Multithreading

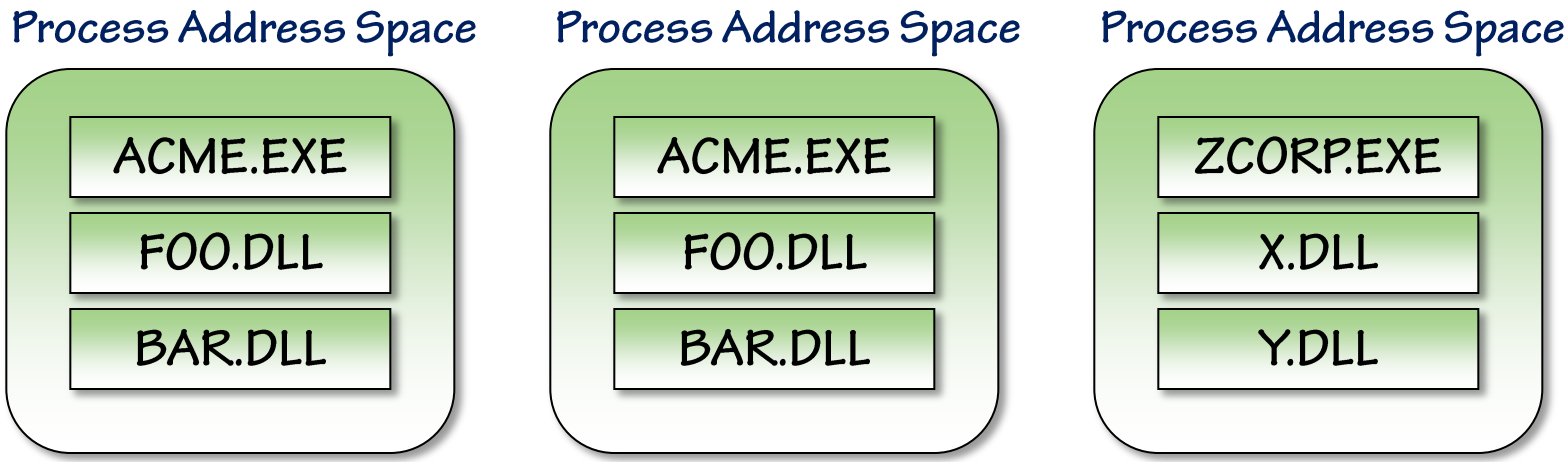
introduction to concurrent execution in .NET

Overview

- **Introduction to multithreading in .NET**
  - Vocabulary: process versus thread
  - Multi-threading use-cases
  - Multi-threading caveats
  - Working with threads
    - starting
    - argument passing
    - shutdown coordination
  - Working with the thread pool
    - directly
    - asynchronous delegate invocation
    - asynchronous I/O

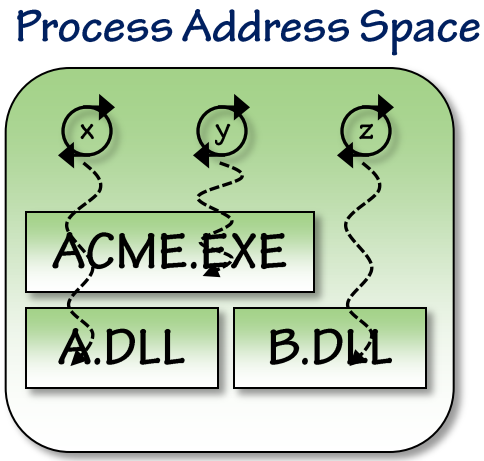
Process Anatomy

- **A *process* is an inert container**
  - Defines a virtual address space
    - contents not addressable from another process
  - Libraries of code are mapped into the address space
    - 1 EXE + N DLLs (dynamically loaded libraries)



Processes & Threads

- ***Threads* execute code**
  - A path of execution through any/all code within a single process
  - Have access to any/all data within that process
    - for managed threads, within an AppDomain
  - Each thread has its own callstack & copy of the CPU registers
    - technically, a CLI implementation-specific detail
  - A process with no threads exits (because it can no longer perform work)

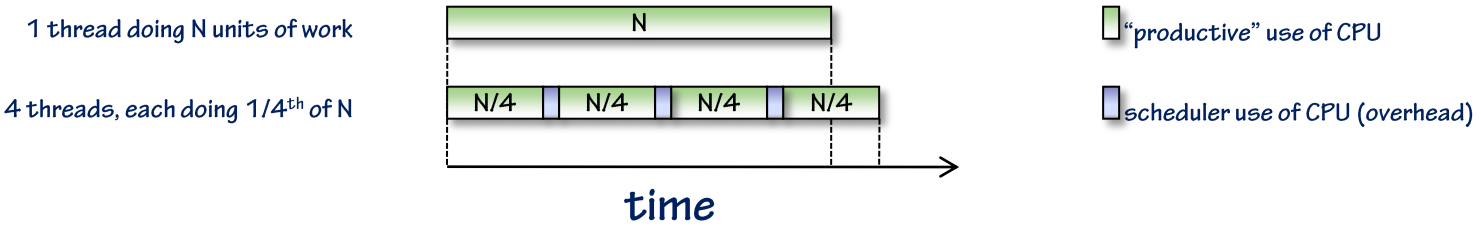


Multi-threading Use-Cases

- **The benefits of leveraging multi-threading include**
  - Opportunity to scale by parallelizing CPU-bound operations
    - assuming multi-core/multi-processor hardware
  - Perform CPU-bound work while I/O operations are waiting
  - Maintain a responsive user interface
    - farming off lengthy and/or blocking operations to a separate thread
    - using thread priorities to ensure the “UI thread” has priority

Multi-threading Caveats

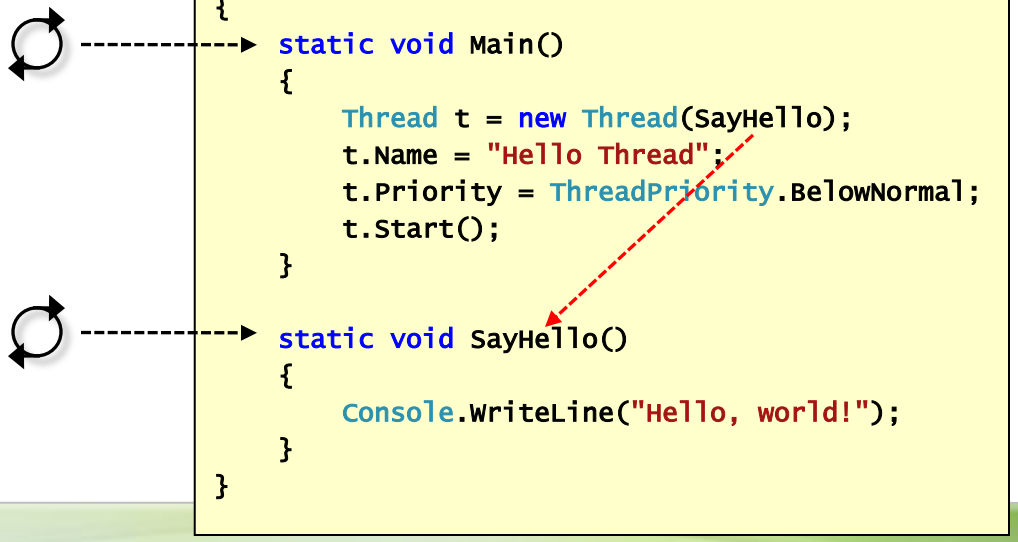
- **The price to pay for multi-threading includes**
  - Slower execution time on single-core/processor machines
    - context-switching overhead means…



  - Added program complexity
    - lines of code
    - readability/maintainability
    - debuggability
    - testability

Starting Threads

- **Threads are started explicitly using System.Threading.Thread**
  - Constructor is used to set *thread entry point*
  - Properties can be set post-construction; prior to thread starting
  - Thread execution begins when ***Start*** is called



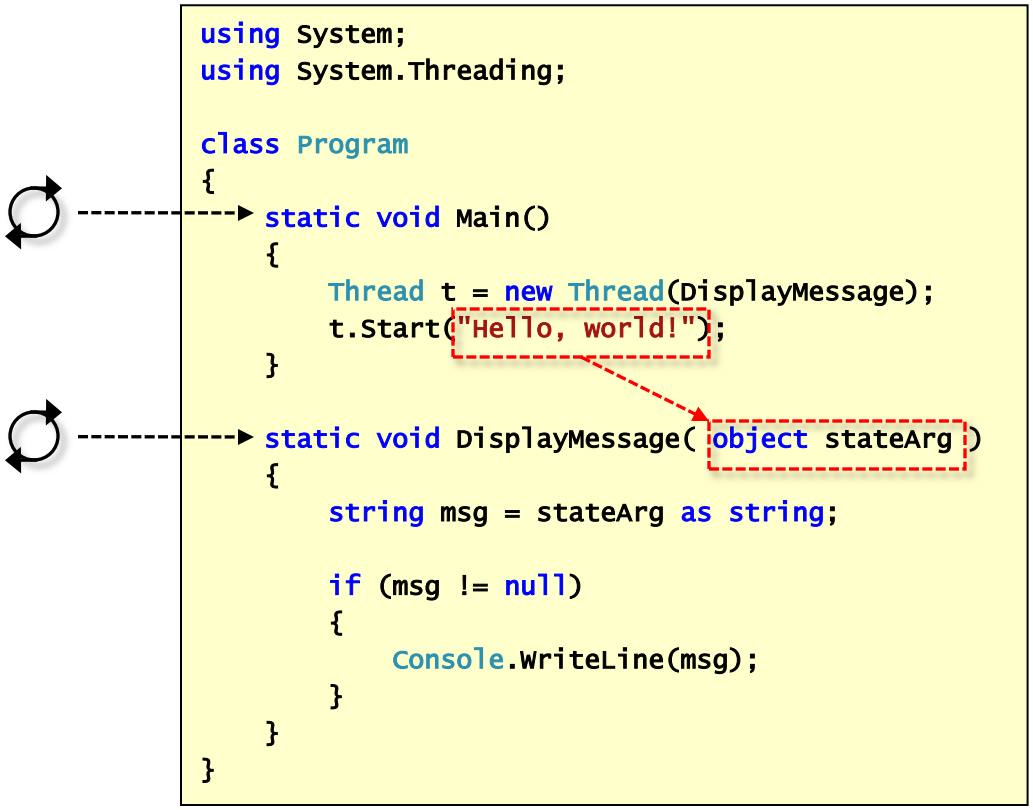
```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        Thread t = new Thread(SayHello);
        t.Name = "Hello Thread";
        t.Priority = ThreadPriority.BelowNormal;
        t.Start();
    }

    static void SayHello()
    {
        Console.WriteLine("Hello, world!");
    }
}
```

Thread Entry Point Methods

- **Thread entry point methods must have one of two signatures**
  - `void EntryPointMethod()`
    - MSDN/***ThreadStart***
  - `void EntryPointMethod( object stateArg )`
    - MSDN/***ParameterizedThreadStart***
  - method may be instance or static

ParameterizedThreadStart

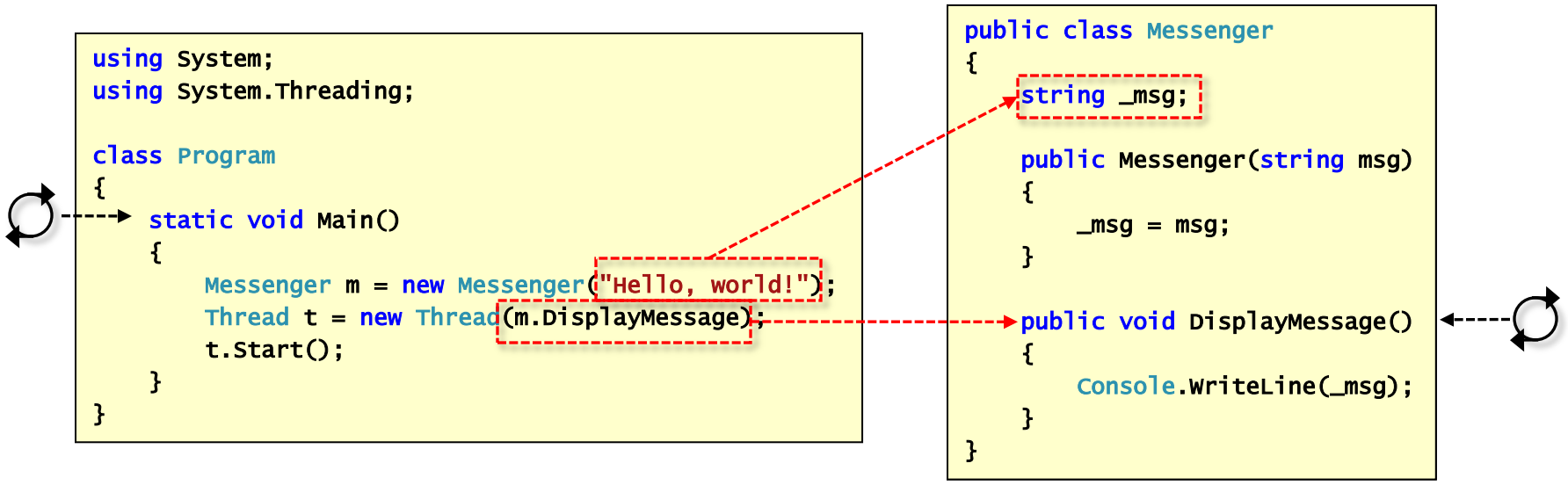
- **Thread.Start can pass an arbitrary object reference to a thread**
  - "sender" and "recipient" must agree on the type of the referenced object



```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        Thread t = new Thread(DisplayMessage);
        t.Start("Hello, world!");
    }

    static void DisplayMessage( object stateArg )
    {
        string msg = stateArg as string;

        if (msg != null)
        {
            Console.WriteLine(msg);
        }
    }
}
```

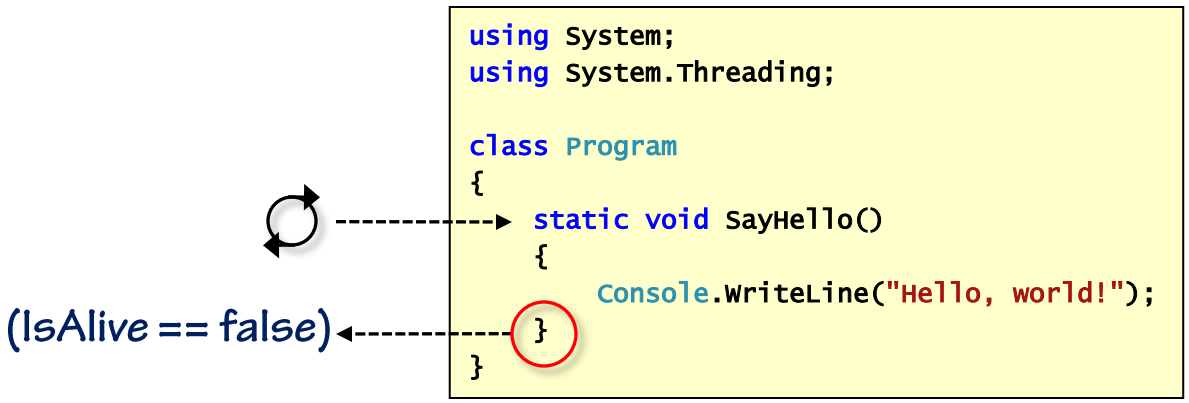
Instance Methods

- **Instance methods are also suitable for thread entry point methods**
  - Method has access to object fields like any other instance method would



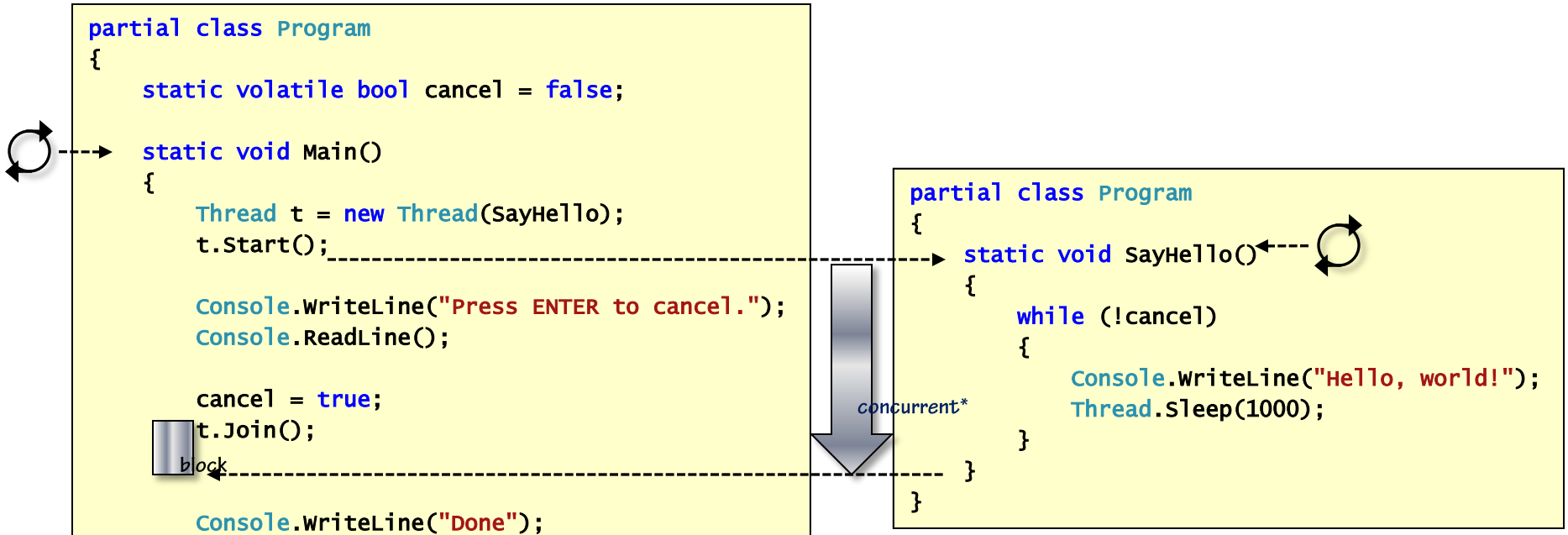
Thread Lifetime

- **Execution continues until thread returns from its entry point method**
  - As a result of a standard method return
  - As result of an unhandled exception
    - encountered by the thread itself ("synchronous exception")
    - induced by another thread using ***Interrupt*** or ***Abort*** ("asynchronous exception")
  - ***IsAlive*** provides an *instantaneous snapshot* of thread execution state



Coordinating Thread Shutdown

- **Ideally, thread shutdown is choreographed**
  - User-defined mechanism used to request orderly shutdown
  - Requesting thread waits until the CLR confirms the thread has exited
    - not so good: polling ***IsAlive***
    - better: calling ***Join***



Thread Pool

- **The CLR also provides a per-process thread pool**
  - Allows threads to be "borrowed" for relatively brief concurrent operations
  - CLR adds threads to, removes threads from the pool based on demand
  - Allows cost of thread start up & teardown to be amortized over life of app
  - Pooled threads have ***IsBackground*** property set to true

- **Three styles of interaction with the thread pool are supported**
  - ***ThreadPool.QueueUserWorkItem***
  - ***Delegate.BeginInvoke***
  - Asynchronous I/O

ThreadPool.QueueUserWorkItem

- **Queues a request for pooled thread to call a given callback**
  - Optional state argument may be passed to callback with request
  - Callback requests (and optional argument) are stored in a FIFO queue
  - Multiple reader threads means callback invocation order is **not** guaranteed

```
using System;
using System.Threading;

class Program
{
    static void Main()
    {
        ThreadPool.QueueUserWorkItem(DisplayMessage, "Hello, world!");
    }

    static void DisplayMessage(object stateArg)
    {
        Console.WriteLine(stateArg);
    }
}
```

Delegates & Async I/O

- **Delegates provide a type-aware interface to the thread pool**
  - Classes that represent a method call (signature & target instance)
  - Compiler-generated ***BeginInvoke*** method matches signature
  - CLR implements ***BeginInvoke*** at run-time to queue request to thread pool

- **Asynchronous I/O is a scalable I/O-centric interface to the thread pool**
  - QueueUserWorkItem/Delegates
    - dispatch a pooled thread to call a method that may take a long time to complete
  - Async I/O
    - queue a non-blocking I/O request to a device (***BeginRead***, ***BeginWrite***, et. al.)
    - dispatch a pooled thread to call a method notifying you *when I/O completes*

Summary

- **Threads are the core concurrent programming construct in .NET**
  - ***System.Threading.Thread***
  - Contained within/confined to a single process
  - Share access to any/all data within that process/AppDomain
  - Independently prioritized & scheduled for CPU time by the OS
    - when backed by a Win32 thread
  - Multiple techniques provided for leveraging threads
    - manual/explicit
    - thread pool
    - asynchronous I/O