Asynchronous Methods

Bart J.F. De Smet
bartde@outlook.com

pluralsight
hardcore dev and IT training

Quick Recap of Asynchronous Methods

- **Methods that can be paused due to asynchronous activity**
  - Decorated with “async” modifier
  - Return Task, Task<T>, or void
  - Can have “await” expressions inside

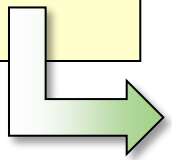
```
static async Task<string[]> GetLinksAsync(Uri uri)
{
  var clnt = new WebClient();
  var html = await clnt.DownloadStringTaskAsync(uri);
  var urls = GetLinks(html);
  return urls;
}
```

- **Eliminates need for callback-based programming**
  - Asynchronous Begin*/End* methods with IAsyncResult
  - Event-based asynchronous programming with *Completed event
  - ContinueWith continuations on Task

Asynchronous Method Builders

- **Control flow of asynchronous methods**
  - Using “return” causes Task to go to RanToCompletion state
  - Exceptions cause Task to go to Faulted or Canceled state
  - “void”-returning asynchronous methods are fire-and-forget

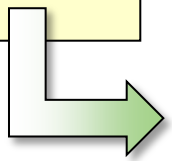
```
async Task<int> GetAsync() {
  return 42;
}
```



```
Task<int> GetAsync() {
  var tcs = new TaskCompletionSource<int>();
  tcs.SetResult(42);
  return tcs.Task;
}
```

Hidden by builder

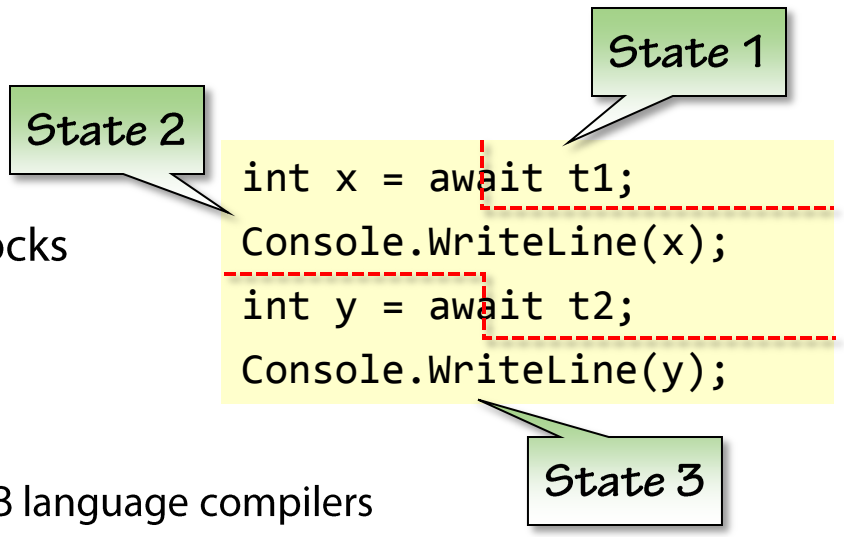
```
async Task<int> FailAsync() {
  throw new FooException();
}
```



```
Task<int> FailAsync() {
  var tcs = new TaskCompletionSource<int>();
  tcs.SetException(new FooException());
  return tcs.Task;
}
```

Asynchronous Method Builders

- **Supporting "await" expressions**
  - Remainder of the method beyond "await" is continuation
  - Awaiter pattern used to
    - Check for completion (optimizes synchronous case)
    - Hook up continuation
    - Retrieve the result
  - Custom awaiters can be built
- **Asynchronous method builders**
  - Breaking up the method in basic blocks
    - Build state machine
    - Similar to "yield" statements
  - Leverage an IAsyncStateMachine
    - Implementation emitted by C# or VB language compilers
  - System.Runtime.CompilerServices namespace



Revisiting the Awaiter Pattern

- Await expression = syntactic sugar

```
Task<int> t = …;
int x = await t;
```

```
void MoveNext() { // Async state machine
  switch (state) {
    …
    case 17: Task<int> t = …;
             awaiter_n = t.GetAwaiter();
             state = 18;
             if (awaiter_n.IsCompleted)
               goto case 18;
             else
               awaiter_n.OnCompleted(MoveNext);
             break;
    case 18: int x = awaiter_n.GetResult();
             …
    …
  }
}
```

Becomes a field

Optimization

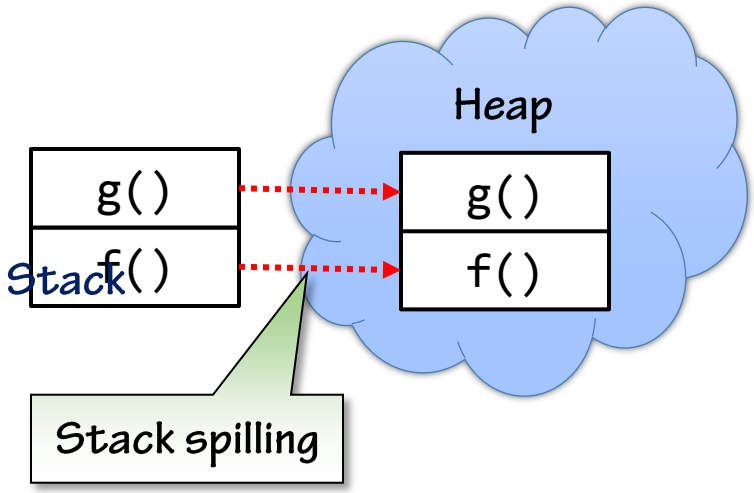
Continuation

Stack Spilling

- **"Await" is an expression**
  - Expression evaluation happens left-to-right
  - Evaluation stack may not be empty
    - Need to stash away evaluation stack to heap
    - Can't trigger re-evaluation (side-effects)

```
int x = f() + g() * await t;
```

May suspend



- **Stack spilling**
  - Only on asynchronous path of "await"
    - Avoids heap allocations
  - Single field for all spills needed in async method
    - Spill sites run sequentially; only one needed at a time
    - Evaluation stack encoded as Tuple<T*> (if > 1 slot needed)

Summary

- **Asynchronous method builders**
  - Types in System.Runtime.CompilerServices used by compiler
  - Parameterized by IAsyncStateMachine
    - Splitting of method in blocks a la iterators
    - One state per “await”
  - Control flow of asynchronous method
    - Successful termination (return)
    - Exceptional termination

- **Awaiter pattern**
  - Optimization of synchronous case
  - Can be implemented by any type
  - Stack spilling to store evaluation stack at point of “await”