Super Invoke

Code delays in simple ways

Super Invoke is made by Jacob Games
Stay in touch with us to get news on updates and new assets!

JACOB GAMES

Follow **Jacob Games** on Twitter

Like **Jacob Games** on Facebook

TABLE OF CONTENTS

1 - Introduction

Super Invoke is a tool that allows you to easily delay 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.

1.1 - Online version

You can also read this document online.

2 - How to use Super Invoke

2.1 - Get started

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

```
using JacobGames.SuperInvoke;
```

Anything else is done using the *SuperInvoke* class and its methods.

2.2 - !!! Attention to c# limitations !!!

Please, go to section # 6 of this document to read how to overcome c# limitations when using Super Invoke.

3.1 - Delay a parameterless method

To delay a parameterless method use the *Run* method of the *SuperInvoke* class.
The first parameter of *Run* must be the method you want to delay, specified through a lambda expression; the second parameter is the delay in seconds expressed as float. The following line of code shows the syntax:

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

Contracted form for parameterless methods

SuperInvoke works with lambda expressions. When you delay parameterless methods you can use the contracted form as shown below:

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

3.2 - Delay a parameterized method

To delay a parameterized method you use the exact same syntax used for parameterless methods. The only difference is that the contracted form cannot be used for parameterized methods.

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

3.3 - Delay any block of code

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

```
SuperInvoke.Run( ()=> {
                      Debug.Log("A debug log.");
                      Debug.Log("Another debug log.");
                      },
                 1f);
```

4 - Sequences

4.1 - 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.

4.2 - Compose sequences

After you create the sequence object you can compose your sequence adding methods and delays:

```
sequence.AddMethod(parameterlessMethod)
        .AddDelay(2f)
        .AddMethod( ()=> parameterizedMethod(aParameter));
```

The methods *AddMethod* and *AddDelay* return an *ISuperInvokeSequence* object which allows you to write high readable code.

4.3 - Run sequences

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

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

Equivalent way to compose the previous sequence

An equivalent way to compose the previous sequence is shown below:

```
sequence.AddMethod(parameterlessMethod)
        .AddMethod( ()=> parameterizedMethod(aParameter), 2f);
```

Depending on your preferences, you can use both ways interchangeably.
When you create sequences with many methods and delays, the first way (explicitly specifying the delay with *AddDelay*) may be more readable.

4.4 - Add a delay to the entire sequence

Sometimes, you compose your sequence with methods and delays and then you might need to run the entire sequence with a delay. You can accomplish this task in three different and equivalent ways.

Method 1 - Add a starting delay

```
sequence.AddDelay(5f)
        .AddMethod(parameterlessMethod)
        .AddDelay(2f)
        .AddMethod( ()=> parameterizedMethod(aParameter))
        .Run();
```

Method 2 - Add a delay to the first method

```
sequence.AddMethod(parameterlessMethod, 5f)
        .AddDelay(2f)
        .AddMethod( ()=> parameterizedMethod(aParameter))
        .Run();
```

Method 3 - Pass the delay to *Run*

```
sequence.AddMethod(parameterlessMethod)
        .AddDelay(2f)
        .AddMethod( ()=> parameterizedMethod(aParameter))
        .Run(5f);
```

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.

5 - Tagging and Killing

WARNING
Tags used by Super Invoke are different from Unity built-in tags. Apart the concept of tagging things, they technically have nothing in common.

After you run a method or a sequence you might need to kill them, i.e. you declare that you don't need to call the method you previously delayed.
You can only kill methods and sequences you have delayed with the method *Run*.
There are different ways to kill methods and sequences.

5.1 - Two interchangeably types of tags

Tags used by SuperInvoke can be of two interchangeably types:

- string
- SuperInvokeTag

The string type might be easier, faster, and more readable. At Jacob Games, we use SuperInvoke for our own projects and in some cases a SuperInvokeTag objects are more suitable than a string tag and it might happen to be useful for you too.

5.2 - Kill a specific method or sequence

To kill a specific method or sequence you need to perform two steps:

1. Tag the method or sequence when you run it;
2. Call SuperInvoke.Kill passing the previously specified tag:

Kill a delayed method using a string tag

```
SuperInvoke.Run(parameterlessMethod, 1f, "aStringTag");
...
SuperInvoke.Kill("aStringTag");
```

Kill a delayed sequence using a string tag

First create and compose your sequence. When you run it pass the string tag which will then be used in the *Kill* method.

```
...
sequence.Run("aStringTag");
...
SuperInvoke.Kill("aStringTag");
```

Kill a delayed method using SuperInvokeTag

```
SuperInvokeTag siTag = SuperInvoke.CreateTag();
...
SuperInvoke.Run(parameterlessMethod, 1f, siTag);
...
SuperInvoke.Kill(siTag);
```

CreateTag is a factory method: every time it is called it returns a different tag object.

Kill a delayed sequence using SuperInvokeTag

To kill a sequence with a SuperInvokeTag use the same syntax of killing a sequence with a string tag:

```
SuperInvokeTag siTag = SuperInvoke.CreateTag();
...
sequence.Run(siTag);
...
SuperInvoke.Kill(siTag);
```

5.3 - SuperInvokeTag vs. string tag

A SuperInvokeTag object might be more useful than a string tag in the following example scenario:

You have a MonoBehaviour script in which you called *SuperInvoke.CreateTag()* in Start. The script will be attached to N different GameObjects in your scene. Hence, at run time every GameObject on which the script is attached will have a different tag.
In the same script you delay some methods and you create an API method to kill the delayed methods in the script.
Using this structure you can individually choose which GameObject should kill their delayed methods. You might have some complex rule for choosing them. Using a different SuperInvokeTag which was created at run time allows you to manage complex situations.

You couldn't model such situation using a string tag because there is no way to dynamically create a unique and safe string for an entire project.

SuperInvokeTag objects are efficient, optimized and safe.

5.4 - Kill all delayed methods and sequences

You can kill all the delayed methods and sequences of your entire project calling KillAll:

```
SuperInvoke.KillAll();
```

5.5 - Kill all except

For some reasons you might need to kill all the delayed methods and sequences except some of them. Tag those you want to remain "alive" which will not be killed.

```
SuperInvoke.Run(parameterlessMethodA, 1f, "blueTag");
SuperInvoke.Run(parameterlessMethodB, 1f, "redTag");
SuperInvoke.Run(parameterlessMethodC, 1f, "blackTag");
SuperInvoke.Run(parameterlessMethodD, 1f, "whiteTag");
...
sequenceW.Run("blueTag");
sequenceX.Run("redTag");
sequenceY.Run("blackTag");
sequenceZ.Run("whiteTag");
...
…
```

You can kill all but those methods and sequences tagged with "whiteTag" in the following manner:

```
SuperInvoke.KillAllExcept("whiteTag");
```

Or, you might need to kill all but those tagged with "redTag" and "blueTag":

```
SuperInvoke.KillAllExcept("blueTag", "redTag");
```

5.6 - Full Control on tagging and killing

You have full control on tagging with those two types of tags:

- **string tags**, which allows you to use the easiness and directness of strings
- **SuperInvokeTag tags**, which allows you to model and control more complex situations

You have full control on killing with those three methods:

- **SuperInvoke.Kill**, which allows you to directly specify what needs to be killed
- **SuperInvoke.KillAllExcept**, which allows to directly specify what needs to remain *alive*
- **SuperInvoke.KillAll**, which allows you to kill everything in one simple line

6 - !!! 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
```

7 - RunRepeat

To repeat the execution of a method 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

7.1 - RunRepeat - Example 1 - Specific number of repeats

Say you want to play a sound every second for 5 times, with no initial delay. You will write:

```
SuperInvoke.RunRepeat(0f, 1f, 5, ()=> PlayMySound());
```

7.2 - RunRepeat - Example 2 - Infinite repeats

Let's say you want to play a sound every second for an infinite number of times, with no initial delay. To repeat ad infinitum you need to write any non-positive number—either 0 (zero) or a negative number— as the *Repeats* parameter.

```
SuperInvoke.RunRepeat(0f, 1f, -1, ()=> PlayMySound());
```

7.3 - Tags in RunRepeat

Like in *SuperInvoke.Run* you can tag usages of *SuperInvoke.RunRepeat*. Please refer to section 5 - Tagging and Killing on how to use tags in SuperInvoke.

8 - Any In Run

If you want to know if there are any specific tasks, whether they are methods or sequences, that have been delayed and not yet finished to execute, you can call *SuperInvoke.AnyInRun* passing the tag of the delayed tasks you want to know more about.

8.1 - Example AnyInRun

You might have delayed some methods and sequences using the string tag "RedCode". You subsequently call:

```
SuperInvoke.AnyInRun("RedCode");
```

It will return *true* if any method or sequence you have delayed with *Run (*or *RunRepeat*) and the tag "RedCode" have yet to be called and executed.
It will return *false* otherwise, i.e. in case you haven't delayed any method or sequence with the tag "RedCode", or all the delayed methods and sequences delayed have been called and completely executed.