



# COMPSCI 280 S1 2011 Enterprise Software Development

Programming Fundamentals  
Exceptions



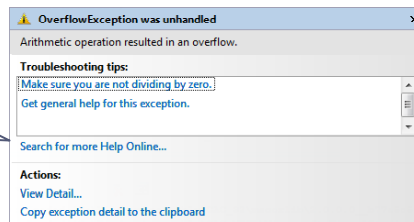
## Agenda & Reading

- ▶ **Agenda:**
  - ▶ Exception
  - ▶ Try-Catch Block
  - ▶ Handling Multiple Catch Clauses
  - ▶ Nested Try-Catch Blocks
  - ▶ Exception Propagation
  - ▶ Finally
  - ▶ Explicitly Throw Exceptions
  - ▶ checked and unchecked
- ▶ **Recommended Reading:**
  - ▶ C# for Java Programmers
    - ▶ Chapter 4
  - ▶ Microsoft Visual C# 2008 Step by Step
    - ▶ Chapter 6
  - ▶ Exceptions and Exception Handling (C# Programming Guide)
    - ▶ <http://msdn2.microsoft.com/en-us/library/ms173160.aspx>
- ▶ **Hands-On Lab:**
  - ▶ Lecture 11 Lab



## 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, unavailable operating system resources, and so on.
- ▶ You can use Structured Error Handlers to recognize run-time errors as they occur in a program, suppress unwanted error messages, and adjust program conditions so that your application can regain control and run again.

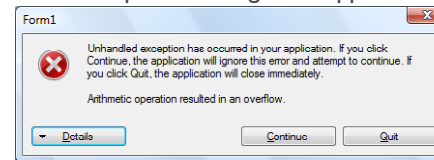


In Debug mode

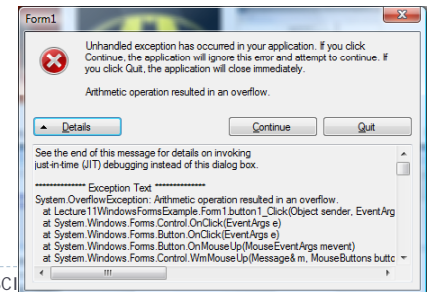


## Exceptions in Windows Forms Application

- ▶ Run the application (double-click the exe file)
- ▶ An Exception message box appears:



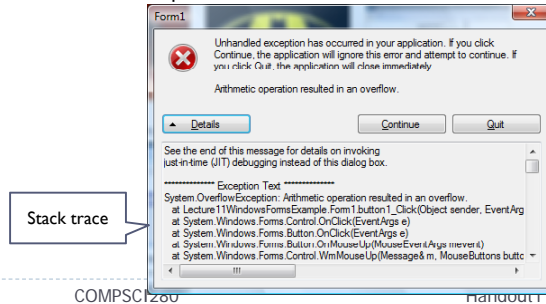
- ▶ Click the Details button to check the details about the error
- ▶ Click the Quit button to exit the application, or
- ▶ Click the Continue button so that your application can regain control and run again.





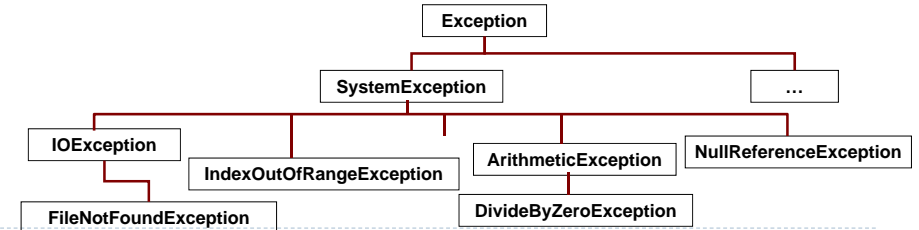
# Exceptions

- ▶ In the .NET Framework, an exception is an object that inherits from the Exception Class.
- ▶ Properties:
  - ▶ StackTrace
    - It contains a stack trace that can be used to determine where an error occurred. (includes the source file name and program line number)
  - ▶ Message
    - It provides details about the cause of an exception.



# Exception Hierarchy

Exception Type	Description
Exception	Base Class
SystemException	Base class for all runtime-generated errors.
IOException	when an I/O error occurs.
FileNotFoundException	when an attempt to access a file that does not exist on disk fails.
IndexOutOfRangeException	when an array is indexed improperly.
ArithmeticException	Errors in an arithmetic, casting, or conversion operation
DivideByZeroException	when an attempt to divide an integral or decimal value by zero
NullReferenceException	when a null object is referenced.

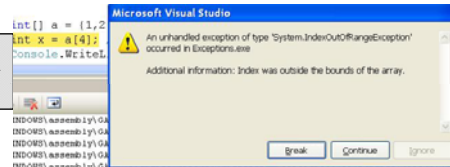


# No Exception Handling

- ▶ 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.
- ▶ If an exception occurs, the exception is propagated back to the calling method, or the previous method.
- ▶ If it also has no exception handler, the exception is propagated back to that method's caller, and so on.
- ▶ If it fails to find a handler for the exception, an error message is displayed and the application is terminated.

### Example:

```
int[] a = {1,2,3};
int x = a[4]; //generates run-time error
Console.WriteLine(x);
```

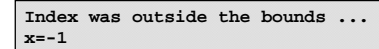


- ▶ By placing exception handling code in your application, you can handle most of the errors users may encounter and enable the application to continue running.



# Try-catch block

- ▶ Try block
  - ▶ Place the code that might cause the exception in a try block
  - ▶ When error happens, the .NET system ignores the rest of the code in the try block and jumps to the catch block
- ▶ Catch block
  - ▶ Specify the exception that you wish to catch or catch the general exception. (Since all exceptions are subclasses of the Exception class, you can catch all exceptions using this way. In this case, exceptions of all types will be handled in the same way.)
  - ▶ Execute the code if the exception is thrown
  - ▶ Skip the code if no exception
- ▶ Statements after the catch block
  - ▶ Execute if either the exception is not thrown or if it is thrown



```
try {
    int[] a = { 1, 2, 3 };
    x = a[4];
    Console.WriteLine("This will not be printed");
} catch (Exception ex){
    x = -1;
    Console.WriteLine(ex.Message);
}
Console.WriteLine("x=" + x + "
```



## Handling Multiple Catch Clauses

- ▶ The catch block is a series of statements beginning with the keyword `catch`, followed by an exception type and an action to be taken.
- ▶ Each catch block is an exception handler and handles the type of exception indicated by its argument
- ▶ The runtime system invokes the exception handler when the handler is the first one matches the type of the exception thrown.
- ▶ It executes the statement inside the matched catch block, the other catch blocks are bypassed and continues after the try-catch block.

```
try {
    String s = "a";
    int x = Convert.ToInt32(s);
} catch (FormatException ex) {
    Console.WriteLine(ex.Message);
} catch (Exception ex){
    Console.WriteLine("General Exception");
}
Console.WriteLine("Finished");
```

Input string was not in a correct format.  
Finished

9 COMPSCI280 Handout11



## Handling Multiple Catch (con't)

- ▶ Exceptions are arranged in an inheritance hierarchy.
  - ▶ A catch specifying an Exception near the top of the hierarchy (a very general Exception) are match any Exception in the subtree.
  - ▶ Note: Exception subclass (specific type of exception) must come before any of their superclass (general Exception) Order of catch Clause. Otherwise, the compiler might issue an error

```
try {
    String s = "a";
    int x = Convert.ToInt32(s);
} catch (Exception ex) {
    Console.WriteLine(ex.Message);
} catch (FormatException ex){
    Console.WriteLine("General Exception");
}
Console.WriteLine("Finished");
```

Compile error

General Exception  
Finished

```
try {
    String s = "a";
    int x = Convert.ToInt32(s);
} catch (ArithmeticException ex){
    Console.WriteLine(ex.Message);
} catch (Exception ex){
    Console.WriteLine("General Exception");
}
Console.WriteLine("Finished");
```

10 COMPSCI280 Handout11



## Nested Try-Catch

- ▶ You can use nested try-catch blocks in your error handlers.
  - ▶ If an inner try statement does not have a matching catch statement for a particular exception, the next try statement's catch handlers are inspected for a match.
  - ▶ This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted and the program terminates.

```
int[] number = { 4, 8, 6, 32 };
int[] denom = { 2, 0, 2, 4 };
try {
    for (int i = 0; i < number.Length; i++) {
        try {
            Console.WriteLine(i + ":" + number[i] / denom[i]);
        } catch (ArithmeticException ex) {
            Console.WriteLine("Inner:Can't divide by ZERO");
        }
    }
} catch (Exception ex) {
    Console.WriteLine("Outer:No matching element found.");
}
```

0:2  
Inner:Can't divide by ZERO  
2:3  
3:8

Inner Try-catch block

Outer Try-catch block

11 COMPSCI280 Handout11



## Exception Propagation

- ▶ If it is not appropriate to handle the exception where it occurs, it can be handled at (propagate to) a higher level
  - ▶ The first method it **finds** that catches the exception will have its catch block executed. At this point the exception has been handled, and the propagation stops (no other catch blocks will be executed). Execution resumes normally after this catch block.

```
try {
    Console.WriteLine("Method+");
    PropagateException();
} catch (Exception ex) {
    Console.WriteLine("General Exception");
}
Console.WriteLine("Method-");
```

```
public void PropagateException() {
    try{
        Console.WriteLine("+");
        int x = Convert.ToInt32("a");
    } catch (FormatException ex) {
        Console.WriteLine(ex.Message);
    }
    Console.WriteLine("-");
}
```

method+  
+  
General Exception  
method-

No matching catch block

12 COMPSCI280



## Finally

- ▶ The Finally block is optional.
- ▶ It allows for cleanup of actions that occurred in the try block but may remain undone if an exception is caught
- ▶ Code within a finally block is **guaranteed to be executed** if any part of the associated try block is executed regardless of an exception being thrown or not

```

string s = "1";
try {
    int x = Convert.ToInt32(s);
} catch (Exception ex) {
    Console.WriteLine("General Exception");
} finally {
    Console.WriteLine("Finally");
}
Console.WriteLine("Finished");

```

No error

Finally  
Finished

```

string s = "a";
try {
    int x = Convert.ToInt32(s);
} catch (Exception ex) {
    Console.WriteLine(ex.Message);
} finally {
    Console.WriteLine("Finally");
}
Console.WriteLine("Finished");

```

Input string was not in ...  
Finally  
Finished

Handout11

13



## Explicitly Throw Exceptions

- ▶ You can explicitly throw an exception using the throw statement.
- ▶ The program stops immediately after the throw statement; and any subsequent statements are not executed.
- ▶ The throw statement requires a single argument
  - ▶ It provides information about the exception to be thrown

```

try {
    if ( i <=0)
        throw new Exception("Throw an error.");
    Console.WriteLine(i);
} catch (Exception ex) {
    Console.WriteLine("General Exception");
} finally {
    Console.WriteLine("Finally");
}
Console.WriteLine("Finished");

```

General Exception  
Finally  
Finished

Handout11

14

COMPSCI280

Handout11



## The checked keyword

- ▶ is used to control the overflow-checking context for integral-type arithmetic operations and conversions.
- ▶ Example:
  - ▶ if an expression produces a value that is outside the range of the destination type, by default,
  - ▶ C# generates code that allows the calculation to silently overflow
  - ▶ i.e. you get the wrong answer

```

int number = int.MaxValue;
Console.WriteLine(number);
number++;
Console.WriteLine(number);

```

2147483647  
-2147483648

- ▶ Use checked keyword to turn on the integer arithmetic overflow checking (or unchecked to turn off)

```

checked {
    number++;
    Console.WriteLine(number);
}

```

15

COMPSCI280

Handout11