

# .Net Training

Intermediate Basics of .Net (C#)  
Part III



# Agenda

---

- Debugging and Tracing
- Exceptions and Exception Handling



# Agenda

---

- Debugging and Tracing

# Debugging and Tracing

---

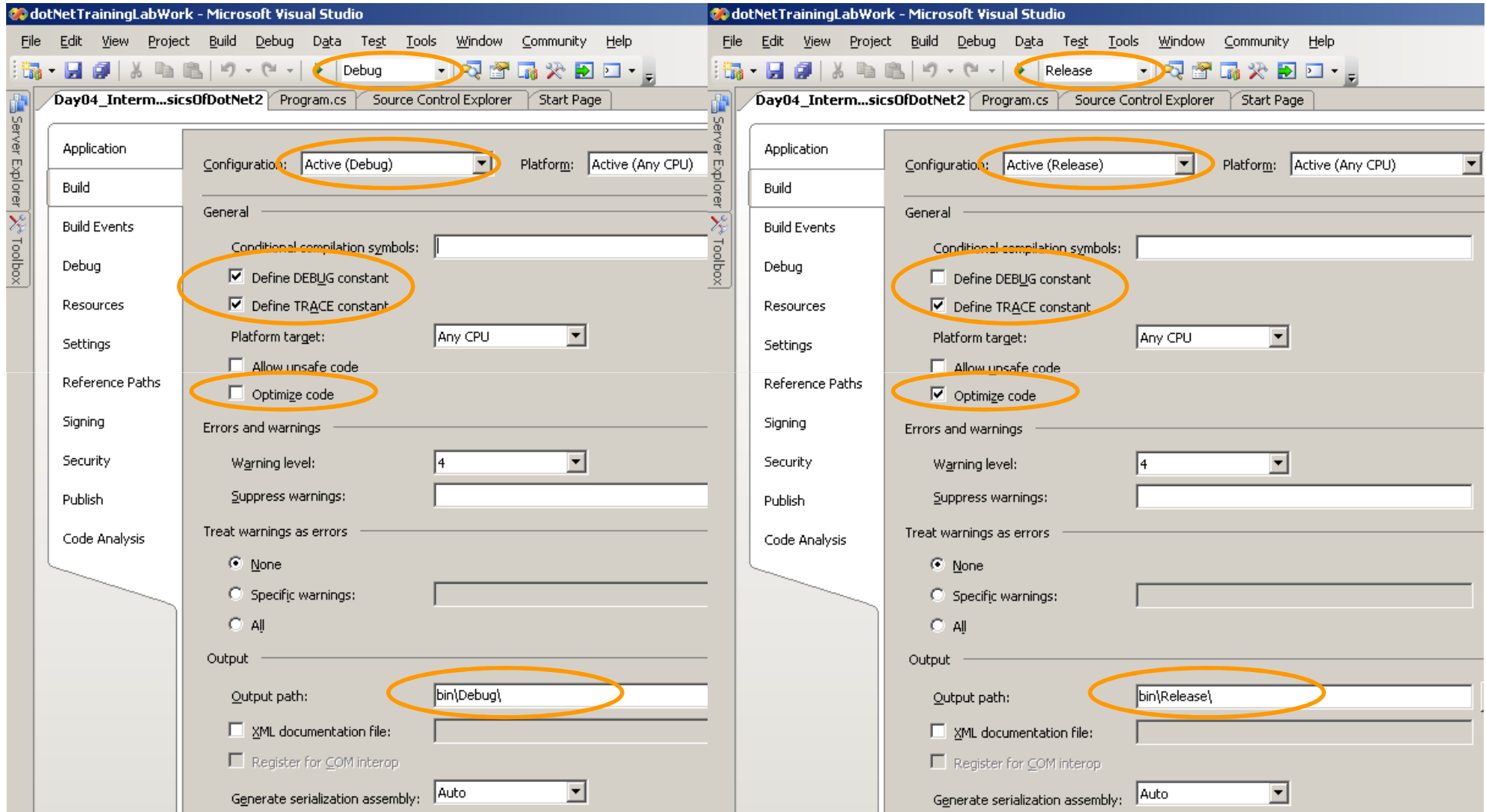
- When things don't go as expected you need a window into the code to see what was going on when things failed
- .Net provides a number of features to facilitate the act of debugging
- Besides the Visual Studio's debugging capabilities there are code debugging tools you can use
- The System.Diagnostics namespace contains two very special classes for this purpose

# Debugging and Tracing

---

- System.Diagnostics
  - Debug
  - Trace
- Both provide the same capabilities and features
- But Debug is present only when you compile the code with debugging symbols (/DEBUG)
  - When you compile in Release mode all Debug statements disappear - so they do not increase the size or reduce the speed of your release code
- Trace statements persist in both Debug and Release modes

# Build Configurations



The image displays two side-by-side screenshots of the Microsoft Visual Studio Build Configuration window for a project named 'Day04\_Interm...sicsOfDotNet2'. The left screenshot shows the 'Debug' configuration, and the right screenshot shows the 'Release' configuration. Key differences between the two configurations are highlighted with orange circles:

- Configuration:** 'Active (Debug)' vs 'Active (Release)'
- General:**
  - Conditional compilation symbols:** Both are empty.
  - Define DEBUG constant:** Checked in Debug, unchecked in Release.
  - Define TRACE constant:** Checked in both.
  - Platform target:** 'Any CPU' in both.
  - Allow unsafe code:** Unchecked in both.
  - Optimize code:** Unchecked in Debug, checked in Release.
- Output path:** 'bin\Debug\' in Debug, 'bin\Release\' in Release.

# Debugging and Tracing

---

- If you create your project using the Visual Studio wizards, the TRACE symbol is defined by default in both Release and Debug configurations. The DEBUG symbol is defined by default only in the Debug build
- Otherwise, for **Trace** methods to work, your program must have one of the following at the top of the source file:
  - #define TRACE in Visual C# and C++
- Or your program must be built with the TRACE option:
  - /d:TRACE in Visual C# and C++
- If you need to use the Debug methods in a C# Release build, you must define the DEBUG symbol in your Release configuration

# Debugging and Tracing

---

- Debug and Trace classes provide basic methods for tracking what goes on inside your code
  - Assert – tests a condition
  - Write() – writes a string of text
  - WriteLine() – writes a line of text terminated with CrLf
  - WriteIf() – writes a string of text if a condition is met
  - WriteLineIf() – write a line of text if a condition is met

# Assert

---

- An assertion, or **Assert** statement, tests a condition, which you specify as an argument to the **Assert** statement
- If the condition evaluates to true, no action occurs
- If the condition evaluates to false, the assertion fails
- If you are running with a debug build, your program enters break mode

# Assert

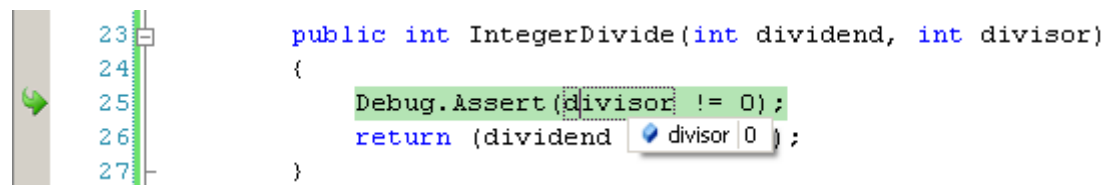
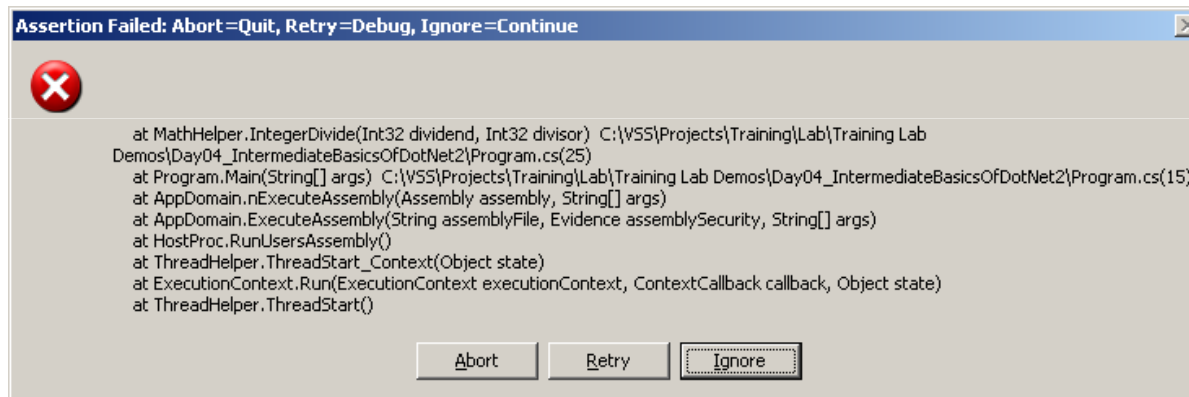
---

```
int IntegerDivide ( int dividend , int divisor )  
{ Debug.Assert ( divisor != 0 );  
  return ( dividend / divisor ); }
```

- When you run this code under the debugger, the assertion statement is evaluated, but in the Release version, the comparison is not made, so there is no additional overhead

# Assert

- When the Assert fails, the debugger will enter Break mode, alerting the developer that an error condition is about to occur



# Writers

---

- Write, WriteLine, Writelf and WriteLinelf
- All write a string of text to the Debug/Trace Listeners
- Write and WriteLine are standard writers – they write to the Listeners whatever string you pass in irregardless
- Conditional writers (Writelf and WriteLinelf) take a boolean condition as a parameter to determine whether to write or not



# Writers

```
public int HoursWorked {
    get {
        int hours = 0;
        int i = _times.GetLowerBound(0); //define counter
        TimeSpan diff = new TimeSpan(0);
        DateTime previousTime = DateTime.MinValue; //this will mean we haven't set it yet
        Debug.WriteLine(string.Format("Time Array from {0} to {1}", _times.GetLowerBound(0), _times.GetUpperBound(0)));
        while (i <= _times.GetUpperBound(0))
        {
            DateTime time = _times[i];
            Debug.WriteLine(string.Format("Array Item #{0}: {1}", i, time.ToShortTimeString()));
            if (i % 2 == 0) //modulo operator - remainder of division by 2
            { //even -> punched in
                previousTime = time; //cache for next time round
                Debug.WriteLine("Entered Even branch");
            }
            else
            { //odd -> punched out
                TimeSpan tempDiff = time.Subtract(previousTime);
                diff += tempDiff;
                Debug.WriteLine("Entered Odd branch");
            }
            Debug.WriteLineIf((i % 2 == 0), "Entered Even branch");
            i++; //increment counter
            Debug.WriteLine("Counter incremented");
        }
        //calc hours
        hours = (diff.Minutes > 30) ? diff.Hours + 1 : hours = diff.Hours;
        Debug.WriteLine(string.Format("Hours calculated: {0} (minutes used: {1})", hours, diff.Minutes));
        return hours;
    }
}
```

# Listeners

---

- OK – so where is all this output going to?!?!?



# Listeners

---

- Diagnostics classes are extensible
- Out of the box there are a few available Listeners. All inherit from the abstract `TraceListener` class:
  - `DefaultTraceListener` – Win32 `OutputDebugString` function and to the [Debugger.Log](#) method (to you that mean your Visual Studio Output Console)
  - `ConsoleTraceListener` – either the standard output or the standard error stream
  - `EventLogTraceListener` – an `EventLog`
  - `TextWriterTraceListener` – a `TextWriter` or `Stream`
- You can also create your own by inheriting from `TraceListener` (for example a `DatabaseTraceListener`)

# TextWriterTraceListener

---

- Directs tracing or debugging output to a TextWriter or to a Stream, such as FileStream
- To add a trace listener, edit the configuration file that corresponds to the name of your application
- Within this file, you can add a listener, set its type and set its parameter, remove a listener, or clear all the listeners previously set by the application

# TextWriterTraceListener

---

- The TextWriterTraceListener can be attached either via code or via a configuration file (App.config or Web.config)

```
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="4">
      <listeners>
        <add name="myListener"
          type="System.Diagnostics.TextWriterTraceListener"
          initializeData="Trace.log" />
        <remove name="Default" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

# TextWriterTraceListener

---

- The TextWriterTraceListener can also be attached via code:

```
using System.IO; using System.Diagnostics;
public class TestTracer {
public static int Main(string[] args) {
    // Create a file for output named TestFile.txt.
    Stream myFile = File.Create("Trace.log");
    /* Create a new text writer using the output stream, and add it to
    * the trace listeners. */
    TextWriterTraceListener myTextListener = new TextWriterTraceListener(myFile);
    Trace.Listeners.Add(myTextListener);
    // Write output to the file.
    Trace.Write("Test output ");
    // Flush the output.
    Trace.Flush();
    return 0;
}}
```



# Agenda

---

- Exceptions and Exception Handling

# Exceptions

---

- The C# language's exception handling features provide a way to deal with any unexpected or exceptional situations that arise while a program is running
- Exception handling uses the **try**, **catch**, and **finally** keywords:
  - to attempt actions that may not succeed
  - to handle failures
  - to clean up resources afterwards
- Exceptions can be generated by the common language runtime (CLR), by third-party libraries, or by the application code using the **throw** keyword

# Exceptions

---

- In this example, a method tests for a division by zero, and catches the error
- Without the exception handling, this program would terminate with a `DivideByZeroException` as an unhandled error

```
int SafeDivision(int x, int y)
{
    try
    {
        return (x / y);
    }
    catch (System.DivideByZeroException dbz)
    {
        System.Console.WriteLine("Division by zero attempted!");
        return 0;
    }
}
```

# Exceptions

---

- When your application encounters an exceptional circumstance, such as a division by zero or low memory warning, an exception is generated
- Once an exception occurs, the flow of control immediately jumps to an associated exception handler, if one is present
- If no exception handler for a given exception is present, the program stops executing with an error message
- Actions that may result in an exception are executed with the **try** keyword
- An exception handler is a block of code that is executed when an exception occurs. In C#, the **catch** keyword is used to define an exception handler
- Exceptions can be explicitly generated by a program using the **throw** keyword
- Exception objects contain detailed information about the error, including the state of the call stack and a text description of the error
- Code in a **finally** block is executed even if an exception is thrown, thus allowing a program to release resources

# Pop-Quiz

---

```
int SafeDivision(int x, int y)
{
    try
    {
        return (x / y);
    }
    catch (System.DivideByZeroException dbz)
    {
        System.Console.WriteLine("Division by zero attempted!");
        return 0;
    }
    finally { System.Console.WriteLine("Thank you for using SafeDivision!");}
}
```

- Will the code in the finally block be called if:
  - x=5 and y=0?
  - x=5 and y=2?

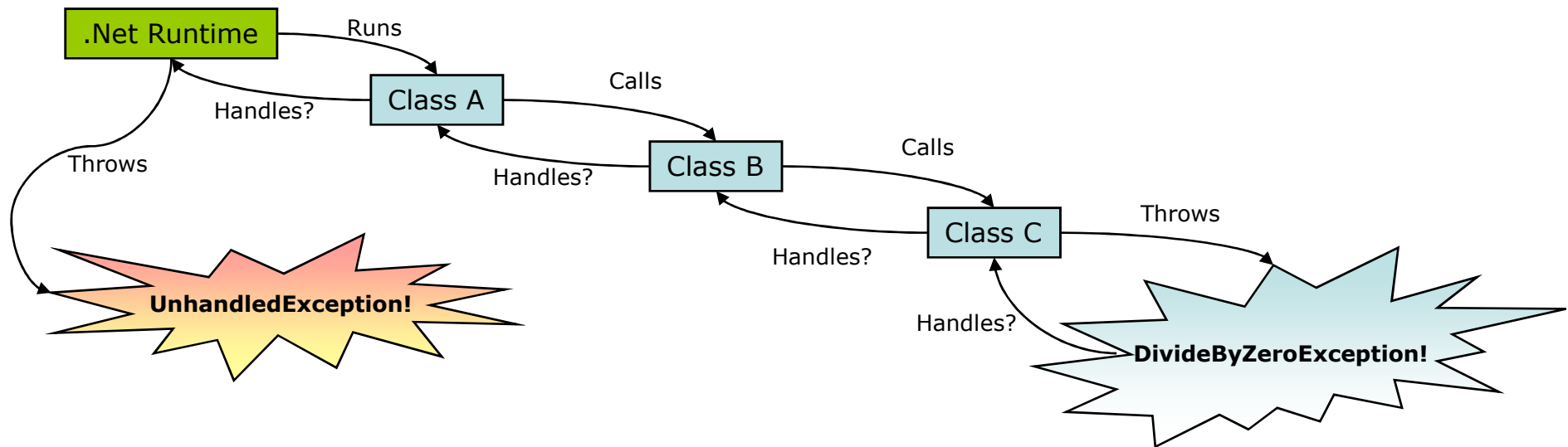
# Exceptions

---

- An exception is any error condition or unexpected behavior encountered by an executing program
- Exceptions can be raised because of a fault in your code or in code you call (such as a shared library), unavailable operating system resources, unexpected conditions the common language runtime encounters (such as code that cannot be verified), and so on
- Your application can recover from some of these conditions, but not others
- While you can recover from most application exceptions, you cannot recover from most runtime exceptions

# Exceptions

- In the .NET Framework, an exception is an object that inherits from the Exception class
- An exception is thrown from an area of code where a problem has occurred
- The exception is passed up the stack until the application handles it or the program terminates



# How to handle Exceptions

---

- You need to determine what you want to happen when an exception is caught in your code:
  - Log the exception to file
  - Log the exception to database
  - Log the exception to EventLog
  - Terminate the application
  - Let the exception pass and be caught by callers
  - Re-wrap the exception with a custom exception type
  - Ignore the Exception
  - HANDLE THE EXCEPTION
  - Etc.

# Exceptions Best Practices

---

- As a general rule:
- Exceptions are expensive
  - They are expensive to throw
  - They are expensive to catch
- If possible avoid Exceptions

# Exceptions Best Practices

---

- A well-designed set of error handling code blocks can make a program more robust and less prone to crashing because the application handles such errors
- Know when to set up a try/catch block.
  - For example, you can programmatically check for a condition that is likely to occur without using exception handling
  - In other situations, using exception handling to catch an error condition is appropriate
- For example, you can use an **if** statement to check whether a connection is closed. You can use this method instead of throwing an exception if the connection is not closed

# Exceptions Best Practices

---

- The method you choose depends on how often you expect the event to occur
- If the event is truly exceptional and is an error (such as an unexpected end-of-file), using exception handling is better because less code is executed in the normal case
- If the event happens routinely, using the programmatic method to check for errors is better. In this case, if an exception occurs, the exception will take longer to handle

# Exceptions Best Practices

---

- Use try/finally blocks around code that can potentially generate an exception and centralize your catch statements in one location. In this way, the try statement generates the exception, the finally statement closes or deallocates resources, and the catch statement handles the exception from a central location
- End exception class names with the word "Exception". For example:  
`EmployeeListNotFoundException`

# Exceptions Best Practices

---

- Use grammatically correct error messages, including ending punctuation. Each sentence in a description string of an exception should end in a period.
- Provide **Exception** properties for programmatic access. Include extra information in an exception (in addition to the description string) only when there is a programmatic scenario where the additional information is useful.
- Return null for extremely common error cases. For example, **File.Open** returns null if the file is not found, but throws an exception if the file is locked.

# Exceptions Best Practices

---

- Design classes so that an exception is never thrown in normal use. For example, a **FileStream** class exposes another way of determining whether the end of the file has been reached. This avoids the exception that is thrown if you read past the end of the file:

```
class FileRead {  
    void Open() {  
        FileStream stream = File.Open("myfile.txt", FileMode.Open);  
        byte b;  
        // ReadByte returns -1 at EOF.  
        while ((b == stream.ReadByte()) != true) {  
            // Do something.  
        }  
    }  
}
```

# Exceptions Best Practices

---

- Throw an **InvalidOperationException** if a property set or method call is not appropriate given the object's current state
- Throw an `ArgumentException` or a class derived from **ArgumentException** if invalid parameters are passed
- The stack trace begins at the statement where the exception is thrown and ends at the catch statement that catches the exception. Be aware of this fact when deciding where to place a throw statement

# Exceptions Best Practices

---

- In most cases, use the predefined exceptions types. Define new exception types only for programmatic scenarios. Introduce a new exception class to enable a programmer to take a different action in code based on the exception class
- Throw exceptions instead of returning an error code or HRESULT
- **Clean up intermediate results when throwing an exception. Callers should be able assume that there are no side effects when an exception is thrown from a method**

# Throw Explicit Exceptions

---

- How to explicitly throw exceptions:
  - throw new FileNotFoundException("File log.txt not found!");

# Exception Hierarchy

---

- There are two types of exceptions: exceptions generated by an executing program, and exceptions generated by the common language runtime
- In addition, there is a hierarchy of exceptions that can be thrown by either an application or the runtime

# Exception Hierarchy

---

- Exception is the base class for exceptions. Several exception classes inherit directly from **Exception**, including `ApplicationException` and `SystemException`. These two classes form the basis for almost all runtime exceptions
- Most exceptions that derive directly from **Exception** add no functionality to the **Exception** class. For example, the `InvalidCastException` Class hierarchy is as follows:
  - Object -> **Exception** -> **SystemException** -> `InvalidCastException`

# Exception Hierarchy

---

- The runtime throws the appropriate derived class of **SystemException** when errors occur
- These errors result from failed runtime checks (such as array out-of-bound errors), and can occur during the execution of any method
- If you are designing an application that creates new exceptions, you should derive those exceptions from the Exception class
- It is not recommended that you catch a **SystemException**, nor is it good programming practice to throw a **SystemException** in your application

# Exception Hierarchy

---

- The most severe exceptions — those thrown by the runtime or in nonrecoverable conditions — include `ExecutionEngineException`, `StackOverflowException`, and `OutOfMemoryException`
- Interoperation exceptions derive from **`SystemException`** and are further extended by `ExternalException`
  - For example, `COMException` is the exception thrown during COM interop operations and derives from **`ExternalException`**
  - `Win32Exception` and `SEHException` also derive from **`ExternalException`**

# Exceptions Best Practices

---

- **Always order exceptions in catch blocks from the most specific to the least specific. This technique handles the specific exception before it is passed to a more general catch block**

```
try
{ ... }
catch(FileNotFoundException fex)
{...}
catch(ApplicationException aex)
{..}
catch(Exception ex)
{..}
```

# Homework

---

- Add exception handling and Debug/Trace code to Day 3's time calculation code