Поддержка асинхронного программирования в будущем C# основана на трех нововведениях: типов Task и Task<T>, оператора await и маркера async.

**Task и Task<T>**

Эти типы описывают асинхронное вычисление в процессе выполнения, а так же его результат. Можно провести аналогию с Thread. Ниже приведена часть сигнатуры Task<T>:

public class Task<TResult> : Task

{

public TResult Result { get; internal set; }

public Task<TNewResult> ContinueWith<TNewResult>(

Func<Task<TResult>, TNewResult> continuationFunction

);

}

Представим, что мы вызвали метод, который вернул нам Task<string>. Скорее всего, это означает, что вызываемый метод запустил асинхронную операцию, результатом который будет string, и вернул нам управление, а так же объект, описывающий саму асинхронную операцию, не дожидаясь завершения её выполнения. Имея этот объект, мы может обратиться к полю Result, что бы получить результат выполнения операции, в этом случае текущий поток приостановиться, пока асинхронная операция не будет завершена.  
  
Другая действие, которое мы можем совершить с объектом типа Task<T>, это указать ему, что как только асинхронная операция завершиться, ему нужно запустить continuationFunction с результатом выполнения операции. Таким образом, объединяя несколько асинхронных операций, мы получаем асинхронную операцию.  
  
В принципе пользоваться таким подходом к организации асинхронных вычислений можно уже сейчас, так как описанные типы принадлежат пространству имен System.Threading.Tasks, которое было введено в .NET Framework 4. Но такое использование не очень удобно, так как один логический метод: получить результат и обработать его, мы должны разбить на метода: один — запустить получение результата, и второй — заняться обработка результата. Поэтому были введены маркер async и оператор await.

В основу TPL положен класс Task. Элементарная единица исполнения инкапсулируется в TPL средствами класса Task, а не Thread. Класс Task отличается от класса Thread тем, что он является абстракцией, представляющей асинхронную операцию. А в классе Thread инкапсулируется поток исполнения. Разумеется, на системном уровне поток по-прежнему остается элементарной единицей исполнения, которую можно планировать средствами операционной системы. Но соответствие экземпляра объекта класса Task и потока исполнения не обязательно оказывается взаимно-однозначным.

Кроме того, исполнением задач управляет планировщик задач, который работает с пулом потоков. Это, например, означает, что несколько задач могут разделять один и тот же поток. Класс Task (и вся остальная библиотека TPL) определены в пространстве именSystem.Threading.Tasks.

**Создание задачи**

Создать новую задачу в виде объекта класса Task и начать ее исполнение можно самыми разными способами. Для начала создадим объект типа Task с помощью конструктора и запустим его, вызвав метод Start(). Для этой цели в классе Task определено несколько конструкторов. Ниже приведен тот конструктор, которым мы собираемся воспользоваться:

*public Task(Action действие)*

где действие обозначает точку входа в код, представляющий задачу, тогда как Action — делегат, определенный в пространстве имен System. Форма делегата Action, которой мы собираемся воспользоваться, выглядит следующим образом:

*public delegate void Action()*

Таким образом, точкой входа должен служить метод, не принимающий никаких параметров и не возвращающий никаких значений. (Как будет показано далее, делегату Action можно также передать аргумент.)

Как только задача будет создана, ее можно запустить на исполнение, вызвав метод Start(). После вызова метода Start() планировщик задач запланирует исполнение задачи.

Следует иметь в виду, что по умолчанию задача исполняется в фоновом потоке. Следовательно, при завершении создающего потока завершается и сама задача. Именно поэтому в рассматриваемой здесь программе метод Thread.Sleep() использован для сохранения активным основного потока до тех пор, пока не завершится выполнение метода MyTask(). Как и следовало ожидать, организовать ожидание завершения задачи можно и более совершенными способами.

В отношении задач необходимо также иметь в виду следующее: после того, как задача завершена, она не может быть перезапущена. Следовательно, иного способа повторного запуска задачи на исполнение, кроме создания ее снова, не существует.

**Async и await**

Маркер очень похож на атрибут или модификатор доступа, которые применяются к методу. Вот пример использования:

public async Task<string> DownloadStringTaskSlowNetworkAsync(Uri address) {

Маркер может применяться к методу, который возвращает Task<T> или void. Применять маркер к методу нужно, когда в теле метода происходит комбинация других асинхронных вызовов (используется оператор await) или когда метод определяет асинхронную операцию, но это редко требуется, так как в поставляемой с расширением библиотеке AsyncCtpLibrary.dll уже определено большое количество методов для работы с основными асинхронными запросами.  
  
Последним ключевым объектом, на котором основаны облегченные асинхронные операции — оператор await. Он нужен для объединения последовательности асинхронных операций в одну. Этот оператор принимает на вход объект, описывающий асинхронную операцию, и так переписывает код, что бы все, что следует после выражения с await, преобразовывалось в замыкание и было аргументом метода ContinueWith объекта к которому применяется await. Можно сказать, что это указание компилятору: «Так, все, что идет после, должно выполняться как только текущая асинхронная операция завершится». Отличие от обращения к свойству Result состоит в том, что текущий поток выполнения не приостанавливается, а создается объект описывающий асинхронную операцию и он тут же возвращается.  
  
Я не написал о средствах синхронизации асинхронных операций (Task.WaitOne, Task.WaitAll), а так же о многопоточных приложениях на основе потока данных (System.Threading.Tasks.Dataflow) и информации о ходе выполнения операции.