Asynchronous Functions in C#

Asynchronous operations are methods and other function members that may have most of their execution take place *after* they return. In .NET the recommended pattern for asynchronous operations is for them to return a ***task*** which represents the ongoing operation and allows waiting for its eventual outcome.

Asynchronous operations are useful when multiple flows of control need to share the threads they run on. For instance they can be used to share a single user interface thread between multiple ongoing operations, or to service thousands of simultaneous ongoing requests on a server with a much smaller pool of threads.

Traditionally, writing and composing asynchronous operations is difficult and error prone. It involves installing callbacks – sometimes called ***continuations*** – on other asynchronous operations to express all the logic that needs to happen after those operations completes. This alters, and usually significantly complicates, the structure of asynchronous code as compared to a corresponding synchronous program.

Asynchronous functions is a new feature in C# which provides an easy means for expressing asynchronous operations. Inside asynchronous functions ***await expressions*** can `await` ongoing tasks, which causes the rest of the execution of the asynchronous function to be transparently signed up as a continuation of the awaited task. In other words, it becomes the job of the programming language, not the programmer, to express and sign up continuations. As a result, asynchronous code can retain its logical structure.

An asynchronous function in C# must either return `void` or one of the types `Task` or `Task<T>`. C# does not dictate a specific type for the representation of tasks that are `await`'ed, as long as they satisfy a certain pattern. The types `Task` and `Task<T>` satisfy this pattern, and will be used in subsequent examples. The reliance on the pattern allows other representations of asynchronous operations to be awaited, for interoperability purposes and to allow APIs to produce tasks with special semantics different from those of `Task` and `Task<T>`.

The following example demonstrates a few simple asynchronous operations:

```
Task<Movie> GetMovieAsync(string title);
Task PlayMovieAsync(Movie movie);
async void GetAndPlayMoviesAsync(string[] titles)
{
    foreach (var title in titles)
    {
        var movie = await GetMovieAsync(title);
        await PlayMovieAsync(movie);
    }
}
```

By convention asynchronous operations use the postfix “Async” to show that part of their execution may take place after the method call has returned. Both `GetMovieAsync` and `PlayMovieAsync` return tasks, which means that their completion can be subsequently `await`’ed. By contrast, `GetAndPlayMoviesAsync` returns `void`, so its completion cannot be `await`’ed. Such asynchronous operations are often referred to as “fire and forget”, and are useful e.g. for implementing event handlers asynchronously.

`GetAndPlayMoviesAsync` is marked by the `async` modifier as an asynchronous function, containing two `await` expressions. This fact is an implementation detail to its callers, but fundamentally changes the way the method is executed: As soon as an unfinished `Task` is awaited, it will return to its caller. When the awaited `Task` completes, it will resume execution of the `GetAndPlayMoviesAsync` method until the next unfinished `Task` is awaited, and so on. In between, while awaiting unfinished `Tasks`, no thread is occupied with the execution of this method: it “borrows time” on threads only when it is active.

Asynchronous functions

An ***asynchronous function*** is a method or anonymous function which is marked with the `async` modifier. A function without the `async` modifier is called ***synchronous***.

Asynchronous functions are evaluated differently from other functions in that their evaluation is *discontinuous*: Their evaluation may be suspended at any *await-expression* until the awaited task determines that it is time to resume their execution.

Syntax

The grammar of C# is extended as follows:

method-modifier:
    …
    `async`$_{opt}$

lambda-expression:
    `async`$_{opt}$ *anonymous-function-signature* `=>` *anonymous-function-body*

anonymous-method-expression:
    `async`$_{opt}$ *`delegate`* *explicit-anonymous-function-signature*$_{opt}$ *block*

The `async` modifier is not allowed on *method-declaration*s where the *method-body* is a ‘`;`’.

Note that `async` is a contextual keyword. In all syntactic contexts other than the ones above it is considered an identifier. Thus, the following is allowed (though strongly discouraged!):

```
using async = System.Threading.Tasks.Task;
…
async async async(async async) { }
```

Return types of asynchronous functions

The return type of an asynchronous function must be either `void`, `Task` or `Task<T>` for some `T`. The return type of an anonymous function is the return type – if any – of the delegate or expression type that it is being converted to.

In an asynchronous function with the return type `void` or `Task`, `return` statements must not have an expression.

In an asynchronous function with the return type `Task<T>` for some `T`, `return` statements must have an expression that is implicitly convertible to `T`, and the endpoint of the body must be unreachable.

Evaluation of Task-returning asynchronous functions

Invocation of a `Task`-returning asynchronous function initially is no different from a synchronous function. However, when reaching an `await` expression, the asynchronous function *may* return to its caller, even though its execution is not yet finished.

A `Task` or `Task<T>` will be returned to the caller. It is guaranteed not to be `null`. If execution of the asynchronous method completes successfully, the returned task will be marked as succeeded, and any value produced through a return statement with an expression will become the result of the returned `Task<T>`. If an exception occurs and is not caught within the asynchronous method, the task is marked as faulted or cancelled and the exception stored within it. The task is marked cancelled only in the case of certain implementation dependent exceptions designating cancellation.

When an asynchronous function is invoked it can be seen as being in one of three states: ***running***, ***suspended*** at an `await` expression, or ***completed***.

Evaluation of void-returning asynchronous functions

If the return type of the asynchronous function is `void`, evaluation differs from the above in various ways. Because no Task is returned, the function instead communicates completion and exceptions to the current thread's ***context***. The exact definition of context is implementation defined, but is a representation of "where" the current thread is running. The context is notified when evaluation of a void returning asynchronous function

- commences,
- completes successfully, or
- causes an unhandled exception to be thrown.

This allows the context to keep track of how many void-returning asynchronous functions are running under it, and to decide how to propagate exceptions coming out of them.

Await expressions

Await expressions are used to suspend the execution of an asynchronous function until the awaited task completes.

Syntax

The grammar of C# is extended as follows:

unary-expression:
    …
    await-expression

await-expression:
    `await` *unary-expression*

statement-expression:
    …
    await-expression

An `await` expression is only allowed when occurring in the body of an ***asynchronous function***. Inside of the innermost enclosing asynchronous function it may furthermore not occur inside of the body of a synchronous function, in a `catch` or `finally` block of a *try-statement*, inside the block of a *lock-statement*, or in an `unsafe` context.

Note that this means that an `await` expression cannot occur in most places within a *query-expression*, because those get rewritten to use synchronous lambda expressions.

Since an `await` expression can only occur in an asynchronous function, it can never be confused with uses of `await` as an identifier already occurring in code written in older versions of C#.

The await pattern

The expression `t` of an *await-expression* `await t` is called the *task* of the `await` expression. The task `t` is required to be ***awaitable***, which means that one of the following must hold:

- The task `t` is of type `dynamic`, or
- An accessible instance or extension method `GetAwaiter` with no parameters and return type `A` is available on t, where `A` has the following accessible instance members:
    - `bool IsCompleted { get; }` – with a setter being also permitted.
    - `void OnCompleted(Action);`
    - One of:
        - `void GetResult();`
        - `T GetResult();` – where `T` is any type.

`A` is called the *awaiter* type for the await expression. The `GetAwaiter` method is used to obtain an awaiter for the task.

The `IsCompleted` property is used to determine if the task is already complete. If so, there is no need to sign up a continuation and suspend evaluation.

The `OnCompleted` method is used to sign up a continuation on the awaited task.

The `GetResult` method is used to obtain the outcome of the task once it is complete.

All of `GetAwaiter`, `IsCompleted`, `OnCompleted` and `GetResult` are intended to be "non-blocking"; that is, not cause the calling thread to wait for a significant amount of time, e.g. for an operation to complete.

Classification

If an *await-expression* `await t` is valid, it is classified the same way as the expression `(a).GetResult()`. That is:

- If the expression `(a).GetResult()` invokes a method that returns `void`, the result is nothing.
- Otherwise, the result is a value of the type returned by `(a).GetResult()`.

Evaluation of await expressions

During the execution of the body of the asynchronous function, `await t` is evaluated as follows:

- An awaiter `a` is obtained by evaluating the expression `(t).GetAwaiter()`.
- The expression `(a).IsCompleted` is evaluated.
- If the result is `false` then
    - The expression `(a).OnCompleted(r)` is evaluated, where `r` is a delegate whose invocation will cause execution of the enclosing asynchronous function to be resumed after the current `await` expression.
    - Evaluation is suspended, and control is returned to the function that invoked or resumed the asynchronous function.
- Then (either immediately following or upon resumption) the expression `(a).GetResult()` is evaluated. If it returns a value, that value is the result of the *await-expression*. Otherwise the result is nothing.

If the type of `t` is `dynamic`, this may lead to binding errors at runtime, if corresponding members cannot be found. But beyond that, the requirements that non-dynamic awaitables are subject to at compile-time, are not enforced at runtime on dynamic awaitables.

The `IsCompleted` property allows the `await`'ing of already-completed tasks to avoid suspending execution only to have it immediately resumed.

The implementation of `OnCompleted` should make sure that the delegate `r` is invoked at most once. If `r` is invoked beyond these restrictions, its behavior is undefined.

Conversions

In order to account for asynchronous anonymous functions, the rules for conversion of anonymous functions at the beginning of section 6.5 are modified as follows:

An *anonymous-method-expression* or *lambda-expression* is classified as an anonymous function (§**Error! Reference source not found.**). The expression does not have a type but can be implicitly converted to a

compatible delegate type or expression tree type. Specifically, a delegate type `D` is compatible with an anonymous function `F` provided:

- If `F` contains an *anonymous-function-signature*, then `D` and `F` have the same number of parameters.
- If `F` does not contain an *anonymous-function-signature*, then
    - `D` may have zero or more parameters of any type.
    - No parameter of `D` may have the `out` parameter modifier.
    - If `F` is asynchronous, no parameter of `D` may have the `ref` parameter modifier.
- If `F` has an explicitly typed parameter list, each parameter in `D` has the same type and modifiers as the corresponding parameter in `F`.
- If `F` has an implicitly typed parameter list, `D` has no `ref` or `out` parameters.
- If the body of `F` is an expression and *either* `D` has a `void` return type, *or* `F` is asynchronous and the return type of `D` is `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt §7) that would be permitted as a *statement-expression* (§8.6).
- If the body of `F` is a statement block and *either* `D` has a `void` return type, *or* `F` is asynchronous and the return type of `D` is `Task`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt §8.2) in which no `return` statement specifies an expression.
- If the body of `F` is an expression, and *either* `F` is synchronous and `D` has a non-void return type `T`, *or* `F` is asynchronous and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid expression (wrt §7) that is implicitly convertible to `T`.
- If the body of `F` is a statement block, and *either* `F` is synchronous and `D` has a non-void return type `T`, *or* `F` is asynchronous and `D` has a return type `Task<T>`, then when each parameter of `F` is given the type of the corresponding parameter in `D`, the body of `F` is a valid statement block (wrt §8.2) with a non-reachable end point in which each `return` statement specifies an expression that is implicitly convertible to `T`.

An expression tree type `Expression<D>` is compatible with an anonymous function `F` if the delegate type `D` is compatible with `F`.

Certain anonymous functions cannot be converted to expression tree types: Even though the conversion *exists*, it fails at compile-time. This is the case if the anonymous expression:

- contains simple or compound assignment operators
- contains dynamically bound expressions
- is asynchronous

Type inference and overload resolution

Where anonymous functions are treated specially in type inference and overload resolution contexts, special care is also given to anonymous asynchronous functions.

Inferred return type

To express the notion of inferred return types for asynchronous anonymous functions, a helper notion of inferred *result* type is introduced. It is used only in the definition of inferred return type.

The inferred return type of an anonymous function `F` is used during type inference and overload resolution. The inferred return type can only be determined for an anonymous function where all parameter types are known, either because they are explicitly given, provided through an anonymous function conversion or inferred during type inference on an enclosing generic method invocation.

The inferred *result* type of `F` is determined as follows:

- If the body of `F` is an expression that has a type, then the inferred result type of F is the type of that expression.
- If the body of F is a block and the set of expressions in the block's return statements has a best common type T (§7.5.2.14), then the inferred result type of `F` is `T`.
- Otherwise, a result type cannot be inferred for `E`.

The inferred *return* type of `F` is determined as follows:

- If F is asynchronous and the body of F is either an expression classified as nothing, or a statement block where no return statements have expressions, the inferred return type is `Task`
- If `F` has an inferred result type `T`:
  - If F is synchronous the inferred return type is `T`.
  - If F is asynchronous the inferred return type is `Task<T>`.
- Otherwise a return type cannot be inferred for `F`

Better conversion from expression

During overload resolution, the inferred return type is used to determine which conversion is better from an anonymous function to a delegate or expression type. This mechanism is extended to better select between overloads when called with asynchronous anonymous functions, as follows:

Given an implicit conversion C_1 that converts from an expression E to a type T_1, and an implicit conversion C_2 that converts from an expression E to a type T_2, C_1 is a better conversion than C_2 if at least one of the following holds:

- `E` has a type `S` and an identity conversion exists from `S` to T_1 but not from `S` to T_2
- `E` is not an anonymous function and T_1 is a better conversion target than T_2 (§7.5.3.5)
- `E` is an anonymous function, T_1 is either a delegate type D_1 or an expression tree type `Expression<`D_1`>`, T_2 is either a delegate type D_2 or an expression tree type `Expression<`D_2`>` and one of the following holds:

- D_1 is a better conversion target than D_2
- D_1 and D_2 have identical parameter lists, and one of the following holds:
  - D_1 has a return type Y_1, and D_2 has a return type Y_2, an inferred return type X exists for E in the context of that parameter list (§7.5.2.12), and the conversion from X to Y_1 is better than the conversion from X to Y_2
  - E is an asynchronous function, D_1 has a return type Task<Y_1>, and D_2 has a return type Task<Y_2>, an inferred return type Task<X> exists for E in the context of that parameter list (§7.5.2.12), and the conversion from X to Y_1 is better than the conversion from X to Y_2
  - D_1 has a return type Y, and D_2 is void returning

Implementation example

The following shows an approach to implementing asynchronous functions via syntactic expansion.

Asynchronous functions returning Task<T>

User code	Syntactic expansion	
async Task<string> Fred(int p) **{**	Task<string> Fred(int p) {	*exactly the same signature*
	var $builder = AsyncTaskMethodBuilder<string>.Create(); var $state = 0;	*get a builder for Task<T>*
	TaskAwaiter<…> $a1; TaskAwaiter $a2;	*one declaration for each "await" point*
	Action $resume = null; $resume = delegate {	
	try { ... jump table based on $state	*if the user had nested try blocks, then we use nested jump tables; and there's additional logic so that the bodies of "finally" blocks are skipped where appropriate*
var x = await e;	$a1 = (e).GetAwaiter(); $state=1; if ($a1.IsCompleted) { $a1.OnCompleted($resume)) return; } JUMP_LABEL_1: var x = $a1.GetResult();	*these calls are just syntactic expansions – they follow the usual rules for overload resolution &c. It's a compile-error to await something if the calls can't be resolved.*
await s;	$a2 = (s).GetAwaiter(); $state=2; if ($a2.IsCompleted) { $a2.OnCompleted($resume))	*In the statement form of "await", the GetResult method is allowed to return*

	return;	*void.*
	}	
	JUMP_LABEL_2:	
	$a2.GetResult();	
throw new Exception();	throw new Exception();	
return r;	$builder.SetResult(r);	
	return;	
	}	
	catch (Exception $ex)	
	{	
	$builder.SetException($ex);	
	return;	
	}	
	};	
	$resume();	
	return $builder.Task;	
}	}	

Asynchronous functions returning Task

User code	Syntactic expansion	
async Task Fred(int p)	Task Fred(int p)	
{	{	
	var $builder = AsyncTaskMethodBuilder.Create();	*$builder.SetResult() is now parameterless*
	var $state = 0;	
	TaskAwaiter $a1;	
	Action $resume = null; $resume = delegate	
	{	
	try	
	{	
	... jump table based on $state	
await s;	$a1 = (s).GetAwaiter;	
	$state=1;	
	if ($a1.IsCompleted)	
	{	
	$a1.OnCompleted($resume))	
	return;	
	}	
	JUMP_LABEL_1:	
	$a1.EndAwait();	
return;	$builder.GetResult();	
	return;	
	}	
	catch (Exception $ex)	
	{	
	$builder.SetException($ex);	
	return;	
	}	

	$builder.SetResult();	*Automatically inserted at the end of the delegate to allow omission of explicit return statement*
	};	
	$resume(); return $builder.Task;	
}	}	

Asynchronous functions returning void

User code	Syntactic expansion	
async void Fred(int p) {	Task Fred(int p) {	
	var $builder = AsyncVoidMethodBuilder.Create(); var $state = 0;	*different builder for void async methods*
	TaskAwaiter $a1;	
	Action $resume = null; $resume = delegate {	
	try { ... jump table based on $state	
await s;	$a1 = (s).GetAwaiter; $state=1; if ($a1.IsCompleted) { $a1.OnCompleted($resume)) return; } JUMP_LABEL_1: $a1.GetResult();	
return;	$builder.SetResult(); return;	
	} catch (Exception $ex) { $builder.SetException($ex); return; }	
	$builder.SetResult();	
	};	
	$resume();	*no Task returned*
}	}	