# On writing loops in continuation-passing style, part 4

devblogs.microsoft.com/oldnewthing/20230825-00

August 25, 2023

Raymond Chen

So far, we've been look at writing loops in PPL and continuation-passing style, and a lot of the complications came from creating `shared_ptr`s to manage shared state without copying, and trying to reduce the number of such pointers we had to make. The equivalent helper functions in C# and JavaScript are simpler because in those languages, references act like `shared_ptr` already; there's no need to convert them into shared pointers explicitly.

```
class TaskHelpers
{
    public static Task DoWhileTask(Func<Task<bool>> callable)
    {
        return callable().ContinueWith(t =>
            t.Result ? DoWhileTask(callable)
                     : Task.CompletedTask).Unwrap();
    }
}
```

The C# Task Parallel Library's `ContinueWith` method is the equivalent to the PPL `then()` method: You give it a `Func<Task<T>, Result>` which is called with the preceding task. In our case, we are given a `Task<bool>`: We check the result, and if it is `true`, then we recurse back and do the whole thing again.

The gotcha is that `ContinueWith` returns a task whose result type matches the return value of the `Func` you passed in. In our case, that `Func` returns a `Task`, so the return value of `ContinueWith` is a rather confusing `Task<Task>`. You need to follow up with the `Unwrap()` method to unwrap one layer and get a `Task` back. (More generally, the `Unwrap` method converts a `Task<Task<T>>` to a `Task<T>`.)

The JavaScript version is comparable.

```
function do_while_task(callable) {
    return callable().then(loop =>
        loop ? do_while_task(callable) : undefined);
}
```

We take advantage of the JavaScript convenience that the continuation function can return either a Promise or a value, so instead of returning a settled Promise, we just return `undefined` and let that be the result of the promise chain.

We can code golf it a little more by using the `&&` operator:

```
function do_while_task(callable) {
    return callable().then(loop =>
        loop && do_while_task(callable));
}
```

In time, C++, C#, and JavaScript all gained some variation of the `await` keyword, and it's probably easier to use that keyword if you can.

```
// C++/PPL
task<void> create_many_widgets(Widget* widgets, int count)
{
    for (int i = 0; i < count; i++) {
        widgets[0] = co_await create_widget();
    }
}

// C#
async Task CreateManyWidgets(Widget[] widgets)
{
    for (int i = 0; i < widgets.Count; i++) {
        widgets[i] = await CreateWidget();
    }
}

// JavaScript
async function createManyWidgets(widgets) {
    for (var i = 0; i < widgets.length; i++) {
        widgets[i] = await createWidget();
    }
}
```