Coroutines

> Read the code in the Examples Folder.

Coroutines

Coroutines are much easier to deal with than preemptive multitasking. The can make asynchronous code look sequential.

```
IEnumerator ShowScore() {
  while (true) {
    yield return ScoreChanged();
    yield return UpdateScore();
  }

IEnumerator ScoreChanged() {
  while (score == lastScore) {
    yield return null;
  }
  lastScore = score;
}
```

But, what to do when one coroutine is reliant on another when the calls are remote.

```
IEnumerator Start() {
  yield return InitAuth();
}

public IEnumerator Login() {
  yield return LoginAction();
  yield return CheckLoginAction();
}

public IEnumerator SendMessage() {
  yield return SendMessageAction();
}
```

Can you see the problem? If `Login()` is called before `InitAuth()` completes, it will fail. You could set a ready variable in `Start()` and wait for it whenever you call a dependent function. Do you create another variable to wait on in `SendMessage()` until `Login()` is complete? Can a server tolerate requests coming in out of order? We can recode it more cleanly with ***Coroutines***.

Coroutines.Sequential

```
Coroutines auth = null;

IEnumerator Start() {
  auth = Coroutines.Sequential(this, InitAuth());
}

public void Login() {
  auth.Queue(LoginAction(), CheckLoginAction());
}

public void SendMessage() {
  auth.Queue(SendMessageAction());
}

public IEnumerator Ready() {
  yield return auth.Completed();
}
```

Coroutines.Queue

Here, `Coroutines.Queue(params IEnumerator[])` will create a queue of actions that will always be run one after another - in the order injected.

Coroutines.Completed

Wait for all the currently queued coroutines to complete.

Tasks

As I have discussed before, Unity3D uses Coroutines to provide a form of multitasking. While it is very efficient, it is not very granular. The refresh rate is typically 60Hz. So, 60 times a second all coroutines that are ready to continue will have access to the CPU until they rerelease it.

C#/.NET also has preemptive multitasking support. Microsoft introduced an implementation of the Task-based asynchronous pattern. It is likely that Unity3D will move away from coroutines. Tasks are not limited to the 60Hz granularity. Nor are they limited to a single CPU or core. Long-running processes do not require special care. You have to understand how to make code thread-safe. With that comes the risk of the dreaded deadlock.

Google Firebase uses tasks for Unity. We need a thread-safe bridge since the rest of the current code relies on coroutines. Fortunately, most, if not all, of the asynchronous operations are not so critical that they can't fit into the 60-hertz coroutine cycle.

Tasks.WaitFor

WaitFor has a signature of:

```
public static IEnumerator WaitFor(Task task, Action<string> error =
null);
```

And here is an example of converting Task time to Coroutine time:

```
 int counter = 0;

  // Start an asynchronous task that completes after a time in
milliseconds
  Task Delay(int ms, string msg) {
    return Task.Run(async () => {
        Debug.Log(msg);
        await Task.Delay(ms);
        counter++;
      });
  }

  [UnityTest]
  public IEnumerator TasksExampleWithEnumeratorPasses() {
    Task task = Delay(500, "1. Wait for task to complete");
    yield return Tasks.WaitFor(task);
    Assert.AreEqual(counter, 1);

    task = Delay(500, "2. Wait for task to complete with error
processing");
    yield return Tasks.WaitFor(task, (msg) => Debug.Log(msg));
    Assert.AreEqual(counter, 2);

```

```
22      Debug.Log("3. All Done");
23    }
```

This simple example creates a task that runs on a different thread for a specified amount of time. By setting a counter, we can be sure the coroutine is not returning until the task has completed.

The error action is optional. If not supplied, the error goes to the debug log.

Tasks.WaitFor<T>

If the Task is to return a value, then use the generic version.

```
1   public static IEnumerator WaitFor<T>(Task<T> task, Action<T> action,
Action<string> error = null);
```

Here you must provide an action to do when the Task completes and returns a value.

The first hiccup is the DotNET version. `System.Threading.Tasks` was not implemented until DotNet 4. If you want code to compile when using DotNet 2, wrap it in `#if (!NET_2_0 && !NET_2_0_SUBSET)`.