Object-Oriented Programming Using Microsoft Visual C# .NET

Programming 3

Collections
(more about collections)

Working with the ArrayList Class

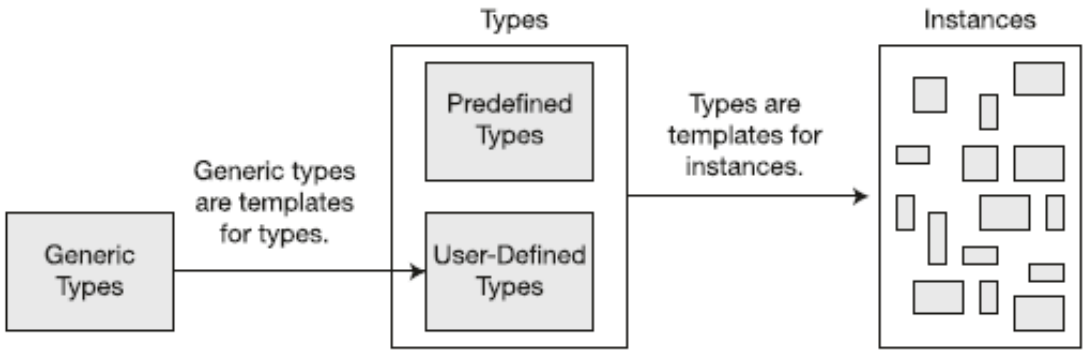
- The `ArrayList` is a class defined in the `System.Collections` namespace.
- It represents a dynamically sized array of objects.
- Features of `ArrayList` class:
  - Allows you to add, remove, insert, and sort the items in it.
  - Contains items in the order of addition.

Working with the ArrayList Class

- Consists of an index identifier assigned to its items.
- Enables you to retrieve items in any order by means of their associated index numbers.

Introducing Generics

- Benefits of using Generics:
  - Allows you to work with any data type.
  - Provides many of the advantages of the strongly typed collections.
  - Increases code performance by reducing the number of required casting operations.



Generic types are templates for types

Introducing Generics

Generic Class	Nongeneric Class	Description
List<T>	*ArrayList, StringCollection*	A dynamically resizable list of items
Dictionary<T,U>	*HashTable, ListDictionary, HybridDictionary, OrderedDictionary, NameValueCollection, StringDictionary*	A generic collection of name-value pairs
Queue<T>	*Queue*	A generic implementation of a FIFO list

Introducing Generics

Generic Class	Nongeneric Class	Description
Stack<T>	*Stack*	A generic implementation of a LIFO list
SortedList<T,U>	*SortedList*	A generic implementation of a sorted list of generic name/value pairs
Comparer<T>	*Comparer*	Compares two generic objects for equality

Introducing Generics

Generic Class	Nongeneric Class	Description
LinkedList<T>	*N/A*	A generic implementation of a doubly linked list
Collection<T>	*CollectionBase*	Provides the basis for a generic collection
ReadOnlyCollection<T>	*ReadOnlyCollectionBase*	A generic implementation of a set of read-only items

Lists

- The most common sort of collection is a List<T>. Once you create a List<T> object, it's easy to add an item, remove an item from any location in the list, peek at an item, and even move an item from one place in the list to another.
- Here's how a list works:
  - First you create a new instance of List<T>
  - You need to specify the type of object or value that the list will hold by putting it in angle brackets <> when you use the new keyword to create it.
  - List<Card> cards = new List<Card>();
- The <T> at the end of List<T> means it's *generic*.
- The T gets replaced with a type—so List<int> just means a List of ints.

The `List<T>` Class

- Implements the abstract data structure list using an array
  - All elements are of the same type `T`
  - `T` can be any type, e.g. `List<int>`, `List<string>`, `List<DateTime>`
  - Size is dynamically increased as needed
- Basic functionality:
  - `Count` – returns the number of elements
  - `Add(T)` – appends given element at the end

List<T> – Simple Example

```
static void Main()
{
   List<string> list = new List<string>() { "C#",
"Java" };

   list.Add("SQL");
   list.Add("Python");

   foreach (string item in list)
   {
      Console.WriteLine(item);
   }

   // Result:
   //    C#
   //    Java
   //    SQL
   //    Python
}
```

Inline initialization: the compiler adds specified elements to the list.

`List<T>` – Functionality

- `list[index]` – access element by index
- `Insert(index, T)` – inserts given element to the list at a specified position
- `Remove(T)` – removes the first occurrence of given element
- `RemoveAt(index)` – removes the element at the specified position
- `Clear()` – removes all elements
- `Contains(T)` – determines whether an element is part of the list

`List<T>` – Functionality (2)

- `IndexOf()` – returns the index of the first occurrence of a value in the list (zero-based)
- `Reverse()` – reverses the order of the elements in the list or a portion of it
- `Sort()` – sorts the elements in the list or a portion of it
- `ToArray()` – converts the elements of the list to an array
- `TrimExcess()` – sets the capacity to the actual number of elements

Primes in an Interval – Example

```
static List<int> FindPrimes(int start, int end)
{
   List<int> primesList = new List<int>();
   for (int num = start; num <= end; num++)
   {
      bool prime = true;
      for (int div = 2; div <= Math.Sqrt(num); div++)
      {
         if (num % div == 0)
         {
            prime = false;
            break;
         }
      }
      if (prime)
      {
         primesList.Add(num);
      }
   }
   return primesList;
}
```

Stacks

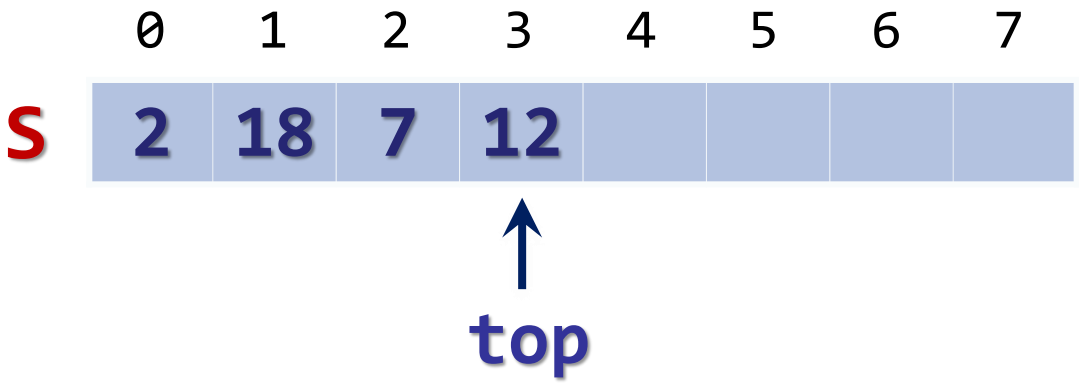
Static and Dynamic Implementation

The Stack

- LIFO (Last In First Out) structure
- Elements inserted (push) at “top”
- Elements removed (pop) from “top”
- Useful in many situations
  - E.g. the execution stack of the program
- Can be implemented in several ways
  - Statically (using array)
  - Dynamically (linked implementation)
  - Using the `Stack<T>` class

Static Stack

- Static (array-based) implementation
  - Has limited (fixed) capacity
  - The current index (`top`) moves left / right with each pop / push



The Stack<T> Class

The Standard Stack Implementation in .NET

The `Stack<T>` Class

- Implements the stack data structure using an array
  - Elements are from the same type `T`
  - `T` can be any type, e.g. `Stack<int>`
  - Size is dynamically increased as needed
- Basic functionality:
  - `Push(T)` – inserts elements to the stack
  - `Pop()` – removes and returns the top element from the stack

The Stack<T> Class (2)

- Basic functionality:
  - `Peek()` – returns the top element of the stack without removing it
  - `Count` – returns the number of elements
  - `Clear()` – removes all elements
  - `Contains(T)` – determines whether given element is in the stack
  - `ToArray()` – converts the stack to an array
  - `TrimExcess()` – sets the capacity to the actual number of elements

Stack<T> – Example

- Using `Push()`, `Pop()` and `Peek()` methods

```
static void Main()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("1. Joe");
    stack.Push("2. Steven");
    stack.Push("3. Maria");
    stack.Push("4. George");
    Console.WriteLine("Top = {0}", stack.Peek());
    while (stack.Count > 0)
    {
        string personName = stack.Pop();
        Console.WriteLine(personName);
    }
}
```

Queues

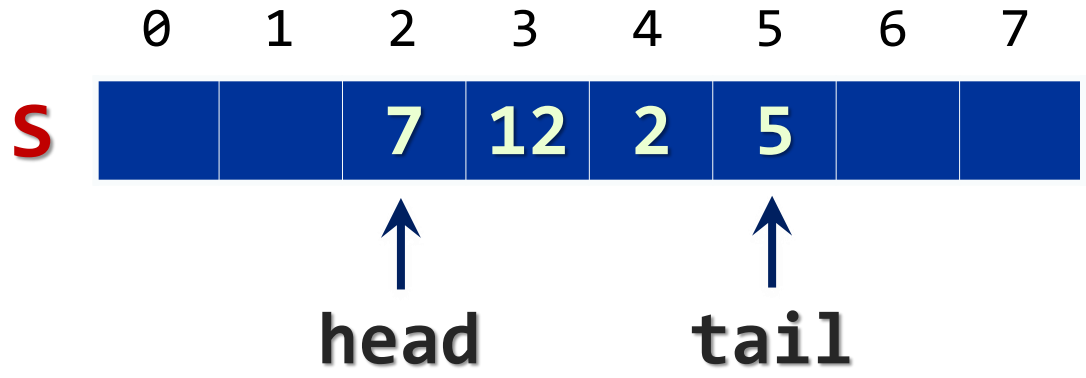
Static and Dynamic Implementation

The Queue

- FIFO (First In First Out) structure
- Elements inserted at the tail (Enqueue)
- Elements removed from the head (Dequeue)
- Useful in many situations
  - Print queues, message queues, etc.
- Can be implemented in several ways
  - Statically (using array)
  - Dynamically (using pointers)
  - Using the `Queue<T>` class

Static Queue

- Static (array-based) implementation
  - Has limited (fixed) capacity
  - Implement as a “circular array”
  - Has `head` and `tail` indices, pointing to the head and the tail of the cyclic queue



The Queue<T> Class

Standard Queue Implementation in .NET

The Queue<T> Class

- Implements the queue data structure using a circular resizable array
  - Elements are from the same type `T`
  - `T` can be any type, e.g. `Stack<int>`
  - Size is dynamically increased as needed
- Basic functionality:
  - `Enqueue(T)` – adds an element to the end of the queue
  - `Dequeue()` – removes and returns the element at the beginning of the queue

The Queue<T> Class (2)

- Basic functionality:
  - `Peek()` – returns the element at the beginning of the queue without removing it
  - `Count` – returns the number of elements
  - `Clear()` – removes all elements
  - `Contains(T)` – determines whether given element is in the queue
  - `ToArray()` – converts the queue to an array
  - `TrimExcess()` – sets the capacity to the actual number of elements in the queue

Queue<T> – Example

- Using `Enqueue()` and `Dequeue()` methods

```
static void Main()
{
    Queue<string> queue = new Queue<string>();
    queue.Enqueue("Message One");
    queue.Enqueue("Message Two");
    queue.Enqueue("Message Three");
    queue.Enqueue("Message Four");
    while (queue.Count > 0)
    {
        string message = queue.Dequeue();
        Console.WriteLine(message);
    }
}
```

The String Class

String, String class

- Contains Unicode characters (characters collection)
- The String is a reference type.
- Objects of the string class store an immutable series of characters.They are considered immutable because once you give a string a value; it cannot be modified. Methods that seem to be modifying a string are actually returning a new string containing the modification.
- You can use the **Length** property of a string to determine its length.
- String type allows individual characters to be accessed using an index with [ ].

String class

Properties	Description
Length	Gets the number of characters
Static methods	
Compare(stringA,stringB)	Compares two strings. Returns: (-1, 0, 1)
Bool Equals(stringA, stringB)	Determines whether two strings have the same value. (return true or false)
Concat()	Concatenates one or more string
Non static methods	
CompareTo()	Compares the string with the string given as a parameter
Equals() pl.: string name = "Ann"; name.Equals("Andy")	**Equals** () returns true if the two string have the same value
Contains() string today = DateTime.Now.ToLongDateString(); today.Contains("july")	Returns with true, if the string contains the other string given as a parameter

String class

Non static methods	Description
StartsWith(string) ; EndsWith(string)	Determines wether the beginning (the end) of this instance string matches the specified string.
IndexOf(string, [startindex])	Returns the index of the first occurence of a string wthitin the instance. Returns -1 if there is no any occurence.
LastIndexOf(string, [startindex])	Returns the index of the last occurence of a specified string or character. (If no returns -1.)
Insert(startindex, string)	Insert a specified instance of a string at a specified index position.
Remove(startindex, count)	Deletes a specified number of characters beginning at a specified position.
Replace(oldString, newstring) string Replace(char oldChar, char newChar) string Replace(string oldValue, string newValue)	Replaces all occurences of a character or string with another character or string.

String class

Non static methods	Description
Substring(startIndex, [length])	Retrieves a substring from the string beginning at the specified position.
ToLower(), ToUpper()	Returns a copy of the string in lowercase (uppercase) .
PadLeft(), PadRight() string PadLeft(int totalWidth) string PadLeft(int totalWidth, char paddingChar) string PadRight(int totalWidth) string PadRight(int totalWidth, char paddingChar)	Right-aligns (Left-aligns) the characters in the string padding on the left (padding on the right) with spaces or a specified character.
Trim(), TrimStart() és TrimEnd()	Removes all occurences of a set of specified characters from the beginning and end.

String class methods

Non static methods	Description
Split() string[] Split(params char[] separator) string[] Split(char[] separator, int count) string[] Split(char separator, StringSplitOptions options) string[] Split(string[] separator, StringSplitOptions options) string[] Split(char[] separator, int count, StringSplitOptions options) string[] Split(string[] separator, int count, StringSplitOptions options)	Identifies the substrings in the string that are delimited by one or more characters specified in an array, then places the substrings into a string array.

String class methods

```
string s = "verylonglongstring";
char[] chs = new char[] { 'y', 'z', 'o' };

Console.WriteLine(s.IndexOf('r')); // 2
Console.WriteLine(s.IndexOfAny(chs)); // 3
Console.WriteLine(s.LastIndexOf('n')); // 16
Console.WriteLine(s.LastIndexOfAny(chs)); // 9
Console.WriteLine(s.Contains("long")); // true
```

String class methods

```
static void Main(string[] args)
{
    string s = "smallstring";
    char[] chs = new char[] { 's', 'g' };

    Console.WriteLine(s.Replace('s', 'l')); // lmallltring
    Console.WriteLine(s.Trim(chs)); // mallstrin
    Console.WriteLine(s.Insert(0, "one")); // onesmallstring
    Console.WriteLine(s.Remove(0, 2)); // allstring
    Console.WriteLine(s.Substring(0, 3)); // sma
    Console.WriteLine(s.ToUpper()); // SMALLSTRING
    Console.WriteLine(s.ToLower()); // smallstring

    Console.ReadKey();
}
```

StringBuilder class

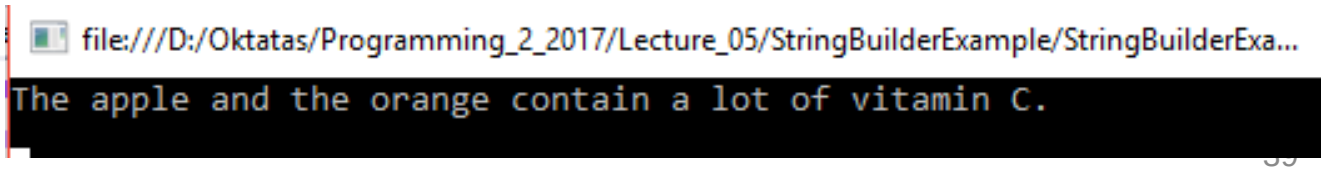
- NET includes another class, StringBuilder, which represents a mutable string of characters. Objects of this class can have data appended onto the same object. The StringBuilder class offers many of the same members as the string class does.
- For applications that concatenate or add characters to the string, you will want to consider instantiating objects of the StringBuilder class
- Most often used methods of the StringBuilder class ara `Append()`,`Insert()` and `AppendFormat()`.

StringBuilder class

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace StringBuilderExample
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            StringBuilder sb = new StringBuilder();
            sb.Append("The apple ");
            sb.Append("and the orange ");
            sb.Append("contain a lot of vitamin C.");
            string result = sb.ToString();
            Console.WriteLine(result);
            Console.ReadKey();
        }
    }
}
```



Exceptions

What are Exceptions?

- Exception
  - Error condition that is unexpectedly encountered during program execution
- Exception class
  - Base class for all exceptions
- Exception handler
  - Code written to bypass default error messages
- Execution call stack
  - Keeps track of all methods that are in execution

Types of Errors

- Syntax error
  - Occurs when C# grammar is violated
- Semantic error
  - Occurs when syntax is correct, but what you want the code to do is not
- Logic error
  - Program with semantic errors is able to complete execution but displays incorrect results
- Run-time error
  - Triggered during program execution

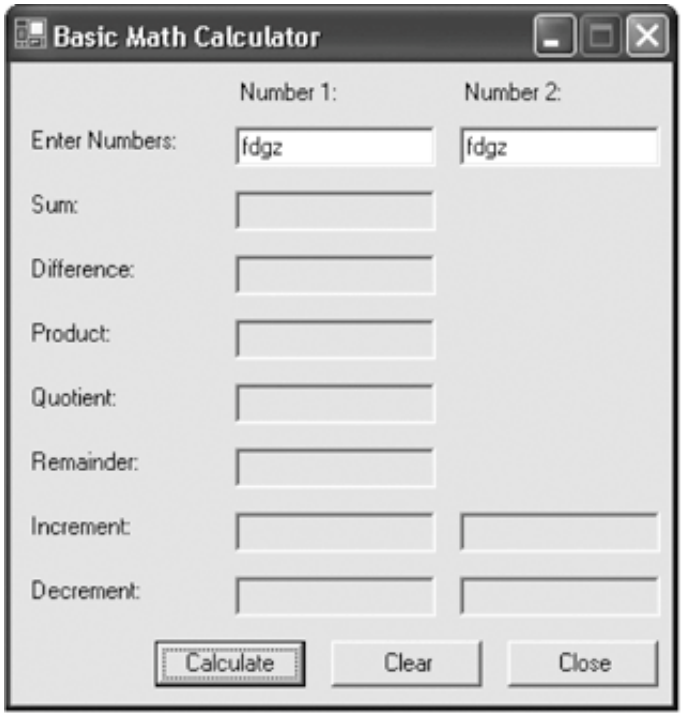
Code snippet from the Restaurant Bill Calculator application

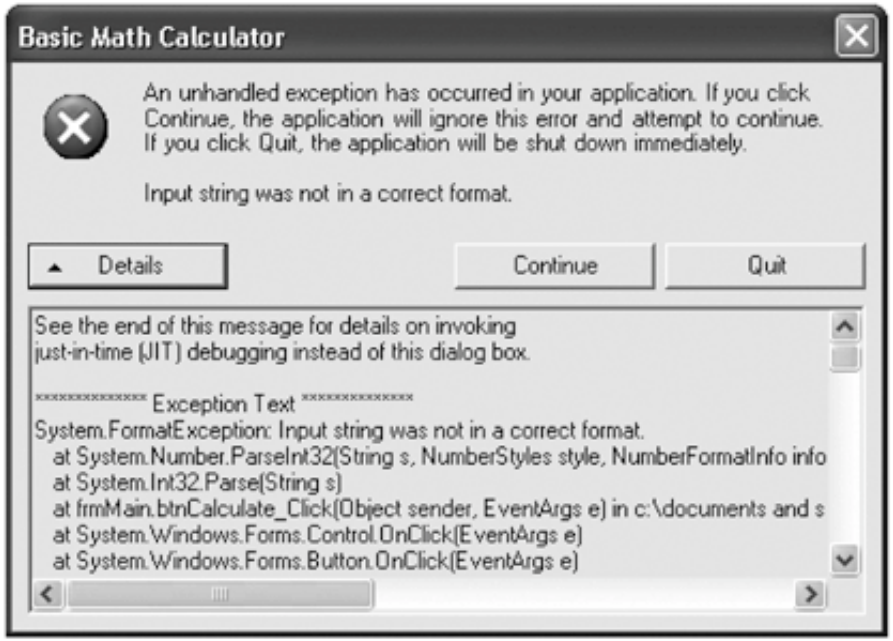
```
287             // Calculate the total per diner.
288
289             totalCost = Math.Round((mealCost + drinkCost)
290                 + ((decimal) taxRate * (mealCost + drinkCost))
291                 + ((decimal) tipPct * (mealCost + drinkCost))
292                 - discounts, 2);
293             costPerDiner = Math.Round(totalCost % (decimal) nbrDiners, 2);
294
295             // Display the results back to the user.
296
297             txtCostPerDiner.Text = costPerDiner.ToString("C");
```

What Causes Exceptions?

- Exceptions are thrown in the following cases
  - You try to loop through an array and go past a valid location
  - Your program uses up considerable system resources
  - Your program specifies the wrong location for the text file or database
  - Your file does not have enough data, and your program attempts to read from it

Exception Thrown by Entering Alphabetic Characters into the Basic Math Calculator





Syntax for Exception Handlers

- Try block
  - Surrounds regular processing code that may throw an exception
- Catch block
  - Contains code that provides alternative processing steps in the event that an exception is thrown
- Finally block
  - Identifies processing steps executed after the try or catch blocks, whether an exception is thrown or not

General Syntax for an Exception Handler

```
try
{
    normal program statements that may trigger an error
}
catch
{
    exception processing statement block if exception thrown in try block
}
finally
{
    clean-up processing statement block in all cases (error or no error)
}
```

Data Type Check

- It is important to verify that the data you are processing are the correct types
- If you perform calculations on an integer value
  - Ensure that user entered a string that can be converted and stored as an *int* data type
- Strings
  - Do not have to be checked because Text property of a text box is already of type *string*

Exception Handler for Quantity Input Field

```
 1        try
 2        {
 3            quantity = int.Parse(txtQuantity.Text);
 4        }
 5        catch
 6        {
 7            MessageBox.Show("Quantity must be an integer value.",
 8                Text, MessageBoxButtons.OK, MessageBoxIcon.Information);
 9            txtQuantity.Focus();
10            return;
11        }
```

Data Type Exception Handler for Optional Input Field

```
1       if (txtBirthDate.Text != "")
2       {
3           try
4           {
5               birthDate = DateTime.Parse(txtBirthDate.Text);
6           }
7           catch
8           {
9               MessageBox.Show(("Birthdate must be a valid date.",
10                  Text, MessageBoxButtons.OK, MessageBoxIcon.Information);
11              txtBirthDate.Focus();
12              return;
13          }
14      }
```

A General Exception Handler

- To enhance usability of your program
  - Write a generalized exception handler to deal with any type of exception that could be thrown
- Message property
  - Returns an English description of the current exception
- *StackTrace property*
  - Can help locate where exception was thrown

Syntax for General Exception Handler without the Finally Block

```
try
{
    normal program statements that may trigger an error
}
catch
{
    exception processing statement block if exception thrown in try block
}
```

Syntax for the General Exception Handler to use the Exception Object

```
try
{
    normal program statements that may trigger an error
}
catch (Exception ex)
{
    exception processing statement block if exception thrown in try block
}
```

Using the Message Property of the Exception Object

```
catch (Exception ex)
{
    MessageBox.Show("Error: " + ex.Message,
        Text, MessageBoxButtons.OK, MessageBoxIcon.Information);
    return;
}
```

When to Catch Exceptions

- When you include multiple exception handlers in a single event
  - Each try block should have its own catch or finally block
- Keep the general exception handler
  - But nest individual exception handlers for individual or sets of program statements
- When you nest try-catch blocks
  - If exception is triggered within a nested try block control passes to the associated catch block

Nested Exception Handlers in Average Item Price Program

```
 1   private void btnOK_Click(object sender, System.EventArgs e)
 2  {
 3       // Define variables to hold values in corresponding text boxes.
 4
 5       decimal total;
 6       int number;
 7       decimal average;
 8
 9       try
10       {
11
12           // If txtTotal has a numeric value, convert it to decimal and verify range.
13
14           try
15           {
16               total = decimal.Parse(txtTotal.Text);
17           }
```

Catching Exceptions of a Specific Type

- *DivideByZeroException*
  - Performing division in which the denominator is zero
- *FormatException*
  - Attempting to convert “ten” to a numeric data type using Parse
- *OverflowException*
  - Attempting to store a value larger than the range limit for a particular numeric data type

Throwing your Own Exceptions

- *Syntax*

```
If (BooleanExpression)
{
    ApplicationException exVar =
        new ApplicationException("Message String");
    throw exVar;
}
```

Exceptions and the Call Stack

- Execution call stack
  - Keeps track of all the methods that are in execution currently
- Stack trace
  - Trace of all method calls
  - Provides information to help you pinpoint problems

Revised General Exception Handler to Show Stack Trace

```
192    catch (Exception ex)
193    {
194        MessageBox.Show("Error: " + ex.StackTrace, Text,
195            MessageBoxButtons.OK, MessageBoxIcon.Information);
196        txtUser.Focus();
197        return;
198    }
```

Screen Shots from Revised General Exception Handler to Show Stack Trace

Thank you for your attention!