Super Invoke

Time-control your code

SUPERINVOKE

Super Invoke is handcrafted by Jacob Games
Stay in touch to get news and updates

Follow **Jacob Games** on Twitter

JACOB
GAMES

TABLE OF CONTENTS

1 - Introduction

1.1 - How to use Super Invoke

Super Invoke is a tool that allows you to easily time-control any piece of code.
You can delay parameterless methods as well as parameterized methods. You can also create high-readable sequences with methods and custom delays between methods, and much more.

To use Super Invoke you need to add the following using statement to your source file:

```
using JacobGames.SuperInvoke;
```

All APIs are offered through the *SuperInvoke* class.

1.2 - Initialization (optional)

Super Invoke will automatically initialize its internal structure the first time you use it. In case you prefer to take control of this operation use:

```
SuperInvoke.Init();
```

An optional bool argument is used to set the DontDestroyOnLoad flag on the SuperInvoke game object manager.

1.3 - Beware of C# limitations

Please, go to section 8 of this document to read how to overcome C# limitations when using Super Invoke.

1.4 - Online version

A live version of this document is available online.

2 - Delays

2.1 - Delay a parameterless method

To delay a parameterless method simply use *SuperInvoke.Run*, specifying

- The method
- The delay
- An optional tag (discussed later)

```
SuperInvoke.Run( ()=> PlaySound(), 1f);
```

You may use the contracted form as well:

```
SuperInvoke.Run(PlaySound, 1f);
```

And you may also specify the delay before the method:

```
SuperInvoke.Run(1f, PlaySound);
```

2.2 - Delay a method with parameters

To delay a method with parameter use the exact same syntax used for parameterless methods. Contracted form cannot be used though.

```
SuperInvoke.Run( ()=> PlaySound(VictoryJingle), 1f);
```

2.3 - Delay any block of code

You may actually delay any block of code if you need to. Any line in the following block of code will be executed after 5 seconds:

```
SuperInvoke.Run( ()=> {
                      Debug.Log("Prefab instantiation.");
                      Instantiate(prefab);
                      },
                 5f);
```

3 - Sequences

3.1 - Create sequences

To use a sequence you must first create an object of type *ISuperInvokeSequence* :

```
ISuperInvokeSequence sequence = SuperInvoke.CreateSequence();
```

The method *CreateSequence* is a factory: every time it is called it returns a different sequence object.

3.2 - Compose sequences

Once you've created the sequence object you may compose your sequence adding methods and delays:

```
sequence.AddMethod(RemovePinAndThrowGrenade)
        .AddDelay(5f)
        .AddMethod(ExplodeGrenade);
```

3.3 - Run sequences

After you composed the sequence, you can run it as shown below:

```
sequence.Run();
```

Equivalent way to compose the previous sequence

The previous sequence can also be written in the following way:

```
sequence.AddMethod(RemovePinAndThrowGrenade)
        .AddMethod(5f, ExplodeGrenade)
```

Depending on your preferences, you can use both ways interchangeably.
When you create sequences with several methods and delays, using *AddDelay* may be more readable.

3.4 - Add a delay to the entire sequence

Sometimes, you may need to delay the execution of the entire sequence after you have composed it. You can accomplish this task in three different and equivalent ways.

Method 1 - Add a starting delay

```
sequence.AddDelay(2f)
        .AddMethod(RemovePinAndThrowGrenade)
        .AddDelay(5f)
        .AddMethod(ExplodeGrenade)
        .Run();
```

Method 2 - Add a delay to the first method

```
sequence.AddMethod(RemovePinAndThrowGrenade, 2f)
        .AddDelay(5f)
        .AddMethod(ExplodeGrenade)
        .Run();
```

Method 3 - Pass the delay to *Run*

```
sequence.AddMethod(RemovePinAndThrowGrenade)
        .AddDelay(5f)
        .AddMethod(ExplodeGrenade)
        .Run(2f);
```

Method 3 might be the most readable one because you add a global delay in *Run* which do not interfere with the delays of your composed sequence.

4 - RunRepeat

4.1 - Repeats

To repeat the execution of a method or any piece of code use *SuperInvoke.RunRepeat* with the following parameters:

```
SuperInvoke.RunRepeat(Delay, RepeatRate, Repeats, Method);
```

- *Delay:* an initial delay, in seconds, after which the repeats will start
- *RepeatRate*: the time, in seconds, between each repeat
- *Repeats*: the number of repeats (either a fixed number or infinite times)
- *Method:* actual code to repeat

4.2 - RunRepeat - Example 1 - Specific number of repeats

Play a sound every second for 5 times, with no initial delay:

```
SuperInvoke.RunRepeat(0f, 1f, 5, PlaySound);
```

4.3 - RunRepeat - Example 2 - Infinite repeats

To repeat ad infinitum you may use SuperInvoke.INFINITY or directly write "-1" as the *Repeats* parameter.

```
SuperInvoke.RunRepeat(0f, 1f, SuperInvoke.INFINITY, PlaySound);
```

N. B.: RunRepeat works for sequences too.

5 - Job and JobRepeat

5.1 - What is a Job

Any execution of *Run*, either through *SuperInvoke.Run* or *sequence.Run* returns an *IJob* object. Any execution of *RunRepeat*, either through *SuperInvoke.RunRepeat* or *sequence.RunRepeat* returns an *IJobRepeat* object.

A job (and a job-repeat) is a conceptual object made of its delay(s) and its delayed action. The following points are related to both Jobs and JobRepeats unless specified otherwise.

5.2 - Job states

A job, and a job-repeat, can be in one of the following states:

- Scheduled
- Paused
- Killed
- Completed

Scheduled Job

This is the default state for a job when it is initially run. In this state, the delay(s) of the job are elapsing. When the delay(s) are elapsed, the job is still considered as scheduled if its method(s) are currently being executed and not yet completed.

Paused Job

A job is paused when its method *Pause* is called. A pause essentially freezes the delays. If *Resume* is called the job goes back in the "scheduled" state.

Killed Job

A job is killed when its method *Kill* is called. It means it won't be executed at all. If it is called on a sequence job, the sequence will be abruptly interrupted.

Completed Job

A job is completed when its delay(s) has been elapsed and all its related code has been completely executed.

5.3 - Job - OnComplete

OnComplete is a callback you may use to be notified when the job has been completed, i.e. its delay(s) has elapsed and the action has been fully executed. It is especially useful when you cannot predict how much time it will take to execute the delayed action.

Example:

```
SuperInvoke.Run(2f, CallServer).OnComplete(PlaySound);
```

5.4 - Job - Pause & Resume

You may need to pause a particular job and resume it after an external event happens. Use the *Pause* method to pause the delay of the job which puts it into the Paused state, resume it using *Resume* to put it back into the Scheduled state.

If the delay of a job has elapsed and its method is currently executing, calling *Pause* won't have any effect.

Calling *Pause* on a job created by calling *sequence.Run* will have the effect of freezing the sequence in the middle if any delay of the sequence.

5.5 - Job - Kill

Kills the job.

5.6 - Job - KillOnDestroy

KillOnDestroy is useful when you run a job in a MonoBehaviour attached to a game object that might be destroyed before the delay of the job elapses.

Passing the game object to *KillOnDestroy* assures you that the job will be killed if it is not yet been completed and the game object is destroyed.

Example:

```
SuperInvoke.Run(2f, CallServer).KillOnDestroy(this);
```

5.8 - Job - PauseOnDisable

PauseOnDisable automatically pauses the job when the game object is disabled and, by default, it automatically resumes the job when the game object is subsequently enabled.

5.9 - JobRepeat-only features

IJobRepeat offers all the IJob methods because it extends IJob. Additional methods for IJobRepeat are:

- GetCompletedRepeats
- GetRemainingRepeats
- GetTotalRepeats

6 - Tags

6.1 - What are Tags

Tags are strings that are used to group multiple jobs. When you execute a *Run* or a *RunRepeat* you can optionally choose to tag the job with a chosen string.

6.2 - SuperInvoke.CreateTag()

SuperInvoke.CreateTag() is a helpful factory method that returns a different string any time it is called. It is useful when multiple and different tags are needed at run-time.
To further decreased the probability of customly using an identical string to one created by *SuperInvoke.CreateTag()*, all strings created by *SuperInvoke.CreateTag()* start with the upper-case sigma symbol " Σ ".

6.3 - Important note for versions compatibility

In version 3.0 we have simplified the tag feature. Previously, there were two ways to tag a job: either through using a string or using an *ISuperInvokeTag* object. We have eliminated the *ISuperInvokeTag* interface to reduce the complexity.
If you used *ISuperInvokeTag* in your project simply search and replace the string "ISuperInvokeTag" with the string "string" and everything will work the same.

7 - Additional global features of *SuperInvoke*

7.1 - Global Killing

Killing a singular job has been discussed previously. You can *kill a group* of jobs using tags, or you can *kill all* the current job in the project, or you can *kill all except* essentially choosing which jobs must remain alive using:

- *SuperInvoke.Kill(tags)*
- *SuperInvoke.KillAll()*
- *SuperInvoke.KillAllExcept(tags)*

7.2 - Global Pause & Resume

Pausing and resuming a singular job has been discussed previously. With tags, you can pause and resume group of jobs using:

- *SuperInvoke.Pause(tags)*
- *SuperInvoke.Resume(tags)*

7.3 - Skip Frames

The feature SkipFrames allows you to easily delay the execution of any method by specifying the number of frames to wait:

```
SuperInvoke.SkipFrames(100, rotateCube);
```

8 - C# Limitations that affect Super Invoke

ATTENTION
Due to c# internal structure regarding actions and scopes, you have to take a safe step when you use SuperInvoke with parameterized methods.
The following examples show the issue:

```
int amount = 100;
SuperInvoke.Run(()=> withdraw(amount), 1f);
amount = 500;
```

How much will be actually withdrawn?
The answer is 500 because **the variable *amount* was changed before the delayed method *withdraw* was actually executed**.
Essentially, the method *withdraw* can only know the last value of the variable *amount*.

For loops are also affected by the c# limitation:

```
for(int i = 1; i <= 3; i++) {
      SuperInvoke.Run(()=> Debug.Log(i), 1f);
}
```

The log output is:

```
3
3
3
```

It always logs "3" because the delayed method (*Log* in this case) can only know the last value of *i,* which was 3, when SuperInvoke.Run was actually executed (after 1s in this case).
To overcome the c# limitation you need to specify a new variable inside the body of the loop in which you assign the current loop value of the variable *i*:

```
for(int i = 1; i <= 3; i++) {
      int k = i;
      SuperInvoke.Run(()=> Debug.Log(k), 1f);
}
```

The log output is:

```
1
2
3
```