Safe Coroutine

by

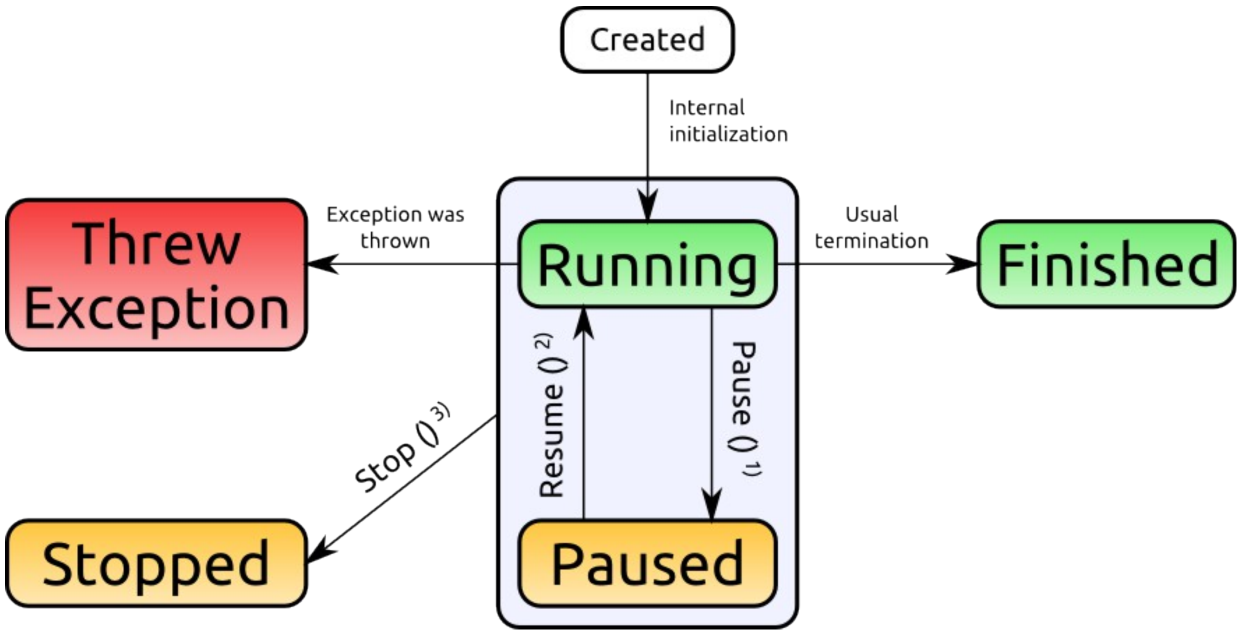
Edelweiss Interactive

1.1. Introduction

Safe Coroutine is a programming extension for Unity that overcomes several limitations of Unity's coroutines. Contrary to usual coroutines, any safe coroutine can be stopped, paused and resumed. Keeping track of those state changes can be achieved with a notification mechanism.
Exceptions are automatically caught and become accessible to allow an appropriate reaction.
Generic safe coroutine may additionally provide a result of that generic type.

Safe Coroutine States

The diagram illustrates all the possible states of safe coroutines and shows how to transition between the individual states.



1) Pause () or any enclosing safe coroutine is paused.
2) Resume () and no enclosing safe coroutine is paused, or the only paused enclosing safe coroutine was resumed.
3) Stop () or any enclosing safe coroutine is stopped.

Created

Starting a safe coroutine is almost identical to starting a Unity coroutine. The coding examples assume that the namespace *Edelweiss.Coroutine* is imported with the using directive.

```
      // Starting a Unity coroutine
StartCoroutine (SomeCoroutine ());
```

```
      // Starting a safe coroutine
this.StartSafeCoroutine (SomeCoroutine ());
```

```
     // Get a reference of the started safe coroutine
SafeCoroutine l_Coroutine = this.StartSafeCoroutine (SomeCoroutine ());

     // Get a reference to the started safe coroutine which may
     // provide a result of type float.
SafeCoroutine <float> l_CoroutineWithResult =
     this.StartSafeCoroutine <float> (SomeCoroutineWithResult ());
```

Running

After a safe coroutine is started, it is in the running state. This state isn't different from any Unity coroutine being executed.

Finished

A terminated safe coroutine that was not interrupted, ends up in the finished state.
Generic safe coroutines can return a result using *yield*.

```
private IEnumerator SomeCoroutineWithResult () {
     ...

            // Provide pi as result.
     yield return (Mathf.PI);
}
```

As the generic safe coroutine is finished, the result may be queried. The reference to generic safe coroutines can e.g. be preserved using an instance variable, or through the state change notification mechanism. Notifications are discussed in the State Change Notification section.

```
if (m_CoroutineWithResult.HasResult) {
     float l_CoroutineResult = m_CoroutineWithResult.Result;
     ...
}
```

Paused

The execution of running safe coroutines can be paused.

```
m_Coroutine.Pause ();
```

The execution of paused safe coroutines can be continued.

```
m_Coroutine.Resume ();
```

Pausing of nested safe coroutines is discussed in the Nested Safe Coroutines section.

Stopped
Every running or paused coroutine can be stopped.

```
m_Coroutine.Stop ();
```

Stopping of nested safe coroutines is discussed in the Nested Safe Coroutines section.

Threw Exception
When an exception is thrown within a safe coroutine, its state changes to *ThrewException*. Further information about the exception can be queried. Exceptions are only cought if the *CatchException* property is set to *true*.

```
if (m_Coroutine.ThrewException) {
     Exception l_CoroutineException = m_Coroutine.ThrownException;
     ...
}
```

Nested Safe Coroutines

Safe coroutines can be nested. That means a safe coroutine can be started, which we are going to call the outer safe coroutine and yield another safe coroutine, which we are going to call the inner safe coroutine.

```
private SafeCoroutine m_OuterCoroutine;
private SafeCoroutine m_InnerCoroutine;

private void Awake () {
     m_OuterCoroutine = this.StartSafeCoroutine (OuterCoroutine ());
}

private IEnumerator OuterCoroutine () {
     ...
     m_InnerCoroutine = this.StartSafeCoroutine (InnerCoroutine ());
     yield return (m_InnerCoroutine);
     ...
}

private IEnumerator InnerCoroutine () {
     ...
}
```

Paused

There are two kinds of pausing. First there is the self pausing which means that for the safe coroutine itself the pause method was called. Besides that, there is also the parent pausing which means that at least one enclosing safe coroutine is paused and as such the safe coroutine for which a parent is paused gets paused as well.
Applied to our example, that means if we pause the outer safe coroutine, this will also pause the inner safe coroutine. The outer safe coroutine is self paused while the inner safe coroutine is parent paused.

Stopped

If a safe coroutine is stopped, all the safe coroutines it encloses will be stopped as well.
In our example that means if the outer safe coroutine is stopped, this will also stop the inner one.

State Change Notifications

Keeping track of the state of safe coroutines can be achieved with a notification mechanism that is based on delegates. It can be used in many scenarios. E.g. if a safe coroutine is responsible to play an audio clip, it may need to pause the playback, as the coroutine itself is paused. That can easily be achieve with notifications.

```
private void Awake () {
     SafeCoroutine l_Coroutine = this.StartSafeCoroutine (SomeCoroutine ());
     l_Coroutine.StateChangeNotifier.Subscribe (OnCoroutineStateChange1);
     l_Coroutine.StateChangeNotifier.Subscribe (OnCoroutineStateChange2);
}

private void IEnumerator SomeCoroutine () {
     ...
}

private void OnCoroutineStateChange1 (SafeCoroutineState a_State) {
     if (a_State == SafeCoroutineState.Paused) {
            ...
     }
}

private void OnCoroutineStateChange2
     (SafeCoroutine a_SafeCoroutine, SafeCoroutineState a_State)
{
     ...
}
```

Notifications can also be used for generic safe coroutines. An obvious application is to let it perform a computation that leads to a result. Instead of checking the state in every Update (), the result can be obtained through a notification.

```
private void Awake () {
     SafeCoroutine <float> l_CoroutineWithResult =
            this.StartSafeCoroutine <float> (SomeCoroutineWithResult ());
     l_CoroutineWithResult.GenericStateChangeNotifier.Subscribe
            (OnCoroutineStateChange1);
     l_CoroutineWithResult.GenericStateChangeNotifier.Subscribe
            (OnCoroutineStateChange2);
}

private void IEnumerator SomeCoroutineWithResult () {
     ...
}

private void OnCoroutineStateChange1 (SafeCoroutineState a_State) {
     if (a_State == ...) {
            ...
     }
}

private void OnCoroutineStateChange2
     (SafeCoroutine <float> a_SafeCoroutine, SafeCoroutineState a_State)
{
     if (a_SafeCoroutine.HasResult) {
            Debug.Log (a_SafeCoroutine.Result);
     }
}
```

Know Limitations

It is explained in the Nested Safe Coroutines section how safe coroutines can be yielded inside of safe coroutines without loosing any control. That is not anymore the case if a Unity coroutine is yielded inside of a safe coroutine. There is no way to control the Unity coroutine, even stopping the enclosing safe coroutine will have no impact on the Unity coroutine.