.

Can I expose generic methods?

No. Ramdal Events currently does not support generic methods (for example, ***MyMethod<T>(T value)***). If you need to expose one, create a non-generic wrapper method instead.

Do I need to call ClearCache()?

No. Only call it if you destroy objects referenced by cached persistent listeners or if you explicitly want to force the event to rebuild its cache.

Can I have multiple methods with the same Method ID?

Yes, as long as they're in different classes. Method IDs only need to be unique within the same class.

Why does my method invoke in the Editor but not in a Build?

This can happen if the method is being removed by aggressive code stripping during build. If the method is private or protected and is not directly referenced in code, Unity may strip it from the build **even if it is used by Ramdal Events**.

To prevent this, use the **[Preserve]** attribute on the method (alongside **[RamdalEvent]**) to ensure it is not removed during stripping.

Support and Useful Links

Video Tutorials: youtube.com/playlist?list=PLPoPvIVslWnE
Support: support@ramdalgames.com
Contact: contact@ramdalgames.com
Ramdal Games: https://ramdalgames.com
Mohamed Haftari: https://mohamedhaftari.com
Join Our Discord Community
Explore More Assets and Tools