.NET Exception Handling Traps and Pitfalls

ACCU 2007

Seb Rose, Claysnow Limited

Let’s be CLR

- Managed code is executed by the Common Language Runtime (CLR)
- Managed applications are compiled to Common Intermediate Language (CIL)

Exceptions

- Exception handling in .NET is superficially very similar to Java/C++
- C# exception handling syntax looks very similar to Java/C++
- In many cases you can handle .NET exceptions just like Java/C++

RAII

- Resource Acquisition Is Initialisation (RAII) is an important idiom for managing resources.
- If constructor fails, no resources are retained
- Destructor releases all resources

RAII 2

- In C++, destructor will be called when stack allocated object goes out of scope
- This includes stack unwinding caused by unhandled exception
- .NET creates all objects on managed heap
- Calling of destructor is not deterministic (or even guaranteed)

RAII 3

- .NET provides IDisposable interface:
  interface IDisposable {
  void Dispose();
  }
- All resources should be released by calling Dispose
- Object should not be used after calling Dispose

RAII 4

- If you implement destructor & IDisposable:

```
class MyClass : IDisposable {
  void Dispose() {
     Dispose(true);
     GC.SuppressFinalize();}
  ~MyClass {Dispose(false);}
  void Dispose(bool disposing)
    { /* Release resources */ }
}
```

RAII 5

```
MyClass myObject = new MyClass();
try
{ … }
finally
{
  myObject.Dispose();
}
```

RAII 6

- C# provides *using* construct:
  using (MyClass myObject = new MyClass())
  { … }
- Which translates exactly to the code on previous slide
- To perform your own exception handling you will need to add your own *try/catch/finally* block(s).

RAII 7

- More than one *disposable* object:

```
using (MyClass obj1 = new MyClass()) {
  using (MyClass obj2 = new MyClass()) {
    …
  }
}
```

- Each *using* statement translates to a *try/finally* block, ensuring correct operation.

RAII 8

- The *using* statement can also be used with objects that already exist:

```
MyClass obj = new MyClass();
// You are responible for calling Dispose
using (obj) {
  // Dispose will be called when exiting block
}
// Dispose will ALWAYS have been called
```

Boxing & Wrapping

- CLR only allows references to be thrown
- If managed C++ code throws an *int* then this will be *boxed*
- An object of type *System::Int32* will be created on heap & value will be copied
- Logic of C++ code will not be affected, but consider interoperation with other components

CLS

- Common Language Specification (CLS) defines subset of CIL that languages should adhere to, to allow interaction between managed components written in different languages
- CLS says that all exceptions should derive from *System.Exception*

C# - to catch an *int*

- Prior to .NET 2.0, the only way C# code could catch an *int* exception would be in a catch-all block: *catch { … }*
- In a catch-all block there is no access to the actual exception that was thrown.
- .NET 2.0 introduced the *RuntimeWrappedException* class

RuntimeWrappedException

- Derives from *System.Exception*, so includes a stack trace
- Behaviour controlled by assembly attribute *WrapNonExceptionThrows*
- Sample code demonstrates some of this.

Win32

- A lot of functionality is provided in .NET framework, but there's plenty that you need to go direct to Win32 for.
- .NET allows direct access via platform invoke (P/Invoke) – just import the function and use it
- However, many Win32 calls return an error code rather than throw an exception.

Win32Exception

```
[DllImport("coredll.dll", SetLastError=true)]
public static extern int RegCreateKeyEx(…);
if (!RegCreateKeyEx(…)) {
    throw new Win32Exception();
}
```

- *Win32Exception* will initialise by calling *GetLastError* internally.

External Exceptions

- .NET will try to map exceptions from unmanaged code to .NET exceptions
- Many mapping for OS & hardware level errors exist – see references.
- If no mapping exists exception will be wrapped in *SEHException* or *COMException.*

Called from COM

- If your component might be called via COM, then you can set an HRESULT code on any managed exception that you throw.
- The *System.Exception* class has an HResult member that can be set in the constructor.

Called Remotely

- .NET makes it easy to access components remotely
- For a custom exception to be transferred over remoting boundaries correctly it must exist in a shared assembly that is copied to, and referenced from, your client and server applications

Called Remotely, continued

Your custom exception must:

- Override GetObjectData
- Provide custom constructor for de-serialization

See template in reference document.

Incorrect deployment may lead to:

The constructor to deserialize an object of type MyException was not found

Re-throwing

Having caught an exception you might want to re-throw it:

1. *catch (Exception e) { throw; }*
2. *catch (Exception e) { throw e; }*

The difference is that 1) will preserve the stacktrace, while 2) will reset it.

Swallowing

- You can swallow an exception:
  catch (Exception e) { }
- The *ThreadAbortException* is an undeniable exception
- If you swallow an undeniable exception it will be reintroduced by the runtime at the end of the block that swallowed it.

For completeness…

- FailFast – no stack unwinding, no messing about – call *System.Environment.FailFast()*
- Critical Finalizers – if you need absolute guarantees that your finalizer will be called, then:

1. Derive from: *CriticalFinalizerObject*
2. Annotate with: *[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]*

Exceptions & Threads

- Prior to .NET 2.0 exceptions from worker threads were lost
- From 2.0 unhandled exceptions in any thread invokes one of your applications unhandled exception handlers
- One for the CLR, one for Windows Forms
- Other hosts (ASP.NET) have other policies

2 Pass Exception Handling

- Unlike C++/Java, the CLR implements a 2 pass exception handling mechanism.
- Pass 1 – the stack is walked looking for matching exception handler
- Pass 2 – Stack is unwound and exception handler is invoked
- If no matching handler found then follow unhandled exception handling policy

Why 2 Passes?

- It makes for easier debugging
- Same model used for Windows Structured Exception Handling (SEH). In this model the SEH exception filters can try to ‘fix’ the cause of the exception and ask the OS to RESUME execution at the point the exception was thrown. This is not currently implemented in the CLR

Exception Filters

- In C# the only permitted exception filter is by the **type** of the exception thrown
- CIL allows for exception filters that execute code to determine if the exception should be handled.
- These sorts of filters are also supported in VB.NET

Exception Filters, continued

- Since execution filters are examined during the 1st Pass, the stack has not been unwound
- Any privileges that may have been granted to trusted code further down the stack are still granted during execution of the filter
- Potential for untrusted code to perform actions that it should not be allowed to.

Belts & Braces Fix

- MS reworked many of its libraries to wrap calls in a ‘redundant’ *try/catch* block:

```
try {
  try { /* Call trusted code */  }
  finally { /* Rollback trust */ }
} catch {
  throw;
}
```

.NET Partial Fix

- CLR in .NET 2.0 looks for impersonations while it walks the stack in Pass 1
- Any impersonations found are rolled back during Pass 1
- The impersonation must be directly ON the call stack – if it was performed by method call, then it won’t be found.

Conclusion

- .NET Exception handling is generally intuitive for C++/Java developers
- C++ developers cannot rely on destructors
- There is lots of functionality for interoperation with legacy Windows
- The fundamental implementation of exception handling is significantly different

Questions?