

# Unit 1

## Introducing C#

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 1 Overview

- C# language builds on previous C-style languages: C, C++, and Java.
- C# is used for building software components and applications.
- This Unit focuses on the fundamentals of C# syntax including variables, console I/O, and comments.
- The managed execution context is also discussed.

# Hello World Application

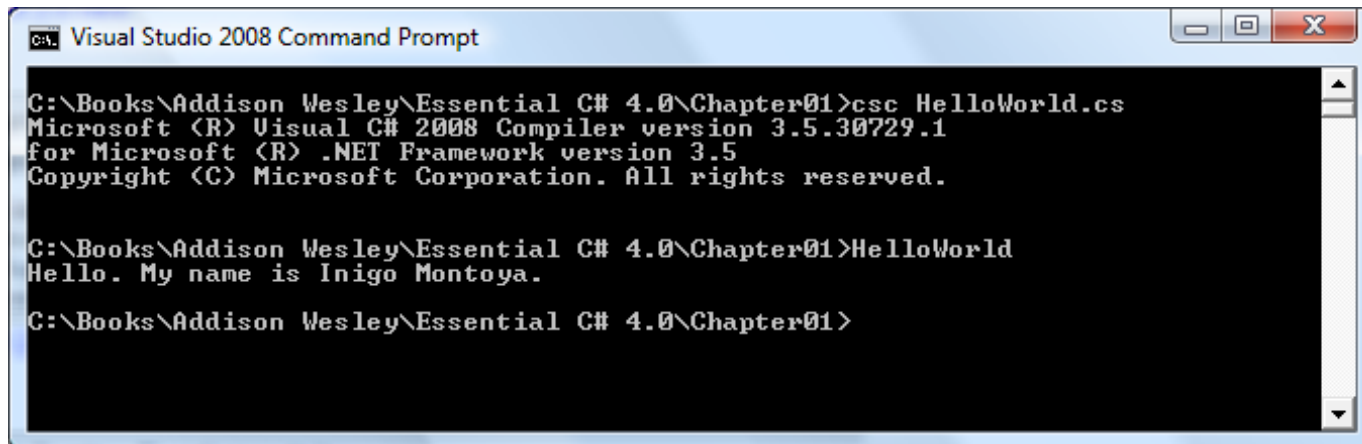
- Hello World, or some variation, is traditionally the first program written in a new language.

```
class Program
{
    class HelloWorld
    {
        static void Main()
        {
            System.Console.WriteLine(
                "Hello. My name is Inigo Montoya.");
        }
    }
}
```

# Facts about Hello World

- C# files use the “.cs” file extension:
  - For example: HelloWorld.cs
- Command-line compilation of the HelloWorld program results in the creation of an executable assembly (HelloWorld.exe).
- A class library can also be used to generate an assembly with a file extension of “\*.dll.”

# Compiling and Running the Application



```
Visual Studio 2008 Command Prompt

C:\Books\Addison Wesley\Essential C# 4.0\Chapter01>csc HelloWorld.cs
Microsoft (R) Visual C# 2008 Compiler version 3.5.30729.1
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

C:\Books\Addison Wesley\Essential C# 4.0\Chapter01>HelloWorld
Hello. My name is Inigo Montoya.

C:\Books\Addison Wesley\Essential C# 4.0\Chapter01>
```

# C# Syntax Fundamentals

- Keywords are reserved words that provide the concrete syntax that the compiler interprets.
- Examples of keywords include `class`, `static`, and `void`.
- The compiler uses keywords to identify the structure and organization of the code.
- The compiler enforces strict rules concerning the use and the location of keywords.

# C# Identifiers

- Identifiers are the names for variables used by the application.
- A variable name can be used to assign a value to a variable and then refer to it later.
- The variable names chosen by the developer should be meaningful so that the resulting code is easier to understand.
- Keywords should be avoided as identifiers.

# Type Definitions

- C# code is enclosed within a type definition, the most common type being the **class** type.
- The standard for naming types is called **Pascal casing**, capitalizing the first letter of each word: **Greetings, HelloInigoMontoya, Hello.**
- An alternative naming standard is **camel casing**, capitalizing the first letter of each word except for the first word: **quotient, firstName, theDreadPirateRoberts.**



# Class Type Example

```
class HelloWorld  
{  
  
}
```

# What is a Method?

- A method is a named block of code introduced by a method declaration and followed by zero or more statements enclosed within curly braces ({...}).
- Methods perform computations or actions.
- They provide a means to structure and organize code, avoiding the need to duplicate code.

# The Main Method

- In C#, the Main() method is the entry point into the application.
- It is always static, meaning that it is called directly through the class name.
- It must return either a void or integer value.
- It may have an **args** parameter, which is an array of strings representing command-line arguments or no parameters at all.

# Main Method Variations

```
static void Main(string[] args)
{
}
```

```
static int Main(string[] args)
{
}
```

```
static void Main()
{
}
```

```
static int Main()
{
}
```

# Line Termination

- C# uses a semicolon to end a statement.
- Examples of statements include variable declarations and method calls.
- Some programming elements do not end in a semicolon, for example, the `switch` statement or code blocks enclosed within curly braces.
- Multiple statements can be placed on one line, or a single statement can span multiple lines.

# White Space

- Whitespace is the name given to formatting characters such as tabs, spaces, and newline characters.
- The C# compiler ignores whitespace in code, but it can improve readability.
- Whitespace in a quoted string is not ignored.

# Variable Declarations

- A variable refers to a named storage location.
- Variables can be assigned a value after being declared and then used in various operations.
- The type of a variable cannot be changed once it has been declared.
- For example:  
`string max; // string type named max`
- Local variables are declared inside methods.

# Data Types

- The type of data specified in the variable declaration is called the data type.
- Examples of data types include the following:
  - `int` is an integer type (32-bits in size);
  - `char` is a character type (16-bits in size).
- A data type is, therefore, a way to classify data having similar characteristics.



# Variable Naming

- The name of a variable must begin with a letter or underscore followed by any number of letters, numbers, and/or underscores.
- By convention, local variables are camel-cased and do not include underscores.

# Variable Declaration and Assignment

- After declaring a variable, a local variable must be assigned a value before it is accessed.
- A variable can be assigned a value in the variable declaration or in a separate statement.

```
string valerie;  
string max = "Have fun storming the castle!";  
  
valerie = "Think it will work?";
```

# String Immutability

- String types are immutable, meaning not modifiable.
- A new literal can be assigned to a string variable, but this reassigns the variable to a new location in memory.

# Console Input and Output

- Console Input/Output (I/O) involves reading and writing from the console window.
- The console is an operating system window where users interact with a text-based application.
- The Console class provides basic support for applications that read characters from, and write characters to, the console.

# Example of Console I/O

```
static void Main()
{
    string firstName;
    string lastName;

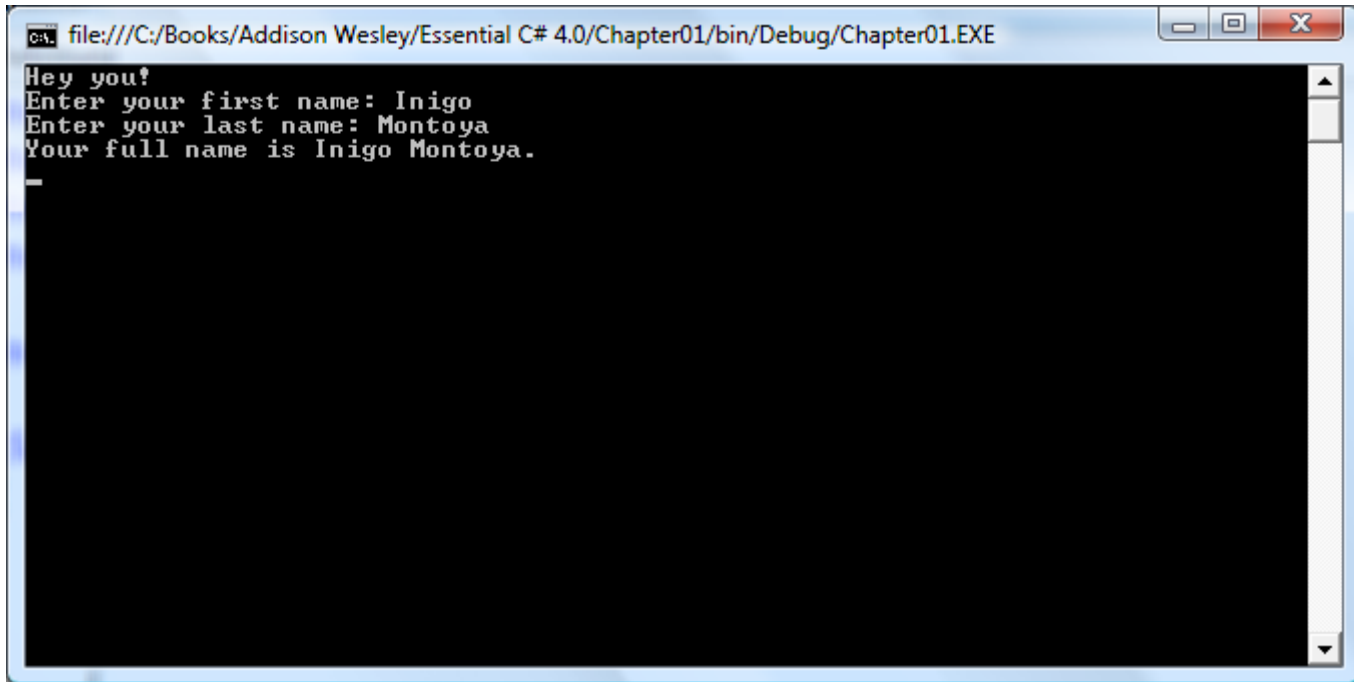
    System.Console.WriteLine("Hey you!");

    System.Console.Write("Enter your first name: ");
    firstName = System.Console.ReadLine();

    System.Console.Write("Enter your last name: ");
    lastName = System.Console.ReadLine();

    System.Console.WriteLine("Your full name is {0} {1}.",
        firstName, lastName);
}
```

# Example of Console I/O (cont'd)



A screenshot of a Windows console window. The title bar shows the file path: `file:///C:/Books/Addison Wesley/Essential C# 4.0/Chapter01/bin/Debug/Chapter01.EXE`. The console text is as follows:

```
Hey you!  
Enter your first name: Inigo  
Enter your last name: Montoya  
Your full name is Inigo Montoya.  
_
```

The text is displayed in a monospaced font on a black background. A small white cursor is visible on the line following the program's output.

# Reading from the Console

- `Console.Read(...)` reads the next character from the standard input stream.
- `Console.ReadKey(...)` obtains the next character or function key pressed by the user.
- `Console.ReadLine(...)` reads the next line of characters from the standard input stream.

# Writing to the Console

- `Console.Write(...)` writes the text representation of the specified value or values to the standard output stream, without a line terminator.
- `Console.WriteLine(...)` writes the specified data, followed by a line terminator, to the standard output stream.



# Comments

- Comments are used to describe and explain the code being written, for example, if the syntax is difficult to understand.
- The compiler ignores comments and generates an assembly that removes the comments from the original source code.

# Types of Comments

Comment Type	Description	Example
Delimited comments	Comments of this type may span multiple lines or appear embedded within a line of code.	<code>/* comment */</code>
Single-line comments	The compiler treats all text from the delimiter to the end of the line as a comment.	<code>// comment</code>
XML-delimited comments	These have the same characteristics as regular delimited comments except the compiler can save them into a separate XML text file.	<code>/**comment**/</code>
XML single-line comments	The compiler can also save these comments into a separate XML text file.	<code>///comment</code>

# Extensible Markup Language (XML)

- XML is a simple, flexible text format designed to describe the structure of data in an exchangeable format.

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  <book title=" 4.0">
    <Units>
      <Unit title="Introducing C#"/>
      <Unit title="Operators and Control Flow"/>
    </Units>
  </book>
</body>
```

# C# Compilation

- The C# compiler transforms the C# source code file into intermediate language (CIL).
- At runtime, the final compilation step, just-in-time (JIT) compilation, takes place, translating the intermediate language into platform-specific machine code.
- The Common Language Runtime (CLR) executes this final version of the program, a process called managed execution.

# Common Language Infrastructure (CLI)

- An international standard that includes specifications for the following items:
  - The Common Language Runtime (CLR)
  - Common Intermediate Language (CIL)
  - Common Type System (CTS)
  - Common Language Specification (CLS)
  - Metadata supporting CLI services
  - Base Class Library (BCL).

# CLI Features

- Language interoperability
- Type safety
- Code access security
- Garbage collection
- Platform portability
- Base Class Library

# C# and .NET Versioning

C#/.NET Version	Description
C# 1.0 with .NET Framework 1.0/1.1 (Visual Studio 2002 and 2003)	The initial release of C#.
C# 2.0 with .NET Framework 2.0 (Visual Studio 2005)	Generics were added.
.NET Framework 3.0	Windows Communication Foundation (WCF), Windows Presentation Foundation (WPF), Windows Workflow (WF), and CardSpace (web authentication).
C# 3.0 with .NET Framework 3.5 (Visual Studio 2008)	Added support for LINQ.
C# 4.0 with .NET Framework 4.0 (Visual Studio 2010)	Added support for dynamic typing.

# Common Intermediate Language

- Common Intermediate Language (CIL) is the lowest-level programming language in the Common Language Infrastructure (CLI) and in the .NET Framework.
- ildasm.exe, a CIL disassembler utility, creates a viewable CIL representation of an assembly or executable.



# Unit 1 Summary

- This Unit served as a rudimentary introduction to C#.
- C# resembles C, C++, and Java in many ways.
- C# supports object-oriented programming methodology.
- The next Unit examines the fundamental data types that are part of the C# language.

# Unit 2

## Data Types

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 2 Overview

- This Unit builds on the discussion of C# basics from Unit 1 by investigating C# data types.
- Predefined types (or primitives) include integer, floating-point, decimal, datetime, boolean, and character types.
- The string and array types are also discussed.

# Fundamental Numeric Types

- Basic numeric types in C# include integer, floating-point, and decimal types.
- All types in C# have a short name and a long name (fully-qualified class name):
  - For example: `Int32` (short for `System.Int32`).
- In addition, numeric types have an alias assigned to them:
  - For example: `bool` (alias for `System.Boolean`)

# C# Integer Types

Type	Size	Range (Inclusive)	BCL Name	Signed
sbyte	8 bits	-128 to 127	System.SByte	Yes
byte	8 bits	0 to 255	System.Byte	No
short	16 bits	-32,768 to 32,767	System.Int16	Yes
ushort	16 bits	0 to 65,535	System.UInt16	No
int	32 bits	-2,147,483,648 to 2,147,483,647	System.Int32	Yes
uint	32 bits	0 to 4,294,967,295	System.UInt32	No
long	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	System.Int64	Yes
ulong	64 bits	0 to 18,446,744,073,709,551,615	System.UInt64	No

# C# Floating-Point Types

Type	Size	Range (Inclusive)	BCL Name	Significant Digits
float	32 bits	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{+38}$	System.Single	7
double	64 bits	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{+308}$	System.Double	15-16

# C# Decimal Type

Type	Size	Range (Inclusive)	BCL Name	Significant Digits
<code>decimal</code>	128 bits	$1.0 \times 10^{-28}$ to approximately $7.9 \times 10^{+28}$	System. Decimal	28-29

# Floating-Point vs. Decimal Types

- Floating-point numbers can represent very large numbers but introduce round-off error.
- Decimal numbers represent numbers exactly, with no round-off error, but have a smaller range.
- Decimal numbers are better for financial calculations, which must be precise; floating point numbers are better for scientific calculations.



# Literal Values

- A numeric literal is the representation of a fixed value within source code.
- Hardcoded values generally default to **int** or **double** data types, depending on the presence of a decimal point.
- Suffixes can be appended to numeric literals to coerce them to a particular data type:
  - For example, the **f** suffix indicates a **float** type.

```
System.Console.WriteLine(6.023E23f);
```

# Numeric Suffixes

Suffix Type	Numeric Suffix	Example
unsigned <code>int</code>	U or u	<code>uint</code> x = 100U;
<code>long</code>	L or l	<code>long</code> x = 100L;
unsigned <code>long</code>	UL or ul	<code>ulong</code> x = 100UL;
<code>float</code>	F or f	<code>float</code> x = 100F;
<code>double</code>	D or d	<code>double</code> x = 100D;
<code>decimal</code>	M or m	<code>decimal</code> x = 100M;

# Hexadecimal Notation

- Numbers can be represented in base16 format, meaning sixteen symbols are used (0-9 and A-F) instead of ten symbols (0-9).
  - 0x000A corresponds to the decimal value 10.
  - 0x002A corresponds to the decimal value 42.
- Numeric values can be displayed in hexadecimal format using the format specifier x (or X):

```
System.Console.WriteLine("0x{0:X}",42);    // Displays "0x2A"
```

# Round-Trip Formatting

- The round-trip format specifier, r or R, returns a numeric-valued string which can be converted back to its original numeric value.

# bool Data Type

- The `bool` (System.Boolean) data type represents true and false values.
- It is stored in a single byte and can be used in conditional statements and expressions.

```
bool married = false;

if (married)
    System.Console.WriteLine("You are married!");
else
    System.Console.WriteLine("You are not married!");
```

# char Data type

- The `char` (System.Char) data type represents the 16-bit Unicode character set.
- It is stored in two bytes and is used to display and manipulate individual letters, digits, symbols, and special characters.

```
char grade = 'A';
```

# The Unicode Standard

- Unicode is an international standard for representing the characters found in most languages.
- This standard enables applications to be localized, i.e., to display the appropriate language and cultural characteristics for different locales.

# Unicode Characters

- Literal characters are enclosed within single quotes, 'A', when inserted into code.
- Sometimes, characters cannot be inserted directly and must be prefixed with a backslash character: for example, '\n' represents a newline character; '\t' represents a tab.
- Any Unicode character can be represented by prefixing it with the special character '\u', followed by its hexadecimal value.



# Unicode Escape Characters

Escape Sequence	Character Name	Unicode Encoding
\'	Single quote	\u0027
\"	Double quote	\u0022
\\	Backslash	\u005C
\0	Null	\u0000
\a	Alert (system beep)	\u0007
\b	Backspace	\u0008
\f	Form feed	\u000C
\n	Newline	\u000A
\r	Carriage return	\u0009
\v	Vertical tab	\u000B
\uxxxx	Unicode character in hex	\u0029 (example)

# string Data type

- The **string** type, `System.String`, is a fundamental data type for representing sequences of characters.
- String literals can be inserted into code by placing text within double quotes.

```
static void Main()
{
    System.Console.WriteLine("\"Truly, you have a dizzying intellect.\"");
    System.Console.WriteLine("\n\"Wait 'til I get going!\"\n");
}
}
```

# String Methods

- String methods can be either **static** (called through the class name) or instance (called through a reference variable) methods.

# String Static Method Examples

Static Method	Example
<code>static void string.Format(...)</code>	<pre>string text, firstName, lastName; ... text = string.Format("Your full name is {0} {1}.",                     firstName, lastName);</pre>
<code>static void string.Concat(...)</code>	<pre>string text, firstName, lastName; ... text = string.Concat(firstName, lastName);</pre>
<code>static void string.Compare(...)</code>	<pre>string option; ... // String comparison in which case matters. int result = string.Compare(option, "/help");</pre>
<code>static void string.Compare(...)</code>	<pre>// Case-insensitive string comparison ... int result = string.Compare(option, "/Help", true);</pre>

# String Instance Method Examples

Instance Method	Example
<code>bool StartsWith(...), EndsWith(...)</code>	<code>bool isPhd = lastName.EndsWith("Ph.D."); bool isDr = lastName.StartsWith("Dr.");</code>
<code>string ToLower(), ToUpper()</code>	<code>string severity = "warning"; <i>// Display the severity in uppercase</i> System.Console.WriteLine(severity.ToUpper());</code>
<code>string Trim(), Trim(...), TrimEnd(), TrimStart()</code>	<code><i>// Remove any whitespace at the start or end.</i> username = username.Trim();</code>
<code>string Replace(...)</code>	<code>string filename; ... <i>// Remove ?'s altogether from the string</i> filename = filename.Replace("?", "");</code>

# String Length Property

- The Length member of the string class is actually a read-only property.
- A property is a special type of method that returns and/or sets an underlying value.

```
static void Main()
{
    System.Console.Write("Enter a palindrome: ");

    string palindrome = System.Console.ReadLine();

    System.Console.WriteLine("The palindrome, \"{0}\" is {1} characters.",
                             palindrome, palindrome.Length);
}
```

# String Immutability

- Immutability means that a string value cannot be modified without first creating a new copy of the original string value.
- The `System.Text.StringBuilder` class can be used, instead, to modify strings without creating a new copy for each modification.
- `StringBuilder` methods include `Append()`, `AppendFormat()`, `Insert()`, `Remove()`, and `Replace()`.

# Null Strings

- The **null** value means that a string variable or string literal has no value.
- Only reference types (and pointer types) may contain the null value.
- Code that sets a string variable to null assigns the string variable to reference nothing at all.
- Note that assigning a null value is different from assigning an empty string ("" ) to a string variable.



# void Return Value

- The **void** type (System.Void) is used to identify a method that returns no value.
- The Main() function in the Program class is an example of the void return value.

```
class Program
{
    static void Main()
    {
    }
}
```

# Implicit Typing

- C# 3.0 introduced a contextual keyword, `var`, for declaring a variable without explicitly specifying its data type.
- Instead, the data type is implied from the assignment made to the variable.

```
System.Console.WriteLine("Enter text: ");  
var text = System.Console.ReadLine();  
  
// Return a new string in uppercase  
var uppercase = text.ToUpper();
```

# Categories of Data Types

- All types fall into two categories: value types and reference types.
- Value types contain their values directly; i.e., the variable refers to the same location in memory where the value is stored.
- Reference types do not contain their values but instead a reference (pointer) to their values stored elsewhere (on the heap).

# Copying Semantics

- Assigning one value-type variable to another copies the original **value** to the destination variable; both variables will reference different memory locations but the same value.
- Assigning one reference-type variable to another copies the original **reference** to the destination variable; both variables reference the same memory location.

# Nullable Modifier

- Value types, such as integers, cannot accept the **null** value.
- To declare value-type variables that can store null values, the nullable modifier is used.
  - ➔ **int?** count = null;
- Assigning null to value types is useful when doing database programming because database tables may contain fields that allow null values.

# Data Type Conversions

- Any numeric conversion that could result in a loss of data requires an explicit cast; otherwise, an implicit cast is sufficient.
- An explicit cast requires the use of a cast operator, i.e., an operator specifying (within parentheses) the target type.

```
int x;  
long y = 100;
```

```
x = (int)y; // Explicit conversion requires a cast operator (int)  
y = x;     // Implicit conversion requires no cast operator
```

# Arithmetic Overflow

- By default, if the result of a calculation is too large for the target data type, an arithmetic overflow occurs with no exception thrown.
- Overflowing an Integer Value:

```
public static void Main()
{
    // int.MaxValue equals 2147483647
    int n = int.MaxValue;

    n = n + 1 ;
    System.Console.WriteLine(n); // Displays -2147483648
}
```

# checked Keyword

- The **checked** keyword throws an `OverflowException` when a calculation causes an arithmetic overflow to occur.

```
public static void Main()
{
    checked
    {
        // int.MaxValue equals 2147483647
        int n = int.MaxValue;

        n = n + 1 ; // Throws OverflowException here
        System.Console.WriteLine(n);
    }
}
```



# unchecked Keyword

- The **unchecked** keyword prevents the runtime from throwing an `OverflowException` during execution, which is actually the default.

```
public static void Main()
{
    unchecked
    {
        // int.MaxValue equals 2147483647
        int n = int.MaxValue;

        n = n + 1 ; // No OverflowException here
        System.Console.WriteLine(n);
    }
}
```

# The Parse() Method

- Conversion from a string type to a numeric type requires the use of the Parse() method, which is defined by each numeric data type.
- Converting a string to a **float** data type:

```
string text = "9.11E-31";  
float kgElectronMass = float.Parse(text);
```

# System.Convert

- A special type containing static methods for converting from any primitive type to any other primitive type.

```
string middleCText = "278.4375";  
double middleC = System.Convert.ToDouble(middleCText);  
bool boolean = System.Convert.ToBoolean(middleC);
```

# ToString() Method

- The ToString() method provides a string representation of a numeric type.

```
bool boolean = true;  
string text = boolean.ToString();  
  
// Display "True"  
System.Console.WriteLine(text);
```

# TryParse(...) Method

- The TryParse() method is similar to the Parse() method except that if the conversion fails, false is returned instead, not an exception.

```
double number;  
string input;  
  
System.Console.WriteLine("Enter a number: ");  
input = System.Console.ReadLine();  
  
if (double.TryParse(input, out number))  
{  
    // Converted correctly, now use number  
}
```

# Arrays

- An array stores multiple items of the same type using a single variable reference and accesses them through an index.
- Arrays are **zero-based** in C#; i.e., their indexes begin at 0 and end at the array length – 1.
- Arrays are declared by first specifying the element type of the array, followed by square brackets and then the name of the variable:

```
string[] languages;
```

# Array Declaration and Initialization

- The runtime initializes arrays to their default values: zero for numeric types; false for boolean types; binary zero ('\0') for character types; and null for reference types.
- Arrays can also be initialized using a comma-delimited list of items enclosed within braces.
- A two-dimensional array is declared as follows:  
    ➔ **int[,] cells;**
- An array's **rank** is the number of dimensions.

# Array Initialization Examples

// Array Declaration with Assignment

```
string[] languages = { "C#", "COBOL", "Java", "C++",  
                      "Visual Basic", "Pascal", "Fortran", "Lisp", "J#" };
```

// Array Assignment Following Declaration

```
string[] languages;  
languages = new string[] { "C#", "COBOL", "Java", "C++",  
                          "Visual Basic", "Pascal", "Fortran", "Lisp", "J#" };
```

// Array Assignment with new during Declaration

```
string[] languages = new string[] { "C#", "COBOL", "Java", "C++",  
                                   "Visual Basic", "Pascal", "Fortran", "Lisp", "J#" };
```

// Declaration and Assignment with the new Keyword

```
string[] languages = new string[9] { "C#", "COBOL", "Java", "C++",  
                                   "Visual Basic", "Pascal", "Fortran", "Lisp", "J#" };
```

// Assigning without Literal Values

```
string[] languages = new string[9];
```



# Array Initialization Examples (cont'd)

// Defining the Array Size at Runtime

```
string[] groceryList;  
System.Console.WriteLine("How many items on the list? ");  
int size = int.Parse(System.Console.ReadLine());  
groceryList = new string[size];
```

//Declaring a Two-Dimensional Array

```
int[,] cells = int[3,3];
```

//Initializing a Two-Dimensional Array of Integers

```
int[,] cells = { {1, 0, 2},  
                 {1, 2, 0},  
                 {1, 2, 1} };
```

# Jagged Arrays

- The jagged array is an array of arrays; it does not need to be consistently sized.

```
// Initializing a Jagged Array
```

```
int[][] cells = {  
    new int[]{1, 0, 2, 0},  
    new int[]{1, 2, 0},  
    new int[]{1, 2},  
    new int[]{1}  
};
```

```
cells[1][0] = 1;
```

# Using an Array

- Access a specific item in an array using the array accessor (square brackets) and the index number.

# Using an Array Examples

```
// Declaring and Accessing an Array
```

```
string[] languages = new string[9]{"C#", "COBOL", "Java","C++",  
                                   "Visual Basic", "Pascal","Fortran", "Lisp", "J#"};
```

```
// Retrieve 3rd item in languages array (Java)
```

```
string language = languages[4];
```

```
// Swapping Data between Positions in an Array
```

```
// Save "C++" to variable called language.
```

```
language = languages[3];
```

```
// Assign "Java" to the C++ position.
```

```
languages[3] = languages[2];
```

```
// Assign language to location of "Java".
```

```
languages[2] = language;
```

```
//Initializing a Two-Dimensional Array of Integers
```

```
int[,] cells = { { 1, 0, 2 }, { 0, 2, 0 }, { 1, 2, 1 } };
```

```
// Set the winning tic-tac-toe move to be player 1.
```

```
cells[1, 0] = 1;
```

# Length of an Array

- Arrays have a fixed length (accessed through the Length property) that cannot be changed without recreating the array.
- Overstepping the bounds of an array, i.e., attempting to access an element outside the range of the array, causes a `System.IndexOutOfRangeException` to be thrown.

# Array Length Examples

```
// Retrieving the Length of an Array
Console.WriteLine("There are {0} languages in the array.", languages.Length);

// Accessing Outside the Bounds of an Array, Throwing an Exception
string languages = new string[9];
...
// RUNTIME ERROR: index out of bounds – should be 8 for the last element
languages[4] = languages[9];
```

# Static Array Methods

```
// Static Array Methods
```

```
System.Array.Sort(languages);
```

```
string searchString = "COBOL";
```

```
int index = System.Array.BinarySearch(languages, searchString);
```

```
System.Console.WriteLine("The wave of the future, {0}, is at index {1}.",  
                          searchString, index);
```

```
System.Console.WriteLine();
```

```
System.Console.WriteLine("{0,-20}\t{1,-20}", "First Element", "Last Element");
```

```
System.Console.WriteLine("{0,-20}\t{1,-20}", "-----", "-----");
```

```
System.Console.WriteLine("{0,-20}\t{1,-20}", languages[0],  
                          languages[languages.Length - 1]);
```

```
System.Array.Reverse(languages);
```

```
System.Console.WriteLine("{0,-20}\t{1,-20}", languages[0],  
                          languages[languages.Length - 1]);
```

# More Static Array Methods

```
// Note this does not remove all items from the array.  
// Rather it sets each item to the type's default value.  
System.Array.Clear(languages, 0, languages.Length);  
System.Console.WriteLine("{0,-20}\t{1,-20}",  
                           languages[0], languages[languages.Length - 1]);  
System.Console.WriteLine("After clearing, the array size is: {0}",  
                           languages.Length);
```



# Notes about Static Array Methods

- Before using the `BinarySearch()` method, the array must be sorted. If the returned value is negative, the complement operator (`~`) returns the index of the next highest value.
- The `Clear()` method sets the elements to their default values; it does not set the length to 0.
- The `Array.Resize()` method also does not resize the array but creates a new resized array and then copies all the elements to the new array.

# Array Instance Members

- Arrays also have instance members (accessed through an array variable, not the data type).
- Length is an instance property, for example.
- Rank returns the number of dimensions.
- Clone() returns a copy of the array.
- GetLength() retrieves the length (number of elements) of a particular dimension.

```
//Retrieving a Particular Dimension's Size
```

```
bool[, ,] cells = new bool[2, 3, 3];
```

```
System.Console.WriteLine(cells.GetLength(0)); // Displays 2
```

# Strings as Arrays

- Under the covers, strings are stored as character arrays.
- Characters can be accessed individually.
- The entire string can be returned as a array using the `ToCharArray()` method.

```
// Looking for Command-Line Options (Simplified)
// Accessing individual characters in a string
string arg = args[0];
if (arg[0] == '-')
{ //This parameter is an option }
```

# String as Character Array Example

```
// Palindrome: Reversing a string
string reverse, palindrome;
char[] temp;

System.Console.WriteLine("Enter a palindrome: ");
palindrome = System.Console.ReadLine();

reverse = palindrome.Replace(" ", "");
reverse = reverse.ToLower();
temp = reverse.ToCharArray();
System.Array.Reverse(temp);

// Convert the array back to a string; check if reverse string is the same.
if (reverse == new string(temp))
    System.Console.WriteLine("{0} is a palindrome.", palindrome);
else
    System.Console.WriteLine("{0} is NOT a palindrome.", palindrome);
```

# Common Array Errors

Common Mistake	Error Description	Corrected Code
<code>int numbers[];</code>	The square braces for declaring an array appear after the data type, not after the variable identifier.	<code>int[] numbers;</code>
<code>int[] numbers; numbers = {42, 84, 168 };</code>	When assigning an array after declaration, it is necessary to use the <code>new</code> keyword and then the data type.	<code>int[] numbers; numbers = new int[]{ 42, 84, 168 }</code>
<code>int[3] numbers = { 42, 84, 168 };</code>	It is not possible to specify the array size as part of the variable declaration.	<code>int[] numbers = { 42, 84, 168 };</code>
<code>int[] numbers = new int[];</code>	The array size is required at initialization time unless an array literal is provided.	<code>int[] numbers = new int[3];</code>
<code>int[] numbers = new int[3]{};</code>	The array size is specified as 3, but there are no elements in the array literal. The array size must match the number of elements in the array literal.	<code>int[] numbers = new int[3] { 42, 84, 168 };</code>

# Common Array Errors (cont'd)

Common Mistake	Error Description	Corrected Code
<pre>int[] numbers = new int[3]; Console.WriteLine( numbers[3]);</pre>	Array indexes start at zero. Therefore, the last item is one less than the array size. (Note that this is a runtime error, not a compile-time error.)	<pre>int[] numbers = new int[3]; Console.WriteLine( numbers[2]);</pre>
<pre>int[] numbers = new int[3]; numbers[ numbers.Length] =42;</pre>	Same as previous error: 1 needs to be subtracted from the Length to access the last element. (Note that this is a runtime error, not a compile-time error.)	<pre>int[] numbers = new int[3]; numbers[ numbers.Length-1] =42;</pre>
<pre>int[] numbers; Console.WriteLine( numbers[0]);</pre>	numbers has not yet been assigned to an instantiated array, and therefore, it cannot be accessed.	<pre>int[] numbers = {42, 84}; Console.WriteLine( numbers[0]);</pre>

# Common Array Errors (cont'd)

Common Mistake	Error Description	Corrected Code
<code>int[,] numbers = { {42},{84, 42} };</code>	Multidimensional arrays must be structured consistently.	<code>int[,] numbers = { {42, 168},{84, 42} };</code>
<code>int[][] numbers = { {42, 84}, {84, 42} };</code>	Jagged arrays require instantiated arrays to be specified for the arrays within the array.	<code>int[][] numbers = { new int[]{42, 84},   new int[]{84, 42} };</code>

# Unit 2 Summary

- This Unit discusses the C# data types and their uses.
- Next, implicit and explicit casting between different data types was reviewed.
- The Unit closed with a discussion of arrays and the various means of manipulating them.
- The next Unit looks at C# expressions and control statements.



# Unit 3

## Operators and Control Flow

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 3 Overview

- This Unit introduces C# operators used in assignments and arithmetic operations.
- Operators provide syntax for calculations and actions.
- Control flow statements provide syntax for conditional logic and looping operations.
- The Unit ends with a discussion of the C# preprocessor and its directives

# Operators

- **Operators** specify operations within an expression, such as a mathematical expression to be performed on a set of values, called **operands**, to produce a new value or result.
- Operators are broken down into **unary**, **binary**, and **ternary** operators, depending on the number of operands they require (one, two, or three, respectively).

```
// A Simple Operator Example  
int difference = 4 - 2; // "-" is a binary operator
```

# Arithmetic Binary Operators

- Arithmetic binary operators include addition (+), division (/), multiplication (\*), subtraction (-), and remainder (%).
- The operands appear on each side of the arithmetic operator, and the value is assigned.
- The expression is evaluated from left to right.

```
// Using Binary Operators
```

```
int numerator, denominator, quotient, remainder;  
quotient = numerator / denominator;  
remainder = numerator % denominator;
```

# Order of Precedence

- The **order of precedence** determines which operators are evaluated first.
- The multiplication operator is evaluated before the plus operator, for example:  
→  $a + b * c$  is equivalent to  $a + (b * c)$
- Arithmetic order of precedence
  - $*, /, \%$
  - $+, -$
  - $=$

# Associativity

- **Associativity** refers to how operands are grouped and, therefore, the order in which they are evaluated.
- For example, given an operator that appears more than once in an expression, the first pair of operands are associated and then the next, until all operators have been evaluated.
  - $a-b-c$  associates as  $(a-b) - c$

# Other Examples of '+' Operator

```
// Using plus operator with strings (concatenation)
short windSpeed = 42;
System.Console.WriteLine("The original Tacoma Bridge in Washington\nwas "
    + "brought down by a "
    + windSpeed + " mile/hour wind.");
```

```
// Using the + Operator with the char data type
// The digit 3 contains a Unicode value of 0x33.
// The digit 4 contains a Unicode value of 0x34.
// Adding 3 and 4 in a Unicode value of 0x167.
int n = '3' + '4';
char c = (char)n;
System.Console.WriteLine(c); // Writes out 'g'.
```

# Floating-Point Characteristics

- Precision refers to the number of significant digits a numeric data type can hold, e.g., a **float** can hold seven significant digits.
- Inaccurate calculations can result, if precision is not taken into account:
  - $1,234,567f + 0.1234567f = 1,234,567f$
- Also, when comparing floating point numbers for equality, discrepancies can occur, unless equality evaluations include a tolerance.



# Some Floating-Point Inaccuracies

```
decimal decimalNumber = 4.2M;  
double doubleNumber1 = 0.1F * 42F, doubleNumber2 = 0.1D * 42D;  
float floatNumber = 0.1F * 42F;  
  
// Displays: 4.2 != 4.20000006258488  
System.Console.WriteLine("{0} != {1}", decimalNumber, (decimal)doubleNumber1);  
  
// Displays: 4.2 != 4.20000006258488  
System.Console.WriteLine("{0} != {1}", (double)decimalNumber, doubleNumber1);  
  
// Displays: (float)4.2M != 4.2F  
System.Console.WriteLine("(float){0}M != {1}F", (float)decimalNumber, floatNumber);  
  
// Displays: 4.20000006258488 != 4.20000028610229  
System.Console.WriteLine("{0} != {1}", doubleNumber1, (double)floatNumber);  
  
// Displays: 4.20000006258488 != 4.2  
System.Console.WriteLine("{0} != {1}", doubleNumber1, doubleNumber2);
```

# More Floating-Point Characteristics

- Dividing a floating-point number by zero results in a value of 'NaN' ((N)ot (a) (Number)), unlike an integer (undefined).
- A floating-point number that overflows its bounds results in a value of “positive infinity” ('Infinity') or “negative infinity” ('-Infinity').
- If the floating-point value contains a value close to zero, it can be represented as “positive zero” (0) or “negative zero” (-0).

# Floating-Point Examples

```
// Dividing a Float by Zero, Displaying NaN
```

```
float n = 0f;
```

```
// Displays: NaN
```

```
System.Console.WriteLine(n / 0);
```

```
// Overflowing the Bounds of a float
```

```
// Displays: -Infinity
```

```
System.Console.WriteLine(-1f / 0);
```

```
// Displays: Infinity
```

```
System.Console.WriteLine(3.402823E+38f * 2f);
```

# Parenthesis Operator

- Parentheses allow the grouping of operands and operators so that they can be evaluated together
- They provide a means to override the default order of precedence:
  - $(60 / 10) * 2 = 12$
  - $60 / (10 * 2) = 3$
- Parentheses can make code more readable, simply clarifying the order of precedence.

# Combination Assignment and Increment/Decrement Operators

- Combination assignment operators include `+=`, `-=`, `/=`, `*=`, and `%=`.
- The increment operator, `++`, increments a variable by one each time it is used.
- The decrement operator, `--`, decrements a variable by one each time it is used.

# Combination Assignment and Increment/Decrement Examples

// Assignment Operator Examples

```
int x = 0;
```

```
x += 2; x -= 2; x /= 2; x *= 2; x %= 2;
```

// Increment Operator

```
int spaceCount = 0;
```

```
spaceCount = spaceCount + 1;
```

```
spaceCount += 1;
```

```
spaceCount++;
```

// Decrement Operator

```
int lines = 0;
```

```
lines = lines - 1;
```

```
lines -= 1;
```

```
lines--;
```

# A Longer Decrement Example

```
//Displaying Each Character's ASCII Value in Descending Order
char current;
int asciiValue;

// Set the initial value of current.
current = 'z';
do
{
    // Retrieve the ASCII value of current.
    asciiValue = current;
    System.Console.WriteLine("{0}={1}\\t", current, asciiValue);

    // Proceed to the previous letter in the alphabet;
    current--;
}
while (current >= 'a');
```

# Pre-Increment and Post-Increment Operators

- Where the increment or decrement operator is placed determines how the code performs.
- If the increment or decrement operator appears before the operand, the value returned is assigned to the variable **after** the increment, e.g.,  $\rightarrow y = ++x;$
- If the operator appears after the operand, the value returned does not include the increment or decrement operation, e.g.,  $\rightarrow y = x++;$



# Some Examples of Increment Operators

```
// Using the Post-Increment Operator
// result = 0 and count = 1
int count;
int result;
count = 0;
result = count++;
System.Console.WriteLine("result = {0} and count = {1}",
                          result, count);
```

```
// Using the Post-Increment Operator
// result = 1 and count = 1
count = 0;
result = ++count;
System.Console.WriteLine("result = {0} and count = {1}",
                          result, count);
```

# More Examples of Increment Operators

```
//Comparing the Prefix and Postfix Increment Operators
```

```
int x = 1;
```

```
// Display 1, 2, 3
```

```
System.Console.WriteLine("{0}, {1}, {2}", x++, x++, x);
```

```
// x now contains the value 3.
```

```
// Display 4, 5, 5
```

```
System.Console.WriteLine("{0}, {1}, {2}", ++x, ++x, x);
```

```
// x now contains the value 5.
```

# Constant Expressions

- A constant expression contains literal values which are evaluated at compile time and cannot be changed when the program runs, e.g., the mathematical constant  $\pi$ .
- A constant is declared with the keyword `const`.

# Control Flow Statements

Statement	General Syntax Structure	Example
if	if (boolean-expression) embedded-statement	<i><b>if</b> (input == "quit") {     System.Console.WriteLine("Game end");     <b>return</b>; }</i>
	if (boolean-expression) embedded-statement <b>else</b> embedded-statement	<i><b>if</b> (input == "quit") {     System.Console.WriteLine("Game end");     <b>return</b>; } <b>else</b>     GetNextMove();</i>

# Control Flow Statements

Statement	General Syntax Structure	Example
<b>while</b>	<b>while</b> (boolean-expression) embedded-statement	<b><i>while</i></b> ( <i>count &lt; total</i> ) { <i>System.Console.WriteLine</i> ( " <i>count = {0}</i> ", <i>count</i> ); <i>count++</i> ; }
<b>do while</b>	<b>do</b> embedded-statement <b>while</b> (boolean-expression);	<b><i>do</i></b> { <i>System.Console.WriteLine</i> ("Enter name:"); <i>input =System.Console.ReadLine</i> (); } <b><i>while</i></b> ( <i>input != "exit"</i> );

# Control Flow Statements

Statement	General Syntax Structure	Example
<b>for</b>	<b>for</b> (for-initializer; boolean-expression; for-iterator) embedded-statement	<i><b>for</b> (int count = 1; count &lt;= 10; count++)</i> <i>{</i> <i>    System.Console.WriteLine(</i> <i>        "count = {0}", count);</i> <i>}</i>
<b>foreach</b>	<b>foreach</b> (type identifier in expression) embedded-statement	<i><b>foreach</b> (char letter in email)</i> <i>{</i> <i>    System.Console.Write(letter);</i> <i>}</i>

# Control Flow Statements

Statement	General Syntax Structure	Example
<b>continue</b>	<b>continue;</b>	<pre><i>foreach</i> (char letter in email) {     <i>if</i>(!insideDomain)     {         <i>if</i> (letter == '@')         {             insideDomain = <b>true</b>;         }         <b>continue</b>;     }     System.Console.Write(letter); }</pre>

# Control Flow Statements

Statement	General Syntax Structure	Example
<b>switch</b>	<b>switch</b> (governing-type-expression) { <b>case</b> const-expression: statement-list jump-statement <b>default</b> : statement-list jump-statement }	<pre> <b>switch</b>(input) {     <b>case</b> "exit":     <b>case</b> "quit":         System.Console.WriteLine(             "Exiting app....");          <b>break</b>;     <b>case</b> "restart":         Reset();         <b>goto case</b> "start";     <b>case</b> "start":         GetMove();         <b>break</b>;     <b>default</b>:         System.Console.WriteLine(input);         <b>break</b>; } </pre>
<b>break</b>	<b>break</b> ;	
<b>goto</b>	<b>goto</b> identifier;	
	<b>goto</b> case const-expression;	
	<b>goto</b> default;	



# if/else Statement

- The if statement evaluates a Boolean (true/false) expression.
- If the result is true, the following code block (or statement) is executed; otherwise, the else code block is executed, if one exists.
- The general syntactical form is as follows:

```
if (condition)
    consequence
[else
    alternative]
```

# if/else Example

```
// if/else Statement Example
string input;

// Prompt the user to select a 1- or 2- player game.
System.Console.Write(
    "1 – Play against the computer\n" +
    "2 – Play against another player.\n" +
    "Choose:"
);

input = System.Console.ReadLine();
if (input == "1")
    // The user selected to play the computer.
    System.Console.WriteLine("Play against computer selected.");
else
    // Default to 2 players (even if user didn't enter 2).
    System.Console.WriteLine("Play against another player.");
```

# Nested if Statement Example

```
int input; // Declare a variable to store the input.
System.Console.WriteLine("What is the maximum number " +
    "of turns in tic-tac-toe?" + "(Enter 0 to exit.): ");

// int.Parse() converts the ReadLine() return to an int data type.
input = int.Parse(System.Console.ReadLine());
if (input <= 0) // Input is less than or equal to 0.
    System.Console.WriteLine("Exiting...");
else
    if (input < 9) // Input is less than 9.
        System.Console.WriteLine(
            "Tic-tac-toe has more than {0}" + " maximum turns.", input);
    else
        if (input > 9) // Input is greater than 9.
            System.Console.WriteLine(
                "Tic-tac-toe has fewer than {0}" + " maximum turns.", input);
        else // Input equals 9.
            System.Console.WriteLine("Correct, " + "tic-tac-toe has a max. of 9 turns.");
```

# if/else Formatted Sequentially

```
// if/else Formatted Sequentially
if (input < 0)
    System.Console.WriteLine("Exiting...");
else if (input < 9)
    System.Console.WriteLine(
        "Tic-tac-toe has more than {0}" + " maximum turns.", input);
else if (input > 9)
    System.Console.WriteLine(
        "Tic-tac-toe has less than {0}" + " maximum turns.", input);
else
    System.Console.WriteLine(
        "Correct, tic-tac-toe has a maximum of 9 turns.");
```

# Code Blocks

- Code blocks can contain one statement or multiple statements.
- If multiple statements appear, braces are needed to identify the code block.
- Indenting the contents of the code block improves readability.

```
if(radius>=0)
{
    area = 3.14*radius*radius; // Calculate the area of the circle.
    System.Console.WriteLine("The area of the circle is: {0}", area);
}
```

# Scope and Declaration Space

- Scope refers to visibility, i.e., it is the area of the program in which a variable is visible.
- The declaration space is the area in which the declared variable is recognized.
- For example, a local variable is not visible outside of its defining method, and it cannot be re-declared within the same method, even in a child code block.

# Scope Example

```
static void Main(string[] args)
{
    int playerCount;
    System.Console.Write("Enter the number of players (1 or 2):");
    playerCount = int.Parse(System.Console.ReadLine());
    if (playerCount != 1 && playerCount != 2)
    {
        string message = "You entered an invalid number of players.";
    }
    else
    {
        // ...
    }
    // Error: message is not in scope.
    System.Console.WriteLine(message);
}
```

# Boolean Expression

- A Boolean expression or conditional statement evaluates to true or false.

```
// Boolean Expression
int input = 0;

if (input < 9) // Evaluates to true or false
{
    // Input is less than 9.
    System.Console.WriteLine("Tic-tac-toe has more than {0}"
        + " maximum turns.", input);
}
```



# Relational and Equality Operators

Operator	Description	Example
<	Less than	input < 9
>	Greater than	input > 9
<=	Less than or equal to	input <= 9
>=	Greater than or equal to	input >= 9
==	Equality operator	input == 9
!=	Inequality operator	input != 9

# Logical Operators

- The logical operators, '||', '&&', and '^', can be used to combine other Boolean operators into conditional expressions.
- The OR operator (||) evaluates to true if either operand is true.
- The AND operator (&&) evaluates to true if both operands are true.
- The XOR operator (exclusive OR: '^') evaluates to true if exactly one operand is true.

# Logical Operator Examples

```
int hourOfTheDay = 0;
```

```
// Using the OR Operator
```

```
if ((hourOfTheDay > 23) || (hourOfTheDay < 0))  
    System.Console.WriteLine("The time you entered is invalid.");
```

```
// Using the AND Operator
```

```
if ((10 < hourOfTheDay) && (hourOfTheDay < 24))  
    System.Console.WriteLine("Hi-Ho, Hi-Ho, it's off to work we go.");
```

# Conditional Values for XOR Operator

Left Operand	Right Operand	Result
True	True	False
True	False	True
False	True	True
False	False	False

# Logical Negation Operator (!)

- The NOT operator, the logical negation operator '!', is a unary operator inverts a bool data type to its opposite.

```
// Using the Logical Negation Operator
bool result;
bool valid = false;
result = !valid;

// Displays "result = True".
System.Console.WriteLine("result = {0}", result);
```

# Conditional Operator (?)

- The conditional operator has the following format:  
    → conditional ? consequence : alternative;
- This operator is a ternary operator, i.e., it takes three operands: conditional, consequence, and alternative.
- If the conditional evaluates to true, the operator returns consequence; if false, the alternative.

# Conditional Operator Example

```
// Conditional Operator
int currentPlayer = 1;
// ...
for (int turn = 1; turn <= 10; turn++)
{
    // ...
    // Switch players
    currentPlayer = (currentPlayer == 2) ? 1 : 2;
}
```

# Null Coalescing Operator (??)

- The Null Coalescing Operator '??', a binary operator, evaluates an expression for null and returns a second expression if the value is null:  
→ expression1 ?? expression2;
- If the first expression (expression1) is not null, then expression1 is returned; if not null, expression2 is returned.

```
// Null Coalescing Operator  
string fileName = null;  
string fullName = fileName ?? "default.txt";
```



# Bitwise Operators

- Bitwise operators manipulate binary data, i.e., the individual bits stored within each byte.
- The Shift Operators (<<, >>, <<=, >>=), shift the binary value of a number to the right or left, bit-by-bit, the number of times specified by the right operand.
- The Bitwise Operators (&, |, ^) perform a bitwise AND, OR, or XOR operation between two binary operands.

# Notes on Bitwise Operators

- All values in a computer are stored in bytes, which are sequences of eight bits having a value of 0 or 1, in base-2 representation.
- Negative numbers are stored in two's complement notation, with a 1 in the most significant (leftmost) bit indicating a negative value.
- Bitwise operators can be combined with assignment operators, e.g., '&=', '|=', and '^='.

# Shift Operators

- In a left shift (<<), the bits in a number are shifted left by the number of positions specified by the right operand; zeroes are used to fill the right-side locations.
- In a right shift (>>), the bits are shifted right, instead; if the number is negative, the values used to fill the left-side locations are ones; if the number is positive, zeroes.

# Bitwise Operators Examples

```
// Using the Right-Shift Operator
```

```
int x;
```

```
x = (-7 >> 2);
```

```
// 11111111111111111111111111111111001 becomes
```

```
// 1111111111111111111111111111111110
```

```
// Write out "x is -2."
```

```
System.Console.WriteLine("x = {0}.", x);
```

```
// Using Bitwise Operators
```

```
byte and, or, xor;
```

```
and = 12 & 7; // and = 4
```

```
or = 12 | 7; // or = 15
```

```
xor = 12 ^ 7; // xor = 11
```

```
System.Console.WriteLine("and = {0} \nor = {1} \nxor = {2}",  
and, or, xor);
```

# Bitwise Operators Examples (cont'd)

```
// Using bitwise operators
byte and, or, xor;
and = 12;
and &= 7; // and = 4
or = 12;
or |= 7; // or = 15
xor = 12;
xor ^= 7; // xor = 11
System.Console.WriteLine("and = {0} \nor = {1} \nxor = {2}",
                          and, or, xor);
```

# Binary Display String Example

```
// Getting a String Representation of a Binary Display
const int size = 64;
ulong value;
char bit;
System.Console.Write("Enter an integer: ");
// Use long.Parse() so as to support negative numbers
// Assumes unchecked assignment to ulong.
value = (ulong)long.Parse(System.Console.ReadLine());
ulong mask = 1ul << size - 1; // Set initial mask to 100....
for (int count = 0; count < size; count++)
{
    bit = ((mask & value) > 0) ? '1' : '0';
    System.Console.Write(bit);
    mask >>= 1; // Shift mask one location over to the right
}
System.Console.WriteLine();
```

## Binary Display String Example (cont'd)

Enter an integer: 42

[illegible]

# Bitwise Complement Operator (~)

- The bitwise complement operator takes the complement of each bit in the operand.
- For example:
  - ~1 returns 1111 1111 1111 1111 1111 1111 1111 1110**
  - ~(1 << 31) returns 0111 1111 1111 1111 1111 1111 1111 1111**



# while Loop

- The while loop will perform a repetitive operation as follows:

```
while (boolean-expression)  
statement
```

- As long as the boolean-expression is true, the statement will execute.
- Execution continues at the instruction following statement.
- The number of iterations is indeterminate.

# while Loop Example

```
// while Loop Example : Fibonacci Calculator
decimal current, previous, temp, input;
System.Console.WriteLine("Enter a positive integer:");
// decimal.Parse convert the ReadLine to a decimal.
input = decimal.Parse(System.Console.ReadLine());

// Initialize current and previous to 1, the first two Fibonacci numbers
current = previous = 1;
// While the current Fibonacci number in the series is
// less than the value input by the user.
while (current <= input)
{
    temp = current;
    current = previous + current;
    previous = temp;
}
System.Console.WriteLine("The Fibonacci number following this is {0}",
                        current);
```

# do/while Loop

- The do/while loop will perform a repetitive operation as follows:  
    **do**  
        **statement**  
    **while (boolean-expression);**
- As long as the boolean-expression is true, the statement will execute (at least one time).
- The number of iterations is indeterminate, but the loop will execute at least once.

# do/while Loop Example

```
// do/while Loop Example
// Repeatedly request player to move until he
// enter a valid position on the board.
bool valid = false;
do
{
    valid = false;

    // Request a move from the current player.
    System.Console.WriteLine("\nPlayer {0}: Enter move:", currentPlayer);
    input = System.Console.ReadLine();

    // Check the current player's input.
    // ...
} while (!valid);
```

# for Loop

- The for loop will perform a repetitive operation as follows:  

```
for (initial; boolean-expression; loop)  
statement
```
- Until the boolean-expression is true, the statement will execute.
- The for loop initializes, increments/decrements, and tests the value of a counter variable.
- The number of iterations is determinate.

# for Loop Example

```
// Using the for Loop: Binary Converter
const int size = 64;
ulong value;
char bit;
System.Console.WriteLine("Enter an integer: ");
// Use long.Parse() so as to support negative numbers
// Assumes unchecked assignment to ulong.
value = (ulong)long.Parse(System.Console.ReadLine());
// Set initial mask to 100....
ulong mask = 1ul << size - 1;

for (int count = 0; count < size; count++)
{
    bit = ((mask & value) > 0) ? '1' : '0';
    System.Console.WriteLine(bit);
    // Shift mask one location over to the right
    mask >>= 1;
}
```

# Choosing Between Loops

- Use the **for** loop whenever there is some type of counter, and the number of iterations is known.
- Use the **while** loop when iterations are not based on a counter, and the number of iterations is unknown.

# The foreach Loop

- The **foreach** loop will perform a repetitive operation on a collection type, setting a variable to represent each item in turn.
- The general format of the **foreach** statement is as follows:  
**foreach (type variable in collection)**  
**statement**
- Operations may be performed on each item returned by the loop.



# foreach Loop Example

```
// Determining Remaining Moves Using the foreach Loop
// Hardcode initial board as follows
// ---+---+---
// 1 | 2 | 3
// ---+---+---
// 4 | 5 | 6
// ---+---+---
// 7 | 8 | 9
// ---+---+---
char[] cells = { '1', '2', '3', '4', '5', '6', '7', '8', '9' };
System.Console.WriteLine("The available moves are as follows: ");
// Write out the initial available moves
foreach (char cell in cells)
{
    if (cell != 'O' && cell != 'X')
    {
        System.Console.WriteLine("{0} ", cell);
    }
}
```

# The switch Statement

- The **switch** statement compares a variable against a list of literal values as follows:

```
switch (test-expression or variable)
{
    [case option-constant:
        statement]
    [default:
        statement]
}
```

- A jump statement like **break** exits a case block.

# switch Statement Example

```
// Replacing the if Statement with a switch Statement
static bool ValidateAndMove(int[] playerPositions, int currentPlayer, string input)
{
    bool valid = false;
    // Check the current player's input.
    switch (input)
    {
        case "1":
        case "2":
        case "3":
        case "4":
        case "5":
        case "6":
        case "7":
        case "8":
        case "9":
            // Save/move as the player directed.
            // ...
            valid = true;
            break;
    }
}
```

# switch Statement Example (cont'd)

```
case "":
case "quit":
    valid = true;
    break;
default:
    // If none of the other case statements is encountered then the text is invalid.
    System.Console.WriteLine("\nERROR: Enter a value from 1-9. " +
        "Push ENTER to quit");
    break;
}
return valid;
}
```

# Jump Statements

- A jump statement makes it possible to escape from a loop or skip the remaining portion of an iteration and start with the next.
- The `break` statement escapes from a loop or switch statement.
- The `continue` statement jumps to the end of the current iteration and begins the next.
- The `goto` statement branches to another line of code but is a poor programming practice.

# break Statement Example

```
// Using break to Escape Once a Winner Is Found
```

```
// Iterate through each winning mask to determine if there is a winner.
```

```
foreach (int mask in winningMasks)
{
    if ((mask & playerPositions[0]) == mask)
    {
        winner = 1;
        break;
    }
    else if ((mask & playerPositions[1]) == mask)
    {
        winner = 2;
        break;
    }
}
```

# continue Statement Example

```
// Determining the Domain of an Email Address
string email;
bool insideDomain = false;
System.Console.WriteLine("Enter an email address: ");
email = System.Console.ReadLine();
System.Console.Write("The email domain is: ");

// Iterate through each letter in the email address.
foreach (char letter in email)
{
    if (!insideDomain)
    {
        if (letter == '@')
        {
            insideDomain = true;
            continue;
        }
        System.Console.Write(letter);
    }
}
```

# goto Statement Example

```
// Demonstrating a switch with goto Statements
switch (option)
{
    case "/out":
        isOutputSet = true;
        isFiltered = false;
        goto default;
    case "/f":
        isFiltered = true;
        isRecursive = false;
        goto default;
    default:
        // ...
        break;
}
```



# C# Preprocessor Directives

- The preprocessor commands are directives to the C# compiler specifying the sections of code to compile or identifying how to handle specific errors and warnings with the code.
- A preprocessor sweeps over the code before the compiler compiles the code looking for special tokens.
- Each preprocessor directive begins with a hash (#) symbol and must appear on one line.

# Types of Preprocessor Directives

- Excluding and including code (`#if`, `#elif`, `#else`, `#endif`)
- Defining preprocessor symbols (`#define`, `#undef`)
- Emitting errors and warnings (`#error`, `#warning`)
- Turning off warning messages (`#pragma`)
- Specifying line numbers (`#line`)
- Hints for visual editors (`#region`, `#endregion`)

# C# Preprocessor Examples

```
// Excluding C# 2.0 Code from a C# 1.x Compiler
```

```
#if CSHARP2
```

```
System.Console.Clear();
```

```
#endif
```

```
// Using #if, #elif, and #endif Directives
```

```
#if LINUX
```

```
// ...
```

```
#elif WINDOWS
```

```
// ...
```

```
#endif
```

# C# Preprocessor Examples (cont'd)

```
// A #define Example
```

```
#define CSHARP2
```

```
// Defining a Warning with #warning
```

```
#warning "Same move allowed multiple times."
```

```
// Using the Preprocessor #pragma Directive to Disable the #warning Directive
```

```
#pragma warning disable 1030
```

```
// Using the Preprocessor #pragma Directive to Restore a Warning
```

```
// ...
```

```
#pragma warning restore 1030
```

```
// The #line Preprocessor Directive
```

```
#line 113 "TicTacToe.cs"
```

```
#warning "Same move allowed multiple times."
```

```
#line default
```

# #region/#endregion Example

```
#region Display Tic-tac-toe Board
// Display the current board
int border = 0; // set the first border (border[0] = " | ")
// Display the top line of dashes.
// ("\n---+---+---\n")
System.Console.Write(borders[2]);
foreach (char cell in cells)
{
    // Write out a cell value and the border that comes after it.
    System.Console.Write(" {0} {1}", cell, borders[border]);
    // Increment to the next border;
    border++;
    // Reset border to 0 if it is 3.
    if (border == 3)
        border = 0;
}
#endregion Display Tic-tac-toe Board
```

# Unit 3 Summary

- This Unit discussed C# arithmetic and assignment operators, along with relational and logical operators.
- Constant expressions were also presented.
- Conditional logic and control flow statements, which make possible decision and looping constructs, were also studied.
- The Unit finished with a discussion of bitwise operators and masks.

# Unit 4

## Methods and Parameters

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 4 Overview

- This Unit covers methods and parameters.
- A **method** is a group of statements treated as a unit.
- A **parameter** is a variable or data that is passed to or returned from a method.
- The Unit also covers more advanced topics including recursion and overloading, along with some new features like optional and named parameters.

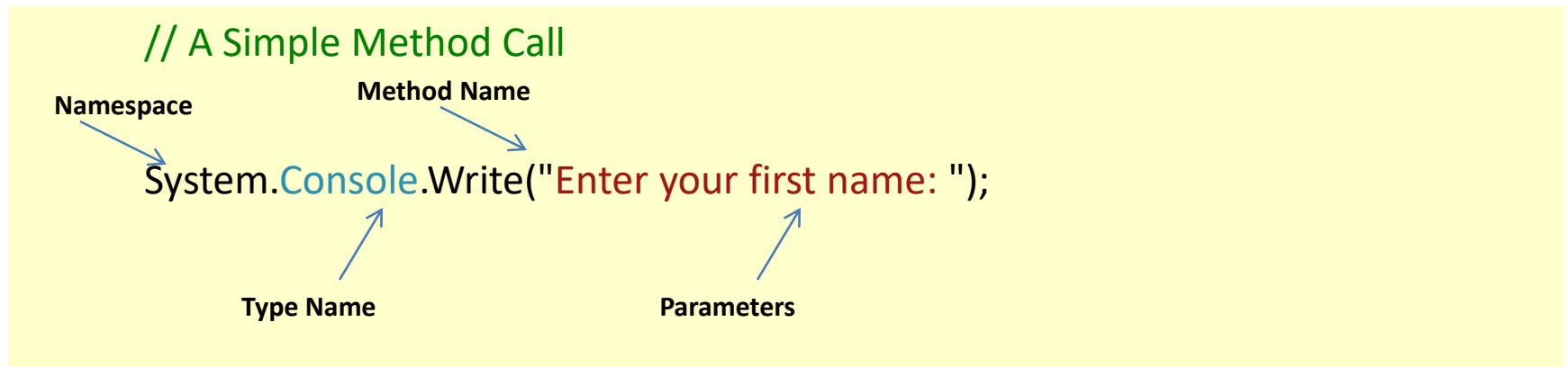


# What is a Method?

- A **method** is a means of grouping together a sequence of statements that perform a particular action or return a result.
- A **method** is always associated with a class or type, which groups related methods together.
- Methods receive data via parameters, variables used for passing data.
- Methods can also return data via a return value, but it is not necessary to do so.

# A Simple Method Call

- The parts of the method call include the namespace, type name, method name, and the return data type.
- A period separates each part of a fully-qualified method name.



# Namespaces

- The **namespace** is a means of categorizing types related to a particular functionality.
- Namespaces are hierarchical and can have numerous levels.
- A common convention is to make the outer namespace the company name, followed by the product name and then the functional area, e.g., Microsoft.Win32.Networking.
- Namespaces help avoid naming collisions.

# More on Namespaces

- `System.Collections` and `System.Collections.Generic` are examples of valid names for namespaces.
- Namespaces are hierarchical, the different levels of the namespace delimited by periods, which improves readability.
- The **using** statement imports a namespace making it unnecessary to use the fully-qualified class name when calling a method.

# Common Namespaces

Namespace	Description
System	Contains the definition of fundamental types, conversion between types, mathematics, program invocation, and environment management.
System.Collections	Includes types for working with collections of objects. Collections can generally follow either list or dictionary type storage mechanisms.
System.Collections.Generic	This C# 2.0 added namespace works with strongly typed collections that depend on generics (type parameters).
System.Data	Contains types used for working with data that is stored within a database.
System.Drawing	Contains types for drawing to the display device and working with images.

# Common Namespaces (cont'd)

Namespace	Description
System.IO	Contains types for working with files and directories and provides capabilities for manipulating, loading, and saving files.
System.Linq	Provides classes and interfaces for querying data in collections using a C# 3.0 added API, Language Integrated Query.
System.Text	Includes types for working with strings and various text encodings, and for converting between those encodings. This namespace includes a subnamespace called System.Text.RegularExpressions, which provides access to regular-expression-related APIs.
System.Threading	Handles thread manipulation and multithreaded programming.

# Common Namespaces (cont'd)

Namespace	Description
System.Threading.Tasks	A family of classes for working with Threads that first appeared in .NET 4.
System.Web	A collection of types that enable browser-to-server communication, generally over HTTP. The functionality within this namespace is used to support a .NET technology called ASP.NET.
System.Web.Services	Contains types that send and retrieve data over HTTP using the Simple Object Access Protocol (SOAP).
System.Windows.Forms	Includes types for creating rich user interfaces and the components within them.
System.Xml	Contains standards-based support for XML processing.

# Type Names

- Types allow the grouping of methods and their associated data.
- For example, Console is the type name containing methods for console I/O: Write(), WriteLine(), and ReadLine().



# Declaring and Calling a Method

```
// Declaring a Method
static void Main(string[] args)
{
    System.Console.WriteLine("Hey you!");
    string firstName = Get userInput("Enter your first name: ");
    string lastName = Get userInput("Enter your last name: ");
    string fullName = GetFullName(firstName, lastName);
    /.../
}
static string Get userInput(string prompt)
{
    System.Console.Write(prompt);
    return System.Console.ReadLine();
}
static string GetFullName(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

# Refactoring

- **Refactoring** means to reduce code duplication by moving a set of statements that occurs in multiple locations into a single method.
- **Refactoring** reduces complexity and improves readability.

# Parameters

- Methods may specify parameters in their declaration.
- Parameters appear within parentheses after the method name in the parameter list.
- Each parameter declaration must include the parameter type and name.
- For example:

```
static string GetFullName(string firstName, string lastName)
```

# Method Return Declaration

- The method return type appears immediately before the method name in the declaration.
  - `static string` GetUserInput(`string` prompt)
- A `return` statement appears with the return keyword followed by the return value.
  - `return` firstName + " " + lastName;
- A method can have multiple return statements.
- A return type of void indicates no return value.

# The using Directive

- The `using` directive is used to import a namespace.
- As a result, it is no longer necessary to fully qualify a type name.
- The `using` directive does not import nested namespaces, however; they must be imported explicitly, too.

# using Directive Example

```
// using Directive Example
// The using directive imports all types from the
// specified namespace into the entire file.
using System;

class HelloWorld
{
    static void Main()
    {
        // No need to qualify Console with System
        // because of the using directive above.
        Console.WriteLine("Hello, my name is Inigo Montoya");
    }
}
```

# Aliasing

- The `using` directive has a provision for aliasing a namespace or type.
- An alias is an alternative name for a namespace.
- The two most common reasons for aliasing are to clarify references to types with the same name and to abbreviate a long name.

# Aliasing Example

```
// Declaring a Type Alias
using CountdownTimer = System.Timers.Timer;
namespace ConsoleApplication13
{
    class Program
    {
        static void Main(string[] args)
        {
            CountdownTimer timer;
            // ... //
        }
    }
}
```



# Main() Command-Line Arguments and Return Values

- The runtime (CLR) passes command-line arguments to the Main() method using a single string-array parameter.
- The GetCommandLineArgs() method (System.Environment namespace) also returns the command-line arguments array.
- The Main() method can also return a value, which can be used as a status code in a script or batch file.

# Main() Function Example

```
static int Main(string[] args)
{
    int result = 0;

    //...//

    return result;
}
```

# The Call Stack

- The **call stack** is the name for all of the currently active (not finished) method calls.
- As more method calls are made, the call stack generally grows larger and larger.
- As calls complete, the call stack shrinks down until more methods are called.
- The term describing the process of removing calls from the call stack is called **stack unwinding**.

# More on Parameters

- By default, parameters are **passed by value**, meaning that each variable passed is copied into the parameters of the target method (on the stack, of course).
- Note that the names of the variables being passed to the target method do not need to match its parameter names.

# Passing Variables by Value

```
static void Main()
{
    string fullName;
    string driveLetter = "C:";
    string folderPath = "Data";
    string fileName = "index.html";
    fullName = Combine(driveLetter, folderPath, fileName);
    Console.WriteLine(fullName);
}

static string Combine(string driveLetter, string folderPath, string fileName)
{
    string path;
    path = string.Format("{1}{0}{2}{0}{3}",
        System.IO.Path.DirectorySeparatorChar,
        driveLetter, folderPath, fileName);
    return path;
}
```

# Reference Type Variables

- A reference type variable contains an address of the memory location where the data is stored.
- If the reference type variable is passed by value, the address is copied from the caller to the method parameter, not the data, itself.

# Reference Parameters (ref)

- Reference parameters are indicated by the keyword `ref`.
- With reference parameters, the called method can update the caller's variable.
- Variables passed as `ref` must be initialized because target methods may read data from the parameter.
- A `ref` parameter is an alias for the variable passed.

# Reference Parameters (ref) Example

// Listing 4.12: Passing Variables by Reference

```
static void Main()
{
    string first = "first";
    string second = "second";
    Swap(ref first, ref second);

    System.Console.WriteLine(@"first = ""{0}"" , second = ""{1}""",
                             first, second);
}

static void Swap(ref string first, ref string second)
{
    string temp = first;
    first = second;
    second = temp;
}
```



# Output Parameters (out)

- Output parameters, marked with the keyword **out**, must pass data out of a target method; otherwise, the compiler will issue an error.

# Output Parameters (out) Example

```
// Passing Variables Out Only
static bool GetPhoneButton(char character, out char button)
{
    bool success = true;
    switch (char.ToLower(character))
    {
        case '1':
            button = '1';
            break;
        / ... /
        default:
            // Set the button to indicate an invalid value
            button = '_';
            success = false;
            break;
    }
    return success;
}
```

# Parameter Arrays (params)

- The `params` keyword allows the number of parameters to vary in the calling code.
- The parameter array declaration must be the last parameter in the parameter declaration list, and a method can have only one.
- Parameter arrays are strongly typed and can receive an explicit array instead of a list of values.

# Parameter Arrays (params) Example

```
// Passing a Variable Parameter List
static void Main()
{
    // Call Combine() with four parameters
    string fullName = Combine(Directory.GetCurrentDirectory(),
                              "bin", "config", "index.html");
    Console.WriteLine(fullName);
}
static string Combine(params string[] paths)
{
    string result = string.Empty;
    foreach (string path in paths)
    {
        result = System.IO.Path.Combine(result, path);
    }
    return result;
}
```

# Recursion

- **Recursion** means that a method calls itself.
- This technique is often used with hierarchical structures such as the file system in Windows.
- A common programming error in recursive method implementations is **stack overflow**, which happens because of **infinite recursion**, i.e., because the recursion never terminates.

# Example of Recursion

```
// Returning All the Filenames, Given a Directory
static int DirectoryCountLines(string directory)
{
    int lineCount = 0;
    foreach (string file in Directory.GetFiles(directory, "*.cs"))
    {
        lineCount += CountLines(file);
    }

    foreach (string subdirectory in Directory.GetDirectories(directory))
    {
        lineCount += DirectoryCountLines(subdirectory);
    }
    return lineCount;
}
```

# Method Overloading

- **Method overloading** refers to the situation where a class has multiple methods having the same name but which differ in the number and/or types of their parameters.
- The effect of **method overloading** is to provide optional ways to call a method.
- A common pattern is that core logic is implemented in one method, and the other overloaded methods call that single method.

# Method Overloading Example

```
// Returning All the Filenames, Given a Directory
static int DirectoryCountLines(string directory)
{ return DirectoryCountLines(directory, "*.cs"); }

static int DirectoryCountLines()
{ return DirectoryCountLines(Directory.GetCurrentDirectory()); }

static int DirectoryCountLines(string directory, string extension)
{
    int lineCount = 0;
    foreach (string file in Directory.GetFiles(directory, extension))
    { lineCount += CountLines(file); }

    foreach (string subdirectory in Directory.GetDirectories(directory))
    { lineCount += DirectoryCountLines(subdirectory); }

    return lineCount;
}
```



# Optional Parameters

- Optional parameters, new to C# 4.0, allow the assignment of a parameter to a constant value as part of the method declaration.
- It is now possible to call a method without passing every parameter for the method.
- Optional parameters must appear after all required parameters (non-optional), and the default value must be a literal value.

# Optional Parameters Example

```
// Methods with Optional Parameters
public static void Main(string[] args)
{
    if (args.Length > 0)
    {
        totalLineCount = DirectoryCountLines(args[0]);
    }
    else
    {
        // ... //
    }
    System.Console.WriteLine(totalLineCount);
}

static int DirectoryCountLines(string directory, string extension = "*.cs")
{ // ... // }
```

# Named Parameters

- Named parameters, also new to C# 4.0, make it possible for the caller to explicitly identify the name of the parameter to be assigned a value, rather than relying on the parameter order.

# Named Parameters Example

```
static void Main()
{
    DisplayGreeting(firstName: "Inigo", lastName: "Montoya");
}

// Specifying Parameters by Name
static void DisplayGreeting(string firstName,
                           string middleName = default(string),
                           string lastName = default(string))
{
    // ...
}
```

# Basic Error Handling with Exceptions

- An **exception** (error) is thrown when an error is not handled.
- The process of handling an exception is referred to as **catching an exception**.
- A **try...catch...finally** block is used to handle and exception.
- The **finally** block of code always executes.
- All exceptions inherit from `System.Exception`.

# Exception Handling Recommendations

- Catch blocks must be ordered from the most specific exception to the least specific; the `System.Exception` type is least specific and should appear last.
- It is preferable to include a catch block that is specific to the most specific exception type in order to handle the exception specifically.
- Finally blocks are used to clean up resources regardless of whether an exception is thrown.

# Catching an Exception Example

```
try
{
    age = int.Parse(ageText);
    Console.WriteLine("Hi {0}! You are {1} months old.",
                      firstName, age * 12);
}
catch (FormatException)
{
    Console.WriteLine("The age entered, {0}, is not valid.", ageText);
}
catch (Exception exception)
{
    Console.WriteLine("Unexpected error: {0}", exception.Message);
}
finally
{
    Console.WriteLine("Goodbye {0}", firstName); }
```

# Common Exception Types

Exception Type	Description
System.Exception	A generic exception from which other exceptions derive.
System.ArgumentException	A means of indicating that one of the parameters passed into the method is invalid.
System.ArgumentNullException	Indicates that a particular parameter is null and that this is not valid for that parameter.
System.ApplicationException	To be avoided.
System.FormatException	Indicates that the string format is not valid for conversion.
System.IndexOutOfRangeException	Indicates that an attempt was made to access an array element that does not exist.
System.InvalidCastException	Indicates that an attempt to convert from one data type to another was not a valid conversion.



# Common Exception Types (cont'd)

Exception Type	Description
<code>System.NotImplementedException</code>	Indicates that although the method signature exists, it has not been fully implemented.
<code>System.NullReferenceException</code>	Throws when code tries to find the object referred to by a reference (such as a variable) which is null.
<code>System.ArithmeticException</code>	Indicates an invalid math operation, not including divide by zero.
<code>System.ArrayTypeMismatchException</code>	Occurs when attempting to store an element of the wrong type into an array.
<code>System.StackOverflowException</code>	Generally indicates that there is an infinite loop in which a method is calling back into itself (known as recursion).

# Generic catch

- A catch block that takes no parameters is referred to as a generic catch block.
- Generic catch blocks provide no way to capture exception information.
- These catch blocks are equivalent to specifying a catch block for the `System.Exception` type, i.e., one that will catch all exceptions not caught by earlier blocks.

# Generic catch Block Example

```
try
{
    age = int.Parse(ageText);
    System.Console.WriteLine("Hi {0}! You are {1} months old.",
                             firstName, age * 12);
}
catch (System.FormatException exception)
{
    System.Console.WriteLine("The age entered ,{0}, is not valid.", ageText);
}
catch
{
    System.Console.WriteLine("Unexpected error!");
}
finally
{
    System.Console.WriteLine("Goodbye {0}", firstName);
}
```

# Reporting Errors Using a Throw Statement

- Exceptions can be thrown from within the application code using the **throw** keyword.
- To throw an exception, an instance of an exception must first be created.
- An exception can also be re-thrown inside of a catch block.
- Most exception types allow a message to be passed when the exception is thrown.

# Example of Throwing an Exception

```
// Throwing an Exception
try
{
    Console.WriteLine("Begin executing");
    Console.WriteLine("Throw exception...");
    throw new Exception("Arbitrary exception");
}
catch (FormatException exception)
{
    Console.WriteLine("A FormatException was thrown");
}
catch (Exception exception)
{
    Console.WriteLine("Unexpected error: {0}", exception.Message);
}
```

# Exception Best Practices

- The use of exceptions should be reserved for exceptional, unexpected, or fatal situations.
- Exceptions should not be thrown or allowed to occur for predictable or expected situations because they interrupt program flow and impact performance negatively.

# Numeric Conversion with TryParse()

- The Parse() method will throw an exception if the requested casting operation is unsuccessful.
- TryParse() will attempt the cast operation but will return a false if unsuccessful instead of throwing an exception.
- TryParse() is available with all numeric types in the .NET Framework 4.0.

# TryParse() Example

```
// TryParse() Example
if (int.TryParse(ageText, out age))
{
    System.Console.WriteLine("Hi {0}! You are {1} months old.",
                             firstName, age * 12);
}
else
{
    System.Console.WriteLine("The age entered ,{0}, is not valid.",
                             ageText);
}
```



# Unit 4 Summary

- This Unit discussed the details of declaring and calling methods.
- Methods are a fundamental construct that is a key to writing readable code.
- The next Unit considers the class type and how it encapsulates methods and fields.

# Unit 5

# Classes

C# Data Structures and Design Patterns

Spring Quarter, 2017

# Unit 5 Overview

- This Unit presents the basics of object-oriented programming using C#.
- The key focus is on how to define classes, which are templates for objects.
- This Unit also delves into how C# supports encapsulation through its support of classes, properties, and access modifiers.
- The next Unit discusses inheritance and polymorphism, which are related topics.

# Object-Oriented Programming

- Object-oriented programming (OOP) is a programming methodology for organizing and structuring applications.
- The class type is fundamental to the OOP framework and provides a programming abstraction or template for a real-world entity or a concept.
- The three pillars of OOP are encapsulation, inheritance, and polymorphism.

# Encapsulation

- **Encapsulation** refers to the hiding of data and implementation details.
- By encapsulating data and details into classes and objects, large programs become easier to manage and understand.
- For example, methods are encapsulated within classes.

# Inheritance

- **Inheritance** describes an “is-a” relationship between two different types (or entities); e.g., a triangle is a shape.
- The more specialized type is the derived type or subtype; the more generalized type is the base type or the super type.
- Derived types inherit the members of the base type.

# Polymorphism

- **Polymorphism** translates into “many forms.”
- In the context of objects and methods, **polymorphism** means that a single type can have many forms of implementation.
- For example, the base type `OpticalStorageMedia` may define a method `Play()` that is implemented by the derived type `CD`, i.e., the derived type takes care of all of the implementation details.

# Class Definition and Declaration

- A class definition is a new type identified by the **class** keyword.
- A variable or parameter can then be declared of this new type.
- A class is a template or blueprint for an object, whereas, an object is an instance of a class.
- The process of creating an object is called instantiation.



# Class Definition and Declaration Example

```
// Defining, declaring, and instantiating a class
class Employee
{
    // ... //
}
static void Main(string[] args)
{
    Employee employee1 = new Employee();
    Employee employee2;
    employee2 = new Employee();

    IncreaseSalary(employee1);
}
static void IncreaseSalary(Employee employee)
{
    // ... //
}
```

# Garbage Collection

- The **garbage collector** is responsible for reclaiming memory after an object is released.
- It determines which objects are no longer referenced by active objects and then reclaims the memory used by those objects.
- Object destruction is **non-deterministic** meaning that it is handled by the garbage collector, not the application itself, at a time of its choosing.

# Instance Fields

- A **field** is a variable within a class that is declared outside of any method declarations.
- Fields allow data to be stored with an object or instance of the class.
- Fields are initialized to their default values when an object is instantiated, but they can also be explicitly assigned an initial value.
- An access modifier of public indicates that a field is accessible to other objects.

# Declaring and Accessing Fields

```
class Employee
{
    public string FirstName;
    public string LastName;
    public string Salary = "Not enough";
}
static void Main(string[] args)
{
    Employee employee1 = new Employee();
    employee1.FirstName = "Inigo";
    employee1.LastName = "Montoya";
    employee1.Salary = "Too Little";

    Console.WriteLine("{0} {1}: {2}",employee1.FirstName,
                      employee1.LastName,employee1.Salary);
}
```

# Instance Methods

- Methods can be encapsulated within the class and then called from an instance of the class.

```
class Employee
{
    public string FirstName;
    public string LastName;
    public string Salary = "Not enough";

    public string GetName()
    {
        return FirstName + " " + LastName;
    }
}
```

# The **this** Keyword

- The keyword **this** is a reference to the current instance of a class that is accessible from within the class definition.
- The **this** keyword can serve to resolve ambiguities in variable naming practices, e.g., when parameters have the same name and case as the corresponding field names.

```
public void SetName(string FirstName, string LastName)
{
    this.FirstName = FirstName;
    this.LastName = LastName;
}
```

# Example of **this** Keyword

// Using this to Identify the Field's Owner Explicitly

```
class Employee
{
    public string FirstName;
    public string LastName;
    public string Salary = "Not enough";

    public string GetName()
    {
        return FirstName + " " + LastName;
    }

    public void SetName(string newFirstName, string newLastName)
    {
        this.FirstName = newFirstName;
        this.LastName = newLastName;
    }
}
```

# Pass **this** in a Method Call

// Passing this in a method call

```
class DataStorage
```

```
{
```

```
    // Save an employee object to a file named with the Employee name.
```

```
    public static void Store(Employee employee)
```

```
    { // ... // }
```

```
}
```

// Using this to Identify the Field's Owner Explicitly

```
class Employee
```

```
{
```

```
    public string FirstName;
```

```
    public string LastName;
```

```
    public string Salary = "Not enough";
```

```
    public void Save()
```

```
    { DataStorage.Store(this); }
```

```
    // ... //
```

```
}
```



# Access Modifiers

- Five access modifiers are available: **public**, **private**, **protected**, **internal**, and **protected internal**.
- Controlling access to data and methods through access modifiers is an important part of encapsulation.
- **Public** data is accessible from outside the class; **private** data from within the class.

# Access Modifier Example

// Using the private and public access modifiers

```
public class Employee
{
    public string FirstName;
    public string LastName;
    public string Salary;
    private string Password;
    private bool IsAuthenticated;

    public bool Logon(string password)
    {
        if (Password == password)
        {
            IsAuthenticated = true;
        }
        return IsAuthenticated;
    }

    public bool GetIsAuthenticated()
    {
        return IsAuthenticated;
    }
}
```

# Properties

- A **property** enables the getting and setting of the underlying field data just like a field does.
- Property definitions use three keywords: `get` and `set` to identify the retrieval or the assignment portion of the property and `value` to refer to the value being assigned.
- Note that auto-implemented properties are simpler because they do not require any explicit implementation or backing store.

# Properties Example

```
// Defining Properties
class Employee
{
    // FirstName property
    private string _FirstName;           // backing store
    public string FirstName
    {
        get
        { return _FirstName; }
        set
        { _FirstName = value; }
    }
    // ... //
    // Automatically implemented properties (no backing store)
    public string Title { get; set; }
    public Employee Manager { get; set; }
}
```

# Properties Example (cont'd)

```
static void Main()
{
    Employee employee1 = new Employee();
    Employee employee2 = new Employee();

    // Call the FirstName property's setter
    employee1.FirstName = "Inigo";

    // Call the FirstName property's getter.
    System.Console.WriteLine(employee1.FirstName);

    // Assign an auto-implemented property
    employee2.Title = "Computer Nerd";
    employee1.Manager = employee2;

    // Print employee1's manager's title.
    System.Console.WriteLine(employee1.Manager.Title);
}
```

# Properties and Validation

- Unlike a public field, properties can intercept an assignment and validate the parameters, cancelling the assignment if necessary.
- Inside the class itself, always access a property value through its property implementation rather than through its backing field so any validation code can execute.

# Property Validation Example

```
// Providing Property Validation
public string LastName
{
    get { return _LastName; }
    set
    {
        if (value == null) // Validate LastName assignment
            throw new ArgumentNullException(); // Report error
        else
        {
            value = value.Trim(); // Remove whitespace
            if (value == "")
                throw new ArgumentException("LastName cannot be blank.");
            else
                _LastName = value;
        }
    }
}
```

# Property Access Modifiers

- An access modifier can be placed on either the set or get portion of the property implementation overriding the access modifier on the property declaration.

```
// Placing Access Modifiers on the Setter
```

```
private string _Id;  
public string Id  
{  
    get { return _Id; }
```

```
// Property can be set only from inside class--not outside
```

```
private set { _Id = value; }  
}
```



# Calculated Properties

- Some properties return a calculated value from the property getter and additional parsing takes place in the property setter.

# Example of Calculated Properties

```
// Calculated Name property
public string Name
{
    get { return FirstName + " " + LastName; }
    set
    {
        string[] names;
        names = value.Split(new char[] { ' ' });
        if (names.Length == 2)
        {
            FirstName = names[0];
            LastName = names[1];
        }
        else // Throw an exception if the full name was not assigned.
            throw new System.ArgumentException(string.Format(
                "Assigned value '{0}' is invalid", value));
    }
}
```

# Read-Only and Write-Only Properties

- Properties with only a property get are **read-only** properties; properties with only a property set are **write-only** properties.

```
// Read-only Id property declaration
private string _Id;
public string Id
{
    get
    {
        return _Id;
    }
    // No setter provided.
}
```

# Constructors

- A **constructor** is a method with no return type; its name is identical to the class name.
- The constructor is called to create an instance of the object defined by the class type definition using the **new** operator.
- Constructors are often used to initialize fields.
- The **default** constructor takes no parameters and is added automatically unless a constructor is added explicitly to the class.

# Constructor Example

```
// Defining a Constructor
class Employee
{
    // Employee default constructor
    public Employee()
    {}
    // Employee overloaded constructor (parameterized)
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
    // Properties
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Salary { get; set; }
}
```

# Object Initializers

- Object initializers, new to C# 3.0, allow the initialization of an object's fields and properties inside an object initializer, which appears after the constructor's parameter list (inside the curly braces).

```
static void Main()
{
    // Calling an Object Initializer
    Employee employee1 = new Employee("Inigo", "Montoya") {
        Title = "Computer Nerd", Salary = "Not enough" };
    // ... //
}
```

# Collection Initializers

- Collection, also new to C# 3.0, initializers allow the assignment of items within the collection at the time of the collection's instantiation.

```
// Calling an Object Initializer
static void Main()
{
    List<Employee> employees = new List<Employee>()
    {
        new Employee("Inigo", "Montoya"),
        new Employee("Chuck", "McAtee")
    };
    // ... //
}
```

# Chaining Constructors Using **this**

- To avoid duplicating code, i.e., to reuse code occurring in multiple places, one constructor can chain to another constructor using **this**.

```
//Calling One Constructor from Another
public Employee(string firstName, string lastName)
{
    FirstName = firstName;
    LastName = lastName;
}

public Employee(int id, string firstName, string lastName)
    : this(firstName, lastName)
{ Id = id; }
```



# Anonymous Types

- Anonymous types are generated by the compiler (on the fly) rather than through explicit class definitions.
- Anonymous types are primarily used with lambda and LINQ expressions.

```
// Anonymous Types
var patent =
    new
    {
        Title = "Bifocals",
        YearOfPublication = "1784"
    };
```

# Static Members

- Static members are associated with the class itself rather than the object created from the class; thus a static member is accessed through the class name not the object name.
- A static field, method, or property takes a **static** modifier.
- Static members cannot access instance members or the **this** keyword.
- Static members can be globally accessed.

# Example of Static Field

// Declaring a Static Field

```
public static int NextId = 42;  
public Employee(string firstName, string lastName)  
{  
    FirstName = firstName;  
    LastName = lastName;  
    Id = NextId;  
    NextId++;  
}
```

// Accessing a Static Field

```
static void Main()  
{  
    Employee.NextId = 1000000;  
}
```

# Example of Static Method

```
// Defining a Static Method on DirectoryInfo
public static class DirectoryInfoExtension
{
    public static void CopyTo( DirectoryInfo sourceDirectory, string target,
                             SearchOption option, string searchPattern)
    {
        // ... //
    }
}
```

# Static Constructors

- Static constructors initialize a class not the object instance and are used to initialize only static data.
- The CLR calls the static constructor only once when the class is first accessed, either through a call to the instance constructor or when accessing a static member.
- Static constructors cannot be called by any application code.

# Static Constructor Example

```
// Declaring a Static Constructor  
public static int NextId = 42;  
static Employee()  
{  
    Random randomGenerator = new Random();  
    NextId = randomGenerator.Next(101, 999);  
}
```

# Static Classes

- Static classes contain only static members and cannot be instantiated.
- Static classes cannot be used as base classes.
- The static keyword identifies a class as static.

```
public static class SimpleMath
{
    static int Max(params int[] numbers)
    {
        // ... //
    }
    // ... //
}
```

# Extension Methods

- C# simulates the creation of a new instance method on a class via extension methods even if the class is defined in another assembly.



# Example of Extension Method

```
// Static Copy Method for DirectoryInfo
public static class DirectoryInfoExtension
{
    public static void CopyTo(
        this DirectoryInfo sourceDirectory, string target,
        SearchOption option, string searchPattern)
    {
        // ... //
    }
}

// Calling an Extension Method
// ... //
DirectoryInfo directory = new DirectoryInfo(".\\Source");
directory.CopyTo(".\\Target", SearchOption.AllDirectories, "*");
// ... //
```

# Rules for Extension Methods

- The first parameter corresponds to the type the method extends.
- To define the extension method, prefix the extended type with the `this` operator.
- To access the method as an extension method, import the extending type's namespace with a `using` directive (or place the extending class in the same namespace as the calling code).

# Constant Data

- Constant field data is indicated by the keyword `const` and contains literal values that must be present at compile time.
- Constant fields are by definition static and cannot be changed at runtime.

```
// Declaring a Constant Field
class ConvertUnits
{
    public const float CentimetersPerInch = 2.54F;
    public const int CupsPerGallon = 16;
    // ... //
}
```

# Read-Only Data

- The **readonly** modifier declares that a field value is modifiable only from inside the constructor or in the declaration statement.
- Readonly fields can be either static or instance fields.

```
class Employee
{
    public readonly int Id; // Declaring a Field As readonly
    public Employee(int id)
    {
        Id = id;
    }
    // ...
}
```

# Nested Classes

- Nested classes declared within another class.
- Nested classes can be specified as **private**.
- The **this** member, if used within a nested class, refers to the instance of the nested class not the containing class.
- Nested classes can access any member of the containing class including private members.
- Nested classes are rarely used.

# Example of Nested Class

```
class Program
{
    // Define a nested class for processing the command line.
    private class CommandLine
    {
        // ... //
    }

    // ... //
    static void Main(string[] args)
    {
        CommandLine commandLine = new CommandLine(args);
    }
}
```

# Partial Classes

- Partial classes are portions of classes that can be combined by the compiler at compile time.
- Partial classes can be split across multiple files.
- The keyword `partial` is used to declare a partial class:
  - `partial class Program { // ... // }`
- Note that partial classes cannot be merged with partial classes in other assemblies.

# Partial Methods

- Partial methods allow the declaration of a method without an implementation inside a partial class.
- Partial methods allow generated code to call methods that have not been implemented yet.
- The implementation of the partial method can be located in an associated partial class.
- All partial method invocations are removed if the method is not implemented.



# Partial Method Example

```
// File: Person.Designer.cs
public partial class Person
{
    partial void OnLastNameChanging(string value);
    private string _LastName;
    public string LastName
    {
        get
        { return _LastName; }
        set
        {
            if ((_LastName != value))
            {
                OnLastNameChanging(value);
                _LastName = value;
            }
        }
    }
}
```

# Partial Method Example (cont'd)

```
// File: Person.cs
partial class Person
{
    partial void OnLastNameChanging(string value)
    {
        if (value == null)
        {
            throw new ArgumentNullException("LastName");
        }
        if (value.Trim().Length == 0)
        {
            throw new ArgumentException("LastName cannot be empty.");
        }
    }
}
```

# Unit 5 Summary

- This Unit explained C# classes and objects.
- Static and instance data stored in fields and accessible through properties were discussed.
- Encapsulation of private fields and access modifiers for methods, properties, and data were explored.
- The next Unit discusses inheritance and polymorphism.

# Unit 6

## Inheritance

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 6 Overview

- The previous Unit discussed how one class references another class via properties, fields, and methods.
- This Unit discusses the inheritance relationship between classes, how it is implemented, and how it is used to build class hierarchies.





# Inheritance

- **Inheritance** describes an “is-a” relationship between two entities.
- For example, a hard drive “is a” storage device.
- The parent entity is called the base type or supertype.
- The child entity is called the derived type or subtype.
- Inheritance is about specializing the base type.



# Inheritance Definitions

- **Derive/Inherit:** To specialize a base class to include additional members or customize base-class members
- **Derived Type/Subtype/Child Type:** The specialized type that inherits members from the general type.
- **Base Type/Supertype/Parent Type:** The general type from whose members a derived type inherits.

# Derivation

- Inheritance allows for the extension of a base class, i.e., the addition of data and behaviors inside of the inheriting derived class.
- Classes deriving from each other form an inheritance chain.
- When specifying inheritance between classes, insert a colon after the derived-class name followed by the base-class name.

**→public class Contact : PdaItem**

# Example of Derivation

```
// Deriving One Class from Another
public class Pdaltem
{
    public string Name { get; set; }
    public DateTime LastUpdated { get; set; }
}
// Define the Contact class as inheriting the Pdaltem class
public class Contact : Pdaltem
{
    public string Address { get; set; }
    public string Phone { get; set; }
}
static void Main(string[] args)
{
    Contact contact = new Contact();
    contact.Name = "Inigo Montoya";
}
```

# Casting Between Related Types

- A derived type can always be assigned to a base type because the derivation forms an “is-a” relationship.
- This assignment is sometimes referred to as an implicit conversion because no specific casting operator is required.

```
// Derived types can be implicitly converted to base types
Contact contact = new Contact();
Pdaltem item = contact;
// Base types must be cast explicitly to derived types
contact = (Contact)item;
```

# Implicit and Explicit Conversions

- Explicit conversion between types requires the use of a conversion operator; implicit conversion does not.
- A conversion from a derived class to its base class is implicit, but to go from a base class to a derived class, an explicit operator is used.
- Type conversion is not always limited to types in a single inheritance chain, e.g., some numeric types are convertible to other numeric types.

# Access Modifiers

- Derived classes cannot access base-class members declared as **private**.
- The **protected** access modifier allows only derived classes to access a base-class member, i.e., public access is not granted.
- The **public** access modifier, of course, allows public access and derived-class access.

# Single Inheritance

- A child class cannot derive from more than one parent class directly in C#; thus multiple inheritance is not allowed, unlike C++.
- Aggregation is an alternative to multiple inheritance.

# Aggregation as an Alternative

- **Aggregation** is an alternative to multiple inheritance in which instead of inheriting from a second class, a class contains an instance of the second class as a field.
- Non-private members of the second class are redefined as members of the original class, which delegate to corresponding members of the second class.



# Example of Aggregation

```
// Working around Single Inheritance Using Aggregation
public class Pdaltem
{    // ... //
public class Person
{    // ... //
public class Contact : Pdaltem
{
    private Person InternalPerson { get; set; } // Aggregation here
    public string FirstName
    {
        get { return InternalPerson.FirstName; }
        set { InternalPerson.FirstName = value; }
    }
    // ... //
}
```

# Sealed Classes

- Inheritance can be prevented by marking a class as **sealed**.
- The string type is an example of a sealed class.

```
// Preventing Derivation with Sealed Classes
```

```
public sealed class CommandLineParser
```

```
{    // ...    }
```

```
// ERROR: Sealed classes cannot be derived from
```

```
public sealed class DerivedCommandLineParser : CommandLineParser
```

```
{    // ...    }
```

# The virtual Modifier

- A base class marks each **public** or **protected** member for which overriding is allowed as **virtual**.
- If the member is not marked **virtual**, it cannot be overridden.
- In the derived class, the overridden member is identified with the keyword **override**.
- Note that the runtime calls the most derived implementation of a virtual method.

# Example of Overriding a Method

```
// Overriding a Property
public class Pdaltem
{
    public virtual string Name { get; set; }
}
public class Contact : Pdaltem
{
    public override string Name
    {
        get
        { return FirstName + " " + LastName; }
        set
        {
            string[] names = value.Split(' ');
            FirstName = names[0];
            LastName = names[1];
        }
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

# Facts About Virtual Methods

- Virtual methods should not contain critical code because if they are overridden, the code may never be called.
- Converting a method from a **virtual** method to a non-virtual method may break derived classes that already override the method.
- Virtual methods provide default implementations but can be overridden.
- Only instance members can be **virtual**.

# The new Modifier

- The **new** modifier is used to hide an inherited member rather than override it.
- If neither **override** nor **new** is specified when a derived class declares a method already defined in the base class, the **new** modifier will be assumed.
- A base class reference to a derived class will call the most derived member prior to any corresponding member with the **new** modifier.

# The sealed Modifier

- Virtual members, like classes, may be sealed to prevent them from being overridden again in a derived class.
  - `public override sealed void Method() { ... }`

# Example of the sealed Modifier

```
// Sealing Members
class A
{
    public virtual void Method()
    { // ... // }
}
class B : A
{
    public override sealed void Method()
    { // ... // }
}
class C : B
{
    // ERROR: Cannot override sealed members
    public override void Method()
    { // ... // }
}
```



# Accessing a Base Member

- The **base** keyword allows an overridden method in a derived class to call the base-class implementation of the method.
- The syntax used is similar to that of the **this** keyword.

```
//Accessing a Base Member
public class InternationalAddress : Address
{
    public string Country;
    public override string ToString()
    { return base.ToString() + Environment.NewLine + Country; }
}
```

# Chaining Constructors

- Constructors in a derived class will sometimes explicitly chain back to the appropriate base-class constructor when there is no accessible default base-class constructor.

```
// Specifying Which Base Constructor to Invoke
```

```
public class Contact : Pdaltem
{
    public Contact(string name)
        : base(name)
    { Name = name; }
    public string Name { get; set; }
}
```

# Abstract Classes

- Abstract classes are designed for derivation only, i.e., they cannot be instantiated .
- Classes that are not abstract and which can be instantiated are called **concrete** classes.
- Abstract members define the functionality that a derived class must support but do not provide the implementation, just the format.
- Derived classes must implement all abstract members to be considered concrete classes.

# Abstract Class Example

```
// Define an abstract class
public abstract class Pdaltem
{
    public Pdaltem(string name)
    { Name = name; }
    public virtual string Name { get; set; }
    public abstract string GetSummary();
}
```

# Abstract Class Example (cont'd)

```
// Implement an abstract class
public class Contact : Pdaltem
{
    public override string GetSummary()
    {
        return string.Format("FirstName: {0} LastName: {1} Address: {2}",
                               FirstName, LastName, Address);
    }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Address { get; set; }

    public override string Name
    {
        // ... //
    }
}
```

# Polymorphism

- Polymorphism encourages code reuse by factoring out commonalities in the base class.
- Polymorphism means “many forms,” which means that different implementations of an abstract class or member may exist in different derived class(es).
- Thus, the base class specifies the signature of the method, and the derived class provides the implementation.

# Polymorphism Example

```
// Using Polymorphism to list PdaItems
public static void Main()
{
    PdaItem[] pda = new PdaItem[3];
    Contact contact = new Contact("Sherlock Holmes");
    contact.Address = "221B Baker Street, London, England";
    pda[0] = contact;
    Appointment appointment = new Appointment("Soccer tournament");
    appointment.StartDateTime = new DateTime(2008, 7, 18);
    appointment.EndDateTime = new DateTime(2008, 7, 19);
    appointment.Location = "Est-dio da Machava";
    pda[1] = appointment;
    contact = new Contact("Anne Frank");
    contact.Address = "263 Prinsengracht, Amsterdam, Netherlands";
    pda[2] = contact;
    List(pda);
}
```

# Polymorphism Example (cont'd)

```
public static void List(PdalItem[] items)
{
    // Implemented using polymorphism. The derived
    // type knows the specifics of implementing GetSummary().
    foreach (PdalItem item in items)
    {
        Console.WriteLine("_____");
        Console.WriteLine(item.GetSummary());
    }
}
```



# System.Object

- All classes derive from System.Object, whether or not the inheritance is indicated explicitly.

# Members of System.Object

Method Name	Description
<b>public virtual bool Equals(object o)</b>	Returns true if the object supplied as a parameter is equal in <i>value</i> , not necessarily in reference, to the instance.
<b>public virtual int GetHashCode()</b>	Returns an integer corresponding to an evenly spread hash code. This is useful for collections such as HashTable collections.
<b>public Type GetType()</b>	Returns an object of type System.Type corresponding to the type of the object instance.
<b>public static bool ReferenceEquals(object a, object b)</b>	Returns true if the two supplied parameters refer to the same object.

# Members of System.Object (cont'd)

Method Name	Description
<b>public virtual string ToString()</b>	Returns a string representation of the object instance.
<b>public virtual void Finalize()</b>	An alias for the destructor; informs the object to prepare for termination. C# prevents calling this method directly.
<b>protected object MemberwiseClone()</b>	Clones the object in question by performing a shallow copy; references are copied, but not the data within a referenced type.

# The is Operator

- The **is** operator is used to determine the underlying type.

```
// is Operator Determining the Underlying Type
public static void Save(object data)
{
    if (data is string)
    {
        data = Encrypt((string) data);
    }
    // ... //
}
```

# The as Operator

- The **as** operator verifies a data type and attempts to do a type conversion, assigning null if the type is not convertible.
- This operator therefore avoids throwing an exception when the types are incompatible.

```
// Data Conversion Using the as Operator
object Print(IDocument document)
{ // ... // }
static void Main()
{
    object data;
    Print(data as Document); }
```

# Unit 6 Summary

- This Unit discussed how to implement inheritance, i.e., how one class can inherit from another.
- The details of overriding the base-class implementation were studied.
- The Unit finished with a discussion of how all class types inherit from `System.Object`.
- The next Unit discusses interfaces.

# Unit 7

## Interfaces

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 7 Overview

- The previous Unit discussed inheritance and polymorphism.
- This Unit discusses defining, implementing, and using interfaces.



# Introducing Interfaces

- An interface is a set of abstract member signatures that define a particular type of functionality.
- Interfaces are often referred to as “contracts.”
- Interfaces have no implementation or data.
- All interface members are public, therefore no access modifiers are allowed.
- See Listing 7.2: Implementing and Using Interfaces for a good example of interfaces.

# Interface Example

// Defining an Interface

interface IFileCompression

{

void Compress(string targetFileName, string[] fileList);

void Uncompress(string compressedFileName,  
string expandDirectoryName);

}

# Interface Implementation

- Once a class declares that it implements an interface, all members of the interface must be implemented.
- Note that interfaces can never be instantiated and cannot have constructors or finalizers.
- Interfaces also cannot have static members.
- Interface members behave like an abstract method, forcing the derived class to implement the member.

# Interface Implementation Example

```
// Implementing an Interface
public class Contact : Pdaltem, IListable, IComparable
{
    #region IComparable Members
    public int CompareTo(object obj)
    {
        // ... //
    }
    #endregion

    #region IListable Members
    string[] IListable.ColumnValues
    {
        // ... //
    }
    #endregion
}
```

# Explicit Member Implementation

- Explicitly implemented interface methods can be invoked only through an interface reference.

```
// Calling Explicit Interface Member Implementations
```

```
string[] values;
```

```
Contact contact1, contact2;
```

```
// ... //
```

```
// ERROR: Unable to call ColumnValues() directly on a contact.
```

```
values = contact1.ColumnValues;
```

```
values = ((IListable)contact2).ColumnValues;
```

```
// ... //
```

# Explicit Interface Implementation

- To declare an explicit interface member implementation, prefix the member name with the interface name.

```
// Explicit Interface Implementation
public class Contact : PdaItem, IListable, IComparable
{
    IComparable Members

    #region IListable Members
    string[] IListable.ColumnValues
    { /* ... */ }
    #endregion
}
```

# Implicit Member Implementation

- With implicit member implementation, the class member's signature matches the interface member's signature with no prefix.
- Implicitly implemented members must have a **public** modifier and can be virtual if they can be overridden in a derived class.
- Code that calls an implicitly implemented member do so directly (no cast is required):  
→ `result = contact1.CompareTo(contact2);`

# Implicit Interface Implementation

```
// Implicit Interface Implementation
public class Contact : Pdaltem, IListable, IComparable
{
    #region IComparable Members
    public int CompareTo(object obj)
    { /* ... */ }
    #endregion

    IListable Members
}
```



# Explicit vs. Implicit Interface Implementation

- Guidelines for choosing between explicit and implicit member implementations:
  - Is the member a core part of class functionality?
  - Is the member name appropriate as a class member?
  - Is there already a member with the same name?
- The key difference between implicit and explicit interface implementation is in the visibility outside the class.

# Object to Interface Conversion

- A conversion from an object to its implemented interface is an implicit conversion--no cast operator is required.
- Conversions in the opposite direction, from an interface reference to an object reference, must be explicit.

# Interface Inheritance

- Interfaces can derive from each other, resulting in an interface that inherits all the members in its base interfaces.

# Interface Inheritance Example

```
// Deriving One Interface from Another
```

```
interface IReadableSettingsProvider  
{ string GetSetting(string name, string defaultValue); }
```

```
// Deriving One Interface from Another
```

```
interface ISettingsProvider : IReadableSettingsProvider  
{ void SetSetting(string name, string value); }
```

```
class FileSettingsProvider : ISettingsProvider  
{  
    #region ISettingsProvider Members  
    public void SetSetting(string name, string value) { /* ... */ }  
    #endregion  
    #region IReadableSettingsProvider Members  
    public string GetSetting(string name, string defaultValue) { /* ... */ }  
    #endregion  
}
```

# Multiple Interface Example

```
// Multiple Interface Inheritance
interface IReadableSettingsProvider
{ string GetSetting(string name, string defaultValue);    }

interface IWritableSettingsProvider
{ void SetSetting(string name, string value);    }

interface ISettingsProvider : IReadableSettingsProvider,
                              IWritableSettingsProvider
{ }
```

# Extension Methods on Interfaces

- Extension methods can be used with interfaces, too.

```
// Interface Extension Methods
static class Listable
{
    public static void List(this IListable[] items, string[] headers)
    {
        // ... //
    }
}
```

# Implementing Multiple Interfaces

- A single class can implement any number of interfaces in addition to deriving from a single class.
- This feature provides a possible workaround got the lack of support for multiple inheritance in C#.

# Facts about Interfaces

- Interfaces are the one type in .NET that does not inherit from System.Object.
- Interfaces cannot be instantiated and are only accessible via a reference to an object that implements the interface.
- Static members are not allowed on interfaces.
- Interfaces are similar to abstract classes—neither of which can be instantiated.



# Versioning

- Interfaces define a contract between the implementing class and the class using the interface.
- The contract says that any class implementing the interface must provide implementations for all member signatures.
- Moreover, if new members need to be added to the interface later, it is best to create a new interface inheriting from the old one.

# Abstract Classes vs. Interfaces

Abstract Classes	Interfaces
Cannot be instantiated independently from their derived classes. Abstract class constructors are called only by their derived classes.	Cannot be instantiated.
Define abstract member signatures that base classes must implement.	Implementation of all members of the interface occurs in the base class. It is not possible to implement only some members within the implementing class.
Are more extensible than interfaces, without breaking any version compatibility. With abstract classes, it is possible to add additional non-abstract members that derived classes can inherit.	Extending interfaces with additional members breaks the version compatibility.

# Abstract Classes vs. Interfaces (cont'd)

Abstract Classes	Interfaces
Can include data stored in fields.	Cannot store any data. Fields can be specified only on the deriving classes. The workaround for this is to define properties, but without implementation.
Allow for (virtual) members that have implementation and, therefore, provide a default implementation of a member to the deriving class.	All members are automatically virtual and cannot include any implementation.
Deriving from an abstract class uses up a subclass's one and only base class option.	Although no default implementation can appear, classes implementing interfaces can continue to derive from one another.

# Unit 7 Summary

- This Unit discussed interfaces, a critical part of the object-oriented programming framework, which provide functionality similar to abstract classes.
- Interfaces can be implemented implicitly or explicitly.
- The next Unit looks at value types and discusses the process of defining custom value types.

# Unit 8

## Value Types

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 8 Overview

- The previous Unit discussed defining, implementing, and using interfaces.
- This Unit discusses not only using value types but also defining custom value types.
- There are two categories of value types: structs, which allow the definition of new value types, and enums, which define sets of constant values.

# Value Types

- Value types contain their values, i.e., the variable name equates to the location in memory where the value is stored.
- When assigning a value-type variable to another value-type variable, a copy of the underlying value is made.
- Value types are stored in an area of memory called the **stack**.

# Reference Types

- Reference types and the variables that refer to them reference the data storage location.
- Reference types, therefore, store the reference (memory address) where the data is located, instead of containing the data, itself.
- This area of memory is referred to as the **heap**.
- When assigning one reference type to another, only a copy of the reference occurs, not the data.



# Important Fact About Types

- All primitive types in C# are value types except string and object.

# Creating Value Types With struct

- Developers can define their own value types that behave like user-defined primitives using the keyword **struct**.
- Value types should be immutable, although nothing in the language requires it.
- In addition to properties and fields, structs may contain methods and constructors, too.
- Default constructors are not allowed; however, parameterized constructors are.

# Rules for Value Types

- Value types (structs) do not support finalizers because memory is allocated on the stack.
- All value types are sealed and derive from `System.ValueType`; the inheritance chain for structs is object to `System.ValueType` to `struct`.
- Value types can implement interfaces.

# Boxing

- Boxing involves converting from a value type to a reference type.
- The reverse operation is unboxing, converting from a reference type to a value type.
- Boxing is sometimes a significant concern because it can affect both performance and type-safety.

# Boxing Example

```
// Subtle Box and Unbox Instructions
static void Main(string[] args)
{
    int totalCount;
    System.Collections.ArrayList list = new System.Collections.ArrayList();

    Console.Write("Enter a number between 2 and 1000:");
    totalCount = int.Parse(Console.ReadLine());
    list.Add((double)0);    // Boxing occurs here
    list.Add((double)1); // and here
    // Unboxing occurs within the for loop
    for (int count = 2; count < totalCount; count++)
    { list.Add(((double)list[count - 1] + (double)list[count - 2])); }

    foreach (double count in list)
    { Console.Write("{0}, ", count); } // Boxing occurs here, too
}
```

# Unboxing to the Wrong Type

```
// Unboxing Must Be to the Underlying Type
```

```
// ...
```

```
int number;
```

```
object thing;
```

```
double bigNumber;
```

```
number = 42;
```

```
thing = number;
```

```
// ERROR: InvalidCastException
```

```
// bigNumber = (double)thing;
```

```
bigNumber = (double)(int)thing;
```

```
// ...
```

# Avoiding Unboxing and Copying

```
// Avoiding Unboxing and Copying
```

```
int number;
```

```
object thing;
```

```
number = 42;
```

```
// Boxing
```

```
thing = number;
```

```
// No unbox instruction.
```

```
string text = ((IFormattable)thing).ToString("X", null);
```

```
Console.WriteLine(text);
```

# Enums

- An enum is a type that identifies a compile-time set of integer values, each value referred to by a name, making the code easier to read.

```
// Defining an Enum
enum ConnectionState
{
    Disconnected,
    Connecting,
    Connected,
    Disconnecting
}
```



# Enum Example

```
ConnectionState connectionState;  
// ...  
switch (connectionState)  
{  
    case ConnectionState.Connected:  
        // ...  
        break;  
    case ConnectionState.Connecting:  
        // ...  
        break;  
    case ConnectionState.Disconnected:  
        // ...  
        break;  
    case ConnectionState.Disconnecting:  
        // ...  
        break;  
}
```

# Facts About Enums

- To refer to an enum value, prefix it with the enum name, e.g., `ConnectionState.Connected`.
- By default, the first enum value is 0, and each subsequent entry increases by 1; however, explicit values can be assigned to enums.

```
enum ConnectionState : short
{
    Disconnected,
    Connecting = 10,
    Connected,
    Joined = Connected,
    Disconnecting
}
```

# More on Enums

- Enums have a default type of `System.Int32` but can be changed to any integer type.
- The base class for all enums is `System.Enum`.
- Enums support type conversion from integer types.
- The `ToString()` method writes out the enum value identifier:
  - `ConnectionState.Disconnecting.ToString()`

# Converting a String to an Enum

- The Enum.Parse() method can convert a string value to an enum, as can Enum.TryParse().

```
// Converting a String to an Enum Using Enum.Parse()
// Must use exception handling in case string doesn't convert
ThreadPriorityLevel priority = (ThreadPriorityLevel)Enum.Parse(
    typeof(ThreadPriorityLevel), "Idle");

Console.WriteLine(priority);

// Converting a String to an Enum Using Enum.TryParse()
// No need to use exception handling if string doesn't convert
System.Threading.ThreadPriorityLevel priority;
if(Enum.TryParse("Idle", out priority))
{
    Console.WriteLine(priority);
}
```

# Enums as Flags

- Enums that support combined values can be used as flags; they must include the `FlagsAttribute ([Flags])` in their declaration.
- The bitwise OR operator allows the combining of more than one bit flag; the bitwise AND operator can check for specific settings.
- The `System.Enum.IsDefined()` method checks whether a value is included as an enum.
- `ToString()` writes out the flags that are set.

# Defining Enum Values

```
// Defining Enum Values for Frequent Combinations
```

```
enum DistributedChannel
```

```
{
```

```
    None = 0,
```

```
    Transacted = 1,
```

```
    Queued = 2,
```

```
    Encrypted = 4,
```

```
    Persisted = 16,
```

```
    FaultTolerant =
```

```
        Transacted | Queued | Persisted
```

```
}
```

# Defining Enum Values (cont'd)

```
// FileAttributes defined in System.IO.  
// Using FlagsAttribute  
[Flags]  
public enum FileAttributes  
{  
    ReadOnly = 1 << 0,      // 0000000000000001  
    Hidden = 1 << 1,       // 0000000000000010  
    // ... //  
}
```

# Enum Values Example

```
// Using Bitwise OR and AND with Flag Enums
static void Main(string[] args)
{
    string fileName = @"enumtest.txt";

    System.IO.FileInfo file = new System.IO.FileInfo(fileName);
    file.Attributes = FileAttributes.Hidden | FileAttributes.ReadOnly;

    Console.WriteLine("{0} | {1} = {2}",
        FileAttributes.Hidden, FileAttributes.ReadOnly, (int)file.Attributes);

    if ((file.Attributes & FileAttributes.Hidden) != FileAttributes.Hidden)
        throw new Exception("File is not hidden.");

    if ((file.Attributes & FileAttributes.ReadOnly) != FileAttributes.ReadOnly)
        throw new Exception("File is not read-only.");
}
```



# Unit 8 Summary

- This Unit discussed how to define custom value types using structs.
- This Unit also introduced enums, a .NET type that helps improve code readability.
- Boxing was also explored, along with generics as a solution to boxing issues.
- The next Unit highlights additional guidelines for creating well-formed types.

# Unit 9

## Well-Formed Types

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 9 Overview

- The previous Units covered many of the constructs for defining classes and structs.
- This Unit discusses the details for adding the final touches to a type declaration.

# Overriding object Members

- All types derive from `System.Object`.
- This section discusses the details for overriding the virtual methods of this type: `ToString()`, `GetHashCode()`, and `Equals()`.

# Overriding ToString()

- By default, calling the ToString() method on an object returns its fully-qualified type name.
- The ToString() method can be overridden to return useful information; calling ToString() on a `string` variable, for example, returns its value.
- Write methods, such as Console.WriteLine(), call an object's ToString() method when passed an object as a parameter.

# Overriding ToString() Example

```
// Overriding ToString()
public struct Coordinate
{
    public Coordinate(Longitude longitude, Latitude latitude)
    {
        _Longitude = longitude;
        _Latitude = latitude;
    }

    private readonly Longitude _Longitude;
    public Longitude Longitude { get { return _Longitude; } }
    private readonly Latitude _Latitude;
    public Latitude Latitude { get { return _Latitude; } }

    public override string ToString()
    { return string.Format("{0} {1}", Longitude, Latitude); }
}
```

# Overriding GetHashCode()

- A hash code is a number corresponding to the value of an object, which is used to efficiently balance a hash table.
- Overriding the method GetHashCode() is a good practice when a good key is needed for a hash table.
- When overriding the Equals() method, GetHashCode() should also be overridden; the compiler will issue a warning to that effect.

# Guidelines for Implementing HashCode()

- Equal objects must have equal hash codes.
- The value returned by HashCode() should be constant over the life of an object.
- HashCode() should not return exceptions.
- Hash codes should be unique when possible.
- Hash codes should be optimized for performance and distributed evenly.
- Hash codes should use a secure algorithm.



# Overriding GetHashCode() Example

```
// Overriding GetHashCode() and generating a hash code
public struct Coordinate
{
    // ... //
    public override int GetHashCode()
    {
        int hashCode = Longitude.GetHashCode();

        // As long as the hash codes are not equal
        if (Longitude.GetHashCode() != Latitude.GetHashCode())
        {
            hashCode ^= Latitude.GetHashCode(); // eXclusive OR
        }
        return hashCode;
    }
    // ... //
}
```

# Overriding Equals()

- The Equals() method can be overridden to check whether two objects have the same identifying field data, but this is nonstandard.
- Value types, for example, override the Equals() implementation to compare their values.
- The default is that Equals() returns the same result as ReferenceEquals() does for objects.

# Overriding Equals() Example

```
// Overriding ToString(), GetHashCode(), and Equals()
public struct Coordinate
{
    // ... //
    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;
        if (this.GetType() != obj.GetType())
            return false;
        return Equals((Coordinate)obj);
    }
    public bool Equals(Coordinate obj)
    {
        return
            ((Longitude.Equals(obj.Longitude)) && Latitude.Equals(obj.Latitude));
    }
}
```

# Guidelines for Implementing Equals()

- Equals() , the == operator, and the != operator must be implemented together and should use the same algorithm.
- When implementing equality for a type, the GetHashCode() method should also be implemented.
- Equality methods should not return exceptions.
- When implementing IComparable, the equality methods should also be implemented.

# Operator Overloading

- Operator overloading refers to implementing an operator on a defined type.
- For example, the `string` class overloads the `+` operator to implement concatenation.
- Most binary (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`), comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`), logical (`&`, `|`, `!`) and unary operators (`+`, `-`, `!`, `~`, `++`, `--`, `true`, `false`) operators can be overloaded.
- Assignment operators cannot be overloaded.

# Overloading == and !=

```
public sealed class Coordinate
{
    // ... //
    public static bool operator ==(Coordinate leftHandSide,
                                   Coordinate rightHandSide)
    {
        // Check if leftHandSide is null. (operator== would be recursive)
        if (ReferenceEquals(leftHandSide, null))
        {
            // Return true if rightHandSide is also null but false otherwise.
            return ReferenceEquals(rightHandSide, null);
        }
        return (leftHandSide.Equals(rightHandSide));
    }
    public static bool operator !=(Coordinate leftHandSide,
                                   Coordinate rightHandSide)
    { return !(leftHandSide == rightHandSide); }
}
```

# Overloading – and + Operators

```
public struct Latitude
{
    // ... //
    public static Latitude operator -(Latitude latitude)
    {
        return new Latitude(-latitude.DecimalDegrees);
    }

    public static Latitude operator +(Latitude latitude)
    {
        return latitude;
    }
}
```

# The Cast Operator ()

- Defining a conversion operator is common terminology for implementing explicit or implicit conversion.
- Implementing a conversion operator is similar to defining any other operator.



# Defining Implicit Conversion

// Providing an Implicit Conversion between Latitude and double

```
public struct Latitude
{
    // ...
    public Latitude(double decimalDegrees)
    { _DecimalDegrees = Normalize(decimalDegrees); }

    public double DecimalDegrees
    { get { return _DecimalDegrees; } }

    private readonly double _DecimalDegrees;
    // ...
    public static implicit operator double(Latitude latitude)
    { return latitude.DecimalDegrees; }

    public static implicit operator Latitude(double degrees)
    { return new Latitude(degrees); }
}
```

# Guidelines for Conversion Operators

- Conversion operators that throw exceptions should always be defined as explicit.
- Any conversions that could lose data (e.g., converting from a `float` to an `int`) and not successfully convert back to the original type should be defined as explicit.

# Class Libraries

- **Class libraries**, or just libraries, help organize the complexity of an application by breaking up programs into separate modules, which are compiled.
- Applications can reference these compiled libraries, called **assemblies**, to incorporate their functionality.
- Class libraries are thus reusable across different applications.

# Assembly Types

- Console executable: default assembly type.
- Class library: classes that can be shared across different applications.
- Windows executable: executables designed to run in the Microsoft Windows family of operating systems.
- Module: a part of an assembly; multiple modules can be combined to form an assembly.

# Using Assemblies

// Compiling an assembly (on the command-line)

```
>csc /target:library /out:Coordinates.dll Coordinate.cs IAngle.cs  
Latitude.cs Longitude.cs Arc.cs  
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1  
Copyright (C) Microsoft Corporation. All rights reserved.
```

// Referencing an assembly (on the command-line)  
csc.exe /R:Coordinates.dll Program.cs

# Access Modifiers for Class Types

- Access modifiers at the class level provide for encapsulation of classes within an assembly.
- By default, a class without an access modifier is defined as **internal**, meaning it is inaccessible from outside the assembly.
- Classes may be **public** or **internal** in an assembly; to expose a class, it must be public.
- Members may not have greater accessibility than the type that contains them.

# Type Member Accessibility

Modifier	Description
<b>public</b>	Declares that the member is accessible anywhere that the type is accessible. If the class is internal, the member will be internally visible. Public members will be accessible from outside the assembly if the containing type is public.
<b>internal</b>	The member is accessible from within the assembly only.
<b>private</b>	The member is accessible from within the containing type, but inaccessible otherwise. This is the default access level.
<b>protected</b>	The member is accessible within the containing type and any subtypes derived from it, regardless of assembly.
<b>protected internal</b>	The member is accessible from anywhere within the containing assembly <i>and from any types derived from the containing type</i> , even if the derived types are within a different assembly.

# Defining Namespaces

- Namespaces are named groupings of types that exist to help prevent naming collisions when types have the same name.
- Classes that are not in an explicit namespace are said to be in the **global** namespace.
- Namespaces often contain the company name and/or the product name.



# More on Namespaces

- The namespace combined with the short class name provides the fully-qualified class name.
- Namespaces support nesting, which provides for a hierarchical organization of classes.

```
// Define the simple namespace AddisonWesley
namespace AddisonWesley
{
    class Program
    {
        // ... //
    }
}
```

# Nested Namespace Example

```
// Nesting Namespaces within Each Other
namespace AddisonWesley
{
    // Define the namespace AddisonWesley.Michaelis
    namespace Michaelis
    {
        // Define the namespace
        // AddisonWesley.Michaelis.EssentialCSharp
        namespace EssentialCSharp
        {
            // Declare the class
            // AddisonWesley.Michaelis.EssentialCSharp.Program
            class Program
            { // ... // }
        }
    }
}
```

# XML Comments

- XML comments can be used to generate source-code documentation for a program.
- C# defines an explicit set of XML comment tags.
- Various utilities such as NDoc can extract the XML comments and create help files.
- For these comments to be useful, the developer must take the time to enter the XML comments into the code and create the XML file.

# XML Comments in Visual Studio

```

- /// <summary>
-   /// Display the text specified
-   /// </summary>
-   /// <param name="text">The text to be displayed in the console.</param>
- private static void Display(string text)
- {
-     Console.WriteLine(text);
- }
-
- static void Main()
- {
-     Display(|
-         void Program.Display(string text)
-         Display the text specified
-         text: The text to be displayed in the console.
-     )
- }
```

# Generating XML Documentation

**>csc /doc:Comments.xml DataStorage.cs**

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>DataStorage</name>
  </assembly>
  <members>
    <member name="T:DataStorage">
      <summary>
        DataStorage is used to persist and retrieve
        employee data from the files.
      </summary>
    </member>
    <member name="M:DataStorage.Store(Employee)">
      <summary>
        Save an employee object to a file named with the Employee name.
      </summary>
    </member>
  </members>
</doc>
```

# Garbage Collection

- The garbage collector manages memory consumed by objects on the heap.
- During a garbage collection, the garbage collector restores memory for objects no longer referenced and compacts memory.
- Garbage collections are handled automatically by the garbage collector but can be forced by calling the method `GC.Collect()`.

# Weak References

- Weak object references do not prevent an object from being garbage-collected while maintaining a reference to the object.
- Weak references are designed for objects that are expensive to create and keep alive.
- Weak references therefore serve as a type of memory cache; if the object has been removed from the cache, it can be recreated, if necessary.

# Weak Reference Example

```
// Using a Weak Reference
private WeakReference Data;
public FileStream GetData()
{
    FileStream data = (FileStream)Data.Target;
    if (data != null)
    {
        return data;
    }
    else
    {
        // Load data here
        // ... //
    }
    return data;
}
```



# Finalizers

- Finalizers are methods that run when an object is destroyed; they may be used to clean up resources.
- The garbage collector alone is responsible for calling the finalizer on an object instance, i.e., .NET uses **nondeterministic finalization**.
- The finalizer may not execute until the process terminates, but it is guaranteed to execute.
- A finalizer's declaration looks like a destructor.

# Finalizer Example

```
// Defining a Finalizer
public class TemporaryFileStream
{
    // Constructor
    public TemporaryFileStream()
    {
        // ... //
    }

    // Finalizer
    ~TemporaryFileStream()
    {
        // ... //
    }

    // ... //
}
```

# More Facts about Finalizers

- Finalizers are declared with the ‘~’ symbol.
- Finalizers allow no parameters or overloading
- Only the garbage collector can call a finalizer; it cannot be called explicitly.
- Finalizers can be used to free unmanaged resources; but the use of finalizers is not recommended because finalization is not determinate.

# The IDisposable Interface

- Finalizers cannot be called explicitly, but often there is a need to execute cleanup code.
- The IDisposable interface defines a method called `Dispose()`, which can be called manually for the purpose of executing cleanup code.
- Finalizers can then serve as a backup mechanism for cleaning up code in case `Dispose()` is never called by the application.

# The IDisposable Interface

```
namespace System
{
    // Summary:
    //   Defines a method to release allocated resources.
    [ComVisible(true)]
    public interface IDisposable
    {
        // Summary:
        //   Performs application-defined tasks associated with freeing,
        //   releasing, resetting unmanaged resources.
        void Dispose();
    }
}
```

# Implementing IDisposable

```
using System;
using System.IO;
public class TemporaryFileStream : IDisposable
{
    public TemporaryFileStream()
    { // ... // }

    ~TemporaryFileStream()
    { Close(); }

    // ... //

    #region IDisposable Members
    public void Dispose()
    { Close(); }
    #endregion
}
```

# The using Statement

- The using statement guarantees that the `Dispose()` method is called on any object implementing the `IDisposable` interface.
- One using statement can contain more than one variable declaration, but all variables referenced in the **using** statement must implement `IDisposable`.

# Invoking the using Statement

```
// Invoking the using Statement
static void Search()
{
    using (TemporaryFileStream fileStream1 = new TemporaryFileStream(),
           fileStream2 = new TemporaryFileStream())
    {
        // Use temporary file stream
    }
}
```



# Finalization Guidelines

- Finalization delays garbage collection, so use it sparingly.
- Objects with finalizers should implement `IDisposable`.
- Finalizers should invoke the same code as that called by the `Dispose()` method.
- The `Dispose()` method should call `System.GC.SuppressFinalize` after executing.

# Finalization Guidelines (cont'd)

- Code in `Dispose()` may be called multiple times.
- Cleanup code should be simple.
- If the base class implements `Dispose()` then the derived implementation should call the base-class implementation.
- Objects should be flagged as unusable after `Dispose()` is called and call the `ObjectDisposedException`, if necessary.
- Avoid causing any exceptions in a finalizer.

# Lazy Initialization

- Lazy Initialization means to create objects only when they are needed, not before.
- If the object is not needed, it is never instantiated.

# Lazy Initialization Example

```
class DataCache
{
    // Lazy Loading a Property
    public TemporaryFileStream FileStream
    {
        get
        {
            if (_FileStream == null)           // Not loaded until needed
            {
                _FileStream = new TemporaryFileStream();
            }
            return _FileStream;
        }
    }
    private TemporaryFileStream _FileStream = null;
    // ... //
}
```

# Lazy Loading with Generics

- .NET 4.0 provides a `System.Lazy<T>` class that takes a type parameter (T) that identifies what type the `Value` property on `System.Lazy<T>` will return when called.
- Instead of assigning a fully-constructed object to the field, a light-weight instance of `System.Lazy<T>` is assigned, delaying the instantiation of the actual object of type T until `Value` is called.

# Lazy Loading a Property

```
// Lazy Loading a Property
using System.IO;
class DataCache
{
    // ... //
    public string FileStreamName { get; set; }
    public DataCache()
    { // Note the Lambda expression
        _FileStream = new Lazy<TemporaryFileStream>(
            () => new TemporaryFileStream(FileStreamName));
    }
    public TemporaryFileStream FileStream
    {
        get
        { return _FileStream.Value; } // Actual Instantiation of T
    }
    private Lazy<TemporaryFileStream> _FileStream;
}
```

# Unit 9 Summary

- This Unit provided a tour of topics related to building robust class libraries.
- Topics covered included namespaces, garbage collection, `System.Object`, overriding virtual members, operator overloading, and XML comments.
- In the next Unit exception handling will be discussed.

# Unit 10

## Exception Handling

C# Data Structures and Design Patterns  
Spring Quarter, 2017



# Unit 10 Overview

- In Unit 4, the `try...catch...finally` block, used for exception handling, was discussed.
- This Unit discusses more details of exception handling including additional exception types, custom exceptions, and using multiple catch blocks for handling different exception types.

# Specific Exception Types

- The **throw** keyword is used to notify the application of an exception (error condition).
- All exceptions inherit from `System.Exception`.
- Specific exception types such as `ArgumentException`, `ArgumentNullException`, and `NullReferenceException` provide additional information about the error.

# Example of Throwing an Exception

```
// Throwing an Exception
public sealed class TextNumberParser
{
    public static int Parse(string textDigit)
    {
        string[] digitTexts = { "zero", "one", "two", "three", "four",
                                "five", "six", "seven", "eight", "nine" };
        int result = Array.IndexOf(digitTexts, textDigit.ToLower());
        if (result < 0)
        {
            throw new ArgumentException(
                "The argument did not represent a digit", "textDigit");
        }
        return result;
    }
}
```

# Catching Exceptions

- C# allows the use of multiple `catch` blocks.
- When an exception occurs, execution jumps to the `catch` block with the best-matching exception type.
- The best match is determined by navigating the inheritance chain from the Exception class.
- Catch blocks appear in order from the most derived (specific) to the most general.

# Catching Different Exceptions

```
// Catching Different Exception Types
try
{
    // ... //
    throw new InvalidOperationException("Arbitrary exception");
}
catch (NullReferenceException exception)
{ /* Handle NullReferenceException */ }
catch (ArgumentException exception)
{ /* Handle ArgumentException */ }
catch (InvalidOperationException exception)
{ /* Handle ApplicationException */ }
catch (SystemException)
{ /* Handle SystemException */ }
catch (Exception exception)
{ /* Handle Exception */ }
}
```

# Generic Exceptions

- All exception types extend `System.Exception`.
- `System.Exception` **catch** blocks catch all exceptions.
- Similarly, a generic catch block (`catch {...}`) acts like a `System.Exception` catch block but does not instantiate an exception type object.
- A generic catch block must always be the last catch block.

# Guidelines for Exception Handling

- Catch only exceptions that can be handled.
- Don't hide exceptions that can't be handled.
- Avoid `System.Exception` and generic catch blocks.
- Avoid exception reporting or logging too low in the call stack.

# Guidelines (cont'd)

- Use `throw` by itself rather than `throw <exception object>` inside a catch block in order to preserve the stack trace.
- Use caution when rethrowing different exceptions.



# Rethrowing an Exception

- Rethrow an exception in the following situations:
  - Changing the exception type clarifies the problem.
  - Private data is part of the original exception.
  - The exception type is too specific for the caller to handle appropriately.

# Defining Custom Exceptions

- Generally, it is preferable to use the .NET Framework exception classes because they are well established.
- Sometimes a custom exception class is needed when special handling or actions are required.
- Defining a custom exception involves deriving from `System.Exception` or some other exception type.

# Creating a Custom Exception

```
// Creating a Custom Exception
class DatabaseException : System.Exception
{
    public DatabaseException()
    { /* ... */ }
    public DatabaseException(string message)
    { /* ... */ }

    public DatabaseException(
        System.Data.SqlClient.SqlException exception)
    { InnerException = exception; /* ... */ }
    public DatabaseException(
        System.Data.OracleClient.OracleException exception)
    { InnerException = exception; /* ... */ }
    public DatabaseException(string message, Exception innerException)
    { InnerException = innerException; /* ... */ }
}
```

# More on Custom Exceptions

- Custom exceptions must derive from `System.Exception` or one of its descendants.
- Best practice is for custom exceptions to use the “Exception” suffix.
- Custom exceptions should have at least three constructors: (1) a default constructor; (2) a constructor taking a message string parameter; and (3) a constructor taking a message string parameter and an inner exception parameter.

# Integer Overflow

- By default, C# does not implement overflow checking.

```
public static void Main()
{
    // Overflowing an Integer Value
    // int.MaxValue equals 2147483647
    int n = int.MaxValue;
    n = n + 1;
    Console.WriteLine(n); // No exception: result is -2147483648
}
```

# Checked Keyword

- Placing a numeric assignment inside a **checked** block instructs the CLR to check for arithmetic overflow and underflow.
- If an overflow occurs, an `OverflowException` is thrown.

```
checked // A Checked Block Example
{
    // int.MaxValue equals 2147483647
    int n = int.MaxValue;
    n = n + 1; // OverflowException is thrown here
    Console.WriteLine(n);
}
```

# Unchecked Keyword

- Placing a numeric assignment inside an unchecked block instructs the CLR not to check for arithmetic overflow and underflow.
- If an overflow occurs, no exception is thrown.

```
// A Unchecked Block Example
unchecked
{
    // int.MaxValue equals 2147483647
    int n = int.MaxValue;
    n = n + 1;
    Console.WriteLine(n); // No exception: result is -2147483648
}
```

# Unit 10 Summary

- This Unit discussed built-in exceptions, custom exceptions, and proper exception handling.
- In the next Unit, generics will be discussed along with the new `System.Collections.Generic` namespace.



# Unit 11

## Generics

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 11 Overview

- This Unit discusses generics, a type-safe means to define types and method implementations once, rather than for each type.
- Generics are similar to templates in the C++ and Java languages.
- Parameterized types are defined such that the types of data and method signatures vary based on the program's needs.

# C# Without Generics Example

- The example in Listing 11.2 shows the use of a stack type collection to facilitate multiple undo operations.
- The previous move is saved by pushing a custom type, Cell, onto the stack using the Stack.Push() method.
- If the user enters a Z (Ctrl+Z), the previous move is undone by calling the Pop() method.

# C# Without Generics Example

// The Stack Definition Using a Data Type Object

```
public class Stack
{
    public virtual object Pop();
    public virtual void Push(object obj);
    // ...
}
```

```
public struct Cell
{
    readonly public int X;
    readonly public int Y;
    public Cell(int x, int y)
    {
        X = x;
        Y = y;
    }
}
```

# C# Without Generics Issues

- The `System.Collections.Stack` class places variables of type `object` onto a stack.
- Every object in .NET derives from `System.Object`, so any object can be pushed onto the stack without any validation.
- When popping an item off the stack, it must be converted back to its native type; if the item isn't the right type, an exception is thrown.
- Value types require both boxing and unboxing.

# Nullable Types and Generics

- Before generics there were only two options for implementing nullable types:
  - Declare a nullable data type for each value type that needs to handle null values.
  - Declare a nullable type that contains a Value property of type object.

# Nullable Types Before Generics

// 1. Declaring Versions of Various Value Types That Store null

struct NullableInt

{

/// <summary>Provides value when HasValue returns true.</summary>

public int Value { get; set; }

/// <summary>Indicates if there is a value or if the value is null</summary>

public bool HasValue { get; set; }

// ... //

}

// 2. Declaring a Nullable Type That Contains a Value Property of Type object

struct Nullable

{

/// <summary>Provides value when HasValue returns true</summary>

public object Value { get; set; }

/// <summary>Indicates if there is a value or if the value is null</summary>

public bool HasValue { get; set; }

// ... //

}

# A Generic Class

- When declaring a variable using a generic data type, the actual type must be identified.
- In Listing 11.6, a new instance of the `System.Collections.Generic.Stack<T>` is declared.
- Every object that is added or retrieved from the stack must be of the same type.

```
// A Generic Stack class  
Stack<Cell> path; // Generic variable declaration  
path = new Stack<Cell>(); // Generic object instantiation
```



# Generic Type Substitution

- A type parameter identifier or type parameter (T) is specified within the angle brackets after the class declaration:  
→ `System.Collections.Generic.Stack<T>`
- Instances of Stack look for the type corresponding to the variable declaration.
- The parameter T is a placeholder until variable declaration and instantiation.

# The Generic Stack<T> Class

```
// Declaring a Generic Class, Stack<T>
public class Stack<T>
{
    private T[] _Items;
    public void Push(T data)
    {
        // ... //
    }
    public T Pop()
    {
        // ... //
    }
}
```

# Benefits of Generics

- Generic types implement a strongly-typed programming model.
- Compile-time checking reduces the likelihood of `InvalidCastException` errors at runtime.
- Value types do not require boxing operations.
- Performance is enhanced because no type conversions are required.
- Generics reduce code bloat.

# Benefits of Generics (cont'd)

- Memory consumption is reduced because boxing is avoided.
- Code is more readable because the need to do type conversions and type checking is gone.
- The code editor assists through IntelliSense with the programming task.
- Code patterns can be implemented and reused wherever the patterns appear.

# Type Parameter Naming Guidelines

- It is customary to use the “T” prefix to distinguish a type parameter:
  - `EntityCollection<TEntity>`
- “T” can be used when a specific parameter is not needed:
  - `Stack<T>`

# Generic Interfaces and Structs

- Interfaces and structs can use generics, too.

```
// Declaring a Generic Interface
interface IPair<T>
{
    T First { get; set; }
    T Second { get; set; }
}
```

# Generic Struct with Interface

// Implementing a Generic Interface and declaring a generic constructor

```
public struct Pair<T> : IPair<T>
{
    public Pair(T first, T second)
    {
        _First = first;
        _Second = second;
    }

    private T _First;
    public T First
    {
        get { return _First; }
        set { _First = value; }
    }

    private T _Second;
    public T Second
    {
        get { return _Second; }
        set { _Second = value; }
    }
}
```

# Specifying a Default Value

- The `default` keyword provides the default value of any data type.
- For example, `default(int)`, for example, is the value 0, and `default(string)` is null.

```
// Initializing a Field with the default Operator
public struct Pair<T> : IPair<T>
{
    public Pair(T first, T second)
    {
        _First = first;
        _Second = default(T);
    }
    // ...
}
```



# Multiple Type Parameters

- Generic types can have more than one type parameter.
- The number of type parameters is called the **arity** and uniquely distinguishes the class.

```
// Using a Type with Multiple Type Parameters
Pair<int, string> historicalEvent = new Pair<int, string>(1914,
    "Shackleton leaves for South Pole on ship Endurance");
Console.WriteLine("{0}: {1}", historicalEvent.First,
    historicalEvent.Second);
```

# Multiple Type Parameter Example

```
// Declaring a Generic with Multiple Type Parameters
interface IPair<TFirst, TSecond>
{
    TFirst First { get; set; }
    TSecond Second { get; set; }
}
public struct Pair<TFirst, TSecond> : IPair<TFirst, TSecond>
{
    public Pair(TFirst first, TSecond second)
    {
        _First = first;
        _Second = second;
    }
    // ... //
}
```

# Arity in Abundance

- In C# 4.0, there are nine new generic types called **Tuple**, each with a different **arity**.

```
// Covariance Using the out Type Parameter Modifier
public class Tuple { ... }
public class Tuple<T1>:
    IStructuralEquatable, IStructuralComparable, IComparable {...}
public class Tuple<T1, T2>: ... {...}
public class Tuple<T1, T2, T3>: ... {...}
public class Tuple<T1, T2, T3, T4>: ... {...}
public class Tuple<T1, T2, T3, T4, T5>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6, T7>: ... {...}
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>: ... {...}
```

# Notes on Tuples

- The **TRest** type parameter can be used to store another Tuple, making the potential size of the tuple effectively unlimited.
- The Tuple with no type parameters has eight static Create() methods for instantiating various generic tuple types.

```
// Covariance Using the out Type Parameter Modifier
Tuple<string, Contact> keyValuePair;
keyValuePair = Tuple.Create(
    "555-55-5555", new Contact("Inigo Montoya"));
keyValuePair = new Tuple<string, Contact>(
    "555-55-5555", new Contact("Inigo Montoya"));
```

# Nested Generic Types

- Typed parameters on the containing type will cascade down to the nested type.
- Reuse of the same type parameter in a nested type will cause a compiler warning.

```
class Container<T, U>
{ // Nested classes inherit type parameters.
  class Nested<U>
  { // Reusing a type parameter name will cause a warning.
    void Method(T param0, U param1)
    { /* ... */ }
  }
}
```

# Generic Constraints

- Generics support constraints on the types allowed for type parameters.
- A generic type constraint is declared using the **where** keyword following by type constraint.
- The use of constraints prevents errors when defining the generic type or method rather than at execution time.
- Constraint types include interface, base class, struct, class, and default constructor.

# Interface Constraints

- An interface constraint requires that a particular interface be implemented on the generic type parameter.

```
// Declaring an Interface Constraint
public class BinaryTree<T> where T: System.IComparable<T>
{
    // ... //
}
```

# Base Class Constraints

- A generic base-class constraint limits the constructed type to a particular class derivation.
- Base class constraints must appear first when multiple constraints are specified.

```
// Declaring a Base Class Constraint
public class EntityDictionary<TKey, TValue>
    : System.Collections.Generic.Dictionary<TKey, TValue>
    where TValue : EntityBase
{
    // ... //
}
```



# struct/class Constraints

- A **struct** or **class** constraint restricts the generic type parameter to be a value type or reference type.
- A base class constraint cannot be used with either the **struct** or **class** constraint.

```
// Specifying the Type Parameter As a Value Type
public struct Nullable<T> : IFormattable, IComparable,
    IComparable<Nullable<T>>, INullable where T : struct
{
    // ... //
}
```

# Multiple Constraints

//Specifying Multiple Constraints

```
public class EntityDictionary<TKey, TValue>: Dictionary<TKey, TValue>
    where TKey : IComparable<TKey>, IFormattable
    where TValue : EntityBase
{
    // ... //
}
```

# Constructor Constraints

- In some cases, it is desirable to instantiate an instance of a type parameter inside the generic class.
- The constructor constraint requires that the constrained type have a default constructor.

```
// Requiring a Default Constructor Constraint
public class EntityDictionary<TKey, TValue> : Dictionary<TKey, TValue>
    where TKey: IComparable<TKey>, IFormattable
    where TValue : EntityBase<TKey>, new()
{ /* ... */ }
```

# Constraint Inheritance

- Constraints are inherited by a derived class, but they must be specified explicitly on the derived class.
- In contrast, constraints on generic overridden methods or explicit interface methods are inherited implicitly.

# Constraint Inheritance Example

```
// Inherited Constraints Specified Explicitly
public class EntityDictionary<TKey, TValue>
    : System.Collections.Generic.Dictionary<TKey, TValue>
        where TValue : EntityBase
{ /* ... */ }

class Entity<T> : EntityBase<T>
{
    // Error: Constraints may not be repeated on overriding members
    public virtual void Method<T>(T t) where T : IComparable<T>
    { /* ... */ }
}
```

# Generic Constraint Limitations

- A base class constraint cannot be combined with a struct or class constraint.
- `Nullable<T>` cannot be used on struct constraints.
- Constraints cannot restrict inheritance to special types such as `object`, arrays, `System.ValueType`, `System.Enum`, `System.Delegate`, or `System.MulticastDelegate`.
- More examples follow on upcoming slides.

# No Operator Constraints

- Operator constraints are not allowed.

```
// Constraint Expressions Cannot Require Operators
public abstract class MathEx<T>
{
    public static T Add(T first, T second)
    {
        // Error: Operator '+' cannot be applied to operands of type 'T' and 'T'.
        return first + second;
    }
}
```

# OR Criteria Not Supported

- When multiple interfaces or class constraints are supplied for a type parameter, the compiler assumes an AND relationship.

```
// Combining Constraints Using an OR Relationship Is Not Allowed
public class BinaryTree<T>
// Error: OR is not supported.
where T: System.IComparable<T> || System.IFormattable
{
    // ... //
}
```



# No Delegate and Enum Constraints

```
// Inheritance Constraints Cannot Be of Type System.Delegate
// Error: Constraint cannot be special class 'System.Delegate'
public class Publisher<T> where T : System.Delegate
{
    public event T Event;
    public void Publish()
    {
        if (Event != null)
        {
            Event(this, new EventArgs());
        }
    }
}
```

# Constraints Only for Default Constructors

- There is no constraint to force a type T to support a non-default constructor.
- One way to circumvent this restriction is to supply a factory interface that includes a method for instantiating the type.

```
public TValue New(TKey key)
{ // Error: 'TValue': Cannot provide arguments when creating instance of type
  TValue newEntity = null;
  // newEntity = new TValue(key);
  Add(newEntity.Key, newEntity);
  return newEntity;
}
```

# Generic Methods

- Generic methods use generics even when the parent class is not a generic class.
- To define a generic method, add the type parameter after the method name.

```
// Defining Generic Methods
public static class MathEx
{
    public static T Max<T>(T first, params T[] values)
        where T: IComparable<T>
    { /* ... */ }
}
```

# Generic Type Inference

- Specifying the generic type in a generic method call is redundant if the compiler can infer the type from the method arguments.

```
// Specifying the Type Parameter Explicitly
```

```
Console.WriteLine(MathEx.Max<int>(7, 490));
```

```
Console.WriteLine(MathEx.Min<string>("R.O.U.S.", "Fireswamp"));
```

```
// Inferring the Type Parameter
```

```
Console.WriteLine(MathEx.Max(7, 490));
```

```
Console.WriteLine(MathEx.Min("R.O.U.S.", "Fireswamp"));
```

# Covariance and Contravariance

- Covariance and Contravariance are not supported by generic types.
- Covariance means assigning a more derived type to a less derived type.
- Contravariance is the assigning of a less derived type to a more derived type.

# Unit 11 Summary

- This Unit discussed the use of generics in C#.
- Generics improve type safety, avoid casting, and reduce code bloat; they also eliminate boxing of value types and improve performance.
- Types defined in the `System.Collections` namespace should now be replaced by types defined in the `System.Collections.Generic` namespace.

# Unit 12

## Delegates and Lambda Expressions

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 12 Overview

- The use of methods as a data type and their support for publish-subscribe patterns is the focus of this Unit.
- In this vein, both delegates and lambda expressions will be explored, along with anonymous methods.



# Delegates

- Delegates encapsulate methods as objects, enabling an indirect method call to be bound at runtime.
- All delegates derive from `System.Delegate`.
- The `Delegate` type has two properties:
  1. `Method`, of type `MethodInfo`, gets the method represented by the delegate.
  2. `Target` gets the class instance on which the current delegate invokes the instance method.

# Defining a Delegate Type

- C# does not allow a class to inherit directly from System.Delegate; instead, the delegate keyword is used to declare a delegate type.

```
// Declaring a Delegate Data Type
```

```
public delegate bool ComparisonHandler(int first, int second);
```

```
// Declaring a Nested Delegate Data Type
```

```
class DelegateSample
```

```
{
```

```
    public delegate bool ComparisonHandler(int first, int second);
```

```
}
```

# Instantiating a Delegate

- A method that corresponds to the signature of the delegate type itself is needed first.
- The name of the method is not significant.
- Prior to C# 2.0, the delegate instance had to be instantiated with the name of the method it encapsulated.
- After C# 2.0, **delegate inference** allows the application to pass the name of the method rather than using explicit instantiation.

# Delegate Example

```
//Passing a Delegate Instance As a Parameter in C# 2.0
public delegate bool ComparisonHandler (int first, int second);
class DelegateSample
{
    public static void BubbleSort(int[] items, ComparisonHandler
                                comparisonMethod)
    { /* ... */ }
    public static bool GreaterThan(int first, int second)
    { return first > second; }
    // ... //
}
static void Main()
{
    // ... //
    BubbleSort(items, GreaterThan);
    // ... //
}
```

# Delegate Example (before C# 2.0)

**//Passing a Delegate Instance As a Parameter before C# 2.0**

```
public delegate bool ComparisonHandler (int first, int second);  
class DelegateSample  
{  
    public static void BubbleSort(int[] items, ComparisonHandler  
                                comparisonMethod)  
    { /* ... */ }  
    public static bool GreaterThan(int first, int second)  
    { return first > second; }  
    // ... //  
}  
static void Main()  
{  
    // ... //  
    BubbleSort(items, new ComparisonHandler(GreaterThan));  
    // ... //  
}
```

# Alternative ComparisonHandler

```
// Using a Different ComparisonHandler-Compatible Method
public static bool AlphabeticalGreaterThan(int first, int second)
{
    int comparison = (first.ToString().CompareTo(second.ToString()));
    return comparison > 0;
}

static void Main(string[] args)
{
    // ... //
    BubbleSort(items, AlphabeticalGreaterThan);
    // ... //
}
```

# Anonymous Methods

- Anonymous methods are delegate instances with no actual method declaration; instead they are defined inline in the program code.

```
// Passing an Anonymous Method
ComparisonHandler comparisonMethod;
// ... //
comparisonMethod =
    delegate(int first, int second)
    {
        return first < second;
    };
BubbleSort(items, comparisonMethod);
```

# Anonymous Method Example

```
// Using an Anonymous Method without Declaring a Variable
BubbleSort(items,
    delegate(int first, int second)
    {
        return first < second;
    }
);
```



# System-Defined Delegates: Func<>

- .NET 3.5 included a series of predefined generic delegates: `System.Func<>` represents delegates that have return types while `System.Action<>` represents delegates with no return type.
- Some examples follow on the next two slides.

# Func<> Examples

// Func Delegate Declarations

```
public delegate TResult Func<out TResult>();
```

```
public delegate TResult Func<in T, out TResult>(T arg)
```

```
public delegate TResult Func<in T1, in T2, out TResult>(
    in T1 arg1, in T2 arg2)
```

```
public delegate TResult Func<in T1, in T2, in T3, out TResult>(
    T1 arg1, T2 arg2, T3 arg3)
```

```
public delegate TResult Func<in T1, in T2, in T3, in T4,
    out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

...

```
public delegate TResult Func<
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16,
    out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4,
        T5 arg5, T6 arg6, T7 arg7, T8 arg8,
        T9 arg9, T10 arg10, T11 arg11, T12 arg12,
        T13 arg13, T14 arg14, T15 arg15, T16 arg16)
```

# Action<> Examples

// Action Delegate Declarations

```
public delegate void Action ();
```

```
public delegate void Action<in T>(T arg)
```

```
public delegate void Action<in T1, in T2>(
    in T1 arg1, in T2 arg2)
```

```
public delegate void Action<in T1, in T2, in T3>(
    T1 arg1, T2 arg2, T3 arg3)
```

```
public delegate void Action<in T1, in T2, in T3, in T4>(
    T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

...

```
public delegate void Action<
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16>(
    T1 arg1, T2 arg2, T3 arg3, T4 arg4,
    T5 arg5, T6 arg6, T7 arg7, T8 arg8,
    T9 arg9, T10 arg10, T11 arg11, T12 arg12,
    T13 arg13, T14 arg14, T15 arg15, T16 arg16)
```

# Func<> and Action<> Delegates

- Func<> delegates avoid the necessity of defining custom delegates in many situations.
- The last type of the Func<> is always the return type of the delegate; the preceding type parameters are the delegate parameters.
- Action<> delegates are be used when there is no return type.

# Contravariance and Covariance

```
// Using Variance for Delegates
// Contravariance
Action<object> broadAction = delegate(object data)
{ Console.WriteLine( data); }
Action<string> narrowAction = broadAction;

// Covariance
Func<string> narrowFunction = delegate()
{ return Console.ReadLine(); };
Func<object> broadFunction = narrowFunction;

// Contravariance & Covariance Combined
Func<object, string> func1 = delegate(object data)
{ return data.ToString(); };
Func<string, object> func2 = func1;
```

# Lambda Expressions

- Lambda expressions are a more succinct syntax for anonymous functions, which comprise both lambda expressions and anonymous methods.
- Statement lambdas include the delegate keyword and the lambda operator, `=>`, which is read as “go/goes to.”

```
// Passing a Delegate with a Statement Lambda
```

```
BubbleSort(items, (int first, int second) => {return first < second;} )
```

# Omitting Lambda Parameter Types

- In general, statement lambdas do not need parameter types as long as the compiler can infer the types or can implicitly convert the parameters to the requisite expected types.

```
// Omitting Parameter Types from Statement Lambdas
```

```
BubbleSort(items, (first, second) => {return first < second; } );
```

# Expression Lambda

- An expression lambda has only an expression, with no statement block, unlike a statement lambda which has a statement block with curly braces and a return statement.
- Sometimes it is clearer with an expression lambda to read the lambda operator as “such that.”

```
// Passing a Delegate with a Statement Lambda  
BubbleSort(items, (first, second) => first < second; );
```



# Lambda Expression Rules

- Lambda expressions do not have type; therefore, there are no members to call from a lambda expression.
- Since a lambda expression doesn't have a type, it can't appear to the right of an **is** operator.
- Once assigned or cast, the lambda expression does take on a type.
- A lambda expression cannot be assigned to an implicitly typed local variable.

# Outer Variables

- Local variables declared outside a lambda expression but captured within the lambda expression are called **outer variables**.
- Outer variables captured by lambda expressions live on until the corresponding delegate is destroyed.

# Expression Trees

- Lambda expressions that represent data about expressions rather than compiled code are called **expression trees**.
- Since the expression tree represents data, it can easily be converted to an alternative format such as an SQL query.
- The expression data is actually an object graph, which is the data required to compile the lambda expression into CIL.

# Unit 12 Summary

- This Unit began with a discussion of delegates and callbacks.
- Anonymous methods were then introduced, along with the related concept of lambda expressions, which allow programmers to assign a set of instructions to a variable directly.
- The Unit finished with a discussion of expression trees and how they compile into data representing a lambda expression.

# Unit 13

## Events

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 13 Overview

- The last Unit discussed delegates, the building blocks for the publish-subscribe pattern.
- This Unit focuses on **events**, which provide important encapsulations, making the publish-subscribe pattern easy to implement and less error-prone.
- Multicast delegates, which reference a chain of delegate instances will also be explored.

# Delegate Example

- See the Heater and Cooler Example, which implements a publisher-subscriber pattern.
- Subscriber methods must have the same parameter types and return type as the publisher delegate.
- The publisher stores a list of all subscribers who will receive notifications.
- Publishing a notification to the subscribers means that the publisher invokes the delegate.

# Defining the Event Subscriber

// Cooler Event Subscriber Implementation

```
class Cooler
{
    public Cooler(float temperature)
    { Temperature = temperature; }
    private float _Temperature;
    public float Temperature
    {
        get { return _Temperature; }
        set { _Temperature = value; }
    }
    public void OnTemperatureChanged(float newTemperature)
    {
        if (newTemperature > Temperature)
            System.Console.WriteLine("Cooler: On");
        else
            System.Console.WriteLine("Cooler: Off");
    }
}
```



# Defining the Event Publisher

```
// Defining the Event Publisher, Thermostat
public class Thermostat
{
    // Define the delegate data type
    public delegate void TemperatureChangeHandler(float newTemperature);

    // Define the event publisher
    public TemperatureChangeHandler OnTemperatureChange
    {
        get { return _OnTemperatureChange; }
        set { _OnTemperatureChange = value; }
    }
    private TemperatureChangeHandler _OnTemperatureChange;
    private float _CurrentTemperature;
    public float CurrentTemperature
    {
        get { return _CurrentTemperature; }
        set
        { if (value != CurrentTemperature) _CurrentTemperature = value; }
    }
}
```

# Connecting the Subscriber and Publisher

```
// Connecting the Publisher and Subscribers
static void Main(string[] args)
{
    Thermostat thermostat = new Thermostat();
    Heater heater = new Heater(60);
    Cooler cooler = new Cooler(80);
    string temperature;

    // Using C# 2.0 or later syntax.
    thermostat.OnTemperatureChange += heater.OnTemperatureChanged;
    thermostat.OnTemperatureChange += cooler.OnTemperatureChanged;

    Console.Write("Enter temperature: ");
    temperature = Console.ReadLine();
    thermostat.CurrentTemperature = int.Parse(temperature);
}
```

# Invoking a Delegate

```
// Invoking a Delegate
private TemperatureChangeHandler _OnTemperatureChange;
public float CurrentTemperature
{
    get { return _CurrentTemperature; }
    set
    {
        if (value != CurrentTemperature) ) // Notify subscribers of changes
        {
            _CurrentTemperature = value;
            TemperatureChangeHandler localOnChange =
            OnTemperatureChange;
            if (localOnChange != null      // Check for null reference
            {
                localOnChange(value); // Call subscribers
            }
        }
    }
}
```

# More on the Delegate Example

- Because multiple subscribers can be notified with a single call, delegates are known as **multicast delegates**.
- It is necessary to check for the **null** value before calling a delegate because it is possible that there are no subscribers.
- To add subscribers to a delegate, use the “+=” operator; to remove subscribers, use “-=“.
- Notifications are broadcast to all subscribers.

# Error Handling in Delegates

- If one subscriber throws an exception, later subscribers no longer receive the event notification.
- For all subscribers to receive notification in that case, manual enumeration of the list of subscribers and individual invocation is required; see the `GetInvocationList()` method.

# Delegate Return Values

- When delegates have non-void return values or have ref or out parameters, enumeration of the invocation list is also necessary to process returned values.

# Events

- **Events** are specialized delegates that provide additional protections that delegates do not.
- The event keyword provides the needed encapsulation that prevents any external class from publishing an event or unsubscribing previous subscribers they did not add.

# Declaring an Event

```
// Using the event Keyword with the Event-Coding Pattern
// Define the delegate data type
public delegate void TemperatureChangeHandler (
    object sender, TemperatureArgs newTemperature);

// Define the event publisher
public event TemperatureChangeHandler OnTemperatureChange =
    delegate { /* ... */ };
```



# Standard Event Parameters

- By convention the first parameter of an event declaration, **sender**, is of type `object` and contains a reference to the object that invoked the delegate; the second parameter is of type **System.EventArgs** (or derived from it) and contains additional data about the event.

```
// Define the delegate data type  
public delegate void TemperatureChangeHandler (  
    object sender, TemperatureArgs newTemperature);
```

# Generics and Delegates

- With generics, the same delegate type can be used in many locations with a host of different parameter types and remain strongly typed.

```
// Using Generics with Delegates
public delegate void EventHandler<T>(object sender, T e) where T : EventArgs;

// TemperatureChangeHandler no longer needed
// public delegate void TemperatureChangeHandler(
//     object sender, TemperatureArgs newTemperature);

// Define the event publisher without using TemperatureChangeHandler
public event EventHandler<TemperatureArgs> OnTemperatureChange;
```

# C# Equivalent of Event CIL Code

// C# Equivalent of the Event CIL Code Generated by the Compiler

```
public class Thermostat
{
    // Define the delegate data type
    public delegate void TemperatureChangeHandler(object sender,
        TemperatureArgs newTemperature);

    // Declaring the delegate field to save the list of subscribers.
    private TemperatureChangeHandler OnTemperatureChange;
    public void add_OnTemperatureChange(TemperatureChangeHandler handler)
        { System.Delegate.Combine(OnTemperatureChange, handler); }
    public void remove_OnTemperatureChange(TemperatureChangeHandler handler)
        { System.Delegate.Remove(OnTemperatureChange, handler); }
    public event TemperatureChangeHandler OnTemperatureChange
    {
        add
            { add_OnTemperatureChange(value) }
        remove
            { remove_OnTemperatureChange(value) }
    }
}
```

# Unit 13 Summary

- This Unit discussed events, which should always be used to implement the observer pattern because of the additional encapsulation provided and the ability to customize the implementation.

# Unit 14

## Collection Interfaces with Standard Query Operators

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 14 Overview

- This Unit first introduces anonymous types and collection initializers.
- Next, various collection-related classes and interfaces are covered, although most of the discussion concerns the generic versions.
- The `IEnumerable<T>` extension methods are also investigated.

# Anonymous Types

- Anonymous types are data types implicitly provided by the compiler rather than by explicit declaration.
- When the compiler encounters an anonymous type declaration, it generates a class definition with properties corresponding to the named values and data types.

# Example of Anonymous Types

```
// Implicit Local Variables with Anonymous Types
var patent1 = new
{
    Title = "Bifocals",
    YearOfPublication = "1784"
};
var patent2 = new
{
    Title = "Phonograph",
    YearOfPublication = "1877"
};
var patent3 = new
{
    patent1.Title,
    // Renamed to show property naming.
    Year = patent1.YearOfPublication
};
```



# Implicitly Typed Local Variables

- The local variable declared with the keyword `var` is implicitly typed.
- The compiler determines the type of the implicitly typed variable from the type assigned to it.
- Implicitly typed variables are strongly typed and receive a system-generated type name.
- Implicitly typed variables should generally be used for anonymous type declarations only.

# More about Anonymous Types

- For two anonymous types to be type-compatible, they must be a match in property names, data types, and the order of the properties.
- Anonymous types are immutable, i.e., properties are read-only.
- Methods cannot declare **var**-type parameters in their method declaration.

# Collection Initializers

- Collection initializers allow a collection to be initialized with an initial set of members at the time of instantiation.
- The syntax is similar to that used with arrays

```
List<string> sevenWorldBlunders;  
sevenWorldBlunders = new List<string>()  
{  
    // Quotes from Ghandi  
    "Wealth without work",  
    "Pleasure without conscience",  
    "Knowledge without character",  
    "Commerce without morality"  
};
```

# Initializing Anonymous Type Arrays

```
// Initializing Anonymous Type Arrays
var worldCup2006Finalists = new[]
{
    new
    {
        TeamName = "France",
        Players = new string[]
        {
            "Fabien Barthez", "Gregory Coupet",
            "Mickael Landreau", "Eric Abidal",
            // ... //
        }
    },
    // ... //
};
```

# Example of foreach

```
// foreach with Arrays  
int[] array = new int[] { 1, 2, 3, 4, 5, 6 };  
foreach (int item in array)  
{  
    Console.WriteLine(item);  
}
```

# IEnumerable<T> Interface

- A generic collection in .NET must implement the IEnumerable<T>.
- The methods of the IEnumerable<T> interface provide the minimum implementation to support iterating over a collection.
- This interface enables the **foreach** statement, which is used to iterate over an array or collection of elements.

# IEnumerable<T> Members

- `IEnumerator`, from which `IEnumerator<T>` derives, includes three members:
  - **`MoveNext()`** moves from one element to the next and detects the end of the list by returning `true` or `false`.
  - **`Current`**, a read-only property, returns the element currently in process.
  - **`Reset()`** is not called by the `foreach` loop and usually throws a `NotImplementedException`.

# Iterating a Collection Using While

```
// Iterating over a Collection Using while
System.Collections.Generic.Stack<int> stack =
    new System.Collections.Generic.Stack<int>();
int number;

// ... //

// This code is conceptual, not the actual code.
while (stack.MoveNext())
{
    number = stack.Current;
    Console.WriteLine(number);
}
```



# IEnumerable<T> Interface

- Usually classes do not support IEnumerator<T> or IEnumerator directly; instead, the IEnumerable<T> interface returns the enumerator to the caller.
- IEnumerable<T> has only one method, GetEnumerator(), which returns an object, i.e., an enumerator, that supports IEnumerator<T>.
- Usually a nested class will support the IEnumerator<T> interface and keep state.

# Using a Separate Enumerator

```
// A Separate Enumerator Maintaining State during an Iteration
System.Collections.Generic.Stack<int> stack =
    new System.Collections.Generic.Stack<int>();

int number;
System.Collections.Generic.Stack<int>.Enumerator enumerator;

// If IEnumerable<T> is implemented explicitly, then a cast is required.
// ((IEnumerable<int>)stack).GetEnumerator();
enumerator = stack.GetEnumerator();
while (enumerator.MoveNext())
{
    number = enumerator.Current;
    Console.WriteLine(number);
}
```

# Cleaning Up Following Iteration

- The `IEnumerator<T>` interface derives from `IDisposable`.
- This enables the calling of `Dispose()` after the foreach loop exists.

```
// Error Handling and Resource Cleanup with using
using (System.Collections.Generic.Stack<int>.Enumerator<int>
    enumerator = stack.GetEnumerator()) {
while (enumerator.MoveNext())
{
    number = enumerator.Current;
    Console.WriteLine(number);
}
}
```

# Restriction on Modifying Collections

- No changes to any of the items in the collection are allowed during the execution of the `foreach` loop.
- An exception of type `InvalidOperationException` is thrown if the collection is modified within the `foreach` loop.

# Standard Query Operators

- Extension methods defined in the `System.Linq.Enumerable` class make available numerous methods to classes implementing `IEnumerable<T>`.
- Each method on `IEnumerable<T>` is called a **standard query operator** and provides querying capability over a collection.

# Patent Class

```
public class Patent
{
    // Title of the published application
    public string Title { get; set; }
    // The date the application was officially published
    public string YearOfPublication { get; set; }
    // A unique number assigned to published applications
    public string ApplicationNumber { get; set; }
    public long[] InventorIds { get; set; }
    public override string ToString()
    {
        return string.Format("{0}{1}", Title, YearOfPublication);
    }
}
```

# Inventor Class

```
public class Inventor
{
    public long Id { get; set; }
    public string Name { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
    public override string ToString()
    {
        return string.Format("{0}({1}, {2})", Name, City, State);
    }
}
```

# Filtering with Where()

- The Where() method provides a filter that returns true or false, indicating whether a particular element passes the test.
- The Where() method depends on predicates for identifying filter criteria.
- A **predicate** is a delegate expression that takes an argument and returns a boolean.

```
// Filtering with System.Linq.Enumerable.Where()  
IEnumerable<Patent> patents = PatentData.Patents;  
patents = patents.Where(patent => patent.YearOfPublication.StartsWith("18"));
```



# Projecting with Select()

- The Select() operator can be used to project data into another collection.
- The new collection can contain fewer columns or be transformed entirely.

```
// Projection with System.Linq.Enumerable.Select()
IEnumerable<Patent> patents = PatentData.Patents;
IEnumerable<Patent> patentsOf1800 = patents.Where(
    patent => patent.YearOfPublication.StartsWith("18"));
IEnumerable<string> items =
    patentsOf1800.Select(patent => patent.ToString());
```

# More Select() Examples

```
//Projection with System.Linq.Enumerable.Select() and new
// ... //
IEnumerable<string> fileList = Directory.GetFiles(rootDirectory, searchPattern);
IEnumerable<FileInfo> files = fileList.Select(file => new FileInfo(file));
// ... //
// Projection to an Anonymous Type
var items = fileList.Select(
    file =>
    {
        FileInfo fileInfo = new FileInfo(file);
        return new
        {
            FileName = fileInfo.Name,
            Size = fileInfo.Length
        };
    });
```

# LINQ Queries in Parallel

- A new feature in C# 4.0, **Parallel LINQ**, makes possible the running of LINQ queries in parallel, i.e., on different CPUs on the same computer, using multiple threads.

```
IEnumerable<string> fileList = Directory.GetFiles(rootDirectory, searchPattern);  
var items = fileList.AsParallel().Select(file =>  
{  
    // Executing LINQ Queries in Parallel  
    FileInfo fileInfo = new FileInfo(file);  
    return new  
    {  
        FileName = fileInfo.Name,  
        Size = fileInfo.Length  
    };  
});
```

# Counting Elements with Count()

- The Count() query operator counts all elements in a collection or takes a predicate that only counts items identified by the predicate expression.

```
// Counting Items with Count()
IEnumerable<Patent> patents = PatentData.Patents;
Console.WriteLine("Patent Count: {0}", patents.Count());
Console.WriteLine("Patent Count in 1800s: {0}", Count(
    patent => patent.YearOfPublication.StartsWith("18")));
```

# Deferred Execution

- The **deferred execution** of standard query operators can result in subtle triggering of standard query operators.
- The query object represents the query, not the results.
- When the query is asked for the results, the whole query executes at that time.
- Assign the data queried to a local collection using a “To” method to cache the data.

# Sorting with OrderBy() and ThenBy()

- The OrderBy() and ThenBy() operators sort the elements in a collection representing primary and secondary sort fields, respectively.

```
// Ordering with System.Linq.Enumerable.OrderBy()/ThenBy()
IEnumerable<Patent> items;
Patent[] patents = PatentData.Patents;
items = patents.OrderBy(
    patent => patent.YearOfPublication).ThenBy(patent => patent.Title);
Print(items);
Console.WriteLine();
items = patents.OrderByDescending(
    patent => patent.YearOfPublication).ThenByDescending(
    patent => patent.Title);
Print(items);
```

# Facts About Sorting

- The actual sort doesn't occur until accessing the members in the collection, at which point the entire query is processed (deferred execution).
- Each subsequent call to sort the data (for example, `OrderBy()` followed by `ThenBy()`) involves additional calls to the `keySelector` expression of the earlier sort calls.

# Join Operations

- Imagine two collections of objects: Inventors (left) and Patents (right) in a Venn diagram.
- An **inner join** represents all inventors that have at least one patent.
- A **left outer join** represents all inventors, whether or not they have a patent.
- A **right outer join** would be the same as an inner join (in this case), since there are no patents without inventors.



# Important Relationships

- An example of a **one-to-many** relationship is described by the situation where a company department has many employees.
- An example of a **many-to-many** relationship might be the relationship between inventors and patents: Each patent can have multiple inventors, while each inventor can have multiple patents.

# Example of Join()

```
// An Inner Join Using System.Linq.Enumerable.Join()
Department[] departments = CorporateData.Departments;
Employee[] employees = CorporateData.Employees;
var items = employees.Join(departments,
    employee => employee.DepartmentId,
    department => department.Id,
    (employee, department) => new
    {
        employee.Id,
        employee.Name,
        employee.Title,
        Department = department
    });
foreach (var item in items)
{
    Console.WriteLine("{0} ({1})", item.Name, item.Title);
    Console.WriteLine("\t" + item.Department);
}
```

# Example of Join() Output

Mark Michaelis (Chief Computer Nerd)  
Corporate  
Michael Stokesbary (Senior Computer Wizard)  
Engineering  
Brian Jones (Enterprise Integration Guru)  
Engineering  
Jewel Floch (Bookkeeper Extraordinaire)  
Finance  
Robert Stokesbary (Expert Mainframe Engineer)  
Information Technology  
Paul R. Bramsman (Programmer Extraordinaire)  
Engineering  
Thomas Heavey (Software Architect)  
Engineering  
John Michaelis (Inventor)  
Research

# Another Inner Join with Join()

```
// Another Inner Join with System.Linq.Enumerable.Join()
Department[] departments = CorporateData.Departments;
Employee[] employees = CorporateData.Employees;
var items = departments.Join(employees,
    department => department.Id,
    employee => employee.DepartmentId,
    (department, employee) => new
    {
        department.Id,
        department.Name,
        Employee = employee
    });
foreach (var item in items)
{
    Console.WriteLine("{0}",
        item.Name);
    Console.WriteLine("\t" + item.Employee);
}
```

# Inner Join with Join() Output

Corporate

Mark Michaelis (Chief Computer Nerd)

Finance

Jewel Floch (Bookkeeper Extraordinaire)

Engineering

Michael Stokesbary (Senior Computer Wizard)

Engineering

Brian Jones (Enterprise Integration Guru)

Engineering

Paul R. Bramsman (Programmer Extraordinaire)

Engineering

Thomas Heavey (Software Architect)

Information Technology

Robert Stokesbary (Expert Mainframe Engineer)

Research

John Michaelis (Inventor)

# Grouping Results with GroupBy()

- Frequently, objects with like characteristics are grouped together.
- The items that result from a GroupBy() call are of the type `IGrouping<TKey, TElement>`

# Grouping Results Output

Mark Michaelis (Chief Computer Nerd)

Count: 1

Michael Stokesbary (Senior Computer Wizard)

Brian Jones (Enterprise Integration Guru)

Paul R. Bramsman (Programmer Extraordinaire)

Thomas Heavey (Software Architect)

Count: 4

Jewel Floch (Bookkeeper Extraordinaire)

Count: 1

Robert Stokesbary (Expert Mainframe Engineer)

Count: 1

John Michaelis (Inventor)

Count: 1

# Grouping Items Using GroupBy()

```
// Grouping Items Together Using System.Linq.Enumerable.GroupBy()
IEnumerable<Employee> employees = CorporateData.Employees;
IEnumerable<IGrouping<int, Employee>> groupedEmployees =
    employees.GroupBy((employee) => employee.DepartmentId);

foreach (IGrouping<int, Employee> employeeGroup in
                                                groupedEmployees)
{
    Console.WriteLine();
    foreach (Employee employee in employeeGroup)
    {
        Console.WriteLine("\t" + employee);
    }
    Console.WriteLine("\tCount: " + employeeGroup.Count());
}
```



# One-to-Many Relationship with GroupJoin()

- The GroupJoin() method allows for the creation of a one-to-many relationship between two objects, e.g., Departments and their associated Employees.
- Note that unlike Join(), GroupJoin() does not have an SQL equivalent, since SQL is record-based, not hierarchical.

# GroupJoin() Child Collection

```
// Creating a Child Collection with GroupJoin()
var items = departments.GroupJoin(
    employees,
    department => department.Id,
    employee => employee.DepartmentId,
    (department, departmentEmployees) => new
    {
        department.Id,
        department.Name,
        Employees = departmentEmployees
    });
foreach (var item in items)
{
    Console.WriteLine("{0}", item.Name);
    foreach (Employee employee in item.Employees)
    { Console.WriteLine("\t" + employee); }
}
```

# GroupJoin() Child Collection Output

## Corporate

Mark Michaelis (Chief Computer Nerd)

## Finance

Jewel Floch (Bookkeeper Extraordinaire)

## Engineering

Michael Stokesbary (Senior Computer Wizard)

Brian Jones (Enterprise Integration Guru)

Paul R. Bramsman (Programmer Extraordinaire)

Thomas Heavey (Software Architect)

## Information Technology

Robert Stokesbary (Expert Mainframe Engineer)

## Research

John Michaelis (Inventor)

# Calling SelectMany()

- The SelectMany() method handles collections of collections.
- This method combines each array (or collection) selected and produces a single collection of all items.

# Example of Calling SelectMany()

```
// Calling SelectMany()
var worldCup2006Finalists = new[]
{
    new
    {
        TeamName = "France",
        Players = new string[]
        { "Fabien Barthez", "Gregory Coupet", "Mickael Landreau" }
    },
    new
    {
        TeamName = "Italy",
        Players = new string[]
        { "Gianluigi Buffon", "Angelo Peruzzi", "Marco Amelia" }
    }
};
IEnumerable<string> players =
    worldCup2006Finalists.SelectMany(team => team.Players);
```

# Standard Query Operators

Comment Type	Description
OfType<T>()	Forms a query over a collection that returns only the items of a particular type, where the type is identified in the type parameter of the OfType<T>() method call.
Union()	Combines two collections to form a superset of all the items in both collections. The final collection does not include duplicate items even if the same item existed in both collections to start.
Concat()	Combines two collections together to form a superset of both collections. Duplicate items are not removed from the resultant collection. Concat() will preserve the ordering. That is, concatenating {A, B} with {C, D} will produce {A, B, C, D}.
Intersect()	Extracts the collection of items that exist in both original collections.
Distinct()	Filters out duplicate items from a collection so that each item within the resultant collection is unique.

# Standard Query Operators

Comment Type	Description
SequenceEquals()	Compares two collections and returns a Boolean indicating whether the collections are identical, including the order of items within the collection. (This is a very helpful message when testing expected results.)
Reverse()	Reverses the items within a collection so that they occur in reverse order when iterating over the collection.

# Aggregate Functions

Comment Type	Description
Count()	Provides a total count of the number of items within the collection
Average()	Calculates the average value for a numeric key selector
Sum()	Computes the sum values within a numeric collection
Max()	Determines the maximum value among a collection of numeric values
Min()	Determines the minimum value among a collection of numeric values



# Unit 14 Summary

- This Unit discussed anonymous types, implicit variables, and collection initializers.
- The Unit also discussed using the standard query operators from LINQ.
- In the next Unit, LINQ with query expressions will be explored.

# Unit 15

## LINQ with Query Expressions

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 15 Overview

- In Unit 14, standard query operators and the queries they enable, which have functionality similar to SQL, were discussed.
- This Unit discusses using the syntax of query expressions to express many of the queries from the preceding Unit into more readable syntax.

# Simple Query Expression

- **Query expressions** have a syntax that is similar to SQL syntax.
- Query expressions begin with a **from** clause and end with a **select** clause or a **groupby** clause:

```
IEnumerable<string> selection = from word in Keywords  
                                where !word.Contains('*') select word;
```

- The reason query expressions begin with a **from** clause is to enable Intellisense to work.

# Simple Query Expression Example

```
// Simple Query Expression
static string[] Keywords = {
    "abstract", "add*", "alias*", "as", "ascending*", "base",
    "bool", "break", "by*", "byte", "case", "catch", "char",
    /* ... */
    "unsafe", "ushort", "using", "value*", "var*", "virtual",
    "void", "volatile", "where*", "while", "yield*"};

private static void ShowContextualKeywords1()
{
    IEnumerable<string> selection = from word in Keywords
                                   where !word.Contains('*')
                                   select word;

    foreach (string keyword in selection)
    { Console.WriteLine(" " + keyword); }
}
```

# Projection

- The type returned from a query expression is not limited to be a collection of the original element type.
- The `select` clause allows for the **projection** of data into an entirely different type, an **anonymous type**.
- Anonymous types enable support for types to be defined by the compiler—types that contain only the data needed at that time.

# Projection Example

```
// Projection Using Query Expressions
static void List1(string rootDirectory, string searchPattern)
{
    IEnumerable<FileInfo> files =
        from fileName in Directory.GetFiles(rootDirectory, searchPattern)
        select new FileInfo(fileName);

    foreach (FileInfo file in files)
    {
        Console.WriteLine("{0} ({1})",file.Name, file.LastWriteTime);
    }
}
```

# Projection Example Using Anonymous Types

```
// Anonymous Types within Query Expressions
static void List2(string rootDirectory, string searchPattern)
{
    var files =
        from fileName in Directory.GetFiles(rootDirectory, searchPattern)
        select new
        {
            Name = fileName,
            LastWriteTime = File.GetLastWriteTime(fileName)
        };

    foreach (var file in files)
    {
        Console.WriteLine("{0}{1}", file.Name, file.LastWriteTime);
    }
}
```



# Deferred Execution with Query Expressions

- Deferred execution applies to query expressions also, i.e., the assignment itself does not execute the query expression.
- The query expression is saved off until iteration (**foreach**) of the collection takes place.
- In the following example, the `IsKeyword()` function isn't called until the code iterates over the selection rather than when the assignment is made.

# Deferred Example 1

```
// Deferred Execution and Query Expressions (Example 1)
private static void ShowContextualKeywords2()
{
    IEnumerable<string> selection = from word in Keywords
                                   where IsKeyword(word)
                                   select word;
    foreach (string keyword in selection)
        { Console.WriteLine(keyword); }
}
// Side effect (console output) included in predicate to show
// deferred execution not as a best practice.
private static bool IsKeyword(string word)
{
    if (word.Contains('*'))
    {
        Console.WriteLine(" ");
        return true;
    }
    else { return false; }
}
```

# Deferred Example 2 Part 1

```
// Deferred Execution and Query Expressions (Example 2)
private static void CountContextualKeywords()
{
    int delegateInvocations = 0;
    Func<string, string> func = text =>
        {
            delegateInvocations++;
            return text;
        };
    IEnumerable<string> selection = from keyword in Keywords
        where keyword.Contains('*')
        select func(keyword);

    // Continued on next page //
```

# Deferred Example 2 Part 2

```
// Deferred Execution and Query Expressions (Example 2)
private static void CountContextualKeywords()
{
    // Continued from previous page //
    Console.WriteLine("1. delegateInvocations={0}", delegateInvocations);
    // Executing count should invoke func once for each item selected.
    Console.WriteLine("2. Contextual keyword count={0}",
        selection.Count());
    Console.WriteLine("3. delegateInvocations={0}", delegateInvocations);
    // Executing count should invoke func once for each item selected.
    Console.WriteLine("4. Contextual keyword count={0}",
        selection.Count());
    Console.WriteLine("5. delegateInvocations={0}", delegateInvocations);
    // Cache the value so future counts will not trigger another invocation of the query.
    List<string> selectionCache = selection.ToList();
    Console.WriteLine("6. delegateInvocations={0}", delegateInvocations);
    // Retrieve the count from the cached collection.
    Console.WriteLine("7. selectionCache count={0}", selectionCache.Count());
    Console.WriteLine("8. delegateInvocations={0}", delegateInvocations);
}
```

# Deferred Example 2 Output

1. delegateInvocations=0
2. Contextual keyword count=15
3. delegateInvocations=15
4. Contextual keyword count=15
5. delegateInvocations=30
6. delegateInvocations=45
7. selectionCache count=15
8. delegateInvocations=45

# Filtering

- The **where** clause filters the collection, so that there are fewer items within the collection.
- The filter criteria are expressed with a predicate, i.e., a lambda expression that returns a boolean value.

# Filtering Example

```
// Anonymous Types within Query Expressions
static void FindMonthOldFiles(string rootDirectory, string searchPattern)
{
    IEnumerable<FileInfo> files =
        from fileName in Directory.GetFiles(rootDirectory, searchPattern)
        where File.GetLastWriteTime(fileName) <
            DateTime.Now.AddMonths(-1)
        select new FileInfo(fileName);
    foreach (FileInfo file in files)
    {
        // As simplification, current directory is assumed to be
        // a subdirectory of rootDirectory
        string relativePath = file.FullName.Substring(
            Environment.CurrentDirectory.Length);
        Console.WriteLine(".{0}{{1}}", relativePath, file.LastWriteTime);
    }
}
```

# Sorting

- Use the **orderby** clause to order the items in a query expression.
- **ascending** (default) and **descending** are optional keywords indicating the sort order.



# Sorting Example

```
// Sorting Using a Query Expression with an orderby Clause
static void ListByFileSize1(string rootDirectory, string searchPattern)
{
    IEnumerable<string> fileNames =
        from fileName in Directory.GetFiles(rootDirectory, searchPattern)
        orderby (new FileInfo(fileName)).Length descending, fileName
        select fileName;

    foreach (string fileName in fileNames)
    {
        Console.WriteLine("{0}", fileName);
    }
}
```

# The Let Clause

- To avoid unnecessary overhead such as having to instantiate an object twice for each item, the **let** expression can be included.
- The **let** clause provides a location to place an expression that is used in the query.
- Additional **let** expressions can be added after the first **from** clause.

# The Let Clause Example

// Ordering the Results in a Query Expression

```
IEnumerable<FileInfo> files =  
    from fileName in Directory.GetFiles(rootDirectory, searchPattern)  
    orderby new FileInfo(fileName).Length, fileName  
    select new FileInfo(fileName);
```

// A better way using the let expression

```
IEnumerable<FileInfo> files =  
    from fileName in Directory.GetFiles(rootDirectory, searchPattern)  
    let file = new FileInfo(fileName)  
    orderby file.Length, fileName  
    select file;
```

# Grouping

- **Grouping** refers to aggregating the items into a summary header or total—an aggregate value.
- Query expressions also allow for the individual aggregate items to form a series of sub-collections to each item in the overall parent list.

# Grouping Example

```
// Grouping Together Query Results
private static void GroupKeywords1()
{
    IEnumerable<IGrouping<bool, string>> selection =
        from word in Keywords
        group word by word.Contains('*');
    foreach (IGrouping<bool, string> wordGroup in selection)
    {
        Console.WriteLine(Environment.NewLine + "{0}:",
            wordGroup.Key ? "Contextual Keywords" : "Keywords");
        foreach (string keyword in wordGroup)
        {
            Console.Write(" " + (wordGroup.Key ?
                keyword.Replace("*", null) : keyword));
        }
    }
}
```

# Another Grouping Example

// Selecting an Anonymous Type Following the groupby Clause

```
private static void GroupKeywords1()
{
    IEnumerable<IGrouping<bool, string>> keywordGroups =
        from word in Keywords group word by word.Contains('*');
    var selection = from groups in keywordGroups
        select new
        {
            IsContextualKeyword = groups.Key,
            Items = groups };
    foreach (var wordGroup in selection)
    {
        Console.WriteLine(Environment.NewLine + "{0}:",
            wordGroup.IsContextualKeyword ?
                "Contextual Keywords" : "Keywords");
        { Console.Write(" " + keyword.Replace("*", null)); }
    }
}
```

# Query Continuation with into

- The **into** clause serves as a generator for additional query commands, specifically a select clause.
- **Query continuation** shortcuts the need to write a second query using the results of the first query.
- The ability to run additional queries on the results of an existing query using **into** is a feature of all query expressions.

# Query Continuation Example

```
// Selecting without the Query Continuation
private static void GroupKeywords1()
{
    var selection =
        from word in Keywords
        group word by word.Contains('*')
        into groups
        select new
        {
            IsContextualKeyword = groups.Key,
            Items = groups
        };
    // ... //
}
```



# Distinct Members

- The query operator Distinct() returns only distinct items from within a collection.

```
// Obtaining Distinct Members from a Query Expression
public static void ListMemberNames()
{
    IEnumerable<string> enumerableMethodNames = (
        from method in typeof(Enumerable).GetMembers(
            System.Reflection.BindingFlags.Static |
            System.Reflection.BindingFlags.Public)
        select method.Name).Distinct();
    foreach (string method in enumerableMethodNames)
    {
        Console.WriteLine("{0},", method);
    }
}
```

# Query Expressions As Method Invocations

- The C# compiler translates the query expression into method calls.

## // Simple Query Expression

```
private static void ShowContextualKeywords1()
{
    IEnumerable<string> selection =
        from word in Keywords
        where word.Contains('*') select word;
}
```

## // Query Expression Translated to Standard Query Operator Syntax

```
private static void ShowContextualKeywords3()
{
    IEnumerable<string> selection =
        Keywords.Where(word => word.Contains('*'));
}
```

# Unit 15 Summary

- This Unit introduced a new syntax, that of query expressions, which frequently provide a simpler syntax for expressing queries.
- There are many similarities between query expressions and SQL syntax.
- The end result is a significant improvement in the approach to coding queries.
- In the next Unit, the discussion of collections is continued.

# Unit 16

## Building Custom Collections

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 16 Overview

- Unit 16 provides an introduction to the different types of collections and interfaces.
- In addition, the Unit discusses how to define a custom collection that supports standard collection functionality, such as indexing and foreach iteration.
- Collections can be divided into those that support generics and those that don't.

# ICollection<T> vs. IDictionary<TKey,TValue>

- ICollection<T> and IDictionary<TKey,TValue> are interfaces implemented by collection classes.
- The ICollection<T> interface provides support for element retrieval from a collection by numeric index.
- The IDictionary<TKey,TValue> interface provides support for element retrieval from a collection by key field.

# IComparable<T>

- The IComparable<T> interface is required for sorting the elements in a collection.
- IComparable<T> has only one method, CompareTo(), which returns an integer indicating whether the element passed is greater than, less than, or equal to the current element.
- The elements in a collection must implement IComparable<T> for sorting to operate.

# IComparer<T>

- Another way to implement sorting is to implement the IComparer<T> interface, often used to create an alternative sort class.
- IComparer<T> is not implemented by the element itself but by a separate class.
- IComparer<T> provides a single sort method Compare() (not CompareTo()).
- An instance of the sort class is passed as an argument to the Sort method of a collection.



# Implementing IComparer<T>

```
// Implementing IComparer<T>
class NameComparison : IComparer<Contact>
{
    private static int StringCompare(string x, string y)
    {
        int result;
        if (x == null)
        {
            if (y == null)
            { result = 0; }
            else
            { result = ; }
        }
        else
        { result = x.CompareTo(y); }
        return result;
    }
}
```

# ICollection<T>

- Both IList<T> and IDictionary<TKey,TValue> implement ICollection<T>.
- ICollection<T> defines two members: Count and CopyTo().
- The Count property returns the total number of elements in the collection.
- The CopyTo() method provides the ability to convert the collection into an array.

# List Collections: List<T>

- The List<T> class has properties similar to those of an array, but, unlike an array, can expand or contract.
- Elements in the list can be accessed by numeric index, with the index parameter indicating the position in the list.
- List indexes are zero-based.
- The Add() method will add to the end of a list.

# List<T> Methods

- The Sort() method sorts the underlying elements.
- The Remove() method removes an element.
- To search the list, the Contains(), IndexOf(), LastIndexOf(), and BinarySearch() methods are provided.

# Using the List<T>

```
// Using List<T>
static void Main(string[] args)
{
    List<string> list = new List<string>();

    // Lists automatically expand as elements are added.
    list.Add("Sneezy");
    list.Add("Happy");
    list.Add("Dopey");
    list.Add("Doc");
    list.Add("Sleepy");
    list.Add("Bashful");
    list.Add("Grumpy");
    list.Sort();
    Console.WriteLine("In alphabetical order {0} is the "
        + "first dwarf while {1} is the last.", list[0], list[6]);
    list.Remove("Grumpy");
}
```

# The BinarySearch() Method

- The BinarySearch() method uses a binary search algorithm that requires the elements be sorted.
- If an element is found, a positive integer indicating its list position is returned.
- If the element is not found, a negative integer containing the bitwise complement ( $\sim$ ) of the index of the next larger element is returned, or, if no greater value, the total element count.

# BinarySearch() Example

```
// Using the Bit Complement of the BinarySearch() Result
static void Main()
{
    List<string> list = new List<string>();
    int search;
    list.Add("public");
    list.Add("protected");
    list.Add("private");
    list.Sort();
    search = list.BinarySearch("protected internal");
    if (search < 0)
    {
        list.Insert(~search, "protected internal");
    }
    foreach (string accessModifier in list)
    { Console.WriteLine(accessModifier); }
}
```

# The FindAll() Method

- The FindAll() methods finds multiple items in a list and allows search criteria to be more complex than just looking for specific values.
- FindAll() returns a new List<T> instance of all found items.
- FindAll() takes a parameter of type Predicate<T>, a delegate reference returning true to indicate that an item is to be added to a new List<T> instance.



# FindAll() Example

// Demonstrating FindAll() and Its Predicate Parameter

```
static void Main()
{
    List<int> list = new List<int>();
    list.Add(1);
    list.Add(2);
    list.Add(3);
    list.Add(2);

    List<int> results = list.FindAll(Even);
    foreach (int number in results)
    { Console.WriteLine(number); }
}

public static bool Even(int value)
{ return (value % 2) == 0; }
```

# Dictionary Collections:

## Dictionary<TKey, TValue>

- Dictionary classes store name/value pairs.
- The name functions as a unique key used to look up the corresponding element (value).
- The Add() method or the index operator ([]) can be used to add items to a dictionary.
- The Remove() method removes an element.
- The Keys or Values properties return an ICollection<T> of keys or values, respectively.

# Adding and Inserting Items into Dictionary

```
static void Main()
{
    // Adding Items to a Dictionary<TKey, TValue>
    Dictionary<Guid, string> dictionary = new Dictionary<Guid, string>();
    Guid key = Guid.NewGuid();
    dictionary.Add(key, "object");

    // Inserting Items in a Dictionary<TKey, TValue>
    // Using the Index Operator
    Guid key = Guid.NewGuid();
    dictionary[key] = "object";
    dictionary[key] = "byte";
}
```

# Iterating Over a Dictionary

```
// Iterating over Dictionary<TKey, TValue> with foreach
int index = 0;
Dictionary<string, string> dictionary = new Dictionary<string, string>();

dictionary.Add(index++.ToString(), "object");
dictionary.Add(index++.ToString(), "byte");
/* ... */
dictionary.Add(index++.ToString(), "double");
dictionary.Add(index++.ToString(), "string");

Console.WriteLine("Key Value Hashcode");
Console.WriteLine("----");
foreach (KeyValuePair<string, string> i in dictionary)
{
    Console.WriteLine("{0,-5}{1,-9}{2}",
                      i.Key, i.Value, i.Key.GetHashCode());
}
```

# Iterating Over a Dictionary

- Iterating through a dictionary using the `foreach` loop accesses values in no particular order.
- Elements are arranged into a hash table using the hash codes generated by calling `GetHashCode()` on the key field.
- The data type returned from each iteration of the `foreach` loop is of the type `KeyValuePair.`

# More on Dictionaries

- Accessing a value from a dictionary using the index operator ([]) with a nonexistent key throws a `KeyNotFoundException`.
- The `ContainsKey()` method allows checking to determine whether a particular key is used before accessing its value.
- Keys are stored in a hash table, but values are not; therefore, the `ContainsValue()` method must search sequentially.

# Sorted Collections

- The `SortedDictionary<TKey, TValue>` represents a collection of key/value pairs sorted by keys.
- The `SortedList<T>` represents a collection of key/value pairs sorted by keys and accessible by key and by index.
- A **foreach** enumeration of a sorted collection returns the elements in sorted order.
- Maintenance of the order slightly increases execution time.

# SortedDictionary Example

```
// Using SortedDictionary<TKey, TValue>
static void Main(string[] args)
{
    SortedDictionary<string, string> sortedDictionary =
        new SortedDictionary<string, string>();
    int index = 0;
    sortedDictionary.Add(index++.ToString(), "object");
    sortedDictionary.Add(index++.ToString(), "string");

    Console.WriteLine("Key Value Hashcode");
    Console.WriteLine("----");
    foreach (KeyValuePair<string, string> i in sortedDictionary)
    { Console.WriteLine("{0,-5}{1,-9}{2}",
        i.Key, i.Value, i.Key.GetHashCode()); }
}
```



# Stack Collections: Stack<T>

- Stacks are last in, first out (LIFO) collections.
- The two key methods are
  - Push(): pushes elements on top of the stack.
  - Pop(): retrieves and removes elements from the top of the stack.
- The Peek() and Contains() methods access the elements on the stack without modifying it.
- A **foreach** loop can be used to enumerate the stack, as with all collection types.

# Queue Collections: Queue<T>

- Queues are first in, first out (FIFO) collections.
- The queue uses the Enqueue() and Dequeue() methods to add and remove elements.
- The queue acts like a pipeline: objects are entered into the queue at one end and removed at the other end.
- Queues are sometimes referred to as waiting lines.

# Linked Lists: LinkedList<T>

- A linked list collection enables both forward and reverse traversal.
- The LinkedList<T> has no non-generic version.
- A LinkedList<T> type is populated with LinkedListNode<T> types, which have Next, Previous, and Value properties.
- The LinkList<T> type has Count, Head, and Tail properties, and also various Add and Remove methods.

# Providing an Index Operator

- The index operator is represented by a pair of square brackets used to index into a dictionary or collection.
- Programmers can add an index operator to a custom collection class, but most have one.
- To define an index operator, name the member **this** and follow it with square brackets.

```
public T this[PairItem index] { .. }
```

# Iterators

- An **iterator** provides shortcut implementation of the `IEnumerable<T>` and `IEnumerator<T>` interfaces , which enable the `foreach` loop.
- Implementing the iterator pattern requires tracking the current position in the collection; iterators have a built-in means of tracking the current and next elements.
- Iterators are like functions, but instead of returning values, they *yield* them.

# More on Iterators

- The `yield return` statement returns values each time it is encountered.
- When the next iteration starts, the program executes the statement immediately following the last `yield return` statement, i.e., where it previously exited.
- Thus the iterator maintains its state in between each iteration of the `foreach` loop.

# yield return Example

```
// Yielding the C# Keywords Sequentially
public class CSharpPrimitiveTypes : IEnumerable<string>
{
    public IEnumerator<string> GetEnumerator()
    { // Note the hard-coded yield return statements
      yield return "object";
      yield return "byte";
      yield return "uint";
      /* ... */
    }
    // IEnumerator also required because IEnumerator<T> derives from it.
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
    { // Invoke IEnumerator<string> GetEnumerator() above
      return GetEnumerator();
    }
}
```

# yield return Example (cont'd)

```
// Yielding the C# Keywords Sequentially
class Program
{
    static void Main(string[] args)
    {
        CSharpPrimitiveTypes primitives = new CSharpPrimitiveTypes();
        foreach (string primitive in primitives)
        {
            Console.WriteLine(primitive);
        }
    }
}
```



# yield return Iteration Example

```
public class BinaryTree<T> : IEnumerable<T>
{
    // Placing yield return Statements within a Loop
    public IEnumerator<T> GetEnumerator()
    {
        yield return Value; // Return the item at this node.
        // Iterate through each of the elements in the pair.
        foreach (BinaryTree<T> tree in SubItems)
        {
            if (tree != null)
            {
                // Since each element in pair is a tree, traverse the tree, yield each element
                foreach (T item in tree)
                { yield return item; }
            }
        }
    }
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator()
    { return GetEnumerator(); }
}
```

# Iteration Example (cont'd)

```
// JFK
jfkFamilyTree = new BinaryTree<string>("John Fitzgerald Kennedy");

jfkFamilyTree.SubItems = new Pair<BinaryTree<string>>>(
    new BinaryTree<string>("Joseph Patrick Kennedy"),
    new BinaryTree<string>("Rose Elizabeth Fitzgerald"));

// Grandparents (Father's side)
jfkFamilyTree.SubItems.First.SubItems =
    new Pair<BinaryTree<string>>>(
        new BinaryTree<string>("Patrick Joseph Kennedy"),
        new BinaryTree<string>("Mary Augusta Hickey"));

// Grandparents (Mother's side)
jfkFamilyTree.SubItems.Second.SubItems =
    new Pair<BinaryTree<string>>>(
        new BinaryTree<string>("John Francis Fitzgerald"),
        new BinaryTree<string>("Mary Josephine Hannon"));

foreach (string name in jfkFamilyTree){ Console.WriteLine(name); }
```

# yield break

- The **yield break** statement cancels iteration and returns control to the caller.

```
// Escaping Iteration via yield break
public System.Collections.Generic.IEnumerable<T> GetNotNullEnumerator()
{
    if((First == null) || (Second == null))
    {
        yield break;
    }
    yield return Second;
    yield return First;
}
```

# Multiple Iterators in a Single Class

- Sometimes specialized iterators are needed, for example, iterating in reverse or filtering the results.
- Additional iterators in the class can be created by encapsulating them within properties or methods that return `IEnumerable<T>` or `IEnumerable`.

# Multiple Iterators Example

```
// Using yield return in a Method That Returns IEnumerable<T>
```

```
public struct Pair<T>: IEnumerable<T>
{
    public IEnumerable<T> GetReverseEnumerator()
    {
        yield return Second;
        yield return First;
    }
    /* ... */
}
```

```
// Using an alternate iterator
```

```
foreach (string name in game.GetReverseEnumerator())
{
    Console.WriteLine(name);
}
```

# yield Statement Characteristics

- The **yield return** statement can be used only in GetEnumerator() methods that return IEnumerable<T> or in methods that return IEnumerable<T> that not called GetEnumerator().
- All code paths must contain a **yield return** statement if they are to return any data.
- The yield statement may not appear outside a method, operator, or property accessor.

# Unit 16 Summary

- This Unit discussed the generic collection classes and interfaces, which have now taken the place of their non-generic counterparts.
- The `yield` keyword is used to generate code implementing the iterator pattern and is used by the `foreach` loop.

# Unit 17

## Reflection, Attributes, and Dynamic Programming

C# Data Structures and Design Patterns  
Spring Quarter, 2017



# Unit 17 Overview

- This Unit investigates the details of attributes built into the .NET Framework, as well as how to define custom attributes.
- This Unit also begins to examine reflection, including dynamic binding and the invoking of members at compile time.
- Finally, the Unit concludes with a discussion of dynamic programming, new to C# 4.0.

# Reflection

- Using reflection, it is possible to do the following:
  - Access the metadata for types within an assembly.
  - Dynamically invoke a type's member at runtime.
- Reflection is the process of examining the metadata within an assembly at runtime.
- A type's metadata, which includes the full type name, member names, and any attributes is accessed through an instance of `System.Type`.

# Accessing Metadata Using System.Type

- `System.Type` provides all of the methods needed for retrieving type information.
- The key to reading a type's metadata is to obtain an instance of `System.Type` class representing the target type.
- The two primary means for obtaining a type reference are through the `GetType()` method and the `typeof()` operator.

# GetType()

- The GetType() method is inherited from System.Object; all types include this function.
- Calling GetType() retrieves an instance of System.Type for the original object.

```
// Using Type.GetProperties() to Obtain an Object's Public Properties
DateTime dateTime = new DateTime();
Type type = dateTime.GetType();
foreach (System.Reflection.PropertyInfo property
           in type.GetProperties())
{
    Console.WriteLine(property.Name);
}
```

# typeof()

- The typeof() expression also can be used to retrieve a Type object.

```
// Using typeof() to Create a System.Type Instance  
ThreadPriorityLevel priority;  
priority = (ThreadPriorityLevel)Enum.Parse(typeof(ThreadPriorityLevel), "Idle");
```

# Member Invocation

- Members can be dynamically invoked once the metadata is retrieved.
- See the example in Listing 17.3, Dynamically Invoking a Member.

```
// Dynamically Invoking a Member
PropertyInfo property = commandLine.GetType().GetProperty(option,
    BindingFlags.IgnoreCase | BindingFlags.Instance | BindingFlags.Public);
if (property.PropertyType == typeof(bool))
{
    // Last parameters for handling indexers
    property.SetValue(commandLine, true, null);
}
```

# Reflection with Generics

- Reflection can determine whether a given type supports generic parameters.
- A generic argument is a type parameter specified when a generic class is instantiated.
- The `typeof` operator can also be used on generic types and methods.
- `IsGenericType` evaluates if a type is generic.
- The `GetGenericArguments` obtains a list of generic arguments or type parameters.

# Reflection with Generics

```
static void Main(string[] args)
{
    // Reflection with Generics
    Type type;
    type = typeof(System.Nullable<>);
    Console.WriteLine(type.ContainsGenericParameters);
    Console.WriteLine(type.IsGenericType);

    type = typeof(System.Nullable<DateTime>);
    Console.WriteLine(!type.ContainsGenericParameters);
    Console.WriteLine(type.IsGenericType);
}
```



# Using Reflection with Generics

```
static void Main(string[] args)
{
    // Using Reflection with Generic Types
    Stack<int> s = new Stack<int>();
    Type t = s.GetType();
    foreach (Type types in t.GetGenericArguments())
    {
        System.Console.WriteLine("Type parameter: " + types.FullName);
    }
}
```

# Attributes

- Attributes are a means of associating additional data with classes, interfaces, structs, enums, delegates, events, methods, constructors, fields, parameters, return values, assemblies, type parameters, and modules.
- Attributes appear within square brackets preceding the construct they decorate.
- Multiple attributes can be used on some constructs.

# Using Attributes

// Decorating a Property with an Attribute

```
class CommandLineInfo
{
    [CommandLineSwitchAlias("?")]
    public bool Help
    {
        get { return _Help; }
        set { _Help = value; }
    }
    private bool _Help;
    [CommandLineSwitchRequired]
    public string Out
    {
        get { return _Out; }
        set { _Out = value; }
    }
    private string _Out;
}
```

# Using Multiple Attributes

// Decorating a Property with Multiple Attributes

```
[CommandLineSwitchRequired]  
[CommandLineSwitchAlias("FileName")]  
public string Out  
{  
    get { return _Out; }  
    set { _Out = value; }  
}
```

// Decorating a Property with Multiple Attributes (alternative)

```
[CommandLineSwitchRequired, CommandLineSwitchAlias("FileName")]  
public string Out  
{  
    get { return _Out; }  
    set { _Out = value; }  
}
```

# Assembly Attributes

- Assembly attributes define data such as company, product, and assembly version.
- Assembly attributes must appear after the using directive but before the namespace of class declarations.
- Other attributes requiring explicit target identifications include return:, module:, class, and method:.

# Assembly Attributes in AssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("ConsoleApplication1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Microsoft")]
[assembly: AssemblyProduct("ConsoleApplication1")]
[assembly: AssemblyCopyright("Copyright © Microsoft 2010")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

/* ... */
```

# Custom Attributes

- Attributes are defined by classes.
- An attribute class must derive from `System.Attribute`.

```
// Defining a Custom Attribute  
public class CommandLineSwitchRequiredAttribute : Attribute  
{ /* ... */ }
```

# Searching for Attributes

- All reflection types and the Type class itself include members for retrieving a list of attributes.
- The GetCustomAttributes() method will return an array of custom attributes for a member.

```
// Retrieving a Custom Attribute  
Attribute[] attributes = (Attribute[])property.GetCustomAttributes(  
    typeof(CommandLineSwitchRequiredAttribute), false);
```



# Initializing an Attribute

- Attributes can have constructors, both default and parameterized.
- Only literal values and types (such as `typeof(int)`) are allowed as arguments to a constructor.

# Providing an Attribute Constructor

```
// Providing an Attribute Constructor
public class CommandLineSwitchAliasAttribute : Attribute
{
    public CommandLineSwitchAliasAttribute(string alias)
    {
        Alias = alias;
    }
    public string Alias
    {
        get { return _Alias; }
        set { _Alias = value; }
    }
    private string _Alias;
}
```

# Using an Attribute Constructor

```
// Using an Attribute Constructor
class CommandLineInfo
{
    [CommandLineSwitchAlias("?")]
    public bool Help
    {
        get { return _Help; }
        set { _Help = value; }
    }
    private bool _Help;
    // ...
}
```

# System.AttributeUsageAttribute

- Custom attributes can be decorated with the AttributeUsageAttribute to restrict usage.
- AttributeUsageAttribute's constructor takes an AttributesTargets enumerated type containing a list of possible targets.

```
// Restricting the Constructs an Attribute Can Decorate
[AttributeUsage(AttributeTargets.Property)]
public class CommandLineSwitchAliasAttribute : Attribute
{
    /* ... */
}
```

# Named Parameters

- **Named parameters** provide a mechanism for setting specific public properties and fields within the attribute constructor call, even though the constructor includes no corresponding parameters.
- The named parameters are optional.
- See the AllowMultiple public member below.

```
// Using a Named Parameter  
[AttributeUsage(AttributeTargets.Property, AllowMultiple=true )]  
public class CommandLineSwitchAliasAttribute : Attribute { }
```

# FlagsAttribute

- The FlagsAttribute targets enumerated types (enum) that represent flag values.

```
// FileAttributes defined in System.IO.  
// Decorating an enum with FlagsAttribute  
[Flags].  
public enum FileAttributes  
{  
    ReadOnly = 1<<0, // 0000000000000001  
    Hidden = 1<<1,   // 0000000000000010  
    /* ... */  
}
```

# File Attributes in Action

```
// Using FlagsAttribute
public static void Main()
{
    string fileName = @"enumtest.txt";
    FileInfo file = new FileInfo(fileName);
    file.Attributes = FileAttributes.Hidden | FileAttributes.ReadOnly;
    Console.WriteLine("\"{0}\" outputs as \"{1}\"",
        file.Attributes.ToString().Replace(",", " |"), file.Attributes);
    FileAttributes attributes =(FileAttributes) Enum.Parse(
        typeof(FileAttributes), file.Attributes.ToString());
    Console.WriteLine(attributes);
}
```

# Displaying Enumeration Values

- The ToString() method called on a variable whose type is a flags enumeration displays the strings for each enumeration flag that is set.
- Parsing a value from a string to the enumerated value works if each enumeration value in the string is separated by commas.
- See the previous example.



# Predefined Attributes

- Some attributes are predefined in the .NET classes libraries such as `AttributeUsageAttribute`, `FlagsAttribute`, `ObsoleteAttribute`, and `ConditionalAttribute`.
- These attributes are understood by the CLR and compiler.

# System.ConditionalAttribute

- ConditionalAttribute acts behaves similarly to placing a #if/#endif preprocessor directive around the method invocation.
- Instead of eliminating code from the assembly, this attribute causes a method call to behave like a no-op instruction.
- The C# compiler notes the attribute on the called method, and if the preprocessor directive exists, eliminates calls to the method.

# ConditionalAttribute Example

```
#define CONDITION_A
using System;
using System.Diagnostics;
/* ... */
static void Main(string[] args)
{
    Console.WriteLine("Begin...");
    MethodA();
    MethodB(); // Do not call this method
    Console.WriteLine("End...");
}
[Conditional("CONDITION_A")]
static void MethodA()
{ Console.WriteLine("MethodA() executing..."); }
[Conditional("CONDITION_B")]
static void MethodB()
{ Console.WriteLine("MethodB() executing..."); }
```

# Rules for ConditionalAttributes

- The ConditionalAttribute may not be used on methods that include an **out** parameter or that have a non-**void** return value.
- Properties cannot be decorated with the ConditionalAttribute, only classes deriving from System.Attribute and methods returning void.

# System.ObsoleteAttribute

- The ObsoleteAttribute displays a warning indicating that a method is has been superseded by another method.
- An additional message string can be provided, giving guidance on what steps should be taken.

# ObsoleteAttribute Example

```
// Using ObsoleteAttribute
static void Main(string[] args)
{
    ObsoleteMethod();
}

[Obsolete]
public static void ObsoleteMethod()
{ /* ... */ }
```

# SerializableAttribute

- The SerializableAttribute indicates that an object can be **serialized**, i.e., copied to a stream of bytes.
- This attribute allows a formatter to reflect over the serializable object.
- The object can later be deserialized.
- For example, a document can be saved when shutting down and retrieved at start-up time.

# Serializable Class

```
// Serializable classes use SerializableAttribute.  
[Serializable]  
class Document  
{  
    public string Title = null;  
    public string Data = null;  
  
    // Handles and passwords should not be serialized  
    [NonSerialized]  
    public long _WindowHandle = 0;  
  
    class Image  
    { /* ... */ }  
  
    [NonSerialized]  
    private Image Picture = new Image();  
}
```



# Saving a Serialized Document

```
// Saving a Document Using System.SerializableAttribute
static void Main(string[] args)
{
    Stream stream;
    Document documentBefore = new Document();
    documentBefore.Title =
        "A cacophony of ramblings from my potpourri of notes";

    Document documentAfter;
    using (stream = File.Open(
        documentBefore.Title + ".bin", FileMode.Create))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(stream, documentBefore);
    }

    /* Continued on next page */
}
```

# Saving ... Document (cont'd)

```
// Saving a Document Using System.SerializableAttribute
```

```
/* Continued from previous page */
```

```
using (stream = File.Open(
    documentBefore.Title + ".bin", FileMode.Open))
{
    BinaryFormatter formatter = new BinaryFormatter();
    documentAfter = (Document)formatter.Deserialize(stream);
}
Console.WriteLine(documentAfter.Title);
}
```

# Notes on Serialization

- Serializing an object requires instantiating a formatter and calling the `Serialize()` method.
- Deserializing an object requires a call to the formatter's `Deserialize()` method.
- The return value from `Deserialize()` is of type **object** and must be cast appropriately.
- Passwords and Windows handles should never be serialized because data is not encrypted.

# Custom Serialization

- Custom serialization requires implementing the `ISerializable` interface in addition to using the `SerializableAttribute`.
- The `ISerializable` interface requires that the `GetObjectData()` method be implemented.
- Moreover, a constructor taking parameters of `SerializationInfo` and `StreamingContext` (`System.Runtime.Serialization`) must be provided to support deserialization.

# ISerializable Interface Example

```
// Implementing System.Runtime.Serialization.ISerializable
```

```
[Serializable]
```

```
class EncryptableDocument : ISerializable
```

```
{
```

```
    public EncryptableDocument() { }
```

```
    enum Field { Title, Data }
```

```
    public string Title, Data;
```

```
    public static string Encrypt(string data)
```

```
{
```

```
        string encryptedData = data;
```

```
        // Place key-based encryption here . . . //
```

```
        return encryptedData;
```

```
}
```

```
    public static string Decrypt(string encryptedData)
```

```
{
```

```
        string data = encryptedData;
```

```
        // Place key-based decryption here . . . //
```

```
        return data;
```

```
}
```

```
/* Continued on next page */
```

# ISerializable ... Example (cont'd)

```
// Implementing System.Runtime.Serialization.ISerializable
```

```
/* Continued from previous page */
```

```
#region ISerializable Members
```

```
public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue(Field.Title.ToString(), Title);
    info.AddValue(Field.Data.ToString(), Encrypt(Data));
}
```

```
public EncryptableDocument(SerializationInfo info, StreamingContext context)
{
    Title = info.GetString(Field.Title.ToString());
    Data = Decrypt(info.GetString(Field.Data.ToString()));
}
```

```
#endregion
```

```
}
```

# More Notes on Serialization

- The `SerializationInfo` object is a collection of name/value pairs stored in a dictionary-like structure.
- When serializing, `GetObject()` calls the `AddValue()` method; when deserializing, call one of the `Get` methods, e.g., `GetString()`.
- Encryption and decryption must occur prior to serialization and deserialization, respectively.

# Versioning the Serialization

- Version incompatibilities can arise when a document is serialized using one version of assembly and deserialized using a newer version, which might look for new fields.
- If a field was added to a newer version of the document, deserializing the older version will cause an exception to occur.



# The OptionalFieldAttribute

- The OptionalFieldAttribute introduced in .NET 2.0 can be used to decorate new serialized fields to preserve compatibility with older implementations.
- Prior to .NET 2.0, when changes were made, serialization had to be manually implemented in GetObjectData() (See the next example).

# Backward Compatibility pre-.NET 2.0

```
// Backward Compatibility Prior to the 2.0 Framework
[Serializable]
public class VersionableDocument : ISerializable
{
    enum Field
    { Title, Author, Data }
    public VersionableDocument() { }
    public string Title, Author, Data;

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue(Field.Title.ToString(), Title);
        info.AddValue(Field.Author.ToString(), Author);
        info.AddValue(Field.Data.ToString(), Data);
    }
    /* Continued on the next page */
}
```

# Backward Compatibility (cont'd)

```
/* Continued from the previous page */
public VersionableDocument(SerializationInfo info, StreamingContext context)
{
    foreach (SerializationEntry entry in info)
    {
        switch ((Field)Enum.Parse(typeof(Field), entry.Name))
        {
            case Field.Title:
                Title = info.GetString(Field.Title.ToString());
                break;
            case Field.Author:
                Author = info.GetString(Field.Author.ToString());
                break;
            case Field.Data:
                Data = info.GetString(Field.Data.ToString());
                break;
        }
    }
}
```

# Programming with Dynamic Objects

- **Dynamic objects** are resolved at execution time, i.e., they are not statically typed.
- Reflection is used against the underlying type to dynamically find and invoke a member.
- If no such member is available, the call results in a `RuntimeBinderException`.
- At compile time, no checks are made as to whether the members specified are available.

# Dynamic Programming Example

```
// Dynamic Programming Using "Reflection"
// ... //
dynamic data = "Hello! My name is Inigo Montoya";

Console.WriteLine(data);

data = (double)data.Length;
data = data*3.5 + 28.6;
if(data == 2.4 + 112 + 26.2)
{
    Console.WriteLine("{0} makes for a long triathlon.", data);
}
else
{
    data.NonExistentMethodCallStillCompiles()
}
// ... //
```

# Dynamic Principles

- **dynamic** is a directive to the compiler to generate code involving an interception mechanism wrapping the original type so that no compile-time validation occurs.
- When a member is invoked at runtime, the wrapper intercepts the call and dispatches (or rejects) it appropriately.
- Any type will convert to **dynamic**.

# Dynamic Principles (cont'd)

- Successful conversion from `dynamic` to an alternate type depends on support in the underlying type; conversion from a `dynamic` object to a CLR type requires an explicit cast.
- The type underlying the `dynamic` type can change from one assignment to the next, unlike the implicitly typed variable (`var`), which cannot be reassigned.

# Dynamic Principles (cont'd)

- Verification that the specified signature exists on the **dynamic** type occurs at runtime.
- The return value from any **dynamic** member invocation is a **dynamic** object.
- Calling GetType() on the **dynamic** object returns the correct compiled type.
- If the member does not exist at runtime, the `RuntimeBinderException` is thrown.



# Dynamic Principles (cont'd)

- `dynamic` with reflection does not support extension methods.
- At its core, `dynamic` is a `System.Object`, given that any object will successfully convert to `dynamic` and that the default value is `null`.
- Custom types can also be invoked dynamically.

# Example of Dynamic Objects

## // Strongly typed syntax

```
XElement person = XElement.Parse(
    @"<Person>
    <FirstName>Inigo</FirstName>
    <LastName>Montoya</LastName>
    </Person>");
Console.WriteLine("{0} {1}",
    person.Descendants("FirstName").FirstOrDefault().Value,
    person.Descendants("LastName").FirstOrDefault().Value);
```

## // Dynamically typed object

```
dynamic person = DynamicXml.Parse(
    @"<Person>
    <FirstName>Inigo</FirstName>
    <LastName>Montoya</LastName>
    </Person>");
Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
```

# Static Compilation vs. Dynamic Programming

- In situations where type safety cannot be checked at compile time, a dynamic call may be more readable and succinct.
- If compile-time verification is possible, statically typed programming is preferred.
- Once again, in cases where this isn't possible, dynamic programming provides simpler code.

# Implementing a Custom Dynamic Object

- The key to defining a custom dynamic type is to implement the `IDynamicMetaObjectProvider` interface (`System.Dynamic` namespace).
- The preferred approach is to derive a custom dynamic type from `System.Dynamic.DynamicObject`, which already implements the interface and provides default implementation for a host of members.

# Custom Dynamic Object

```
// Implementing a Custom Dynamic Object
public class DynamicXml : DynamicObject
{
    private XElement Element { get; set; }
    public DynamicXml(System.Xml.Linq.XElement element)
    { Element = element; }

    public static DynamicXml Parse(string text)
    { return new DynamicXml(XElement.Parse(text)); }

    /* Continued on next page */
}
```

# Custom Dynamic Object (cont'd)

```
/* Continued from previous page */  
public override bool TryGetMember(GetMemberBinder binder, out object result)  
{  
    bool success = false;  
    result = null;  
    XElement firstDescendant =  
        Element.Descendants(binder.Name).FirstOrDefault();  
    if (firstDescendant != null)  
    {  
        if (firstDescendant.Descendants().Count() > 0)  
        { result = new DynamicXml(firstDescendant); }  
        else  
        { result = firstDescendant.Value; }  
        success = true;  
    }  
    return success;  
}  
/* Continued on next page */
```

# Custom Dynamic Object (cont'd)

```
/* Continued from previous page */  
public override bool TrySetMember(SetMemberBinder binder, object value)  
{  
    bool success = false;  
    XElement firstDescendant =  
        Element.Descendants(binder.Name).FirstOrDefault();  
  
    if (firstDescendant != null)  
    {  
        if (value.GetType() == typeof(XElement))  
        { firstDescendant.ReplaceWith(value); }  
        else  
        { firstDescendant.Value = value.ToString(); }  
        success = true;  
    }  
    return success;  
}
```

# Unit 17 Summary

- This Unit discussed the use of reflection to read the metadata compiled into an assembly and to use late binding to invoke methods defined at execution time.
- Reflection enables the retrieval of metadata in the form of attributes.
- Finally, the Unit included dynamic programming, a new C# 4.0 feature that enables working with dynamic data.



# Unit 18

## Multithreading

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 18 Overview

- This Unit introduces multithreading.
- Writing code to take advantage of multiple processing units is also discussed.
- .NET 4.0 introduced two new APIs for multithreaded programming, the Task Parallel Library (TPL) and Parallel LINQ (PLINQ), each of which will be reviewed.

# Thread Basics

- A **thread** is a sequence of instructions that may run concurrently with other instructions.
- A program that enables more than one sequence of instructions to execute concurrently is **multithreaded**.
- An operator system simulates the simultaneous execution of multiple threads via a mechanism called **time slicing**, which switches execution quickly between threads.

# Time Slicing

- The period of time that the processor is executing a particular thread is called a **time slice** or **quantum**.
- Each thread of a multithreaded process appears to run continuously with other threads.
- Since a thread is often waiting for various events, like I/O, switching to a different thread may result in more efficient execution.

# Atomicity

- A set of operations is said to be **atomic** if one of the following conditions is met:
  - The entire set of operations must complete before any operation appears to have executed.
  - The apparent state of the system must return to the state prior to any operation executing—as though no steps were executed.
- Identifying and implementing **atomicity** is one of the primary complexities of multithreading.

# Deadlocks

- C# supports the ability to restrict access to blocks of code to prevent **deadlocks**, in which two threads are blocked waiting for the other.
- The problem with code that is not atomic or causes deadlocks is that it depends on the order in which instructions execute across multiple threads.
- This introduces uncertainty and instability into the application.

# Running and Controlling a Thread

- The operating system creates and manages threads and provides various unmanaged APIs to manipulate them.
- The Common Language Runtime (CLR) wraps these unmanaged threads and exposes them in managed code through the `System.Threading.Tasks.Task` class, representing an asynchronous class.

# The Task

- The Task does not map directly to an unmanaged thread, but rather provides a degree of abstraction to an underlying thread.
- Because creating a thread is an expensive operation, the Task requests threads from the **thread pool** and may reuse an existing thread, instead of creating a new one.
- This process is managed by the Task class.



# Starting a Separate Thread

```
// Starting a Method in a Separate Thread
public static void Main()
{
    const int repetitions = 10000;
    Task task = new Task(() =>
    {
        for (int count = 0; count < repetitions; count++)
            { Console.Write('-'); }
    });
    task.Start();
    for (int count = 0; count < repetitions; count++)
        { Console.Write('.'); }
    // Wait until the Task completes
    task.Wait();
}
```

# Starting a Separate Thread Output

```
.....-----  
-----  
-----  
-----  
.....  
.....  
.....-----  
-----  
-----  
-----
```

# Returning a Result from a Task

```
// Returning a Result from a Task<TResult>
public static void Main()
{
    Task<string> task = Task.Factory.StartNew<string>(
        () => PiCalculator.Calculate(100));
    foreach (char busySymbol in Utility.BusySymbols())
    {
        if (task.IsCompleted)
        {
            Console.Write('\b');
            break;
        }
        Console.Write(busySymbol);
    }
    // Blocks until task completes
    Console.WriteLine(task.Result);
    System.Diagnostics.Trace.Assert(task.IsCompleted);
}
```

# Returning a Result (cont'd)

```
public class Utility
{
    public static IEnumerable<char> BusySymbols()
    {
        string busySymbols = @"-\\|/-\\|/";
        int next = 0;
        while (true)
        {
            yield return busySymbols[next];
            next = (++next) % busySymbols.Length;
            yield return '\\b';
        }
    }
}
```

# Some Properties of Task Class

- **Status**: returns a TaskStatus enum indicating the status of the task.
- **IsCompleted**: set to true when a task completes whether or not it faults.
- **Id**: a unique identifier for the task.
- **AsyncState**: additional data about the task.
- **Task.CurrentId**: returns an identifier for currently executing task.

# ContinueWith()

- The ContinueWith() method of the Task class provides a means of chaining tasks together
- The ContinuationOptions enum is a list of bitwise flags for managing continuation tasks.
- ContinueWith() can be called multiple times on the same task, meaning multiple tasks will commence when the original task completes.
- The effect is to register for “events” on the antecedent’s task.

# ContinueWith() Example

```
// Registering for "Notifications" with ContinueWith()
static void Main(string[] args)
{
    Task<string> task =
        Task.Factory.StartNew<string>(() => PiCalculator.Calculate(10));
    Task faultedTask = task.ContinueWith((antecedentTask) =>
        { Console.WriteLine("Task State: Faulted"); },
        TaskContinuationOptions.OnlyOnFaulted);

    Task canceledTask = task.ContinueWith((antecedentTask) =>
        { Console.WriteLine("Task State: Canceled"); },
        TaskContinuationOptions.OnlyOnCanceled);

    Task completedTask = task.ContinueWith((antecedentTask) =>
        { Console.WriteLine("Task State: Completed"); },
        TaskContinuationOptions.OnlyOnRanToCompletion);
    completedTask.Wait();
}
```

# Unhandled Exceptions on Task

- An unhandled exception during a Task's execution will be suppressed until one of the task completion members is called: `Wait()`, `Result`, `Task.WaitAll()`, or `Task.WaitAny()`.
- Each of these members will throw an exception if an unhandled exception occurred during the task's execution.
- By placing the task completion member inside a `try...catch` block, the exception can be caught.



# Handling Unhandled Exceptions

```
// Handling a Task's Unhandled Exception
public static void Main()
{
    Task task = Task.Factory.StartNew(() =>
    {
        throw new ApplicationException();
    });
    try
    {
        task.Wait();
    }
    catch (AggregateException exception)
    {
        foreach (Exception item in exception.InnerExceptions)
        { Console.WriteLine("ERROR: {0}", item.Message); }
    }
}
```

# Exceptions Using ContinueWith()

```
public static void Main()
{ // Unhandled Exceptions Using ContinueWith()
    bool parentTaskFaulted = false;
    Task task = new Task(() => { throw new ApplicationException(); });

    Task faultedTask = task.ContinueWith((parentTask) =>
    {
        parentTaskFaulted = parentTask.IsFaulted;
    }, TaskContinuationOptions.OnlyOnFaulted);
    task.Start();
    faultedTask.Wait();
    Trace.Assert(parentTaskFaulted);

    if (!task.IsFaulted)
    { task.Wait(); }
    else
    { Console.WriteLine("ERROR: {0}", task.Exception.Message); }
}
```

# Canceling Tasks

- In .NET 3.5, cancellation of a thread was forced, and the target thread had little choice in the matter; a `ThreadAbortException` was then thrown in the target thread.
- In .NET 4.0, cancellation is cooperative; instead of the calling thread cancelling the target thread, a cancellation flag is set.
- The task targeted for cancellation can then respond appropriately to the cancel request.

# Cancelling a Task

```
// Canceling a Task Using CancellationToken
public static void Main()
{
    string stars = "*".PadRight(Console.WindowWidth - 1, '*');
    Console.WriteLine("Push ENTER to exit.");

    CancellationTokenSource cancellationTokenSource =
        new CancellationTokenSource();
    Task task = Task.Factory.StartNew(
        () => WritePi(cancellationTokenSource.Token),
        cancellationTokenSource.Token);

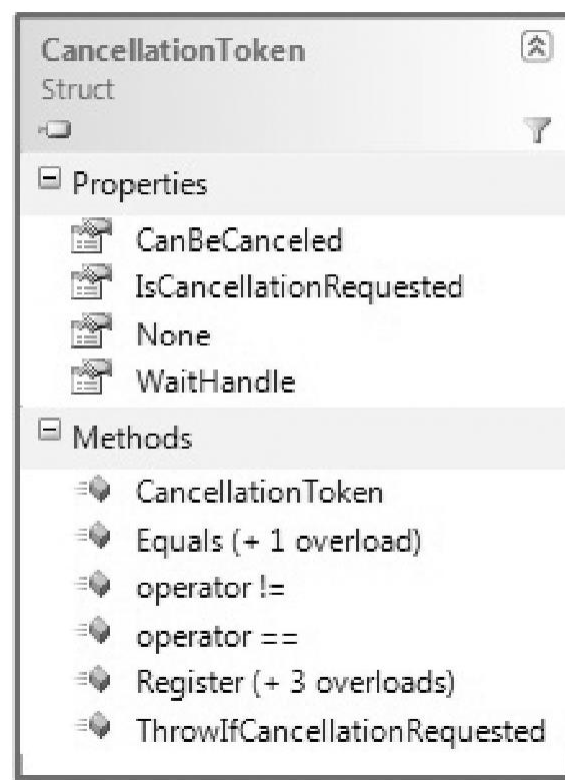
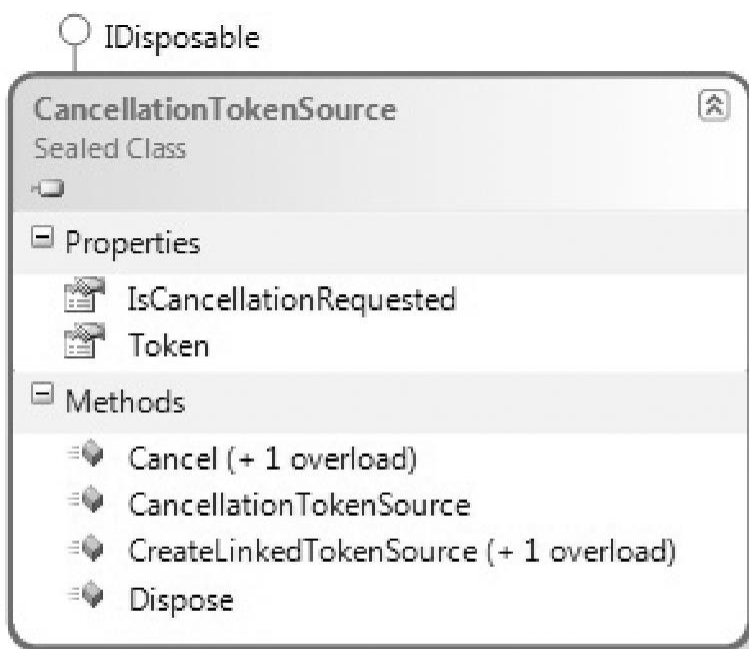
    Console.ReadLine();    // Wait for the user's input
    cancellationTokenSource.Cancel();
    Console.WriteLine(stars);
    task.Wait();
    Console.WriteLine();
}
```

# Cancelling a Task (cont'd)

```
private static void WritePi(CancellationToken cancellationToken)
{
    const int batchSize = 1;
    string piSection = string.Empty;
    int i = 0;

    while (!cancellationToken.IsCancellationRequested || i == int.MaxValue)
    {
        piSection = PiCalculator.Calculate(batchSize, (i++) * batchSize);
        Console.Write(piSection);
    }
}
```

# CancellationTokenSource



# Notes on Cancellation Tokens

- The `Cancel()` call sets the `IsCancellationRequested` property on all cancellation tokens copied from `CancellationTokenSource.Token`.
- Note that a `CancellationToken`, not a `CancellationTokenSource`, is evaluated in the asynchronous task.

# More Notes on Cancellation Tokens

- A CancellationToken is a struct, so calling CancellationTokenSource.Token will create a thread-safe copy of the token.
- To monitor the IsCancellationRequested property, an instance of the CancellationToken is passed to the parallel task.



# Long-Running Tasks

- When a task is expected to hold a thread for a long time, a best practice is to notify the thread pool when creating the task by passing the `TaskCreationOptions.LongRunning` option.

```
// Cooperatively Executing Long-Running Tasks
Task task = Task.Factory.StartNew(() =>
    WritePi(cancellationTokenSource.Token),
    TaskCreationOptions.LongRunning);
```

# Disposing of a Task

- The Task class supports the IDisposable, so it is considered a best practice to invoke the Dispose() method on exit.

# Executing Iterations in Parallel

- .NET 4.0 implements a **parallel for** capability through an API on `System.Threading.Tasks.Parallel`.
- With a parallel for loop, multiple iterations can run in parallel, if there are multiple processors, resulting in decreased execution time.
- The parallel for loop looks similar to a standard for loop.

# Parallel Iterations Example

```
// For Loop Synchronously Calculating Pi in Sections
using System;

const int TotalDigits = 100;
const int BatchSize = 10;
class Program
{
    void Main()
    {
        string pi = null;
        int iterations = TotalDigits / BatchSize;
        for (int i = 0; i < iterations; i++)
        { pi += PiCalculator.Calculate(BatchSize, i * BatchSize); }

        Console.WriteLine(pi);
    }
}
```

# Parallel Iterations (cont'd)

```
// For Loop Calculating Pi in Sections in Parallel
using System;
using System.Threading;
const int TotalDigits = 100;
const int BatchSize = 10;
class Program
{
    void Main()
    {
        string pi = null;
        int iterations = TotalDigits / BatchSize;
        string[] sections = new string[iterations];
        Parallel.For(0, iterations, (i) =>
            { sections[i] += PiCalculator.Calculate(BatchSize, i * BatchSize); });
        pi = string.Join("", sections);
        Console.WriteLine(pi);
    }
}
```

# Parallel Iterations of foreach loop

```
// Parallel Execution of a foreach Loop
using System;
using System.Threading.Tasks;
class Program
{
    // ... //
    static void EncryptFiles(string directoryPath, string searchPattern)
    {
        IEnumerable<string> files = Directory.GetFiles(
            directoryPath, searchPattern, SearchOption.AllDirectories);

        Parallel.ForEach(files, (fileName) =>
        {
            Encrypt(fileName);
        });
    }
}
```

# Parallelism

- The **degree of parallelism** corresponds to the number of threads that run simultaneously at any particular time.
- Efficiency is determined by a “hill climbing” algorithm in which additional threads are created until the overhead of adding threads begins to decrease overall performance.

# Parallel Exception Handling

- With parallel threads, the potential exists for multiple exceptions.
- If an exception is thrown, the exception type is `System.AggregateException`, which groups together multiple exceptions using inner exceptions.
- Thus, all exceptions within a loop can be handled within a single try/catch block.



# Parallel Exception Handling

// Unhandled Exception Handling for Parallel Iterations

```
static void EncryptFiles(string directoryPath, string searchPattern)
{
    IEnumerable<string> files = Directory.GetFiles(
        directoryPath, searchPattern, SearchOption.AllDirectories);
    try {
        Parallel.ForEach(files, (fileName) => { Encrypt(fileName); });
    }
    catch (AggregateException exception) {
        Console.WriteLine("ERROR: {0}:",
            exception.GetType().Name);
        foreach (Exception item in exception.InnerExceptions)
        {
            Console.WriteLine(" {0} - {1}",
                item.GetType().Name, item.Message);
        }
    }
}
```

# Cancelling a Parallel Loop

- Cancelling a parallel loop requires invocation of the cancellation request from a thread other than the one executing the parallel loop because a parallel loop blocks until finished.
- Parallel loops use the same cancellation token pattern that Tasks use.
- The `CancellationTokenSource.Token` property is associated with the parallel loop via overloads on the parallel loops.

# Cancelling a Parallel Loop Example

```
static void EncryptFiles(string directoryPath, string searchPattern)
{
    IEnumerable<string> files = Directory.GetFiles(
        directoryPath, searchPattern, SearchOption.AllDirectories);
    CancellationSource cts = new CancellationSource();
    ParallelOptions parallelOptions =
        new ParallelOptions { CancellationToken = cts.Token };
    cts.Token.Register(() => Console.WriteLine("Cancelling..."));

    Console.WriteLine("Push ENTER to exit.");
    Task task = Task.Factory.StartNew(() =>
    {
        try
        {
            Parallel.ForEach(files, parallelOptions,
                (fileName, loopState) => { Encrypt(fileName); });
        }
        catch (OperationCanceledException) { }
    });
}
```

**// Continued on next page //**

# Cancelling a Parallel Loop (cont'd)

```
static void EncryptFiles(string directoryPath, string searchPattern)
{
    // Continued from previous page //

    // Wait for the user's input
    Console.Read();

    // Cancel the query
    cts.Cancel();
    Console.Write(stars);
    task.Wait();
}
```

# Running LINQ Queries in Parallel

- LINQ queries can be run in parallel with minimal changes.
- All that is needed is a standard query operator, `AsParallel()`, on the static class `System.Linq.ParallelEnumerable`.
- `ParallelEnumerable` contains a superset of query operators available from `System.Linq.Enumerable`.
- Parallel LINQ means better performance.

# Parallel LINQ Example 1

```
// LINQ Select()
class Cryptographer
{
    // ... //
    public List<string> SynchronousEncrypt(List<string> data)
    { return data.Select((item) => Encrypt(item)).ToList(); }
}

// Parallel LINQ Select()
class Cryptographer
{
    // ... //
    public List<string> ParallelEncrypt(List<string> data)
    { return data.AsParallel().Select((item) => Encrypt(item)).ToList(); }
}
```

# Parallel LINQ Example 2

```
// Parallel LINQ with Standard Query Operators
```

```
ParallelQuery<IGrouping<char, string>> parallelGroups;  
parallelGroups = data.AsParallel().OrderBy(item => item);
```

```
// Show the total count of items still matches the original count
```

```
System.Diagnostics.Trace.Assert(data.Count ==  
    parallelGroups.Sum(item => item.Count()));
```

# Parallel LINQ Example 3

```
// Parallel LINQ with Query Expressions
ParallelQuery<IGrouping<char, string>> parallelGroups;
parallelGroups = from text in data.AsParallel()
                 orderby text
                 group text by text[0];

// Show the total count of items still matches the original count
System.Diagnostics.Trace.Assert(data.Count ==
    parallelGroups.Sum(item => item.Count()));
```



# Parallel LINQ Exceptions

- The mechanism for catching multiple exceptions in a Parallel LINQ query is the `AggregateException` class.
- Wrapping a Parallel LINQ (PLINQ) query in a `try...catch` block with an `AggregateException` catch block will successfully handle any exceptions encountered.

# Cancelling a PLINQ Query

- As with a parallel loop, a PLINQ query requires a `CancellationToken`, available on a `CancellationTokenSource.Token` property.
- The `ParallelQuery<T>` object returned by the `IEnumerable`'s `AsParallel()` method includes a `WithCancellation()` extension method that takes a `CancellationToken`.
- Calling `Cancel()` on the `CancellationTokenSource` object will request the parallel query to cancel.

# Cancelling a PLINQ Query Example

// Canceling a Parallel Loop

```
public static List<string> ParallelEncrypt(List<string> data,  
    CancellationToken cancellationToken)  
{  
    return data.AsParallel().WithCancellation(  
        cancellationToken).Select((item) => Encrypt(item)).ToList();  
}
```

// Continued on the next page //

# Cancelling a PLINQ (cont'd)

```
// Continued from the previous page //
public static void Main()
{
    List<string> data = Utility.GetData(1000000).ToList();
    CancellationTokenSource cts = new CancellationTokenSource();

    Console.WriteLine("Push ENTER to exit.");
    Task task = Task.Factory.StartNew(() =>
        { data = ParallelEncrypt(data, cts.Token); }, cts.Token);

    // Wait for the user's input
    Console.Read();
    cts.Cancel();
    Console.Write(stars);

    try { task.Wait(); }
    catch (AggregateException) { }
}
```

# Multithreading before .NET 4.0

- The fundamental Thread type (System.Threading) was used to control an asynchronous operation before .NET 4.0.
- The Thread class includes a Start method and a wait equivalent, Join().
- A delegate of type ThreadStart or ParameterizedThreadStart identifies the method to execute.
- The thread is started with a call to Start().

# Example of Thread Class

```
// Starting a Method Using System.Threading.Thread
public class RunningASeparateThread
{
    public const int Repetitions = 1000;
    public static void Main()
    {
        ThreadStart threadStart = DoWork;
        Thread thread = new Thread(threadStart);
        thread.Start();

        for (int count = 0; count < Repetitions; count++)
        { Console.WriteLine('-'); }
        thread.Join();
    }
    public static void DoWork()
    { for (int count = 0; count < Repetitions; count++)
      { Console.WriteLine('.'); } }
}
```

# Example of Thread Class (cont'd)

```
.....-----  
-----  
-----  
.....  
.....-----  
-----  
-----  
-----  
.....  
.....  
.....-----
```

# Thread Management Members 1

Thread Member	Description
<b>Join()</b>	This method causes the calling thread to wait for the thread instance to terminate.
<b>IsBackground</b>	By default, a thread runs in the foreground, meaning a process will not terminate until the thread completes. Setting this property to true allows process execution to terminate before the thread's completion.
<b>Priority</b>	The priority with which a thread runs which is described by the ThreadPriority enumeration: Lowest, BelowNormal, Normal, AboveNormal, or Highest.



# Thread Management Members 2

Thread Member	Description
<b>ThreadState</b>	The state of the thread, which is described by the ThreadState enumeration: Aborted, AbortRequested, Background, Running, Stopped, StopRequested, Suspended, SuspendRequested, Unstarted, and WaitSleepJoin.
<b>Thread.Sleep()</b>	This static method pauses the current thread for a period of time.
<b>Abort()</b>	This method aborts the thread and causes a ThreadAbortException to be thrown within the target thread. Uncertainty is introduced into the thread's behavior, and data integrity or resource cleanup problems may result. Abort should be a last resort, only.

# Good Advice About Threads

- In general, do not abort a thread: wait for it to complete. Aborting a thread introduces instability into the target thread and the application.
- The priority for selecting from the available asynchronous options is the following: Task, ThreadPool, and Thread—in that order.

# Thread Pooling

- TPL makes use of the CLR's thread pool, `System.Threading.ThreadPool`.
- The thread pool determines when to reuse an existing thread rather than create a new one.
- The `ThreadPool` accesses threads through the `QueueUserWorkItem()` static method.
- Note that the thread pool does not return a handle to the thread or task; this prevents the calling thread from managing the target thread.

# Using the ThreadPool

```
class Program
{
    // Using ThreadPool Instead of Instantiating Threads Explicitly
    public const int Repetitions = 1000;
    public static void Main()
    {
        ThreadPool.QueueUserWorkItem(DoWork, '.');
        for (int count = 0; count < Repetitions; count++)
        { Console.WriteLine('-'); }
        Thread.Sleep(1000); // Pause until the thread completes
    }

    public static void DoWork(object state)
    {
        for (int count = 0; count < Repetitions; count++)
        { Console.WriteLine(state); }
    }
}
```

# Unhandled Exceptions on AppDomain

- To catch exceptions from any thread started by the program itself, surround the root code block with a try..catch..finally statement.
- The guideline for unhandled exceptions is for the program to shut down and restart because the application may be in an unstable state.
- By registering for the UnhandledException event on the AppDomain, notification will be received in time to log the error or save data.

# Registering for Unhandled Exceptions

```
// Registering for Unhandled Exceptions
static void Main(string[] args)
{
    try
    {
        // Register a callback to receive notifications of unhandled exception.
        AppDomain.CurrentDomain.UnhandledException +=
            OnUnhandledException;

        ThreadPool.QueueUserWorkItem(
            state => { throw new Exception("Arbitrary Exception"); });

        Thread.Sleep(10000); // Wait for the unhandled exception to fire
        Console.WriteLine("Still running...");
    }
    finally
    { Console.WriteLine("Exiting..."); }
}
```

# Registering Exceptions (cont'd)

```
// Continued from previous slide //
static void OnUnhandledException(object sender,
    UnhandledExceptionEventArgs eventArgs)
{
    Exception exception = (Exception)eventArgs.ExceptionObject;
    Console.WriteLine("ERROR ({0}):{1} ---> {2}",
        exception.GetType().Name,
        exception.Message,
        exception.InnerException.Message);
}
```

## Output:

Still running...

Exiting...

ERROR (AggregateException):One or more errors occurred. ---> Arbitrary  
Exception

# Unhandled Exception Callback

- The UnhandledException callback will fire on all unhandled exceptions on threads within the application domain.
- This is a notification mechanism only, not a means to catch and process exceptions and then to continue the application.
- After the event, the application must exit; in fact, an error dialog will display.



# Unit 18 Summary

- This Unit delved into the creation and manipulation of threads using the new Task Parallel Library (TPL).
- In addition, Parallel LINQ (PLINQ), in which a single extension method, `AsParallel()`, transforms all further LINQ queries to run in parallel, was also presented.
- Finally, the Unit closed with a section on multithreaded programming prior to TPL.

# Unit 19

## Synchronization and More Multithreading Patterns

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 19 Overview

- The last Unit discussed the details of multithreaded programming using the Task Parallel Library (TPL) and Parallel LINQ (PLINQ).
- Thread synchronization is the topic of this Unit.
- The examples used in this Unit make use of the Task class from the System.Threading.Tasks namespace.

# Synchronization

- A **race condition** occurs when multiple threads have access to the same data elements.
- When multiple threads access data simultaneously, access must be synchronized to prevent data integrity problems.
- Code or data synchronized for simultaneous access by multiple threads is **thread-safe**.

# Synchronization Example

```
class Program
{
    const int _Total = int.MaxValue;
    static long _Count = 0;
    static void Main()    // Unsynchronized State
    {
        Task task = Task.Factory.StartNew(Decrement);
        for (int i = 0; i < _Total; i++) // Increment
        { _Count++; }
        task.Wait();
        Console.WriteLine("Count = {0}", _Count);
    }
    static void Decrement()
    {
        for (int i = 0; i < _Total; i++)
        { _Count--; }
    }
} // Output : Count = 113449949
```

# Multiple Threads and Local Variables

- Local variables do not need to be synchronized because each local variable has its own instance for each method call.
- Local variables are loaded on the stack, and each thread has its own logical stack.
- Some applications do expose local variables to multiple threads, e.g., a parallel-for loop sharing a local variable between iterations, exposing the variable to race conditions.

# Unsynchronized Local Variables

```
// Unsynchronized Local Variables
public static void Main()
{
    int x = 0;
    Parallel.For(0, int.MaxValue, i =>
    {
        x++;
        x--;
    });
    Console.WriteLine("Count = {0}", x);
}
```

# Synchronization with Monitor

- The Monitor class (`System.Threading.Monitor`) is used to protect a block of code from simultaneous access by multiple threads.
- The beginning and end of the protected code block is marked by calls to the static methods `Monitor.Enter()` and `Monitor.Exit()`.
- A good practice is to enclose calls to `Monitor.Enter()` and `Monitor.Exit()` within an exception handler (`try...catch...finally`).



# Monitor Synchronization Example

```
// Synchronizing with a Monitor Explicitly
static void Decrement()
{
    for (int i = 0; i < _Total; i++)
    {
        bool lockTaken = false;
        Monitor.Enter(_Sync, ref lockTaken);
        try
        { _Count--; }
        finally
        {
            if (lockTaken)
            { Monitor.Exit(_Sync); }
        }
    }
}
```

# Using the lock Keyword

- Locking a block of code using the **lock** keyword will render it thread-safe.

```
// Synchronization Using the lock Keyword
readonly static object _Sync = new object();
public static void Main()
{
    // ... //
    for (int i = 0; i < _Total; i++)
    {
        lock (_Sync)
        { _Count++; }    // Increment
    }
    // ... //
}
```

# The Locking Object

- The locking object should be carefully chosen and not be a value or string type.
- A common pattern is to use `this` as a locking object for instance data and `typeof(<type>)` for static data.
- A better approach is to define a private, read-only field for each block of code to be synchronized.

# System.Threading.Interlocked Methods

Method Signature	Description
<code>public static T CompareExchange&lt;T&gt;(T location, T value, T comparand);</code>	Checks location for the value in comparand. If the values are equal, it sets location to value and returns the original data stored in location.
<code>public static T Exchange&lt;T&gt;(T location, T value);</code>	Assigns location with value and returns the previous value.
<code>public static int Decrement(ref int location);</code>	Decrements location by one. It is equivalent to the – operator, except Decrement() is thread-safe.

# System.Threading.Interlocked Methods (cont'd)

Method Signature	Description
<code>public static int Increment(   ref int location );</code>	Increments location by one. It is equivalent to the ++ operator, except Increment() is thread-safe.
<code>public static int Add(   ref int location,   int value );</code>	Adds value to location and assigns location the result.
<code>public static long Read(   ref long location );</code>	Returns a 64-bit value in a single atomic operation.

# System.Threading.InterLocked Example

```
// Synchronization Using System.Threading.Interlocked
class SynchronizationUsingInterlocked
{
    private static object _Data;

    // Initialize data if not yet assigned.
    static void Initialize(object newValue)
    {
        // If _Data is null then set it to newValue.
        Interlocked.CompareExchange(ref _Data, newValue, null);
    }
    // ... //
}
```

# Synchronization Best Practices

- All static data should be thread-safe, i.e., synchronization should be used to protect access to static data that is mutable.
- Instance data is not expected to include synchronization logic but may include it if objects are shared across multiple threads.
- Unnecessary synchronization should be avoided for performance reasons.

# System.Threading.Mutex

- The System.Threading.Mutex class is similar to System.Threading.Monitor except that it is not used by the **lock** keyword.
- Mutex is a cross-process resource and can be used, for example, to restrict an application from executing multiple times.
- Mutex inherits from System.Threading.WaitHandle.



# Mutex Example

```
// Creating a Single Instance Application
public static void Main()
{
    // Indicates whether this is the first application instance
    bool firstApplicationInstance;

    // Obtain the mutex name from the full assembly name.
    string mutexName = Assembly.GetEntryAssembly().FullName;

    using (Mutex mutex = new Mutex(false, mutexName,
                                     out firstApplicationInstance))
    {
        if (!firstApplicationInstance)
        { Console.WriteLine("This application is already running."); }
        else
        {
            Console.WriteLine("ENTER to shutdown");
            Console.ReadLine();
        }
    }
}
```

# Concurrent Collection Classes

- The concurrent collection classes provide built-in synchronization code that supports simultaneous access by multiple threads.
- Concurrent collections support thread-safe access by producers and consumers.
- These classes are contained in the `System.Collections.Concurrent` namespace.

# Concurrent Collection Classes

Collection Class	Description
BlockingCollection<T>	Provides a blocking collection that enables producer/consumer scenarios in which producers write data into the collection while consumers read the data.
*ConcurrentBag<T>	A thread-safe unordered collection of T type objects.
ConcurrentDictionary<TKey, TValue>	A thread-safe dictionary; a collection of keys and values.
*ConcurrentQueue<T>	A thread-safe queue supporting first in, first out (FIFO) semantics on objects of type T.
*ConcurrentStack<T>	A thread-safe stack supporting last in, first out (LIFO) semantics on objects of type T.

\* Collection classes that implement `IProducerConsumerCollection<T>`.

# Thread Local Storage

- With thread local storage, each thread has its own dedicated instance of a variable, so there is no need for synchronizing access.
- Two examples of thread local storage implementation are `ThreadLocal<T>` and `ThreadStaticAttribute`.

# Examples of Thread Local Storage

- Declaring a field as `ThreadLocal<T>` results in a different instance of the field for each thread.

```
// Using ThreadLocal<T> for Thread Local Storage
static ThreadLocal<double> _Count =
    new ThreadLocal<double>(() => 0.01134);
```

- The `ThreadStaticAttribute` also designates a static variable as one instance per thread.

```
// Using ThreadStaticAttribute for Thread Local Storage
[ThreadStatic]
static double _Count = 0.01134;
```

# Timers

- Several classes are available:
  - `System.Windows.Forms.Timer`
  - `System.Timers.Timer`
  - `System.Threading.Timer`.
- `System.Windows.Forms.Timer` is designed for use on a Windows form.
- `System.Timers.Timer` is a wrapper for `System.Threading.Timer`, adding functionality .

# Various Timer Characteristics 1

Feature Description	System. Timers. Timer	System. Threading. Timer	System. Windows. Forms.Timer
Support for adding and removing listeners after the timer is instantiated	Yes	No	Yes
Supports callbacks on the user interface thread	Yes	No	Yes
Calls back from threads obtained from the thread pool	Yes	Yes	No
Supports drag-and-drop in the Windows Forms Designer	Yes	No	Yes
Suitable for running in a multithreaded server environment	Yes	Yes	No

# Various Timer Characteristics 2

Feature Description	System. Timers. Timer	System. Threading. Timer	System. Windows. Forms.Timer
Includes support for passing arbitrary state from the timer initialization to the callback	No	Yes	No
Implements IDisposable	Yes	Yes	Yes
Supports on-off callbacks as well as periodic repeating callbacks	Yes	Yes	Yes
Accessible across application domain boundaries	Yes	Yes	Yes
Supports IComponent; hostable in an IContainer	Yes	No	Yes



# Using System.Timers.Timer

```
// Using System.Timers.Timer
// ... //

using( Timer timer = new Timer() )
{
    // Initialize Timer
    timer.AutoReset = true;
    timer.Interval = 1000;
    timer.Elapsed += new ElapsedEventHandler(Alarm);
    timer.Start();
}

// ... //
static void Alarm(object sender, ElapsedEventArgs eventArgs)
{
    // ... //
}
```

# Using System.Threading.Timer

```
// Using System.Threading.Timer
// ... //

// Timer(callback, state, dueTime, period)
using( Timer timer =new Timer(Alarm, null, 0, 1000) )
{
    // ... //
}

// ... //
static void Alarm(object state)
{
    // ... //
}
```

# Asynchronous Programming

- Multithreading programming includes the following complexities:
  1. Monitoring an asynchronous operation state for completion;
  2. Thread pooling;
  3. Avoiding deadlocks;
  4. Providing atomicity across operations and synchronizing data access.

# Asynchronous Programming Model

- The Asynchronous Programming Model (APM) uses a `BeginX()` method to start an asynchronous operation and an `EndX()` method to conclude the operation.

# The IAsyncResult Object

- The BeginX() method returns an object implementing IAsyncResult and providing access to the state of the asynchronous call and a convenient handle.
- The EndX() method takes this object as an input parameter.
- The IAsyncResult's WaitHandle object can be used to determine when the asynchronous method call is complete.

# Calling the APM on WebRequest

```
// Calling the APM on WebRequest
```

```
// ... //
```

```
IAsyncResult asyncResult = webRequest.BeginGetResponse(null, null);
```

```
// ... //
```

```
while (!asyncResult.AsyncWaitHandle.WaitOne(100))  
{ // ... // }
```

```
WebResponse response =webRequest.EndGetResponse(asyncResult);
```

```
// ... //
```

# APM Signatures Example

```
// APM Parameter Distribution Example
```

```
// BeginX method
```

```
System.IAsyncResult BeginTryDoSomething(  
    string url, ref string data, out string[] links,  
    System.AsyncCallback callback, object state)
```

```
// EndX method
```

```
bool EndTryDoSomething (ref string data, out string[] links,  
    System.IAsyncResult result);
```

```
// The asynchronous method
```

```
bool TryDosomething(string url, ref string data, out string[] links)
```

# Notes on APM Signatures

- The **callback** parameter to the BeginX() method, System.AsyncCallback, is a delegate that is called when the method completes.
- The **state** parameter is used to pass additional data to the callback when it executes.

```
// Invoking the APM with Callback and State
State state = new State(webRequest);
IAsyncResult asyncResult = webRequest.BeginGetResponse(
    GetResponseAsyncCompleted, state);
```



# Calling the APM on WebRequest

```
// Calling the APM on WebRequest
```

```
// ... //
```

```
State state = new State(webRequest);
```

```
IAsyncResult asyncResult =
```

```
    webRequest.BeginGetResponse(GetResponseAsyncCompleted, state);
```

```
// ... //
```

```
while (!asyncResult.AsyncWaitHandle.WaitOne(100))
```

```
{ // ... // }
```

```
WebResponse response = webRequest.EndGetResponse(asyncResult);
```

```
// ... //
```

# Asynchronous Delegates

- A **delegate** is a type that references a method.
- Delegate instances provide compiler-generated `BeginInvoke()` and `EndInvoke()` methods and use threads from the thread pool.
- Given a delegate instance of `Func<string,int>`, a pair of methods such as the following exists:

```
System.IAsyncResult BeginInvoke(  
    string arg, AsyncCallback callback, object @object)  
int EndInvoke(IAsyncResult result)
```

# Asynchronous Delegate Invocation

```
// Asynchronous Delegate Invocation
static void Main(string[] args)
{
    Console.WriteLine("Application started....");
    Console.WriteLine("Starting thread....");

    Func<int, string> workerMethod = PiCalculator.Calculate;
    IAsyncResult asyncResult = workerMethod.BeginInvoke(500, null, null);

    // Display periods as progress bar.
    while (!asyncResult.AsyncWaitHandle.WaitOne(100, false))
    { Console.Write('.'); }
    Console.WriteLine();

    Console.WriteLine("Thread ending....");
    Console.WriteLine(workerMethod.EndInvoke(asyncResult));
    Console.WriteLine("Application shutting down....");
}
```

# Event-Based Asynchronous Pattern

- The Event-based Asynchronous Pattern (EAP) is used to implement long-running methods.
- A simple EAP pattern involves appending Async to the method name and avoiding any outgoing parameters or return values.
- See Listing 19.20: Event-Based Asynchronous Pattern.

# EAP Example

// Event-Based Asynchronous Pattern

```
public event EventHandler<CalculateCompletedEventArgs>
```

```
    CalculateCompleted = delegate { };
```

```
public class CalculateCompletedEventArgs: AsyncCompletedEventArgs
```

```
{
```

```
    public CalculateCompletedEventArgs(
```

```
        string value, Exception error, bool cancelled, object userState)
```

```
        : base(error, cancelled, userState)
```

```
    { Result = value; }
```

```
    public string Result { get; private set; }
```

```
}
```

# BackgroundWorker Pattern

- The BackgroundWorker class executes an operation on a separate thread.
- The BackgroundWorker can be used to execute a time-consuming operation in the background.
- Events are provided that report the progress of the operation and signal completion.

# BackgroundWorker Example

```
// Using the Background Worker Pattern
public static BackgroundWorker calculationWorker = new BackgroundWorker();

// C# 2.0 Syntax for registering delegates: register long-running method
calculationWorker.DoWork += CalculatePi;

// Register the ProgressChanged callback: receive progress notifications
calculationWorker.ProgressChanged += UpdateDisplayWithMoreDigits;
calculationWorker.WorkerReportsProgress = true;

// Register a callback for when the calculation completes
calculationWorker.RunWorkerCompleted +=
    new RunWorkerCompletedEventHandler(Complete);
calculationWorker.WorkerSupportsCancellation = true;

// Begin calculating pi for up to digitCount digits
calculationWorker.RunWorkerAsync(digitCount);
```

# BackgroundWorker Events

- To set up a time-consuming operation to run in the background, simply add an event handler for the DoWork event.
- Call RunWorkerAsync() to start the method.
- Register for the ProgressChanged event to receive notifications of progress updates.
- Register for the RunWorkerCompleted event to receive a notification when the operation is completed.



# Cancelling BackgroundWorker

- Set the WorkerSupportsCancellation property to **true** to support cancellation.
- When this property is set to **true**, a call to the CancelAsync() method will set the DoWorkEventArgs.CancellationPending flag.
- Check the CancellationPending property in the DoWork method; set Cancel to true to exit.

# Windows UI Programming

- The Microsoft Windows operating systems use a single-threaded user interface thus only one thread can actually access the user interface.
- Windows forms provide a means for checking whether user-interface invocation is required to update the user interface.
- If the `InvokeRequired` method returns true, a call to `Invoke()` must be made to update the UI.

# Calling Invoke() to Update the UI

```
// Accessing the User Interface via Invoke()
private void Increment()
{
    for (int i = 0; i < 100; i++)
    {
        UpdateProgressBar();
        Thread.Sleep(100);
    }
    if (InvokeRequired)
    {
        // Close cannot be called directly from a non-UI thread.
        Invoke(new MethodInvoker(Close));
    }
    else
    {
        Close();
    }
}
```

# Calling Invoke() (cont'd)

```
// Accessing the User Interface via Invoke()
void UpdateProgressBar()
{
    if (_ProgressBar.InvokeRequired)
    {
        MethodInvoker updateProgressBar = UpdateProgressBar;
        _ProgressBar.BeginInvoke(updateProgressBar);
    }
    else
    {
        _ProgressBar.Increment(1);
    }
}
```

# Windows Presentation Foundation

- WPF provides a static member property called `Current` (type `DispatcherObject`) on the `System.Windows.Application` class.
- Calling `CheckAccess()` on `Current` (the dispatcher) serves the same function as `InvokeRequired` on controls in Windows forms.
- See Listing 19.24: Safely Invoking User Interface Objects.

# Unit 19 Summary

- The Unit began with a look at various synchronization mechanisms and at the classes used to protect against race conditions.
- Several multithreading patterns were also presented:
  - Asynchronous Programming Model;
  - Event-Based Asynchronous Pattern;
  - Background Worker Pattern.

# Unit 20

## Platform Interoperability and Unsafe Code

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 20 Overview

- C# allows the developer to step into the world of memory addresses and pointers in three ways: (1) Platform Invoke (P/Invoke); (2) unsafe code; and (3) COM interoperability (not covered in this book).
- This Unit finishes with a small program that determines whether the computer is a virtual computer.



# What is a Virtual Computer

- A **virtual computer** (also virtual machine (VM) or guest computer) is a virtualized or emulated computer that runs on top of the host operating system and interacts with the host computer's hardware.
- Once installed, a VM can be configured as if it were a real computer.
- Examples are VMWare Workstation and Microsoft Virtual PC.

# Platform Invoke (P/Invoke)

- The CLR provides the capability to call unmanaged code through P/Invoke.
- With P/Invoke, API (Application Programming Interface) calls can be made to the exported functions of unmanaged Dynamic Link Libraries (DLL).
- Unmanaged code may perform faster by avoiding the overhead of type checking and garbage collection.

# Declaring External Functions

- Once the target function is identified, it must be declared in the context of a managed class using the **extern** modifier.
- Extern methods are always static.

```
// Declaring an External Method
class VirtualMemoryManager
{
    [DllImport("kernel32.dll", EntryPoint = "GetCurrentProcess")]
    internal static extern IntPtr GetCurrentProcessHandle();
}
```

# Parameter Data Types

- The most difficult step is creating the managed data types that correspond to the unmanaged types in the external function.
- The size of memory pointers varies depending on the processor; C# uses the **System.IntPtr** type to handle external pointers.

# Example of IntPtr

```
// The VirtualAllocEx() API
LPVOID VirtualAllocEx(
    HANDLE hProcess, // The handle to a process. The
    // function allocates memory within
    // the virtual address space of this
    // process.
    LPVOID lpAddress, // The pointer that specifies a
    // desired starting address for the
    // region of pages that you want to
    // allocate. If lpAddress is NULL,
    // the function determines where to
    // allocate the region.
    SIZE_T dwSize, // The size of the region of memory to
    // allocate, in bytes. If lpAddress
    // is NULL, the function rounds dwSize
    // up to the next page boundary.
    DWORD flAllocationType, // The type of memory allocation.
    DWORD flProtect); // The type of memory allocation.
```

# Example of IntPtr (cont'd)

```
// Declaring the VirtualAllocEx() API in C#
class VirtualMemoryManager
{
    [DllImport("kernel32.dll", EntryPoint = "GetCurrentProcess")]
    internal static extern IntPtr GetCurrentProcessHandle();

    [DllImport("kernel32.dll", SetLastError = true)]
    private static extern IntPtr VirtualAllocEx(
        IntPtr hProcess,
        IntPtr lpAddress,
        IntPtr dwSize,
        AllocationType flAllocationType,
        uint flProtect);
}
```

# Using ref Rather than Pointers

- When unmanaged code uses pointers for pass-by-reference parameters, **ref** or **out** parameters may be used in C#.

```
// Using ref and out Rather Than Pointers
class VirtualMemoryManager
{
    static extern bool VirtualProtectEx(
        IntPtr hProcess, IntPtr lpAddress,
        IntPtr dwSize, uint flNewProtect,
        ref uint lpflOldProtect);
    // ... //
}
```

# Using StructLayoutAttribute

- Some API functions use unmanaged types that have no corresponding managed type.
- Calling these functions requires redeclaration of the type in managed code.



# StructLayoutAttribute Example

```
// Declaring Types from Unmanaged Structs
[StructLayout(LayoutKind.Sequential)]
struct ColorRef
{
    public byte Red;
    public byte Green;
    public byte Blue;

    public ColorRef(byte red, byte green, byte blue)
    {
        Blue = blue;
        Green = green;
        Red = red;
        Unused = 0;
    }
}
```

# API Error Handling

- Error Handling in the Win32 API is inconsistently handled.
- P/Invoke provides a mechanism for handling API errors in .NET that provides for standardized error checking.
- Setting the SetLastError named parameter of the DllImport attribute to true allows for the instantiation of a Win32Exception initialized with Win32 error data after a P/Invoke call.

# Win32 Error Handling Example

```
// Win32 Error Handling
class VirtualMemoryManager
{
    [DllImport("kernel32.dll", "", SetLastError = true)]
    private static extern IntPtr VirtualAllocEx(
        IntPtr hProcess,
        IntPtr lpAddress,
        IntPtr dwSize,
        AllocationType flAllocationType,
        uint flProtect);

    // ... //
    if (codeBytesPtr == IntPtr.Zero)
    {
        throw new System.ComponentModel.Win32Exception();
    }
    // ... //
```

# Using SafeHandle

- Resources accessed by P/Invoke may require cleanup after use, e.g., `VirtualAllocEx()` and `VirtualProtectEx()` require a follow-up call to `VirtualFreeEx`.
- The `SafeHandle` class (`System.Runtime.InteropServices`) provides the abstract members `IsInvalid()` and `ReleaseHandle()` for this purpose.
- See Listing 20.7 for an implemented example.

# Calling External Functions

- After declaring P/Invoke functions, they can be invoked just like any class member.
- The imported DLL must be in the path of the executable directory, however.
- A good approach is to provide a simplified managed API that wraps the underlying Win32 API.
- Note that function pointers in unmanaged code map to delegates in managed code.

# Guidelines for P/Invoke

- Always check first whether any managed classes already expose the API call.
- Define API external methods as private or internal.
- Provide public wrapper methods around external methods handling data type conversions and error handling.
- Overload wrapper methods and insert default values for calls with fewer parameters.

# Guidelines for P/Invoke (cont'd)

- User `enum` or `const` to provide constant values for the API as part of the declaration.
- For all P/Invoke methods that support `GetLastError()`, assign the `SetLastError` named attribute to `true`, allowing the reporting of errors via the `Win32Exception` class.
- Wrap resources such as handles into classes deriving from `SafeHandle` or implementing `IDisposable`.

# Guidelines for P/Invoke (cont'd)

- Function pointers in unmanaged code map to delegate instances in managed code.
- Map input/output and output parameters to **ref** parameters instead of relying on pointers.



# Unsafe Code

- Unsafe code allows access to memory addresses directly via C++-style pointers.
- The `unsafe` modifier is a directive to the compiler that a block of code, type, or members within a type may contain unsafe constructs such as pointers.
- The compiler must be notified when unsafe code is being used because of the possibility of memory corruption or holes in security.

# Unsafe Code Example

// Designating a Method for Unsafe Code

```
unsafe static void Main(string[] args)
```

```
{
```

```
    // ... //
```

```
}
```

// Designating a Code Block for Unsafe Code

```
static void Main(string[] args)
```

```
{
```

```
    unsafe
```

```
    {
```

```
        // ... //
```

```
    }
```

```
}
```

# Pointer Declarations

- An example of a typical pointer declaration is the following:

```
byte* pData;
```

- The pointer pData is either **null**, or its value points to a location in memory containing one or more sequential bytes.
- The type indicated (before the \*) is called the **referent type**; in this example pData is the pointer and **byte** is the referent type.

# Rules for Pointer Types

- Valid pointer types include enums, predefined value types (such as bytes, signed and unsigned integer types, floating point types, decimal, char, and bool) and pointer-to-pointer types (byte\*\*).
- Valid pointer types also include the void\* type, which is a pointer to an unknown type.
- Pointers cannot reference any other managed types.

# Assigning Pointers

- The value stored by a pointer is the address of a location.
- The address operator (&) is often used to retrieve the address of a type.
- To assign the address of some data to a pointer requires the following:
  1. The data must be classified as a variable.
  2. The data must be an unmanaged type.
  3. The variable must be fixed.

# Fixing Data

- Fixing data is referred to as pinning the data.
- Within the code block of a **fixed** statement, assigned data will not move, remaining at the same memory address throughout the block.
- The garbage collector will not compact fixed objects, so fragmentation may be an issue.

```
// Fixed Statement  
byte[] bytes = new byte[24];  
fixed (byte* pData = &bytes[0]) // pData = bytes also allowed  
{ /* ... */ }
```

# Allocating on the Stack

- Allocating an array on the stack is an alternative to fixing an array on the heap, but stack space is limited.
- Stack allocated data is not subject to garbage collection or finalization.

```
// Allocating Data on the Call Stack  
byte* bytes = stackalloc byte[42];}
```

# Dereferencing Pointers

- Dereferencing a pointer is used to retrieve or alter the data at the memory location referred to by the pointer, e.g.,  $\rightarrow$  `data = *pdata;`
- The `void*` pointer may not be dereferenced because it represents a pointer to an unknown type; it must first be converted to a known pointer type and then dereferenced.
- The increment, decrement, and comparison operators can all be used with pointers.



# Modifying an Immutable String

```
// Modifying an Immutable String
string text = "S5280ft";
Console.WriteLine("{0} = ", text);

unsafe // Requires /unsafe switch.
{
    fixed (char* pText = text)
    {
        char* p = pText;
        *++p = 'm';
        *++p = 'i';
        *++p = 'l';
        *++p = 'e';
        *++p = ' ';
        *++p = ' ';
    }
}
```

# Using the Index Operator

```
// Modifying an Immutable with the Index Operator in Unsafe Code
unsafe // Requires /unsafe switch
{
    fixed (char* pText = text)
    {
        pText[1] = 'm';
        pText[2] = 'i';
        pText[3] = 'l';
        pText[4] = 'e';
        pText[5] = ' ';
        pText[6] = ' ';
    }
}
```

# Accessing Members of a Type

- A referent type's members can be accessed using the `->` operator; thus `a->b` is shorthand for `(*a).b`

```
// Directly Accessing a Referent Type's Members
unsafe
{
    Angle angle = new Angle(30, 18, 0);
    Angle* pAngle = &angle;
    System.Console.WriteLine("{0}° {1}' {2}\"",
        pAngle->Hours, pAngle->Minutes, pAngle->Seconds);
}
```

# Unit 20 Summary

- This Unit discussed how to use low-level access to the operating system in C#.
- Low-level access can sometimes improve performance and flexibility of a program.

# Unit 21

## Common Language Infrastructure

C# Data Structures and Design Patterns  
Spring Quarter, 2017

# Unit 21 Overview

- This last Unit investigates the Common Language Infrastructure (CLI) which C# relies on at compile time and execution time.
- The execution engine that controls a C# program at runtime is also discussed.
- Finally, the position C# occupies in the broader set of .NET languages is explored.

# Common Language Infrastructure

- The specification for the Common Intermediate Language (CIL) and the Common Language Runtime (CLR) are contained in the Common Language Infrastructure (CLI), an international standard.
- This specification describes the context in which a C# program executes and how it must interact with other .NET languages.

# CLI Concepts

- C# generates instructions in an intermediate language called Common Intermediate Language (CIL).
- A second compilation step converts the CIL to machine code (binary).
- The Virtual Execution System (VES), more commonly referred to as the Common Language Runtime (CLR), is responsible for managing the execution of a C# program.



# CLI Standard

- Virtual Execution System (VES, or runtime)
- Common Intermediate Language (CIL)
- Common Type System (CTS)
- Common Language Specification (CLS)
- Metadata
- .NET Framework

# Primary C# Compilers

C# Compiler	Description
Microsoft Visual C# .NET Compiler	Microsoft's .NET C# compiler is dominant in the industry, but it is limited to running on the Windows family of operating systems.
Microsoft Silverlight	This is a cross-platform implementation of the CLI that runs on both the Windows family of operating systems and the Macintosh.
Microsoft Compact Framework	This is a trimmed-down implementation of the .NET Framework designed to run on PDAs and phones.
Microsoft XNA	This is a CLI implementation for game developers targeting Xbox and Windows Vista.

# Primary C# Compilers (cont'd)

C# Compiler	Description
Mono Project	The Mono Project is an open source implementation sponsored by Ximian and designed to provide a Windows-, Linux-, and Unix-compatible version of the CLI specification and C# compiler. Source code and binaries are available at <a href="http://www.go-mono.com">www.go-mono.com</a> .
DotGNU	This is focused on creating platform-portable applications that will run under both the .NET and the DotGNU. Portable.NET implementations of the CLI. This implementation is available from <a href="http://www.dotgnu.org">www.dotgnu.org</a> .
Rotor	The Rotor program, also known as the Shared Source CLI, is an implementation of the CLI that Microsoft developed to run on Windows, Mac OS X, and FreeBSD. Both the implementation and the source code are available free at <a href="http://msdn.microsoft.com/en-us/library/ms973880.aspx">http://msdn.microsoft.com/en-us/library/ms973880.aspx</a> .

# C# Compilation

- C# compilation requires two steps:
  1. Conversion from C# to CIL by the C# compiler
  2. Conversion from CIL to instructions that the processor can execute
- The just-in-time (JIT) compiler performs the translation from CIL to machine code.
- .NET provides a tool called **ngen** that enables compilation to machine code before execution, reducing startup time.

# Common Language Runtime (CLR)

- The Common Language Runtime (CLR) is Microsoft's implementation of the Virtual Execution System.
- The CLR converts CIL code to machine code and runs it in a **managed** environment.
- The CLR also provides various services to the managed environment: garbage collection, exception handling, code access security, etc.

# Garbage Collection in .NET

- Garbage collection refers to the automatic management of memory on the heap.
- Garbage collection uses a generational, mark-and-sweep algorithm to manage memory.
- During a garbage collection, objects that are to be retrieved are marked and then the now unused memory they occupied is compacted.
- The garbage collector is optimized, retrieving memory only when needed.

# Type Safety

- The CLR also enforces type checking at runtime.
- Type compatibility is checked when assigning data to a variable, for example.

# Code Access Security (CAS)

- The CLR makes security checks when a program executes, allowing and disallowing specific types of operations.
- The runtime grants permissions based on who created the program and if they are trusted.
- CAS also applies security policy based on the location of the code, e.g., code on the local machine is trusted more than code on the Internet.



# Platform Portability

- Portability means the capability of running on multiple operating systems and executing under different CLI implementations.
- .NET applications are designed to be portable across different operating environments, but problems exist because of variations in the Base Class Libraries (BCL).

# Performance Tradeoffs

- Managed environments impose some overhead on applications.
- The tradeoff is one of decreased development time and reduced bugs versus faster runtime performance.
- The CLR does make some adjustments, however; code efficiencies generated by the just-in-time compiler rival those of programs compiled directly to machine code.

# Application Domains

- Application domains (**app domains**) are virtual processes in which .NET applications run.
- An app domain offers isolation from other app domains, just like a process would.
- Static data is not shared between app domains.
- Each app domain has its own threads, and threads cannot cross app domain boundaries.
- Multiple app domains can run in a single process.

# Assemblies, Manifests, Modules

- An **assembly** is the output from the C# compiler, containing the CIL and a **manifest**.
- The manifest is made up of
  - The types that an assembly defines and imports,
  - Version information about the assembly,
  - Additional files the assembly depends upon,
  - Security permissions for the assembly.
- Assemblies can be class libraries or executables.
- Assemblies can be broken in separate modules.

# More on Assemblies

- Assemblies form the smallest deployment unit that can be versioned and installed.
- Most assemblies contain only one module.
- Microsoft provides a utility called **ILMerge.exe** for combining multiple modules into a single-file assembly.
- The manifest includes a reference to all the files required by the assembly, including modules.

# Common Intermediate Language

- CIL is the intermediate language for C# and VB.NET, and a host of other languages.
- Languages that compile to CIL are called source languages.
- CIL enables programming-language interoperability as well as platform portability.

# Common Type System (CTS)

- The Common Type System (CTS) defines how types are structured and laid out in memory, as well as various concepts and behaviors.
- The CTS breaks types into two categories:
  - Values, such as integers and dates, which are stored in underlying structures (**struct**), but have a separate type designation;
  - Objects, which do contain identifying type designation.

# Common Language Specification

- The Common Language Specification (CLS) provides standards for writing libraries that are accessible from all .NET source languages.
- Some data types like the unsigned integer, are not CLS-compliant, meaning that they should not be exposed externally to other languages.



# Base Class Library (BCL)

- The Base Class Library (BCL) is a core set of class libraries that programs may use.
- These libraries provide basic foundational types and APIs, allowing interaction with the CLR and underlying operating system.
- The Microsoft-specific Framework Class Library (FCL) adds to the BCL the support for rich client and web interfaces, database access, distributed communication, etc.

# Metadata

- Metadata includes information about the types and files included in an assembly:
  - Descriptions of each type within a program or class library;
  - The manifest information containing data about the program itself, along with the libraries it depends on;
  - Custom attributes embedded in the code, providing additional information about the constructs the attributes decorate.

# More on Metadata

- Metadata is used by the runtime to determine what dependent libraries to load, to check versioning information, and to perform security checks.
- Reflection is a mechanism available at runtime to retrieve type information and metadata from a loaded assembly.

# Unit 21 Summary

- This Unit described terms and acronyms important to the understanding of the context in which C# runs.