How To Learn A New Programming Language

Derek, Zhao  
March 2019

Contents

[Overview 2](#_Toc4939742)

[Basic Grammar 2](#_Toc4939743)

[Keywords 2](#_Toc4939744)

[Data Types 3](#_Toc4939745)

[Constructs 3](#_Toc4939746)

[Flow Control 3](#_Toc4939747)

[Choosing Which Overloads to Provide 5](#_Toc4939748)

[IDE 6](#_Toc4939749)

[Short Keys 6](#_Toc4939750)

[Project Management 6](#_Toc4939751)

[Debug Tips 7](#_Toc4939752)

[Platform Library 7](#_Toc4939753)

[I/O 7](#_Toc4939754)

[Advanced Data Structure 7](#_Toc4939755)

[Multiple Thread 7](#_Toc4939756)

[Metadata programming 7](#_Toc4939757)

[Frameworks 7](#_Toc4939758)

[ORM 8](#_Toc4939759)

[IOC 8](#_Toc4939760)

[Log 8](#_Toc4939761)

[Unit Test 8](#_Toc4939762)

# Overview

The Task-based Asynchronous Pattern (TAP) is a new pattern for asynchrony in the .NET Framework. It is based on the Task and Task<TResult> types in the System.Threading.Tasks namespace, which are used to represent arbitrary asynchronous operations.

# Basic Grammar

## Keywords

Initiation and completion of an asynchronous operation in the TAP are represented by a single method, and thus there is only one method to name. This is in contrast to the IAsyncResult pattern, or APM pattern, where Begin*MethodName* and End*MethodName* methods are required, and in contrast to the event-based asynchronous pattern, or EAP, where a *MethodName*Async is required in addition to one or more events, event handler delegate types, and EventArg-derived types. Asynchronous methods in the TAP are named with an “Async” suffix that follows the operation’s name, e.g. *MethodName*Async. The singular TAP method returns either a Task or a Task<TResult>, based on whether the corresponding synchronous method would return void or a type TResult, respectively. (If adding a TAP method to a class that already contains a method *MethodName*Async, the suffix “TaskAsync” may be used instead, resulting in “*MethodName*TaskAsync”.)

For example, consider a “Read” method that reads a specified amount of data into a provided buffer starting at a specified offset:

public class MyClass

{

public int Read(byte [] buffer, int offset, int count);

}

The APM counterpart to this method would expose the following two methods:

public class MyClass

{

public IAsyncResult BeginRead(

byte [] buffer, int offset, int count,

AsyncCallback callback, object state);

public int EndRead(IAsyncResult asyncResult);

}

The EAP counterpart would expose the following set of types and members:

public class MyClass

{

public void ReadAsync(byte [] buffer, int offset, int count);

public event ReadCompletedEventHandler ReadCompleted;

}

public delegate void ReadCompletedEventHandler(

object sender, ReadCompletedEventArgs eventArgs);

public class ReadCompletedEventArgs : AsyncCompletedEventArgs

{

public int Result { get; }

}

The TAP counterpart would expose the following single method:

public class MyClass

{

public Task<int> ReadAsync(byte [] buffer, int offset, int count);

}

The parameters to a basic TAP method should be the same parameters provided to the synchronous counterpart, in the same order. However, “out” and “ref” parameters are exempted from this rule and should be avoided entirely. Any data that would have been returned through an out or ref parameter should instead be returned as part of the returned Task<TResult>’s Result, utilizing a tuple or a custom data structure in order to accommodate multiple values.

Methods devoted purely to the creation, manipulation, or combination of tasks (where the asynchronous intent of the method is clear in the method name or in the name of the type on which the method lives) need not follow the aforementioned naming pattern; such methods are often referred to as “combinators.” Examples of such methods include Task.WhenAll and Task.WhenAny, and are discussed in more depth later in this document.

## Data Types

### Number

Int, Float, double, Complex.

### Characters

Char, String

### Boolean

True, False

### Special Types

NULL, Customized types such as enum, class, struct.

### Type Cast

## Constructs

Class, Method, Enum, Struct,Tuple, Dictionary, Set, Variables etc

## Flow Control

Some asynchronous operations benefit from providing progress notifications; these are typically utilized to update a user interface with information about the progress of the asynchronous operation.

In the TAP, progress is handled through an IProgress<T> interface (described later in this document) passed into the asynchronous method as a parameter named “progress”. Providing the progress interface at the time of the asynchronous method’s invocation helps to eliminate race conditions that result from incorrect usage where event handlers incorrectly registered after the invocation of the operation may miss updates. More importantly, it enables varying implementations of progress to be utilized, as determined by the consumer. The consumer may, for example, only care about the latest progress update, or may want to buffer them all, or may simply want to invoke an action for each update, or may want to control whether the invocation is marshaled to a particular thread; all of this may be achieved by utilizing a different implementation of the interface, each of which may be customized to the particular consumer’s need. As with cancellation, TAP implementations should only provide an IProgress<T> parameter if the API supports progress notifications.

For example, if our aforementioned ReadAsync method was able to report intermediate progress in the form of the number of bytes read thus far, the progress callback could be an IProgress<int>:

public Task<int> ReadAsync(

byte [] buffer, int offset, int count,

IProgress<int> progress);

If a FindFilesAsync method returned a list of all files that met a particular search pattern, the progress callback could provide an estimation as to the percentage of work completed as well as the current set of partial results. It could do this either with a tuple, e.g.:

public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(

string pattern,

IProgress<Tuple<double,ReadOnlyCollection<List<FileInfo>>>> progress);

or with a data type specific to the API, e.g.:

public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(

string pattern,

IProgress<FindFilesProgressInfo> progress);

In the latter case, the special data type should be suffixed with “ProgressInfo”.

If TAP implementations provide overloads that accept a progress parameter, they must allow the argument to be *null*, in which case no progress will be reported. TAP implementations should synchronously report the progress to the IProgress<T> object, making it cheap for the async implementation to quickly provide progress, and allowing the consumer of the progress to determine how and where best to handle the information (e.g. the progress instance itself could choose to marshal callbacks and raise events on a captured synchronization context).

### Sequence

, execute commands one by one from up to down.

### Selection

If, switch statement

### Loops

For, foreach, while, do while

## Choosing Which Overloads to Provide

With both the optional CancellationToken and optional IProgress<T> parameters, an implementation of the TAP could potentially demand up to four overloads:

public Task MethodNameAsync(…);

public Task MethodNameAsync(…, CancellationToken cancellationToken);

public Task MethodNameAsync(…, IProgress<T> progress);

public Task MethodNameAsync(…,

CancellationToken cancellationToken, IProgress<T> progress);

However, many TAP implementations will have need for only the shortest overload, as they will not provide either cancellation or progress capabilities:

public Task MethodNameAsync(…);

If an implementation supports either cancellation or progress but not both, a TAP implementation may provide two overloads:

public Task MethodNameAsync(…);

public Task MethodNameAsync(…, CancellationToken cancellationToken);

// … or …

public Task MethodNameAsync(…);

public Task MethodNameAsync(…, IProgress<T> progress);

If an implementation supports both cancellation and progress, it may expose all four potential overloads. However, it is valid to provide only two:

public Task MethodNameAsync(…);

public Task MethodNameAsync(…,

CancellationToken cancellationToken, IProgress<T> progress);

To make up for the missing two intermediate combinations, developers may pass CancellationToken.None (or default(CancellationToken)) for the cancellationToken parameter and/or *null* for the progress parameter.

If it is expected that every usage of the TAP method should utilize cancellation and/or progress, the overloads that don’t accept the relevant parameter may be omitted.

If multiple overloads of a TAP method are exposed to make cancellation and/or progress optional, the overloads that don’t support cancellation and/or progress should behave as if they’d passed CancellationToken.None for cancellation and null for progress to the overload that does support these.

# IDE

## Short Keys

### Compiler

In the .NET Framework 4.5, the C# and Visual Basic compilers are capable of implementing the TAP. Any method attributed with the *async* keyword (*Async* in Visual Basic) is considered to be an asynchronous method, and the compiler will perform the necessary transformations to implement the method asynchronously using the TAP. Such a method should return either a Task or a Task<TResult>. In the case of the latter, the body of the function should return a TResult, and the compiler will ensure that this result is made available through the resulting Task<TResult>. Similarly, any exceptions that go unhandled within the body of the method will be marshaled to the output task, causing the resulting Task to end in the Faulted state; the one exception to this is if an OperationCanceledException (or derived type) goes unhandled, such that the resulting Task will end in the Canceled state.

### Manual

Developers may implement the TAP manually, just as the compiler does or with greater control over exactly how the method is implemented. The compiler relies on public surface area exposed from the System.Threading.Tasks namespace (and supporting types in the System.Runtime.CompilerServices namespace built on top of System.Threading.Tasks), functionality also available to developers directly. For more information, see the following section on Workloads. When implementing a TAP method manually, a developer must be sure to complete the resulting Task when the represented asynchronous operation completes.

### Hybrid

It is often useful to manually implement the TAP pattern with the core logic for the implementation implemented in a compiler-generated implementation. This is the case, for example, when arguments should be verified outside of a compiler-generated asynchronous method in order for the exceptions to escape to the method’s direct caller rather than being exposed through the Task, e.g.

public Task<int> MethodAsync(string input)

{

if (input == null) throw new ArgumentNullException("input");

return MethodAsyncInternal(input);

}

private async Task<int> MethodAsyncInternal(string input)

{

… // code that uses await

}

Another case where such delegation is useful is when a “fast path” optimization can be implemented that returns a cached task.

## Project Management

parallelism.

### Compute-Bound

.

## Debug Tips

Tasks

# Platform Library

## I/O

## Advanced Data Structure

.

## Multiple Thread

Some asynchronous methods expose progress through a progress interface passed into the asynchronous method. For example, consider a function which asynchronously downloads a string of text, and along the way raises progress updates that include the percentage of the download that has completed thus far. Such a method could be consumed in a Windows Presentation Foundation application as follows:

private async void btnDownload\_Click(object sender, RoutedEventArgs e)

{

btnDownload.IsEnabled = false;

try

{

txtResult.Text = await DownloadStringAsync(txtUrl.Text,

new Progress<int>(p => pbDownloadProgress.Value = p));

}

finally { btnDownload.IsEnabled = true; }

}

## Metadata programming

.

### Reflector

# Frameworks

.

## ORM

## IOC

## Log

## Unit Test