**Managed threading basics**

* 8 minutes to read

The first five articles of this section are designed to help you determine when to use managed threading and to explain some basic features. For information on classes that provide additional features, see [Threading Objects and Features](https://docs.microsoft.com/en-us/dotnet/standard/threading/threading-objects-and-features) and [Overview of Synchronization Primitives](https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives).

The remaining articles in this section cover advanced topics, including the interaction of managed threading with the Windows operating system.

**Note**

Starting with .NET Framework 4, the Task Parallel Library and PLINQ provide APIs for task and data parallelism in multi-threaded programs. For more information, see [**Parallel Programming**](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/).

**In this section**

[Threads and Threading](https://docs.microsoft.com/en-us/dotnet/standard/threading/threads-and-threading)  
Discusses the advantages and drawbacks of multiple threads, and outlines the scenarios in which you might create threads or use thread pool threads.

[Exceptions in Managed Threads](https://docs.microsoft.com/en-us/dotnet/standard/threading/exceptions-in-managed-threads)  
Describes the behavior of unhandled exceptions in threads for different versions of .NET, in particular the situations in which they result in termination of the application.

[Synchronizing Data for Multithreading](https://docs.microsoft.com/en-us/dotnet/standard/threading/synchronizing-data-for-multithreading)  
Describes strategies for synchronizing data in classes that will be used with multiple threads.

[Foreground and Background Threads](https://docs.microsoft.com/en-us/dotnet/standard/threading/foreground-and-background-threads)  
Explains the differences between foreground and background threads.

[Managed and Unmanaged Threading in Windows](https://docs.microsoft.com/en-us/dotnet/standard/threading/managed-and-unmanaged-threading-in-windows)  
Discusses the relationship between managed and unmanaged threading, lists managed equivalents for Windows threading APIs, and discusses the interaction of COM apartments and managed threads.

[Thread Local Storage: Thread-Relative Static Fields and Data Slots](https://docs.microsoft.com/en-us/dotnet/standard/threading/thread-local-storage-thread-relative-static-fields-and-data-slots)  
Describes thread-relative storage mechanisms.

**Reference**

[Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread)  
Provides reference documentation for the **Thread** class, which represents a managed thread, whether it came from unmanaged code or was created in a managed application.

[BackgroundWorker](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.backgroundworker)  
Provides a safe way to implement multithreading in conjunction with user-interface objects.

**Related sections**

[Overview of Synchronization Primitives](https://docs.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives)  
Describes the managed classes used to synchronize the activities of multiple threads.

[Managed Threading Best Practices](https://docs.microsoft.com/en-us/dotnet/standard/threading/managed-threading-best-practices)  
Describes common problems with multithreading and strategies for avoiding problems.

[Parallel Programming](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/)  
Describes the Task Parallel Library and PLINQ, which greatly simplify the work of creating asynchronous and multi-threaded .NET applications.

**Using threads and threading**

With .NET, you can write applications that perform multiple operations at the same time. Operations with the potential of holding up other operations can execute on separate threads, a process known as *multithreading* or *free threading*.

Applications that use multithreading are more responsive to user input because the user interface stays active as processor-intensive tasks execute on separate threads. Multithreading is also useful when you create scalable applications, because you can add threads as the workload increases.

**Note**

If you need more control over the behavior of the application's threads, you can manage the threads yourself. However, multithreaded programming is greatly simplified with the [**System.Threading.Tasks.Parallel**](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel) and [**System.Threading.Tasks.Task**](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task) classes, [**Parallel LINQ (PLINQ)**](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq), concurrent collection classes in the **[System.Collections.Concurrent](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent)** namespace, and a programming model that is based on the concept of tasks rather than threads. For more information, see [**Parallel Programming**](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/) and [**Task Parallel Library (TPL)**](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl).

**How to: Create and start a new thread**

You create a new thread by creating a new instance of the [System.Threading.Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread) class and providing the name of the method that you want to execute on a new thread to the constructor. To start a created thread, call the [Thread.Start](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.start) method. For more information and examples, see the [Creating threads and passing data at start time](https://docs.microsoft.com/en-us/dotnet/standard/threading/creating-threads-and-passing-data-at-start-time) article and the [Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread) API reference.

**How to: Stop a thread**

To terminate the execution of a thread, use the [System.Threading.CancellationToken](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken). It provides a unified way to stop threads cooperatively. For more information, see [Cancellation in managed threads](https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads).

Sometimes it is not possible to stop a thread cooperatively, because it runs third-party code not designed for cooperative cancellation. In this case, you might want to terminate its execution forcibly. To terminate the execution of a thread forcibly, in .NET Framework you can use the [Thread.Abort](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.abort) method. That method raises a [ThreadAbortException](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadabortexception) on the thread on which it's invoked. For more information, see [Destroying threads](https://docs.microsoft.com/en-us/dotnet/standard/threading/destroying-threads). The [Thread.Abort](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.abort) method is not supported in .NET Core. If you need to terminate the execution of third-party code forcibly in .NET Core, run it in the separate process and use [Process.Kill](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.kill).

The [System.Threading.CancellationToken](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken) is not available before .NET Framework 4. To stop a thread in older .NET Framework versions, implement the cooperative cancellation manually using the thread synchronization techniques. For example, you can create the volatile boolean field shouldStop and use it to request the code executed by the thread to stop. For more information, see [volatile](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/volatile) in C# Reference and [System.Threading.Volatile](https://docs.microsoft.com/en-us/dotnet/api/system.threading.volatile).

Use the [Thread.Join](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.join) method to make the calling thread wait for the termination of the thread being stopped.

**How to: Pause or interrupt a thread**

You use the [Thread.Sleep](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.sleep) method to pause the current thread for a specified amount of time. You can interrupt a blocked thread by calling the [Thread.Interrupt](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.interrupt) method. For more information, see [Pausing and interrupting threads](https://docs.microsoft.com/en-us/dotnet/standard/threading/pausing-and-resuming-threads).

**Thread properties**

The following table presents some of the [Thread](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread) properties:

| **THREAD PROPERTIES** | |
| --- | --- |
| **Property** | **Description** |
| [IsAlive](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.isalive) | Returns true if a thread has been started and has not yet terminated normally or aborted. |
| [IsBackground](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.isbackground) | Gets or sets a Boolean that indicates if a thread is a background thread. Background threads are like foreground threads, but a background thread doesn't prevent a process from stopping. Once all foreground threads that belong to a process have stopped, the common language runtime ends the process by calling the [Abort](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.abort) method on background threads that are still alive. For more information, see [Foreground and Background Threads](https://docs.microsoft.com/en-us/dotnet/standard/threading/foreground-and-background-threads). |
| [Name](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.name) | Gets or sets the name of a thread. Most frequently used to discover individual threads when you debug. |
| [Priority](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.priority) | Gets or sets a [ThreadPriority](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpriority) value that is used by the operating system to prioritize thread scheduling. For more information, see [Scheduling threads](https://docs.microsoft.com/en-us/dotnet/standard/threading/scheduling-threads) and the [ThreadPriority](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpriority) reference. |
| [ThreadState](https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.threadstate) | Gets a [ThreadState](https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadstate) value containing the current states of a thread. |