Building Applications in C#

Mike Teodorescu ’11
Microsoft

Hello

- CS50 TF ‘09, ‘10
- CS concentrator class of 2011
- Worked on Windows Home and Small Business Server 2011 as an intern in Redmond, WA
- Now engineer in the Application Virtualization team at the Microsoft New England Research and Development Center (NERD)

Microsoft NERD

- Teams:
  - Application Virtualization
  - Microsoft Research
  - Sharepoint Workspace 2010
  - Office 365
  - Microsoft Lync
  - Interactive Entertainment Business
  - Microsoft Advertising
  - Microsoft-Novell Interoperability Lab
- Address: 1 Memorial Drive in Cambridge
- http://microsoftcambridge.com

- Allows applications to run in separate virtual environments, eliminating conflicts with other applications or with different versions of the same application.
- Previously incompatible apps can be run simultaneously
- Fully functional Windows apps are streamed on demand to users' desktops
- We work on very large scale systems
- More at http://microsoftcambridge.com/Teams/ApplicationVirtualization/tabid/83/Default.aspx

TOC

1. Intro to C#
2. Hello World
3. Object-Oriented concepts
4. Collections
5. Control Flow
6. File I/O
7. Strings and REGEX
8. XML and LINQ Queries
9. Exception Handling
10. Debugging

Intro to C#

- Object-oriented programming language
- Type-safety guaranteed
  - cannot store data of one type in a location corresponding to a different type
  - cannot have uninitialized variables
- Garbage collection – automatic memory management (no need to worry about freeing memory you allocated)
- Built-in datastructures that handle their own sizing
- Exception handling allows detection and recovery from errors during execution of your code

Intro to C# (2)

- Code is sandboxed into a *Virtual Execution System* which ensures safe code execution and memory management
- C# comes with the .NET *Framework Class Library,* which means it is packed with libraries that simplify many common tasks that you would otherwise have to code:
  - XML and string parsing
  - web programming
  - data structures
  - scalable networks and server programming
  - database management
  - GUI programming, and others…
- Parallel programming made easy (*Parallel Computing Platform* – takes care of the lower level coding for you)

Intro to C# (3)

- Standard types are built-in
- You can create your custom types through OO
- C# enables component-oriented programming, which means you can create self-contained units of functionality that can communicate
- Your code is a living ecosystem – extensible and easy to maintain.
- The goal is to allow you to spend more time on the concepts and less on lower-level details. It enables you to think more about your solution and the abstract model behind it.

Development Environment

- Some benefits of using Visual Studio 2010:
  - Multitude of languages you can use (Visual C#, Visual C++, VB) – the .NET languages all use a common type system
  - Auto-Complete
  - Auto-formatting
  - Cross-platform development
  - Multitude of code templates and stubs
  - Debugging made easy
  - Integrated GUI design tools
  - Integrated Testing tools

Development Environment (2)

- Source control
- Can develop anything from a console app to:
  - cloud development,
  - GUI apps,
  - web servers,
  - web apps,
  - testing frameworks,
  - Windows 8 apps,
  - Windows Phone apps,
  - Office add-ins,
  - to very large collaborative projects

TOC

1. Intro to C#
2. Hello World
3. Object-Oriented concepts
4. Collections
5. Control Flow
6. File I/O
7. Strings and REGEX
8. XML and LINQ Queries
9. Exception Handling
10. Debugging

Basics

- Using statement
- Referencing
- Namespace - see hierarchy denoted using .
  - System
  - System.Collections
  - System.Collections.Generic
- Class
- Method
- DEMO

TOC

1. Intro to C#
2. Hello World
3. Object-Oriented concepts
4. Collections
5. Control Flow
6. File I/O
7. Strings and REGEX
8. XML and LINQ Queries
9. Exception Handling
10. Debugging

Object Oriented

- Programming with classes makes for more extensible code (allowing you for instance to define your own types and control what is accessible
- Encapsulation = hide internal implementation details of a class (data hiding, this is private)
- Abstraction = public interface of a class, the external details such as actions that the class can perform (this is visible to everyone)

OO (2)

- To create an object of the type defined in a class, we need to *instantiate* that class. To be able to do so, the class must define a *constructor*:

  myClass foo = new myClass();

- Constructors can be overloaded – we can create several methods with the same name but different parameters (thus different signatures)

OO (3)

- Modifiers
  - private – only accessible inside the class
  - public – accessible by everyone
  - static – data and methods that can be accessed without creating an instance of the class
  - const – for constants
- Scope: Namespaces, classes, and methods

OO (4)

- A *method* in a class is a function – it's basically an action that an object defined by the class can take
- A *property* in a class is a data item whose read and write access is defined by you (require a get and a set:

```
public class Student
{
        //privately set
        private string _name;
        public string Name
        {
                //public read
                get
                        return _name;
        }
}
```

OO (5)

- Inheritance: a class can inherit all characteristics and actions from a parent class
- The child class can have its own private data, added methods that are not found in the parent, or redefine the behavior of inherited methods
- A class can only inherit from one parent
- DEMO

TOC

1. Intro to C#
2. Hello World
3. Object-Oriented concepts
4. Collections
5. Control Flow
6. File I/O
7. Strings and REGEX
8. XML and LINQ Queries
9. Exception Handling
10. Debugging

Arrays

```
// Single-dimensional
int[] array1 = new int[10];

// Declare and set values for a single-dimensional
array
int[] array2 = new int[] { 1, 2, 3 };

// OR
int[] array2 = { 1, 2, 3 };

// Declare a 2D array
int[,] twoDimensionalArray1 = new int[2,1];

// Declare and set 2D array
int[,] twoDimensionalArray2 = { { 1 }, { 2 } };
```

Array Initialization and Indexing

```
Class Program
{
      static void Main()
      {
            int[] array = new int[10];
            for(int i=0; i < array.Length; i++)
                  array[i] = i*i;
      }
}
```

Array Class Static Methods

- `void Clear(array, index, len)` – sets a range to null, or zero, or false, depending on the element type of the array
- `int BinarySearch(array, value, comparer)` – returns either the index of the searched value, or a negative number if the value is not found
- `void ForEach(action)` – performs the action on each element of the array

Generic Collections

- Work with built-in **and** developer-defined types (thus “Generic” – all types you define inherit from object)
- Immediate benefits: type safety
- Are defined in the System.Collections.Generic namespace
- Provide a large variety, ready for your use: Lists, Dictionaries, SortedLists, LinkedLists, SortedDictionaries, Sets, Stacks, Queues, and more…

List<T>

- Unlike an array, a List dynamically sizes as needed
- If you need sequential access to elements, use a LinkedList<T>
- Lists support adding an element or a range of elements at the end of the list (Add and AddRange)

Dictionary <Tkey, Tvalue>

- Useful methods:
  - ContainsKey,
  - ContainsValue,
  - Remove (removes the value associated with the passed key)
  - Clear
  - Add (will throw an exception if key already exists → must check if the dict contains the key you try to add)
  - TryGetValue (gets the value associated with the key you're passing to it)

More Advanced

- For future development interests:

Thread-safe collections (ConcurrentDictionary)

Readily usable in System.Threading.Tasks.Parallel

(generally the task parallel library makes multithreaded programming projects simpler to develop by taking care of some lower level details)

Control Flow

- If/else, switch, for, while, do while, are the same
- FOREACH statement – neat since we have datastructures that have an enumerator defined on them (such as lists, dicts, etc)

  foreach(var keyvalue in dictionary)

  {…}

  DEMO for data structures bundled with Strings demo

TOC

1. Intro to C#
2. Hello World
3. Object-Oriented concepts
4. Collections
5. Control Flow
6. File I/O
7. Strings and REGEX
8. XML and LINQ Queries
9. Exception Handling
10. Debugging

Strings and REGEX

- Built-in types char and string
- Empty string is String.Empty or “”
- Unassigned string is a null
- Can easily create strings from other objects by calling .ToString()
- String.Compare(string, string)
- String.Concat(string, string)
- String.Split(tokenChar)

REGEX (1)

- Regular expressions enable you to perform pattern matching
- Require using System.Text.RegularExpression
- Steps:
  - Define a pattern you wish to match (this is of type string)
  - Regex expression = new Regex(pattern);
  - expression.IsMatch(stringYouWishToMatch);

  OR
  - MatchCollection list = expression.Matches(stringYouWishToMatch);

  (this is the set of all matches by applying the regex repeatedly)

REGEX (2)

- Patterns:
  - ? matches preceding element 0 or 1
  - + matches preceding element 1 or more times
  - * Matches preceding element 0 or more
  - \d matches decimal digit (equivalent to [0-9])
  - \w matches an alphanumeric char (equiv to [a-zA-Z_0-9]
  - {p} matches preceding element exactly p times
  - {p,q} matches preceding element at least p but less than q times
  - . matches any character except \n
  - [ ] matches the character within the brackets
  - [^] NOT (e.g, [^0-9] matches everything not 0 thru 9)
  - \s matches a space

REGEX (3)

- Groups: you can designate groups within your pattern using ( )
- Example patterns:

  [0-9]*

  \d{3}\w+

  ([a-zA-Z^0-9]){3,5}(\d*(\w+))

  (\d*(\w+)?)+

FILE I/O

- Covered in Strings REGEX DEMO:
  - Path builder
  - Reading from a stream
  - Writing to a stream
  - Also featured: starting a child process and closing that process

TOC

1. Intro to C#
2. Hello World
3. Object-Oriented concepts
4. Collections
5. Control Flow
6. File I/O
7. Strings and REGEX
8. XML and LINQ Queries
9. Exception Handling
10. Debugging

XML and LINQ

- This section will show you how easy it is to create an XML parser with just a few lines of code
- LINQ is the querying mechanism (can be used for also for databases)
- We will need:
  - using System.Xml.Linq
  - using System.Xml
  - using System.Collections.Generic

XML Examples

- An XML element corresponds to the XElement type:

```
<courses>
   <course Title = “CS50” Location =
        />
   <course Title = “CS121” Location =
“Maxwell Dworkin” />
  …
</courses>
```

- Here course is an element, Title and Location are attributes. We can store them using the XElement type; moreover we can navigate attributes storing them using type XAttribute.

XML Examples

```
<students>
  <student>
    <name>Barney</name>
    <class>2013</class>
    <house>Currier</house>
  </student>
…
</students>

<!--Here we just have a simple key-value
pair arrangement and no attributes(value
of name=Barney for instance)-->
```

Setting XML elements

```
XElement xmlDocument =
      new XElement("students",
            new XElement("student",
                  new XElement("name"),
                  new XElement("class"),
                  new XElement("house")
                  )
            );
```

- Now we have to set the values of the elements we just created:

```
XElement student=xmlDocument.Element("student");
student.SetElementValue("name", "Mike");
//you can see that an XElement stores a
//key-value pair - great for storing in a dict
```

Querying XML

- You need to first set the root element of the XML:

  Root = XElement.Load("C:\Foo\bar.xml");

- To access all elements under the root node:

  Root.Elements()

- To pick the value of the first element:

  Root.Elements().First().FirstAttribute.Value

Querying XML

- LINQ query is very similar to the SQL query you're familiar with:

  FROM element in Root.Elements()

  SELECT element

  WHERE (…)

  Notice that the FROM acts like a foreach

VS 2010 and XMLs in Projects

- It is not sufficient to add the XML file to your project solution
- You must also set in the XML file's Properties tab the "Copy to Output Directory" to "Copy Always"
- This does not apply if your XML is from a URL (then you don't include a copy of it, you just pass the URL path to the Load method).

TOC

1. Intro to C#
2. Hello World
3. Object-Oriented concepts
4. Collections
5. Control Flow
6. File I/O
7. Strings and REGEX
8. XML and LINQ Queries
9. Exception Handling
10. Debugging

Exceptions (1)

- Exceptions serve as a means to report failures, classify, and handle them
- Exceptions encapsulate all the information about an error in a single class - to use exceptions add a using statement for

  using System.Exception;
- All exceptions derive from the System.Exception class

Exceptions (2)

- The System.Exception class provides several useful properties for debugging:
  - Message
  - StackTrace
  - HelpLink (optional URL)
  - Data (this is a dictionary)
  - InnerException (if an exception is wrapped into a different type of exception)
- How to “throw” an exception:

```
throw new System.Exception();
```

Exceptions (3)

- Standard exception types:
  - ArgumentException
  - ArgumentNullException
  - DivideByZeroException
  - NullReferenceException
  - InvalidOperationException
  - IndexOutOfRangeException
  - OutOfMemoryException
  - StackOverFlowException
  - SystemException

Exceptions (4) – Handling Logic

- Try-Catch block: you execute code in the Try block which may throw an exception of a specific type which you then handle in the catch block:

```
try
{
   int foo = 3/0;
}
catch(DivideByZeroException e)
{
   Console.Writeline("Why did you
                      divide by
zero?");
}
```

Exceptions (5) – Handling Logic

- Try-Catch block: you may have multiple catch blocks for different types of exceptions such that depending on the exception you execute a different block of code.
- Example:

```
catch(SystemException e1)
{ …}
catch(ArgumentException e2)
{…}
catch(InvalidOperationException e3)
{…}
```

Exceptions (6) – Handling Logic

- Catch blocks can swallow errors – if you just log an error message in your catch block and do not handle the root cause of the error, then you will fall into the bad design practice called “swallowing an exception”.
- If your application throws an exception that is unhandled by a catch block will result in an application crash.

Exceptions (7) – Handling Logic

- Try-Finally – the finally block of code executes regardless of whether or not an exception was thrown in the try block; the finally block executes immediately after the try block
- Try-Catch-Finally - may have multiple catch blocks, as before, but always a single finally block.
- DEMO

More Advanced

- IF you enter a situation in which it is unsafe to continue (security risks for example, memory corruption, etc.), instead of using an exception use System.Environment.FailFast.
- FailFast immediately terminates the application
- Such terminations are shown in the event log for the application (in the Windows event log)

1. Intro to C#
2. Basic Syntax
3. Hello World
4. Object-Oriented concepts
5. Data structures
6. Control Flow
7. File I/O
8. REGEX
9. Exception Handling
10. Debugging [DEMO]

More references

- http://msdn.microsoft.com
- Always good to use: F12 allows you to travel to the definition of a type that belongs to a referenced library

Thanks!