using

- using
  - Defines a scope, outside of which an object or objects will be disposed of.
    - It is usually best to release limited resources such as file handles and network connections as quickly as possible.

```
using (Font font1 = new Font("Arial", 10.0f)) {
}
```

```
using (Resource res = new
Resource()) {
    res.DoWork();
}
```

```
Resource res = new Resource(...);
try {
    res.DoWork();
}
finally {
    if (res != null)
((IDisposable)res).Dispose();
}
```

using directive

- The using directive has two uses:
  - You can reference types in the library without fully qualifying the type name
  - To create an alias for a namespace.

```
using System;
namespace B {
  public class Program1 {
    static void Main(string[] args){
      Console.WriteLine("Hello");
    }
  }
}
```

With using directive

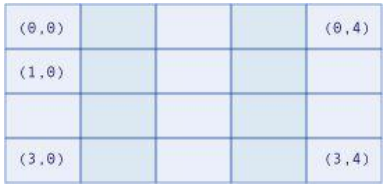
```
namespace B {
  public class Program2 {
    static void Main(string[] args){
      System.Console.WriteLine("Hello");
    }
  }
}
```

Fully qualifying

using alias

```
using C = System.Console;
namespace B {
  public class Program3 {
    static void Main(string[] args){
      C.WriteLine("Hello");
    }
  }
}
```

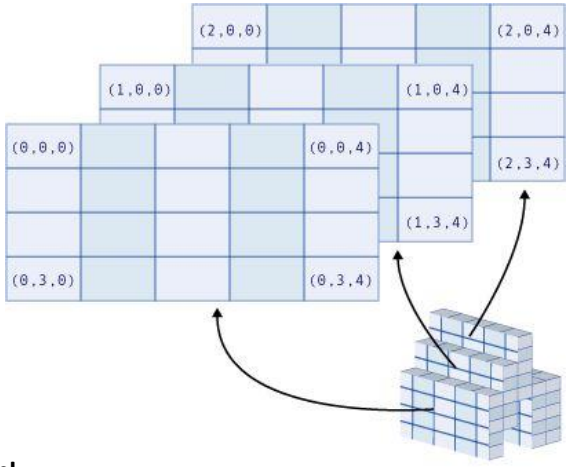
Multidimensional Arrays



- You can declare arrays with up to 32 dimensions
  - A Two-dimensional array is like a table with rows and columns

```
int[,] rectangle = new int[4, 5];
```

  - A Three-dimensional array is like a cube,
    - with rows, columns, and pages




```
int[, ,] cube = new int[5, 4, 5];
```

- Overview
  - The rank/dimension of an array is held in its Rank property
  - The lowest subscript value for a dimension is always 0
  - The length of each dimension is returned by the GetLength method
    - Note that the argument you pass to GetLength and GetUpperBound (the dimension for which you want the length or the highest subscript) is 0-based.
  - The highest subscript value is returned by the GetUpperBound method for that dimension.
  - The length property of the array is the total size of an array

```
Console.WriteLine(rectangle.Length);
Console.WriteLine(rectangle.GetLength(0));
Console.WriteLine(rectangle.GetUpperBound(0));
```

20
4
3

Creating and Using 2D Arrays

- Creating Arrays
  - To create an array using the New clause
    - Declare a variable, create the array and initialized with default values

```
double[,] weights = new double[2, 2];
```

  - To create an array and supply initial element values in the New clause

```
int[,] nums2 = new int[,] { {1, 2, 3}, {4, 5, 6}};
```

```
int[,] nums3 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

1	2	3
4	5	6

- Accessing elements

```
Console.WriteLine(nums2[rowToGet, colToGet]);
```

  - To access elements, we use pretty much the same technique as a 1D array
- Navigating elements
  - Nested loops are used to navigate the array elements

```
for (int i = 0; i < nums2.GetLength(0); i++){
  for (int j = 0; j < nums2.GetLength(1); j++)
    Console.Write(nums2[i, j] + " ");
  Console.WriteLine();
}
```

The this keyword

- The **this** keyword refers to the current instance of the class
  - It is useful if you have name clashes between the parameter name and your internal variable

```
public Employee(string name, string alias) {
    this.name = name;
    this.alias = alias;
}
```

Refers to the argument from the method call

  - To pass an object as a parameter to other methods

```
CalcTax(this);
```

  - To invoke another constructor in the same object from a constructor

```
public MyPoint(): this(0,0) {
                ...
}
```

A 'constructor' is a method with the same name as its class; run when someone wants a 'new' instance of that class

Static Classes

- Note: Methods and properties are only available after creating an instance of the class
- Static classes and class members are used to create data and functions that can be accessed without creating an instance of the class
  - Static methods can be invoked directly from the class (not from a specific instance of a class)
  - Rules:
    - Static classes contain static members
    - Static classes cannot be instantiated

```
static class CompanyInfo
{
    public static string GetCompanyName() { return "CompanyName"; }
    public static string GetCompanyAddress() { return "CompanyAddress"; }
    //...
}
```

Fields & Properties

Attributes represent the internal "state" of a given instance of this class.

```
class MyPoint {
  private int _x;
  private int _y;

}
```

- Fields
  - Fields store the data a class needs to fulfill its design
  - Fields should (generally) be declared as Private.
- Properties
  - Properties are retrieved and set like fields, but are implemented using property **Get** and property **Set methods**, which provide more control on how values are set or returned.
    - It helps isolate your data and allows you to validate values before they are assigned or retrieved.
    - Property has Public access
    - The **value** keyword is used to define the value being assigned by the set method.
    - Property can be read-only (with the Get portion only) or write-only (with the Set portion only)

```
public int X {
  get {
    return _x;
  }
  set {
    if (value > 0)
      _x = value;
  }
}
```

```
public  int Y {
  get {
    return _y;
  }
}
```

Read-only property

Auto-implemented properties

- Auto-implemented properties make property declaration more concise when no additional logic is required in the property accessor methods
  - Compiler creates a private, anonymous backing field that can only be accessed through the property's get and set accessors

```
 public class emp {
        public decimal salary { get; set; }
    }

public class emp2 : emp {
        public new decimal salary {
            get {return base.salary;}
            set {
                if (value >= 0) base.salary = value;
                else Console.WriteLine("Cannot accept negative salary!");
            }
        }
    }
```

Auto-implemented accessors for anonymous private variable (the 'backing field')

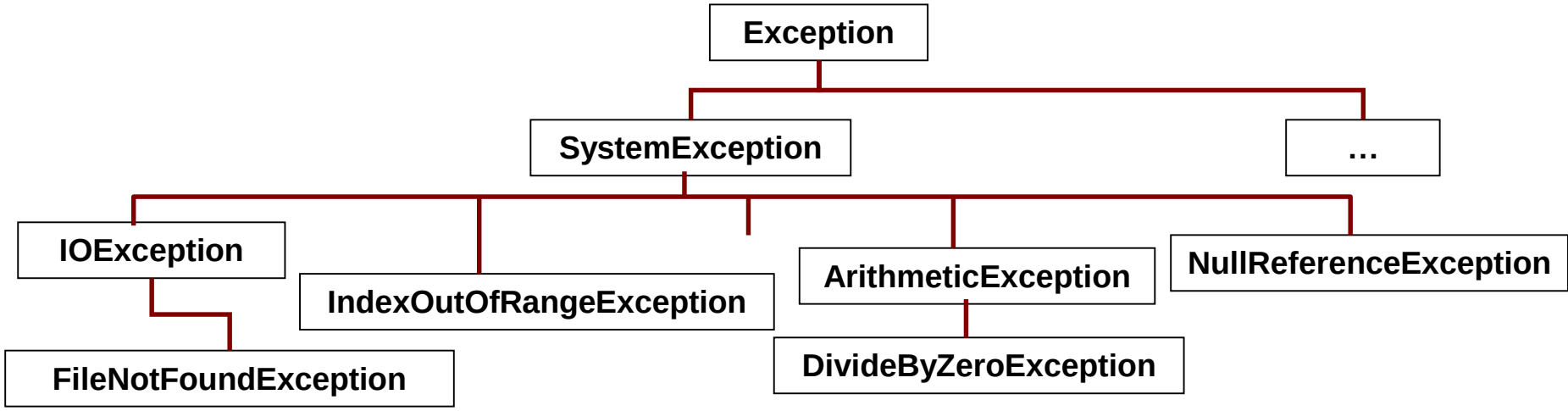
A child class can override the parent property and selectively reuse the parent ('base') class get and set methods

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, by unavailable operating system resources, or other unexpected circumstances
- In the .NET Framework, an exception is an object that inherits from the Exception Class.
  - Has properties, such as a Message
- 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 continue running

Exception Hierarchy

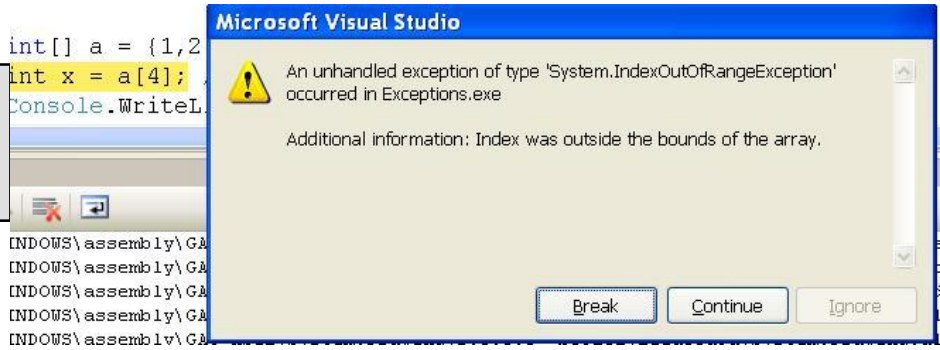
Exception Type	Description
Exception	Base Class
SystemException	Base class for all runtime-generated errors.
IOException	when an I/O error occurs.
FileNotFoundException	when an attempt to access a file that does not exist on disk fails.
IndexOutOfRangeException	when an array is indexed improperly.
ArithmeticException	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

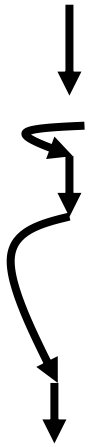
- Try block
  - Place the code that might cause the exception in a try block
  - When an error happens, the .NET system ignores the rest of the code in the try block and jumps to the catch block
- Catch block
  - Specify the exception that you wish to catch or catch the general exception. (Since all exceptions are subclasses of the Exception class, you can catch all exceptions 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

```
Index was outside the bounds ...
x=-1
```

```
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 matching 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
```

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) will match any Exception in the subtree.
  - Note: Exception subclasses (specific types of exception) must come before any of their superclasses (e.g., the general Exception) in the order of the **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");
```

Nested Try-Catch

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

- 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.");
}
```

Inner Try-catch block

Outer Try-catch block

Exception Propagation

- If it is not appropriate to handle the exception where it occurs, it can be handled at (propagated 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("Starting calls...");
      PropagateException("123");          //this is fine
      PropagateException("a");            //format Exception
      PropagateException("3000000000");   //overflow
      PropagateException("234");          // doesn't happen
   }
catch (Exception ex)
   {
      Console.WriteLine("General Exception");
   }
Console.WriteLine("Done calls...");
```

```
Starting calls...
Entering subroutine...
x=123
Exiting subroutine...
Entering subroutine...
Input string was not in correct format.
Exiting subroutine...
Entering subroutine...
General Exception
Done calls...
```

```
public static void PropagateException(string s)
  {
    try
      {
         Console.WriteLine("Entering subroutine...");
         int x = Convert.ToInt32(s);
         Console.WriteLine("x={0}",x);
      }
    catch (FormatException ex)
      {
         Console.WriteLine(ex.Message);
      }
    Console.WriteLine("Exiting subroutine...");
  }
```

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 the try block succeeds **or** if the exception is thrown and caught (even if catch is at a higher level)

```
sting 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
```

Handy if the exception gets handled by a higher level routine, as per previous slide

```
sting 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
```

Explicitly Throw Exceptions

- You can explicitly throw an exception using the **throw** statement
- If the exception is unhandled, the program stops immediately after the throw statement and any subsequent statements are not executed
- You have the option to throw exceptions of specific types, and can specify the message as a parameter
  - E.g. throw new System.OverflowException("Boo!");

ex.Message would be “Boo!”

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

```
General Exception
Finally
Finished
```

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 the **checked** keyword to turn on the integer arithmetic overflow checking (or **unchecked** to turn off)

Throws an overflow exception

```
int number = int.MaxValue;
checked {
  number++;
  Console.WriteLine(number);
}
```

TryParse

- Replaces the need to use try/catch exception handling for a format exception
- Returns boolean of whether the conversion succeeded
  - If successful, the converted result goes into the 'out' variable

`value` is the source string, `number` gets the converted result (e.g. 123 if value="123"

```
bool result = Int32.TryParse(value, out number);
    if (result)
       {
          Console.WriteLine("Converted '{0}' to {1}.", value, number);
       }
    else
       {
          Console.WriteLine("Attempted conversion of '{0}' failed.",
              value == null ? "<null>" : value);
       }
```

TryParse is available for other types (e.g. DateTime.TryParse)

Nullable types

- Normally a simple data type (e.g. int, float, bool, char, DateTime) cannot be assigned the value null
  - E.g. <null> isn't an integer, so it can't go in an int
  - Note string can be null (since strings are really sort of arrays anyhow)
- Nullable types take the normal range of their data type, but can also have the value of null
  - See https://msdn.microsoft.com/en-us/library/1t3y8s4s.aspx

? denotes a nullable type

```
int? x;
x = null;
if (x == null)
  Console.WriteLine("x is null");
x = 123;
Console.WriteLine("x=" + x);
```

Nullables and EF

- Nullable types are handy to represent a numeric or DateTime database field that is allowed to be null

```
[Table("Employee")]
   public class Employee
   {
        [Key]
        public int id { get; set; }
        public string surname { get; set; }
        public string givenNames { get; set; }
        public int rank { get; set; }
        public DateTime appointDate { get; set; }
        public int? city { get; set; }
   }
```

city might be foreign key for another table, but not every Employee is necessarily assigned to a city

Writing To Files

- Declare a new StreamWriter object
  - If the file does not exist, a new one is created
  - The default location for file is the bin\Debug directory

```
StreamWriter sw = new StreamWriter(filename, true);
```

True = Append data to this existing file

- Use StreamWriter's WriteLine/Write methods
  - Copies the data to a buffer in memory
    - Write, WriteLine Method

```
sw.WriteLine("Well done.");
```

- Call StreamWriter's Close method
  - Transfers the data from buffer to the file and release system resource used by the stream (or use a **using** statement, see http://msdn.microsoft.com/en-us/library/6ka1wd3w.aspx)

```
sw.Close();
```

- Use try and catch for exception handling
  - To make your application more robust

```
StreamWriter sw = null;
try {
  ...
} catch (Exception ex) {
  ...
} finally {
  if (sw != null)
    sw.Close();
}
```

Always close the file

Reading From Files

```
StreamReader sr = new StreamReader(filename);
```

- Declare a new StreamReader object
  - Note: The file must already exist in the specified location or an error is generated
- Use StreamReader's ReadLine/ReadToEnd methods
  - ReadLine(), or
    - Reads the next line of data
    - Use a loop to read through

```
while ((line = sr.ReadLine()) != null){
    Console.WriteLine(line);
}
```

  - ReadToEnd()
    - Reads the stream from the current position to the end of the stream.

```
Console.WriteLine(sr.ReadToEnd());
```

- Call StreamReader's Close method to releases system resources
- Again, use try and catch for exception handling

```
StreamReader sr = null;
try {  ... } catch (Exception ex) { ...
} finally {
  if (sr != null) sr.Close();
}
```

Conclusion

- We've learned how to work with run-time errors ('exceptions') to control program execution even when things go wrong
  - Using the IDE to understand errors
  - Controlling error processing with try-catch-finally and using the error type hierarchy
  - Avoiding errors with TryParse
- Use of nullables to directly represent values that can be within a data type's range or can be null
- File I/O as a case where error handling is essential to stable program execution