

Async Execution



4: Async Execution

The need for Async programming

- **Async techniques keep programs responsive**
 - users *hate* hanging UIs
- **Async techniques increasingly important for performance**
 - future CPUs *won't* have significantly higher clock speeds
 - future CPUs *will* rely on multiple cores for performance

developmentor



4: Async Execution

The .NET Asynchronous Execution pattern

- **Many Framework classes expose Async APIs**
 - BeginFoo accepts input parameters and starts operation
 - EndFoo collects return value (or raises deferred exception)
- **Call object mediates async operation**
 - used to query for completion status

```
class WebRequest {  
    public virtual WebResponse GetResponse();  
  
    public virtual IAsyncResult BeginGetResponse(  
        AsyncCallback callback, object state);  
  
    public virtual WebResponse EndGetResponse(  
        IAsyncResult asyncResult);  
}
```

developmentor



4: Async Execution

IAsyncResult

```
public interface IAsyncResult
{
    object AsyncState { get; }
    WaitHandle AsyncWaitHandle { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
}
```

```
WebRequest req = WebRequest.Create(url);
IAsyncResult ar = req.BeginGetResponse(null, null);
...
WebResponse response = req.EndGetResponse(ar);
```

developmentor



4: Async Execution

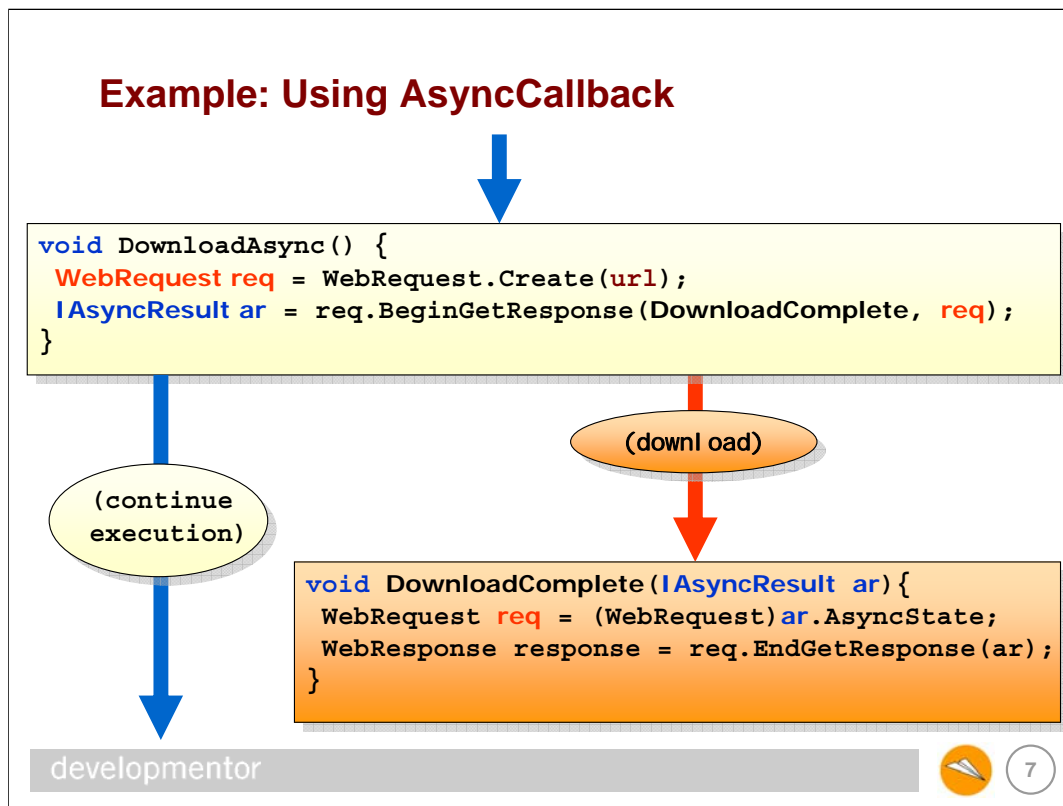
Extending the pattern - AsyncCallback

- Waiting for the async operation on the calling thread sort of defeats the purpose of Async..
- **AsyncCallback** pattern provides callback when operation is complete
 - "State" object (**IAAsyncResult.AsyncState**) used to hold contextual information

developmentor



4: Async Execution



4: Async Execution

Asynchrony via Delegates

- **Delegate classes support Async invocation**
 - BeginInvoke extends model to arbitrary code
 - all parameters mapped to BeginInvoke
 - **ref** and **out** parameters mapped to EndInvoke

```
delegate bool PrintDelegate(string title);
```

```
class PrintDelegate : System.MulticastDelegate  
{  
    public PrintDelegate(object, native int);  
    public IAsyncResult BeginInvoke(string title,  
        AsyncCallback cb, object State);  
    public bool EndInvoke(IAsyncResult ar);  
}
```

developmentor



8

4: Async Execution

Example: Using an Async delegate

```
void BeginPrint(string title){  
    PrintDelegate del = Print;  
    AsyncCallback cb = PrintComplete;  
    del.BeginInvoke(title, cb, null);  
}
```

`null` here indicates no state object needed

```
bool Print(string title){  
    return true;  
}
```

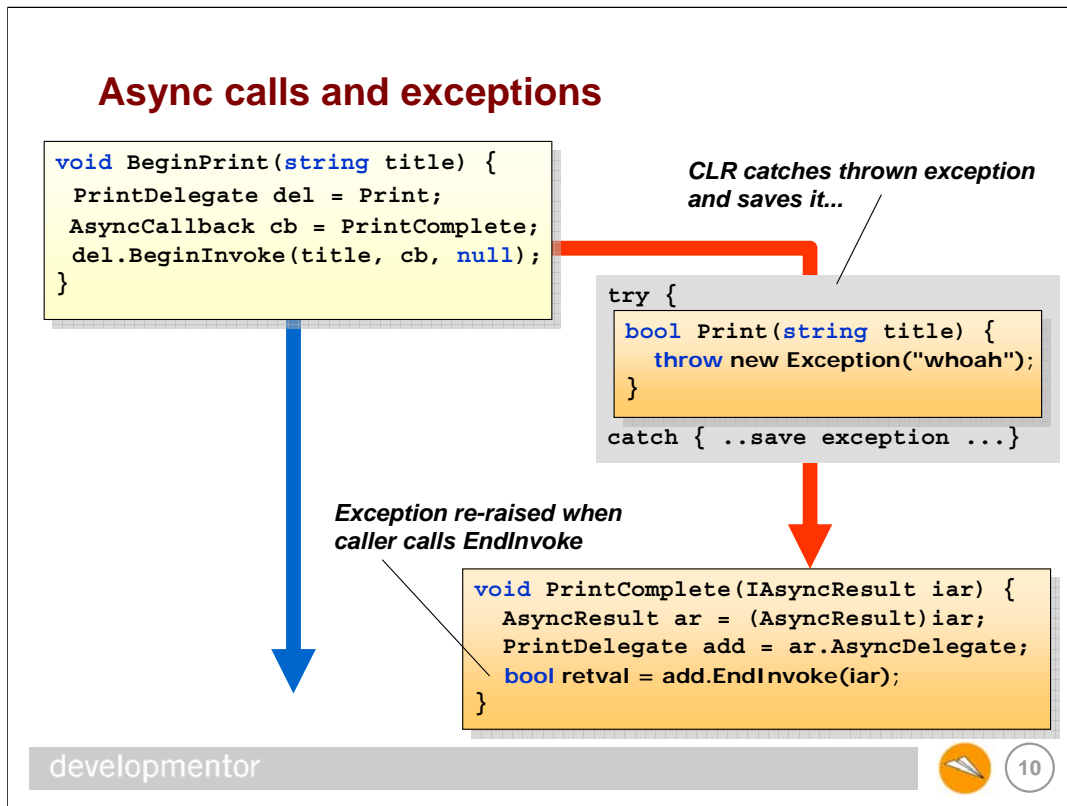
```
void PrintComplete(IAsyncResult iar){  
    AsyncResult ar = (AsyncResult)iar;  
    PrintDelegate add =  
        (PrintDelegate)ar.AsyncDelegate;  
    bool retval = add.EndInvoke(iar);  
}
```

developmentor



9

4: Async Execution



4: Async Execution

Threads

- **Async operations rely on independent threads**
 - independently scheduled units of execution
 - have their own variables and security context
- **CLR maps Managed threads onto underlying OS threads**
 - mapping not necessarily 1:1!
 - CLR / Host may choose to do manual scheduling
- **Most Thread management done by the CLR**
 - custom threads sometimes needed

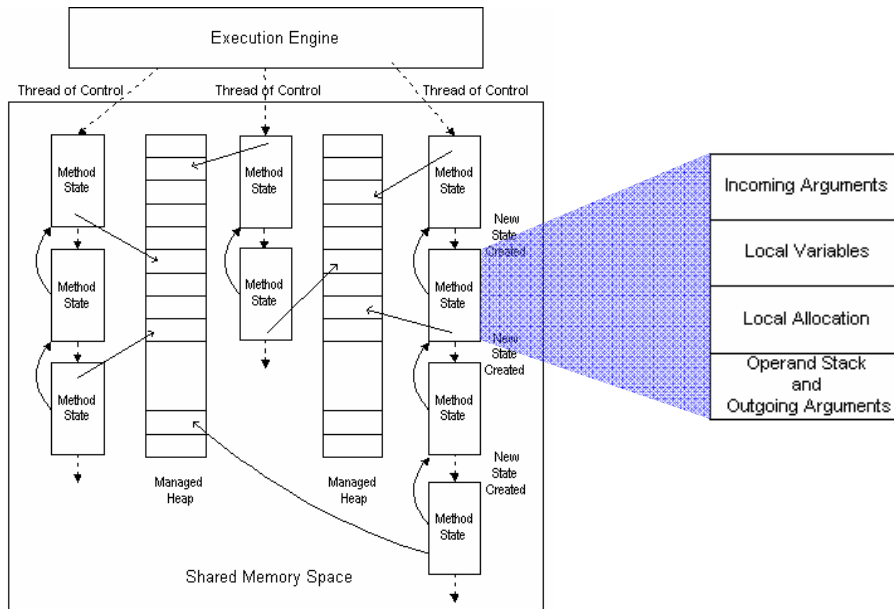
developmentor



“...a singly linked list of method states, where a new state is created and linked back to the current state by a method call instruction – the traditional model of a stack-based calling sequence.” (CLI specification Partition 1, 12.3.1)

4: Async Execution

Threads: The formal definition



developmentor



12

4: Async Execution

The thread pool

- **CLR-controlled thread pool used for most async needs**
 - services most **Begin / EndXXX** APIs
 - work items queued up by CLR
 - threads added and removed based on demand
- **Thread pool accessed several ways**
 - timers
 - `Delegate.BeginInvoke` (most common)
 - `ThreadPool` class

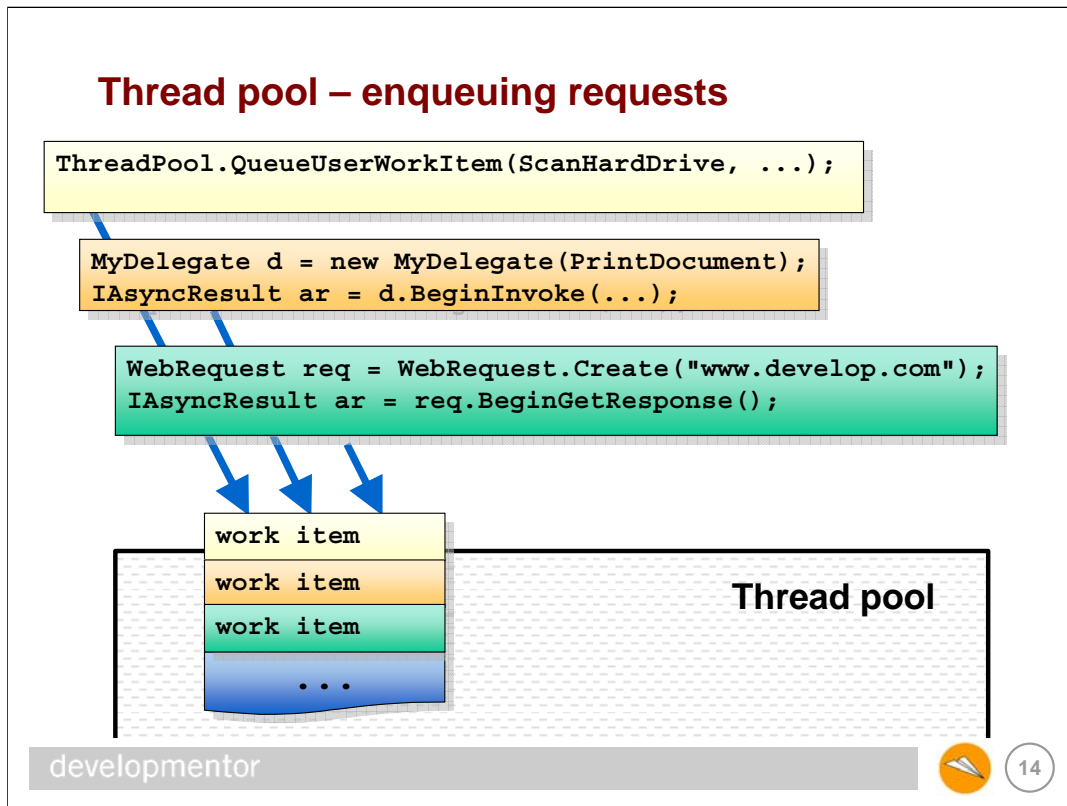
developmentor



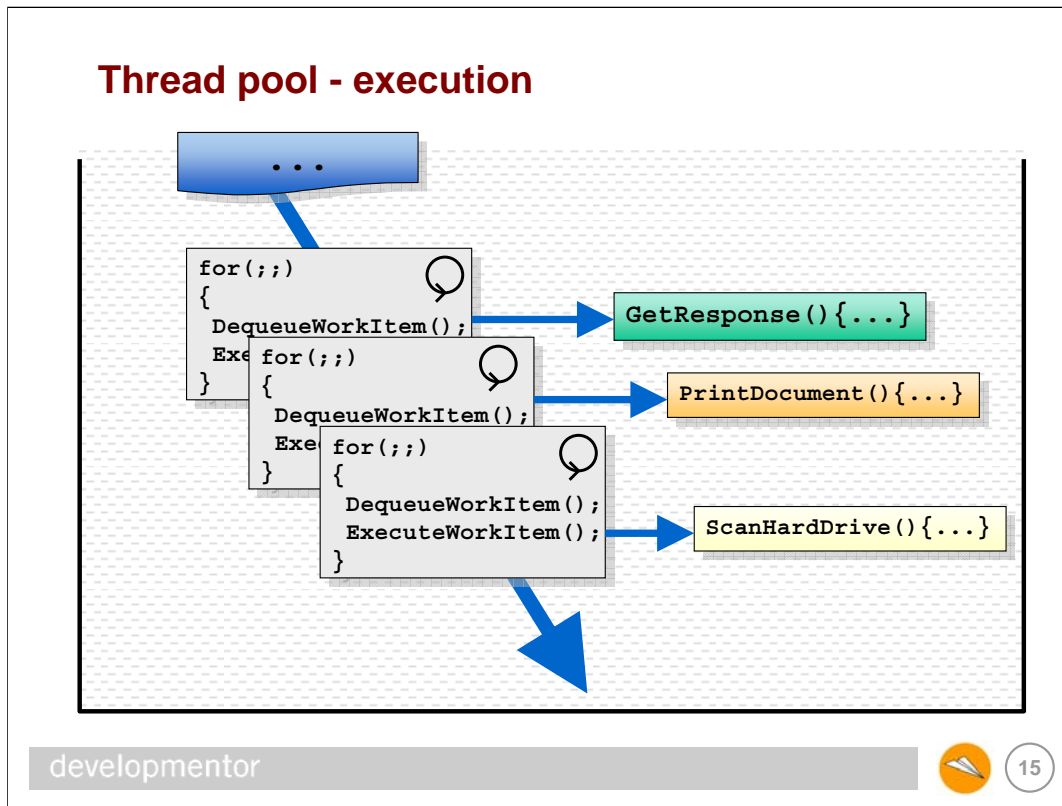
13

`Thread.IsThreadPoolThread` will tell you if a given thread originates from the thread pool.

4: Async Execution



4: Async Execution



4: Async Execution

Example: Exercising the thread pool

```
static void UsePool(int reps, FooDelegate f)
{
    WorkDelegate[] dels = new WorkDelegate[reps];
    IAsyncResult[] ars = new IAsyncResult[reps];

    for(int i = 0; i < reps; i++)
        dels[i] = f;

    for(int i = 0; i < reps; i++)
        ars[i] = dels[i].BeginInvoke(null, null);

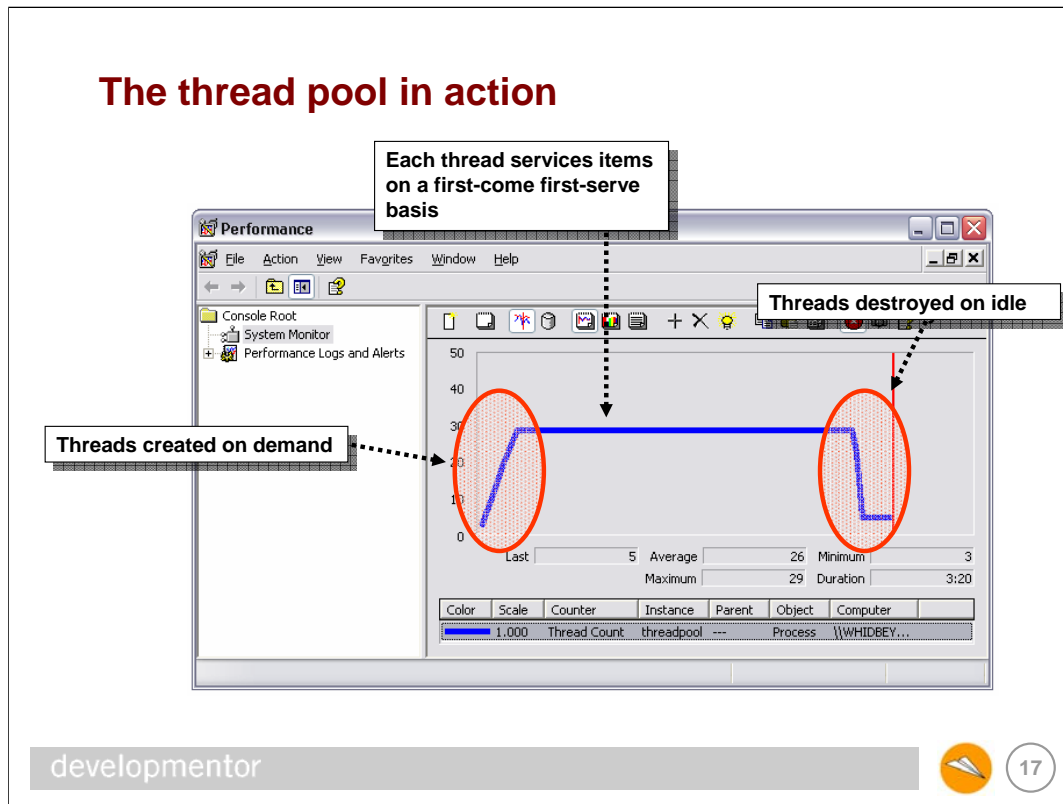
    for(int i = 0; i < reps; i++)
        dels[i].EndInvoke(ars[i]);
}
```

Place a request in the
ThreadPool queue



4: Async Execution

The thread pool in action



4: Async Execution

Controlling the thread pool



- **Min/Max thread pool size can be controlled via APIs**
 - `ThreadPool.SetMinThreads()` "warms up" the thread pool
 - avoids 500msec penalty for thread creation
 - `ThreadPool.SetMaxThreads()` caps number of threads
 - avoids thrashing under heavy load
 - defaults to 25 threads per CPU
- **Thread pool also controls overlapped I/O threads**
 - use same APIs to control I/O threads as well

developmentor



18

.NET version 1.1 didn't allow you to tweak the thread pool this way.

4: Async Execution

Custom threads

- **The thread pool isn't suitable for everything**
 - can't be used for common Interop scenarios
 - long-running tasks degrade effectiveness
 - tasks can't be prioritized
 - ThreadPool activity doesn't keep process alive
- **System.Threading.Thread** allows creation of custom threads

developmentor



19

4: Async Execution

System.Threading.Thread

- **Thread constructor takes a delegate parm**
 - `delegate void ThreadStart();`
 - 2.0 `delegate void ParameterizedThreadStart(object o);`
- **Thread constructor optionally takes Maximum Stack Size**
 - typically defaults to 1MB
- **Various properties can be set before starting**
 - **Name** property aids in debugging
 - **Priority**, **IsBackground** control behavior
 - properties also on current thread via **Thread.CurrentThread**
- **Start method begins async execution**

developmentor



ParameterizedThreadStart is new for version 2.0, as is the ability to control maximum stack size. You can also use the "anonymous delegate two step" to sneak parameters into threads, making ParameterizedThreadStart a little redundant in C# (though its use is admittedly much cleaner).

4: Async Execution

Example: Starting a custom thread

```
Thread t = new Thread(MyWorker);  
t.Name = "My Worker Thread";  
t.Priority = ThreadPriority.Normal;  
t.Start();
```

```
void MyWorker() {  
    Thread t = Thread.CurrentThread;  
    Console.WriteLine(t.Name);  
}
```

The screenshot shows a list of thread priorities: AboveNormal, BelowNormal, Highest, Lowest, and Normal. A red 'DANGER' warning icon is positioned next to the 'Normal' priority level. A red arrow points from the 't.Start()' line in the first code block to the 'MyWorker()' method in the second code block. A blue arrow points from the first code block down to the 'developer' status bar. The status bar also contains a '21' in a circle and a 'DANGER' icon.

Note that you probably should not set the Priority of threads unless you really know what you are doing. Mixing threads with differing priorities can introduce all sorts of nasty problems.

Stopping a thread

- **Execution ends when target method returns**
 - as a result of method logic
 - as a result of an unhandled exception
- **Calling threads can set shared variable to signal child thread to exit**
 - use `Thread.Join` to block until child thread exits
- **`Thread.Interrupt` and `Thread.Abort` can be used if you're desperate**
 - try to avoid these API's
 - CLR makes best effort; not guaranteed



4: Async Execution

Example: Orderly shutdown

```
static volatile bool Cancelled = false;

bool StopPrinting(Thread t,
                  int timeout) {
    Cancelled = true;
    if( t.Join(timeout) )
        return true;
    else
        return false;
}
```

```
void PrintThread() {
    while( !Cancelled && !Done) {
        PrintNextPage();
    }
}
```

Shared variable

Set Cancellation notice and wait for the thread to exit

Occasionally check shared variable

developmentor



23

Coordinating multithreaded code

- **Synchronization code is usually left to the application writer**
 - few objects written with async access in mind
 - hard to anticipate how clients will use objects
 - framework classes usually document expectations
- **Some Framework classes use `Synchronized` pattern**
 - `System.Collections` classes and a few others
 - instances of class are not thread-safe
 - static `Synchronized` method creates thread-safe accessor
- **The best strategy: Share as little as possible between threads**
 - even if it means copying nontrivial amounts of memory

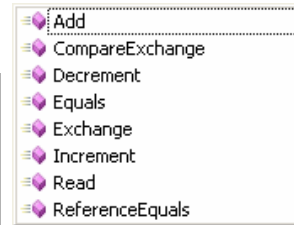


4: Async Execution

Interlocked

- **Interlocked class provides thread-safe numeric operations**
 - primarily on ints, with some ability for doubles, longs

```
class SaferX {  
    static int Value = 0;  
  
    public static void Increment() {  
        Interlocked.Increment(ref Value);  
    }  
    public static void IncrementBy(int x) {  
        Interlocked.Add(ref Value, x);  
    }  
    public static void Decrement() {  
        Interlocked.Decrement(ref Value);  
    }  
}
```



developmentor



25

4: Async Execution

Locking for more complex scenarios

- Interlocked can't help when multiple operations have to be performed atomically

```
class SmallBusiness {  
    int Cash = 0;  
    int Receivables = 1000;  
  
    public void ReceivePayment(  
        int amount) {  
        Cash += amount;  
        Receivables -= amount;  
    }  
    public int NetWorth {  
        get { return Cash+Receivables; }  
    }  
}
```



What happens if thread is interrupted here and NetWorth is accessed from a different thread?

developmentor



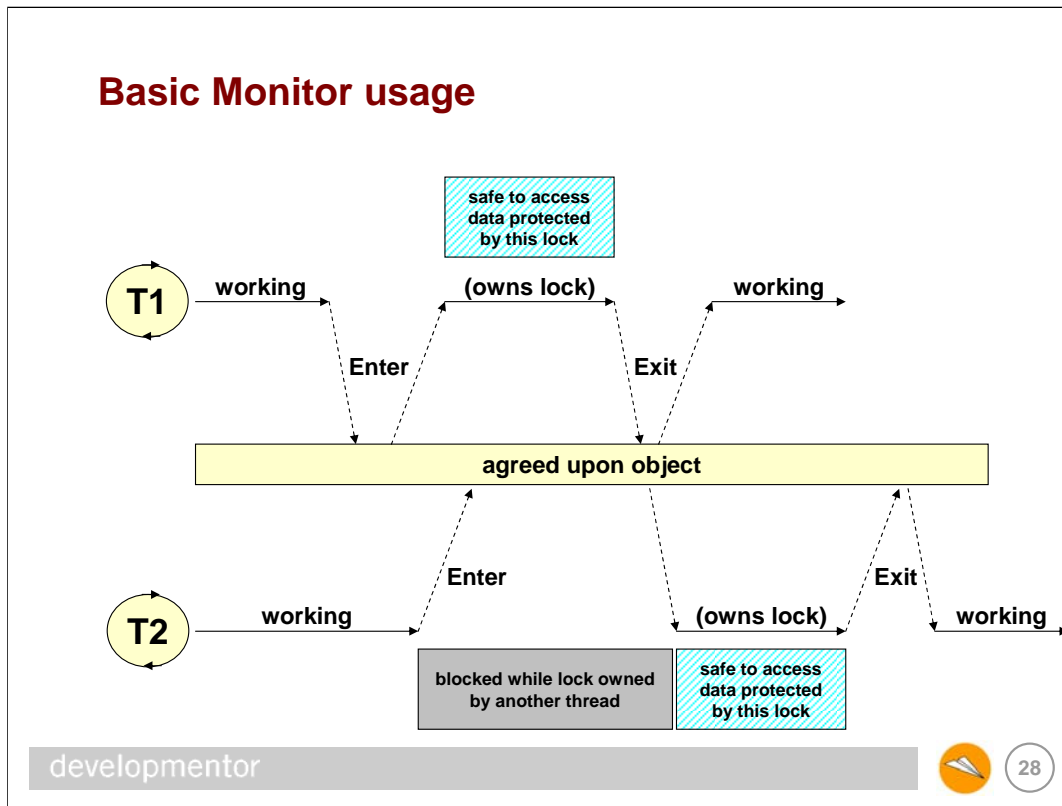
26

Mutual exclusion using Monitors

- **Monitor** provides a mechanism to ensure that "critical sections" of code never run simultaneously
 - integrated into CLR
 - integrated into C# programming language
- **Coordination requires use of a shared lock object**
 - any object will do
 - code regions must explicitly acquire lock before touching shared objects
- **C# lock construct automates Monitor use**
 - wraps Enter / Exit calls in try...finally block



4: Async Execution



4: Async Execution


Using Monitors

```
class SmallBusiness {  
    int Cash = 0;  
    int Receivables = 1000;  
    private object accountLock = new object();  
    ...  
}
```

```
public void ReceivePayment(int amount) {  
    Monitor.Enter(accountLock);  
    Cash += amount;  
    Receivables -= amount;  
    Monitor.Exit(accountLock);  
}
```

```
public int NetWorth {  
    get {  
        lock(accountLock)  
        { return Cash+Receivables; }  
    }  
}
```

DANGER
what if there is an exception?

developmentor  29

Note that the middle example doesn't wrap Monitor use with a try...finally block. This is just asking for trouble, which is why lock { } is often preferred. If you need to catch application exceptions you can either nest your own try...finally in the lock block or just write your own try...finally and be sure to call Monitor.Exit in it.

4: Async Execution

Grace under pressure

- Don't lock (**this**) !
 - leaves you vulnerable to someone else using you for a lock
- **Monitor.TryEnter** more robust than **lock**
 - timeout and emit diagnostic code

```
public void ReceivePayment(int amount){  
    bool success = Monitor.TryEnter(accountLock, 5000);  
    if(success){  
        try {  
            Cash += amount;  
            Receivables -= amount; }  
        finally { Monitor.Exit(accountLock); }  
    }  
    else  
        EventLog.WriteEntry("Failed to acquire Monitor!");  
}
```

developmentor



30

4: Async Execution

Summary: Async Dos and Don'ts

- **Do use asynchronous programming when appropriate**
 - increased throughput of I/O-bound programming
 - take advantage of multiple CPU cores
- **Don't create more threads than you absolutely have to**
- **Do prefer the thread pool to custom threads**
 - but *Don't* abuse thread pool threads
- **Don't use Abort / Interrupt except in times of dire need**
- **Do Expect to spend a lot more time debugging...**
 - starting Async operations is easy
 - making them robust is hard. *Really* hard.

