CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 4

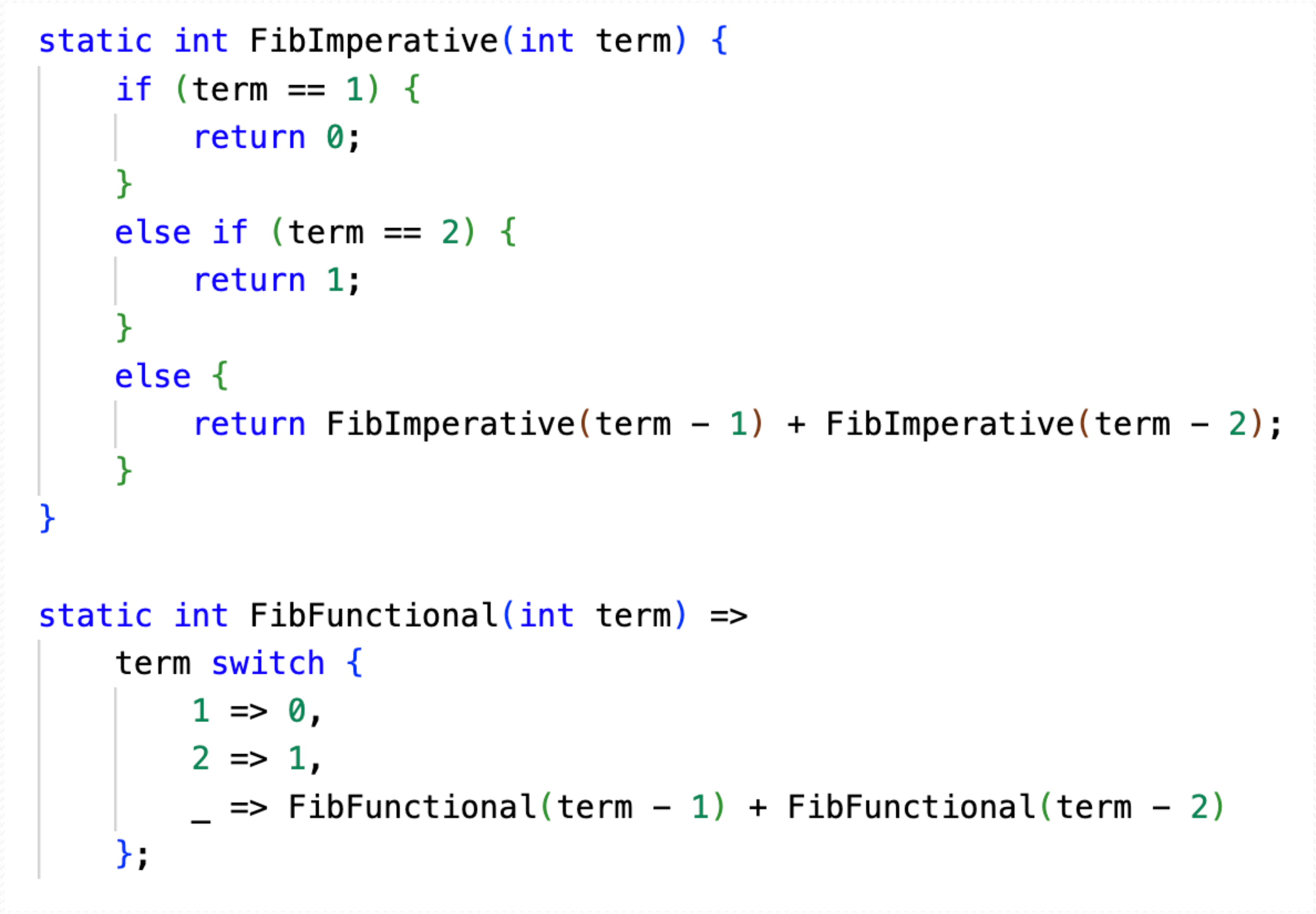
Using lambdas in function implementations

- **F#** is Microsoft's strongly typed functional-first programming language that compiles to IL to be executed by .NET
- Functional languages evolved from lambda calculus
  - Computational system based only on functions
- The code looks more like mathematical functions than steps in a recipe
- Some important attributes of functional languages are:
- **Modularity**:
  - Break up a large complex code base into smaller pieces
- **Immutability**:
  - Data values inside a function cannot change
  - A new data value can be created from an existing one – reduces bugs
- **Maintainability**:
  - Code is cleaner and clearer (for mathematically inclined programmers!)

Using lambdas in function implementations

- Since C# 6, Microsoft has worked to add features to the language to support a more functional approach
  - adding **tuples** and **pattern matching** in C# 7
  - **non-null reference types** in C# 8
  - improving pattern matching and adding records, **immutable objects,** in C# 9
- In C# 6, Microsoft added support for **expression-bodied function members**
  - In C#, lambdas are the use of the => character to indicate a return value from a function
- The **Fibonacci sequence** of numbers always starts with 0 and 1
  - Then the rest of the sequence is generated using the rule of adding together the previous two numbers 0 1 1 2 3 5 8 13 21 34 55 …

Using lambdas in function implementations



```
static int FibImperative(int term) {
    if (term == 1) {
        return 0;
    }
    else if (term == 2) {
        return 1;
    }
    else {
        return FibImperative(term - 1) + FibImperative(term - 2);
    }
}

static int FibFunctional(int term) =>
    term switch {
        1 => 0,
        2 => 1,
        _ => FibFunctional(term - 1) + FibFunctional(term - 2)
    };
```

Throwing and catching exceptions in functions

- Only catch and handle an exception if you have enough information to handle the issue
- If not, then allow the exception to pass up through the call stack to a higher level

- **Usage errors** are when a programmer misuses a function, typically by-passing invalid values as parameters
- They could be avoided by that programmer changing their code to pass valid values
- When some programmers first learn C# and .NET, they sometimes think exceptions can always be avoided because they assume all errors are usage errors
- Usage errors should all be fixed before production runtime

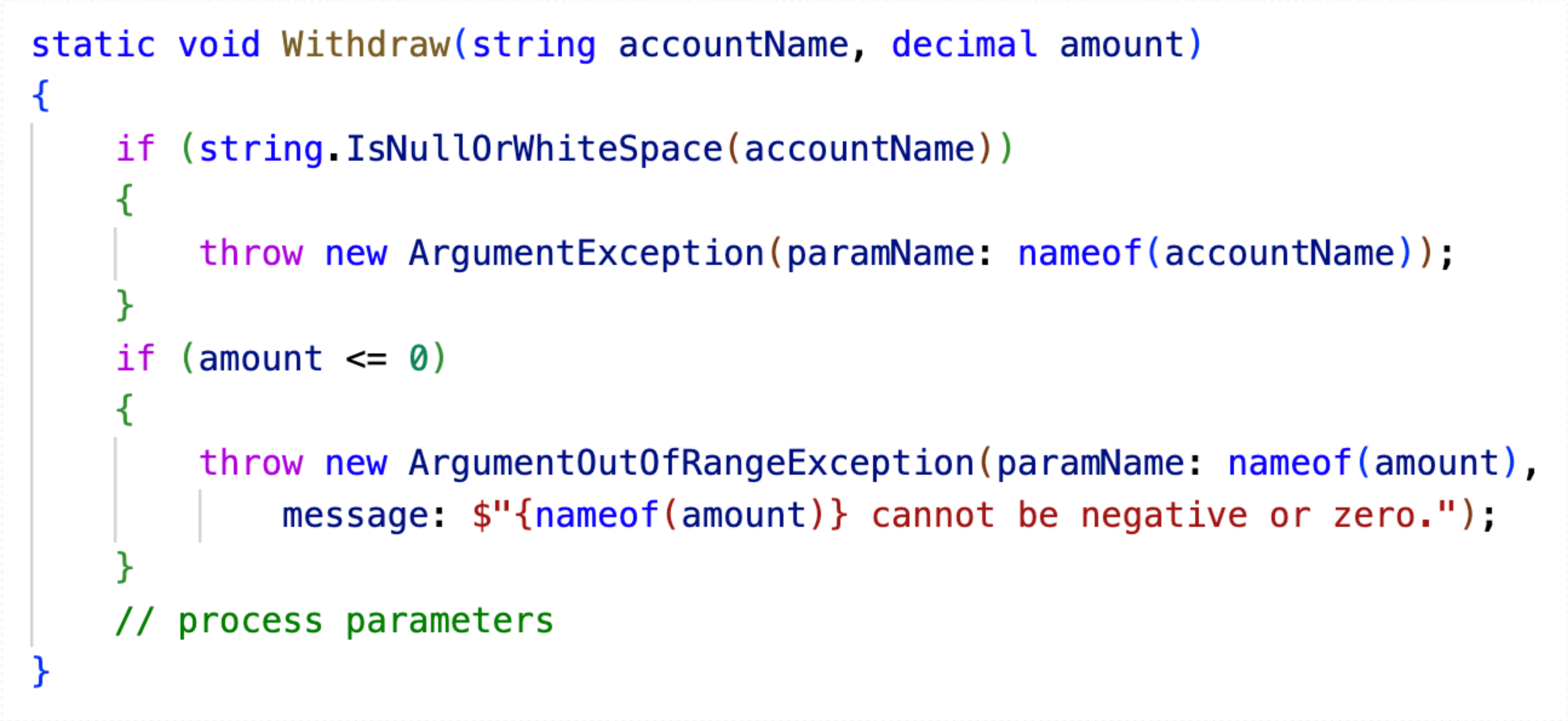
Throwing and catching exceptions in functions

- **Execution errors** are when something happens at runtime that cannot be fixed by writing "better" code
- Execution errors can be split into **program errors** and **system errors**
- If you attempt to access a network resource but the network is down, you need to be able to handle that system error by logging an exception, and possibly backing off for a time and trying again
- Some system errors, such as running out of memory, simply cannot be handled
- If you attempt to open a file that does not exist, you might be able to catch that error and handle it programmatically by creating a new file
- Program errors can be programmatically fixed by writing smart code
- System errors often cannot be fixed programmatically

Commonly thrown exceptions in functions

- Very rarely should you define new types of exceptions to indicate usage errors
  - .NET already defines many that you should use
- When defining your own functions with parameters, your code should check the parameter values and throw exceptions if they have values that will prevent your function from properly functioning
- E.g., if a parameter should not be null, throw **ArgumentNullException**
- For other problems, throw
  - **ArgumentException**
  - **NotSupportedException**
  - **InvalidOperationException**
- For any exception, include a message that describes the problem for whoever will have to read it
  - Typically, a developer audience for class libraries and functions, or end users if it is at the highest level of a GUI app

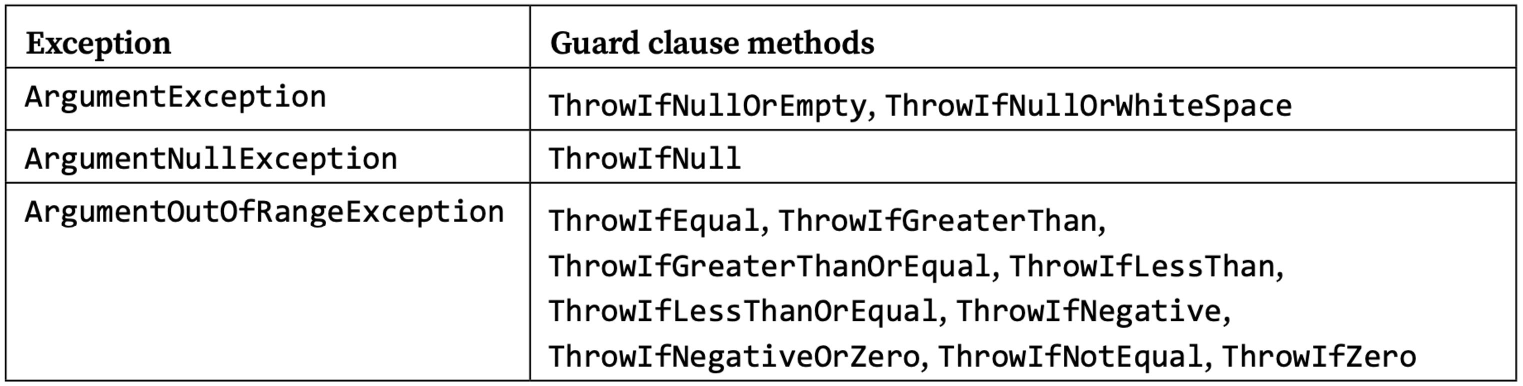
Commonly thrown exceptions in functions



```
static void Withdraw(string accountName, decimal amount)
{
    if (string.IsNullOrWhiteSpace(accountName))
    {
        throw new ArgumentException(paramName: nameof(accountName));
    }
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(paramName: nameof(amount),
            message: $"{nameof(amount)} cannot be negative or zero.");
    }
    // process parameters
}
```

Throwing exceptions using guard clauses

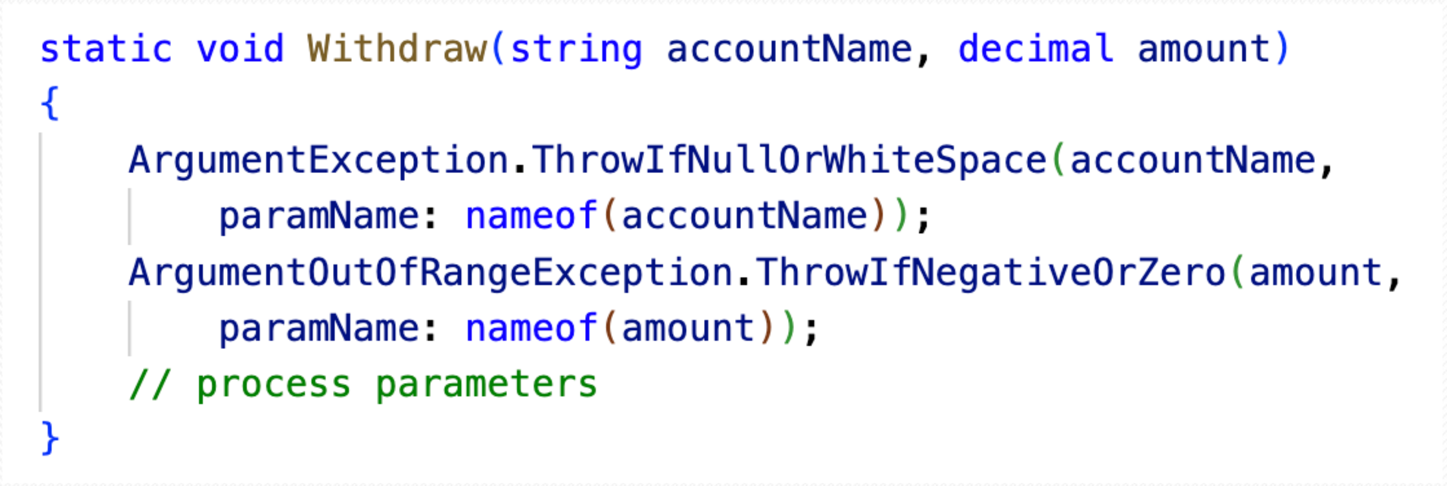
- Instead of instantiating an exception using new, you can use static methods on the exception itself
- When used in a function implementation to check argument values, they are known as **guard clauses**



Exception	**Guard clause methods**
`ArgumentException`	`ThrowIfNullOrEmpty`, `ThrowIfNullOrWhiteSpace`
`ArgumentNullException`	`ThrowIfNull`
`ArgumentOutOfRangeException`	`ThrowIfEqual`, `ThrowIfGreaterThan`, `ThrowIfGreaterThanOrEqual`, `ThrowIfLessThan`, `ThrowIfLessThanOrEqual`, `ThrowIfNegative`, `ThrowIfNegativeOrZero`, `ThrowIfNotEqual`, `ThrowIfZero`

Throwing exceptions using guard clauses

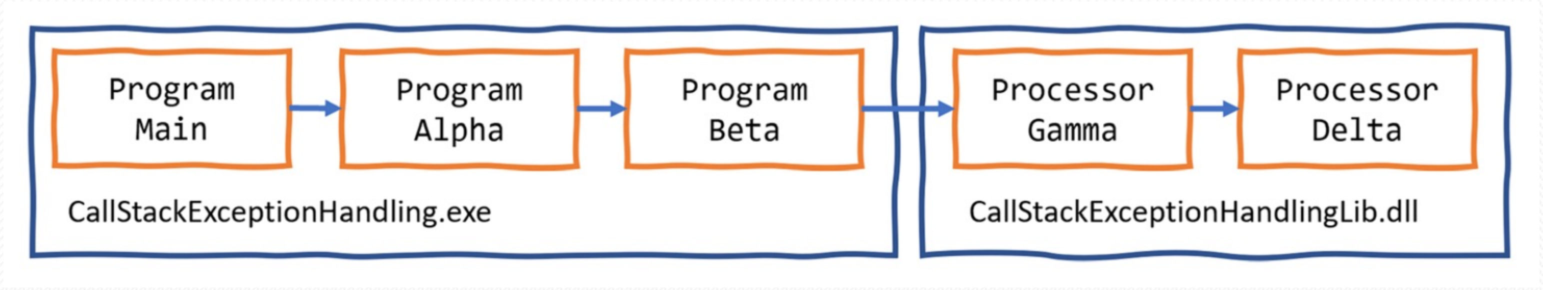
- Instead of writing an if statement and then throwing a new exception, we can simplify the previous example



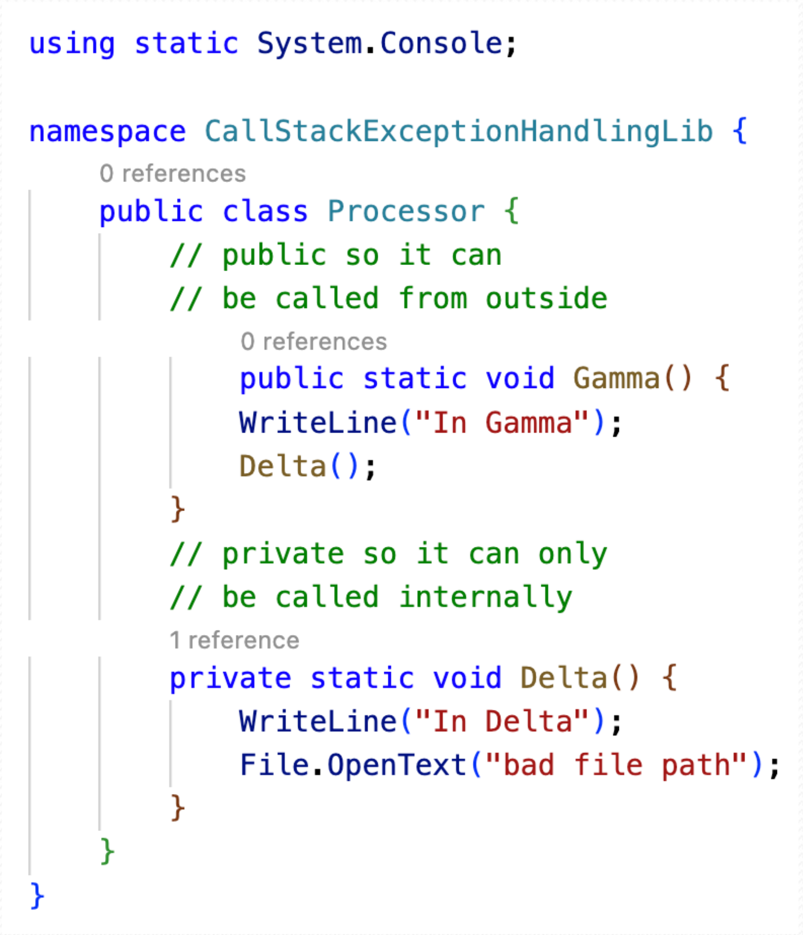
```
static void Withdraw(string accountName, decimal amount)
{
    ArgumentException.ThrowIfNullOrWhiteSpace(accountName,
        paramName: nameof(accountName));
    ArgumentOutOfRangeException.ThrowIfNegativeOrZero(amount,
        paramName: nameof(amount));
    // process parameters
}
```

Understanding the call stack

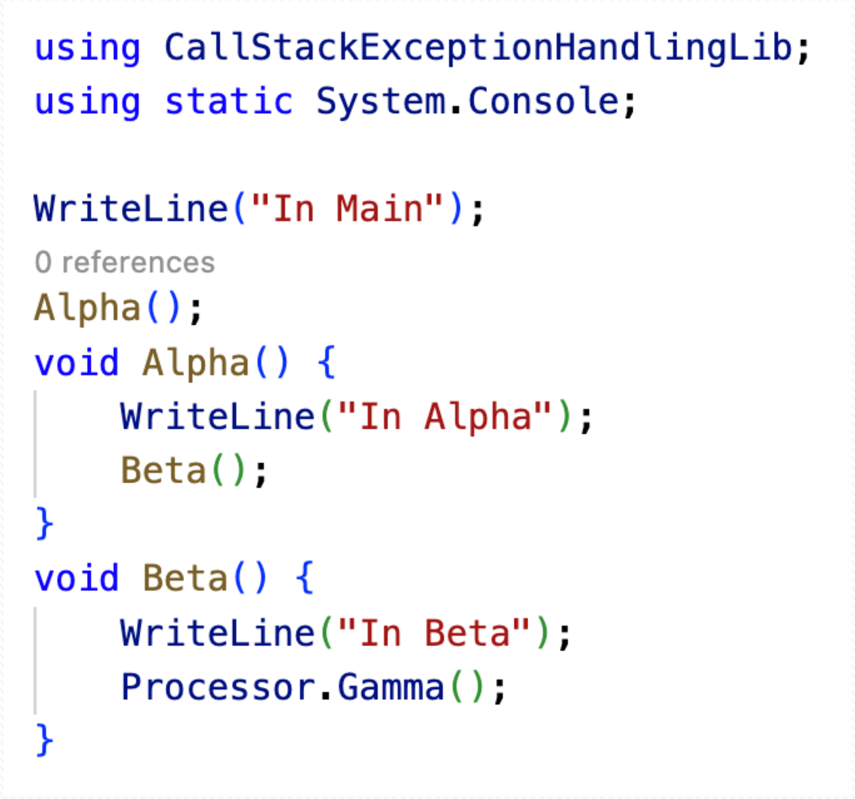
- The entry point for a .NET console application is the Main (if you have explicitly defined this class) or <Main>$ (if it was created for you by the top-level program feature) of the Program class
- The Main method will call other methods, that call other methods, and so on, and these methods could be in the current project or in referenced projects



Understanding the call stack

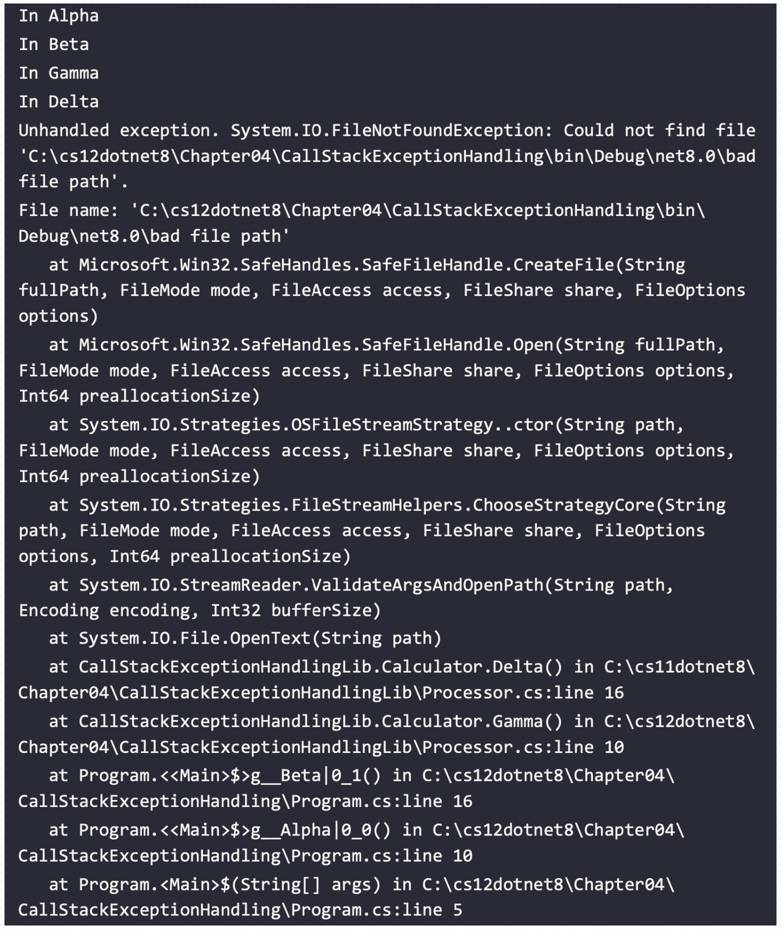


```
using static System.Console;

namespace CallStackExceptionHandlingLib {
    0 references
    public class Processor {
        // public so it can
        // be called from outside
            0 references
            public static void Gamma() {
            WriteLine("In Gamma");
            Delta();
        }
        // private so it can only
        // be called internally
        1 reference
        private static void Delta() {
            WriteLine("In Delta");
            File.OpenText("bad file path");
        }
    }
}
```



```
using CallStackExceptionHandlingLib;
using static System.Console;

WriteLine("In Main");
0 references
Alpha();
void Alpha() {
    WriteLine("In Alpha");
    Beta();
}
void Beta() {
    WriteLine("In Beta");
    Processor.Gamma();
}
```

Understanding the call stack



```
In Alpha
In Beta
In Gamma
In Delta
Unhandled exception. System.IO.FileNotFoundException: Could not find file
'C:\cs12dotnet8\Chapter04\CallStackExceptionHandling\bin\Debug\net8.0\bad
file path'.
File name: 'C:\cs12dotnet8\Chapter04\CallStackExceptionHandling\bin\
Debug\net8.0\bad file path'
   at Microsoft.Win32.SafeHandles.SafeFileHandle.CreateFile(String
fullPath, FileMode mode, FileAccess access, FileShare share, FileOptions
options)
   at Microsoft.Win32.SafeHandles.SafeFileHandle.Open(String fullPath,
FileMode mode, FileAccess access, FileShare share, FileOptions options,
Int64 preallocationSize)
   at System.IO.Strategies.OSFileStreamStrategy..ctor(String path,
FileMode mode, FileAccess access, FileShare share, FileOptions options,
Int64 preallocationSize)
   at System.IO.Strategies.FileStreamHelpers.ChooseStrategyCore(String
path, FileMode mode, FileAccess access, FileShare share, FileOptions
options, Int64 preallocationSize)
   at System.IO.StreamReader.ValidateArgsAndOpenPath(String path,
Encoding encoding, Int32 bufferSize)
   at System.IO.File.OpenText(String path)
   at CallStackExceptionHandlingLib.Calculator.Delta() in C:\cs11dotnet8\
Chapter04\CallStackExceptionHandlingLib\Processor.cs:line 16
   at CallStackExceptionHandlingLib.Calculator.Gamma() in C:\cs12dotnet8\
Chapter04\CallStackExceptionHandlingLib\Processor.cs:line 10
   at Program.<<Main>$>g__Beta|0_1() in C:\cs12dotnet8\Chapter04\
CallStackExceptionHandling\Program.cs:line 16
   at Program.<<Main>$>g__Alpha|0_0() in C:\cs12dotnet8\Chapter04\
CallStackExceptionHandling\Program.cs:line 10
   at Program.<Main>$(String[] args) in C:\cs12dotnet8\Chapter04\
CallStackExceptionHandling\Program.cs:line 5
```

Understanding the call stack

- The call stack is upside-down. Starting from the bottom, you see:
  - The first call is to the <Main>$ entry point function in the auto-generated Program class
    - This is where arguments are passed in as a String array
  - The second call is to the <<Main>$>g__Alpha|0_0 function
    - C# compiler renames it from Alpha when it adds it as a local function
  - The third call is to the Beta function
  - The fourth call is to the Gamma function
  - The fifth call is to the Delta function
    - This function attempts to open a file by passing a bad file path
    - This causes an exception to be thrown
    - Any function with a try-catch statement could catch this exception
    - If it does not, the exception is automatically passed up the call stack until it reaches the top, where .NET outputs the exception along with the details of this call stack

Rethrowing exceptions

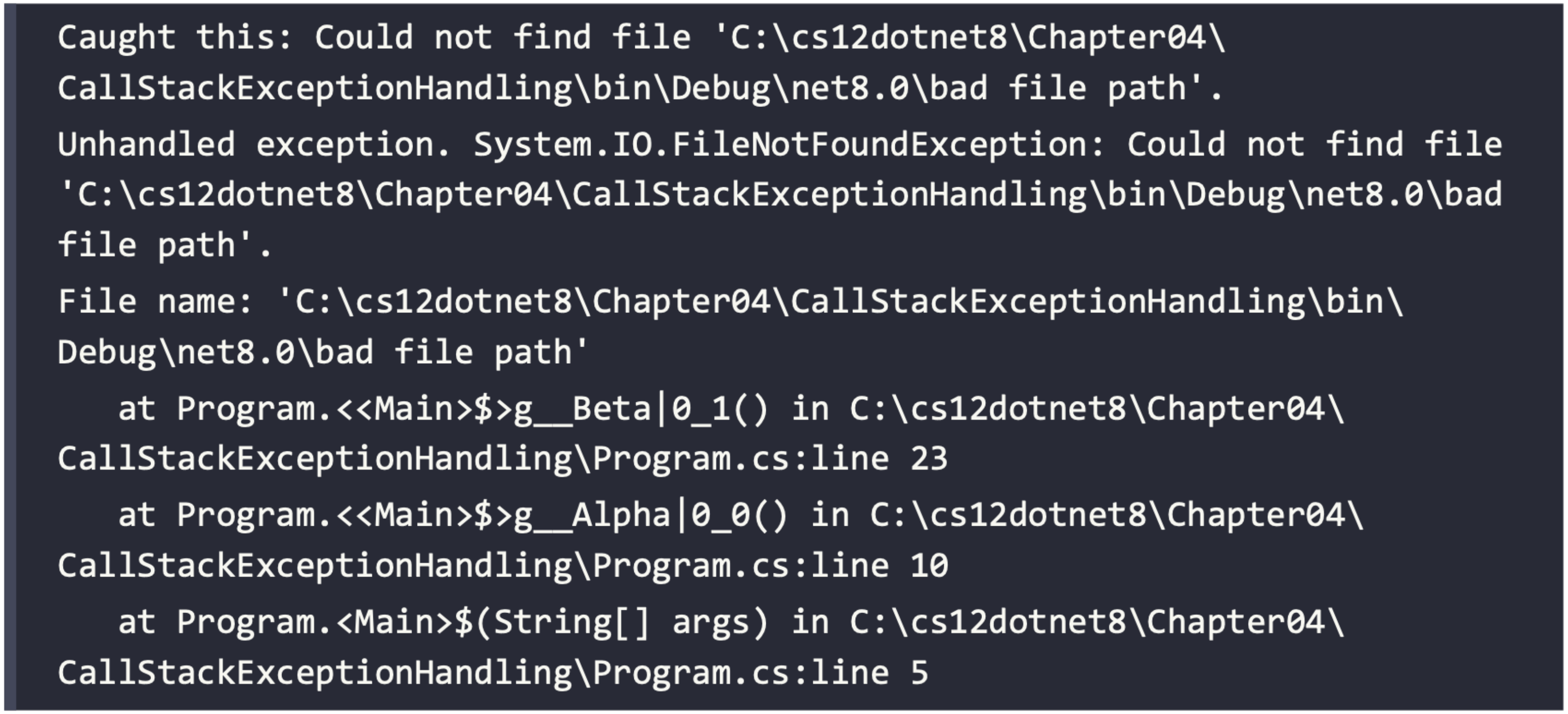
- Sometimes you want to catch an exception, log it, and then rethrow it
- There are three ways to rethrow an exception inside a catch block:
  1. To throw the caught exception with its original call stack, call throw
  2. To throw the caught exception as if it was thrown at the current level in the call stack, call throw with the caught exception, for example, throw ex
     - This is usually poor practice because you have lost some potentially useful information for debugging
  3. To wrap the caught exception in another exception that can include more information in a message that might help the caller understand the problem
     - throw a new exception and pass the caught exception as the **innerException** parameter

Rethrowing exceptions

```
try {
    Gamma();
}
catch (IOException ex) {
    LogException(ex);
    // Throw the caught exception
    // as if it happened here
    // this will lose the
    // original call stack
    throw ex;
    // Rethrow the caught exception and
    // retain its original call stack
    throw;
    // Throw a new exception with the
    // caught exception nested within it
    throw new InvalidOperationException(
        message: "Calculation had invalid values. See inner exception for why.",
        innerException: ex);
}
```

```
void Beta() {
    WriteLine("In Beta");
    try {
        Processor.Gamma();
    }
    catch (Exception ex) {
        WriteLine($"Caught this: {ex.Message}");
        throw ex;
    }
}
```

Rethrowing exceptions



```
Caught this: Could not find file 'C:\cs12dotnet8\Chapter04\
CallStackExceptionHandling\bin\Debug\net8.0\bad file path'.
Unhandled exception. System.IO.FileNotFoundException: Could not find file
'C:\cs12dotnet8\Chapter04\CallStackExceptionHandling\bin\Debug\net8.0\bad
file path'.
File name: 'C:\cs12dotnet8\Chapter04\CallStackExceptionHandling\bin\
Debug\net8.0\bad file path'
   at Program.<<Main>$>g__Beta|0_1() in C:\cs12dotnet8\Chapter04\
CallStackExceptionHandling\Program.cs:line 23
   at Program.<<Main>$>g__Alpha|0_0() in C:\cs12dotnet8\Chapter04\
CallStackExceptionHandling\Program.cs:line 10
   at Program.<Main>$(String[] args) in C:\cs12dotnet8\Chapter04\
CallStackExceptionHandling\Program.cs:line 5
```