Thread Dispatcher

Thread Dispatcher is an open source tool to pass the execution of a Delegate, Coroutine or Task from a background thread to the main thread, and await its completion or result on the calling thread as needed.

Table of Contents

Installation

Option 1. **Install via Open UPM (recommended)** openupm v5.0.0

- open `Edit/Project Settings/Package Manager`
- add a new Scoped Registry:
    - Name: OpenUPM
    - URL: https://package.openupm.com
    - Scope(s): com.baracuda
- click `Save`

- open `Window/Package Manager`
- click `+`
- click `Add package by name...`
- paste and `Add` `com.baracuda.thread-dispatcher`
- take a look at Setup to see what comes next

Option 2. Install via Git URL

- open `Window/Package Manager`
- click `+`
- click `Add package from git URL`
- paste and `Add` `https://github.com/JohnBaracuda/com.baracuda.thread-dispatcher.git`
- take a look at Setup to see what comes next

Option 3. Get Thread Dispatcher from the Asset Store

Option 4. Download a `.unitypackage` from Releases

> If you like thread dispatcher, consider leaving a good review on the Asset Store regardless of which installation method you chose 😃

Quick Guide

```
// Task is running on a background thread.
public async Task  WorkerTask()
{
    // Dispatch an Action that is executed on the main thread.
    Dispatcher.Invoke(() =>
    {
    // Executed on main thread.
    });

    // Dispatch an Action that is executed on the main thread and await its
completion.
    await Dispatcher.InvokeAsync(() =>
    {
    // Executed on main thread.
    });

    // Dispatch a Func<TResult> that is executed on the main thread and await its
result.
    var player = await Dispatcher.InvokeAsync(() =>
    {
```

```
    // Executed on main thread.
    return  FindObjectOfType<Player>();
    });
}
```

Features

• Dispatch the execution of an Action to the main thread.
• Dispatch the execution of a **Func** to the main thread.
• Dispatch the execution of a **Coroutine** to the main thread.
• Dispatch the execution of a **Task** to the main thread.
• Dispatch the execution of a **Task** to the main thread.
• **Await** the **execution & result** of a delegate or task on the calling thread.
• **Await** the **start** or the **competion** of a Coroutine on the calling thread.
• Asynchronous overloads have **full cancellation support**.
• Multiple **extension methods** to reduce boilder plate code.
• Full C# **source code** included.

Dispatch an Action

you can dispatch the execution of a System.Action to the main thread. Actions are presumably the most common type that will be dispatched and are by default executed during the next available Update, LateUpdate, FixedUpdate or Tick cycle. Use Dispatcher.Invoke(Action action, ExecutionCycle cycle,...) for more control over the cycle in which the Action will be executed.

```
public static void Invoke(Action action);
public static void Invoke(Action action, ExecutionCycle cycle);
```

```
// Tasks are executed on separate background threads.

public Task WorkerTask()
{
    Dispatcher.Invoke(() =>
    {
        // Logic here is executed on the main thread.
    });

    return Task.CompletedTask;
}

public Task WorkerTaskCycle()
{
    var context = ExecutionCycle.FixedUpdate;

    Dispatcher.Invoke(() =>
```

```
    {
        // Logic here is executed on the main thread during the next FixedUpdate.
    }, context);

    return Task.CompletedTask;
}
```

Awaiting a dispatched Action

Await the completion of a dispatched Action by calling Dispatcher.InvokeAsync(Action action) which returns a Task that represents the execution of the dispatched action. The overload Dispatcher.InvokeAsync(Action action, ExecutionCycle cycle) can be used to determine the execution cycle to which the action should be dispatched to.

> ⚠ Exceptions thrown in a dispatched action are returned to the calling thread! If those exceptions are not handled within the passed action itself or on the calling thread, this will result in the thread being cancelled without notice.

```
public static Task InvokeAsync(Action action);
public static Task InvokeAsync(Action action, ExecutionCycle cycle);
```

```
// Tasks are executed on separate background threads.

public async Task WorkerTask()
{
    // Create a try-catch block to handle potential exceptions.
    try
    {
        // Dispatch and await the execution of an anonymous delegate.
        await Dispatcher.InvokeAsync(() =>
        {
            // Logic here is executed on the main thread during the next Update or
Tick call.
        });
        // Logic here is executed after the anonymous delegate has been executed
on the main thread.
    }
    catch (Exception exception)
    {
        // Handle potential exceptions.
    }
```

Cancellation of a dispatched Action

Every Dispatcher.InvokeAsync(Action action, ...) method returns a Task and has an optional overload that accepts a CancellationToken . Since actions return a non-generic task, an additional argument: throwOnCancellation can be passed, that determines whether an exception is thrown in the event of premature cancellation or not. By default, the value of this argument is true and an OperationCanceledException is thrown if the task is cancelled prematurely. If the value is set to false, no exception is thrown and the Task will return without notice.

Note that the throwOnCancellation argument will only prevent an OperationCanceledException from being thrown when the Task is canceled manually. Other exceptions that occur during the execution of the passed delegate are returned to the calling thread nonetheless.

```
public static Task InvokeAsync(Action action, CancellationToken ct, bool
throwOnCancellation = true);
public static Task InvokeAsync(Action action, ExecutionCycle cycle,
CancellationToken ct, bool throwOnCancellation = true);
```

```
// Tasks are executed on separate background threads.

public async Task WorkerTask(CancellationToken ct)
{
    // Prepare a try catch block to catch the exception if the operation is
cancelled.
    try
    {
        // Dispatch and await the execution of an anonymous delegate.
        // This will throw an exception if the operation is cancelled.
        await Dispatcher.InvokeAsync(() =>
        {
            // Logic here is executed on the main thread during the next Update or
Tick call.
        }, ct);
    }
    catch (OperationCanceledException oce)
    {
        // Handle task cancellation.
    }
    catch (Exception exception)
    {
        // Handle other exceptions.
    }


    try
    {
        // Dispatch and await the execution of an anonymous delegate.
        // If cancelled, this will return without an exception.
        await Dispatcher.InvokeAsync(() =>
```

```
        {
            // ...
        }, ct, throwOnCancellation: false);
    }
    catch (Exception exception)
    {
        // Handle exceptions.
    }
}
```

Extension Methods for Actions

Multiple extension methods can be directly called on an Action to reduce boilerplate code. This naturally includes events. Every overload of Dispatcher.Invoke(Action action,...) or Dispatcher.InvokeAsync(Action action,...) has an affiliated extension method. Extension methods for actions will also accept up to four generic parameters. This is done by boxing the generic action within the extension method. Extension methods are located at Baracuda.Threading.DispatchExtensions.

The amount of reduced boilerplate code is modes if you just want to dispatch an action without anything else. It will however become noticeable if you pass additional parameter to your action or if you need to perform a null-check first. Since we are calling the extension method on the action itself we can use the Null-conditional operator ?. to check if the action is null instead of creating a dedicated if statement.

```
public Task ActionExtensionMethodExampleA(Action action)
{
    try
    {
        // Ordinary approach requires additional boilerplate code.
        if (action != null)
        {
            Dispatcher.Invoke(action);
        }

        // Alternative extension method called directly on the Action.
        action?.Dispatch();
    }
    catch (Exception exception)
    {
        // Handle exceptions.
    }

    return Task.CompletedTask;
}
```

```
public Task ActionExtensionMethodExampleB(Action<int> action)
{
```

```
    try
    {
        // Ordinary approach requires additional boilerplate code.
        // Note that we are using an alternative approach to check if the action
is null.
        Dispatcher.Invoke(() =>
        {
            action?.Invoke(1337);
        });

        // Alternative extension method called directly on the Action.
        action?.Dispatch(1337);
    }
    catch (Exception exception)
    {
        // Handle exceptions.
    }
    return Task.CompletedTask;
}
```

```
public async Task ActionExtensionMethodExampleC(Action action, CancellationToken
ct)
{
    try
    {
        // Ordinary approach requires additional boilerplate code.
        if (action != null)
        {
            await Dispatcher.InvokeAsync(action, ct);
        }

        // Alternative extension method called directly on the Action.
        // Extension Method will return a completed task if the action is null.
        await action.DispatchAsync(ct);
    }
    catch (OperationCanceledException oce)
    {
        // Handle exceptions.
    }
    catch (Exception exception)
    {
        // Handle exceptions.
    }
}
```

```
public static void Dispatch(this Action action);
public static void Dispatch<T>(this Action<T> action);
public static void Dispatch<T, S>(this Action<T, S> action);
public static void Dispatch<T, S, U>(this Action<T, S, U> action);
```

```
public static void Dispatch<T, S, U, V>(this Action<T, S, U, V> action);

public static void Dispatch(this Action action, ExecutionCycle cycle);
public static void Dispatch<T>(this Action<T> action, ExecutionCycle cycle);
public static void Dispatch<T, S>(this Action<T, S> action, ExecutionCycle cycle);
public static void Dispatch<T, S, U>(this Action<T, S, U> action, ExecutionCycle
cycle);
public static void Dispatch<T, S, U, V>(this Action<T, S, U, V> action,
ExecutionCycle cycle);

public static Task DispatchAsync(this Action action);
public static Task DispatchAsync(this Action action, ExecutionCycle cycle);
public static Task DispatchAsync(this Action action, CancellationToken ct, bool
throwOnCancellation = true);
public static Task DispatchAsync(this Action action, ExecutionCycle cycle,
CancellationToken ct, bool throwOnCancellation = true);
```

Func<TResultResult>

Dispatch and await a Func

Dispatch a Func<TResultResult> to the main thread and await its result on the calling thread using Dispatcher.InvokeAsync(Func<TResultResult> func). This method returns a Task<TResultResult> object which yields the result of the delegate. Pass in an optional ExecutionCycle argument when calling this method to determine when the delegate will be executed.

> ⚠ Exceptions thrown in a dispatched delegate are returned to the calling thread when awaited! If those exceptions are not handled within the passed delegate itself or on the calling thread, this will result in the thread being cancelled without notice.

```
public static Task<TResult> InvokeAsync(Func<TResult> func);
public static Task<TResult> InvokeAsync(Func<TResult> func, ExecutionCycle cycle);
```

```
public async Task WorkerTask()
{
    try
    {
        // Find a CharacterController on the main thread and return it to the
calling thread.
        var player = await Dispatcher.InvokeAsync(() =>
        {
            return FindObjectOfType<CharacterController>();
        });
```

```
    }
    catch (Exception exception)
    {
        // Handle potential exceptions.
    }
}
```

Cancellation of a dispatched Func

Every InvokeAsync(Func func, ...) method returns a Task and has an optional overload that accepts a CancellationToken. If the task is cancelled prematurely, an OperationCanceledException is thrown. Always use a try-catch block if you are passing a CancellationToken!

```
public static Task<TResult> InvokeAsync(Func<TResult> func, CancellationToken ct);
public static Task<TResult> InvokeAsync(Func<TResult> func, ExecutionCycle cycle,
CancellationToken ct);
```

```
// Tasks are not executed on the main thread but on separate background threads.

public async Task WorkerTask(CancellationToken ct)
{
    // Prepare a try catch block to catch the exception if the operation is
cancelled.

    try
    {
        // Dispatch and await the result of the passed delegate.
        // This will throw an exception if the operation is cancelled.
        Canvas result = await Dispatcher.InvokeAsync(() =>
        {
            // Find a Canvas on the main thread and return it to the calling
thread.
            return Object.FindObjectOfType<Canvas>();
        }, ct);

        // Executed logic that requires the Canvas...
    }
    catch (OperationCanceledException oce)
    {
        // Handle task cancellation.
    }
    catch (Exception exception)
    {
        // Handle other exceptions.
    }
}
```

Extension Methods for Func

You can call multiple extension methods directly on a Func<TResult> to reduce boilerplate code.

```
public async Task FuncExtensionMethodExampleA<T>(Func<T> func, CancellationToken
ct)
{
    try
    {
        // We must manually check if func is null because we cannot use the Null-
conditional operator ?
        // if we are dealing with tasks that yield a return value.
        if (func != null)
        {
            // Ordinary approach requires some additional boilerplate code.
            var result1 = await Dispatcher.InvokeAsync(func, ct);

            // Alternative extension method called directly on the delegate.
            var result2 = await func.DispatchAsync(ct);
        }
    }
    catch (OperationCanceledException oce)
    {
        // Handle task cancellation.
    }
    catch (Exception exception)
    {
        // Handle exceptions.
    }
}
```

```
public static Task<TResult> DispatchAsync(this Func<TResult> func);
public static Task<TResult> DispatchAsync(this Func<TResult> func, ExecutionCycle
cycle);
public static Task<TResult> DispatchAsync(this Func<TResult> func,
CancellationToken ct);
public static Task<TResult> DispatchAsync(this Func<TResult> func, ExecutionCycle
cycle, CancellationToken ct);
```

Coroutines

You can dispatch an IEnumerator to be executed as a Coroutine on the main thread. You can determine the target MonoBehaviour on which the Coroutine will run. If no target MonoBehaviour is passed, the coroutine

will run on the Dispatcher Scene Component.

```
public static void Invoke(IEnumerator enumerator);
public static void Invoke(IEnumerator enumerator, MonoBehaviour target);
public static void Invoke(IEnumerator enumerator, ExecutionCycle cycle);
public static void Invoke(IEnumerator enumerator, ExecutionCycle cycle,
MonoBehaviour target);
```

```
public Task WorkerTask()
{
    Dispatcher.Invoke(ExampleCoroutine());

    return Task.CompletedTask;
}

// Coroutines can only run on the main thread.
private IEnumerator ExampleCoroutine()
{
    yield return null;
    // ...
}
```

Awaiting a dispatched Coroutine

There are two options when awaiting a dispatched Coroutine. You can either Await the start of a Coroutine using InvokeAsyncAwaitStart which returns a Task that when awaited will yield the representative Coroutine after it was successfully started on the main thread. Or Await the completion of a Coroutine using InvokeAsyncAwaitCompletion which yields no result but can be awaited indefinitely until the Coroutine has completed on the main thread.

> ⚠ Version 2.0.0 enables you to determine whether the awaited Task returns when the dispatched Coroutine was started, yielding the Coroutine object as a return value or when it has completed, yielding no return value; by using InvokeAsyncAwaitStart(IEnumerator enumerator...) or InvokeAsyncAwaitCompletion(IEnumerator enumerator...) marking InvokeAsync(IEnumerator enumerator...) as Obsolete!

> ⚠ Exceptions thrown during both the dispatchment of a Coroutine or the execution of a Coroutine are returned to the calling thread. If those exceptions are not handled on the calling thread, the thread will be cancelled without notice.

Await the start of a Coroutine

The method Dispatcher.InvokeAsyncAwaitStart(IEnumerator enumerator,...) returns a Task that when awaited yields the representative Coroutine object that was just started on the main thread. You can cache this object and use it to stop the Coroutine using Dispatcher.CancelCoroutine(Coroutine coroutine,...), a method which also has asynchronous overloads and will stop a Coroutine running on the dispatcher itself. Note that this method will only stop Coroutines that are running on the dispatcher itself, aka Coroutines that were dispatched without explicitly passing a target MonoBehaviour for the Coroutine .

```
public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator);
public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator,
MonoBehaviour target);
public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator,
CancellationToken ct);
public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator,
MonoBehaviour target, CancellationToken ct);

public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator,
ExecutionCycle cycle);
public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target);
public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator,
ExecutionCycle cycle, CancellationToken ct);
public static Task<Coroutine> InvokeAsyncAwaitStart(IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target, CancellationToken ct);
```

```
// Task is running on a background thread.
public async Task WorkerTask()
{
    try
    {
        // Get a random value of milliseconds to wait.
        var random = await Dispatcher.InvokeAsync(() =>
UnityEngine.Random.Range(0, 2000));

        // Start a coroutine running on the main thread.
        Coroutine coroutine = await
Dispatcher.InvokeAsyncAwaitStart(ExampleCoroutine());

        // Simulate asynchronous work.
        await Task.Delay(random);

        // Cancel the coroutine after asynchronous work has completed.
        await Dispatcher.CancelCoroutineAsync(coroutine);

        // ...
    }
    catch (Exception exception)
    {
        // Handle potential exceptions.
        Debug.LogException(exception);
```

```
    }
}

// Coroutine has a ~50% chance of being stopped before it completes.
private IEnumerator ExampleCoroutine()
{
    Debug.Log("Start of Coroutine");
    yield return new WaitForSeconds(1f);
    Debug.Log("End of Coroutine");
}
```

Await the completion of a Coroutine

You can not only await the start of a Coroutine but also its completion. The method Dispatcher.InvokeAsyncAwaitCompletion(IEnumerator enumerator,...) returns a Task that can be awaited on the calling thread and returns when the dispatched Coroutine has completed on the main thread. To avoid that the calling thread is awaiting the completion indefinitely, it is very important to receive notification if the Coroutine cannot complete, either because of an exception or because the coroutine was stopped. For this reason every dispatched coroutine must be wrapped in another exception sensitive coroutine, that will catch and return exceptions to the calling thread. Additionally, because the life of a Coroutine is bound to a target MonoBehaviour and a stopped Coroutine will just cease to exist without telling anybody, the target MonoBehaviour must be monitored to receive notice if it is disabled. Those essential operations can become very expensive should they occur in large quantities.

> ⚠ For reasons stated above, awaiting the completion of a dispatched coroutine can become an expensive operation should it occur in large quantities. Please be aware of this and avoid unnecessary usages of this feature if possible, especially in performance critical environments.

```
public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator, bool
throwExceptions = true);
public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator,
CancellationToken ct, bool throwExceptions = true);
public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator,
MonoBehaviour target, bool throwExceptions = true);
public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator,
MonoBehaviour target, CancellationToken ct, bool throwExceptions = true);

public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator,
ExecutionCycle cycle, bool throwExceptions = true);
public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator,
ExecutionCycle cycle, CancellationToken ct, bool throwExceptions = true);
public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target, bool throwExceptions = true);
public static Task InvokeAsyncAwaitCompletion(IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target, CancellationToken ct, bool
throwExceptions = true);
```

```
public async Task WorkerTask()
{
    try
    {
        // Create a cancellation token source.
        var cts = new CancellationTokenSource();

        // Cancel the operation after one second.
        cts.CancelAfter(1000);

        // Start a coroutine and await its completion.
        await Dispatcher.InvokeAsyncAwaitCompletion(ExampleCoroutine(),
cts.Token);

        // ...
    }
    catch (OperationCanceledException operationCanceledException)
    {
        // This exception will occur after one second if the coroutine hasn't
completed yet, which has a chance of ~50%.
        Debug.Log("Operation Cancelled!");
    }
    catch (BehaviourDisabledException behaviourDisabledException)
    {
        // This exception will occur if the coroutines target behaviour is
disabled while the coroutine is still running.
        Debug.Log("Behaviour disabled!");
    }
    catch (Exception exception)
    {
        // Handle other potential exceptions that occur during the execution of
the coroutine.
        Debug.LogError(exception);
    }
}

// Coroutines are only allowed to be executed on the main thread.
private IEnumerator ExampleCoroutine()
{
    Debug.Log("Start of Coroutine");
    yield return new WaitForSeconds(UnityEngine.Random.Range(0f, 2f));
    Debug.Log("End of Coroutine");
}
```

Extension Methods for Coroutines

You can call multiple extension methods directly on an IEnumerator to reduce boilerplate code. Every overload of Dispatcher.Invoke(IEnumerator enumerator,...) or Dispatcher.InvokeAsync(IEnumerator enumerator,...) has an affiliated extension method.

```
public async Task CoroutineExtensionMethodExampleA(MonoBehaviour target)
{
    try
    {
        // Ordinary approach to dispatch a coroutine.
        Dispatcher.Invoke(ExampleCoroutine(5.0f), target);

        // Alternative extension method.
        ExampleCoroutine(5.0f).Dispatch(target);
    }
    catch (Exception exception)
    {
        // Handle potential exceptions.
    }
}

private IEnumerator ExampleCoroutine(float delay)
{
    yield return new WaitForSeconds(delay);
}
```

```
public static void Dispatch(this IEnumerator enumerator);
public static void Dispatch(this IEnumerator enumerator, MonoBehaviour target);
public static void Dispatch(this IEnumerator enumerator, ExecutionCycle cycle);
public static void Dispatch(this IEnumerator enumerator, ExecutionCycle cycle,
MonoBehaviour target);

public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator
enumerator);
public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator enumerator,
MonoBehaviour target);
public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator enumerator,
CancellationToken ct);
public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator enumerator,
MonoBehaviour target, CancellationToken ct);

public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator enumerator,
ExecutionCycle cycle);
public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target);
public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator enumerator,
ExecutionCycle cycle, CancellationToken ct);
public static Task<Coroutine> DispatchAsyncAwaitStart(this IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target, CancellationToken ct);

public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator, bool
throwExceptions = true);
public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator,
MonoBehaviour target, bool throwExceptions = true);
public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator,
```

```
CancellationToken ct, bool throwExceptions = true);
public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator,
MonoBehaviour target, CancellationToken ct, bool throwExceptions = true);

public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator,
ExecutionCycle cycle, bool throwExceptions = true);
public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target, bool throwExceptions = true);
public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator,
ExecutionCycle cycle, CancellationToken ct, bool throwExceptions = true);
public static Task DispatchAsyncAwaitCompletion(this IEnumerator enumerator,
ExecutionCycle cycle, MonoBehaviour target, CancellationToken ct, bool
throwExceptions = true);
```

Task

Dispatch work encapsulated in a Func<Task> which is then preformed on the main thread. When passing in an optional CancellationToken ct, it is forwarded to the Task wrapped in function which must accept it as an argument.

```
public static void Invoke(Func<Task> function);
public static void Invoke(Func<Task> function, ExecutionCycle cycle);
public static void Invoke(Func<CancellationToken, Task> function,
CancellationToken ct, bool throwOnCancellation = true);
public static void Invoke(Func<CancellationToken, Task> function, ExecutionCycle
cycle, CancellationToken ct, bool throwOnCancellation = true);
```

```
// Provides cancellation token.
private CancellationTokenSource cts = new CancellationTokenSource();

// Running on a background thread.
public Task WorkerTask()
{
    // The cancellation token from cts is passed along to MainThreadTask
    // when it is run on the main thread.
    Dispatcher.Invoke(MainThreadTask, cts.Token);

    // using cts.Cancel() will cancel the work done by MainThreadTask.
}

// Running on the main thread.
private async Task MainThreadTask(CancellationToken ct)
{
    // Simulating async work on the main thread and returning a result.
    await Task.Delay(300, ct);
}
```

Awaiting a dispatched Task

Dispatch work encapsulated in a Func<Task> which is then preformed on the main thread. Await the completion of the passed operation by awaiting the Task handle returned by the method call. When passing in an optional CancellationToken ct, it is forwarded to the Task wrapped in function which must accept it as an argument.

> ⚠ Exceptions thrown in a dispatched task are returned to the calling thread if the completion or the result of the dispatched work is awaited! If those exceptions are not handled within the passed task itself this will result in the thread being cancelled without notice.

```
public static Task InvokeAsync(Func<Task> function);
public static Task InvokeAsync(Func<Task> function, ExecutionCycle cycle);
public static Task InvokeAsync(Func<CancellationToken, Task> function,
CancellationToken ct, bool throwOnCancellation = true);
public static Task InvokeAsync(Func<CancellationToken, Task> function,
ExecutionCycle cycle, CancellationToken ct, bool throwOnCancellation = true);
```

```
// Running on a background thread.
public async Task WorkerTask(CancellationToken ct)
{
    try
    {
        // Perform and await work on the main thread.
        await Dispatcher.InvokeAsync(MainThreadTask, cts.Token);

        // ...
    }
    catch (OperationCanceledException canceledException)
    {
     // Task was canceled.
    }
    catch (Exception exception)
    {
        // Another exception occurred.
    }
}

// Running on the main thread.
private async Task MainThreadTask(CancellationToken ct)
{
    // Simulating async work on the main thread.
    await Task.Delay(300, ct);
}
```

Awaiting a dispatched Task TResult

Dispatch work encapsulated in a Func<Task<TResult>> which is then preformed on the main thread. Await the result of the passed operation by awaiting the Task<TResult> handle returned by the method call. When passing in an optional CancellationToken ct, it is forwarded to the Task wrapped in function which must accept it as an argument.

> ⚠ Exceptions thrown in a dispatched task are returned to the calling thread if the completion or the result of the dispatched work is awaited! If those exceptions are not handled within the passed task itself this will result in the thread being cancelled without notice.

```
public static Task<TResult> InvokeAsync<TResult>(Func<Task<TResult>> function);
public static Task<TResult> InvokeAsync<TResult>(Func<Task<TResult>> function,
ExecutionCycle cycle);
public static Task<TResult> InvokeAsync<TResult>(Func<CancellationToken,
Task<TResult>> function, CancellationToken ct, bool throwOnCancellation = true);
public static Task<TResult> InvokeAsync<TResult>(Func<CancellationToken,
Task<TResult>> function, ExecutionCycle cycle, CancellationToken ct, bool
throwOnCancellation = true);
```

```
// Running on a background thread.
public async Task WorkerTask(CancellationToken ct)
{
    try
    {
        // Perform and await work on the main thread.
        int result = await Dispatcher.InvokeAsync(MainThreadTask, cts.Token);

        // Do something with result.
    }
    catch (OperationCanceledException canceledException)
    {
        // Task was canceled.
    }
    catch (Exception exception)
    {
        // Another exception occurred.
    }
}

// Running on the main thread.
private async Task<int> MainThreadTask(CancellationToken ct)
{
    // Simulating async work on the main thread and return a result.
    await Task.Delay(300, ct);
    return 1337;
}
```

Extension Methods for Task and Task TResult

```
public static void Invoke(this Func<Task> function);
public static void Invoke(this Func<Task> function, ExecutionCycle cycle);
public static void Invoke(this Func<CancellationToken, Task> function,
CancellationToken ct, bool throwOnCancellation = true);
public static void Invoke(this Func<CancellationToken, Task> function,
ExecutionCycle cycle, CancellationToken ct, bool throwOnCancellation = true);

public static Task InvokeAsync(this Func<Task> function);
public static Task InvokeAsync(this Func<Task> function, ExecutionCycle cycle);
public static Task InvokeAsync(this Func<CancellationToken, Task> function,
CancellationToken ct, bool throwOnCancellation = true);
public static Task InvokeAsync(this Func<CancellationToken, Task> function,
ExecutionCycle cycle, CancellationToken ct, bool throwOnCancellation = true);

public static Task<TResult> InvokeAsync<TResult>(this Func<Task<TResult>>
function);
public static Task<TResult> InvokeAsync<TResult>(this Func<Task<TResult>>
function, ExecutionCycle cycle);
public static Task<TResult> InvokeAsync<TResult>(this Func<CancellationToken,
Task<TResult>> function, CancellationToken ct, bool throwOnCancellation = true);
public static Task<TResult> InvokeAsync<TResult>(this Func<CancellationToken,
Task<TResult>> function, ExecutionCycle cycle, CancellationToken ct, bool
throwOnCancellation = true);
```

Execution Cycle

You can determine the exact execution cycle in which a passed delegate or coroutine is invoked on the main thread by passing an optional ExecutionCycle argument when dispatching it.

```
public enum ExecutionCycle
{
    // Executed at the beginning of the next Update call.
    Update = 1,
    // Executed at the beginning of the next LateUpdate call.
    LateUpdate = 2,
    // Executed at the beginning of the next FixedUpdate call.
    FixedUpdate = 3,

#if UNITY_EDITOR
    // Executed at the beginning of the next editor update call.
    EditorUpdate = 5,
#endif
}
```

Miscellaneous

You can use the Dispatcher to validate if a method is currently running on the main thread or not by calling Dispatcher.IsMainThread(). This method will return true if it is called from the main thread.

```
// It is unknown if the task is executed on the main thread or not.

public Task WorkerTask()
{
    if(Dispatcher.IsMainThread() == true)
    {
        // Work() can be called directly because the current execution is already
happening on the main thread.
        Work();
    }
    else
    {
        // Work() must first be dispatched to the main thread and will be called
during the next available
        // Update() or Tick() cycle.
        Dispatcher.Invoke(Work);
    }

    return Task.CompletedTask;
}

private void Work()
{
    // Logic here is only allowed to be executed on the main thread!
}
```

Support Me

I spend a lot of time working on this and other free assets to make sure as many people as possible can use my tools regardless of their financial status. Any kind of support I get helps me keep doing this, so consider leaving a star ☆ making a donation or follow me on my socials to support me ♡

- Donation (PayPal.me)
- Linktree
- Twitter
- Itch