Cobra Logging

Cobra Logging provides developers with a rapid and scalable logging solution suitable for all types of projects.

Unity's default debug class may meet fundamental needs, but more complex projects often call for a deeper, more nuanced logging system. Arm yourself with a logging solution built to support you throughout your entire product cycle, from the initial stages of development to the launch of your product.

Features

Multi-Tiered Logging

Capture messages across various levels: Debug, Info, Warning, Assert, Error, and Fatal.

Interactive logs

Click on any log within Unity to be taken directly to its origin in your favorite code editor. Every log displays crucial data: timestamp, log type, originating file, and exact line.

Sample log display: 10:00:30 [INFO] MyScript[123]: This is a test log message.

Generated from MyScript.cs on line 123 at 10:00:30, this log isn't just informative - it's interactive. One click within the Unity editor, and you're directly navigated to that exact line of code in your favorite code editor.

Moreover, you're not restricted to this format. Every component of this log display can be molded to match your requirements. To further enhance readability, each log type is designated a unique color - yet another feature you can personalize to your liking.

Deep Customization

Easily modify tags, colors, and even overhaul logs via code or inspector. Choose to halt execution for Asserts and Errors. Fatal logs always halt execution, but an optional 'quit' parameter offers an added layer of security.

Preserve Object States

Save states within logs, capturing the full context of each event. Record any C# object, either through direct JSON serialization or through custom serialization methods. Take real-time snapshots of class instances, storing variable states for in-depth future analysis.

Save Logs To Files

This feature allows for the automatic saving of log entries to designated files, ensuring a permanent record of system events and activities for review and analysis.

Thread Safety

Experience unhindered multi-threading without the dread of file access errors.

Flexible Filtering

Pre-configured filters: Namespace, Directory, Message, and Tag. Configure them effortlessly either via the custom inspector or directly through C#. Choose the method that best aligns with your preferences.

You also have full liberty to craft your own filters by simply inheriting and implementing the filter class.

Additional customization options:

- Namespace: Target specific namespaces.
- Directory: Specify directories of origin.

Intuitive Inspector Interface

User-centric design ensuring effortless navigation and operation. Store and switch between multiple log settings, ideal for collaborative projects.

Test Suite Include

Modify this asset with confidence. Every purchase comes with an inclusive test suite, ensuring any changes you make are robust and error-free.

Full Source Code Access

Dive into the inner workings and make it uniquely yours.

Getting Started

Embarking on your journey with Cobra Logging is a straightforward process. Begin by importing the package into your Unity project and incorporating the namespace into your scripts:

```
using ByteCobra.Logging;
```

Customizing the Logging System

Cobra Logging is not only robust but also flexible, offering a scriptable object used for storing your custom settings. To create a new settings object, right-click within any of your project folders and navigate through:

`ByteCobra -> Logging -> Create Settings`

This action spawns a new settings object, presenting you with the ability to personalize the logging system. Settings can be configured either programmatically or via one of the included prefabs.

For prefab-based settings selection, simply add the Cobra Settings prefab from the settings folder to your scene. Upon addition, select it and assign a settings object tailored with the configurations you wish to employ. ## Code Modification Certain components of the logging system are delivered pre-compiled as assemblies (.dll files) to ensure optimal functionality, particularly regarding interactive logging, which may not operate correctly if the classes are directly integrated as source code within Unity.

Nonetheless, the source code is inclusively provided to empower you with the flexibility to make requisite modifications. Simply build the modified source and drag it into your Unity projects. The assembly for this package can be located within the Plugins folder.

Debug Log

Every developer, at some point, feels like a detective. Searching for elusive clues, connecting dots, and making sense of it all. This is where the Debug method of the Cobra Logging system shines brightest. It's your magnifying glass in the intricate world of Unity.

Consider Debug as your personal detective's notebook. Instead of capturing the daily happenings like its cousin Info, it focuses on those nitty-gritty details, the fine prints that often lead to breakthroughs.

Parameters

- message: This is your clue or observation. It's more detailed, often hinting at what's behind the curtain. Maybe it's "Player's jump height adjusted" or "AI pathfinding recalculated.
- state: This is the extra layer of intelligence, a deeper dive. It could be as straightforward as a variable's current value or as complex as an entire object's state.

The Right Times in Unity

- Algorithm Adjustments: If you've just tinkered with an AI's decision-making algorithm, you might want to know its choices in real-time. Debug is your best pal here.
- Physics Interactions: Those moments when objects collide, bounce, or simply fall, and you want to capture the exact force or angle.
- Shader Effects: Monitoring the subtle changes in post-processing effects or shaders can be a lifesaver, especially when chasing visual bugs.
- Memory Monitoring: Tracking object instantiations or deletions to manage memory better.

Input Handling: Understand the sequence and type of user inputs, particularly when debugging complex command sequences or gestures.

An Example

Picture this. You're designing a Unity game with a complex AI. You've just tweaked its decision-making, and now you want insights:

```
void OnAIDecisionMade(AIDecision decision)
{
    var aiStatus = new { Decision = decision.Type, LastActionTime =
        decision.Timestamp };
    Log.Debug("AI made a decision.", state: aiStatus);
}
```

Here, not just the decision type, but even its timestamp is captured. This can help in analyzing patterns or predicting behaviors.

The value of Debug isn't just in solving problems; it's in understanding them. It shines a spotlight on the hidden alleys of your code, ensuring that when a bug does rear its head, you're ready, magnifying glass in hand, to get to its roots. Embrace Debug, and let your detective journey in Unity be ever enlightening!

Info Log

The Info method in the Cobra Logging system is your ally in tracking the narrative of your application.

At its heart, Info is a storyteller. It gives voice to your application's events. Instead of shouting errors, it calmly narrates what's happening, offering vital clues without the drama.

Parameters

- message: Your protagonist. This is the core tale you want to tell, whether it's "Player entered the room" or "Network connection initialized."

- state: The sidekick. It enriches your story, providing optional, additional data. Maybe it's the player's current score or the number of NPCs in a scene.

When to Narrate in Unity?

- Debugging Animations: Not sure when a specific animation starts or ends? Drop an Info log in the trigger functions.
- User Interactions: Curious about the paths players take? Log informational messages when they interact with game objects or UI elements.
- Scene Transitions: When a player moves between scenes, especially in large open-world settings, use Info to track these transitions smoothly.
- Network Activities: If you're working on a multiplayer setup, knowing when a player joins a lobby or sends a message can be invaluable.
- Resource Loading: When loading resources, especially asynchronously, Info logs can offer a breadcrumb trail of what assets have been loaded.

Example

Imagine you're developing a Unity game where a player enters a mysterious room:

```
void OnPlayerEnterRoom()
{
    var playerData = new { Health = 100, InventoryCount = 5 };
    Log.Info("Player has entered the mysterious room.", state: playerData);
}
```

Here, not only do you know when the player entered the room, but you also get a quick snapshot of their health and inventory count at that moment because the player data state is passed in.

Remember, while errors scream for attention, it's often the silent, informative whispers that hold the keys to understanding your application's journey. So, log wisely, and let Info be your guiding light in the dark corners of game development.

Warning Log

Imagine you're the captain of a ship navigating through uncharted waters. The night is dark, but there's a beacon - a light signaling caution. This is what Warning in the Cobra Logging system embodies for a Unity developer. It doesn't scream emergency; instead, it softly whispers, "Tread carefully."

At the heart of development lies a constant struggle to balance ambition with prudence. We push boundaries, but not recklessly so. The Warning log is that gentle nudge, the guiding light ensuring we don't stray too far off the mark.

Parameters

- message: This is your message. Think of it as the warning bell. It tells you straight up what's up, like "Hey, the texture's taking a bit long to load" or "Woah, the player did something unexpected."
- state: Want a closer look? The state gives you a peek into what was happening when the warning came up. It could be how much health a player had or what the score was. This is an optional parameter.

When Would I Use a Warning Log in Unity?

- Performance Heads-Up: Spotting the early signs of things slowing down.
- Resource Check: Telling you when you're running low on memory or any game resource.
- Checking Internet Stuff: Like when the game's talking to online servers and things don't go as planned.
- Game Limits: Like if your game character collects too many coins or goes out of the game's boundary.

Example

Imagine you've got a game where players collect magic orbs. But there's a limit to how many they can have. You'd wanna know if a player is close to that limit:

```
void OnOrbCollection()
{
    if (player.OrbsCollected >= MaxOrbs - 5)
    {
        Log.Warning("Player's close to the orb limit! They've got: "
        + player.OrbsCollected);
    }
}
```

This way, before the player hits the orb limit, you get a heads-up. Maybe you want to pop a message for the player or change how the game works.

The Warning log is like that buddy who always has your back, making sure you know about stuff before it becomes a bigger problem. It's all about giving you the info you need so you can make the best call.

Assert Log

In the vast realm of game development, checks and balances are vital. Among these checks stands the mighty Assert log, designed to ensure conditions meet our expectations and act as a protective barrier against unforeseen errors.

The Assert function primarily checks if the condition is true. If not, and if asserts are enabled in your settings, it then creates an AssertLog. This log captures

the message, the condition state, and any optional context you provide. If set to throw an exception, the code will stop running, allowing you to address the issue immediately.

Parameters

- condition: This is the heart of the assert function. It's a condition you're checking for, like making sure a player's score isn't negative or ensuring an object isn't null before using it.
- message: If the condition fails, this message gets logged. It's like the signal flare, alerting you to what went wrong. This parameter is optional, but it's often a good idea to be descriptive. (Optional)
- throwException: A nifty switch, this determines whether to throw an exception when the condition is false. If you're testing out something critical and want the program to halt at a failure, this is the button you'd press. By default, it's set to true, so it'll throw unless you tell it otherwise.
- state: Sometimes, you want a snapshot of what's going on when the assert checks out. That's what state is for. It can be any relevant data or context, like the player's position or the current level. (Optional)

When and How to Use

Suppose you're crafting a game where a player has to pick up magical orbs. You want to ensure that the player's orb count never exceeds a set limit:

```
int MaxOrbs = 100;

void AddOrb(int count)
{
    Log.Assert((player.OrbsCollected + count) <= MaxOrbs,
        "Player orb count exceeded the limit!");

    player.OrbsCollected += count;
}
```

In this scenario, if the player somehow collects orbs beyond the MaxOrbs value, the Assert will flag it, the message will be logged, and, if set, an exception thrown which stops the error from propagating further into the system.

The Assert log acts as both a guardian and a detective. While it protects the sanctity of conditions, it also quickly points to areas where things are not as they should be. Equip it wisely in your coding arsenal, and let it be a beacon of assurance in your development journey.

Error Log

Programming, much like navigating a vast sea, has its challenges. Among the waves and gusts, an error is a looming storm cloud, a call to action that something isn't right. The Error log is the beacon, the siren that not only alerts you to the issue but offers insights on how to navigate through it.

When invoked, if error logging is toggled on in your settings, an ErrorLog springs to life. Alongside the provided message, it captures the moment's essence, from the stack trace to any added context. Post-creation, if the throwException switch is on, the system halts, demanding attention, and if there's any relevant state data, it's serialized for deeper analysis.

Parameters

- message: This is the alarm bell of the function. When an error occurs, this message provides clarity on the nature of the problem, be it "Failed to load level assets" or "Player HP is below 0."
- throwException: Think of this as the next level alert. If this switch is on (and by default, it is), it's not just about logging the error; the system will actively throw an exception, pausing the code's execution to draw attention to the crisis.
- state: Context is vital when deciphering errors. The state captures this context, like a snapshot in time. Whether it's a character's last action or the state of the game world when things went south, state keeps that memory. (Optional)

Effective Utilization

Imagine a scenario in a strategy game where a player is about to deploy a unit but lacks the necessary resources:

```
void DeployUnit(Unit unit)
{
    if (player.Resources < unit.Cost)
    {
        Log.Error("Insufficient resources to deploy the unit!", throwException: false);
        return;
    }
    // Deploy the unit
}
```

Here, if the resources fall short, the error logs it, but the game continues without crashing, thanks to the throwException set to false.

The Error log is more than just a reporter; it's a safeguard and a guide. It announces the presence of issues and provides a way to understand and address

them. Being informed and agile in responding to such signals ensures that your game remains an enjoyable voyage for all players.

Fatal Log

Amidst the realm of software development, there are many signals, warnings, and alerts. But, among them, the Fatal Log is the loudest cry. It's the signal that something critical has failed and demands immediate attention. It's not just a plea; it's an urgent announcement, one that might even decide if your application should continue running.

Upon invocation and if fatal logging is activated in your settings, a FatalLog is birthed. Beyond the supplied message, this log records the setting of that moment, encapsulating everything from stack traces to any provided state information. Following its creation, if the quit switch is flipped on, the application can be directed to shut down entirely.

Parameters

- message: It's the heartbeat of the function. Whenever a catastrophic failure is detected, this message clarifies the nature of the catastrophe, be it "Failed to initialize core game engine" or "Database connection irretrievably lost."
- quit: The gravity of a fatal error might necessitate bringing the entire operation to a halt. When this flag is raised, post the logging of the fatal error, the application might be instructed to cease running.
- state: A lens to the situation, this parameter, although optional, captures a precise moment. It's the detailing in a detective's case file, giving insights like the status of a process when the critical event occurred.

A Practical Scenario

Imagine a high-stakes online game. In a world where achievements, in-game currencies, and leaderboards matter, hackers might try to gain an unfair edge by injecting unauthorized code. The Fatal log acts as a guard against such attempts:

```
void IntegrityCheck()
{
    if (!IsGameCodeUntampered())
    {
        Log.Fatal("Possible unauthorized code injection detected! Shutting down for security
    }
    // Proceed with game functions
}
```

In this situation, any attempt to tamper with the game's code triggers the Fatal log. The game immediately ceases to function, safeguarding the sanctity of its ecosystem.

The Fatal log is more than just a logger; it's a shield against unwanted intrusions. By utilizing it effectively, developers can not only monitor their application's health but also protect it from those who might wish it harm. It's a vital tool in ensuring the integrity and security of any application.

Directory Filter

Part of the filtration system, DirectoryFilter is a special filter class tailored to assess logs based on the directories they originate from. If you are working on scripts that are located in one or a few specific directories, then you can use this filter to get logs only from those directories.

Namespace Filter

When debugging or inspecting logs, it's often necessary to pinpoint issues originating from specific parts of an application. Different modules or components can span multiple namespaces, and sifting through logs from these namespaces can be cumbersome. Cobra Logging's NamespaceFilter rises to this challenge by providing an elegant way to filter logs based on their originating namespaces.

Housed within Cobra Logging's filtration system, NamespaceFilter is a unique filter designed to assess logs by examining the namespaces present in their stack traces.

Message Filter

Logs are a treasure trove of information. But, with logs coming in thick and fast, how do you focus on the ones that matter? What if you could filter logs based on specific phrases or keywords? Cobra Logging's MessageFilter is designed for this exact purpose.

How Does It Work?

Upon invoking the Validate method: - If Messages property is devoid of any content, every log proceeds unhindered. - With UseContainsCheck set to true, the filter scans if the log's message holds any string from the Messages set. - Conversely, if UseContainsCheck is false, an exact match between the log's message and the Messages set is pursued.

Tags

In the labyrinth of log messages that a software produces, distinguishing between various levels of severity or purposes becomes crucial. Just as a librarian would tag books for easy discovery, developers need a way to categorize logs quickly. The TagSettings class in Cobra Logging offers just the right set of tools for this.

A component of ByteCobra.Logging.Settings, the TagSettings class furnishes the facility to set prefixes or 'tags' for each log type. These tags ensure that a developer can, at a glance, understand the nature of the log.

Why Tags Are Important?

Imagine reading a book without chapter titles or headings. It becomes challenging, right? Similarly, tags in logs serve as these headings, offering an immediate understanding of the message's intent.

Understanding the Tags

- FatalTag: An tag for the logs that signal something went fatally wrong. Default: "[FATAL]".
- ErrorTag: Marks logs that point towards an error in the system. These might not be as dire as fatal logs but certainly require attention. Default: "[ERROR]".
- WarningTag: Highlights logs that warn of potential issues. These aren't necessarily errors but suggest that something might be amiss. Default: "[WARNING]".
- InfoTag: These tags label general informational logs, providing context or details about the system's operation. Default: "[INFO]".
- DebugTag: An insignia for logs used in debugging. These are typically rich in detail, aiding developers in tracing and fixing issues. Default: "[DEBUG]".
- AssertTag: Represents logs that come into play for assertions. These are used to confirm if a condition holds true and log when it doesn't. Default: "[ASSERT]".

States

Logging is vital for tracking the behavior and states of software. In the CatLog namespace, you have classes dedicated to representing, capturing, and serializing object states during the logging process.

Object States

The ObjectStates class provides a snapshot of an object's state during a log event. Object states are saved to the Logs/States directory by

default. However, you can customize this path by adjusting the LogSettings.FileSettings.ObjectStatesDirectory property.

While the default serialization format is JSON, if you wish to use a different format, you can inherit from the serializer class and modify the serializer setting in LogSettings. Ensure that the object is JSON-serializable. If not, either implement a custom serializer for the object or convert it to a serializable data type before logging.

Properties

- TimeStamp: Represents the exact time when the state was captured.
- Message: An optional message associated with the current state.
- State: Represents the captured object's state.
- StackTrace: An associated stack trace, formatted for ease of reading.

Usage

When a specific state of an object needs to be logged, an instance of ObjectStates can be initialized, capturing the current state and any associated (optional) stack trace.

```
var objState = new ObjectStates(myObject, new StackTrace());
```

JsonStateSerializer

JsonStateSerializer provides functionalities to serialize the state of a log object into a JSON format. This is particularly useful for persisting object states to disk or other storage mediums.

Properties

- settings: Contains the JSON serialization settings used internally.

Usage

The JsonStateSerializer can be used to serialize a log's state, providing a robust and readable snapshot of that log event.

```
var serializer = new JsonStateSerializer();
serializer.Serialize(myLog);
```

Colors

Cobra Logging is not just about ensuring that your application's logs are meticulously detailed and well-organized, but also about making sure they are visually distinguishable and easy on the eyes. With the ColorSettings class, you have full control over the colors in which different log messages appear. Tailoring the

color scheme to your preferences or to suit specific environments can greatly enhance readability and immediate recognition of log severities.

The ColorSettings class within the ByteCobra.Logging.Settings namespace is your personal artist's palette for log messages. Each log type can be colored differently, ensuring that at a single glance, you can identify the nature and severity of a message.

How to Customize Programatically

```
var colorSettings = new CatLog.Settings.ColorSettings
{
    DebugLogColor = "purple",
    InfoLogColor = "green",
    // ... and so on for other log types
};
```

With Cobra Logging's ColorSettings, your logs can be as vibrant or as muted as you'd like. Whether you prefer bold colors that jump off the screen or subtle shades that are easy on the eyes, customization is just a property away. By assigning different colors to different log types, you're not just personalizing your log's appearance, but also enhancing its functionality and readability. Happy logging!

Files

File-based logging is an essential component for any significant game or application, serving as a vital tool for maintaining a permanent record of events and activities. With Cobra Logging's enhanced `FileSettings`, managing and controlling the storage of logs as files has become more efficient and customizable, ideal for archiving, analyzing, or sharing logs.

The `FileSettings` class, part of the `ByteCobra.Logging.Settings` namespace, provides a comprehensive set of configuration options to tailor your logging needs.

File Size Restrictions

- **MaxFileSize**: Specifies the maximum allowed size for individual log files. Default: 100 MB. This limit ensures efficient management of disk space.
- **MaxStateDirectorySize**: Determines the size cap for the directory holding serialized object states. Default: 100 MB. This helps in managing the storage space more effectively.

Directory Management

- **LogFilesDirectory**: Designates the storage location for your log files. Updating this setting automatically links it with the provided filename, making management simpler.
- **StatesDirectory**: Identifies the directory for storing serialized object states. Default: Logs/States. This centralizes the storage of object states for easier access and organization.

Logging Preferences

Configure what types of logs are stored:

- **SaveLogs**: Enables or disables saving all log messages to files.
- **SaveStates**: Toggles the saving of serialized object states.
- **FileName**: Determines the filename for stored log messages. Default: Logs.txt. This offers flexibility in naming log files for better organization.

The `FileSettings` class, as defined in the `ByteCobra.Logging.Settings` namespace, includes properties like `MaxFileSize`, `MaxStateDirectorySize`, `LogFilesDirectory`, and `StatesDirectory`. These properties allow for detailed customization of log file behavior, including file size limits, storage locations, and whether to save all logs and states. The addition of the `FileName` property provides further control over log file management, ensuring that logs are not only captured but also organized in a user-friendly manner.

Formatting

When it comes to logs, presentation matters. Whether it's for an editor's immediate view or long-term archival, the manner in which logs are formatted greatly impacts readability and clarity. With Cobra Logging's FormatSettings, developers can stylize and dictate how each log message should appear.

Located within the ByteCobra.Logging.Settings namespace, the FormatSettings class offers a suite of tools to format log messages.

Default Formatting

DefaultFormat: Represents the standard way in which log messages should be formatted. It's flexible and adjusts based on the environment.

- If in the editor: Log messages are colorized using the tag.
- Otherwise: Logs feature a timestamp, tag, and file details.

```
// In-editor format:
<color={log.Color}> {log.Tag} {log.FileInfo.Name}[{log.Line}]</color>: {log.OriginalMessage}

// Standard format:
```

```
[{log.Time.ToString(LogSettings.TimeFormat)}] {log.Tag} {log.FileInfo.Name}[{log.Line}]: {lo
```

Specific Log Formats

While there's a default format provided, you may wish to customize the presentation based on the log's nature.

- DebugFormat: Personalize the presentation of debug logs.
- InfoFormat: Style informational logs uniquely.
- WarningFormat: Highlight warning logs with a distinct format.
- AssertFormat: Ensure assert logs stand out with a tailored format.
- ErrorFormat: Draw attention with a specialized format for error logs.
- FatalFormat: Make fatal logs noticeable with a distinctive style.

Concluding Thoughts

How logs are presented plays a pivotal role in understanding them. A well-formatted log can convey vital information efficiently, while a poorly formatted one can lead to confusion or oversight. Cobra Logging's FormatSettings ensures that developers are equipped with the tools needed to make logs as informative and readable as possible. Remember, it's not just about logging information; it's about conveying it effectively!

Asserts

The `Assert` class is a part of the Cobra Logging package, designed to facilitate assertions within applications. It provides a range of methods for validating conditions, enhancing code robustness by identifying issues early in the development process.

Features

- **Comprehensive Validation**: Supports various assertions including null checks, collection emptiness, numeric comparisons, and more.
- **Informative Logging**: Logs detailed messages upon assertion failure, aiding in quick issue identification.
- **Exception Control**: Optionally throws exceptions on failed assertions to halt execution, signaling critical issues immediately.

Key Methods

- **NotNull**: Ensures a specified parameter is not null.
- **NotEmpty**: Verifies a collection is not empty.
- **NotNullOrWhitespace**: Confirms a string is neither null, empty, nor whitespace only.
- **GreaterThan**: Checks if a value is greater than a specified limit.

- **LessThan**: Verifies a value is less than a specified limit.
- **InRange**: Asserts a value falls within a specified range.
- **NotNullOrEmpty**: Ensures a collection is neither null nor empty.
- **AreEqual**: Verifies the value of a parameter matches an expected value.
- **Contains**: Asserts a collection contains an item that meets a specified condition.
- **DoesNotContain**: Ensures a collection does not contain any item that satisfies a provided condition.
- **Implements**: Checks if an object implements a specified interface.

Usage

Basic Assertions

```
Assert.NotNull(() => myObject, "MyObject must not be null.");
Assert.NotEmpty(() => myList, "The list cannot be empty.");
Assert.NotNullOrWhitespace(() => myString, "Input cannot be blank.");
```

Numeric Comparisons

```
Assert.GreaterThan(() => age, 18, "Age must be over 18.");
Assert.LessThan(() => temperature, 100, "Temperature must be below 100 degrees.");
Assert.InRange(() => score, 0, 100, "Score must be between 0 and 100.");
```

Collection Assertions

```
Assert.NotNullOrEmpty(() => myCollection, "The collection cannot be null or empty.");
Assert.Contains(() => myCollection, item => item.IsValid, "Collection must contain at least
Assert.DoesNotContain(() => myCollection, item => item.IsExpired, "Collection should not co
```

Type Assertions

```
Assert.Implements<IExpectedInterface>(() => myObject, "Object must implement IExpectedInter
```

Advanced Features

- ParameterExpression: Utilizes lambda expressions for parameter checks, enabling precise identification of the assertion's context.
- Custom Messages: Allows specifying custom messages to enhance the information provided by logs on assertion failures.
- Control Over Exceptions: The throwException parameter dictates whether an exception is thrown, offering flexibility based on the criticality of the assertion.

Clock

The `Clock` class offers a simple yet powerful way to measure and log the duration of operations within your applications. It supports both named and default timers, allowing for flexible performance tracking.

Features

- **Start and Stop Timers:** Easily start and stop timers to measure the duration of code execution.
- **Named Timers:** Utilize named timers for tracking multiple operations simultaneously.
- **Default Timer:** Use a default timer for quick and straightforward measurements.
- **Logging:** Option to automatically log the duration of operations upon stopping a timer so that you don't have to do it yourself.

Usage Example

Below are some examples of how you can use the `Clock` class.

Starting and Stopping a Default Timer

```
Clock.Start();
await Task.Delay(1000);
Clock.Stop();
// Console: Process took 00:00:01.0119411
```

Using Named Timers

```
Clock.Start("databaseQuery");
TimeSpan duration = Clock.Stop("databaseQuery", log: false); // Does not log the duration
Log.Info($"Database query took {duration.TotalMilliseconds} ms.");
```

This utility class is designed to provide insights into your application's performance, making it a handy tool for development and debugging.

Memory

The `Memory` class in the ByteCobra.Logging namespace provides a straightforward method for measuring and logging the current memory usage of your application. It reports memory usage in megabytes for easy interpretation.

Features

- **Simple Memory Measurement:** Quickly measure the current memory usage of your application.

- **Automatic Logging:** Option to log the memory usage directly, simplifying monitoring during development or production diagnostics.

Usage

Measure and Log Memory Usage

To measure and automatically log the current memory usage:

```
Memory.Measure();
```

This will output something like: `Current memory usage: 85.32 MB` to the logging system.

Measure Without Logging

If you prefer to measure the memory usage without logging it, you can set the log parameter to false:

```
double memoryUsage = Memory.Measure(log: false);
```

Collect

It is also possible to reclaim unused memory with the `Collect` method. It triggers a manual garbage collection process, aiming to immediately reclaim unused memory. It performs a full garbage collection cycle, waits for any pending finalizers, and then performs another garbage collection to clean up any objects finalized in the first pass.

Usage

```
Memory.Collect();
```

Note: Use this method judiciously, as manual garbage collection can affect application performance. It's recommended to rely on the automatic garbage collection process unless specific scenarios require explicit intervention.