

# Analysis task: Common concurrency issues

---

Objektumorientált szoftvertervezés  
Object-oriented software design

Dr. Balázs Simon  
BME, IIT

# Outline

---

- Lambda expressions and captured variables
- Exception on a different thread
- Stopping a thread
- Using a string literal synchronizing
- Using a new object when synchronizing
- Synchronizing on a non-static field
- Double-checked locking with additional initialization
- Modifying a collection while iterating through it
- Unbalanced locking
- Composition of atomic operations
- Temporary values in fields
- Not making static variables and methods thread-safe
- Arbitrary method calls from within locks
- Starting threads from within constructors
- Starting threads manually

# Problem: Lambda expressions and captured variables

---

```
for (int i = 0; i < 10; i++)  
    new Thread(() => Console.Write(i)).Start();
```

Problem: the output is not deterministic.

A possible outcome: 0223557799

The reason is that the *i* variable refers to the same memory location throughout the loop's lifetime.

Solution: capture the value in a temporary variable.

# Solution: Capture value in temporary variable

---

```
for (int i = 0; i < 10; i++)  
{  
    int tmp = i;  
    new Thread(() => Console.Write(tmp)).Start();  
}
```

Solution: capture the value in a temporary variable.

# Problem: Exception on a different thread

---

```
public static void Main()
{
    try
    {
        new Thread(Go).Start();
    }
    catch (Exception ex)
    {
        Console.WriteLine("Exception!");
    }
}
static void Go() { throw new InvalidOperationException(); }
```

Problem: the exception will never be caught.

Each thread has an independent execution path (own execution stack).

Solution: catch the exception in the same thread it was thrown.

# Solution: catch the exception in the same thread

---

```
public static void Main()  
{  
    new Thread(Go).Start();  
}
```

```
static void Go()  
{
```

```
    try
```

```
    {
```

```
        // ...
```

```
        throw new InvalidOperationException();
```

```
        // ...
```

```
    }
```

```
    catch (Exception ex)
```

```
    {
```

```
        //Typically log the exception, and/ or signal another thread
```

```
        // ...
```

```
    }
```

```
}
```

Solution: catch the exception in the same thread it was thrown.

# Problem: Stopping a thread

---

```
class MyThread
{
    private bool stop = false;

    public void Run()
    {
        while (!stop)
        {
            // do some work
        }
    }

    public void Stop()
    {
        this.stop = true;
    }
}
```

```
var myThread = new MyThread();
new Thread(myThread.Run).Start();
// ...
myThread.Stop();
```

Problem: the thread may never stop

Solution: “stop” should be volatile

# Solution with volatile

```
class MyThread
{
    private volatile bool stop = false;

    public void Run()
    {
        while (!stop)
        {
            // do some work
        }
    }

    public void Stop()
    {
        this.stop = true;
    }
}

var myThread = new MyThread();
new Thread(myThread.Run).Start();
// ...
myThread.Stop();
```

Volatile is about visibility. Threads cache values of variables. Volatile ensures that these caches are updated when the value of the variable changes.

If another thread calls Stop(), MyThread may never see the new value of “stop” if it is not marked as volatile.

Also, if volatile is missing, Run() may be optimized to:

```
while (true) {
    // do some work
}
```



# Problem: Using a String literal when locking

---

```
//use matching strings in two different libraries
private const string locker = "LOCK";

public void DoSomestuff()
{
    lock (locker)
    {
        this.Work();
    }
}
```

Problem: because strings literals are interned in Java and .NET, the literal string "LOCK,, turns out to be the same instance of String (even though they are declared completely disparately from each other). This can cause dead-locks!

Solution: never use a string literal as a lock object

# Lock object solution

---

```
//use a new readonly object instance
private static readonly object locker = new object();

public void DoSomestuff()
{
    lock (locker)
    {
        this.Work();
    }
}
```

# Problem: Using a new object when locking

---

```
private void DoSomething()  
{  
    lock (new object())  
    {  
        this.Work();  
    }  
}
```

Problem: because a new object is created every time, each thread will lock on a different object, and hence, there is no mutual exclusion!

Solution: always use the same object for protecting the same thing.

# Problem: Replacing the locker object

---

```
private List<string> items = new List<string>();

public void DoSomestuff()
{
    lock (items)
    {
        this.Work();
        items = new List<string>();
        this.MoreWork();
    }
}
```

Problem: replacing the object you lock on.

Other concurrent threads are then locking on a different object and this block does not provide the mutual exclusion you expect.

Solution: always lock on readonly fields!

# Solution: always lock on readonly fields

---

```
//use a new readonly object instance
private static readonly object locker = new object();
private List<string> items = new List<string>();

public void DoSomestuff()
{
    lock (locker)
    {
        this.Work();
        items = new List<string>();
        this.MoreWork();
    }
}
```

The “readonly” keyword ensures that you cannot replace the locker object.

Locking on non-readonly field is most likely broken.

# Problem: Double-checked locking with additional initialization

```
public class Singleton
{
    private static object locker = new object();
    private volatile static Singleton singleton = null;

    private Singleton() {}

    public SharedObject Value { get; set; }

    public static Singleton GetInstance()
    {
        if (singleton == null)
        { // 1st check
            lock (locker)
            {
                if (singleton == null)
                { // 2nd (double) check
                    singleton = new Singleton();
                    singleton.Value = new SharedObject();
                }
            }
        }
        return singleton;
    }
}
```

 Problem: additional initialization in double-checked locking

# Problem: Double-checked locking

---

- Double-checked locking is usually implemented incorrectly
- Double-checked locking may not be able to be implemented correctly at all, if the programming language is flawed
- Additional initialization after the constructor call has the same problem as the partially executed constructor
  - this is very nasty, since this is easy to forget
- Therefore, double-checked locking is considered by many an anti-pattern!
- Possible solutions:
  - don't use double-checked locking, always use just simple lock
  - don't forget the volatile keyword, make sure you can implement the pattern correctly, and also that the language has no concurrency flaws
  - don't do additional initialization after the constructor call, or protect the initialization with memory barrier or proper signaling (e.g. `ManualResetEvent`)


# Solution with memory barrier

```
public class Singleton {
    private static object locker = new object();
    private volatile static Singleton singleton = null;

    private Singleton() { }
    public SharedObject Value { get; set; }

    public static Singleton GetInstance()
    {
        if (singleton == null)
        { // 1st check
            lock (locker)
            {
                if (singleton == null)
                { // 2nd (double) check
                    Singleton temp = new Singleton();
                    temp.Value = new SharedObject();
                    Thread.MemoryBarrier();
                    singleton = temp;
                }
            }
        }
        return singleton;
    }
}
```

Memory barrier prevents  
the compiler optimization  
to put this line earlier





# Problem: Modifying a collection while iterating through it

---

```
List<string> list = new List<string>() { "a", "b", "c" };  
foreach (var item in list)  
{  
    list.Remove(item);  
}
```

Problem: we get an “InvalidOperationException: Collection was modified”

This can happen in single-threaded and multi-threaded applications, too!

Solution:

- \* In the multi-threaded case:

  - use lock around reading/iterating/modifying the collection!

- \* In the single-threaded case:

  - use a copy of the collection to iterate through.

# Problem: Unbalanced locking

---

```
Dictionary<string, int> count = new Dictionary<string, int>();  
count.Add("apple", 5);
```

```
new Thread(() => {  
    lock (locker)  
    {  
        count["apple"] += 1;  
    }  
}).Start();
```

```
int appleCount = count["apple"];
```

It is wrong to lock only for write operations and not for read operations!

When a collection is written or read, both the write and read operations must be protected by a lock!

This is also why client-side locking is dangerous: it may be forgotten.

**Solution: use server-side locking, i.e. thread-safe collections.**

# Solution: Server-side locking with thread-safe collection

---

```
ConcurrentDictionary<string, int> count =  
    new ConcurrentDictionary<string, int>();  
count.TryAdd("apple", 5);  
  
new Thread(() => {  
    count["apple"] += 1;  
}).Start();  
  
int appleCount = count["apple"];
```

# Problem: Composition of atomic operations

```
ConcurrentDictionary<string, int> count =  
    new ConcurrentDictionary<string, int>();  
count.TryAdd("apple", 5);  
  
new Thread(() => {  
    int value;  
    count.TryRemove("apple", out value);  
}).Start();
```

```
if (count.ContainsKey("apple"))  
{  
    count["apple"] += 1;  
}
```

Problem: using `ConcurrentDictionary` may give a false sense of thread-safety. Although `ContainsKey()` and the indexer are atomic by themselves, their composition is not!

(Iterating through the “count” variable on one thread and modifying it on another thread will not throw a `InvalidOperationException`, as in the non-thread-safe collection example!)

Solution: use the complex atomic operations of the `ConcurrentDictionary` or lock when you would like the combination of atomic operations to be atomic!

# Solution: Complex atomic operations

---

```
ConcurrentDictionary<string, int> count =  
    new ConcurrentDictionary<string, int>();  
count.TryAdd("apple", 5);
```

```
new Thread(() => {  
    int value2;  
    count.TryRemove("apple", out value2);  
}).Start();
```

The TryUpdate method updates the value of "apple" only if its current value matches the specified old value (value1).

```
int value1;  
if (count.TryGetValue("apple", out value1))  
{  
    count.TryUpdate("apple", value1 + 1, value1);  
}
```

Solution: use the complex atomic operations of the ConcurrentDictionary

# Solution: Locking for composition of atomic operations

---

```
ConcurrentDictionary<string, int> count =  
    new ConcurrentDictionary<string, int>();  
lock (locker)  
{  
    count.TryAdd("apple", 5);  
}  
  
new Thread(() => {  
    lock (locker)  
    {  
        int value;  
        count.TryRemove("apple", out value);  
    }  
}).Start();  
  
lock (locker)  
{  
    if (count.ContainsKey("apple"))  
    {  
        count["apple"] += 1;  
    }  
}
```

Solution: use lock even when you would like the combination of atomic operations to be atomic!

Might as well use the simple non-thread-safe Dictionary here.

# Problem: Temporary values in fields

```
public class Graph
{
    private List<Vertex> vertices;
    private List<Edge> edges;
    private HashSet<Vertex> visited;

    public IEnumerable<Vertex> DepthFirstSearch(Vertex start)
    {
        visited = new HashSet<Vertex>();
        this.DfsVisit(start);
        return visited;
    }

    private void DfsVisit(Vertex v)
    {
        if (visited.Contains(v)) return;
        visited.Add(v);
        foreach (var next in this.GetNeighbors(v))
        {
            this.DfsVisit(next);
        }
    }

    private IEnumerable<Vertex> GetNeighbors(Vertex v) { ... }
}
```

Temporary field for storing visited vertices.

Temporary field reset.

Problem: threads will override each other's temporary fields.

Solution: pass temporary values as parameters or create a separate class for a complex algorithm.

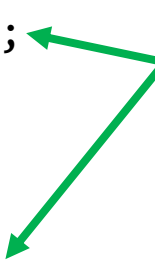
# Solution: pass temporary values as parameters

```
public class Graph
{
    private List<Vertex> vertices;
    private List<Edge> edges;

    public IEnumerable<Vertex> DepthFirstSearch(Vertex start)
    {
        HashSet<Vertex> visited = new HashSet<Vertex>();
        this.DfsVisit(start, visited);
        return visited;
    }

    private void DfsVisit(Vertex v, HashSet<Vertex> visited)
    {
        if (visited.Contains(v)) return;
        visited.Add(v);
        foreach (var next in this.GetNeighbors(v))
        {
            this.DfsVisit(next, visited);
        }
    }

    private IEnumerable<Vertex> GetNeighbors(Vertex v) { ... }
}
```



Separate collection for each thread.



# Solution: create separate class for complex algorithm

```
public class Graph
{
    private List<Vertex> vertices;
    private List<Edge> edges;

    public IEnumerable<Vertex> DepthFirstSearch(Vertex start)
    {
        Dfs dfs = new Dfs(this);
        return dfs.Execute(start);
    }

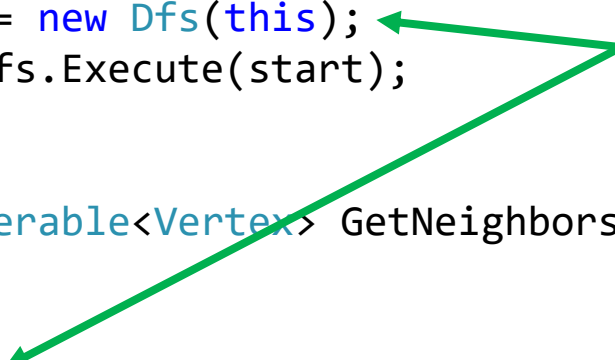
    public IEnumerable<Vertex> GetNeighbors(Vertex v) { ... }
}

public class Dfs
{
    private Graph graph;
    private HashSet<Vertex> visited;

    public Dfs(Graph graph)
    {
        this.graph = graph;
    }

    public IEnumerable<Vertex> Execute(Vertex start) { ... }
```

Separate instance of the algorithm for each thread.



## Problem: Not making static variables and methods thread-safe

---

- Static variables and methods may be accessed from multiple threads
- Always make public static variables and methods thread safe, even if you think you are writing a single-threaded application!
- Others may not use your code from a single thread, or later you may decide to make your application multi-threaded!
- .NET MSDN has the following note at the end of almost all classes:
  - *“Any public static (**Shared** in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.”*
- Threads can always make object instances visible only for themselves or they can protect them by locks
  - But threads cannot make static members thread-local!
- When you are writing a library/API: always document which classes/methods are thread-safe and which ones must only be called from a single thread!

## Problem: Arbitrary method calls from within locks

---

- Problem: calling delegates or unknown (e.g. virtual) methods from within a lock
- Method calls should not be made "into the wild" from within a lock
  - you have no idea if the call will be blocking, long-running or even result in deadlock
- Solution: Reduce the scope of the lock to just protect a critical *private* section of code
  - Call unknown functions only outside of a lock!

# Problem: Starting threads from within constructors

---

- If a constructor starts a thread and the object is no longer referenced, you won't be able to stop the thread
  - this can cause thread leaks and eventually thrashing
- If the class is extended, the thread can be started before subclass' constructor is executed
- If the constructor throws an exception after starting the thread, no one will be able to stop the thread
- Solution: never start a thread from within a constructor
  - use specific functions for starting and stopping the thread

# Problem: Starting threads manually

---

- Calling new Thread() manually is dangerous
  - you can spawn many threads
  - some of them you may forget to stop
  - this may cause thrashing
- Use the dedicated background execution services built into the standard libraries
  - they maintain a thread-pool with a number of threads appropriate for the given machine (processors and cores)
  - .NET: System.Threading.Tasks namespace
  - Java: java.util.concurrent.ExecutorService

# Summary

---

- Lambda expressions and captured variables
- Exception on a different thread
- Stopping a thread
- Using a string literal synchronizing
- Using a new object when synchronizing
- Synchronizing on a non-static field
- Double-checked locking with additional initialization
- Modifying a collection while iterating through it
- Unbalanced locking
- Composition of atomic operations
- Temporary values in fields
- Not making static variables and methods thread-safe
- Arbitrary method calls from within locks
- Starting threads from within constructors
- Starting threads manually