

Foundations

Epic 2016

C Sharp Self Study

Epic Systems Corporation
1979 Milky Way • Verona, WI 53593 • Voice: (608) 271-9000 • Fax: (608) 271-7237
www.epic.com
documentation@epic.com
Last Revision: October 20, 2016

C Sharp Self Study

Writing Basic Programs	1•3
Structuring Behavior and Data	2•3
Promoting Flexible and Reusable Code.....	3•3
Working with Collections	4•4
Working with Strings	5•3
Handling Errors	6•3
Disposing of Structures.....	7•3
Handling Events.....	8•3
Storing and Retrieving Data.....	9•3
Communicating with the Database.....	10•3
Appendix A: Name Conventions Cheat Sheet.....	A•3

Lesson 1: Writing Basic Programs

Introduction.....	1•3
By the end of this lesson, you will be able to.....	1•3
Structuring Code	1•5
Statements.....	1•5
General Structure	1•6
Example: Comments	1•7
Header Documentation.....	1•7
Predefined XML Tags.....	1•7
Example: Header documentation	1•8
Defining collapsible regions.....	1•10
Creating Text-Based Applications.....	1•12
Displaying Output.....	1•12
Kinds of Strings to Display.....	1•12
Collecting Input.....	1•14
Representing Data.....	1•17
Where Values are Stored	1•18
Example: Data Types.....	1•19
Converting between Value and Reference Types.....	1•20
Example: Boxing.....	1•20
Integral Types.....	1•21
Floating-Point Types.....	1•23
Casting.....	1•25
Other Types.....	1•26
char.....	1•27
bool	1•28
string	1•28
void.....	1•28
Defining Variables and Constants	1•30
Identifiers.....	1•30
Constants.....	1•30
Local Variables.....	1•31
Activity: Create a Simple Console Application	1•33
Part 1: Start a New Project.....	1•33
Part 2: Create a simple text interface.....	1•35
If you have time	1•36

Wrap Up.....	1•37
Defining Sets of Constants.....	1•39
Allowing Empty Value Types	1•43
Syntax.....	1•43
Casting Nullable types.....	1•44
Using Operators.....	1•46
Math Operators	1•46
Assignment	1•46
Increment and Decrement Operators.....	1•47
Relational Operators.....	1•47
Logical Operators.....	1•48
Full Order of Operations	1•49
Branching	1•51
Unconditional Branching.....	1•51
Reusing Method Names.....	1•52
Passing Data with Parameters	1•54
Conditional Branching.....	1•56
The If/Else Statements.....	1•56
The Switch Statement.....	1•57
Looping.....	1•61
The While Loop.....	1•61
The Do-While Loop.....	1•63
The For Loop.....	1•65
Activity: Validating User Input	1•69
If You Have Time.....	1•73
Exercise Solution	1•73

Writing Basic Programs

Introduction

The goal of this lesson is to lay a solid foundation for programming in the C# language. This will allow you to:

- Quickly create basic programs in C#
- Read and understand segments of C# code
- Learn the later lessons more efficiently



Although some parts of this lesson may seem basic to an experienced programmer, these concepts are critical for your success during later lessons.

By the end of this lesson, you will be able to

- Concepts (things to understand):
 - Explain why Epic has strictly enforced conventions for identifiers in C#
 - Describe the difference between a field and a constant
 - Describe the difference between a value type and a nullable value type and when you would use one over the other
 - Explain why you would divide code into collapsible regions
 - Describe the difference between a strongly-typed programming language like C# vs. a loosely-typed programming language like M
 - Explain why you would use one data type vs. another
 - Explain why you would use an enumeration
 - Explain the difference between an input variable, a variable passed using the out keyword, and a variable passed using the ref keyword
 - Explain the difference between value and reference types
 - Explain why exceptions can occur during casting
 - Explain how and why you might reuse a method name
 - Explain how namespaces are used at Epic
- Algorithms (things to be able to do):

- Write statements
- Group statements into blocks
- Write comments in various forms
- Read from and write to the Console
- Use operators to form expressions
- Cast between data types
- Define methods
- Identify which method is called given several overloaded methods
- Branch conditionally
- Create looping code

Structuring Code

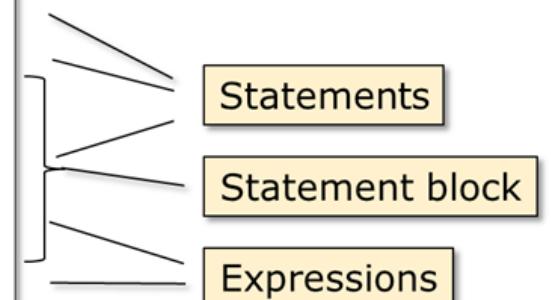
Like other programming languages, there are some basic rules you need to follow when writing valid code in C#.

Statements

There are three general forms of code in C#:

Statement	A line of code in C#. Every statement must end with a semicolon ;
Statement block	A group of statements enclosed in braces: {}. Can be used in place of statements. Statement blocks usually have headers that indicate when that block of code will run.
Expressions	Statements that evaluate to a value. Can sometimes be used in place of statements.

```
_patString = "";
int count = RecordList.Count;
for (int i = 0; i < count)
{
    _patString += RecordList[i].Id + "_";
    i++;
}
GenerateSingleEPT();
```





Try It Yourself: Code Structure

1. Start Visual Studio
2. Create a new project for in-class examples
 - FILE > New Project > Visual C# > Console Application
 - Name: ClassExamples
 - Location: (Use the default location)
3. Notice the code that was created:

```
using System;
using System.Collections.Generic;
using System.Linq;using System.Text;
using System.Threading.Tasks;
namespace ClassExamples
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

4. Based on this example, can code blocks be nested?

▪ _____

5. Is a semicolon required when ending a statement block?

▪ _____

General Structure

The general code structure of C# varies from other languages like M and VB you may already be familiar with:

- White space (e.g., spaces, tabs and blank lines) is allowed. Add white space to make code readable.
- There is no line continuation character. A statement ends when a semicolon is reached.

- There are multiple ways to include comments:
 - Single line; denoted by // (two forward slashes)
 - Multiple lines; enclosed within /* */

Example: Comments

```
/* This is a block comment  
It can span multiple lines */  
  
// End of line comment
```



Try It Yourself: Whitespace and Comments

1. Try adding white space and blank lines in various places. Determine where whitespace is allowed and where it isn't.
2. Try adding several single-line comments using //
3. Try adding several multi-line comments using /* ... */
4. Use the mouse or **SHIFT + UP/DOWN** to select multiple lines of code at once
5. Press **CTRL+K** and then **CTRL+C**. What happened?

6. Press **CTRL+K** and then **CTRL+U**. What happened?

Header Documentation

C# has header-documentation built directly into the language in the form of XML comments, so, there isn't a standard Epic header template.

- XML documentation allows for ease of documentation of code.
- XML comments start with ///
- They can be applied to types or members.
- You may also embed XML tags which are then extracted to build HTML documentation that can be shared.

Predefined XML Tags

Here are a few of the most commonly used XML documentation tags.

summary	<p>Place a general description of what the purpose of the code you are documenting is.</p> <pre><summary>Place your description here</summary></pre>
remarks	<p>Use to add information, supplementing the information specified with summary.</p>
param	<p>Documents the purpose of a parameter. Like other languages, parameters can be input, output or both, so be sure to include that in the description. An attribute of the param element is name, which should match the name of the parameter you are documenting:</p> <pre><param name="nameOfParameter">Description</param></pre>
returns	<p>Describe the data that this code returns, if it returns anything.</p> <pre><returns>description</returns></pre>
exception	<p>Exceptions are thrown if code encounters an unexpected error and cannot continue as a result. Use this to document the kinds of exceptions that could be thrown, if they are known ahead of time.</p> <pre><exception>description</exception></pre>
example	<p>Use this to provide examples of how the code could/should be used.</p> <pre><example>code</example></pre>
c	<p>Use to indicate that some part of another element is code.</p> <pre><example>To call this method, use <c>class.method()</c> </example></pre>



For more XML comment elements, see:

[https://msdn.microsoft.com/en-us/library/b2s063f7\(v=vs.80\).aspx](https://msdn.microsoft.com/en-us/library/b2s063f7(v=vs.80).aspx)

Example: Header documentation

```
/// <summary>
/// Calculate the factorial of a number
/// </summary>
/// <param name="number">
/// The number to calculate the factorial of
/// </param>
/// <returns>The factorial of the number</returns>
static int Factorial(int number)
{
    int answer = 1;

    for (int i = 2; i <= number; i++)
    {
        answer *= i;
    }
    return answer;
}
```



Try it Yourself: Header Documentation

1. Directly above the line block beginning with `static void Main(string args [])`, insert a new line.
2. On the new line, type three slashes: `///`. What happened?

■

3. Directly above the line marked `class Program`, insert a new line.
4. On the new line type three slashes: `///`. How is this result different?

■

5. Try adding three slashes above the namespace and using statements. What happens?

■

6. Remove the `///` comments from above the namespace and using statements.
7. Fill out the header documentation with whatever you want.
8. Within the code block below `Main`, type `Main()` and then wait. What do you notice regarding the header comments?

■

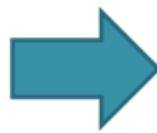
Defining collapsible regions

- Code that collapses to a heading
- Useful for outlining code:

```
static int Factorial(int number)
{
```

```
#region define local variables  
  
int answer = 1;  
#endregion  
  
#region calculate factorial  
for (int i = 2; i <= number; i++)  
{  
    answer *= i;  
}  
#endregion  
  
return answer;  
}
```

```
static int Factorial(int number)  
{  
    #region define local variables  
    int answer = 1;  
    #endregion  
  
    #region calculate factorial  
    for (int i = 2; i <= number; i++)  
    {  
        answer *= i;  
    }  
    #endregion  
  
    return answer;  
}
```



```
static int Factorial(int number)  
{  
    #region define local variables  
    #endregion  
    #region calculate factorial  
    #endregion  
    return answer;  
}
```



Try It Yourself: Regions

1. Try including several regions in your CodeExample project.
2. Try nesting one region inside another one

Creating Text-Based Applications

Throughout this course you'll create text-based programs, which use the console to read and write text, so it's important to be familiar with some of the most common Console methods. In order to use any Console method, you need to have access to the `System` namespace, which includes the `Console`'s definition.



A **method** is a function or subroutine that is also a member of a class. In C# all functions and subroutines are part of a class, so we will refer to all callable code as methods.

Namespace	A way to logically divide code into different modules. By only including namespaces that you actually use, it will be easier to find the code you are looking for. This will make programming faster and less error prone.
using System;	System is the namespace that includes many of the most commonly used framework features from .NET, including the Console.



When using Visual Studio, the `using System;` statement (as well as other namespaces) will be included automatically in every program.

Displaying Output

The console has several methods for writing output to text.

Console.Write	Writes a string to the Console, keeping the cursor on the same line as the string. <code>Console.Write("Enter name: ");</code>
Console.WriteLine	Write a string to the Console and move the cursor to the next line. <code>Console.WriteLine("Hello world!");</code>

Kinds of Strings to Display

A string written to the console can be a literal (enclosed in double quotes) or you can use placeholders inside of the string to dynamically substitute in values from variables. To include multiple place holders in the same string, increment the value of the place holder (enclosed in {}) by 1 for each additional placeholder. The variables are listed after the string in the order in which they should replace the place holders.

Literal	Enclosed in double quotes <code>Console.WriteLine("Hello world!");</code>
Constructed	Use placeholders that are replaced at runtime.

```
string greeting = "Hello";
string subject = "world";
Console.WriteLine("{0} {1}!", greeting, subject);
```



You can also construct strings that you store in variables using the `string.Format` method. For example, given these variables:

```
string greeting = "Hello";
string subject = "world";
```

these two statements are equivalent:

1. `Console.WriteLine("{0} {1}!", greeting, subject);`
2. `string message = string.Format("{0} {1}!", greeting, subject);
Console.WriteLine(message);`

Constructed strings are particularly useful with internationalization (I18N) because the translator can rearrange the placeholders during translation.



Try using code snippets!

Code snippets are pre-defined code-segment templates that you can use to make your time programming more efficient. Think of them as smart text for source code.

To use a code snippet, type the snippet, then press **TAB** twice. If there are fields, you can tab through the fields of the snippet to populate them. Try this snippet:

`cw` to quickly enter `Console.WriteLine()`

More snippets will be introduced as they become relevant.



Collecting Input

There are several ways to get input from the user through the console.

Read	Reads the next character from the input stream. char aChar = Console.Read();
.ReadKey	Reads the next keystroke. Use to determine exactly which key the user pressed, or to respond to any key being pressed. ConsoleKeyInfo key = Console.ReadKey(); One common use is to pause execution until the user enters anything.
.ReadLine	Reads the next line of characters (until a carriage return), resulting in a string value: string name = Console.ReadLine();

If the input is supposed to be numeric, you'll need a way to convert the string into a number. One way to perform this action is to access the Parse method from the desired numeric class. Later in this course we'll discuss alternate methods of converting a string into a number.

```
Console.Write("Please enter a string: ");
string text = Console.ReadLine();

Console.Write("Please enter an integer: ");
int number = int.Parse(Console.ReadLine());

Console.WriteLine("You entered '{0}' and {1}", text, number);
Console.ReadKey();
```



If the input into the Parse method cannot be successfully converted into the specified numeric type, C# will throw an **exception**.



For now, you can think of an **exception** as an error. If an exception is not handled by the surrounding code, the application will crash.



Try It Yourself: Console

1. In your ClassExamples project, within the code block that starts with `static void Main(string args[])`, add the following line of code:
 - `Console.WriteLine("Hello world!");`
2. Run the project by clicking **Start**, or pressing **F5**
3. What happened?
 - _____
4. Add this code below the call to `WriteLine`:
 - `Console.ReadKey();`
5. What happens this time?
 - _____
6. Add the prompt "Press any key to continue" immediately before the call to `Console.ReadKey`.
7. Experiment by prompting the user for information and storing it as a string or an int. Try passing a non-numeric string to the number prompts to see what an exception looks like.

Sample solution:

```
static void Main(string[] args)
{
    // Write a message
    Console.WriteLine("Hello world!");
    //
    // Read in a string
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    Console.WriteLine("You entered {0}.", input);
    //
    // Read in an integer
```

```
Console.Write("Enter a number: ");
int number = int.Parse(Console.ReadLine());
Console.WriteLine("You entered {0}.", number);
//
// Pause until a key is pressed
Console.WriteLine("<Press any key to continue>");
Console.ReadKey();
}
```



When you hit the play button in Visual Studio, how does Visual Studio know what code to run?

You have several files in your project -- you should be able to see App.config, Program.cs, and (if you expand Properties), AssemblyInfo.cs.

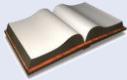
When determining what code to run, Visual Studio looks only at the files in your startup project - the bolded project. (Right now, you only have one project, but you'll have more later.) In those files, it looks for a method called `Main(string[] args)` and tries to run that method.

The code you just wrote is in the `Main` method, so that's what runs.

As a test, try changing the name of the `Main` method or creating another `Main` method. Your code will not compile.

Representing Data

To be able to program effectively in any programming language, it is critical to understand how that language represents data.



In M, there were only two data types, numbers and strings. In C# there are many more. There is string, int, bool, decimals, etc., and because programmers can make new types, the number of types that exist is an ever expanding frontier.

C# is a **strongly-typed language**. In M, you can pretty much store any piece of data in any variable. Not in C#. The data that you put into a variable has to match the type of that variable. If it doesn't match, your code won't compile. This is what it means for a language to be strongly-typed. Strongly-typed languages allow the compiler to check for potential data errors before the code is even run.

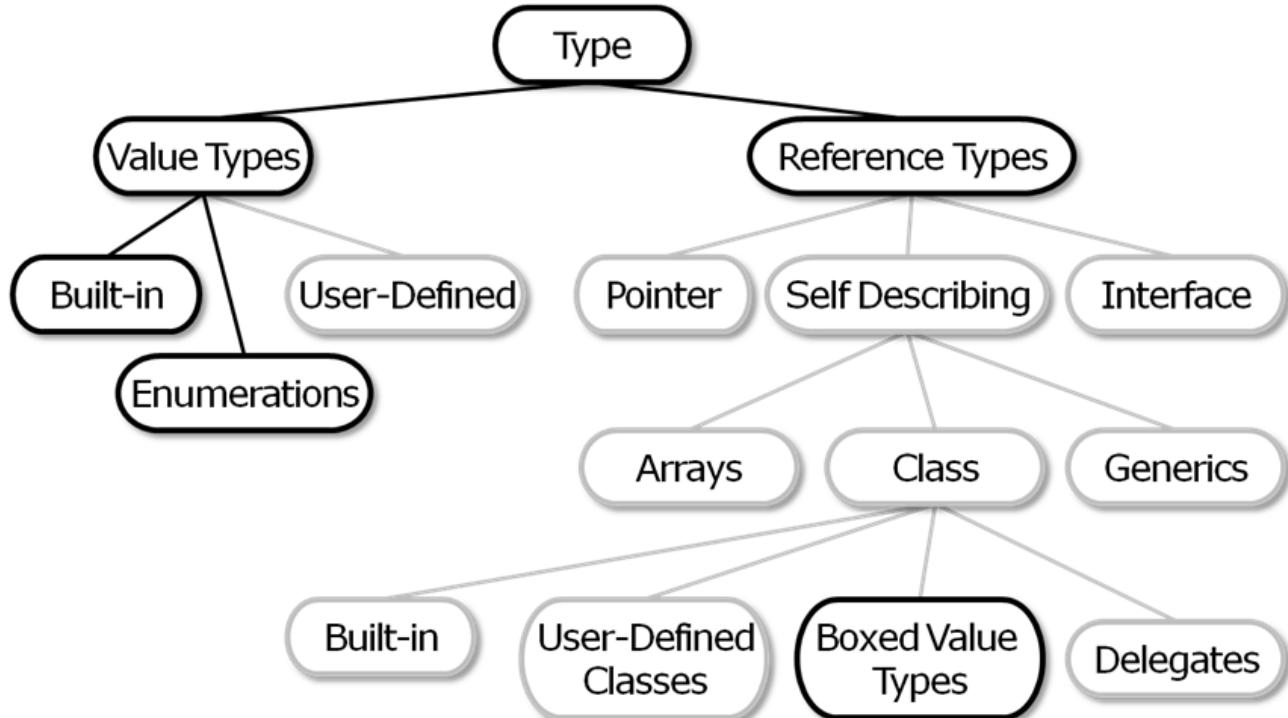


The main advantages of a strongly-typed language (one with many strict data types) are

- **Safety** - a strongly-typed language can prevent errors from occurring that a weakly-typed language couldn't prevent. By demanding that the types of certain variables be declared at compile-time, the compiler can be smarter about what code will work and what code will fail. It can then prevent failing code from ever being created in the first place.
- **Readability** - because variables have a defined type, the code tends to be more readable.
- **Ease of debugging** - data type bugs arising from unintended data type mutations can be difficult to track down. The situation gets particularly nasty when the code runs smoothly but produces unexpected results. Strongly-typed languages make most data type mutations obvious by demanding that the programmer make their intentions clear at compile-time.

C# data types can be broadly broken into two groups: Value Types and Reference Types. These differ based upon where the data is actually stored and how it is accessed. We will explore some data types now and some during later lessons.

Because C# is a strongly-typed language, when you declare a variable, you must declare the data type of that variable along with the name. The variable may then only contain values of or references to the given type.



Where Values are Stored

Value types and reference types are stored in different memory locations. The two locations are the stack and the heap.

Stack	<p>The stack is the location in memory that tracks what code is currently being executed, as well as any local variables that have been defined. The top level stack indicates the code that is currently being executed, while lower levels contain code that will be returned to once the higher levels are complete. The stack has the following properties:</p> <ul style="list-style-type: none"> • Quick access • Minimal overhead
Value Types	<p>The data for a value type variable is allocated in the stack level where it is defined. When the stack level containing the variable moves out of scope, the data stored in that variable is no longer available.</p>
Heap	<p>The heap is a separate memory location. It can be accessed by any reference type variable in any stack level. Objects in the heap are cleaned up periodically by the .NET framework's garbage collector. Objects are only be cleaned up by the garbage collector when there is no way to reach the object from the stack.</p>

Reference Types

When declaring a variable of a reference type, the variable on the stack does not hold the actual data. Instead, it holds a memory address that points other code to a memory location on the heap. This memory address is often called a **reference** or a **pointer**.

If a stack level that holds a reference type variable is removed, then the reference is also removed, but the *data* will remain on the heap until it is garbage collected (assuming there is no other path to that data from the stack).

Example: DataTypes

In this example, there are two value type variables and two reference type variables defined. The value types are firstInt and secondInt, while the reference types are spike and spot.

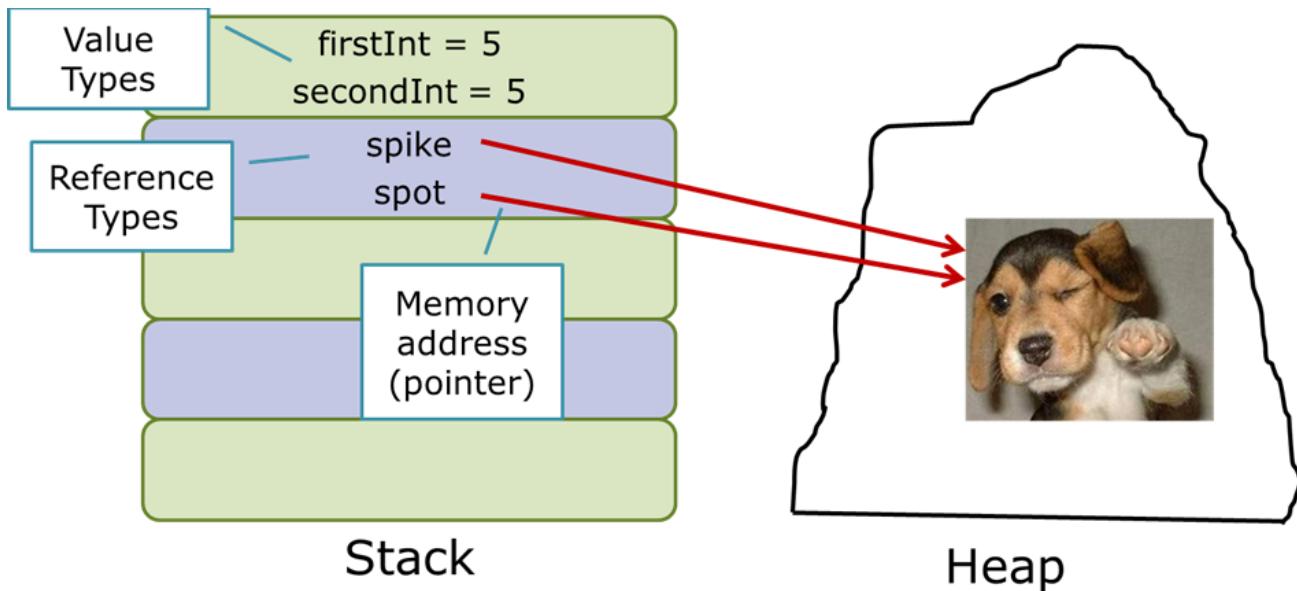
```
// Value types
int firstInt = 5;
int secondInt = firstInt;

// Reference types
Dog spike = new Dog();
Dog spot = spike;
```

When one variable is assigned the value of another, whatever is stored on the **stack** is copied from the source variable and assigned to the target variable. For value types, this means that the actual information that we are interested in (5 in this case) is duplicated and stored in two separate memory locations on the stack.

Because reference types store a heap memory address in the stack, it is the address that is copied from one variable to the next. Although the memory address is stored in two separate variables, the information that they reference on the heap is still the same.

This scenario is illustrated in the following diagram:



Converting between Value and Reference Types

From time to time, you may need to convert between reference and value types. This is particularly useful when using collections (lists, dictionaries, etc.) of mixed types. The term "boxing" refers to the process of converting a value type to a reference type, while "unboxing" is the reverse.

Boxing	Convert a value type to a reference type. This is an implicit conversion and could happen without any indication if a reference type is needed.
Unboxing	Convert a reference type to a value type. This needs to be done explicitly. Not all reference types can be cast into values.

Example: Boxing

```
int myIntegerVar = 12;
object myObjectVar = myIntegerVar;      // Boxing
int myOtherIntVar = (int) myObjectVar; // Unboxing
```

- The type **object** is the most basic built-in reference type and can be used to box up any value type.
- Object is a synonym for object from the Intermediate Language (IL). The lower-case version is preferred.



Try It Yourself: Type Conversions

1. Open the immediate window using **DEBUG > Windows > Immediate**.
 - To test code without making it part of your solution, you can use the immediate window. This is similar to the VB immediate window, or the M prompt.
2. Type the following:


```
int myIntegerVar = 12;
object myObjectVar = myIntegerVar;
int myOtherIntVar = (int) myObjectVar;
```
3. Next, type the following:


```
int myNextInt = myObjectVar;
```
4. What happened?
 -
5. Close the console that opened during step 2.

Integral Types

The integral data types available in C# are listed in the table below. The first column corresponds to C#'s name for each type, while the second column represents the built-in Intermediate Language (IL) data type to which the C# data type maps.



Having a built-in set of IL data types allows every language in the .NET Framework to give different names for each data type.

C# Type	IL Type	Size (bytes)	Signed?
sbyte	Sbyte	1	Yes
short	Int16	2	Yes
int	Int32	4	Yes
long	Int64	8	Yes
byte	Byte	1	No

C# Type	IL Type	Size (bytes)	Signed?
ushort	UInt16	2	No
uint	UInt32	4	No
ulong	Uint64	8	No

- The data type of **int** is the most common integral type, as it is the standard 32 bit value.
- The default value for all integral types is 0.
- The range of values that each type can hold depends on whether or not it is signed. Signed types can hold positive or negative values, ranging from **-2^(n-1) to 2^(n-1)-1**, where **n** is the number of bits in the type. Unsigned types can only hold positive values, ranging from **0 to (2^n)-1**, where **n** is the number of bits in the type.
- When representing literals in C#, apply a suffix to the literal value to use a specific type; otherwise the default int type will be used.

Suffix	Type
U	uint or ulong
L	long or ulong
UL	ulong

```
int intOne = 5; // Assumes int default
uint intTwo = 10U; // Unsigned int
```



Try It Yourself: Integral Types

1. Open the immediate window.
2. Clear it using **right click > Clear All**
3. Type the following:

```
int myInt = int.MaxValue;
```

4. What does this value represent?

5. How could you compute it?

6. Type the following several times in a row (you can use the up arrow instead of retyping it)

- `++myInt;`

7. What happened and why did it happen?

8. Fill out the following table using the immediate window:

Type	<u>Min Value</u>	<u>Max Value</u>
sbyte		
byte		
short		
ushort		
int		
uint		
long		
ulong		

Floating-Point Types

The floating-point types available in C# are listed in the table below. The C# Type, System Type, and Size columns are equivalent to the columns in the integral types table. The Range column shows how many significant figures each type can hold.

C# Type	IL Type	Size (bytes)	Range
float	Single	4	7 figures
double	Double	8	15 figures
decimal	Decimal	16	28 figures

- "float" and "double" carry plenty of precision and should handle most floating-point type needs. Only use "decimal" if absolutely necessary (for example, when representing money).
- The default value for all floating-point types is 0.0
- Much like with integral types, when specifying literals for floating-point types, apply a suffix to force the use of a particular data type. If no suffix is used, the value will default to a double.

Suffix	Type
F or f	float
D or d	double
M or m	decimal

```
double dubOne = 9.1; // Assumes double
```

```
float floatOne = 10.813F; // Float
```



Try It Yourself: Floating-Point Types

1. In the immediate window, examine the max and min values of float, double and decimal.
2. Record the result of each of the following:
 - double value1 = 5f;

□ _____

- double value2 = 5;

□ _____

- double val3 = 5m;

□ _____

□ _____

Casting

In C#, it is possible to convert between the different data types. There are two types of casting:

Implicit casting	<p>Conversion to another data type without any special syntax. Usually, this is used to convert from a data type that uses less memory to a data type that uses more (small to large).</p> <pre>int x = 123456; long y = x; // Implicit conversion</pre>
Explicit casting	<p>Conversion to another data type by explicitly declaring the cast in parentheses. Usually, this is required to convert from a data type that uses more memory to a data type that uses less (large to small).</p> <pre>int x = 123456; short z = (short) x; // Explicit casting</pre>

If during the cast you are simply adding more bytes to the value, all data is preserved, so an implicit cast is okay. If casting removes bytes that could be in use, data loss is possible, so an explicit cast is required.



If the data can't be cast correctly during an explicit cast, an **InvalidCastException** will be thrown at runtime. Issues with implicit casts will be caught at compile time.

```
int x = 123456;  
long y = x; // Implicit conversion  
short z = (short) x; // Explicit conversion
```



Try It Yourself: Conversions

1. Which of the following require explicit conversions?

- a. int to long

- b. sbyte to byte

- c. float to int

- d. int to float

- e. decimal to double

- f. double to decimal

2. Verify your answers in the immediate window.

Other Types

There are several other data types in C#, including:

C# Type	IL Type	Size (bytes)	Default Value
char	Char	2	'\0'

C# Type	IL Type	Size (bytes)	Default Value
bool	Boolean	1	false
string	String	20 minimum	null
void	void	N/A	N/A

char

Represents a single Unicode character.

The following kinds of literal characters should be enclosed in single quotes:

- Characters
- Unicode
- Escape characters

Character	'A'
Unicode format	'\u0041'
Escape character	'\n'

The escape characters recognized by C# are:

\'	Single quote
\"	Double quote
\\"	Backslash
\0	Null
\f	Form feed
\n	New line

\r	Carriage return
\t	Horizontal tab

The escape characters may be used individually or as part of a string.

- Individually: '\\\'
- In a string: "In M, the integer division operator is \\\"

bool

Can be **true** or **false**. Is **false** by default.

string

Strings are reference types in C#. They are composed of an immutable sequence of Unicode characters. String literals are enclosed in double quotes. Within the double quotes, you may include any of the escape characters listed above in addition to your regular characters. Optionally, you can prefix a string with the @ symbol, indicating that the string should be read as is (not evaluating escape characters).

```
// Equivalent strings
string normal = "\\\server\\fileshare";
string verbatim = @"\\server\fileshare";
```

void

A special type that is not a data type used for variables, but instead a return type for a method. If a method does not have a value to return, it will have a void return type. Methods are covered in the Branching section of this lesson.



Try It Yourself: What is the type?

1. What is the most appropriate data type for the following values?

- 10

- 10U

- 15.605

- 15F

- '\n'

- "This is not not a string"

- true or false

2. Verify each of your guesses using the immediate window. For example, if you guessed that 10 is a float, you can check that using:

```
10.GetType() == typeof(float);
```

If it is **false**, then guess again!

Defining Variables and Constants

Identifiers

Identifiers are used to name entities in the system, such as constants, variables and user defined data types. Identifiers in C# are whole words, starting with a letter or an underscore, and are case sensitive.

The type of data being identified will dictate the case style of the identifier. Following Microsoft rules, the two cases to consider are:

- Camel case: myButton
- Pascal case: MyButton



As more structures are introduced, you should ensure that you name them appropriately.
See Name Conventions Cheat Sheet in the C Sharp Training Companion for details.

Constants

A constant is a value that cannot change. When declaring a constant, the identifier needs to be in Pascal case, so every word starts with a capital letter.



Constants are static by default. We'll cover what this means in a later lesson.

Syntax:

```
const type identifier = value;
```

Example:

```
const float TaxRate = 5.5F;
```



Try It Yourself: Constants

Constants cannot be defined in the immediate window, because they must be part of a class, so we'll add a constant to your ClassExamples project.

1. Notice that in Program.cs there is a class named Program defined. Add a constant to that class:
 - `const string YouCannotChange = "Balrog";`
2. On the first line of code in Main, set a break point by clicking in the grey margins to the left of the line. You should see a red dot appear.
3. Run the solution.
4. Open the Immediate window.
 - Because you ran your solution and set a breakpoint prior to opening the immediate window, you now have access to the context of the code that is currently executing. This is a nifty debug trick!
5. Type in the constant name:
 - `YouCannotChange`
6. What happened?
 - _____
7. Try setting the constant to another value
 - `YouCannotChange="Gandalf";`
8. What happened?
 - _____
9. Clear the breakpoint that you set earlier by clicking on the red dot in the margin.

Local Variables

Local variables are variables defined in methods, either in the parameter list or inside the method. To define a local variable, you need to indicate the data type and variable name. Local variable names should be in Camel case, meaning the first letter in the word in the variable name is lower case, while the rest of the words begin with an uppercase letter. Optionally, an initial value assignment can be included on the same line as the declaration.

Syntax:

```
type varName [ = value];
```

Examples:

```
int patientAge;  
string patientName = "Sickly, Sam";
```



You must assign a value to local variables before using them, or your code will not compile.

You can initialize local variables to their defaults, but you need to do so explicitly:

```
int myVar = default(int); // This is equivalent to assigning 0
```



Although local variables do not have an initial state, other kinds of variables do. For example:

- Static fields
- Class instance fields
- Array elements

We will cover these kinds of variables in later lessons.



Try It Yourself: Local Variables

1. In ClassExamples > Program > Main, define a new local variable:
 - `string myString;`
 - `myString = myString + " abcd";`
2. Build the solution using **BUILD > Build Solution** (or by pressing F7). What happens?
 - _____
3. Initialize `myString` to an empty string
 - `string myString = "";`
4. Build the solution again. What happens this time?
 - _____
5. Use the immediate window and `default (<type>)` to discover the default value of each of the following types:
 - a. `string`

 - b. `double`

 - c. `bool`

 - d. `object`

Activity: Create a Simple Console Application

In this exercise you will learn how to write a new console-based program.

Part 1: Start a New Project

The first step to create a new console application is creating an executable project.

1. Start Visual Studio
2. Create a new project
 - File > New > Project
3. Console Application

4. Uncheck **Create directory for solution**
5. Name: SimZoo
6. Location: C:\EpicSource\CS
7. Click **OK**



Take a look at your solution in the Solution Explorer (sidebar).

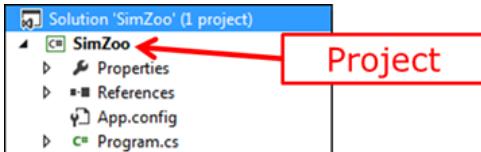
What is a solution? A **solution** is an organized way of viewing a group of projects. Usually, a solution will group projects that depend on each other.

A **project** is simply a collection of files. These files, when compiled with the C# compiler, will collectively produce a single portable executable (e.g. a file with a .dll or .exe extension) and perhaps some other files used by it. In the case of your program, it will produce a .exe file that can be run when double-clicked on.

1. Add a write statement to the console and a Console.ReadKey into your Main method.
2. Compile your project.
3. Try navigating to the Debug folder of your project:
 - a. Solution Explorer > SimZoo (project) > right click > Open folder in file explorer
 - b. Open bin > Debug
4. Double click SimZoo.exe.

Your program runs! This executable is the output produced when your project compiles. If you had multiple projects in your solution, multiple portable executables would be produced.

5. Edit the project properties:
 - a. Solution Explorer > SimZoo (project) > right click > Properties



Solution Explorer

6. Change the default namespace to "Epic.Training.SimZoo.Text"
 - Application Tab > Default namespace > Epic.Training.SimZoo.Text



If you use CodeSearch to search client code for "class Program" (the text at the top of your code), you'll find numerous C# files that also define Programs. Many of these may be included together in the same application (e.g. Hyperspace). If that happens, how does the application know which one to use?

The **namespace** of your project helps to keep the code in your project logically distinct from code written by other teams at Epic, and even other companies, such as Microsoft.

A class you write (like `Program`) might have the same name as other classes written by other people. But because namespaces are long and have many pieces, the chances of two classes sharing the same name and the same namespace are very small.

Namespaces are useful for another reason, too. By making namespaces descriptive and including many pieces, they can aid in organizing large collections of code.

At Epic, we use the following convention for namespaces to help organize and keep our code distinct:

Epic.<Owner>.<Application>.[Functional area].[Platform]

- Epic: This prefix separates our code from that of third parties
- Owner: The division owning the code.
 - Examples: Core (Foundations), Clinical, Billing, Access, etc.
- Application: The application within the division that this code belongs to.
 - Examples in Clinical: Ambulatory, Inpatient, OR, Orders, etc.
- Functional area (optional): A sub-division within the application.
 - Examples in Clinical > Orders: Administration, Data, Services
- Platform (optional): The medium on which the code will run.
 - Examples: Text, Web, WinForms, WPF

Notice that just by looking at the namespace, you can already tell the context in which a piece of code will be used.

7. What is each part of code for the project that you just created?
 - a. Owner: _____
 - b. Application: _____
 - c. Platform: _____
8. Save your changes and close the application properties.

Part 2: Create a simple text interface

1. Remove the "Naming Conventions Cheat Sheet" from the following section so that you can use it while writing code.

2. Edit Program.cs
3. Change the namespace from "SimZoo" to "Epic.Training.SimZoo.Text":

```
namespace Epic.Training.SimZoo.Text
```

4. Using the Console methods that you learned earlier in this lesson, display the message "<Press any key to continue>" and then pause execution until the user presses any key.
5. Define constants within the Program class to store your first and last name (as the author):

```
class Program
{
    const string AuthorFirstName = "<your name>";
    const string AuthorLastName = "<your name>";
    ...
}
```

6. Before displaying the "<Press any key to continue>" message, output the following:
 - "SimZoo Text v. 1.0"
 - "Author: {1}, {0}", where "{0}" is your first name and "{1}" is your last name.
7. Save and run the application to verify that it works as expected.
8. After the "<Press any key to continue>" message,
 - Clear the screen using the method Clear from the Console class.
9. Ask the user to enter a kind of animal to include in the zoo (as a string).
10. Ask the user how many of that kind of animal should live in the zoo (as an int).
11. Display the animal name and number of animals back to the user.
12. Pause the screen a second time.
13. Save and test your code.
14. Document your code.
 - Use single line comments to describe what is going on in main
 - Use XML comments to document the following:
 - The Program class
 - Each constant
 - The main method
15. Define a region called "class constants" around your constants

If you have time

Try using codesearch to find sample code from the new web framework.

1. <http://codesearch/> > Client Code
2. File name ends with: "cs"
3. Try the following search Strings:
 - a. namespace Epic.Core
 - b. namespace Epic.Billing
 - c. namespace Epic.Clinical



Keep in mind that you have not learned enough to fully understand everything that is going on.

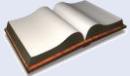
Wrap Up

1. What is the name convention for local variables?
 - _____
2. What is the name convention for constants?
 - _____
3. What is the purpose of a namespace?
 - _____
4. Number the following in the order that they appear in the namespace
 - Platform: _____
 - Epic: _____
 - Application: _____
 - Owner: _____
 - Functional Area: _____
5. Which parts of the namespace are optional?
 - a.

 - b.

Defining Sets of Constants

Enumerations are special types that only allow a finite and discrete set of constant values. Defining enumerations, rather than just using a range of integer values, makes the code easier to read because there is an identifier associated with each value. Additionally, it provides hints to the compiler as to what options are available for variables of the given type.



An enumeration is similar to a Chronicles category list.

Syntax

```
[modifiers] enum EnumTag  
{  
    enumerator-list  
}
```

Example

```
enum Size  
{  
    Small,      // Assumes a value of 0  
    Regular,    // Assumes a value of 1  
    Large = 5,  
    Gigantic,   // Assumes a value of 6  
}
```

Syntax to use an enumeration:

```
enumTag varname [ = enumTag.member];
```

Example

```
ExampleSizes shirtSize = Size.Gigantic;
```

An enumeration consists of a set of named constants. In C#, enumerations are classified as value types. An enumeration type declaration gives the name of what is called the "enumeration tag" and defines the set of named constants. Enumeration tags should be in Pascal case, and may be defined in a namespace or in a class. If an enum is defined in a class, the enum values will be accessed through that class (similar to constants).



Define an enum in a namespace if you want to make that enum available to many classes. If the enum is only useful for a particular class, define the enum inside that class.

The keyword `enum` is used when defining an enumerated type. Commas separate individual members of the enumeration list, and the members can be assigned a specific value, or a value will be automatically calculated by incrementing the previous value.

Here is another example, where each tag is assigned a specific value:

```
enum Temperature
{
    Cold=0,
    FreezingPoint = 32,
    LightJacketWeather = 60,
    SwimmingWeather = 80,
    BoilingPoint = 212,
}
```



Try It Yourself: Enumerations

1. In your ClassExamples project, create an enumeration of various kinds of fruits, just inside the namespace:

```
enum Fruit
{
    Apples,
    Bananas,
    Cranberries,
    Durian,
}
```

2. Set a breakpoint on the first line in the Main method.
3. Run your solution. Execution should stop at your breakpoint.
4. In the immediate window, type `Fruit myFavorite = Fruit` and then a period. What happened?

■

-
5. Select Apples, enter a semicolon and then press enter
 6. Cast the variable `myFavorite` to its int equivalent: `(int)myFavorite`
 7. What is the output?
-
8. Stop execution and remove the breakpoint.
 9. Modify your Main method by adding the following:

```
string fruitOptions = string.Join(", ", Enum.GetNames(typeof(Fruit)));
Console.Write("Enter a fruit ({0}):", fruitOptions);
string intput = Console.ReadLine();
Fruit myFruit = (Fruit)Enum.Parse(typeof(Fruit), intput);
Console.WriteLine("You chose {0}", myFruit);
```



Try It Yourself: Enumerations (Continued)

1. Run it a few times.
2. Try entering in both the names of fruit as well as numbers inside and outside the range 0-3
3. Does entering an invalid string throw an exception?

■

4. Does entering an invalid number throw an exception?

■

5. Describe what each line of code above does

a.

b.

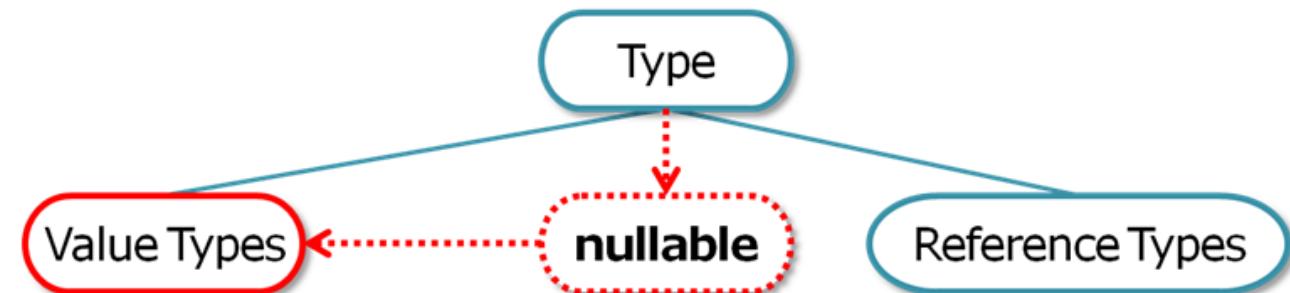
c.

d.

e.

Allowing Empty Value Types

There are times where it might be useful to represent the absence of information within a value-type. By default, this isn't possible, but it can be done if by using a **nullable** value type rather than the normal kind.



Why use nullable types?

Think about a patient's age.

- What does 0 (the default for int) mean?
■ _____
- What does a null value mean?
■ _____

Syntax

To define:

```
T? variable
```

To check for a value:

```
if (variable != null){ ... }
```

or

```
if (variable.HasValue){ ... }
```

```
int? age = null;  
if (age == null) {
```

```
Console.WriteLine("Age unknown");
}
age = 25;
if (age.HasValue) {
    Console.WriteLine("Now the age is {0}", age);
}
```



You can also use `System.Nullable<T>` variable to define a nullable type, which is the full name of the class. Typically the abbreviated definition of `T?` is preferred.

Casting Nullable types

One tricky concept is that a value type, `T`, and its nullable equivalent, `T?`, are not the same because `T?` can be null but `T` cannot. This means that `T` can be implicitly cast to `T?` but not the other way around. If you cast a `T?` to `T` and the value of the `T?` is null, you will encounter an invalid-cast exception.

A safer way to cast is to use the `??` operator:

- `valueType = nullableType ?? default;`

If `nullableType` is null, then the default value is assigned, which avoids the invalid-cast exception.

```
int? age;

int knownAge = age; //will not compile
int knownAge = (int)age; //fails at runtime if age is null
int knownAge = age ?? -1; //assigns -1 if age is null
int knownAge = age ?? default(int); //assigns 0 if age is null
```



Try It Yourself: Nullable types

1. Open the Immediate window.
2. Type: `int knownAge = null;`
 - What is the result?

3. Type: `int? age = null;`
4. Try comparing age against null: `age == null`
 - What is the result?

5. A test that is arguably easier to read is: `age.HasValue`
 - What is the result?

6. Type: `age=25;`
7. What is the result of each test?
 - `age == null`

 - `age.HasValue`

8. Type: `knownAge = age;`
 - a. What is the result?

9. Type: `knownAge = age ?? default(int);`
 - a. What is the result?

Using Operators

C# supports all the mathematical operators that you expect to find in a programming language.

Math Operators

*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction

The return value for division depends on dividend and divisor.

Assignment

There are several operators that combine a mathematical expression and assignment together. These provide shorthand that can make your code less verbose.

<code>*=</code>	Multiply-assign <code>X *= Y;</code> is equivalent to <code>X = X * Y;</code>
<code>/=</code>	Divide-assign <code>X /= Y;</code> is equivalent to <code>X = X / Y;</code>
<code>%=</code>	Modulo-assign <code>X %= Y</code> is equivalent to <code>X = X % Y;</code>
<code>+=</code>	Add-assign <code>X += Y;</code> is equivalent to <code>X = X + Y;</code>
<code>-=</code>	Subtract-assign

	X -= Y; is equivalent to X = X - Y;
--	-------------------------------------

Increment and Decrement Operators

The increment and decrement operators are useful when iterating through sets of indexed data. They are used to increase or decrease the value of an integral type by one, either before or after the surrounding expression is evaluated.

++	Increment
--	Decrement

Examples:

```
value++; // Increments by 1 AFTER use
value--; // Decrements by 1 AFTER use
++value; // Increments by 1 BEFORE use
--value; // Decrements by 1 BEFORE use
```



Try It Yourself

What is the output?

```
int x = 2;
int y = 2;
Console.WriteLine("x = {0}", x++);
Console.WriteLine("y = {0}", ++y);
```

Relational Operators

Relational operators are used to compare two expressions and return a bool; **true** or **false**.

==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to

<	Less than
<=	Less than or equal to

Logical Operators

Logical operators are used to compare two Boolean expressions and return **true** or **false**.

&&	And
	Or
!	Not

- C# will short circuit.
 - Example: `1 == 2 && PerformanceIntensiveMethod()` will evaluate to `false` before performing the performance intensive method because the result of the `&&` operation will always be `false` if the first operand is `false`.



C# also supports Bit Manipulation

- `<<` Shift bits left
- `>>` Shift bits right
- `&` Bitwise and
- `^` Bitwise exclusive or
- `|` Bitwise or
- `~` Bitwise not



To form boolean expressions with nullable boolean variables, the bitwise operators must be used. Because null is a non-determinant state, the results are slightly different when one or both of the values in the boolean expression are null:

- `true & null = null`
- `false & null = false`
- `null & null = null`
- `true | null = true`
- `false | null = null`
- `null | null = null`

Full Order of Operations

The order of operations starts with the operators on the first line (Unary) and then works its way down to the lowest line (Assignment).

Operators on the same level are processed from left to right unless otherwise specified. For example, in the expression `2 % 3 * 2`, the mod happens first because they have equal precedence. Parenthesis can be used to adjust the order that operators are evaluated.

Unary	<code>+</code>	<code>-</code>	<code>!</code>
Multiplicative	<code>*</code>	<code>/</code>	<code>%</code>
Additive	<code>+</code>	<code>-</code>	
Bit Shift	<code><<</code>	<code>>></code>	
Relational	<code><</code>	<code>></code>	<code><=</code>
Equality	<code>==</code>	<code>!=</code>	
Bitwise logicals	<code>&</code>	<code>then</code>	<code>^</code>
Conditional logicals	<code>&&</code>	<code>then</code>	<code> </code>
Assignment	<code>=</code>	<code>*=</code>	<code>/=</code>
			<code>%/</code>
			<code>+=</code>
			<code>-=</code>



Try It Yourself: Expressions

1. Evaluate the following expressions:

a. $6 + 2 * 3 - 4 / 2$

b. $3 * 4 / 2 + 3 - 1$

c. $6 + 2 * (3 - 4) / 2$

d. $3 * 4 / (2 + 2) - 1$

e. $(1 < 3) \&\& (9 > 3)$

f. $(3 > 4) \mid\mid (1 < 2)$

g. $(9 != 1)$

h. $!(3 == 2)$

2. Verify your work using the immediate window

Branching

Code branching is essential to building robust programs. Unconditional branching promotes code reuse and maintainability, while conditional branching allows you to execute the right code at the right time.

In this section, you will learn how to perform both types of branching.

Unconditional Branching

Unconditional branching occurs when you call a method. Execution will jump to the method and will return to the calling code once the method is complete.

To call a method (i.e., branch):

- Add parenthesis after the method name
- Use dot notation to call methods in other classes

The method is finished when:

- The end of the method is reached
- The keyword **return** is encountered (with or without an expression, depending on if the method return type is **void** or not)

Defining a method:

```
static string PromptUserForInput(string prompt)
{
    Console.WriteLine(prompt);
    return Console.ReadLine();
}
```

Calling methods:

```
// In the same class
string name = PromptUserForInput("Enter your name: ");
// In a different class
Console.WriteLine("Your name is {0}", name);
```



Try It Yourself: Unconditional Branching

- In your ClassExamples project, highlight the code that you use to read a fruit from the console. Do not include the line that writes the output to the console. In other words, highlight only these lines:

- ```
string fruitOptions =
 string.Join(",", Enum.GetNames(typeof(Fruit)));
Console.Write("Enter a fruit: ({0}):", fruitOptions);
string intput = Console.ReadLine();
Fruit myFruit = (Fruit)Enum.Parse(typeof(Fruit), intput);
```

- Choose: **right click > Refactor > Extract Method**

- Refactoring is the process of changing the structure of code without changing its function. The purpose is usually to make it more modular and extensible.

- Give the new method the name **GetFruit**

- The method that was created should be:

- ```
private static Fruit GetFruit()
{
    string fruitOptions =
        string.Join(",", Enum.GetNames(typeof(Fruit)));
    Console.Write("Enter a fruit: ({0}):", fruitOptions);
    string intput = Console.ReadLine();
    Fruit myFruit = (Fruit)Enum.Parse(typeof(Fruit), intput);
    return myFruit;
}
```

- The location where the code was originally located should now be:

- ```
Fruit myFruit = GetFruit();
Console.WriteLine("You chose {0}", myFruit);
```

- Place a breakpoint on the line where GetFruit is called and assigned to myFruit.

- Run the application

- Press **F11** to step into the method

- Press **F10** to step through a few lines.

- Hover the mouse over various variables to see what the values are. You can also see all local variable values that are defined in the Locals window (**DEBUG > Windows > Locals**, or **ALT + 4**)

- To run execution until you return from the current method call, press **SHIFT + F11**

- Press **F5** to run to completion.

- Clear the breakpoints

## Reusing Method Names



## Why?

Why reuse method names?

- \_\_\_\_\_

Reusing a method name, known as **overloading**, is possible if the signature (parameter list) is different.

```
//A method to drive a car

public double DriveCar(double miles)
{
 _milage += miles;
 return _mileage;
}

//Another method to drive a car that uses different information

public double DriveCar(double averageSpeed, double time) {
 return Drive(averageSpeed * time / 60.0);
}
```

When code calls a method, the actual method called is the one whose signature matches the arguments passed.



### Try It Yourself: Reusing Method Names

1. In your ClassExamples project, hold down **CTRL** and click on `Writeline` in `Console.WriteLine("Hello world!")`. This will take you to the definition of `Console.WriteLine`.
2. How many different methods of the `Console` class are named `WriteLine`?
  - \_\_\_\_\_
3. Notice how they each method takes in different input.
4. Which method did you call when you wrote `Console.WriteLine("Hello world!")`?
  - \_\_\_\_\_
5. Add a new line of code that says `Console.WriteLine(52.3);`
6. Which method will be called this time?
  - \_\_\_\_\_
7. Verify your answer by going to the definition via **CTRL + Click**. Notice which line the cursor defaults to.



In Visual Studio, you can use **CTRL + Click** to go to the definition of a method. The keyboard shortcut for this is **F12**. If the code is defined in the same assembly (project), you'll see the full definition. You can try this with `GetFruit`. If the method is defined in a different assembly, Visual Studio does not have access to the source code. This was the case with `Console.WriteLine`. Instead, it will pull the metadata from the manifest so you can see the XML documentation and the method signature.

## Passing Data with Parameters

By default, passing parameters into a method is input only (passed by value). But there are other ways to pass parameters as well.

- For output parameters use the `out` keyword
- For input/output parameters use the `ref` keyword

You'll need to include the `ref/out` keywords both when defining the method and when calling it.

```
//This method uses the return value to inform whether the car succeeded
in driving or not. It also passes back the gas required to drive the car
via an output parameter.

public bool TryDriveCar(double milesDesired, out double gasRequired)
{
 gasRequired = milesDesired / MilesPerGallon;
 if (gasRequired <= _gallonsOfGas)
 {
 Drive(milesDesired);
 return true;
 }
 return false;
}

//Use the out keyword when calling the method also

double gasNeeded;

bool hasDriven = TryDriveCar(100, out gasNeeded)

if (hasDriven)
{

 Console.WriteLine("The car drove 100 miles and used {0} gallons of
gas.", gasNeeded);

}

else
{

 Console.WriteLine("The car needs {0} gallons of gas to drive that
far.", gasNeeded);

}
```

The `ref` keyword works very similarly to the `out` keyword. The expectation is that a method using `out` will not use the data passed in. A method using `ref` will use the data passed in.



You don't need to pass reference type variables by `ref`! Since a reference type variable is just a pointer to the heap, the pointers for the argument and the parameter (the variable passing the data and the variable receiving the data) will point to the same spot on the heap. Change the data on the heap, and both pointers will point to the changed data.

Passing a reference type variable by `ref` means something very different and is only done in rare circumstances. It means the object passed into the method may not be the same object passed out of the method. Remember a reference type variable contains a memory address. If you pass a memory address by `ref`, that memory address can be changed to point to a completely different object.

## Conditional Branching

Conditional branching occurs when a segment of code may or may not be executed, depending on the result of a Boolean expression. There are two types of conditional branches: `If/else` and `switch`.

### The `If/Else` Statements

The `if` statement allows you to conditionally execute one or more lines of code based on the result of a Boolean expression. The `else` statement allows you to conditionally execute code when the previous `if` statement evaluates to `false`.

There are two forms of the `if` statement:

```
// One statement executed when expression is true
if (Boolean-expression) statement;

// All statements within block are executed when expression is true
if (Boolean-expression)
{
 statement;
 ...
}
```

At Epic, we will always use blocks of code after an `if` statement. In other words, always include the braces: `{}`. An `else` is optional and should only be used when it makes sense to do so.

```
if (value1 > value2)
{
 Console.WriteLine("First is bigger");
}
else if(value1 < value2)
```

```
{
 Console.WriteLine("Second is bigger");
}
else
{
 Console.WriteLine("Values are equal");
}
```



Use the snippet switch to generate the basic code for an if statement.

The `if` code snippet also comes with a field (`true`). When there are multiple fields, you can tab through to populate them. Try these snippets:

**if:**

```
if (true)
{
}
```

**else:**

```
else
{
}
```



### Try It Yourself

1. In the ClassExamples project, after reading in a fruit, add an If/else statement:

- ```
if(myFruit == Fruit.Apples)  
{  
    Console.WriteLine("I also like them apples!");  
}  
else  
{  
    Console.WriteLine("I wish we all liked them apples...");  
}
```

2. Test the code to make sure it works as expected.

The Switch Statement

The **switch** statement allows code to branch to one of several different options depending on the value of a single expression. The value is compared against a set of constant-expressions.

- bool
- char, string
- int, uint, long, ulong
- enum
- Nullable version of above types

A default case is also allowed, which is a catch all for values not explicitly given their own case. The default is not required.

Syntax:

```
switch (expression)
{
    case constant-expression1:
        statements;
        break;
    case constant-expression2:
        statements;
        break;
    ...
    [default:
        statements;
        break;]
}
```



Use the snippet **switch** to generate the basic code for a switch statement:

```
switch (switch_on)
{
    default:
}
```

- Use **break;** to end a case



- Use `goto default;` to jump to the default case from elsewhere in the switch statement.
- You can group cases together to execute same code:

```
switch(expression)
{
    case const-expr1:
    case const-expr2:
        // const-expr1 and const-expr2 fall here
        statements;
        break;
}
```

```
Random random = new Random(); //Number gen
int roll = random.Next(1, 20);
switch (roll) {
    case 20:
        Console.WriteLine("Critical hit! Roll again to confirm...");
        break;
    case 1:
        Console.WriteLine("You swing too hard and dislocate your
shoulder.");
        break;
    default:
        Console.WriteLine("Do the math.");
        break;
}
```



Try It Yourself: Conditional Branching

1. In the ClassExamples project, comment out the if/else statement.
2. Replace it with an equivalent switch statement:

```
■ switch (myFruit)  
{  
    case Fruit.Apples:  
        Console.WriteLine(  
            "I also like them apples!");  
        break;  
    default:  
        Console.WriteLine(  
            "I wish we all liked them apples...");  
        break;  
}
```

3. Modify the switch so there is a specific message of disgust for each non-apple fruit, but they all fall into the default after (other than apples). For example:

```
a. switch (myFruit)  
{  
    case Fruit.Apples:  
        Console.WriteLine(  
            "I also like them apples!");  
        break;  
    case Fruit.Bananas:  
        Console.WriteLine(  
            "Bananas? Really? Looks like you slipped up.");  
        goto default;  
    default:  
        Console.WriteLine(  
            "I wish we all liked them apples...");  
        break;  
}
```

Looping

Looping is another basic task that a programmer must perform in all programming languages. Like branching, it promotes code reuse.

For example, if you need to read in 100 strings from the console, you can write the code once and then repeat it 100 times with a looping construct, rather than copying the code 100 times.

Also, certain tasks are impossible without the ability to repeat some code an indeterminate number of times.

C# has three looping constructs:

- The While Loop
- The Do-While Loop
- The For Loop

The While Loop

A **while** loop executes code within the loop while a condition is true. The Boolean expression is tested before the loop executes. Because the test happens before the code is executed, the code may not execute at all.

Syntax:

```
while (Boolean-expression)
{
    //or statement block
}
```



Use the snippet **while** to generate the basic code for a while loop:

```
while (true)
{
}
```

Example:

```
int number;
Console.Write("Enter an integer: ");
while (!int.TryParse(Console.ReadLine(), out number))
```

```
{  
    Console.WriteLine("That is not an integer.");  
    Console.Write("Enter an integer: ");  
}
```



Try It Yourself: While Loops

1. In your ClassExamples project, modify GetFruit so that it prompts the user for a fruit over and over again until a valid fruit is selected.
 - **Hint 1:** Use the `Enum.TryParse<Fruit>(string, out parsedFruit)` method along with a while loop.
2. If you get stuck, examine the following code example.

```
private static Fruit GetFruit()  
{  
  
    Fruit myFruit = default(Fruit);  
    bool isValid = false;  
    string fruitOptions = string.Join(", ", Enum.GetNames(typeof(Fruit)));  
    Console.Write("Enter a fruit ({0}):", fruitOptions);  
  
    while (!isValid)  
    {  
        if(Enum.TryParse<Fruit>(Console.ReadLine(), out myFruit))  
        {  
            isValid = Enum.IsDefined(typeof(Fruit), myFruit);  
        }  
        if(!isValid)  
        {  
            Console.WriteLine("That is not a Fruit.");  
            Console.WriteLine("Enter the name of a fruit exactly as listed");  
            Console.Write("Enter a fruit ({0}):", fruitOptions);  
        }  
    }  
    return myFruit;  
}
```

In general, TryParse is a safer form of Parse. Rather than throwing an exception if the conversion fails, it simply returns false. If the conversion succeeds, then TryParse returns true, and the second argument (marked with out) returns the parsed value.



The method `Enum.TryParse<T>(string, out enumVal)` is an example of a generic method because it uses the `<T>` syntax as part of the method name. The type that is parsed and returned through `enumVal` is specified using the `T` generic parameter when the method is called.

Generic methods will be covered more in a later lesson.



One limitation of both `Enum.Parse` and `Enum.TryParse` is that they permit integer results when a number that is not defined in the enumerated type is entered.

To filter out integers that are outside the range, pass the resulting (parsed) enum value into the `Enum.IsDefined` method:

```
EnumType parsedValue;
string input = Console.ReadLine();
bool isValid = false;

if(Enum.TryParse<EnumType>(input, out parsedValue))
{
    isValid = Enum.IsDefined(typeof(EnumType), parsedValue);
}
```

The Do-While Loop

A **Do-While Loop** executes code within the loop while a condition is true. The Boolean expression is tested after the loop executes, so the code in the loop will always execute at least one time.

Example:

```
do
{
    //statement block
}
while (Boolean-expression);
```

Example:

```
int number;
do
{
    Console.Write("Enter an integer: ");
    if (int.TryParse(Console.ReadLine(), out number)) {
        break; //Exit the loop
    }
}
```

```
}
```

```
Console.WriteLine("That is not an integer.");
```

```
} while (true);
```



Use the snippet **do** to create the basic code for a do-while loop:

```
do
```

```
{
```

```
} while (true);
```



Try It Yourself: Do-While Loops

Rewrite the fruit validation loop in the ClassExamples project so it uses a do-while loop instead of a while loop. If you get stuck, examine the following code example box.

```
private static Fruit GetFruit()
{
    string fruitOptions = string.Join(", ", Enum.GetNames(typeof(Fruit)));

    Fruit myFruit = default(Fruit);
    bool isValid = false;
    string input = null;

    do
    {
        if (!string.IsNullOrWhiteSpace(input))
        {
            Console.WriteLine("That is not a fruit.");
            Console.WriteLine("Enter the name of a fruit exactly as listed");
        }
        Console.Write("Enter a fruit: ({0}):", fruitOptions);
        input = Console.ReadLine();

        if(Enum.TryParse<Fruit>(input, out myFruit))
        {
            isValid = Enum.IsDefined(typeof(Fruit), myFruit);
        }
    } while (!isValid);

    return myFruit;
}
```

The For Loop

A **For Loop** is an iterative-style looping construct. The Boolean test is performed before the loop executes.

- All parts of `for` are optional
- `continue;` jumps to the next iteration

Syntax:

```
for(initializers; Boolean-expression; iterators) statementOrBlock;
```



Use the snippet **for** to create the basic code for a for loop that iterates over the length of a collection from smallest to largest index:

```
for (int i = 0; i < length; i++)  
{  
}
```

Use the snippet **forr** to create the basic code for a for loop that iterates over the length of a collection in reverse, from largest to smallest:

```
for (int i = length - 1; i >= 0; i--)  
{  
}
```

Example:

```
for (int counter=0; counter<10; counter++)  
{  
    Console.WriteLine("counter: {0}", counter);  
}
```



If you need to track multiple variables at once, separate multiple initializers and iterators with commas:

```
for(int i=0, j=0; i < 10; i++, j++);
```



Try It Yourself: For Loops

Since `Enum.TryParse<Fruit>` will accept both the name and index of a fruit, it would be better to include the index in the list of options. This is easy to do using a for loop.

- Comment out this line of code:

```
string fruitOptions =
    string.Join(", ", Enum.GetNames(typeof(Fruit)));
```

- On the next line, define an array of strings:

- `string [] fruitOptionsArray = Enum.GetNames(typeof(Fruit));`
- Note:** We will cover arrays in more detail in a later lesson.

- Using the snippet, create this for loop to iterate through all the options:

```
for (int optionIndex = 0; optionIndex <
fruitOptionArray.Length; optionIndex++)
{
}
```

- In the for loop, prepend the index of each string in front of the name:

```
fruitOptionArray[optionIndex] = string.Format("[{0}] {1}",
    optionIndex, fruitOptionArray[optionIndex]);
```

- On the next line, join the array elements together into a comma-delimited string:

```
string fruitOptions = string.Join(", ", fruitOptionArray);
```

- Test your code to ensure it works as expected

- The `GetFruit` method is looking a bit cluttered. Extract code that creates the fruit options string to a separate method called `GetFruitOptions`.

- Examine the following code example for the solution.

```
private static Fruit GetFruit()
{
    string fruitOptions = GetFruitOptions();
    Fruit myFruit = default(Fruit);
    bool isValid = false;
    string input = null;

    do
    {
        if (!string.IsNullOrWhiteSpace(input))
        {
            Console.WriteLine("That is not an Fruit.");
            Console.WriteLine(
                "Enter the name of a fruit exactly as listed");
        }

        Console.Write("Enter a fruit: ({0}):", fruitOptions);
```

```
input = Console.ReadLine();

if (Enum.TryParse<Fruit>(input, out myFruit))
{
    isValid = Enum.IsDefined(typeof(Fruit), myFruit);
}
} while (!isValid);
return myFruit;
}

private static string GetFruitOptions()
{
    string[] fruitOptionArray = Enum.GetNames(typeof(Fruit));
    for (int optionIndex = 0; optionIndex < fruitOptionArray.Length;
optionIndex++)
    {
        fruitOptionArray[optionIndex] =
            String.Format("[{0}] {1}", optionIndex,
fruitOptionArray[optionIndex]);
    }
    string fruitOptions = string.Join(", ", fruitOptionArray);
    return fruitOptions;
}
```



Try It Yourself: Generic Methods

If you have time and with a little extra work, you could get this code to generate the list of menu options for any enum, not just the fruit enum.

This is an advanced exercise, so try it out if you're interested.

1. Investigate the definition of `Enum.TryParse` by right clicking on it and selecting **Goto definition** (You can also arrow the cursor to it and then press **F12**)
2. Copy the definition that it jumps to and paste it into your Program class as a new method
3. Modify the method definition as follows:
 - a. Change the return type from **bool** to **string**
 - b. Change the name from **TryParse** to **GetEnumOptions**
 - c. Remove the parameters
 - d. Replace the terminating semicolon with a code block.
 - e. Paste the code from **GetFruitOptions** into the code block of **GetEnumOptions**
 - f. Everywhere that the type **Fruit** is used, use **TEnum** instead. This represents the type passed in by the user.
 - g. Change the variable names to not include the word "fruit". There is a rename option in the refactor menu (**right-click > Refactor**).
 - h. Compare your code to the following example

```
public static string GetEnumOptions<TEnum>() where TEnum : struct
{
    string[] optionArray = Enum.GetNames(typeof(TEnum));
    for (int optionIndex = 0;
        optionIndex < optionArray.Length; optionIndex++)
    {
        optionArray[optionIndex] =
            String.Format("[{0}] {1}", optionIndex, optionArray[optionIndex]);
    }
    string options = string.Join(", ", optionArray);
    return options;
}
```



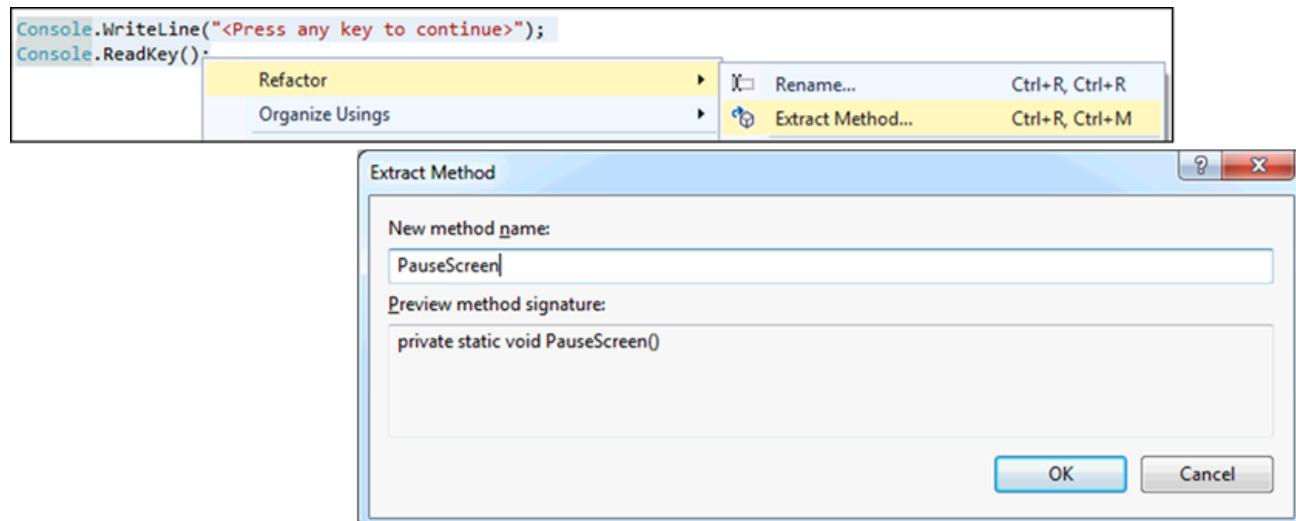
The **where** keyword places restrictions on the kinds of types that can be used in a generic method. In this case, we are forcing the class specified to be a value type using the **struct** keyword. We'll cover structs in more detail in a later lesson.

Activity: Validating User Input

In this exercise you will use branching and looping to validate user input in your SimZoo application.

1. You already have code that informs the user to "Press any key to continue" and then pauses execution. Rather than repeating both lines every time you want to pause, create a custom method in the Program class to do this.
 - Use the refactoring option to do this.

Path: Highlight code > right click > Refactor > Extract method



Refactoring to extract a method

```
/// <summary>  
/// Pause the screen until a key is pressed  
/// </summary>  
private static void PauseScreen()  
{  
    Console.WriteLine("<Press any key to continue>");  
    Console.ReadKey();  
}
```

2. Use the PauseScreen method in place of the original two lines everywhere they were used.
3. Repeat the process that you used to create the PauseScreen method to create a new method called ShowSplashScreen. It should include the following lines of code:

```
Console.WriteLine("SimZoo Text v. 1.0");  
Console.WriteLine("Author: {1}, {0}", AuthorFirstName, AuthorLastName);  
PauseScreen();  
Console.Clear();
```

4. Call the ShowSplashScreen method rather than calling the included lines of code directly.
5. Create another method called GetAnimal. This method will be slightly different because it will have two output parameters:

```

/// <summary>
/// Get the name of a type of animal and the number
/// of that type of animal to add to the zoo.
/// </summary>
/// <param name="animalName">The entered animal name</param>
/// <param name="animalCount">The number of that animal</param>
private static void GetAnimal(out string animalName, out uint animalCount)
{
    // Get the animal name
    ...
    // Get the number of animals
    ...
}

```

6. Change GetAnimal so that it only allows non-null animal names
 - Use a do-while loop that stops looping after a non-null name has been entered
 - To verify that the name isn't null or only consists of white space, use the method `IsNullOrEmpty` from the `string` class. It returns a Boolean.
7. Change GetAnimal so that the entered number of animals parses without throwing an exception and only allows positive integers
 - Use a while loop for this.
 - What kind of data type only allows positive numbers? _____
 - Use that data type to call the TryParse method.
8. The next step is to create a menu for the user to choose options from. You will use an enumerated type to collect the various end-user options in a convenient location.
 - Create an enumerated type of menu options, including one for "Add" and one for "Quit". Be sure to place it above the Program class, just inside the namespace declaration:

```

namespace Epic.Training.SimZoo.Text
{
    /// <summary>
    /// Enumeration of menu options
    /// </summary>
    enum MenuOption
    {
        /// <summary>

```

```
    /// Add an animal to the zoo
    /// </summary>
    Add,
    /// <summary>
    /// Exit SimZoo
    /// </summary>
    Quit
}
```

9. You need to create a menu loop to repeatedly present the two options to the user, but you also want to record the number of times the loop iterates for reporting purposes.

- Use a for loop to take care of both of these requirements:

```
int loopCount;
bool done = false;
for (loopCount = 1; !done ; loopCount++)
{
    ...
    PauseScreen();
}
```

10. Create a new private static method named `DisplayMenu` that returns a nullable `MenuOption` (the enumeration you created in a previous step).

- If the value returned is null, that means the user selected an option that is not part of the enumeration.
- Examine the following code to see how this could be done:

```
/// <summary>
/// Display a list of menu options to the user
/// </summary>
/// <returns>
/// The selected option, or null if a valid option was
/// not selected
/// </returns>
private static MenuOption? DisplayMenu()
{
    MenuOption result;

    // Display the menu
    Console.Clear();
    Console.WriteLine("{0}. ({1}) Add an animal to the zoo",
        (uint)MenuOption.Add, MenuOption.Add);
```

```

Console.WriteLine("{0}. ({1}) Quit SimZoo",
    (uint)MenuOption.Quit, MenuOption.Quit);
Console.Write("\n\nWhat do you want to do?:");

// Get the result
string input = Console.ReadLine();

// Put into title case (matches MenuOption members)
input = CultureInfo.CurrentCulture.TextInfo.ToTitleCase(input);

// Parse the input (accepts number or letter)
if (Enum.TryParse<MenuOption>(input, out result))
{
    return result;
}
return null;
}

```

! In order to make the above code you will have to add the following code to the top of your .cs file:

```
using System.Globalization;
```

The `using` directive will be discussed in more detail later.



The `CultureInfo` class is used to access features that change depending on the current locale and language. For example, title case doesn't always mean the same thing in every language.

! This code is not I18N compliant

The problem is that the enumerated type names are being used directly as user input. This is fine for now, but if this code needed to be translated to other languages then it would need to be adjusted. The problem is that it is impossible to tokenize enum members.

The solution would be to create a method that maps the type names to strings and then to tokenize those strings.

11. Call `DisplayMenu` from the menu loop and store it in a nullable `MenuOption` variable.
12. Use a switch statement to respond to the user's choice.

- If the user chose MenuOption.Quit, set done to true and report the number of iterations of the menu loop.
 - If the user chose Add, Call GetAnimal.
 - If the user chose anything else, inform the user that the selection was not valid.
13. Try running your solution. What number do the menu options start from?
14. Modify the MenuOptions enum so that the options start from 1.

If You Have Time

- Modify the menu so that the option list is generated from the enum definition rather than being coded by hand
- Ensure that invalid numeric options are filtered out using Enum.IsDefined

Exercise Solution

```
using System;
using System.Globalization;

namespace Epic.Training.SimZoo.Text
{
    /// <summary>
    /// Enumeration of menu options
    /// </summary>
    enum MenuOption
    {
        /// <summary>
        /// Add an animal to the zoo
        /// </summary>
        Add = 1,
        /// <summary>
        /// Exit SimZoo
        /// </summary>
        Quit
    }

    /// <summary>
    /// The class used to drive the user interface
    /// </summary>
    class Program
    {
        #region class constants
        /// <summary>
        /// The first name of the author
        /// </summary>
```

```
/// </summary>
const string AuthorFirstName = "FirstName";
/// <summary>
/// The last name of the author
/// </summary>
const string AuthorLastName = "LastName";
#endregion

/// <summary>
/// The entry point into the application
/// </summary>
/// <param name="args">Set of command-line arguments</param>
static void Main(string[] args)
{
    ShowSplashScreen();
    string animalName;
    uint animalCount;
    int loopCount;
    bool done = false;
    for (loopCount = 1; !done; loopCount++)
    {
        MenuOption? selection = DisplayMenu();
        switch (selection)
        {
            case MenuOption.Add:
                GetAnimal(out animalName, out animalCount);
                Console.WriteLine("You added {0} {1} to your zoo!",
                    animalCount, animalName);
                break;
            case MenuOption.Quit:
                Console.WriteLine("Quitting...");
                Console.WriteLine(
                    "(Statistics: The loop iterated {0} times)",
                    loopCount);
                done = true;
                break;
            default:
                Console.WriteLine(
                    "That was not a valid selection. Please try again");
                break;
        }
        PauseScreen();
    }
}

/// <summary>
/// Display a list of menu options to the user
```

```
/// </summary>
/// <returns>
/// The selected option, or null if a valid option was not selected
/// </returns>
private static MenuOption? DisplayMenu()
{
    MenuOption result;

    // Display the menu
    Console.Clear();
    Console.WriteLine("{0}. ({1}) Add an animal to the zoo",
        (uint)MenuOption.Add, MenuOption.Add);
    Console.WriteLine("{0}. ({1}) Quit SimZoo",
        (uint)MenuOption.Quit, MenuOption.Quit);
    Console.Write("\n\nWhat do you want to do?:");

    // Get the result
    string input = Console.ReadLine();

    // Put into title case (matches MenuOption members)
    input = CultureInfo.CurrentCulture.TextInfo.ToTitleCase(input);

    // Parse the input (accepts number or letter)
    if (Enum.TryParse<MenuOption>(input, out result))
    {
        return result;
    }
    return null;
}

/// <summary>
/// Get the name of a type of animal and the number
/// of that type of animal to add to the zoo.
/// </summary>
/// <param name="animalName">The entered animal name</param>
/// <param name="animalCount">The number of that animal</param>
private static void GetAnimal(
    out string animalName, out uint animalCount)
{
    do // Get the animal name
    {
        Console.Write(
            "Enter the name of an animal to include in the zoo: ");
        animalName = Console.ReadLine();
        if (!string.IsNullOrWhiteSpace(animalName))
        {
```

```
        break;
    }
    Console.WriteLine("You must enter a name for the animal.");
} while (true);

// Get the number of animals
Console.Write("Enter the number of this kind of animal: ");
while (!uint.TryParse(Console.ReadLine(), out animalCount))
{
    Console.WriteLine("You must enter a positive integer");
    Console.WriteLine("Enter the number of this kind of animal: ");
}
}

/// <summary>
/// Display the opening splash screen on the console.
/// </summary>
private static void ShowSplashScreen()
{
    Console.WriteLine("SimZoo Text v. 1.0");
    Console.WriteLine("Author: {1}, {0}",
        AuthorFirstName, AuthorLastName);
    PauseScreen();
    Console.Clear();
}

/// <summary>
/// Pause the screen until a key is pressed
/// </summary>
private static void PauseScreen()
{
    Console.WriteLine("<Press any key to continue>");
    Console.ReadKey();
}
}
```

Lesson 2: Structuring Behavior and Data

The Big Picture.....	2•3
By the End of This Lesson, You Will Be Able To.....	2•3
Why do you care?.....	2•4
Using the Model-View-Controller Design Pattern.....	2•4
Defining Custom Types.....	2•7
The Big Picture	2•7
What is a class?	2•7
Creating State	2•10
Hiding State.....	2•11
Controlling Access	2•14
Accessing Class State	2•14
Defining Simple Properties.....	2•18
Adding Behavior.....	2•19
Creating an Instance.....	2•21
Special Constructors.....	2•24
Referring to the Instance in Which the Code is Running.....	2•25
Using an Instance	2•25
Sharing Between Instances	2•27
Why Share State?	2•27
Shared Behavior and State.....	2•27
Activity - Employee Class.....	2•29
Part 1: Create a New project.....	2•29
Part 2: Finish the Employee Class	2•30
Making Your Types Easy to Use.....	2•32
Ways to Improve Usability	2•32
Overloading Operators.....	2•32
Relational Operators.....	2•33
Adding Type Conversion.....	2•39
Implicit Conversion	2•39
Explicit Conversion.....	2•40
Activity - Making Convenient Types.....	2•40

If You Have Time.....	2•40
Defining Custom Value Types.....	2•42
Structs vs. Classes.....	2•42
Activity - Create a custom Value Type.....	2•44

Structuring Behavior and Data

The Big Picture

In this lesson, you will learn to create custom data types, which are the building blocks for robust applications. We will also emphasize good design. Our data models must be reusable and efficient on several applications and platforms, so data structure design is paramount.

By the End of This Lesson, You Will Be Able To...

- Concepts
 - Explain the Model View Controller design pattern
 - Explain where C# code is run in context of HSWeb
 - Explain the importance of object-oriented programming
 - Explain what a class is
 - Explain how a class differs from an object
 - Explain how a class differs from a struct
 - Differentiate between the state and behavior of a class
 - Explain why some fields are encapsulated and others are exposed
 - Explain why Epic prefers properties to fields
 - Identify when get/set methods are being used
 - Explain how scope affects the availability of a field/property/method/class
 - Choose the appropriate scope for a field/property/method/class
 - Describe the default scope for fields/properties/methods/classes
 - Explain how new objects get created
 - Explain why different constructors might be used
 - Explain why you might want an instance of a class to share state and behavior with other instances of that class
 - Describe the difference between a field, a constant, a readonly field, and a static readonly field
 - Explain the difference between instance and static fields/methods
 - Describe the circumstances in which you might use the `this` keyword

- Explain why you might overload an operator and what effects this will have for consuming code
- Explain why you would choose to enable either implicit or explicit casting between a class and some other type
- Explain the significance of a `NullReferenceException`
- Explain what gets implicitly added when using auto-implemented properties
- Algorithms
 - Identify the type of identifier based on its name using Epic's naming conventions
 - Create a new project
 - Create a new class
 - Create a new struct
 - Encapsulate a field using a property
 - Create an auto-implemented property
 - Create an instance of a class using an appropriate constructor
 - Inherit from another constructor
 - Resolve references to an unavailable class
 - Use the Visual Studio debugger to step through code
 - Overload the `==` operator
 - Enable casting between a class and another type

Why do you care?

A properly designed data model will result in:

- Less work
- Faster development
- Fewer bugs
- Readable code

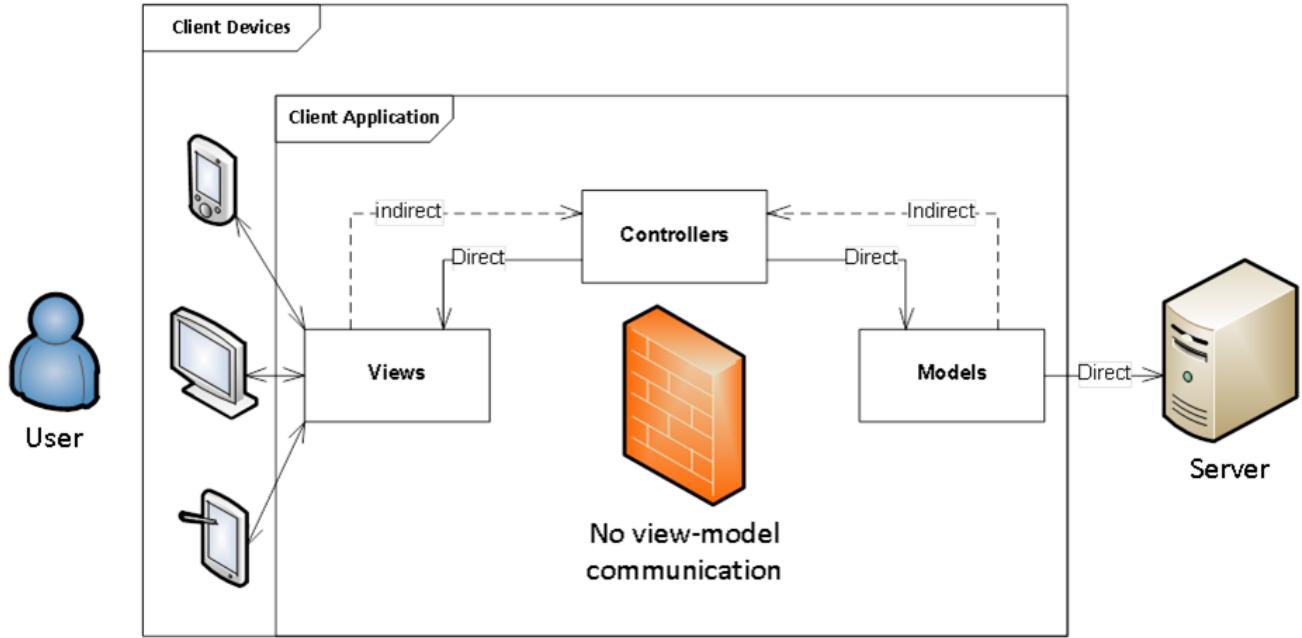
Using the Model-View-Controller Design Pattern

A common design pattern when developing software in C# as well as other object-oriented programming languages is Model-View-Controller (MVC). Hyperspace Web uses a similar design pattern.



In Hyperspace Web, we use a variant of MVC called MVVM, but the principles are the same. For simplicity, we discuss MVC here.

The MVC design pattern is made up of three major components:



The Model-View-Controller Design Pattern

Model	The custom types discussed earlier. The model contains information from your application that will typically be loaded from and saved to the database.
View	Any component of your application that is displayed to the user is known as a View.
Controller	The controllers take data from models and insert it into views so it is available to the user. Controllers also take user input and insert it into the model. Finally, controllers are responsible for implementing navigation and the implementation of required use cases of the application.



Write in Workbook

In the code you have been writing, what is the View?

- _____



Write in Workbook

In the code you have been writing, what is the controller?

- _____

C# code is used for the model in Hyperspace Web. During the rest of this lesson, you will learn how to create the data model for an application using C#.

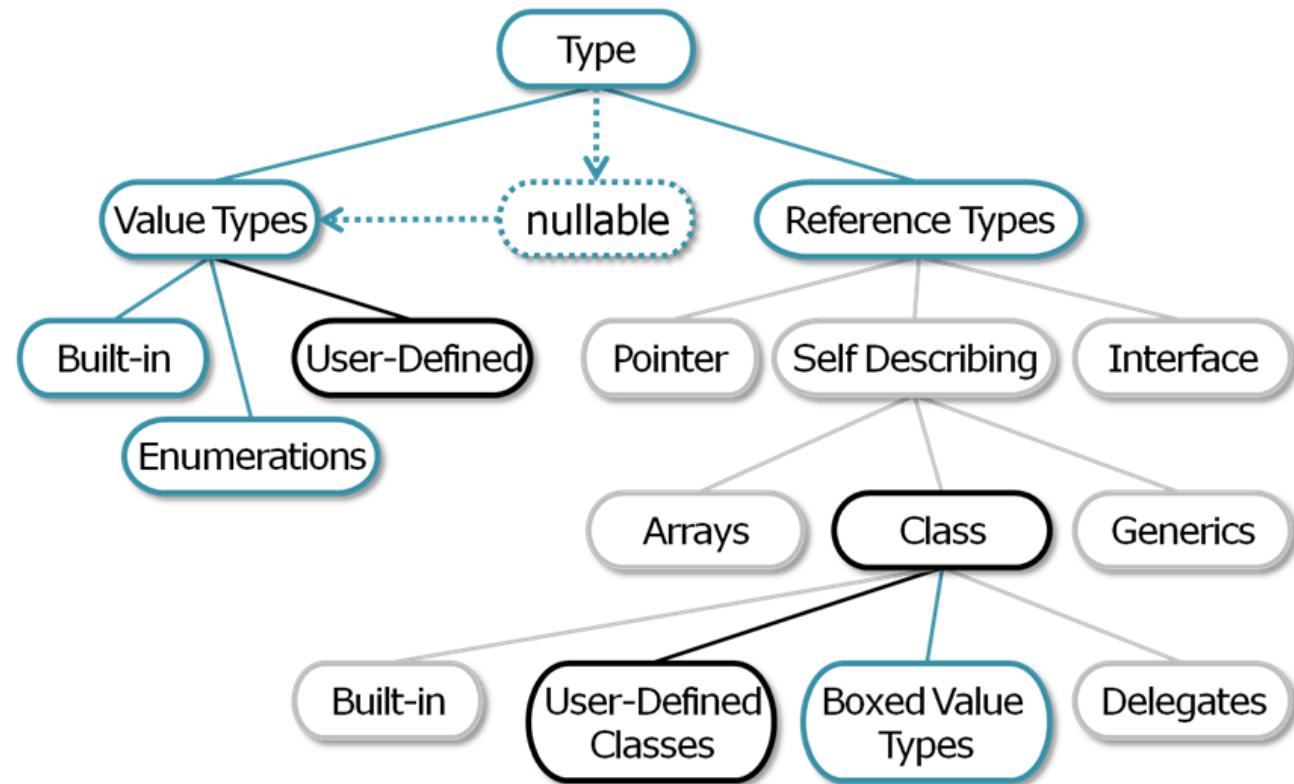


Never access a view from a model! If you do, the model will not be reusable with other views.

- It is okay to do this when testing/debugging.
- There are training examples that do this. These are just for illustration and would never be included in real code.

Defining Custom Types

The Big Picture



What is a class?

- A template for a custom data type
- Defines both the _____ and _____ of a set of objects
- _____ are the instances of those data types
 - The class is the template for objects
 - An object is the actual data in memory
- Objects exist on the heap

```
class Car
{
    // Class Body
}
```



Try It Yourself: Create a Car

We're starting fresh in this chapter. In the ClassExamples project from last chapter, find the Main method of your Program class. Comment out the code from the last chapter, leaving the final Console.ReadKey().

Let's imagine you're creating a racing application. In your ClassExamples project, add another class file:

1. Solution Explorer > ClassExamples > Right click > Add > Class...
2. Name the class Car
3. Click Add

Your code in the Car.cs file should look like this:

```
class Car  
{  
}
```

In the Main method of your Program class, declare and instantiate a variable of type Car:

```
Car mcQueen = new Car();
```

When your code runs, mcQueen will point to a Car object on the heap. Add another Car to your program:

```
Car hudson = new Car();
```

Now hudson will point to a different Car object on the heap.

The **state** variables of a class control what information can be stored. For example, the Dragon class might have

```
string Name;  
  
double Altitude;  
  
double FireRange;  
  
bool IsHungry;
```

Each of these variables can be populated with different values for different Dragon objects.



What kind of state might a car have?

-
-
-
-
-
-

The **behavior** of a class is the set of methods that are available to it. What can the class do? For a dragon, we might have:

```
Fly();  
BreatheFire();  
EatSlaveTrader();
```

Each of these methods might change some of the state variables. For example, `Fly()` might increase the `Altitude` of the Dragon, and `EatSlaveTrader()` might switch `IsHungry` to false.



What kind of behavior might a car have?

-
-
-
-
-

Creating State

- Data is stored in class-level variables called _____

```
class Car
{
    public string CarNickname;
}
```

- Fields can optionally be initialized.
 - Without initializing a field to a value, it will start off with the default value for its type. In the above example, the initial value of CarNickname is null for all cars. In this example, the initial value of CarNickname is "Car" for all cars.

```
class Car
{
    public string CarNickname = "Car";
}
```



Try It Yourself: Add State to a Class

In your car class, add a state variable to store the nickname of a Car:

```
class Car
{
    public string CarNickname;
}
```

In your Program class's Main method, write out the value of mcQueen's CarNickname:

```
Console.WriteLine("mcQueen's nickname: {0}",
mcQueen.CarNickname);
```

Then change the nickname of your McQueen variable to "Lightning". Then write it to the Console, so you can see the change:

```
mcQueen.CarNickname = "Lightning";
Console.WriteLine("mcQueen's nickname: {0}",
mcQueen.CarNickname);
```

Run the solution. Notice how your program was able to change the state of your Car object.

Now change the Car class so the declaration of CarNickname has a default:

```
public string CarNickname = "Car";
```

Run the solution. What's the difference?

- _____

What was the same?

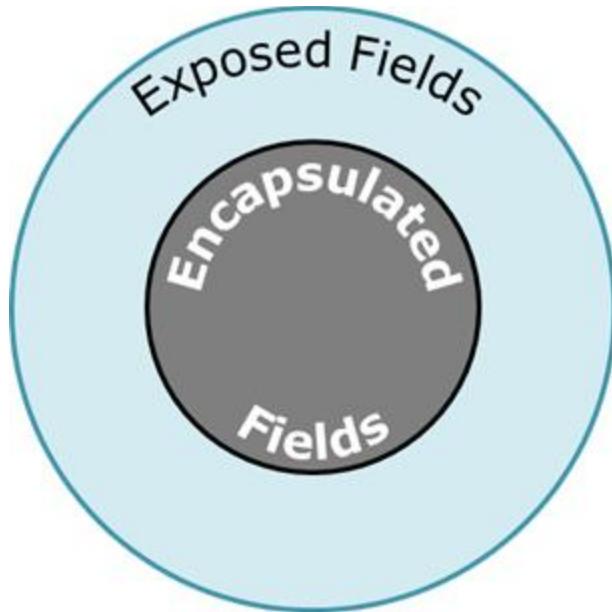
- _____

Now try changing hudson's nickname to "Doc".

Hiding State

Consuming code like the Program class can't always change the state of the object's fields.

- Fields can be _____ or _____



- Consuming code is the code that is using your class. Examples:
 - Consuming code can be code in *another* class.
 - Ex: The Program class consumes the Car class's CarNickname variable when changing the state of CarNickname.
 - Consuming code can be code in *the same* class.
 - Ex: The Dragon class consumes the Dragon class's IsHungry variable when changing the state of IsHungry in the Dragon's EatSlaveTrader method.
- Exposed fields are _____ to consuming code in another class
- Encapsulated fields are _____ to consuming code in another class

```
class Car
{
    public string CarNickname;
    private double _gallonsOfGas;
}
```



Try It Yourself: Encapsulate a Field

One way we can encapsulate fields is to use the `private` access modifier.

In your Car class, add a definition for `_gallonsOfGas` so your code looks like this:

```
class Car
{
    public string CarNickname;
    private double _gallonsOfGas;
}
```

In your Program's `Main` method, set the `_gallonsOfGas` variable for `mcQueen`:

```
mcQueen._gallonsOfGas = 15;
```

Compile your code. What happened? Why?

- _____
- _____
- _____

Remove the offending line of code.

In your Car class, create a method to add fuel to the car:

```
public void AddFuel(double gallonsToAdd)
{
    _gallonsOfGas += gallonsToAdd;
}
```

Compile your code. What happened? Why?

- _____
- _____
- _____

The code that you write for methods *inside* the Car class will be able to access `_gallonsOfGas` and change it. Consuming code *outside* the Car class, like Program, will not.



Why hide state? Why not expose everything?

-

-

-

-

Controlling Access

Other modifiers besides private and public exist. The full list of modifiers is below.

- Access is based on the _____ of the member and the _____ of consuming code.

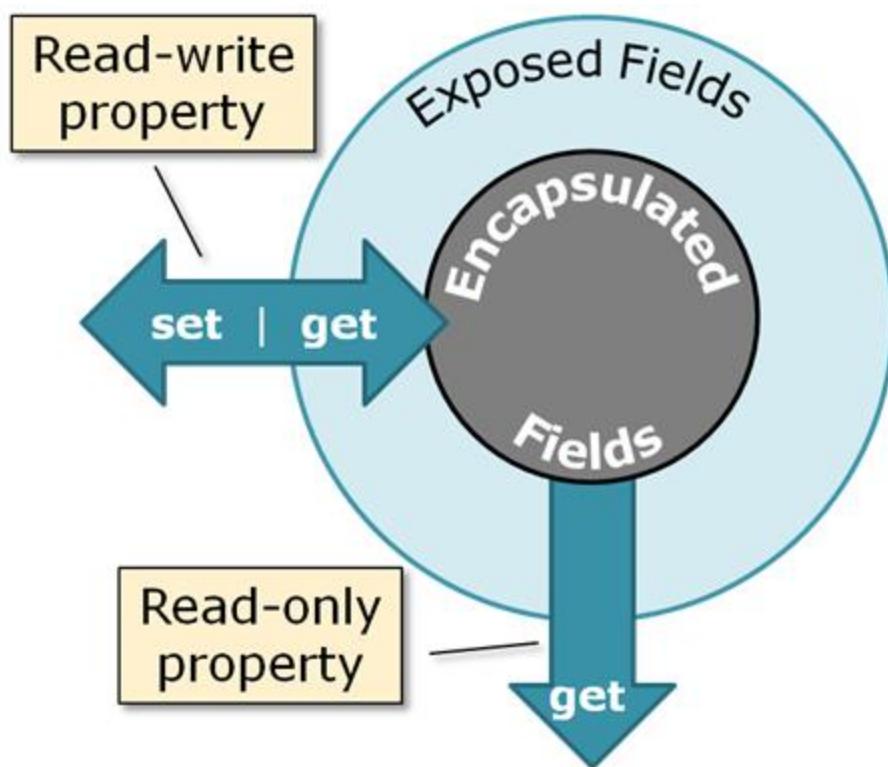
Access Modifier	Exposed to Context
	Any
	Only members of class
	Members of class / derived class (see later lesson)
	All methods in same assembly (project)
	Methods of the same class / derived class or assembly

- Public fields use _____ case
- Private fields use _____ case and are prefixed with "_"

Accessing Class State

Sometimes, you'll want to create more fine-tuned access to state variables.

- _____ allow controlled access to encapsulated fields
- Properties include _____
- Properties should be declared in Pascal Case.



```
public double GallonsOfGas
{
    get { return _gallonsOfGas; }
    private set
    {
        if (value > TankCapacity)
        {
            _gallonsOfGas = TankCapacity;
        }
        else if (value >= 0)
        {
            _gallonsOfGas = value;
        }
    }
}
```



Try it yourself: Control Access with a Property

Instead of completely obscuring `_gallonsOfGas`, we want to make it read-only (allow other consuming code to view it, but not change it). Add the following lines to the `Car` class:

```
public double GallonsOfGas
{
    get { return _gallonsOfGas; }
}
```

You've declared a property called `GallonsOfGas` and defined a `get` method for it. When consuming code examines the value of this property, it will return what its `get` method returns.

In the `Program` class, add some fuel to `mcQueen` and write out how much gas `mcQueen` has:

```
mcQueen.AddFuel(15);
Console.WriteLine("mcQueen's gas: {0}", mcQueen.GallonsOfGas);
```

Run your solution. Does the value of `GallonsOfGas` reflect `_gallonsOfGas`? _____

This time, add fuel by manually changing the `GallonsOfGas` property. Remove the line that calls `AddFuel` and replace it with this:

```
mcQueen.GallonsOfGas = 15;
```

Run your solution. What happens? Why?

- _____
- _____



Try it yourself: Control Access with a Property (continued)

Suppose we want to allow consuming code to change `_gallonsOfGas`, but we don't want the value to exceed the tank capacity of the car. Let's define a new field in the Car class:

```
public double TankCapacity = 12;
```

Now, add a set method to `GallonsOfGas` to allow consuming code to set its value, but no larger than the tank capacity:

```
public double GallonsOfGas
{
    get { return _gallonsOfGas; }
    set
    {
        if (value > TankCapacity)
        {
            _gallonsOfGas = TankCapacity;
        }
        else if (value >= 0)
        {
            _gallonsOfGas = value;
        }
    }
}
```

Here, the **value** keyword represents the value on the right hand side of the assignment operator (=) when assigning a value to `GallonsOfGas`. In the case of the code you wrote in your Program class, it represents 15.

Run your solution. How many gallons of gas does mcQueen have? Why?

- _____
- _____
- _____

- Properties contain **get** and **set** methods allowing controlled access to private fields.
- Get and set methods are sometimes called **getters** and **setters** respectively.
- The keyword **value** in the set method contains the value assigned to the property.
- The private field that is being encapsulated by a property is often called a **backing field**.
- The access level set to the property is the _____ for both the get and the set methods.
- The access level of _____ get or set may be more restrictive than the default, _____.



Try it yourself: Control Access with a Property (continued)

So far, we've allowed any consuming code to change the value of the gas in a car, so long as that value doesn't exceed the car's tank capacity. For a real car, we can't just set the gas level. We must add fuel to what already exists. Let's force other consuming code to add fuel through the `AddFuel` method.

Add the private access modifier to the set method of `GallonsOfGas`:

```
private set { ... }
```

Run your solution. What happens? Why?

- _____

Remove the line that sets `mcQueen`'s `GallonsOfGas`, and use the `AddFuel` method instead. The code will compile.

However, because `AddFuel` directly modifies the private backing field `_gallonsOfGas`, `AddFuel` will allow consuming code to put bad values into `_gallonsOfGas`. Try adding 1000 gallons to `mcQueen`:

```
mcQueen.AddFuel(1000);
Console.WriteLine("mcQueen's gas: {0}", mcQueen.GallonsOfGas);
```

Run your solution. How many gallons of gas does `mcQueen` have? _____

Ack! To prevent `mcQueen`'s front seats from filling up with gasoline, `AddFuel` should be setting the property instead of the private backing field. In `AddFuel`, remove the line that adds to the private backing field, and add to the property instead:

```
GallonsOfGas += gallonsToAdd;
```

Run your solution. How many gallons of gas does `mcQueen` have? _____



To quickly write a property with a backing private field, use the **propfull** snippet.

Defining Simple Properties

Some features of .NET only work with properties, not fields, so sometimes, you will want to make a field into a property to allow for some behavior in the future. (Changing something from a field to a property later is a breaking change.) But the full property syntax is a lot to write!

Use shorthand notation for simple get/set properties:

Public field:

```
public string CarNickname;
```

- **Problem:** Certain features only work with properties

Full Property:

```
private string _carNickname;  
public string CarNickname  
{  
    get { return _carNickname; }  
    set { _carNickname = value; }  
}
```

- **Problem:** Full properties are verbose

Shorthand:

```
public string CarNickname { get; set; }
```

- Auto-implements the private field

**Try it yourself: Define a Simple Property**

Let's remove the default nickname of "Car" for cars and make that public field into a property. In the car class, rewrite the CarNickname field:

```
public string CarNickname { get; set; }
```

Does your code compile? _____

You shouldn't notice any difference in your code's behavior.

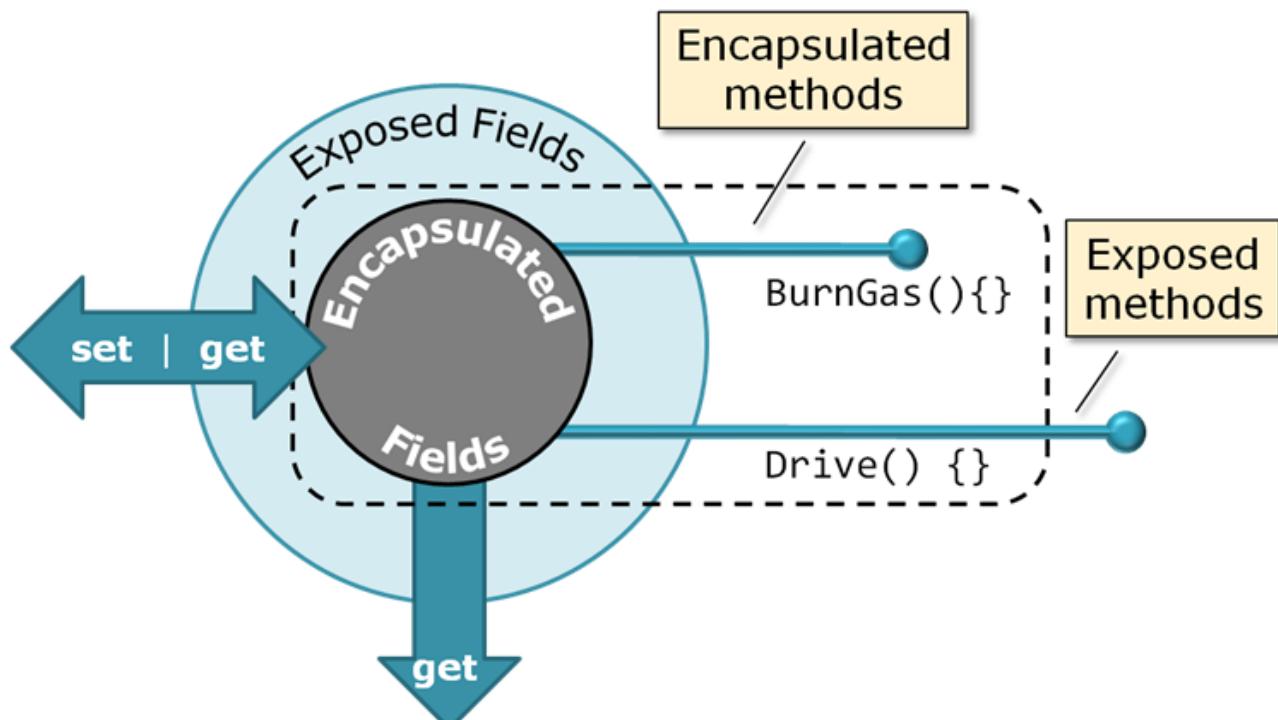
- Epic prefers public properties to public fields because they are more maintainable.



To quickly write an auto-implemented property, try using the **prop** snippet.

Adding Behavior

- Behavior is contained in _____.
- Just like state, methods can be encapsulated or exposed.



Why not expose BurnGas?

- _____
- _____
- _____



Try it yourself: Encapsulate a Method

Add a BurnGas method to your Car class:

```
private void BurnGas(double gasUsed)
{
    GallonsOfGas -= gasUsed;
}
```

Can you call this method from the Program class?

From the Car class?

Now add a method to expose BurnGas. We'll also need to add some fields for this method to use:

```
public readonly double MilesPerGallon = 25;
private double _mileage = 0;
public double Drive(double miles)
{
    double gasUsed = Math.Min(_gallonsOfGas, miles / MilesPerGallon);
    double milesDriven = MilesPerGallon * gasUsed;
    _mileage += milesDriven;
    BurnGas(gasUsed);
    return milesDriven;
}
```

In your Program class, test this out by adding these lines:

```
Console.WriteLine("miles driven: {0}", mcQueen.Drive(150));
Console.WriteLine("mcQueen's gas: {0}", mcQueen.GallonsOfGas);
```

Run your solution.

How many miles did mcQueen drive? _____

How much gas does mcQueen have left? _____

Creating an Instance

We've created the objects `mcQueen` and `hudson` so far, but when they were created, we simply set defaults for some important information (namely `TankCapacity` and `MilesPerGallon`) so their methods could work. Realistically, different cars would have different values for these. We can

declare the information that must be set up when creating a Car object by writing a **constructor** for the class:

- A _____ is a method called to create a class instance
- A constructor is called every time an object is created
- Method name = _____
- _____ return type is specified
- Call a constructor using the _____ keyword

```
// Example constructor
public Car(double tankCapacity, double milesPerGallon)
{
    TankCapacity = tankCapacity;
    MilesPerGallon = milesPerGallon;
}

// Create some cars
Car hybrid = new Car(9.0, 48.9);
Car suv = new Car(25.0, 15.8);
```



Try it yourself: Create a Constructor

In your Program's Main method, comment out the existing code, leaving the `Console.ReadKey()` at the end.

Add a constructor to your Car class that sets up some of the critical information that a car needs to be able to operate:

```
public Car(double TankCapacity, double MilesPerGallon)
{
    TankCapacity = TankCapacity;
    MilesPerGallon = MilesPerGallon;
}
```

In your Program class, create some Cars:

```
Car hybrid = new Car(9.0, 48.9);
Car suv = new Car(25.0, 15.8);
```

To test your cars, write out the values of `hybrid.MilesPerGallon` and `suv.MilesPerGallon` to the Console:

```
Console.WriteLine("hybrid's MPG: {0}", hybrid.MilesPerGallon);
Console.WriteLine("suv's MPG: {0}", suv.MilesPerGallon);
```

Run your solution.

- Use **readonly** to prevent a field from being changed after it is initialized.
 - Readonly fields must be initialized when declared or must be set in the constructor.
 - Readonly does NOT mean Constant. Constants are the same from instance to instance. Readonly fields can be different for different instances of the same class.



Try it yourself: Create Readonly Fields

In an earlier exercise, we declared the public field `MilesPerGallon` in your `Car` class. In your `Program` class, add a line to change the fuel efficiency of the hybrid:

```
hybrid.MilesPerGallon = 5;
```

Compile your solution. Does it compile? Should the `Program` be able to change `MilesPerGallon`?

- _____

We could make `MilesPerGallon` private, but do we even want the `Car` class to change `MilesPerGallon` once it is set?

- _____

`TankCapacity` and `MilesPerGallon` should be `readonly` because once they are set for an instance of the `car` class, they should never be changed, even by the `Car` class. Add the `readonly` modifier to these fields:

```
public readonly double MilesPerGallon;  
public readonly double TankCapacity;
```

Run your solution. What happens? Why?

- _____

Remove the offending line. This time, add it to the `Car`'s `AddFuel` method:

```
MilesPerGallon = 5;
```

Run your solution. What happens? Why?

- _____

Remove the offending line. Your solution should compile.

Special Constructors

It is common to define or use one of these special constructors:

A constructor without parameters.

- Exists implicitly if no other constructors defined
- Otherwise, must be explicitly written
- Required for serialization (covered later)

```
public Car() {  
    TankCapacity = 10.0;  
    MilesPerGallon = 25.0;  
}
```

Takes an instance as its parameter

- Duplicates passed instance

```
public Car(Car copyFrom) {  
    this.TankCapacity = copyFrom.TankCapacity;  
    this.MilesPerGallon = copyFrom.MilesPerGallon;  
}
```



Try it yourself: Create a Default Constructor

Uncomment the code that creates the Car mcQueen. Run your solution. What happens? Why?

- _____

Add a default constructor to the car class:

```
public Car()  
{  
    TankCapacity = 15;  
    MilesPerGallon = 25;  
}
```

The code instantiating mcQueen will now compile because it has a constructor to call.

Referring to the Instance in Which the Code is Running

- Use the `__` keyword
- Useful to:
 - Distinguish instance from parameters
 - Pass a reference to the current instance



Try it yourself: Use the "this" Keyword

Change the casing of the parameters in your constructor so they look like the fields:

```
public Car(double TankCapacity, double MilesPerGallon)
{
    TankCapacity = TankCapacity;
    MilesPerGallon = MilesPerGallon;
}
```

Run your code. What are the values for hybrid's `TankCapacity` and `MilesPerGallon`? Why?

- _____

Modify the lines to include the `this` keyword:

```
this.TankCapacity = TankCapacity;
this.MilesPerGallon = MilesPerGallon;
```

Run your code. What are the values for hybrid's `TankCapacity` and `MilesPerGallon` now? Why?

- _____

There are other uses of the `this` keyword to refer to the current instance that we'll see in future lessons.

Using an Instance

Example:

```
// Create some cars
Car hybrid = new Car(9.0, 48.9) { CarNickname = "Hybi" };
Car suv = new Car(25.0, 15.8) { CarNickname = "Roller" };
// Fill up
```

```
hybrid.AddFuel(9);
suv.AddFuel(25);
// See how far they go! (method)
Console.WriteLine("{0} drove {1:0.0} miles",
    hybrid.CarNickname, hybrid.Drive(500));
Console.WriteLine("{0} drove {1:0.0} miles",
    suv.CarNickname, suv.Drive(500));
```

Output:

```
Hybi drove 440.1 miles
Roller drove 395.0 miles
```

- You can set public properties during creation using _____ syntax (see CarNickname above).



Notice the output of {1} from the WriteLine code in the example includes extra formatting after the colon. In this case, numbers are formatted to display only one digit after the decimal point.

Many other format strings are available. Search the MSDN for more details.

Sharing Between Instances

Why Share State?

- Suppose you want to track the total number of cars on the road.
- Should you store a copy of this with every instance of Car? Why or why not?

■ _____

Shared Behavior and State

- Certain aspects of a class are not specific to an instance.
- These members are called _____ members.
- Indicate this using keyword _____.
- Call these members by referring to the _____ as opposed to the instance name (e.g. Car instead of mcQueen).
- You've seen some static methods already:
 - Console.WriteLine
 - Int.TryParse
 - String.IsNullOrEmpty

```
//Neither method applies to a specific instance.

//a class member that applies to all instances
public static int CarsOnRoad { get; private set; }

//a class member that applies to no instances
public static double CalcAverageMilesPerGallon(params Car[] cars)
{
    double total = 0.0;
    if (cars == null || cars.Length == 0)
    {
        return 0.0;
    }
    foreach (Car c in cars)
    {
        total += c.MilesPerGallon;
    }
}
```

```
    return total / (double)cars.Length;  
}
```



Arrays, the `foreach` loop, and the `params` keyword will be covered in more detail during the collections lesson.



Try it yourself: Create a Shared Property

You'd like to keep track of all the cars on the road. Create a new static property that keeps track of this:

```
public static int CarsOnRoad { get; private set; }
```

When a new car gets created, `CarsOnRoad` should be incremented. In the each of the car constructors, increment the number of cars on the road:

```
CarsOnRoad += 1;
```

Test out your code in the `Program` class:

```
Console.WriteLine("Total cars: (0)", Car.CarsOnRoad);  
Car sedan = new Car();  
Car guzzler = new Car(50, 8.0);  
Console.WriteLine("Total cars: {0}", Car.CarsOnRoad);
```

You should see the number of cars on the road increase by two.



Referring to the Instance You Are In

- What is the keyword? _____
- Will this work from within a class member?
 - _____



Any member field marked as `const` is implicitly static.

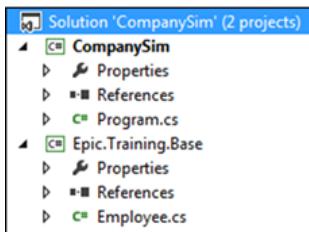
This means that there is no need to have `static const` members. In fact, if you try to do this it is a compile error.

Activity - Employee Class

In this activity you will start a new C# console-based program which will be added to in subsequent exercises. The emphasis is on writing the classes; the testing is up to you.

Part 1: Create a New project

1. Create a new console application project in C:\EpicSource\C Sharp called CompanySim.
2. Change the default namespace of the new project to **Epic.Training.CompanySim.Text**
3. Right-click the solution in the solution explorer and select **Add > new project**
4. Choose the **Class Library** template
5. Name the template **Epic.Training.Base**
6. Rename the automatically created Class1.cs file to **Employee.cs**
7. When prompted, confirm you want to rename all references as well.
8. Your solution should now look similar to the following:



Solution after creating two projects

9. Build your solution
10. Add a reference to **Epic.Training.Base** in **CompanySim**
 - a. Solution Explorer > CompanySim > Right click > Add Reference
 - b. Under projects, choose **Epic.Training.Base** and then click **OK**.



The way to add a reference in the HyperspaceWeb framework is slightly different. This will be covered in more detail during Web Tech Camp.

11. Open Program.cs
12. Change the namespace from CompanySim to Epic.Training.CompanySim.Text
13. In the Main method, define a variable of type Employee.
14. Notice that Employee is not available.
15. Auto-resolve the namespace. This will add a using statement to the top of your file so that when you use the Employee class, it is recognized as the Epic.Training.Base.Employee from your Employee class:

- a. Employee > right click > Resolve > using Epic.Training.Base
 - b. You can also resolve this using the keyboard shortcut **CTRL +**.
16. Open Employee.cs.
17. Examine the class definition. What is the access modifier of the class?
-
-

18. Remove the access modifier and then build your solution. What happens?
-
-



Of the 5 access modifiers available in C#, two of them can be applied to a class.

- internal (the default)
- public

An internal class is not accessible outside the assembly that it is defined in.

19. Place the access modifier that you removed back and then recompile the solution.

Part 2: Finish the Employee Class

Finish the employee class according to the following specifications:

- Properties:
 - Name
 - ID
 - Tenure (how long they've worked)
 - Make sure that tenure cannot be set to less than 0.
 - Base salary
 - Application (Enumeration)
- Static method:
 - CalculateMonthlyPayment - calculates an employee's monthly payment given a salary
- Instance method:
 - GetRaise - raises the employee's base salary by a given dollar amount
- Constructors
 - Add a constructor that takes in an ID string

- Add a copy constructor
- Test the properties, methods and constructors in your Program class.

Making Your Types Easy to Use



Why is it important to make your types easy to use?

-
-
-
-
-

Ways to Improve Usability

- Overload common operators
- Enable casting between types

```
Point a = new Point(2,3), b = new Point(4,5);  
Point c = a + b;
```

It is just like this, but easier and more intuitive:

```
Point c = new Point(a.X + b.X, a.Y + b.Y);
```

But does the below code make sense? _____

```
Car hybrid = new Car(), suv = new Car();  
Car hybridSuv = hybrid + suv;
```

What's the result? A hybrid SUV? A wreck? MegaCar?

Readability is important. If it's not obvious what the result should be, err on the side of NOT overloading the operator.

Overloading Operators

Overloading an operator means giving new meaning to the way an operator (e.g. `+`, `-`, `*`, `==`) is used.

- Operators are static methods
- Operators return a single value
- Parameters of the method are operands
- To overload, use the keyword _____

To enable adding Points together, include the following code in the Point class:

```
public static Point operator +(Point lhs, Point rhs)
{
    return new Point(lhs.X + rhs.X, lhs.Y + rhs.Y);
}

public static Point operator -(Point lhs, Point rhs)
{
    return new Point(lhs.X - rhs.X, lhs.Y - rhs.Y);
}

public static Point operator -(Point rhs)
{
    return new Point(-rhs.X, -rhs.Y);
}
```

- In the example, `lhs` is the Point on the left hand side of the operator, and `rhs` is the Point on the right hand side of the operator.
- All of the methods return Points.

Relational Operators

Relational operators are even more commonly overloaded.

- When overloading a relational operator, always overload the associated operators:

If you overload this...	Also overload this...
<code>==</code>	<code>!=</code> <code>Equals()</code> <code>GetHashCode()</code>
<code><</code>	<code>></code>

If you overload this...	Also overload this...
<=	>=

- Equals is often called to compare one instance of a type with another
- GetHashCode is the method called to determine uniqueness in a hash table.
 - When writing GetHashCode(), make sure that when `a == b`, then `a.GetHashCode() == b.GetHashCode()`.
 - The reverse is often true, but not necessarily
- By default, == will look for reference equality (is it the same spot in memory on the heap)

```
//Overloads the == operator
public static bool operator ==(Point lhs, Point rhs)
{
    //Checks for null before
    //calling lhs.Equals(rhs)
    return object.Equals(lhs, rhs);
}

//Checks for equality - called indirectly by ==
public override bool Equals(object obj)
{
    Point p = obj as Point; //Safely cast to Point
    if (p == null)
    {
        return false;
    }
    return (this.X == p.X && this.Y == p.Y);
}

//Always overload if overloading ==
public static bool operator !=(Point lhs, Point rhs)
{
    return !(lhs == rhs); //Define != in terms of ==
}

//Always override if overloading ==
public override int GetHashCode()
{
    //a clever combination of X and Y is needed here
    return (X * 73856093) ^ (Y * 83492971);
}
```



Try it yourself: Overload ==

For cars, the VIN (Vehicle Identification Number) is the unique identifier for every car on the road. If two car objects have the same VIN, we should consider them equal. Create a VIN property in the Car class:

```
public string VIN { get; private set; }
```

Modify your constructor to take in a VIN:

```
public Car(string VIN, double TankCapacity, double MilesPerGallon)
{
    this.VIN = VIN;
    ...
}
```

Remove your default constructor. Car's shouldn't exist without VINs, and there's no good VIN to give a Car by default.

Since we've changed how objects are constructed, a lot of the code in your Program class you wrote previously won't compile. Comment it out, leaving the final `Console.ReadKey()`.

Now in the Car class, we'll overload `==` to indicate that two Cars are equal if they have the same VIN:

```
public static bool operator ==(Car lhs, Car rhs)
{
    return lhs.VIN == rhs.VIN;
}
```

In your Program class, test this out:

```
Car mcQueen = new Car("5479A", 20.0, 34.8);
Car hudson = new Car("78B4D", 24.0, 26.1);
Car mcTwin = new Car("5479A", 20.0, 34.8);
Console.WriteLine("mcQueen == hudson? {0}", mcQueen == hudson);
;
Console.WriteLine("mcQueen == mcTwin? {0}", mcQueen == mcTwin);
;
```

What do you expect to happen?

- _____
- _____
- _____

Run your solution to test this.



Try it yourself: Overload == (continued)

Let's introduce a wrinkle. What if hudson is removed from memory? What will happen when we run the code?

-
-

Change the line where hudson is created to set hudson to null:

```
Car hudson == null;
```

Run your solution. Is this what you expected?

-
-

To fix this, we must take care of the null case. Fortunately, there is a method that does this for us.

The static method `Equals` from the `Object` class helps us avoid a null reference exception. The following table gives the result of `object.Equals(lhs, rhs)`:

lhs	rhs	Returns
null	null	true
null	not null	false
not null	null	false
not null	not null	<code>lhs.Equals(rhs)</code>

When either lhs or rhs is null, the result can be determined. If both are not null, then the `Equals` method of lhs is called. This method will be your custom code when you override `Equals`.



Try it yourself: Overload == (continued)

Great! To take advantage of this in the Car class, we need to do two things:

- Call `Object.Equals` in the `==` method
- Override the Car's `Equals` in our Car class to use VIN comparison.

Overriding the Car's `Equals` method is necessary because `Object.Equals` will call the Car's `Equals` method.

Change the `==` method to call `Object.Equals`:

```
public static bool operator ==(Car lhs, Car rhs)
{
    return object.Equals(lhs, rhs);
}
```

Override the `Equals` method to compare the VINs.

```
public override bool Equals(object obj)
{
    return this.VIN == obj.VIN;
}
```

Run your solution. What happens? Why?

-

The `Equals` method that gets called by `Object.Equals` does not make any assumptions about the type of the object. It just passes in a generic `object`. Because the `Equals` method is inherited from the `object` class (see next lesson on inheritance), you can't change this, but you wouldn't want to anyway. The `Equals` method still needs to function, because other programmers may want to use it.

To make the `Equals` method work correctly, we must ensure that the object being passed is actually a `Car`. If it is, we can cast the object into a `Car` object and compare VINs. If not, the two are clearly not equal because a `Car` is not equal to not-a-`Car`. To take care of objects that are not `Cars`, we'll cast the object into the `Car` class and check the result:

```
public override bool Equals(object obj)
{
    Car c = obj as Car;
    if (c == null)
    {
        return false;
    }
    return this.VIN == c.VIN;
}
```

The `as` keyword here casts `obj` into a `Car` object `c`. If the cast is unsuccessful, it simply sets `c` equal to `null` instead of throwing an exception. We can check to see if `c` is `null` afterward to know if `obj` was actually a `Car`.



Try it yourself: Overload == (continued)

After adding the above code, run your solution. What happens?

-

Finally, let's override GetHashCode so that VINs are compared when we put Cars into a hash set:

```
public override int GetHashCode() {  
    // Hash code of VIN meets criteria  
    return VIN.GetHashCode();  
}
```



Use override when...

- Method replaces behavior inherited from parent class (in this case, object)
- See lesson on inheritance for details

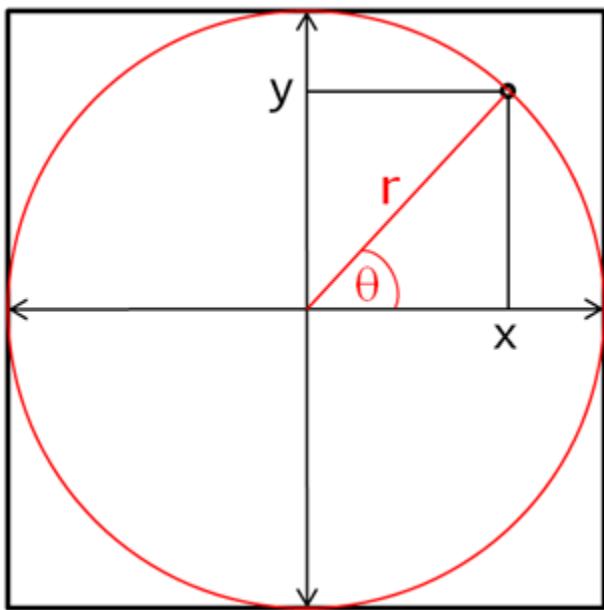
The full example of overloading == is below:

```
//Unique IDs for cars  
public string VIN { get; private set; }  
  
public static bool operator ==(Car lhs, Car rhs)  
{  
    // Checks for null before  
    // calling lhs.Equals(rhs)  
    return object.Equals(lhs, rhs);  
}  
  
public override bool Equals(object obj)  
{  
    Car car = obj as Car; //Safely cast to Car  
    if (car == null) { return false; }  
    return this.VIN == car.VIN; //Same VIN, same car  
}  
  
public static bool operator !=(Car lhs, Car rhs) {  
    return !(lhs == rhs); // Define != in terms of ==  
}  
  
public override int GetHashCode() {
```

```
// Hash code of VIN meets criteria
return VIN.GetHashCode();
}
```

Adding Type Conversion

- Adding common conversions makes it easier to use your custom types.
- For example, suppose you have two classes: one to represent a point in Cartesian coordinates, and another to represent a point in polar coordinates:



Point on a Cartesian and polar axis

Implicit Conversion

- To make your Point class easy to use, you would like other programmers to be able to write the following code:

```
Point p = new Point(5, 5);
Polar pl = p;
```

- In the above example, we made an implicit cast (no parentheses necessary) from a Point to a Polar. This is okay, since the conversion can occur without data loss. Further, it's encouraged because it makes your class easier to use. To enable this, you must add the following to your Point class:

```
public static implicit operator Polar(Point p) {
    double radius = Math.Sqrt(p.X * p.X + p.Y * p.Y);
```

```

    double angle = Math.Atan2(p.X, p.Y);
    return new Polar(radius, angle);
}

```

- The above method returns a Polar given a Point.
- The operator keyword applied to a class name (Polar) indicates that the method represents a cast to that class.
- The **implicit** keyword allows the cast to happen without parenthetical syntax.

Explicit Conversion

- You also know programmers may want to convert your Point to a scalar (single value). When this happens, they are just interested in the point's distance from the origin in Cartesian space, (same as the radius of the Polar type).
- In this case, the Point *loses data* (the angle), so the cast should be explicit:

```

Point p = new Point(5, 5);
Double radius = (double) p;

```

- The code to implement the explicit conversion is:

```

public static explicit operator double(Point p) {
    return Math.Sqrt(p.X * p.X + p.Y * p.Y);
}

```

- The **explicit** operator indicates that the cast must use parenthetical syntax.

Activity - Making Convenient Types

Expand the Employee class:

- Override the Equals method, GetHashCode method, == operator, and != operator for an employee.
 - Two employees are considered the same if their IDs are the same.
- Test by comparing 2 employees in your Program class.

If You Have Time

Read the following section on defining custom value types, then do the exercise at the end of that section.

Defining Custom Value Types

99% of the time, if you need to create a custom type then you should use a class. Only create a struct if the reduced functionality is worth the increased performance.

Why create custom value types instead of a class?

- Value types are lighter-weight
- Results in faster code
- Custom value type is a `struct`

Structs vs. Classes.

- Structs lack some features:

Feature	class	struct
Type	Reference	Value
Fields, properties, methods, operators, custom casting?	Yes	Yes
Constructors	Yes	Yes, but parameters are required
Inheritance	Yes	No
Initializers*	Yes	No
Finalizer	Yes	No



* Structs allow for object initializers, but not instance initializers

Instance initializers, which assign non-default values directly to fields in a struct definition, are not permitted. For example, in the following code, "12345" is not permitted:

```
public struct MyStruct
{
    private int _myIntField = 12345;
    public int MyIntProperty
    {
        get { return _myIntField; }
        set { _myIntField = value; }
    }
}
```

However, when creating a new struct, the following syntax (object initializer syntax) is permitted:

```
MyStruct instance = new MyStruct() { MyIntProperty = 5678 };
```

- The default constructor is automatically defined for structs and is not customizable.
- This allows structs to be created faster than Classes
- Generally prefer classes
- Use structs when
 - Data operates like a primitive type
 - Data is small
 - Data is normally immutable
 - Missing class features are not required

Define:

```
struct Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public Point(int x, int y): this() {
        this.X = x;
        this.Y = y;
    }
}
```

Use:

```
Point p = new Point(5, 8);  
Console.WriteLine("The point is at ({0},{1}).", p.X, p.Y);
```

Activity - Create a custom Value Type

1. Define a struct to represent a complex number
 - A complex number is $v + iw$ where v and w are real numbers and i represents the positive square root of -1
 - A complex number has both a real and an imaginary part
2. Provide a constructor
3. Override the `ToString` (inherited from `object`)
 - This method is called when instances of your type must be converted to strings
4. Override any operators that you choose to
5. Test your new struct

Lesson 3: Promoting Flexible and Reusable Code

Introduction.....	3•3
By the end of this lesson you will learn.....	3•3
Creating Class Hierarchies.....	3•5
Example Hierarchy	3•5
Forming Class Hierarchies	3•6
Inheritance Example.....	3•6
Accessing the Base Class	3•6
Accessing the Base Constructor.....	3•7
Replacing Base Classes	3•7
Polymorphism Example.....	3•8
Hiding Inherited Behavior	3•8
Understanding New	3•8
Replacing Inherited Behavior.....	3•9
Understanding Override.....	3•10
Replacing Object Behavior	3•11
Choosing new vs. override.....	3•12
Deferring Implementation.....	3•13
Animal Example	3•14
Terminating Class Hierarchies	3•15
Defining Stand-Alone Classes	3•15
Activity: Inheritance	3•17
Specifying Behavioral Contracts.....	3•19
Specifying Contracts.....	3•19
Implementing Interfaces.....	3•20
Multiple Interfaces	3•21
Casting to an Interface.....	3•22
Combining Cast and Check.....	3•22
Extending Interfaces	3•23
Overriding Implementations	3•23
Replacing Implementations.....	3•24
Resolving Name Conflicts	3•25

Setting Default Behavior	3•27
Activity: Interfaces	3•28

Promoting Flexible and Reusable Code

Introduction



Why reuse code?

- _____
- _____
- _____
- _____
- _____

Object-oriented languages like C# allow you to reuse code through hierarchical relationships. This feature is commonly referred to as **inheritance**. Using inheritance, code applying generally can be included higher in the hierarchy so it is shared by all classes lower in the hierarchy.

In addition to inheriting code, a class may also sign a contract that requires implementing certain kinds of behavior (properties and methods). In C#, these contracts are known as **interfaces**.

Consuming code knows what methods and properties are available to use. Code that consumes classes of an interface can then be reused with any class implementing that same interface.

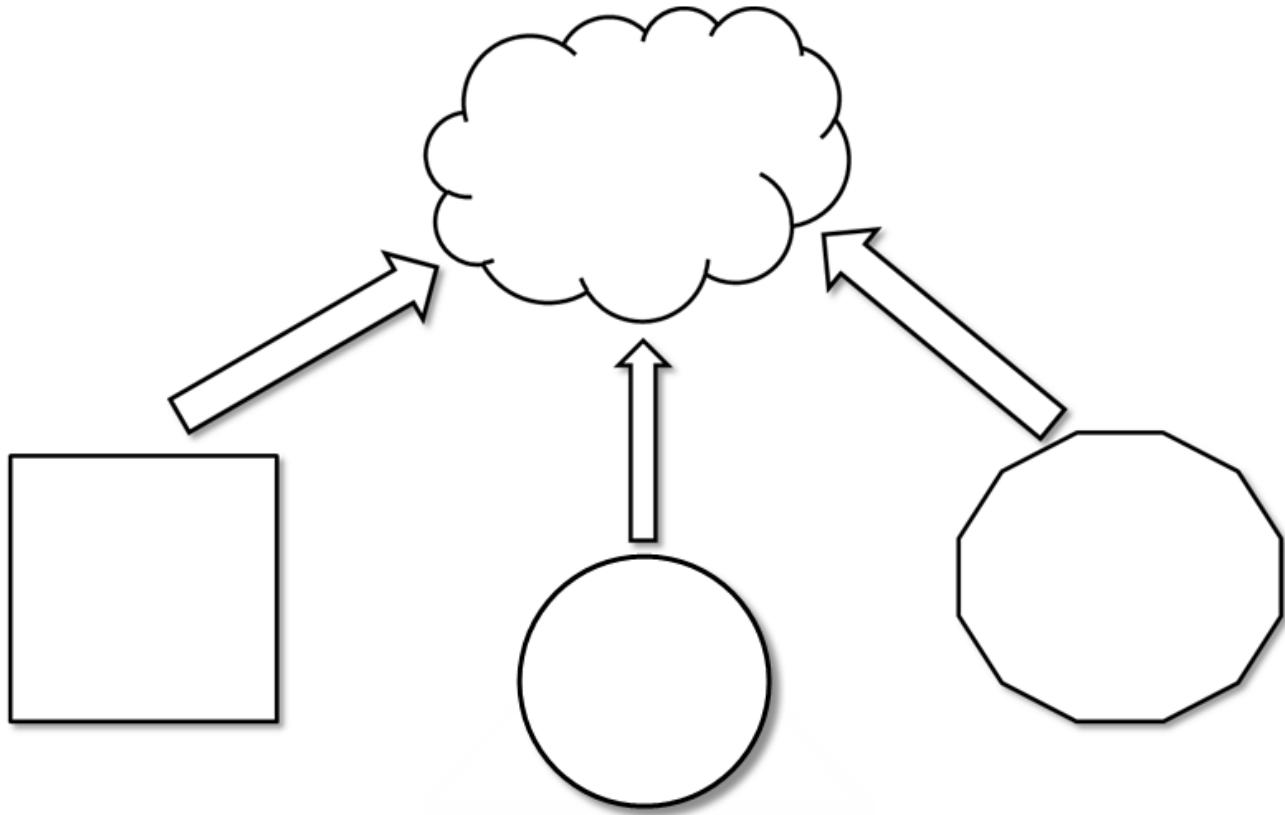
By the end of this lesson you will learn...

- Concepts
 - Explain the importance of code reuse
 - Explain how code can be reused through an inheritance hierarchy
 - Explain how to reuse code from a base class inside a derived class
 - Describe what is inherited from a parent class
 - Explain how consuming code can take advantage of polymorphism

- Explain why you might want to change the behavior of a base class method/property in a derived class
- Describe the difference between using the new and override keywords to change the behavior of a base class method/property
- Explain why you might mark a class as abstract and the consequences of doing so
- Describe the significance of the Object class
- Explain why you might seal a class
- Describe the differences between a regular class and a static class and when a static class is appropriate
- Explain the advantages of the template design pattern
- Explain the difference between a private field and a protected field
- Explain why Epic uses interfaces
- Explain the differences between a class and an interface
- Describe the differences between using the is and the as keyword
- Explain what impact an extension has on an interface
- Describe the differences between an explicitly implemented interface and an implicitly implemented interface
- Algorithms
 - Create a new class that inherits from an existing class
 - Define an interface
 - Implement an interface
 - Explicitly implement an interface
 - Safely cast an object to an interface
 - Determine which method will be called given an inheritance hierarchy and a method call on an object reference
 - Determine which method will be called given an inheritance hierarchy and a method call on an object from a class that implements an interface
 - Examine the metadata of a parent class

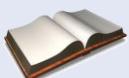
Creating Class Hierarchies

Example Hierarchy



- Classes can inherit features from one parent class.
- All members inherited (excluding constructors)
- Hierarchical relationships formed
 - Circle _____ from Shape
 - Shape is the _____ class
 - Circle is the _____ class

You can create classes that derive from other classes. This is called **inheritance**. Through inheritance, hierarchical relationships can be created. These relationships are often referred to as "is-a" relationships.



For example, a circle **is a** shape. We can represent this relationship in code by building a **base class** called `shape`, and then creating a second class called `circle` that derives or **inherits** from `shape`. `Circle` is referred to as the **derived class**.

Forming Class Hierarchies

Syntax:

```
class DerivedClass : BaseClass
{
    ...
}
```

Inheritance Example

```
public class Shape
{
    public int X { get; set; }
    public int Y { get; set; }

    public void Draw()
    {
        Console.WriteLine("Drawing shape");
    }
}

public class Circle : Shape
{
    public int Radius { get; set; }
}
```

To use:

```
Circle myCircle = new Circle();
myCircle.Draw();
```

In the above example, `myCircle` has a `Draw` method because it inherited the `Draw` method from the `Shape` class.

Accessing the Base Class

- Use the `__` keyword:

```
base.MethodName();
```

Accessing the Base Constructor

Class constructors are **not** inherited. If a base class has a default constructor, then that constructor is called implicitly. If the base does not have a default constructor, a constructor for the base class must be explicitly called with the required parameters.



A class's default constructor is a constructor without parameters. If no such constructor is explicitly defined, one will be implicitly defined if no other constructors for the class exist.

```
public class Shape
{
    public int X { get; set; }
    public int Y { get; set; }

    public Shape(int X, int Y)
    {
        this.X = X;
        this.Y = Y;
    }

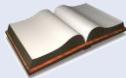
    public void Draw()
    {
        Console.WriteLine("Drawing shape");
    }
}

public class Circle : Shape
{
    public int Radius { get; set; }
    public Circle(int X, int Y) : base(X, Y) { }
}
```

Replacing Base Classes

- Derived class may replace base class in code
- Known as _____

Polymorphism is an object-oriented concept where one type may appear and be used as another type. In general, a class lower in the inheritance hierarchy may be used as a class higher in the hierarchy. In other words, one type has multiple forms.



An example is our shape and circle example. A circle is a shape but the shape is too general to accurately describe what a circle is. The circle class is more specific and thus can model what a circle truly is.

Polymerism Example

Method using Shape:

```
private static void TestShape(Shape s)
{
    s.Draw();
}
```

Pass a Circle instead of a Shape:

```
Circle myCircle = new Circle();
TestShape(myCircle);
```

Hiding Inherited Behavior

- Use __ in derived class
- Which method used depends on _____

Sometimes a derived class needs to implement its own version of a base class member. For this situation, use the **new** access-modifier.

Understanding New

It is important to understand how the **new** keyword works in terms of masking methods. Consider the following definition of a shape and a circle. First, Shape implements the Draw method. Next, Circle inherits from Shape and implements a new version of Draw:

```
public class Shape
{
    public void Draw()
    {
        Console.WriteLine("Drawing shape");
    }
}

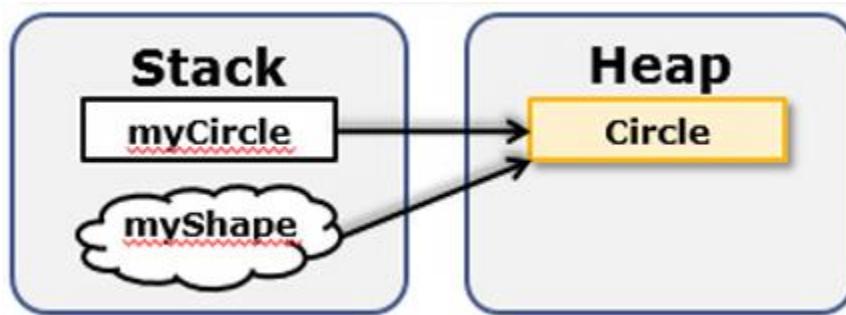
public class Circle : Shape
{
    public new void Draw()
```

```
{
    Console.WriteLine("Drawing circle");
}
}
```

Now consider the following scenario. One circle object is created and assigned to a variable of type circle. Next, a variable of type shape is created, and is assigned the same instance:

```
Circle myCircle = new Circle();
Shape myShape = myCircle;
```

A graphical representation of this scenario is given in the following diagram:



Given this scenario, what is the output of the following code?

- `myCircle.Draw();` _____
- `myShape.Draw();` _____



Summary of `new`:

- The data type of the reference, (**not** the actual instance), dictates which implementation to use.

Replacing Inherited Behavior

- Use _____ in base and _____ in derived class.
- Which used depends on _____

To replace the base class behavior with code from the derived class, the base class member must be marked **virtual** and derived class member must be marked **override**.

Understanding Override

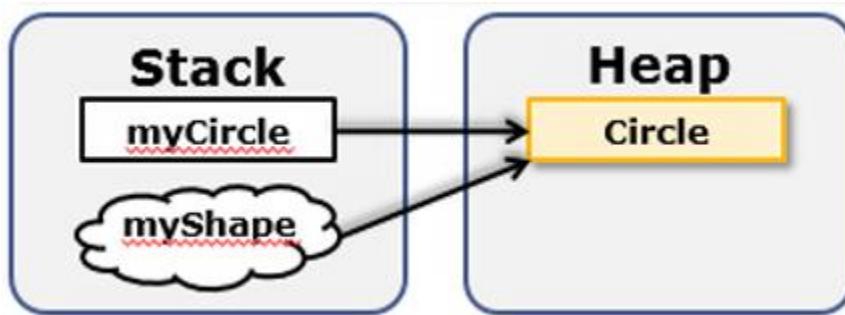
Rather than masking behavior using `new`, it is more common to replace it using `override`. This way the same method of an instance is called independent of how it is referenced.

Once again, consider a shape and a circle, but this time the draw method in the shape is virtual, and the draw method of Circle overrides it:

```
public class Shape  
{  
    public virtual void Draw()  
    {  
        Console.WriteLine("Drawing shape");  
    }  
}  
  
public class Circle : Shape  
{  
    public override void Draw()  
    {  
        Console.WriteLine("Drawing circle");  
    }  
}
```

Consider the same scenario as before. One circle instance is created and assigned to a variable of type Circle. Next, a reference of type shape is created and assigned the same instance of circle:

```
Circle myCircle = new Circle();  
Shape myShape = myCircle;
```



What is the output of the code this time?

- `myCircle.Draw();` _____

- `myShape.Draw(); _____`

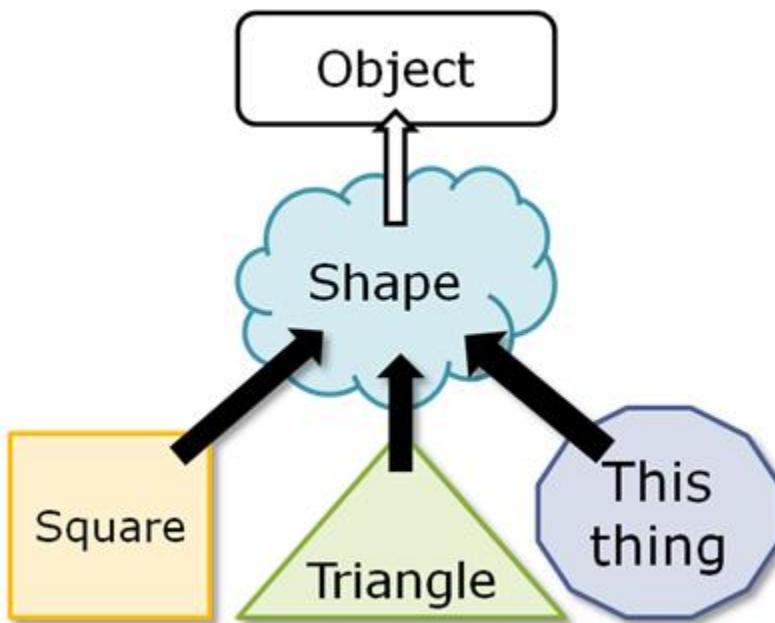
**Summary of override:**

- The data type of the instance, (not the reference), dictates which implementation to use.
- The base class must allow override of individual properties and methods using the keyword **virtual**.

Replacing Object Behavior

- All classes derive from _____
- Override object methods to change default behavior

All classes derive from the Object class. Everything in C# is an object, even value types. The Object class provides methods all subclasses can use and override.



Equals()	Are two objects equivalent?
GetHashCode()	Provides a hash code used to index a hash set or hash table.
GetType()	Returns the type of the object.
ToString()	Creates a string representation of an object. ToString is called implicitly whenever a string representation of an object is required.

Overriding ToString:

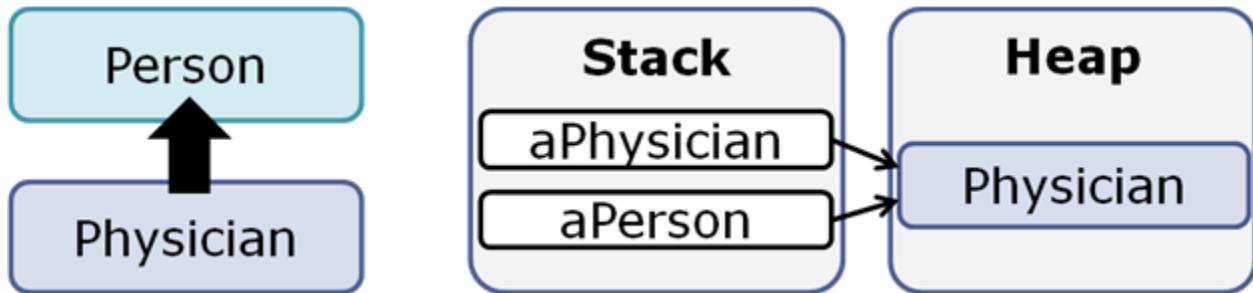
```
public class Circle : Shape
{
    public override string ToString()
    {
        return "Circle";
    }
}
```

Choosing new vs. override

It can be difficult to decide between overriding or creating a new version of base class' member.
Ask the following question:

1. Is the base class' member virtual?
 - If not (and it can't be changed) then **new** is your only option.
 - If so, does the result depend on how the object is referenced?
 - If no, then **override** the member.
 - If yes, **create** a new member.

Consider the scenario where your application is modeling human interaction. You have a class Person and want to address a person by what they would naturally go by in a given social scenario. Now suppose that you have a class Physician that derives from Person:



The Person class is defined as follows:

```
class Person
{
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public virtual string FullName
    {
        get
    }
}
```

```
{  
    return String.Format("{0} {1}", FirstName, LastName);  
}  
}  
public virtual string GoesBy  
{  
    get { return FirstName; }  
}  
}
```

What would be the most natural way to implement FullName and GoesBy? Think of a person reference as how friends and family would interact with the physician vs. the Physician reference as how patients and co-workers would do so.

```
class Physician : Person  
{  
    public _____ string FullName  
    {  
        get  
        {  
            return String.Format("Dr. {0}", base.FullName);  
        }  
    }  
    public _____ string GoesBy  
    {  
        get  
        {  
            return String.Format("Dr. {0}", LastName);  
        }  
    }  
}
```

Deferring Implementation

- Base class may not know how to implement a method or property
- Use _____ to defer implementation to derived class

An abstract method is a method that has no implementation. All abstract methods have no method body. Any class that derives from a class containing an abstract method must implement that method.

Any class that contains an abstract method must be marked as abstract as well. Abstract classes may not be instantiated. They provide a basic framework for other classes to derive from.

If shape is declared as an abstract class:

```
public abstract class Shape {  
    public abstract void Draw();  
}
```

Then instances of shape cannot be created:

```
// Will not compile  
Square myShape = new Shape();
```

**Why create an abstract class?**

•

•

•

Animal Example**Abstract class with implemented constructor:**

```
public abstract class Animal  
{  
    protected string _name;  
    public Animal(string name)  
    {  
        _name = name;  
    }  
    public abstract void MakeSound();  
}
```

Derived classes that implement MakeSound:

```
public class Cat : Animal  
{  
    public Cat(string name)  
        :base(name)  
    { }  
    public override void MakeSound()  
    {  
        Console.WriteLine("Meow");  
    }  
}
```

```
}
```

```
public class Dog : Animal
```

```
{
```

```
    public Dog(string name)
        :base(name)
    { }
```

```
    public override void MakeSound()
    {
        Console.WriteLine("Woof");
    }
}
```

Method that takes an animal and calls MakeSound:

```
public static void PokeAnimal(Animal animal)
```

```
{
```

```
    animal.MakeSound();
}
```

Terminating Class Hierarchies

- Mark class as _____

A class may be marked as `sealed` if need be. You cannot inherit from a sealed class. Use this when it is necessary to terminate the class hierarchy.

```
public abstract class Shape {}
```

```
public sealed class Circle : Shape {}
```

```
// Will not compile
```

```
public class Sphere : Circle {}
```



Why seal a class?

- _____
- _____

Defining Stand-Alone Classes

- Mark class and members _____

- Cannot create _____ of class
- Cannot be a _____ or _____ class

To Define:

```
public static class CupConversions
{
    public static int CupToOz(int cups)
    {
        return cups * 8; // 8 ounces in a cup
    }
    public static double CupToPint(double cups)
    {
        return cups * 0.5; // 1 cup = 1/2 pint
    }
    public static double CupToPeck(double cups)
    {
        return cups / 32; // 8 quarts = 1 peck
    }
    public static double CupToBushel(double cups)
    {
        return cups / 128; // 4 pecks = 1 bushel
    }
}
```

To use:

```
double ozVal = CupConversions.CupToOz(8);
Console.WriteLine("8 cups = {0} oz.", ozVal);
```



These sorts of classes should be used sparingly and for special cases only. When they are used they should have a clear purpose.



Static classes may contain static constructors

- Even though a static class may not be instantiated, it may still contain a static constructor.
- A static constructor is a method that is automatically called the first time a class is accessed during the life of a process. Typically they are used to initialize static fields and properties to values that can only be determined at run time.
- Static constructors may not contain parameters and do not have access modifiers
- Static constructors may not be called directly
- There can only be one static constructor per class.
- A class does not need to be static to contain a static constructor.

Activity: Inheritance

- Continue with the CompanySim solution
- Make the Employee class an abstract class
- Follow Epic's design pattern to add a method to the Employee class that can be overridden by derived classes
 - Add a public method that returns void to the Employee class, **DoWork**. In DoWork, write the message "Working hard" to the console.
 - Add a protected virtual method that returns void, **DoWorkCore**. In this method, you can write the message "or hardly working?" to the console.
 - Have the DoWork method invoke DoWorkCore.



This is an example of the template design pattern. Consider using this when

- There is some code that the base class **should always execute**.
- You want to give derived classes the option of executing code **in addition to** what the base class executes.

- Override the ToString method of an Employee by returning the "<Name> - <ID>"
- Define a ProjectManager class that inherits from Employee.
 - ProjectManager have a base salary of \$2,000
 - Override the DoWorkCore method to display "On a go-live."
- Define a Executive class that inherits from Employee.

- Executives have a base salary of \$3,000
- Override the DoWorkCore method to display "Signing prospective customers."
- Provide constructors for ProjectManager and Executives that accept the Name, ID, Tenure, and App
- Test the ProjectManager and Executive classes

Specifying Behavioral Contracts



Why Specify Contracts?

-

Specifying Contracts

- Specified using _____
- List properties/methods that signing classes must implement

Syntax:

```
access-modifiers interface IName
    : [base-interface list]
{
    [Property / method stubs]
}
```

Example:

```
interface IStorable {
    void Read();
    void Write(object obj);
    int Status { get; set; }
}
```

- Members are implicitly _____
- Members are _____

Many times there will be a set of features that you want to publish or have classes implement. This is accomplished by creating an **interface**. The interface serves as a contract between the implementing class and the client class. Classes may implement multiple interfaces. All interface names should begin with an "I".

Interface members are implicitly public and members are completely abstract. As such they contain only method declarations and property stubs, and cannot be instantiated. Only the implementing classes can be instantiated.



Can an interface be instantiated?

•

Implementing Interfaces

- How many classes can you inherit from? _____
- How many interfaces can you implement? _____

```
public class ClassName : IName1, IName2
{
    // Implement methods and properties
    // from all interfaces
}
```



Write in Workbook

What if you want to implement some, but not all properties and methods?

•

Example Interface:

```
interface IStorable
{
    void Read();
    void Write(object obj);
    int Status { get; set; }
}
```

Example Implementing Class:

```
public class Document : IStorable
{
    public Document(string s) {
        Console.WriteLine(
            "Creating document with: {0}", s);
    }

    public void Read() {
        Console.WriteLine("Read Method for IStorable");
    }
}
```

```
}

public void Write(object o) {
    Console.WriteLine("Write Method for IStorable");
}

public int Status { get; set; }
}
```

Multiple Interfaces

Example interfaces:

```
interface IStorable
{
    void Read();
    void Write(object obj);
    int Status { get; set; }
}

interface ICompressible
{
    void Compress();
    void Decompress();
}
```

Example implementing class:

```
public class Document
    : IStorable, ICompressible
{
    public Document(string s)
    {
        Console.WriteLine("Creating document with: {0}", s);
    }

    public void Read()
    {
        Console.WriteLine("Read Method for IStorable");
    }

    public void Write(object o)
    {
        Console.WriteLine("Write Method for IStorable");
    }

    public int Status { get; set; }

    public void Compress() { }

    public void Decompress() { }
}
```



Interfaces

- Should be small and focused
- Can use with structs and classes

Casting to an Interface

- Like casting to any other type
- Syntax:
 `IName objA = (IName) objB;`
- If `objB` does not implement `IName`, an `InvalidCastException` is thrown at runtime

To protect against runtime casting exceptions, use the `is` operator to test if an object implements an interface.

The `is` operator returns a Boolean value; true if the object implements the interface and false if it does not.

```
if (objB is IName)  
{  
    IName objA = (IName) objB;  
}
```



Write in Workbook

How many casts are in the previous example?

- _____

Combining Cast and Check

- Operator `as` combines the check and cast.
- If `doc` is not an `IStorable`, then `isDoc` will point to null:

```
Document doc = new Document("Test Document");  
IStorable isDoc = doc as IStorable;  
if (isDoc != null)  
{
```

```
    isDoc.Read();  
}  
else  
{  
    Console.WriteLine("Could not cast to IStorable");  
}
```



Write in Workbook

How many casts are in the previous example?

- _____

Extending Interfaces

- Syntax just like inheritance

```
interface ICompressible  
{  
    void Compress();  
    void Decompress();  
}  
  
interface ILoggedCompressible  
    : ICompressible  
{  
    void LogSavedBytes();  
}  
  
public class Document  
    : IStorable, ILoggedCompressible  
{  
    // IStorable, ILoggedCompressible implementation  
}
```



In this example, methods from both ICompressible and ILoggedCompressible are required.

Overriding Implementations

- Implementing class marks interface methods as _____
- Derived classes can override

Interface:

```
interface IStorable
{
    void Read();
}
```

Implementing class:

```
public class Document
    : IStorable
{
    public virtual void Read() {
        Console.WriteLine("Implement Read");
    }
}
```

Class derived from implementing class:

```
public class Note
    : Document
{
    public override void Read() {
        Console.WriteLine("Override Read");
    }
}
```

Replacing Implementations

- Can also mask with **new**

Interface:

```
interface IStorable
{
    void Read();
    void Write();
}
```

Implementing class:

```
public class Document
    : IStorable
{
    public virtual void Read()
```

```
{  
    Console.WriteLine("Implement Read");  
}  
public void Write()  
{  
    Console.WriteLine("Implement Write");  
}  
}
```

Class derived from implementing class:

```
public class Note  
    : Document  
{  
    public override void Read()  
    {  
        Console.WriteLine("Override Read");  
    }  
    public new void Write()  
    {  
        Console.WriteLine("New Write");  
    }  
}
```



What is the output?

IStorable myNote = new Note();

- myNote.Read(); _____
- myNote.Write(); _____

Resolving Name Conflicts

Name conflicts can occur when implementing two interfaces that have methods or properties that share the same name.



The only way to access explicitly implemented interfaces is through a cast to the interface.

Interfaces:

```
interface IStorable
{
    void Read();
}

interface ITalk
{
    void Read();
    void Talk();
}
```

Implementing Class:

```
public class Document
    : IStorable, ITalk
{
    public void Read()
    {
        Console.WriteLine("Which interface does this come from?");
    }

    public void Talk()
    {
        Console.WriteLine("ITalk.Talk");
    }
}
```



Which interface does Read implement?

- _____

The conflict can be resolved by explicitly implementing the methods:

```
public class Document
    : IStorable, ITalk
{
    void IStorable.Read()
    {
        Console.WriteLine("IStorable.Read");
    }

    void ITalk.Read()
    {
        Console.WriteLine("ITalk.Read");
    }

    public void Talk()
    {
```

```
        Console.WriteLine("ITalk.Talk");
    }
}
```



What is the output?

```
Document doc1 = new Document();
IStorable doc2 = doc1;
ITalk doc3 = doc1;
```

- doc3.Read(); _____
- doc2.Read(); _____
- doc1.Read(); _____



You must access explicitly-implemented members through the interface.

Setting Default Behavior

To choose one of the conflicting interface methods as the default when accessing the instance through its class reference rather than through the corresponding interface, make that method public. Implement all other methods by the same name explicitly:

```
public class Document
    : IStorable, ITalk
{
    void IStorable.Read() {
        Console.WriteLine("IStorable.Read");
    }
    public void Read()
    {
        Console.WriteLine("ITalk.Read");
    }
    public void Talk()
    {
        Console.WriteLine("ITalk.Talk");
    }
}
```



What is the output?

```
Document doc1 = new Document();  
IStorable doc2 = doc1;  
ITalk doc3 = doc1;
```

- doc3.Read(); _____
- doc2.Read(); _____
- doc1.Read(); _____

Activity: Interfaces

- Continue with the CompanySim solution
- Define an interface called ITechnical. The interface has a method for WritingCode.
- Define a new class called Developer which inherits from Employee and implements ITechnical.
 - Override the DoWorkCore method to call the WritingCode method.
- Test the Developer class

Lesson 4: Working with Collections

The Big Picture.....	4•4
Introduction.....	4•5
By the end of this lesson you will learn.....	4•5
Using Fixed-Size Collections.....	4•7
Declaring and Using Arrays.....	4•7
Syntax	4•7
Initializing Elements.....	4•8
Iterating using foreach.....	4•10
Passing Parameters	4•11
Using System.Array	4•15
System.Array for One Dimensional Arrays.....	4•15
Complex Arrays.....	4•18
Using Multidimensional Arrays	4•18
Using Jagged Arrays.....	4•19
Activity: Using Arrays	4•20
Using Generic Collections.....	4•21
Lists	4•25
Queues and Stacks	4•27
Dictionaries.....	4•32
Dictionary< TKey, TValue > Members.....	4•32
Converting to an Array	4•37
Creating Custom Collections.....	4•40
Collection Interfaces.....	4•40
Indexers.....	4•41
Custom Generic Classes	4•42
Activity: Custom Generic Collection.....	4•42
If you have time	4•50
Activity: Collections.....	4•50
If you have time	4•50

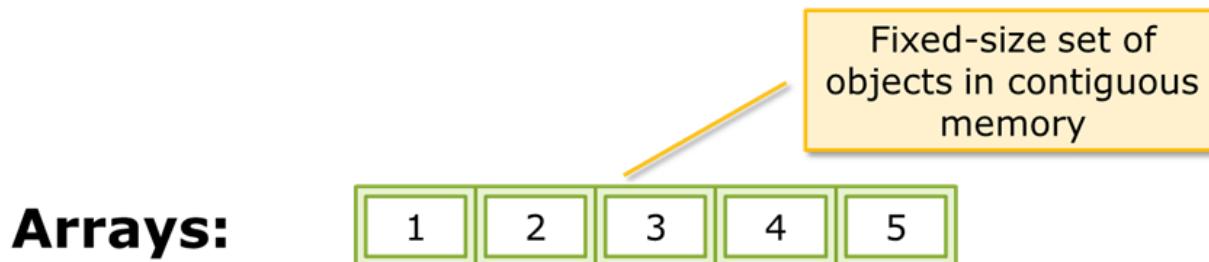
Working with Collections

The Big Picture

In C#, there are several different classes to choose from to represent a collection. In this lesson, collections will be grouped into two buckets: Arrays and Generic Collections.

Arrays are light-weight structures used in scenarios with collections of a fixed-size. They are relatively simple, but also limited in functionality.

Generic collections provide many nice features, such as resizing. They are more flexible and can be used in a variety of circumstances.



Generic Collections:

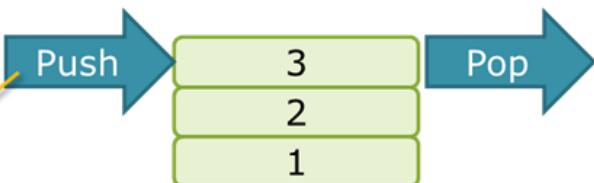
List:

Resizable Array



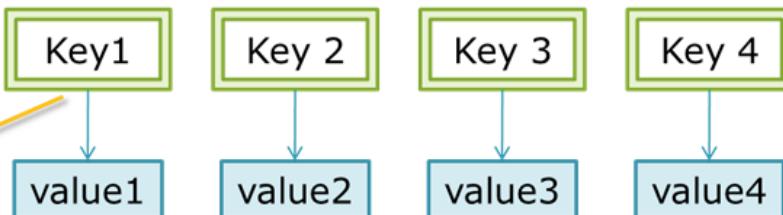
Stack:

Last-in-first-out



Dictionary:

Random-access key-value pairs



Introduction

A "collection" generally refers to a group of elements organized in a list or table. When there are thousands of objects, a collection can be used so that the programmer doesn't have to keep track of a variable name for each object. Different types of collections have different properties and can offer significant organizational and performance advantages over others, so choosing the right collection to group objects is paramount.

By the end of this lesson you will learn...

- Concepts
 - Describe the advantages of using arrays over generic collections
 - Describe the advantages of using a generic collection over an array
 - Describe why you might choose to use for syntax or foreach syntax
 - Interpret some of the basic methods and properties of arrays
 - Explain the difference between a collection and a generic collection
 - Explain the effect of the where keyword on a generic collection
 - Explain the properties and methods of some of the commonly used generic collections
 - List
 - Dictionary
 - Explain the advantages of a custom collection
 - Explain the advantages of a custom generic collection
 - Explain the properties and methods of some of the commonly implemented generic interfaces
 - Explain what happens when the indexer syntax is used on an instance of a custom collection
- Algorithms
 - Declare and instantiate arrays
 - Use for syntax to loop over the elements of an array
 - Use foreach syntax to loop over the elements of an array
 - Appropriately use the params keyword in a method and call methods that use the params keyword
 - Convert a generic collection to an array

- Implement and use an indexer for a custom collection
- Create a custom generic collection
- Implement common generic collection interfaces
- Create a generic method

Using Fixed-Size Collections

Sometimes you are dealing with a known amount of data or you are interested in a specific number of data points. In these cases you can use an array to keep track of the data. C# arrays have the following characteristics:

- An array is an indexed collection of elements
- It has a fixed number of elements
- Elements are indexed from 0 to n-1, where n is the number of elements in the array
- All elements are of the same type (which can be value or reference)
- The array itself is a reference type
- Arrays are instances of the System.Array class

Declaring and Using Arrays

Arrays are declared by including square brackets after the data type they will contain. For example,

```
int [] myArray;
```

Declares an array named myArray holding ints.

An array can be instantiated during declaration by using the following syntax:

```
type[] myArray = new type[size];
```

where type is the data type of the elements and size is the size of the array.

Arrays are created with a fixed length. If at any point there need to be more elements in the array than the length of the array, a new array must be created. The Length property contains the number of elements.

Elements are accessed by including their index in square brackets: arrayName [index]

Syntax

Syntax - Declare

```
type[] arrayName; //array does not exist yet
```

Syntax - Declare and Instantiate

```
type[] arrayName = new type[size]; //allocating slots for 0 to size-1
```

Example

```
int[] intArray = new int[5];
```

Syntax - Accessing Array Elements

```
arrayName [#]
```

Example

```
int val = intArray[2];
```



Try it yourself: Declare and use an array

Create an array called intArray that holds 5 ints.

```
int[] intArray = new int[5];
```

Loop over the indices of the array. Set each element to double the index.

```
for (int i = 0; i < intArray.Length; i++)  
{  
    intArray[i] = i * 2;  
}
```

Find the middle element of the array and write it out.

```
int midPos = intArray.Length / 2;  
  
Console.WriteLine("The middle element is {0}",  
intArray[midPos]);
```

Initializing Elements



Try it yourself: Look at an array's default elements

Take a look at your code using intArray. After instantiating the array but before populating it, add a loop to write out all of the elements.

```
for (int i = 0; i < intArray.Length; i++)  
  
{  
  
    Console.WriteLine("The element at index {0} is {1}", i,  
    intArray[i]);  
  
}
```

Run this code. Based on what you see, what is the initial value of elements in an int array?

In general, there are two cases for the default value of an element in an array.

By default, element values are:

- Default value for value types (e.g., 0 for int or false for bool)
- null for reference types

However, they can be initialized at instantiation time with specified values.

Syntax

```
type[] arrayName = new type[size]  
  
{value1, value2, ...}
```

or

```
type[] arrayName = {value1,value2,...}
```

```
int[] intArray = {1, 2, 3, 4, 5, 6, 7};  
  
Employee[] empArray = {  
    new Employee(15146),  
    new Employee(53593)  
};
```



Try it yourself: Populate an array during instantiation

Create an array of strings called movies. Instantiate it containing "casino royale", 300, and "the dark knight".

```
string[] movies = { "casino royale", 300, "the dark knight" };
```

What happens?

Include quotes around 300 to make it a string.

```
string[] movies = { "casino royale", "300", "the dark knight" };
```

What property can be used to see the size of this array?

What is the size of this array?

Thinking back to our Car class from earlier, create an array called garage that contains two Cars. Recall that the constructor signature is

```
public Car(string VIN, double TankCapacity, double MilesPerGallon)
```

```
Car[] garage = { new Car("1050A", 30.2, 24.2), new Car("2050B", 24.8, 42.3) };
```



If the array is an array of objects, the objects must be created.

Iterating using foreach

Arrays can be iterated over using the `for` loop or the `while` loop. For loops are more common when looping over the entire array or a known subset of the elements of the array. While loops are more common when looping over an unknown subset of the array or when searching for certain criteria.

In addition, `System.Array` implements the `IEnumerable` interface, which lets you iterate over the elements in a `foreach` loop. `foreach` loops are often an easier way to loop over the data in an array. You don't need to know anything about the contents of the array and you don't need to do any bounds checking. However, you don't know which index is being accessed on each iteration of the loop and consequently can't make changes to the array.

Syntax

```
foreach (type element in collection) statement;
```

Example

```
foreach (int currin intArray)
{
    Console.WriteLine(curr.ToString());
}
```



Try it yourself: Loop over an array using `foreach`

Look back to the `intArray` created earlier. After populating the array, write a `foreach` loop to sum the elements. Then write out the sum.

```
int sum = 0;

foreach (int curr in intArray)
{
    sum += curr;
}

Console.WriteLine("The sum is {0}", sum);
```



Take 30 seconds to think about these questions.

What are the advantages of `foreach`?

- _____

What are the disadvantages?

- _____

Passing Parameters

There will be times where you need to pass in multiple pieces of information to a method. If the information is in an array, the method signature can include an array in place of a variable. For example,

```
public static int FindLargest(int[] numbers)
```

is a method which takes in an array of ints.

However, this does require consuming code to construct an array to pass. For additional flexibility, the **params** keyword is used. By including params before the array parameter, consuming code can either pass in an array or pass in multiple comma-separated arguments. The .NET framework will construct an array for the method to use from the arguments passed. Each argument still needs to match the array type.

Syntax

```
access-modifier return-type methodName(params type[] paramname) { ... }
```



Try it yourself: Create a method that uses params

Write the method signature from the example above, which takes in an array of ints and returns the largest one.

```
public static int FindLargest(int[] numbers)
```

If the method is called with bad input, we want to return null. How can you specify that the return type is a nullable int?

Incorporate these changes.

```
public static int? FindLargest(int[] numbers)  
{  
    ...  
    return null;
```

In the method body, first check that numbers isn't null and that it contains at least one element.

```
if (numbers != null && numbers.Length > 0)
```

Set max to the first element in the array. Then use a foreach loop to compare it with all elements in the array. Return max.

```
int max = numbers[0];  
  
foreach (int num in numbers)  
{  
    if (num > max)  
    {  
        max = num;  
    }  
}  
  
return max;
```



Try it yourself: Create a method that uses params (cont.)

Why is it okay to use a foreach loop here?

Call FindLargest with our good friend intArray. Write out the result to confirm it's returning the largest number.

```
Console.WriteLine("Largest number: {0}",  
FindLargest(intArray));
```

Now try calling FindLargest with no input.

```
FindLargest();
```

What happens?

Call FindLargest with 4, 8, 15, 16, 23, and 42.

```
FindLargest(4, 8, 15, 16, 23, 42);
```

What happens?

What do these errors mean?

How could we allow for these two use cases and still allow for an int array to be passed in?

Update the code accordingly and retry the two previous use cases.

```
public static int? FindLargest(params int[] numbers)  
{  
    ...  
    FindLargest();  
  
    FindLargest(4, 8, 15, 16, 23, 42);  
}
```

Using System.Array

In addition to what you've seen so far, arrays have several other useful methods and properties. Since every array is an instance of the class `System.Array`, arrays have the same API regardless of what they're being used for. Below are a few of the most common members.

<code>Array.Clear()</code>	(static) Sets the specified range of elements to the default value for the type (either 0, false, or null)
<code>Array.Copy()</code>	(static) Copies a specified range of elements to another array
<code>Length</code>	Returns the length (size) of the array
<code>Rank</code>	Returns the number of dimensions in the array

System.Array for One Dimensional Arrays

<code>Array.IndexOf()</code>	(static) Returns the index of the first instance of a value in an array
<code>Array.LastIndexOf()</code>	(static) Returns the index of the last instance of a value in an array
<code>Reverse()</code>	Reverses the order of the specified range of elements in an array
<code>Array.Sort()</code>	(class) Sorts the values of an array (assuming elements are sortable)



Try it yourself: Use the System.Array API

Write a method called DisplayArray that takes in an array of objects and writes out the contents.

```
public static void DisplayArray(object[] arrayToDisplay)  
{  
    foreach (object o in arrayToDisplay)  
    {  
        Console.WriteLine(o.ToString());  
    }  
}
```

Create a string array called poignantArray containing "Hope", "Defines", "What", "Is", "Man".

```
string[] poignantArray = { "Hope", "Defines", "What", "Is",  
    "Man" };
```

Write out the contents of poignantArray.

```
Console.WriteLine("Poignant array: ");  
DisplayArray(poignantArray);
```

Write out the first index and the last index where "Hope" is found in poignantArray.

```
Console.WriteLine("First occurrence of Hope: {0}",  
    Array.IndexOf(poignantArray, "Hope"));  
  
Console.WriteLine("Last occurrence of Hope: {0}",  
    Array.LastIndexOf(poignantArray, "Hope"));
```

Why are these the same?



Try it yourself: Use the System.Array API (cont.)

Reverse the elements in poignantArray then write it out again.

```
Array.Reverse(poignantArray);
```

```
Console.WriteLine("\nPoignant array reversed: ");
```

```
DisplayArray(poignantArray);
```

Write out the length and rank of poignantArray. Then clear out its contents and write the length and rank again.

```
Console.WriteLine("\nPoignant array length: {0}",  
poignantArray.Length);
```

```
Console.WriteLine("Poignant array rank: {0}",  
poignantArray.Rank);
```

```
Array.Clear(poignantArray, 0, poignantArray.Length);
```

```
Console.WriteLine("\nPoignant array cleared length: {0}",  
poignantArray.Length);
```

```
Console.WriteLine("Poignant array cleared rank: {0}",  
poignantArray.Rank);
```

Why are length and rank the same after clearing out the array?



Try it yourself: Use the System.Array API (cont.)

Write out the first index of Hope now.

```
Console.WriteLine("First occurrence of Hope: {0}",  
    Array.IndexOf(poignantArray, "Hope"));
```

What happens? Why?

Create another string array called nonsenseArray containing "Warehouse", "Flies", "Duck", "Under", "Baby"

```
string[] nonsenseArray = { "Warehouse", "Flies", "Duck",  
    "Under", "Baby" };
```

Write out the original contents, the reversed contents, and finally the sorted contents.

```
Console.WriteLine("\nNonsense array: ");  
  
DisplayArray(nonsenseArray);  
  
Array.Reverse(nonsenseArray);  
  
Console.WriteLine("\nNonsense array reversed: ");  
  
DisplayArray(nonsenseArray);  
  
Array.Sort(nonsenseArray);  
  
Console.WriteLine("\nNonsense array sorted: ");  
  
DisplayArray(nonsenseArray);  
  
Console.ReadKey();
```

Complex Arrays



Complex arrays are not commonly used at Epic, so these topics are beyond the basics. You can read about them if you are interested.

Using Multidimensional Arrays

So far, the arrays we have seen only have a single dimension. It is possible to declare an array with multiple dimensions, essentially creating a matrix. Dimensions are separated by a comma.

Syntax

```
type[,] arrayName = new type[size,size];
```

Example

```
int[,] intArray = new int[5,5];
```

```
const int Rows = 4;
const int Columns = 3;
int[,] rectangularArray =
{
    {0, 1, 2},
    {3, 4, 5},
    {6, 7, 8},
    {9, 10, 11}
};

for (int i = 0; i < Rows; i++)
{
    for (int j = 0; j < Columns; j++)
    {
        Console.WriteLine(
            "[{0},{1}] = {2}", i, j,
            rectangularArray[i, j]);
    }
}

Console.WriteLine("Rank: {0}", rectangularArray.Rank); // 2-dimensional
Console.WriteLine("Length: {0}", rectangularArray.Length); // 12 total elements

int[, ,] cubicArray = new int[2, 3, 4];

Console.WriteLine("Rank: {0}", cubicArray.Rank); // 3-dimensional
Console.WriteLine("Length: {0}", cubicArray.Length); // 24 total elements
```

Using Jagged Arrays

C# also supports jagged arrays, though they are rare in Epic code. Jagged arrays are multidimensional arrays where each row is a different size. Jagged arrays can be accessed using `arrayName[row][col]`.

	Col 0	Col 1	Col 2
Row 0	1	2	
Row 1	3	4	5
Row 2	6		

```
int[][] jaggedArray =  
{  
    new int[] { 1, 2 },  
    new int[] { 3, 4, 5 },  
    new int[] { 6 }  
};  
  
Console.WriteLine("Row 1, Col 2: {0}", jaggedArray[1][2]);  
Console.WriteLine("Rank: {0}", jaggedArray.Rank); // 1-dimensional  
Console.WriteLine("Length: {0}", jaggedArray.Length); // Think of each  
element as another 1-dimensional array
```

Activity: Using Arrays

- Expand the Developer class
- Maintain a private array containing the last 5 projects completed
- Provide a method to add a project. Always maintain the 5 most recent projects
- Provide a method to print out all values for projects
- Test the project features
- Modify the AddProject method to use the params keyword

Using Generic Collections

C# provides a variety of collection classes, each of which can be used to hold a dynamic number of elements. Each collection class comes with its own additional functionality, so you will want to find the right one for your needs. These classes are all found in the `System.Collections` namespace.

However, there is one major concern with using classes from `System.Collections`: type safety. These collections can only contain objects, nothing more specific.

This is where generic collections come in. Generic collections are found in the `System.Collections.Generic` namespace and have one key difference from classes in the `System.Collections` namespace: when a generic collection is instantiated, a data type must be specified for its elements.

Each generic collection has a `<T>` in its signature as a placeholder for the data type; the `<T>` is what "generic" refers to in the name "generic collection".

Specifying what type of data a generic collection will hold results in safer code. The compiler can call out if the wrong type of data is being added to a generic collection. Because the element data type is declared at instantiation (while remaining generic in the class declaration), generic collections can be reused with various data types while remaining type safe.



Try It Yourself: Compare ArrayList with List<t>

Examine the code below and answer the questions that follow.

```
ArrayList myArrayList = new ArrayList();  
  
List<string> myList = new List<string>();
```

Only strings are allowed in myList. What types are allowed in myArrayList?

Will the following code snippets compile?

```
myArrayList.Add ("Tahm");  
  
myArrayList.Add ("Ekko");  
  
myArrayList.Add ("Bard");
```

```
myList.Add ("Tahm");  
  
myList.Add ("Ekko");  
  
myList.Add ("Bard");
```

```
String.Compare ("Ekko", myArrayList[0]);
```

```
String.Compare ("Ekko", myList[0]);
```

How could you update the code so that it compiles?

In this case, is it better to use List<string> or ArrayList? Why?



Try It Yourself: Compare ArrayList with List<T> (cont.)

Now you try to add Cars to your collections. You've already defined the Car class earlier in your code and it has a constructor that takes no arguments.

Will the following code compile?

```
myArrayList.Add(new Car());
```

```
myList.Add(new Car());
```

```
String.Compare("Ekko", (string)myArrayList[3]);
```

You remove the code that doesn't compile and run the remaining code. What happens?

```
myArrayList.Add("Tahm");  
myArrayList.Add("Ekko");  
myArrayList.Add("Bard");  
myArrayList.Add(new Car());  
String.Compare("Ekko", (string)myArrayList[3]);
```

Why wasn't this error caught at compile time?

How does this compare with myList?



How do generic collections compare to arrays?

-

-

-

Example:

List is defined as...

```
namespace System.Collections.Generic  
{  
    public class List<T> : IList<T>, ICollection<T>, IList, ICollection,  
    ...  
    ...  
}
```

List can be instantiated with...

```
List<string> myList = new List<String>();
```



Generics can be used with

- Classes
- Structs
- Interfaces
- Delegates
- Methods
- Primitive Types

System.Collections.Generic

Contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections. These structures have a <T> in their signature, serving

as a placeholder for the type.

Lists

The List is one of the most common generic collections. Use it when you want to keep track of the order in which elements are added to a collection. Unlike arrays, Lists automatically resize as needed. Like all generic collections, a List contains elements of a particular type, specified at instantiation.

We will use the List class to demonstrate some common generic collection methods.

These common methods are typically easy to use and understand.

For example, try to figure out the output of this code example:



Try It Yourself: Generic Lists

What is the output of each call to Writeline?

```
Using System.Collections.Generic; //necessary to reference the  
List class
```

```
enum Fruit
```

```
{
```

```
    Apples,
```

```
    Bananas,
```

```
    Cranberries,
```

```
    Durian,
```

```
}
```

```
List<Fruit> basketOfFruit =  
    new List<Fruit>() { Fruit.Apples, Fruit.Durian };
```

```
Console.WriteLine(  
    string.Join(", ", basketOfFruit.ToArray()) );
```

•

```
basketOfFruit.Add(Fruit.Cranberries);
```

```
Console.WriteLine(  
    string.Join(", ", basketOfFruit.ToArray()));
```

•

```
basketOfFruit.Sort();
```

```
Console.WriteLine(  
    string.Join(", ", basketOfFruit.ToArray()));
```

•



Try It Yourself: Generic Lists (cont.)

```
Console.WriteLine(  
    "How many pieces of fruit? {0}",  
    basketOfFruit.Count);
```

•

```
Console.WriteLine(  
    "Are there apples? {0}",  
    basketOfFruit.Contains(Fruit.Apples));
```

•

```
basketOfFruit.Remove(Fruit.Apples);
```

```
Console.WriteLine(  
    "How many pieces of fruit? {0}",  
    basketOfFruit.Count);
```

•

```
Console.WriteLine(  
    "Are there apples? {0}",  
    basketOfFruit.Contains(Fruit.Apples));
```

•

```
basketOfFruit.Clear();
```

Queues and Stacks



Queues and Stacks are considered Beyond the Basics.

There are also generic collections for other common data structures, such as queues and stacks. Queues and stacks are used when you only need to access the first element or the most recently added element, respectively. Like Lists, the Queue and Stack generic collections will dynamically resize as needed.

Queue<T>	Represents a first-in, first-out collection of objects. Elements can be accessed in the order they're added.
Stack<T>	Represents a last-in, first-out collection of objects. Elements can be accessed in reverse order from how they're added.



Try it yourself: Work with a Queue

Create a Queue of ints called intQueue.

```
Queue<int> intQueue = new Queue<int>();
```

Use the Enqueue method to populate it with the first five multiples of 5.

```
for (int i = 1; i < 6; i++)  
{  
    intQueue.Enqueue(i*5);  
}
```

Overload the DisplayValues method from earlier to work with a Queue of ints. Keep in mind that Queues implement I Enumerable and that the size of a Queue is tracked in its Count property.

```
public static void DisplayValues(Queue<int> queueToDisplay)  
{  
    if (queueToDisplay != null && queueToDisplay.Count > 0)  
    {  
        foreach (int i in queueToDisplay)  
        {  
            Console.WriteLine("Queue contains {0}", i);  
        }  
    }  
}
```

Write out the contents of intQueue using DisplayValues.

```
DisplayValues(intQueue);
```

Access the first element in intQueue using the Dequeue method and write it out. Then write out the contents of intQueue.

```
Console.WriteLine("\nFirst element of intQueue: {0}",  
intQueue.Dequeue());  
  
DisplayValues(intQueue);
```



Try it yourself: Work with a Queue (cont.)

Write out the first element again, but this time using the Peek method. Then write out intQueue.

```
Console.WriteLine("\nFirst element of intQueue: {0}",  
intQueue.Peek());  
  
DisplayValues(intQueue);
```

How does Peek compare to Dequeue?



Try it yourself: Work with a Stack

The Stack API will look very similar to the Queue API from the last Try it yourself. Start by creating a Stack of ints called intStack.

```
Stack<int> intStack = new Stack<int>();
```

Use Push to populate it with the first five multiples of 5.

```
for (int i = 1; i < 6; i++)  
{  
    intStack.Push(i*5);  
}
```

Overload the DisplayValues method as before, but this time for a Stack of ints.

```
public static void DisplayValues(Stack<int> stackToDisplay)  
{  
    if (stackToDisplay != null && stackToDisplay.Count > 0)  
    {  
        foreach (int i in stackToDisplay)  
        {  
            Console.WriteLine("Stack contains {0}", i);  
        }  
    }  
}
```

Write out intStack using DisplayValues.

```
DisplayValues(intStack);
```

How is this different from the output you had for intQueue?



Try it yourself: Work with a Stack (cont.)

What method was used to view the first element of a Queue without removing it?

Write out the last element of intStack using the same method.

```
Console.WriteLine("\nLast element of intStack: {0}",  
intStack.Peek());
```

Write out the last element of intStack while also removing it from the stack using Pop. Then write out intStack.

```
Console.WriteLine("\nLast element of intStack: {0}",  
intStack.Pop());
```

```
DisplayValues(intStack);
```

What is the Pop analogue for Queues?

Dictionaries

The Dictionary< TKey, TValue > generic collection represents a keyed collection. Each element in the dictionary is referred to as a value and is associated with a key. Values are added and accessed using their corresponding keys.

Like Lists, Dictionaries are useful in scenarios where you want to be able to access any element, but unlike Lists, you can access elements through some other means than the order they were added. They are also useful for tracking something unique, like an ID, because each key in the dictionary can only be used once.

Dictionary< TKey, TValue > Members

Count	Number of elements in the dictionary
Clear()	Removes all elements in the dictionary
ContainsKey(TKey)	Determines whether a key is in the dictionary
Remove(TKey)	Remove object with the given key from the dictionary

Add(TKey,TValue)	Add object to the dictionary at the given key
Values	Returns a collection (ValueCollection) of values in the dictionary.
Keys	Returns a collection (KeyCollection) of keys in the dictionary.
Indexer	dictRef[TKey] = TValue



Try it yourself: Work with a Dictionary

Using the Car class from earlier, create three cars named mrPlow, plowKing, and canyonero. Use any valid VIN, tank capacity, and MPG. Recall the Car constructor is:

```
public Car(string VIN, double TankCapacity, double  
MilesPerGallon)  
  
Car mrPlow = new Car("MP200", 24.5, 23.4);  
  
Car plowKing = new Car("PK600", 40.6, 11.9);  
  
Car canyonero = new Car("C450", 35.7, 8.7);
```

Create a dictionary called myCars with string keys and Car values.

```
Dictionary<string, Car> myCars = new Dictionary<string, Car>();
```

Add the three Cars above using their VIN as their key.

```
myCars.Add(mrPlow.VIN, mrPlow);  
  
myCars.Add(plowKing.VIN, plowKing);  
  
myCars.Add(canyonero.VIN, canyonero);
```

Access each car using a foreach loop over the keys and the dictionary's indexer. Due to advances in fuel technology, the cars can go much further than you initially thought. Double the MPG of each car. Then write out the updated MPGs.

```
foreach (string vin in myCars.Keys)  
{  
  
    Car c = myCars[vin];  
  
    double mpg = 2 * c.MilesPerGallon;  
  
    c.MilesPerGallon = mpg;  
  
    Console.WriteLine("Car with VIN {0} has MPG {1}", vin, mpg);  
}
```



Try it yourself: Work with a Dictionary (cont.)

Remove the canyonero and then loop over the remaining cars. This time, directly access the cars in the foreach loop. Write out their VINs and tank capacities.

```
myCars.Remove(canyonero.VIN);  
  
foreach (Car c in myCars.Values)  
  
{  
  
    Console.WriteLine("Car with VIN {0} has tank capacity {1}",  
c.VIN, c.TankCapacity);  
  
}
```

Now you want to add plowQueen to the dictionary, but you want to read the VIN from the console. Add the code to do this. After getting the VIN, create plowQueen using the tank capacity and MPG of plowKing.

```
String input = "";  
  
Console.WriteLine("Enter the VIN for Plow Queen: ");  
  
input = Console.ReadLine();  
  
Car plowQueen = new Car(input, plowKing.TankCapacity,  
plowKing.MilesPerGallon);
```

Add plowQueen to myCars, then write out each car's VIN and tank capacity.

```
myCars.Add(plowQueen.VIN, plowQueen);  
  
foreach (Car c in myCars.Values)  
  
{  
  
    Console.WriteLine("Car with VIN {0} has tank capacity {1}",  
c.VIN, c.TankCapacity);  
  
}
```

Run the code. When prompted for the VIN, enter the VIN of plowKINGing. What happens?



Try it yourself: Work with a Dictionary (cont.)

While plowKing and plowQueen might have a lot of in common, they must have different VINs if the VIN is used as the dictionary's key. To prevent the exception, wrap the code for collecting plowQueen's VIN in a while loop. Exit the loop with a unique VIN or on null or whitespace.

```
String input = "";

while (true)
{
    Console.WriteLine("Enter the VIN for Plow Queen: ");
    input = Console.ReadLine();
    if (string.IsNullOrWhiteSpace(input))
    {
        break;
    }
    if (myCars.ContainsKey(input))
    {
        Console.WriteLine("VIN {0} is already in use!", input);
    }
    else
    {
        Car plowQueen = new Car(input, plowKing.TankCapacity,
plowKing.MilesPerGallon);

        myCars.Add(plowQueen.VIN, plowQueen);
        break;
    }
}

foreach (Car c in myCars.Values)
{
    Console.WriteLine("Car with VIN {0} has tank capacity {1}",
c.VIN, c.TankCapacity);
}
```



Before adding a new key to a dictionary you must first make sure that the key does not exist in the dictionary already. Use the ContainsKey method to do this.

Converting to an Array

Sometimes you may want to translate a generic collection to an array. The methods `ToArrayList` and `CopyTo` can help with this. `ToArrayList` will return a new array while `CopyTo` is used to populate an existing array.



Try it yourself: Using ToArray and CopyTo

Create a Stack of ints called primeStack and Push 2, 3, 5, and 7.

```
Stack<int> primeStack = new Stack<int>();  
  
primeStack.Push(2);  
  
primeStack.Push(3);  
  
primeStack.Push(5);  
  
primeStack.Push(7);
```

Use DisplayValues, which you wrote earlier, to show the contents of primeStack.

```
Console.WriteLine("Prime Stack: ");  
  
DisplayValues(primeStack);
```

Call ToArray on primeStack and put the result into an int array called primeArray.

```
int[] primeArray = primeStack.ToArray();
```

Use DisplayArray to write out the array.

```
Console.WriteLine("\nPrime Array: ");  
  
DisplayArray(primeArray);
```

Create an array of ten ints called theFours and populate it with the first ten multiples of 4. Write it out using DisplayArray.

```
int[] theFours = new int[10];  
  
for (int i = 0; i < 10; i++)  
  
{  
  
    theFours[i] = (i+1) * 4;  
  
}  
  
Console.WriteLine("\nThe Fours: ");  
  
DisplayArray(theFours);
```



Try it yourself: Using ToArray and CopyTo (cont.)

Use CopyTo to copy primeStack into theFours, starting at index 2 in theFours. Then write out theFours.

```
primeStack.CopyTo(theFours, 2);  
  
Console.WriteLine("\nThe Fours has changed: ");  
  
DisplayValues(theFours);
```

Use CopyTo with primeStack into theFours again, but this time start at index 23.

```
primeStack.CopyTo(theFours, 23);  
  
Console.WriteLine("\nThe Fours is broken: ");  
  
DisplayValues(theFours);
```

What happens?

Creating Custom Collections

Many of the characteristics of generic collections can be found in collection related interfaces. As a result, you can create your own custom generic collections by implementing the relevant interfaces.



Take 30 seconds to think about these questions.

Why create a custom collection?

- _____
- _____

Why make the collection generic?

- _____
- _____

Collection Interfaces

Here are some of the common interfaces you would use if you were to create a custom collection. The interfaces provide consistency for common collection functionality.

ICollection<T>	Implemented by all collections. Contains methods and properties to manipulate generic collections such as: <ul style="list-style-type: none">• Count• Add• Clear• Contains• Remove Also includes IEnumerable<T>
IComparer<T>	Defines a method to compare two objects, Compare.
IEnumerable<T>	Exposes an enumerator, which enables collections to be iterated over

	<p>using a foreach loop:</p> <ul style="list-style-type: none"> ● IEnumerator<T> GetEnumerator() <ul style="list-style-type: none"> ▪ Returns an enumerator to iterate over items in the collection of the type T. ● IEnumerator GetEnumerator() <ul style="list-style-type: none"> ▪ Returns an enumerator to iterate over items in the collection as objects (this is the older method that was used before generics were introduced).
IList<T>	Represents a collection of objects that can be individually accessed by index. Contains properties and methods such as: <ul style="list-style-type: none"> ● Item ● IndexOf ● RemoveAt
IDictionary<TKey, TValue>	Represents a generic collection of key/value pairs. Contains properties and methods such as: <ul style="list-style-type: none"> ● Item (as identified by a key) ● Add (add a key/value pair) ● ContainsKey ● TryGetValue

Indexers

An indexer is a C# construct that allows a class to be treated as a collection using the `[]` notation. It is implemented using a special kind of property. Often times you will want to provide an indexer for your custom collection as syntactic sugar for consuming code.

Syntax - declaration

```
access-modifier type this[type index]
{
    get { ... }
    set { ... }
}
```

Syntax - usage

```
type var = myClass[index]; // Access getter  
myClass[index] = var; // Access setter
```



- Indexers may be of any data type (common examples are int and string)
- Multiple parameters may be provided as an index
- `this` property may be overloaded to provide multiple indexers

Custom Generic Classes

Generic collections are preferred over nongeneric collections for the additional type safety. Use generics as placeholders for the elements when possible.

The syntax for using generics is not limited to custom generic collections.

Syntax

```
access-modifier class class-Name<T> { ... }
```

Notes

- Use T in place of the actual data type in the class definition
- T is defined when a variable of the class type is instantiated
- This is commonly used to create custom generic collections

Activity: Custom Generic Collection

You need to create a custom collection that remembers the order that items were added, but doesn't permit duplicate items. You could use `List<T>` for this, but a list allows duplicates. You could try `HashSet<T>` because it doesn't permit duplicates, but a HashSet makes no guarantees about the order of items.

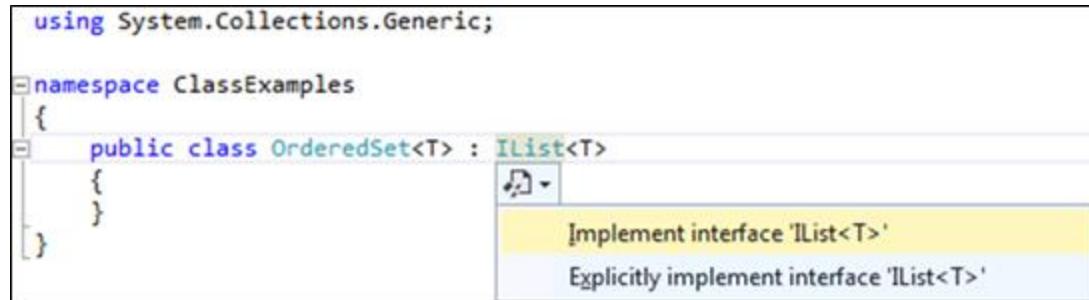
1. To get both features, you'll need a custom collection. Create a new class called `OrderedSet<T>`:

```
public class OrderedSet<T>  
{  
}
```

Suppose you want the `OrderedSet` to have all the features of a `List`. The easiest way to do this is to inherit from `List`, but this won't be ideal if you don't want all of the `List` features to be automatically visible to consumers of your class. Inheriting from `List` will make your collection too cluttered and more difficult to use than necessary.

2. To prevent clutter while still getting all of the features of a list, implement `IList<T>` rather than inheriting from `List<T>`.
3. Next, implement all parts of `IList<T>` explicitly:

Path: Place the cursor on `IList<T>` > CTRL + "." > Explicitly Implement 'IList<T>', or Right-Click `IList<T>` > Implement Interface > Implement Interface Explicitly



```
using System.Collections.Generic;

namespace ClassExamples
{
    public class OrderedSet<T> : IList<T>
    {
    }
```

Implementing `IList<T>` Explicitly

4. The code after explicitly implementing the interface will be similar to the following:

```
class OrderedSet<T>: IList<T>
{
    int IList<T>.IndexOf(T item)
    {
        throw new NotImplementedException();
    }

    void IList<T>.Insert(int index, T item)
    {
        throw new NotImplementedException();
    }

    void IList<T>.RemoveAt(int index)
    {
        throw new NotImplementedException();
    }

    T IList<T>.this[int index]
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }
}
```

```
void ICollection<T>.Add(T item)
{
    throw new NotImplementedException();
}

... More methods ...
```



Notice that the first several methods come from `IList<T>`, while the next come from `ICollection<T>`, then `IEnumerable<T>`, and finally `IEnumerable`. This is because `IList` actually extends the other interfaces. You can tell this by going to the definition of `IList<T>` using **F12**. `IList<T>` is defined as follows:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>,
    IEnumerable
```

5. You now have all of the methods of `IList<T>`, but calling any of them will throw an exception. To actually implement each method, create a private field of type `List<T>`, and use that list to implement each method:

```
class OrderedSet<T> : IList<T>
{
    private List<T> _list = new List<T>();

    int IList<T>.IndexOf(T item)
    {
        return _list.IndexOf(item);
    }

    void IList<T>.Insert(int index, T item)
    {
        _list.Insert(index, item);
    }

    void IList<T>.RemoveAt(int index)
    {
        _list.RemoveAt(index);
    }

    T IList<T>.this[int index]
    {
        get { return _list[index]; }
        set { _list[index] = value; }
    }
}
```

```
}

void ICollection<T>.Add(T item)
{
    _list.Add(item);
}

void ICollection<T>.Clear()
{
    _list.Clear();
}

bool ICollection<T>.Contains(T item)
{
    return _list.Contains(item);
}

void ICollection<T>.CopyTo(T[] array, int arrayIndex)
{
    _list.CopyTo(array, arrayIndex);
}

int ICollection<T>.Count
{
    get { return _list.Count; }
}

bool ICollection<T>.IsReadOnly
{
    get { return ((ICollection<T>)_list).IsReadOnly; }
}

bool ICollection<T>.Remove(T item)
{
    return _list.Remove(item);
}

IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
    return _list.GetEnumerator();
}

System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
```

```

    {
        return _list.GetEnumerator();
    }
}

```

6. To remove clutter from your source file, do the following:
 - EDIT > IntelliSense > Organize Usings > Remove and sort
 - Wrap all of the methods in OrderedSet in a region called "Hidden (Explicitly implemented) members from IList<T>"
7. The result appears similar to the following:

```

OrderedSet.cs  X
Epic.Training.Base
Epic.Training.Base.OrderedSet<T>
System.Collections.IEnumerable.GetEnumerator()

1  using System.Collections.Generic;
2
3  namespace Epic.Training.Base
4  {
5      public class OrderedSet<T> : IList<T>
6      {
7          private List<T> _list = new List<T>();
8
9          Hidden (Explicitly implemented) members from IList<T>
93     }
94 }
95

```

OrderedSet with hidden methods collapsed in a region

Currently, all of the methods and properties of OrderedSet are hidden because they are implemented explicitly. The only way you can see them is by casting to `IList<T>`.

8. In Program.cs, remove **using System.Linq** from the list of using statements



Be sure to remove **using System.Linq** from the top of **Program.cs**, otherwise there will be additional clutter that we are not interested in at this time.

9. Add some test code in the main function that creates an instance of `OrderedSet<int>`. Use IntelliSense to determine the methods that are available:

```

OrderedSet<int> mySet = new OrderedSet<int>();
mySet.|
```

?

Options available in IntelliSense

10. What options are available via IntelliSense?

a.

b.

c.

d.

11. Where are the listed options coming from?

a.

12. Next, suppose you want the Add method to be easy to access. Move it from the region of hidden options and make it public rather than explicitly implemented:

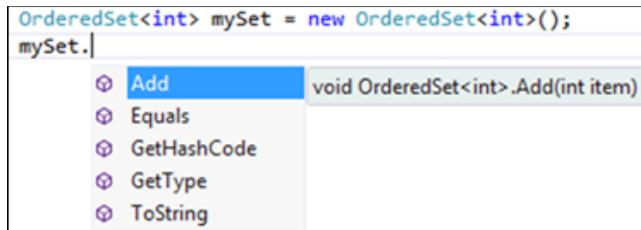
```
using System.Collections.Generic;

namespace ClassExamples
{
    public class OrderedSet<T> : IList<T>
    {
        private List<T> _list = new List<T>();

        public void Add(T item)
        {
            _list.Add(item);
        }

        #region Hidden (Explicitly implemented) members from IList<T>
        ... remaining methods (Add moved above) ...
        #endregion
    }
}
```

13. In Program.cs > Main, verify that the Add method is now available:



Add is now available

14. Now that the Add method is available, it's time to prevent duplicate items. For now, if the item is already in the list then just don't add it:

```
public void Add(T item)
{
    if (!_list.Contains(item))
    {
        _list.Add(item);
    }
}
```

15. Even though the "hidden" methods are not available by default, you can still access them if the `OrderedSet` is cast to an `IList`. This means we'll need to modify the following methods as well:
- The set accessor of the indexer
 - The Insert method

```
...
T IList<T>.this[int index]
{
    get { return _list[index]; }

    set
    {
        if(!_list.Contains(value))
        {
            _list[index] = value;
        }
    }
}
...
void IList<T>.Insert(int index, T item)
{
    if(!_list.Contains(item))
    {
```

```
    _list.Insert(index, item);  
}  
  
}  
  
...
```

16. Now the `OrderedSet` is exactly like a `List`, but doesn't allow duplicates. The final modification is to make any additional methods that you want to be easy to access public. Make both `Remove` and the indexer easy to access:

```
public class OrderedSet<T> : IList<T>  
{  
    private List<T> _list = new List<T>();  
  
    public void Add(T item)  
    {  
        if (!_list.Contains(item))  
        {  
            _list.Add(item);  
        }  
    }  
  
    public bool Remove(T item)  
    {  
        ...  
    }  
  
    public T this[int index]  
    {  
        ...  
    }  
  
    #region Hidden (Explicitly implemented) members from IList<T>  
    ... Moved Remove and Indexer above ..  
    #endregion  
}
```

17. Finally, use the following code to verify that the `OrderedSet` works as expected. Your output should match the output indicated in the comments:

```
OrderedSet<int> mySet = new OrderedSet<int>() { 3, 1, 2, 3 };
```

```
foreach (int num in mySet)
{
    Console.WriteLine(num + " "); //Output: 3 1 2
}

Console.WriteLine();

mySet.Add(4);
mySet.Remove(2);

foreach (int num in mySet)
{
    Console.WriteLine(num + " "); //Output: 3 1 4
}

Console.WriteLine();

Console.WriteLine("<Press any key to continue>");
Console.ReadKey();
```

If you have time

The Contains method of List is not the most efficient method to determine if an item is already in the collection, because it loops over all items to find a match. A more efficient method is to use an additional private collection such as a HashSet, which can find duplicates much faster.

- Modify the OrderedSet so that it has both a private List as well as a private HashSet. Whenever an item is added to the list, first check if it is in the hash set before adding it to both the list and the set. When an item is removed, be sure to remove it from both private collections.

Activity: Collections

- Continue with the CompanySim solution
- Create a Company class that has a Name and an internal list of employees. Use an OrderedSet as the private backing collection for the employees
- Create an indexer for the Company class
- Create a method, **CalculateAvgSalary**, that calculates the average salary for your employee list.
- Test your Company class

If you have time...

Suppose you want to create a single method that will return all employees of a specific type. For example, you could return a list of Developers, ProjectManagers or Executives. The most convenient way to do this is with a **generic method**.

1. The declaration of a generic method is given below. See if you can figure out how to fill its body:

```
public List<TSubEmp> GetEmployees<TSubEmp>()
    where TSubEmp : Employee
{  
}  
}
```

2. Test your solution by returning a list of employees of each sub class that you have defined.

Lesson 5: Working with Strings

Reading Exercise	5•3
Activity: Using Strings	5•3
Strings in C#	5•4
Using the String Class.....	5•4
String Interfaces	5•4
String Manipulation.....	5•4
String Comparison.....	5•5
String Concatenation	5•6
String Copying.....	5•6
Testing Strings for Equality	5•7
Finding Substrings.....	5•7
Splitting Strings.....	5•8
Using the StringBuilder Class	5•10
StringBuilder Members.....	5•10
Example: StringBuilder.....	5•10
Forming Regular Expressions.....	5•12
Regular Expression Syntax	5•12
Example: Regular Expressions	5•14
Example: Greedy vs. Lazy Quantifiers	5•15

Working with Strings

Reading Exercise

Activity: Using Strings

1. Read the following sections about strings, the `StringBuilder` class and using regular expressions.
2. Modify your `CompanySim` solution:
 - Create a method that verifies that a string follows the pattern "LAST, FIRST"
 - Test this method out on some employees to verify that their names fit the appropriate pattern.

Strings in C#

In C# strings are represented as an immutable series of characters. This means every time a string is modified, a new copy is created.

The string class is an alias to `System.String`, a built-in sealed class.

Using the String Class

String Interfaces

IComparable	<ul style="list-style-type: none">Implemented by types that need to sort.Implements the <code>CompareTo</code> method
ICloneable	<ul style="list-style-type: none">Creates a new instance of an object with the same valueImplements the <code>Clone</code> method
IConvertible	<ul style="list-style-type: none">Contains methods to convert to other typesImplement <code>ToInt32</code>, <code>ToDouble</code>, <code>ToDecimal</code>, etc.
IEnumerable	<ul style="list-style-type: none">Allows use of <code>foreach</code> to enumerate character by character

String Manipulation

String[#]	Strings are zero indexed and can be accessed in the same manner as an array.
Compare()	Compares two strings
Copy()	Creates a new string by copying another
EndsWith()	Determines whether or not a string ends with a series of characters
Equals()	Determines whether or not two strings have the same value
Format()	Formats a string using a format specification
Insert()	Inserts a given string at a given location
Length	Returns the number of characters in a string

PadLeft()	Aligns the characters in the string
PadRight()	
Remove()	Deletes a number of characters from the string
Split()	Divides a string based on a delimiter
StartsWith()	Determines whether or not a string starts with a set of characters
Substring()	Returns a substring
ToCharArray()	Copies the characters of a string to a character array
ToLower()	Returns a copy of the string with the case changed
ToUpper()	
Trim()	Removes leading and trailing whitespace from the string
TrimEnd()	Like Trim() but from the beginning or end of the string
TrimStart()	



Often you will want to check to see if a string is null (meaning it does not point to any string), empty (your user did not enter anything), or both. The .NET Framework provides methods for accomplishing both checks in a single method call.

String.IsNullOrEmpty(str) Returns a true if a str is either null or empty.

String.IsNullOrWhiteSpace(str) Returns true if a string is either null, empty, or consists entirely of whitespace characters.

These methods are static so they cannot be accessed through an instance of the String class.

If a string is null, and you try to manipulate that string, a NullReferenceException will occur. Use these methods to check for null, and then prevent the exception!

String Comparison

`String.Compare` is an overloaded method. It has two main forms, a case-sensitive and case-insensitive comparison. The return for both methods is the same.

- Negative value if `string1 < string2`
- Positive value if `string1 > string2`
- Zero if equal

```
string s1 = "abcd";
string s2 = "ABCD";
int result;

result = string.Compare(s1, s2);
string prompt = "{0} compared to {1}: {2}";
Console.WriteLine(prompt, s1, s2, result);

result = string.Compare(s2, s1);
prompt = "{0} compared to {1}: {2}";
Console.WriteLine(prompt, s2, s1, result);

result = string.Compare(s1, s2, true);
prompt = "{0} compared to {1} ignoring case: {2}";
Console.WriteLine(prompt, s1, s2, result);
Console.ReadKey();
```

String Concatenation

Concatenation of strings can occur in two ways:

1. Use the `Concat` method of the `String` class
2. Use the overloaded `+` operator

```
string s1 = "abcd";
string s2 = "ABCD";

string s3 = string.Concat(s1, s2);
Console.WriteLine("string.Concat(s1, s2) = {0}", s3);

string s4 = s1 + s2;
Console.WriteLine("s1 + s2 = {0}", s4);
Console.ReadLine();
```

String Copying

There are two methods available to copy strings.

1. Use the Copy method of the string class. Pass a string and it will return a copy of that string.
2. Use the overloaded = operator

```
string s1 = "abcd";
string s2 = "ABCD";

string s5 = string.Copy(s2);
Console.WriteLine("s5 copied from s2: {0}", s5);

string s6 = s5;
Console.WriteLine("s6 = s5: {0}", s6);
Console.ReadLine();
```

Testing Strings for Equality

There are three ways to test for string equality in C#

1. Use the overridden Equals method of the Object class
2. Use the Equals method of the String class
3. Use the overloaded == operator

```
string s1 = "abcd";
string s2 = "ABCD";

string s5 = string.Copy(s2);
Console.WriteLine("s5 copied from s2: {0}", s5);

string s6 = s5;
Console.WriteLine("s6 = s5: {0}", s6);

Console.WriteLine("\nDoes s6.Equals(s5)?: {0}", s6.Equals(s5));

Console.WriteLine("Does Equals(s6,s5)?: {0}", string.Equals(s6,s5));

Console.WriteLine("Does s6==s5?: {0}", s6==s5);
```

Finding Substrings

IndexOf()

Returns the index of the first occurrence of a substring within a string

LastIndexOf()	Returns the index of the last occurrence of a substring within a string
SubString()	Extracts a series of characters starting at a position within the string and continuing for a given number of characters.

```
string s1 = "One Two Three Four";
int index;

index=s1.LastIndexOf(" ");
string s2 = s1.Substring(index+1);

s1 = s1.Substring(0,index);

index = s1.LastIndexOf(" ");
string s3 = s1.Substring(index+1);

s1 = s1.Substring(0,index);

index = s1.LastIndexOf(" ");
string s4 = s1.Substring(index+1);

s1 = s1.Substring(0,index);

index = s1.LastIndexOf(" ");
string s5 = s1.Substring(index+1);

Console.WriteLine ("s2: {0}\ns3: {1}",s2,s3);
Console.WriteLine ("s4: {0}\ns5: {1}\n",s4,s5);
Console.WriteLine ("s1: {0}\n", s1);
Console.ReadLine();
```

Splitting Strings

`String.Split` can be used to parse a string into an array of substrings. This method returns an array of substrings.



Due to the immutable nature of strings in C# splitting strings can be an incredibly expensive operation. It should be used only when necessary.

```
string s1 = "One,Two,Three Liberty Associates, Inc.";  
  
const char Space = ' ';  
const char Comma = ',';  
  
char[] delimiters = new char[]  
{  
    Space,  
    Comma  
};  
  
string output = "";  
int ctr = 1;  
  
String[] resultArray = s1.Split(delimiters);  
  
foreach (String substring in resultArray)  
{  
    output += ctr++;  
    output += ": ";  
    output += substring;  
    output += "\n";  
}  
Console.WriteLine(output);
```

Using the StringBuilder Class

Because of the immutable property of the String class, string operations can potentially be massive resource sinks. To mitigate this we can utilize a class that handles strings in a mutable manner.

That class is the `StringBuilder`. It resides in the `System.Text` namespace.

StringBuilder Members

Append()	Appends to the end of the current string
AppendFormat()	Appends a formatted string
EnsureCapacity()	Ensures at least as big as the value given is allocated
Capacity	Retrieves or assigns the number of characters
Insert()	Inserts a character at a specified position
Length	Retrieves or assigns the length of the string
MaxCapacity	Retrieves the maximum string capacity
Remove()	Removes the specified characters
Replace()	Replaces all instances of specified characters with other characters
StringBuilder[#]	Implements an indexer so characters can be accessed via array notation

Example: StringBuilder

```
string s1 = "One,Two,Three Liberty Associates, Inc.";  
  
const char Space = ' ';  
const char Comma = ',', ' ';  
  
char[] delimiters = new char[]  
{  
    Space,  
    Comma,
```

```
Comma
};

StringBuilder output = new StringBuilder();
int ctr = 1;

foreach (string substring in s1.Split(delimiters))
{
    output.AppendFormat("{0}: {1}\n", ctr++, substring);
}
Console.WriteLine(output);
```

Forming Regular Expressions

Regular Expressions are incredibly powerful tools in a programmer's skill set. A regular expression is a pattern matching expression that allows for the comparison of a string and/or a series of wildcards. A regular expression can be applied to a string and returns a substring.

C# provides extensive regular expression classes in the `System.Text.RegularExpressions` namespace.

Regular Expression Syntax

Character codes

These codes indicate single characters within a regular expression.

Character	Function
Any character except [\^\$. ?*+(){}	The literal value of that character. To use [\^\$. ?*+(){ } precede any of them with a \ i.e. \+
\Fxx where xx is a hexadecimal number	The hex number corresponds to an ASCII character code
\n, \t, \r	Denotes a line feed, tab, and carriage return character

Character classes

It is possible to create classes or sets of characters that your regular expression can match against.

Syntax	Function
[...]	Brackets denote the beginning and end of a character class. For example [A-Z] would match any uppercase letter.
Any character except ^] or -	The literal value of that character will be included that character in the character class. To use ^] or - precede them with a \. For example

Syntax	Function
	[^\^]] would match a ^ or a] character.
^ followed by any character except ^ or -	The literal value of that character will be excluded from that character class. For example, [^A-Z] would match anything except an uppercase letter.
Any character followed by - followed by any other character	Creates a range of characters. Ex. [A-Za-z] would indicate all alphabetic characters
\d, \w, \s	Matches digits, word characters (alphabetic characters, numbers, and underscores), and white space characters. For example [\w\s] would match any word character or any whitespace character.
\D, \W, \S	Excludes the above pre defined character classes. For example, [\D] would match anything except a numeric character.

The Dot

. will match any character except \r and \n.

Anchoring

It is possible to anchor a pattern to either the beginning or end of a string. Preceding the expression with ^ or ending it with a \$ will anchor the pattern to the beginning or end of the string.

Quantifiers

Quantifiers determine how many times a match will occur. There are two types of quantifiers, lazy and greedy. Lazy quantifiers will stop after they successfully complete a match. Greedy quantifiers will try to continue to match across the entire string.

Syntax	Function
?	Makes the preceding item optional. Works in a greedy fashion meaning it will match as much

Syntax	Function
	as possible. The lazy version is ??
*	Matches zero or more of the previous item. Again this is the greedy form. The lazy form is *?
+	Matches one or more of the previous item. Like the above this is the greedy form. +? is the lazy form.
{n,m}	Matches a specific number of occurrences from n to m where m is optional. Both n and m must be numeric. The lazy form is {n,m}? Ex. {1,10} would match 1-10 occurrences of the previous item.

Alternation

Alternation represents an "or" condition within the regular expression. The pipe character (|) is used to indicate alternation. For example, "abc|def" would match either "abc" or "def".

The pipe has the lowest operator precedence so be sure to group using parentheses to ensure proper operation.

Example

```
^\d{1,2}/\d{1,2}/\d{4}
```

Would match a date in the form MM/DD/YYYY

Example: Regular Expressions

```
// Regex used for matching
string myBadSSN = "123-123-1234", myGoodSSN = "123-45-6789";
Regex ssnRegex = new Regex(@"^\d{3}-\d{2}-\d{4}$");
if (!ssnRegex.IsMatch(myBadSSN))
{
    Console.WriteLine("{0} is a bad SSN!", myBadSSN);
}
if (ssnRegex.IsMatch(myGoodSSN))
{
```

```
Console.WriteLine("{0} is a good SSN!", myGoodSSN);
}

Console.ReadLine();

// Regex used with Split()
string s1 = "One,Two,Three Liberty Associates, Inc.";
Regex splitRegex = new Regex(" |, |,");
StringBuilder sBuilder = new StringBuilder();
int id = 1;
foreach (string substring in splitRegex.Split(s1))
{
    sBuilder.AppendFormat("{0}: {1}\n", id++, substring);
}
Console.WriteLine("{0}", sBuilder);
```

Example: Greedy vs. Lazy Quantifiers

```
static void Main(string[] args)
{
    string input = "isisis";
    //create 2 regexes a greedy regex and a lazy regex
    Regex greedyRegex = new Regex(@"(is)+");
    Regex lazyRegex = new Regex(@"(is)+?");

    //regex can produce multiple matches
    MatchCollection greedy = greedyRegex.Matches(input),
    MatchCollection lazy = lazyRegex.Matches(input);

    Console.WriteLine("Greedy Matches");
    DisplayMatches(greedy);

    Console.WriteLine("\nLazy Matches");
    DisplayMatches(lazy);
    Console.ReadKey();
}

private static void DisplayMatches(MatchCollection matches)
{
    Console.WriteLine("Number of matches " + matches.Count);
    foreach (Match m in matches)
    {
        Console.WriteLine(m.Value);
    }
}
```

```
}
```

Lesson 6: Handling Errors

Introduction.....	6•3
By the end of this lesson you will be able to.....	6•3
Working with Exceptions	6•5
Catching Specific Exceptions.....	6•6
Always Executing Code.....	6•7
Getting Exception Details.....	6•9
In Visual Studio Express	6•10
Creating Custom Exceptions.....	6•12
Define Your Exception Class	6•12
Use Your Exception.....	6•12
Activity: Error Handling	6•12

Handling Errors

Introduction

- Why handle exceptions?
 - If you don't, your _____
 - Makes you (and Epic) _____
- When to use?
 - Deal with "possible" scenarios that are _____ or _____
- When not to use?
 - When _____ will work
 - _____ workflow scenarios

As with all forms of programming, errors should be anticipated and prevented where possible.

When it is not possible to prevent an error, exceptions are thrown and need to be handled. When an exception is raised it can bubble up the stack. If no handler is found the CLR will handle it and terminate the program.

By the end of this lesson you will be able to...

- Concepts
 - Describe the consequences of not using exception handling
 - Describe some scenarios when the use of exception handling would be appropriate
 - Describe some scenarios when the use of exception handling would be inappropriate
 - Explain why you might catch a specific class of exceptions
 - Describe how to use exceptions efficiently
 - Describe why you might throw an exception
 - Describe why you might rethrow a caught exception
 - Explain why some code needs to always execute
 - Describe the significance of checking for null before using objects inside a finally block
 - Explain why a programmer would create a custom exception
- Algorithms
 - Trace through the execution of a try-catch block in cases when exceptions are thrown/not thrown

- Trace through the execution of a try-finally block in cases when exceptions are thrown/not thrown
- Trace through the execution of a try-catch-finally block in cases when exceptions are thrown/not thrown
- Find the root error from an exception detail
- Find the stack trace from an exception detail
- Determine what exceptions might be thrown from a particular method/property

Working with Exceptions

The following code illustrates the basic way in which you should catch exceptions of a non-specific type.

```
try
{
    Console.WriteLine("Everything is fine");
    throw new Exception("Oh noes!");
    Console.WriteLine("All done!");
}

catch (Exception e)
{
    Console.WriteLine(e.Message);
    Console.WriteLine("Everything is !fine");
}
```

The following keywords are used to create and catch exceptions:

- try
 - Enclose code with _____ exceptions
- throw
 - Generates an exception
- catch
 - Traps exception from try block
 - Stack unwinds until catch is reached
 - No catch means _____
 - Catch as close to exception as possible
- Properties of System.Exception
 - Message
 - StackTrace
 - HelpLink

Note a catch _____ to lines following the exception within the try block.

Catching Specific Exceptions

Often a segment of code could result in more than one kind of exception and each one may require different code to deal with it appropriately. In this case you may require additional catch statements. For example, in the following code, a DivideByZeroException will be handled differently than exceptions of any other type:

```
try
{
    int quotient = 0, divisor = 0;
    quotient = 5 / divisor;
}
catch (DivideByZeroException e) {
    Console.WriteLine(e.Message);
}
catch (Exception e) {
    Console.WriteLine("not good");
    Console.WriteLine(e.Message);
}
```

- The first matching catch block will be used
- Place the catch-all section last

Always Executing Code

There are times when you want to ensure that a segment of code runs, even if an exception occurs. The keyword for marking such a code segment is **finally**.

One example of code that you would always want to run is a segment that frees unmanaged resources:

```
try
{
    // Obtain unmanaged resource
}
finally
{
    // Release unmanaged resource
}
```

You can also have a **finally** with a **try** and a **catch**. In this case, the **finally** will run even if an exception is thrown within the **catch**.

```
try
{
    throw new Exception("Oops!");
}
catch
{
    throw new Exception("Oops, I did it again!");
}
finally
{
    Console.WriteLine("I still execute!");
}
```



There is one scenario where `finally` could fail to execute

If the exception is never caught anywhere in the process and the program crashes, `finally` may not run (depends on certain CLR settings).

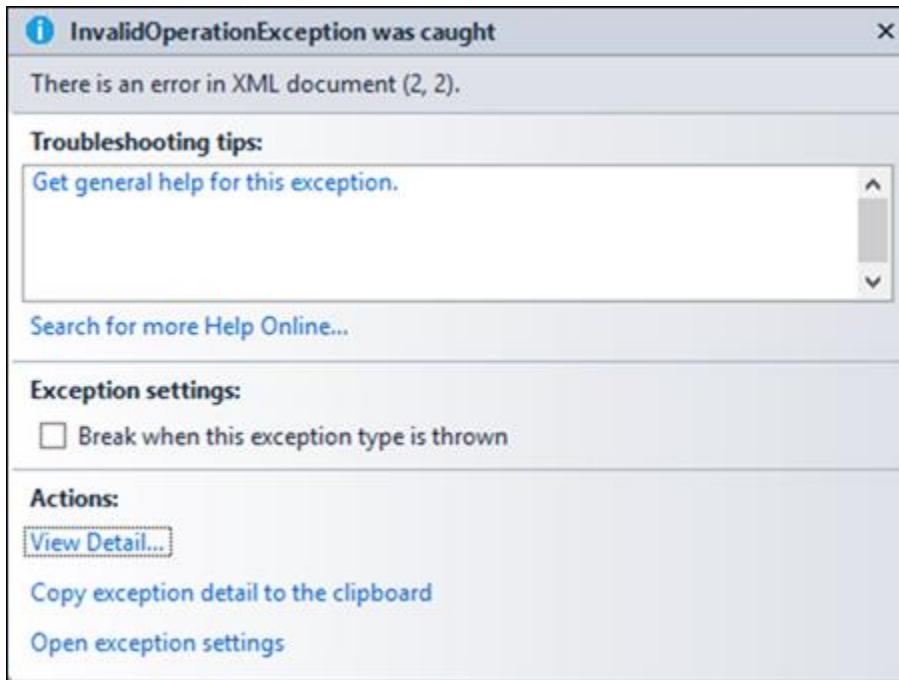
For details, see:

<http://msdn.microsoft.com/en-us/library/zwc8s4fz.aspx>

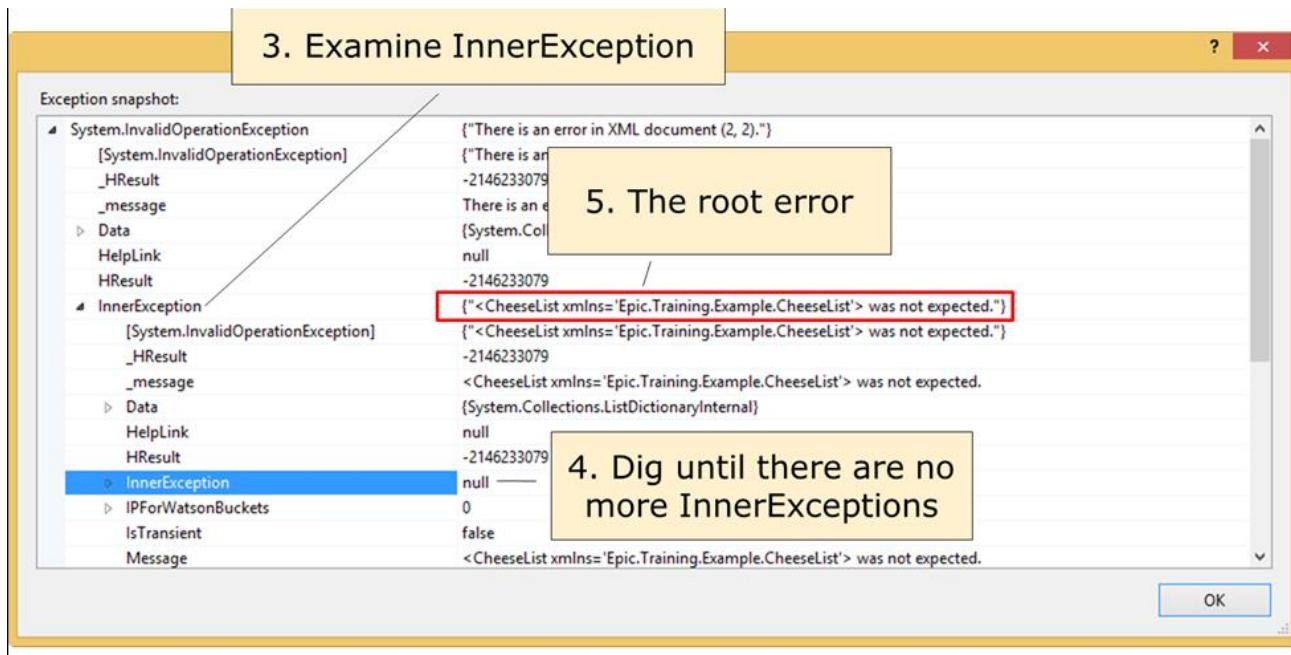
Getting Exception Details

Exceptions carry a lot of useful information about what was happening when the exception was thrown, but sometimes that information is buried inside layers of other exceptions.

1. From the exception assistant, click View Detail...

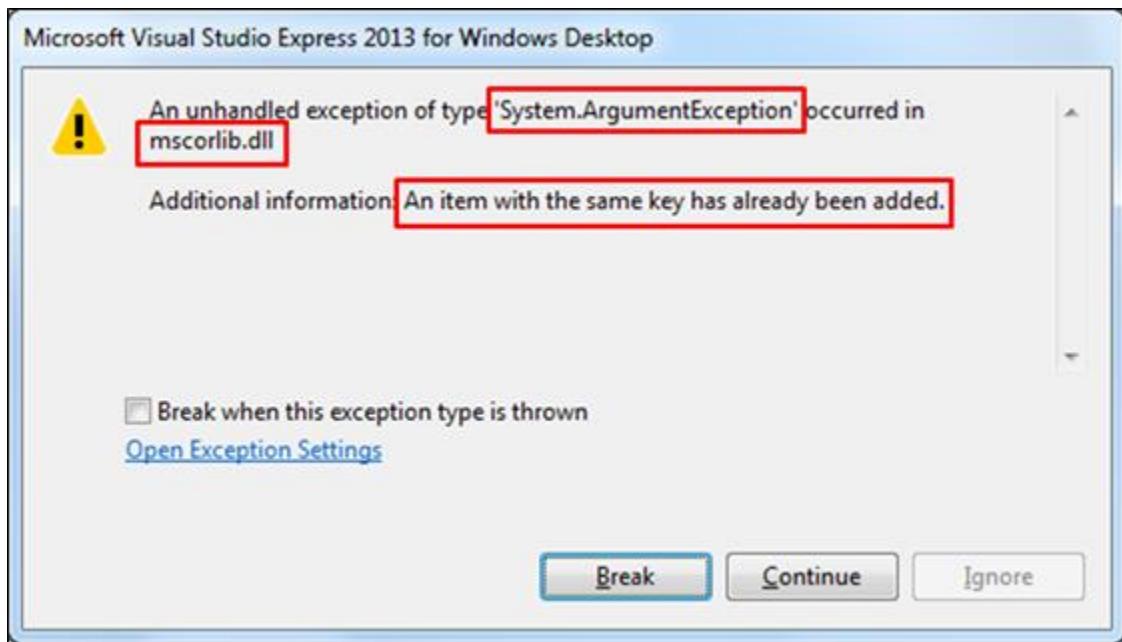


2. Find the Inner Exception
3. Dig until there are no more exceptions
 - The innermost exception is the root cause



In Visual Studio Express

- Basic info still listed in exception window



- Click "Break".
- From the menu, choose Debug > Windows > Watch > Watch 1
- Enter the variable \$exception
- Expand \$exception for details

Name	Value	Type
\$exception	{"An item with the same key has already been added."}	System.E
↳ [System.ArgumentException	{"An item with the same key has already been added."}	System.A
↳ Data	{System.Collections.ListDictionaryInternal}	System.C
↳ HelpLink	null	string
↳ HResult	-2147024809	int
↳ InnerException	null	System.E
↳ Message	"An item with the same key has already been added."	string
↳ Source	"mscorlib"	string

Creating Custom Exceptions

A common scenario is one where you need to throw an exception that is specific to the code you are writing. If there isn't a pre-defined exception that matches, you may need to create your own. Fortunately, doing so is a straightforward process.

Define Your Exception Class

Rules for creating a custom exception class:

- The class name must end with _____
- The class must derive from _____
- Call the base constructor, passing a _____

```
public class RemarkableException
    : System.Exception
{
    public string Remarks { get; protected set; }

    public RemarkableException(string message, string remarks)
        : base(message)
    {
        Remarks = remarks;
    }
}
```

Use Your Exception

Since you went through the trouble of creating a custom exception class, you probably want to use it, and catch it specifically:

```
try
{
    throw new RemarkableException("Error message", "Remarks...");
}
catch (RemarkableException e)
{
    // Handle your exception
}
```

Activity: Error Handling

- Continue with the CompanySim solution
- Define a custom exception to report invalid Employee IDs.
- Create a method to test for invalid employee IDs. An employee ID always starts with 1 alphabetic character.
- Modify your Employee constructor so your custom exception is thrown if the ID is invalid.
- Add error handling in Main to catch any errors
- Test some invalid IDs

Lesson 7: Disposing of Structures

Introduction.....	7•3
By the end of this lesson you will learn.....	7•3
Allowing the System to Handle Managed Resources.....	7•4
Managed Resources.....	7•4
Garbage Collection.....	7•4
Handling Unmanaged Resources.....	7•8
Reading Exercise: Finalize vs. Dispose	7•8
Part 1: Implement IDisposable.....	7•8
Part 2: Free Large Managed Resources	7•9
Part 3: Create a Finalize Method	7•10
Part 4: Suppressing Finalization.....	7•12
Wrap Up	7•14
Implicitly Calling Dispose.....	7•15
Exercise: Garbage Collection.....	7•16

Disposing of Structures

Introduction

By the end of this lesson you will learn...

- Concepts
 - Explain the differences between managed and unmanaged resources
 - Explain the potential consequences of not disposing of structures
 - Describe the circumstances under which the .NET garbage collector will dispose of structures automatically
 - Explain why you might dispose of managed resources
 - Explain when a finalizer method is called
 - Describe why you might declare or suppress a finalizer method
 - Explain the purpose of IDisposable
 - Explain the function of each of the pieces of Epic's IDisposable design pattern
- Algorithms
 - Implement IDisposable according to Epic's design pattern



Why?

Why dispose of structures when you are done with them?

-
-
-

Allowing the System to Handle Managed Resources

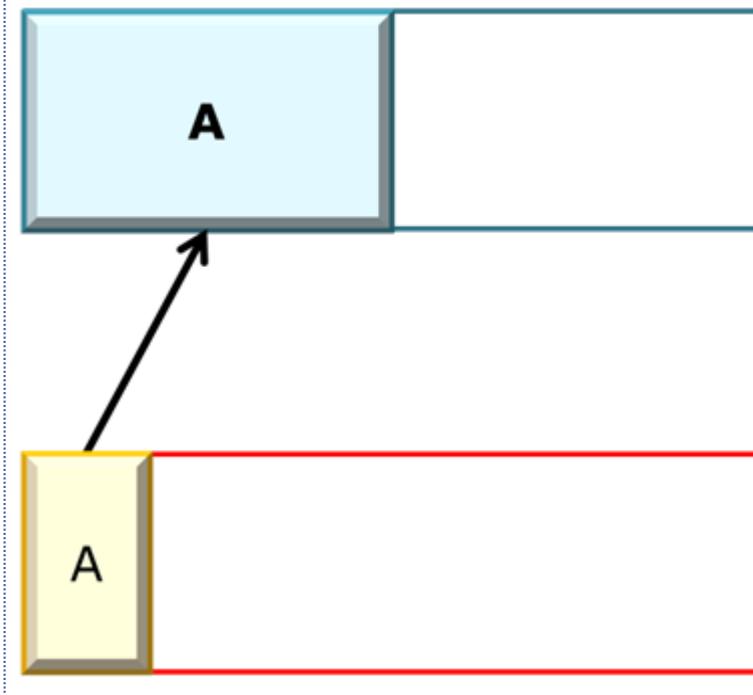
Managed Resources

- Resources cleaned up by the _____
- Application is more stable
- Less work for you

Garbage Collection

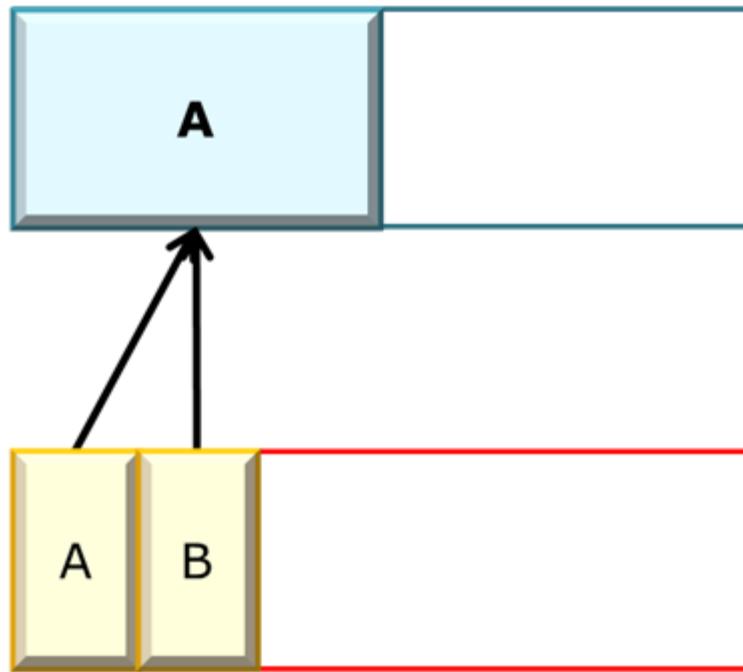
Garbage collection is the process which cleans up managed resources. The following example illustrates how this process works.

```
Type A = new Type();  
Type B = A;  
Type C = new Type();  
A.DoSomething();  
B.DoSomething();  
C = null;  
A = null;  
Type D = new Type();
```



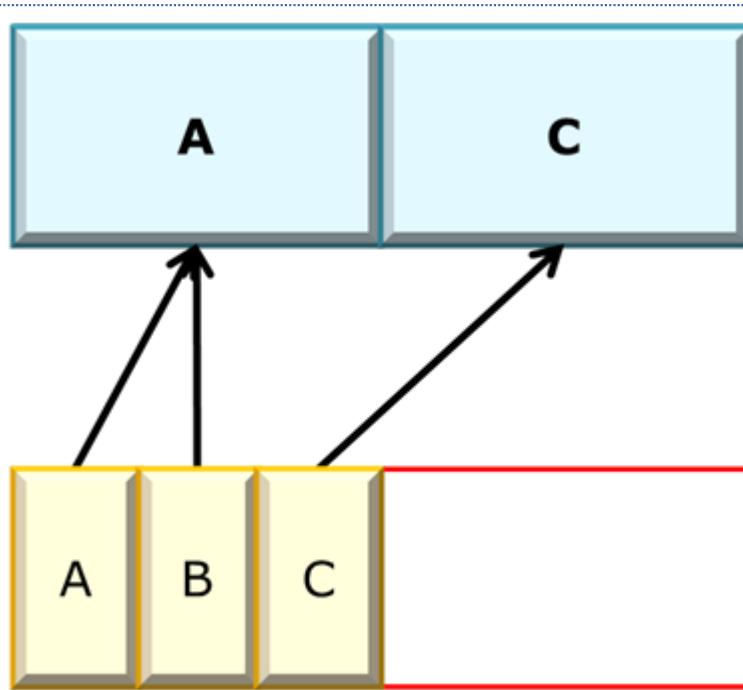
As you know, when the .NET framework creates a new object, it reserves space for the object on the heap and creates a reference to the object on the stack.

```
Type A = new Type();  
Type B = A;  
Type C = new Type();  
A.DoSomething();  
B.DoSomething();  
C = null;  
A = null;  
Type D = new Type();
```



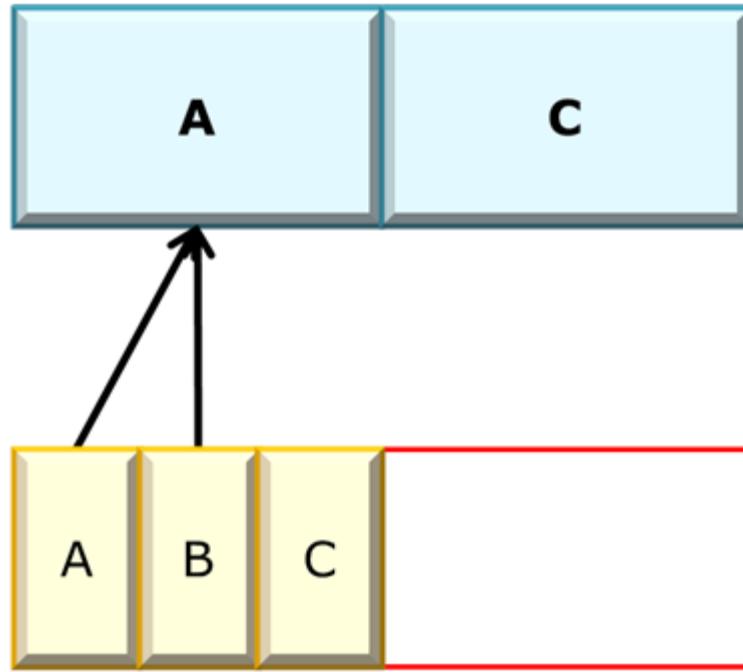
Next, the reference in **A** is assigned to **B**. Since **A** and **B** point to the same object, no additional space is required on the heap.

```
Type A = new Type();  
Type B = A;  
Type C = new Type();  
A.DoSomething();  
B.DoSomething();  
C = null;  
A = null;  
Type D = new Type();
```



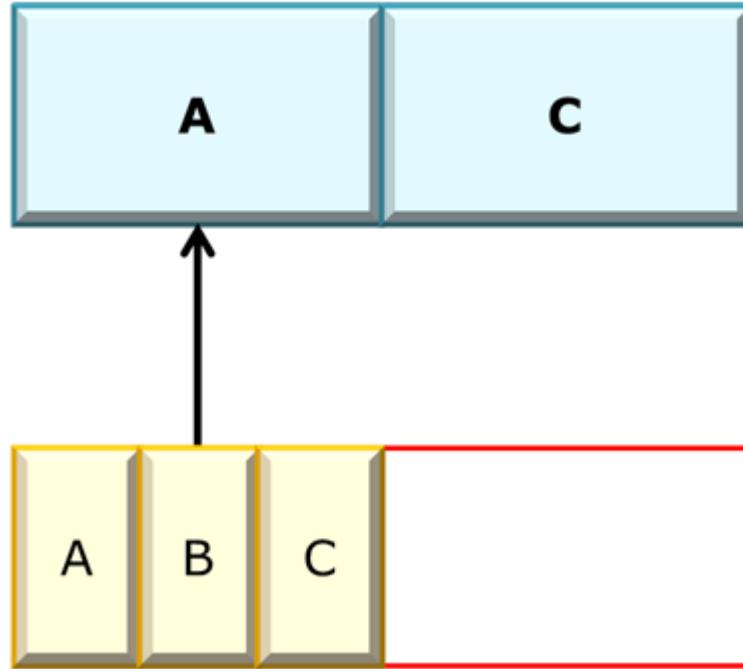
When the framework creates a new object, it requires additional heap space for the new object and also creates another reference on the stack.

```
Type A = new Type();  
Type B = A;  
Type C = new Type();  
A.DoSomething();  
B.DoSomething();  
C = null;  
A = null;  
Type D = new Type();
```



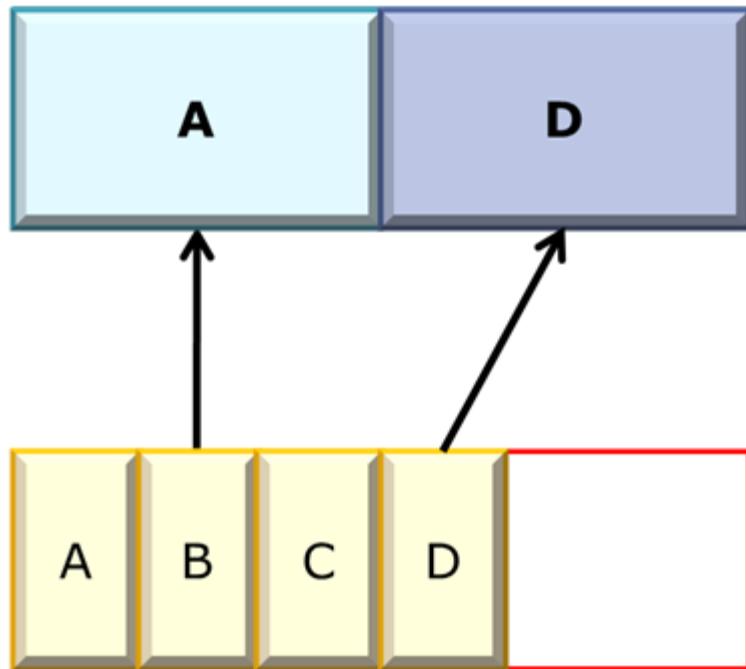
After calling a couple of methods, the reference to object C assigned to null. Notice that the object on the heap is not cleared right away, because the space is not required yet.

```
Type A = new Type();  
Type B = A;  
Type C = new Type();  
A.DoSomething();  
B.DoSomething();  
C = null;  
A = null;  
Type D = new Type();
```



When A is assigned null, the corresponding object also remains on the heap.

```
Type A = new Type();  
Type B = A;  
Type C = new Type();  
A.DoSomething();  
B.DoSomething();  
C = null;  
A = null;  
Type D = new Type();
```



When a new object is created, C is removed from the heap because there isn't enough space to fit object D. A is not removed, because there is still an active reference to it on the stack.

Handling Unmanaged Resources

- Not all resources are managed
- Unmanaged resources include:
 - File handles
 - Window handles
 - Database connections
- You must deal with these

Reading Exercise: Finalize v.s. Dispose

This reading exercise comes from a Stack-Overflow post. Take a moment to read the following text.

Part 1: Implement IDisposable

The point of Dispose is to free unmanaged resources. It needs to be done at some point, otherwise they will never be cleaned up. The garbage collector doesn't know how to call DeleteHandle() on a variable of type IntPtr, it doesn't know whether or not it needs to call DeleteHandle().



What is an unmanaged resource? If you found it in the Microsoft .NET Framework: it's managed. If you went poking around MSDN yourself, it's unmanaged. Anything you've used P/Invoke calls to get outside of the nice comfy world of everything available to you in the .NET Framework is unmanaged - and you're now responsible for cleaning it up.

The object that you've created needs to expose some method, that the outside world can call, in order to clean up unmanaged resources. There is even a standardized name for this method:

```
public void Dispose()
```

There was even an interface created, `IDisposable`, that has just that one method:

```
public interface IDisposable
{
    void Dispose();
}
```

So you make your object expose the `IDisposable` interface, and that way you promise that you've written that single method to clean up your unmanaged resources:

```
public void Dispose()
{
    Win32.DestroyHandle(this.gdiCursorBitmapStreamHandle);
}
```

And you're done. **Except you can do better.**

Part 2: Free Large Managed Resources

What if your object has allocated a 250MB System.Drawing.Bitmap (i.e. the .NET managed Bitmap class) as some sort of frame buffer? Sure, this is a managed .NET object, and the garbage collector will free it. But do you really want 250MB of memory just sitting there - waiting for the garbage collector to eventually come along and free it? What if there's an open database connection? Surely we don't want that connection sitting open, waiting for the GC to finalize the object.

If the user has called Dispose (meaning they no longer plan to use the object) why not get rid of those wasteful bitmaps and database connections?

So now we will:

- Get rid of unmanaged resources (because we have to), and
- Get rid of managed resources (because we want to be helpful)

So lets update our Dispose method to get rid of those managed objects:

```
public void Dispose()
{
    //Free unmanaged resources
    Win32.DestroyHandle(this.gdiCursorBitmapStreamHandle);

    //Free managed resources too
    if (this.databaseConnection != null)
    {
        this.databaseConnection.Dispose();
        this.databaseConnection = null;
    }
    if (this.frameBufferImage != null)
    {
        this.frameBufferImage.Dispose();
        this.frameBufferImage = null;
    }
}
```

And all is good, **except you can do better!**

Part 3: Create a Finalize Method

What if the person forgot to call Dispose() on your object? Then they would leak some unmanaged resources!



They won't leak managed resources, because eventually the garbage collector is going to run, on a background thread, and free the memory associated with any unused objects. This will include your object, and any managed objects you use (e.g. the Bitmap and the DbConnection).

If the person forgot to call Dispose, we can still save them! We still have a way to call it for them: when the garbage collector finally gets around to freeing (i.e. finalizing) our object.



The garbage collector will eventually free all managed objects. When it does, it calls the Finalize method on the object. The GC doesn't know, or care, about your Dispose method. That was just a name we chose for a method we call when we want to get rid of unmanaged stuff.

The destruction of our object by the Garbage collector is the perfect time to free those pesky unmanaged resources. We do this by overriding the Finalize() method.



In C#, you don't explicitly override the Finalize method. You write a method that looks like a C++ destructor, and the compiler takes that to be your implementation of the Finalize method:

```
public ~MyObject()
{
    //we're being finalized (i.e. destroyed), call Dispose in case the user forgot to
    Dispose(); //---Warning: subtle bug! Keep reading!
}
```

But there's a bug in that code. You see, the garbage collector runs on a background thread; you don't know the order in which two objects are destroyed. It is entirely possible that in your Dispose() code, the managed object you're trying to get rid of (because you wanted to be helpful) is no longer there:

```
public void Dispose()
```

```
{  
    //Free unmanaged resources  
    Win32.DestroyHandle(this.gdiCursorBitmapStreamHandle);  
  
    //Free managed resources too  
    if (this.databaseConnection != null)  
    {  
        this.databaseConnection.Dispose(); <- crash, GC already destroyed it  
        this.databaseConnection = null;  
    }  
    if (this.frameBufferImage != null)  
    {  
        this.frameBufferImage.Dispose(); <- crash, GC already destroyed it  
        this.frameBufferImage = null;  
    }  
}
```

So what you need is way for Finalize to tell Dispose that it should not touch any managed resources (because they might not be there anymore), while still freeing unmanaged resources.

The standard pattern to do this is to have Finalize and Dispose both call a third(!) method; where you pass a Boolean saying if you're calling it from Dispose (as opposed to Finalize), meaning it's safe to free managed resources.

This internal method could be given some arbitrary name like "CoreDispose", or "MyInternalDispose", but it's tradition to call it Dispose(Boolean):

```
protected void Dispose(bool disposing)
```

But a more helpful parameter name might be:

```
protected void Dispose(bool freeManagedObjectsAlso)  
{  
    //Free unmanaged resources  
    Win32.DestroyHandle(this.gdiCursorBitmapStreamHandle);  
  
    //Free managed resources too, but only if i'm being called from  
    //Dispose (If i'm being called from Finalize then the objects  
    //might not exist anymore)  
    if (freeManagedObjectsAlso)  
    {  
        if (this.databaseConnection != null)  
        {  
            this.databaseConnection.Dispose();  
        }  
    }  
}
```

```
        this.databaseConnection = null;  
    }  
    if (this.frameBufferImage != null)  
    {  
        this.frameBufferImage.Dispose();  
        this.frameBufferImage = null;  
    }  
}
```

And you change your implementation of the `IDisposable.Dispose()` method to:

```
public void Dispose()
{
    Dispose(true); //I am calling you from Dispose, it's safe
}
```

and your finalizer to:

```
public ~MyObject()
{
    Dispose(false); //I am *not* calling you from Dispose, it's *not* safe
}
```



If your object descends from an object that implements Dispose, then don't forget to call their base Dispose method when you override Dispose:

```
public void Dispose()
{
    try
    {
        Dispose(true); //true: safe to free managed resources
    }
    finally
    {
        base.Dispose();
    }
}
```

And all is good, except you can do better!

Part 4: Suppressing Finalization

If the user calls Dispose on your object, then everything has been cleaned up. Later on, when the Garbage Collector comes along and calls Finalize, it will then call Dispose again.

Not only is this wasteful, but if your object has junk references to objects you already disposed of from the last call to dispose, you'll try to dispose them again!

You'll notice in my code I was careful to remove references to objects that I've disposed, so I don't try to call Dispose on a junk object reference. But that didn't stop a subtle bug from creeping in.

When the user calls Dispose: the handle gdiCursorBitmapStreamHandle is destroyed. Later when the garbage collector runs, it will try to destroy the same handle again.

```
protected void Dispose(Boolean iAmBeingCalledFromDisposeAndNotFinalize)
{
    //Free unmanaged resources
    Win32.DestroyHandle(this.gdiCursorBitmapStreamHandle); <--double destroy
    ...
}
```

The way you fix this is tell the garbage collector that it doesn't need to bother finalizing the object - its resources have already been cleaned up, and no more work is needed. You do this by calling GC.SuppressFinalize in the Dispose method:

```
public void Dispose()
{
    Dispose(true); // I am calling you from Dispose, it's safe
    GC.SuppressFinalize(this); //Hey, GC: don't bother calling finalize later
}
```

Now that the user has called Dispose, we have:

- freed unmanaged resources
- freed managed resources

There's no point in the GC running the finalizer - everything's taken care of.



For the original post, see:

<http://stackoverflow.com/questions/538060/proper-use-of-the-idisposable-interface/538238#538238>

Wrap Up

Examine the following code and then answer the corresponding questions.

```
public class Destroyer : IDisposable {  
  
    public void Dispose() { // (Question 1)  
        Dispose(true);  
        GC.SuppressFinalize(this); // (Question 2)  
    }  
  
    ~Destroyer() { // (Question 3)  
        Dispose(false);  
        Console.WriteLine("In destructor");  
    }  
  
    bool _isDisposed = false;  
  
    protected virtual void Dispose(bool disposing) {  
        if (!_isDisposed) // only dispose once  
        {  
            if (disposing) {  
                // Dispose managed resources (Question 4)  
            }  
            // Clean unmanaged resources (Question 5)  
        }  
        this._isDisposed = true; // (Question 6)  
    }  
}
```

1. Why would the programmer call this method?

■

2. Why is this line necessary?

■

3. When will this method be used?

■

4. Why bother disposing of managed resources?

5. What happens when this task is not done?



6. Why set this flag?



Implicitly Calling Dispose

Sometimes, you'll see a **using** statement:

- Automatically calls `Dispose()`
- Must implement `IDisposable`



Any object obtained with a **using** statement must implement the `IDisposable` interface.

Here is an example of calling `Dispose()` explicitly in a `try-finally` statement:

```
FileStream stream = null;
try
{
    stream = File.OpenRead("test.txt");
    // Use the stream
}
finally
{
    if(stream != null)
    {
        stream.Dispose();
    }
}
```

Here is an example of calling `Dispose()` implicitly with the `using` keyword.

```
using (FileStream stream = File.OpenRead("test.txt"))
```

```
{  
    // Use the stream  
}
```

Notice how concise the **using** example is when compared to the try-finally example.

Exercise: Garbage Collection

- Continue with the CompanySim solution
- Implement IDisposable for Employee
- Dispose of all Employee resources when finished

Lesson 8: Handling Events

Introduction.....	8•3
Event-driven programming is.....	8•3
Declaring Method Types	8•4
Method Types.....	8•4
Declaring Types of Methods	8•4
Combining Methods	8•5
Subscribing to Events.....	8•7
Activity: Using Events.....	8•7
Part 1 - Experimenting with Delegates and Events.....	8•7
Part 2 - Providing Information to Events	8•14
Wrap Up	8•22
If you have time	8•23

Handling Events

Introduction

Event-driven programming is

Event-driven programming is a paradigm where program flow is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.

In other words, you connect code to an event. If the event occurs, then the code runs.



Why use it?

- _____
- _____

Declaring Method Types

Method Types

Method types are as important as other data types. The goal is to assign methods of the correct type to appropriate tasks.

Type safety is key. This allows errors to be caught during compile time as opposed to runtime.

Declaring Types of Methods

Method declaration is done through a special type of member called a delegate. The delegate encapsulates the method.

To declare a delegate you must:

- Include **signature** and **return type**
- Use the keyword **delegate**

```
public delegate Competitor WinPicker(Competitor c1, Competitor c2);

public string PlayGame
(Competitor c1, Competitor c2, WinPicker pickWinner) {
    Competitor ftw = pickWinner(c1, c2);
    return ftw != null ? String.Format("{0} wins!", ftw.Name) :"It's a tie";
}

public Competitor PickGolfWinner(Competitor c1, Competitor c2) {
    if (c1.Score == c2.Score) { return null; }
    return c1.Score < c2.Score ? c1 : c2;
}

public Competitor PickDartsWinner(Competitor c1, Competitor c2) {
    if (c1.Score == c2.Score) { return null; }
    return c1.Score > c2.Score ? c1 : c2;
}

Competitor c1 = new Competitor("Tiger", 25);
Competitor c2 = new Competitor("Obama", 50);
PlayGame(c1, c2, PickGolfWinner);
PlayGame(c1, c2, PickDartsWinner);
```

Combining Methods

It may be useful or even required to have multiple functions associated with a single member.

- A **delegate field** can point to multiple methods
- This is referred to as _____
- The return type should be _____



Method invocation order in the .NET Framework is generally in the order in which they are added to the delegate. This cannot be assumed because creating a dependency between methods is bad style. In that situation, consider wrapping all dependent method calls in a larger method so that the order of invocation is closely controlled.

Model Class:

```
public class PartyPlanner
{
    delegate void PartyPrep(Party p, int numGuests);

    //Delegate field
    public PartyPrep prepParty;
}
```

Model Class:

```
public class Helper
{
    //Subscriber methods must match delegate
    public void CleanHouse(Party p, int numGuests) { }
    public void CookFood(Party p, int numGuests) { }
    public void GetDrinks(Party p, int numGuests) { }
}
```

Controller Class:

```
class PartyBehavior
{
    Party _party { get; set; }
    PartyPlanner _pPlanner { get; set; }
    Helper _h1 { get; set; }
    Helper _h2 { get; set; }
```

```
Helper _h3 { get; set; }

public void PrepTheParty(int numGuests)
{
    //Controller subscribes methods to delegate field
    if(_h1 != null) { _pPlanner.prepParty += _h1.CleanHouse; }
    if(_h2 != null) { _pPlanner.prepParty += _h2.CookFood; }
    if(_h3 != null) { _pPlanner.prepParty += _h3.GetDrinks; }

    //When delegate field is invoked, all subscribers run
    if(_pPlanner != null)
    {
        PartyPrep del = _pPlanner.prepParty;
        if(del != null) { del(_party, numGuests); }
    }
}
```

Subscribing to Events

Activity: Using Events

In this activity, you'll learn how to use events to establish indirect communication between objects in C#. This will enable you to quickly write code that is intuitive and extensible.

During this exercise, you'll learn to:

- Describe the difference between delegate fields and events
- Subscribe to events in existing classes
- Create custom events in your classes
- Reuse events defined by other classes in your class

Part 1 - Experimenting with Delegates and Events

Recall from the previous section that a delegate defines a method type. Before you can create or use an event, you need to investigate the type of methods that can subscribe to that event. In other words, you need to learn about the delegate that the method uses. Do you remember what a method type specifies?



What must all methods of the same type have in common?

1.

2.

Delegate variables support multicasting, which occurs when a delegate variable contains a list of methods to execute, instead of just one. When the delegate variable is invoked as a method (using parentheses), all methods contained in the delegate variable are invoked.



What must the return type be for delegates that support multicasting?

•

In the following steps, you'll experiment with delegates, delegate fields and events. By the end of this exercise you'll have a clear understanding of the differences between each of them.

Suppose programmers using your ordered set want to be notified when any code attempts to add a duplicate item into the set. Currently, when this happens the `OrderedSet` silently discards the duplicate. One option is to raise an exception, but perhaps you would rather handle this situation less rigidly. Events are a good way to do this because if anyone cares about duplicates being added, they can subscribe to the event.

1. Open your CompanySim solution.
2. Open `OrderedSet.cs`
3. In the namespace, but outside the `OrderedSet` class, create a method type for the event. The return type should be `void` so that there can be multiple subscribers (multicasting). For now, leave the signature empty:

```
public delegate void DuplicateAddAttemptedEventHandler();
```



Notice that the name of the method type ends in "EventHandler". This is Epic (and Microsoft) convention for delegates that are used specifically for events.



Because delegates are data types, they can be defined as a standalone type directly in a namespace, or nested inside a class. Where you place the delegate depends on if the delegate can be used independent of any particular class. In this case, multiple collections may want to raise events when a duplicate add attempt occurs, so we're declaring it in the namespace, not inside the `OrderedSet` class.

1. In `OrderedSet`, define a public delegate field named `DuplicateAddAttempted` of the type `DuplicateAddAttemptedEventHandler`:

```
public DuplicateAddAttemptedEventHandler DuplicateAddAttempted;
```

This is the delegate field we'll use to contain the list of subscribers that want to be informed when someone attempts to add a duplicate to the ordered set.

The next thing to do is locate the areas in `OrderedSet` where new values could be added. In those locations, determine if the value being added already exists in the set and invoke the delegate variable if that is the case. There are three methods to consider:

```
public void Add(T item)
{
    if (!list.Contains(item))
```

```
{  
    _list.Add(item);  
}  
else { DuplicateAddAttempted(); }  
}
```

```
public T this[int index]  
{  
    get  
    {  
        return _list[index];  
    }  
  
    set  
    {  
        if (!_list.Contains(value))  
        {  
            _list[index] = value;  
        }  
else { DuplicateAddAttempted(); }  
    }  
}
```

```
void IList<T>.Insert(int index, T item)  
{  
    if (_list.Contains(item))  
    {  
        _list.Insert(index, item);  
    }  
else { DuplicateAddAttempted(); }  
}
```

Next, let's write a short test to verify that everything is working as expected.

2. In Program.cs, add the following test code to verify the delegate variable works as expected:

```
OrderedSet<int> mySet = new OrderedSet<int>() { 3, 1, 2 };  
mySet.Add(3);
```

This code will create a new `OrderedSet<int>`, and initialize it by calling `add` once for each integer in the list. When the second 3 is added, this causes the delegate field to be invoked within the `Add` method.

3. Run your code.



What happens when the delegate field is invoked?

•



An exception is thrown because the delegate variable is empty when it is invoked. In other words, there isn't any object listening to the `DuplicateAddAttempted` event.

In order to prevent null-reference exceptions when invoking delegate fields, it is common to create a separate helper method that checks for subscribers before the invocation. The convention is to name the helper method `Raise<field/event name>`.

4. Add a helper method to your code that checks to see if there are methods in the delegate field (i.e., event subscribers). To do this, check the delegate variable to determine if it is null before invoking it:

```
private void RaiseDuplicateAddAttempted()
{
    DuplicateAddAttemptedEventHandler eventHandlers =
        DuplicateAddAttempted;
    if(eventHandlers != null)
    {
        eventHandlers();
    }
}
```



Setting the delegates to a separate variable prevents a race condition that can happen in multi-threaded applications. For details, see:

<http://blogs.msdn.com/b/ericlippert/archive/2009/04/29/events-and-races.aspx>

5. Next, modify `Add`, `Insert` and the indexer's set accessor to call `RaiseDuplicateAddAttempted` instead of invoking the delegate field directly.

```
public void Add(T item)
{
    if (!list.Contains(item))
    {
        list.Add(item);
    }
}
```

```
    else { RaiseDuplicateAddAttempted(); }
```

```
public T this[int index]
{
    get
    {
        return _list[index];
    }

    set
    {
        if (!_list.Contains(value))
        {
            _list[index] = value;
        }
        else { RaiseDuplicateAddAttempted(); }
    }
}
```

```
void IList<T>.Insert(int index, T item)
{
    if (!_list.Contains(item))
    {
        _list.Insert(index, item);
    }
    else { RaiseDuplicateAddAttempted(); }
}
```

6. Run your solution again. Verify that you don't get the NullReferenceException.

You've taken care of the case where there are no subscribers. Next, you'll add subscribers so they can react to the event being raised.

7. In Program.cs, after creating a new OrderedSet, subscribe to the delegate field using this code:

```
OrderedSet<int> mySet = new OrderedSet<int>() { 3, 1, 2 };

mySet.DuplicateAddAttempted += ReportDuplicateAttempt;

mySet.Add(3);
```

8. Of course, we haven't defined the method ReportDuplicateAttempt yet. Do so now using the auto-resolve feature of VisualStudio. Click the blue bar that appears below the missing method name and choose "Generate method stub for ReportDuplicateAttempt".



You can also create the method manually. Make sure it matches the DuplicateAddAttemptedEventHandler method type (i.e., it is void and has an empty parameter list).

9. In ReportDuplicateAttempt, write to the console:

```
private static void ReportDuplicateAttempt()  
{  
    Console.WriteLine("Someone tried to add a duplicate to mySet.");  
}
```

10. Run the test code and verify that you see the message.

Although everything appears to be working, there is actually a subtle encapsulation problem when using public delegate fields instead of events. The next several steps will illustrate this problem.

11. In program.cs, add this line of code just after subscribing to the event:

```
OrderedSet<int> mySet = new OrderedSet<int>() { 3, 1, 2 };  
mySet.DuplicateAddAttempted += ReportDuplicateAttempt;  
mySet.DuplicateAddAttempted();  
mySet.Add(3);
```

12. Run the project.



What happened?

•



When the first message was displayed, did a duplicate add attempt actually happen?

•



Which class should be responsible for raising the event, Program or OrderedSet?

- _____



This is the key problem with using public delegate fields: doing so exposes the ability to raise the event to code that should not be permitted to do so.

One possible solution to this encapsulation problem is to make the delegate field private and then publicly expose the ability to add and remove subscribers using public methods like this:

```
private DuplicateAddAttemptedEventHandler _duplicateAddAttempted;

public void AddDuplicateAddAttemptedSubscriber(
    DuplicateAddAttemptedEventHandler subscriber)
{
    _duplicateAddAttempted += subscriber;
}

public void RemoveDuplicateAddAttemptedSubscriber (
    DuplicateAddAttemptedEventHandler subscriber)
{
    _duplicateAddAttempted -= subscriber;
}
```

However, C# has a special syntax specifically for this situation that takes the place of the public methods:

```
private DuplicateAddAttemptedEventHandler _duplicateAddAttempted;

public event DuplicateAddAttemptedEventHandler DuplicateAddAttempted
{
    add
    {
        _duplicateAddAttempted += value;
    }

    remove
    {
        _duplicateAddAttempted -= value;
    }
}
```

```
    _duplicateAddAttempted -= value;  
}  
}
```

This syntax allows you to expose add/remove accessors of the private backing delegate to consuming code without exposing the ability to invoke the delegate.

Notice that events look very similar to properties with a private backing field. Like full properties, the full event syntax is somewhat verbose. Fortunately C# also provides a way to condense this syntax down to a single line of code:

```
public event DuplicateAddAttemptedEventHandler DuplicateAddAttempted;
```

This will automatically create an anonymous private backing delegate field with the same name as the event, so it can be invoked from within the class but not outside the class. The add/remove accessors are also added automatically and have the accessibility specified by the event's access modifier, which is public in this example.

1. Change the DuplicateAddAttempted public delegate field to include the **event** keyword:

```
public event DuplicateAddAttemptedEventHandler DuplicateAddAttempted;
```

2. Attempt to compile your solution.



What happens?

-

3. Remove the line from Program.cs that attempts to invoke the DuplicateAddAttempted event.
4. Verify that the solution compiles and runs.

Now your event is encapsulated appropriately.

Part 2 - Providing Information to Events

Recall that your Company class uses an ordered set to ensure that employees with duplicate IDs are not added to the company. Suppose that if a duplicate employee is added, we want to log that event so we can later determine why it happened. In other words, we want to keep an audit trail of all duplicate employee add attempts within Company. You can accomplish this by leveraging the DuplicateAddAttempted event that you just created in OrderedSet.

1. Comment out your OrderedSet event test code from the Program class.
2. In Company's constructor, subscribe the DuplicateAddAttempted event:

```
/// <summary>
/// List of employees .
/// </summary>
OrderedSet<Employee> _employees = new OrderedSet<Employee>();

/// <summary>
/// Constructor to set the name
/// </summary>
/// <param name="name">Name of the company</param>
public Company(string name)
{
    Name = name;
    _employees.DuplicateAddAttempted += DuplicateEmployeeAddAttempted;
}

/// <summary>
/// Called whenever an employee with matching ID is added to the company
/// </summary>
private void DuplicateEmployeeAddAttempted()
{
    throw new NotImplementedException();
}
```

Suppose we want to keep a list all duplicate add attempts along with the ID of the duplicate employee and when the attempt occurred. Notice that there is currently no way to get this information in our event handler.



If we want pass additional information to `DuplicateAddAttempted` event handlers, what do we need to change?

When creating event handler delegates, it is Epic (and Microsoft) convention to pass two parameters:

- A reference to the object raising the event
 - A reference to a class instance containing any additional information related to the event. This class should inherit from the .NET framework class `EventArgs`
3. Create the class that will contain the additional information for the event handlers. In this case, the extra information is a reference to the duplicate:

```
public class DuplicateAddAttemptedEventArgs : EventArgs
{
    public object Duplicate { get; private set; }

    public DuplicateAddAttemptedEventArgs(object duplicate)
    {
        Duplicate = duplicate;
    }
}
```



It is possible to make the `EventArgs` generic, so that the data provided matches the type contained in the ordered set. You will get a chance to try this out later.

4. Next, modify the delegate definition of `DuplicateAddedEventHandler` to include two additional parameters, the object raising the event and the event arguments:

```
public delegate void DuplicateAddAttemptedEventHandler(
    object sender, DuplicateAddAttemptedEventArgs args);
```

5. Modify `RaiseDuplicateAddAttempted` to accept the duplicate, and to pass the required information to the `DuplicateAddAttempted` event when it is raised:

```
private void RaiseDuplicateAddAttempted(T duplicate)
{
    DuplicateAddAttemptedEventHandler eventHandlers =
        DuplicateAddAttempted;
    if(eventHandlers != null)
    {
        eventHandlers(this, new DuplicateAddAttemptedEventArgs(duplicate));
    }
}
```

6. Everywhere RaiseDuplicateAddAttempted is called, pass the duplicate item:

```
public void Add(T item)
{
    if (!_list.Contains(item))
    {
        _list.Add(item);
    }
    else { RaiseDuplicateAddAttempted(item); }
}
```

```
public T this[int index]
{
    get
    {
        return _list[index];
    }
    set
    {
        if (!_list.Contains(value))
        {
            _list[index] = value;
        }
        else { RaiseDuplicateAddAttempted(value); }
    }
}
```

```
void IList<T>.Insert(int index, T item)
{
    if (!_list.Contains(item))
    {
        _list.Insert(index, item);
    }
}
```

```
    }
    else { RaiseDuplicateAddAttempted(item); }
}
```

Now the duplicate is available to subscribers of the event.

7. In Company, modify DuplicateEmployeeAddAttempted to keep a list of strings as an audit trail and to accept and use the new information provided by the event:

```
private List<string> _duplicateAddAttempts = new List<string>();

private void DuplicateEmployeeAddAttempted(
    object sender, DuplicateAddAttemptedEventArgs args)
{
    _duplicateAddAttempts.Add(
        string.Format("Duplicate add of {0} attempted on {1}", args.Duplicate, DateTime.Now));
}
```

8. Next, create a public property that allows you to iterate through each of the warning messages, while at the same time not allowing the list to change from outside the company class. One way to do this is to construct an IEnumerable on the fly using **yield return**:

```
public IEnumerable<string> DuplicateAttemptsTrace
{
    get
    {
        foreach (string duplicateMessage in _duplicateAddAttempts)
        {
            yield return duplicateMessage;
        }
    }
}
```



Returning each element of a collection, one at a time

Use a yield return statement to return each element one at a time. Using yield return causes the property or method to be an iterator method.

You consume an iterator method by using a foreach statement. Each iteration of the foreach loop calls the iterator method. When a yield return statement is reached in the iterator method, the expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator method is called.

This is also a good way to encapsulate contained collections. In the previous example, you could just return _duplicateAddAttempts, but consuming code could cast the object back to a List and then call the Clear method, which would wipe out your trace. Using yield return prevents this, because the private list is never available to consuming code.

9. In Program, add some test code that creates a new Company:

```
Company myCompany = new Company("<your name>'s Company");
```

10. Use foreach to iterate over DuplicateAttemptsTrace and display it to the console. For example, if the variable that references your company is named myCompany, the code would be:

```
Company myCompany = new Company("<your name>'s Company");

foreach (string traceMessage in myCompany.DuplicateAttemptsTrace)
{
    Console.WriteLine(traceMessage);
}
```

11. In program, write a subroutine that allows the user to create new developers and adds them to the Company. For example (your code may vary depending on your Developer constructor):

```
private static void GetNewDevelopers(Company company)
{
    while (true)
    {
        Console.WriteLine("Enter a developer ID (press enter to quit):");
        string id = Console.ReadLine();
        if (string.IsNullOrWhiteSpace(id))
        {
            return;
        }
        else if (!Employee.IsValidID(id))
        {
```

```
        Console.WriteLine("The ID must begin with an alphabetic
character.");
    }
    else
    {
        Developer newDeveloper =
            new Developer("New Developer", id, 0, default(Team));
        company.Add(newDeveloper);
    }
}
```



For the previous code segment to work, ensure that Employee.IsValidID is defined. One possible implementation is the following:

```
public static bool IsValidID(string id)
{
    return !string.IsNullOrWhiteSpace(id) && char.IsLetter(id[0]);
}
```

12. Call GetNewDevelopers before the trace is displayed.

```
Company myCompany = new Company("<your name>'s Company");

GetNewDevelopers(myCompany);

foreach (string traceMessage in myCompany.DuplicateAttemptsTrace)
{
    Console.WriteLine(traceMessage);
}
```

13. Ensure that the Add method of Company attempts to add duplicate employees to the contained OrderedSet:

The add method is either:

```
public void Add(Employee e)
{
    _employees.Add(e);
}
```

or:

```
public void Add(params Employee[] employees)
```

```
{
    foreach (Employee e in employees)
    {
        _employees.Add(e);
    }
}
```

14. Run the program and add several additional employees, some of which have duplicate names.
15. Verify that the trace includes the duplicates.

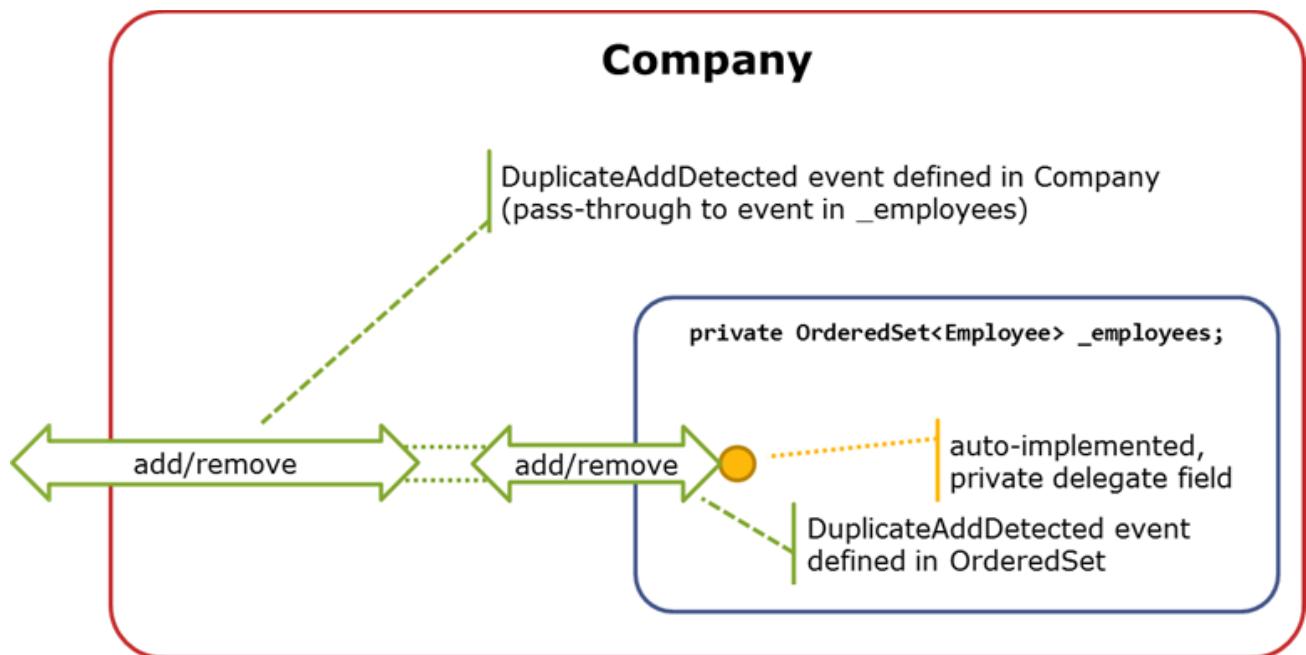


Currently, how many subscribers are there to the DuplicateAddAttempted event?

-

Suppose you want to allow consuming code of the company class to reuse the duplicate detection of the encapsulated ordered set. You could define a completely new event, but why spend the time doing that when OrderedSet already does everything that you need?

You can accomplish this by defining a new full event in Company that has add and remove accessors, but not its own delegate field. Instead, in add and remove it reuses the event and delegate field from the ordered set. The following diagram illustrates this setup:



Reusing events of encapsulated objects

16. Create a new full event in Company called DuplicateAddAttempted that ties into the event by the same name in the OrderedSet containing your employees:

```
public event DuplicateAddAttemptedEventHandler DuplicateAddAttempted
{
    add
    {
        _employees.DuplicateAddAttempted += value;
    }

    remove
    {
        _employees.DuplicateAddAttempted -= value;
    }
}
```

Now, any object that subscribes to Company's DuplicateAddDetected event will actually be subscribing to the DuplicateAddDetected event of _employees, which is the OrderedSet contained within Company.

1. Have program subscribe to the DuplicateAddAttempted event of Company before you call GetNewDevelopers. In the event handler, display a warning message:

```
private static void DisplayWarningMessage(
    object sender, DuplicateAddedEventArgs args)
{
    Console.WriteLine(
        "The employee with ID {0} is already in the company",
        ((Employee)args.Duplicate).ID);
}
```

2. Run your application again. This time, every time a duplicate is entered, you should get a warning message from program and company will log the add attempt. Now you have two listeners to the same event.

Wrap Up

1. In your own words, describe the difference between a delegate, delegate field and an event.

- Delegate:

- Delegate field:

- Event:

2. List the three things that are implicitly added to your class when using a shorthand event declaration.
- _____
 - _____
 - _____

If you have time

If you are up for a challenge, try the following:

1. Currently, when the DuplicateAddAttemptedEventHandler of Company is raised, a reference to _employees is passed as the sender. This could cause confusion to subscribers that expect a reference to Company.
 - Rather than having the DuplicateAddAttempted event of Company rely completely on the DuplicateAddAttempted event of _employees, change it so that Company has its own private backing delegate.
 - When the DuplicateAddAttempted event of _employees occurs, raise the DuplicateAddAttempted event of Company, passing Company as the sender.
 - Verify in Program that you can cast sender to a Company and include its name in the console message.
2. Modify the DuplicateAddAttemptedEventHandler delegate so that it is generic:

```
public delegate DuplicateAddAttemptedEventHandler<TCollection, TItem> ( TCollection sender, DuplicateAddAttemptedEventArgs<TItem> args );
```

 - Modify all code in DuplicateAddAttemptedEventArgs, OrderedSet, Company and program to work correctly without casting.
3. Modify Company to log an event whenever the collection is altered in any way, not just when a duplicate add attempt occurs
 - Have OrderedSet<T> implement the INotifyCollectionChanged interface.
 - Change the contained List<T> to an ObservableCollection<T> which already implements INotifyCollectionChanged

- Piggyback on the contained observable collection's CollectionChanged event in the OrderedSet.
- Once OrderedSet has an event CollectionChagned, have Company use that to log when new employees are added successfully.

Lesson 9: Storing and Retrieving Data

Why Store and Retrieve Data?	9•3
Accessing a Storage Location	9•4
The Big Picture	9•4
System.IO Namespace	9•4
Stream Classes.....	9•4
Using a Stream.....	9•5
Readers and Writers.....	9•5
Using a Reader/Writer.....	9•6
Example: Using a Reader/Writer.....	9•6
File I/O.....	9•6
FileStream Class	9•6
File Enumerations	9•7
Visualizers.....	9•7
Example: Visualizers	9•7
File and Directory Manipulation	9•8
Manipulating Files.....	9•8
Example: Reading Text from a File	9•8
Example: Writing Text to a File	9•9
Manipulating Directories	9•10
Example: Directory Lister.....	9•10
Example: File Watcher.....	9•10
Activity 17: Streams	9•12
Saving and Loading Objects	9•13
The Big Picture	9•13
The Process of Serialization.....	9•13
Object Graph	9•14
Attributes.....	9•14
Predefined and Custom Attributes	9•15
Querying Metadata Using Reflection	9•15
Serialization Attributes.....	9•15
Serialization Classes	9•17
Example: Serialization	9•17
Example: Deserialization.....	9•18

Custom Serialization.....	9•18
Example: Custom Serialization.....	9•19
Activity 18: Serialization.....	9•20

Storing and Retrieving Data

Why Store and Retrieve Data?

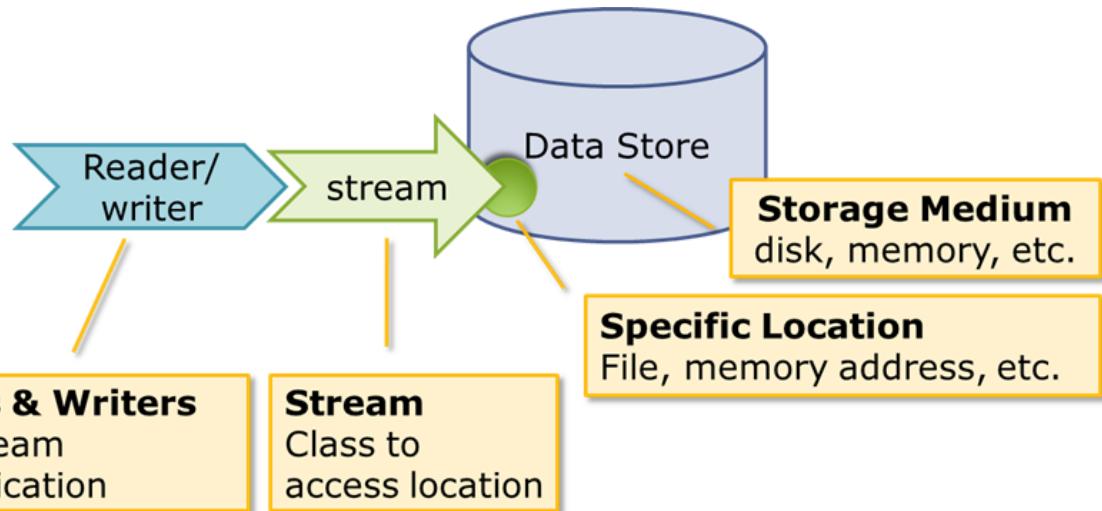
Moving data between storage mediums and memory is a critical programming task. This allows the application to:

- Communicate with the client, server and database
- Save information for later use

Accessing a Storage Location

A stream provides read/write services from/to storage mediums. Providing input/output through a class is more flexible than doing so through the language itself. There are streams that provide access to files or local memory, and other streams that enable buffering and cryptography.

The Big Picture



System.IO Namespace

- Contains all basic IO classes
- Some commonly used classes:

<code>___.Stream</code>	Buffer reads and writes to another stream
<code>___.MemoryStream</code>	Creates streams with memory as the storage medium
<code>___.FileStream</code>	Reading and writing to files
<code>___.CryptoStream</code>	Links data streams to cryptographic transformations

Stream Classes

Stream classes provide read/write services from/to storage mediums

- Standard stream methods include:
 - Read
 - Write
 - Close
 - Seek
 - SetLength
 - Flush
- Standard stream properties include:
 - CanRead
 - CanWrite
 - CanSeek
 - Position
 - Length

Using a Stream

- Creating an Output Stream

```
// Create the stream object  
  
FileStream fs = new FileStream(FileName, FileMode.CreateNew);
```

- Creating an Input Stream

```
// Create the input stream  
  
fs = new FileStream(Filename, FileMode.Open, FileAccess.Read);
```

Readers and Writers

While streams provide methods for reading and writing data, they manipulate data in the form of bytes. Use readers and writers to interpret the data in a more meaningful way.

- Provide points of access to streams
- Commonly used Readers/Writers

• BinaryReader/Writer	Read/write primitive types as binary
• TextReader/Writer	For character based input/output
• StreamReader/Writer	

- StringReader/Writer

Using a Reader/Writer

- Creating a Writer

```
// Create the writer for data  
  
StreamWriter w = new StreamWriter(fs);
```

- Creating a Reader

```
// Create the reader for data  
  
StreamReader r = new StreamReader(fs);
```

Example: Using a Reader/Writer

```
string file = "test.txt";  
FileStream fs;  
using (fs = new FileStream(file, FileMode.Create))  
using (StreamWriter w = new StreamWriter(fs))  
{  
    for (int i = 1; i <= 10; i++)  
    {  
        w.WriteLine(i);  
    }  
    w.Flush();  
}  
using (fs = new FileStream(file, FileMode.Open, FileAccess.Read))  
{  
    using (StreamReader r = new StreamReader(fs));  
    while (!r.EndOfStream)  
    {  
        Console.WriteLine(r.ReadLine());  
    }  
}
```

File I/O

FileStream Class

The FileStream class handles File I/O

Constructor takes 3 arguments:

- FileMode
- FileAccess
- FileShare

Security is checked when the stream is opened.

File Enumerations

There are several enumerations for file attributes.

- FileMode
 - Open
 - Append
 - CreateNew
- FileAccess
 - Read
 - Write
 - ReadWrite
- FileShare
 - Controls access other FileStreams can have to the same file
 - None
 - Read
 - Write
 - ReadWrite

Visualizers



Visualizers can display data in a variable. There are three built-in visualizers in Visual Studio.

- Text
- XML
- HTML

Example: Visualizers

```
class Program
{
    static void Main(string[] args)
    {
        using (StreamReader reader = File.OpenText(@"..\..\BookList.xml"))
        {
            string completeXML = reader.ReadToEnd();

            // When breakpoint hit, hover mouse over completeXML
            // Select dropdown next to magnifying glass to see visualizers
            // Choose XML visualizer to see XML document
            Console.WriteLine("Received: {0}", completeXML);
            Console.ReadLine();
        }
    }
}
```

File and Directory Manipulation



This section on file and directory manipulation is included for reference and will not be covered on the exam.

Manipulating Files

Two classes used for file manipulation.

- File - static
- FileInfo - instance

Both have similar functionality

- Create
- Copy
- Delete
- Move
- Open

Example: Reading Text from a File

```
public class TextFromFile
```

```
{  
    private const string FileName = "C:\\temp\\MyFile.txt";  
  
    public static void Main(String[] args)  
    {  
        if (!File.Exists(FileName))  
        {  
            Console.WriteLine("{0} does not exist!", FileName);  
            Console.ReadLine();  
            return;  
        }  
  
        StreamReader sr = File.OpenText(FileName);  
        string input;  
        while ((input=sr.ReadLine())!=null)  
        {  
            Console.WriteLine(input);  
        }  
        Console.WriteLine ("The end of the stream has been reached.");  
        sr.Close();  
        Console.ReadLine();  
    }  
}
```

Example: Writing Text to a File

```
public class TextToFile  
{  
    private const string FileName = "C:\\temp\\MyFile.txt";  
  
    public static void Main(String[] args)  
    {  
        if (File.Exists(FileName))  
        {  
            Console.WriteLine("{0} already exists!", FileName);  
            Console.ReadLine();  
            return;  
        }  
  
        StreamWriter sw = File.CreateText(FileName);  
        sw.WriteLine("This is my file.");  
        sw.WriteLine("Write ints {0}, floats {1}, ...", 1, 4.2);  
        sw.Close();  
        Console.ReadLine();  
    }  
}
```

{}

Manipulating Directories

Similar to Files

- Directory - static
- DirectoryInfo - instance

Both have similar functionality

- Create
- Move
- Traverse directories

Example: Directory Lister

```
class DirectoryLister
{
    public static void Main(String[] args)
    {
        DirectoryInfo dir = new DirectoryInfo(".");
        foreach (FileInfo f in dir.GetFiles("*.*"))
        {
            string name = f.FullName;
            long size = f.Length;
            DateTime creationTime = f.CreationTime;
            Console.WriteLine(
                "{0,-12:NO} {1,-20:g} {2}", size, creationTime, name);
        }
        Console.ReadLine();
    }
}
```

Example: File Watcher

```
public class Watcher {
    public static void Main(string[] args)
    {
        // If a directory is not specified, exit program.
        if(args.Length != 1)
        {
            // Display the proper way to call the program.
            Console.WriteLine("Usage: Watcher.exe (directory)");
            Console.ReadLine();
        }
    }
}
```

```
        return;
    }

    // Create a new FileSystemWatcher
    // and set its properties.
    FileSystemWatcher watcher = new FileSystemWatcher();
    watcher.Path = args [0];

    /* Watch for changes in LastAccess and LastWrite
       times, and the renaming of files or directories */
    watcher.NotifyFilter = NotifyFilters.LastAccess | 
                           NotifyFilters.LastWrite |
                           NotifyFilters.FileName |
                           NotifyFilters.DirectoryName;

    // Only watch text files.
    watcher.Filter = "*.txt";

    // Add event handlers.

    // Size, system attributes, last write time, last
    // access time, or security permissions are changed
    watcher.Changed += OnChanged;
    watcher.Created += OnChanged; // A file or directory is created
    watcher.Deleted += OnChanged; // A file or directory is deleted
    watcher.Renamed += OnRenamed; //A file or directory is renamed

    // Begin watching.
    watcher.EnableRaisingEvents = true;

    // Wait for the user to quit the program.
    Console.WriteLine("Press '\\q\\' to quit the sample.");
    while (Console.Read() !='q');

}

// Define the event handlers.
public static void OnChanged(object source, FileSystemEventArgs e)
{
    // Specify what is done when a file is changed,
    // created, or deleted.
    Console.WriteLine("File: " + e.FullPath + " " + e.ChangeType);
}
```

```
public static void OnRenamed(object source, RenamedEventArgs e)
{
    // Specify what is done when a file is renamed.
    Console.WriteLine("File: {0} rename to {1}:",
                      e.OldFullPath, e.FullPath);
}
```

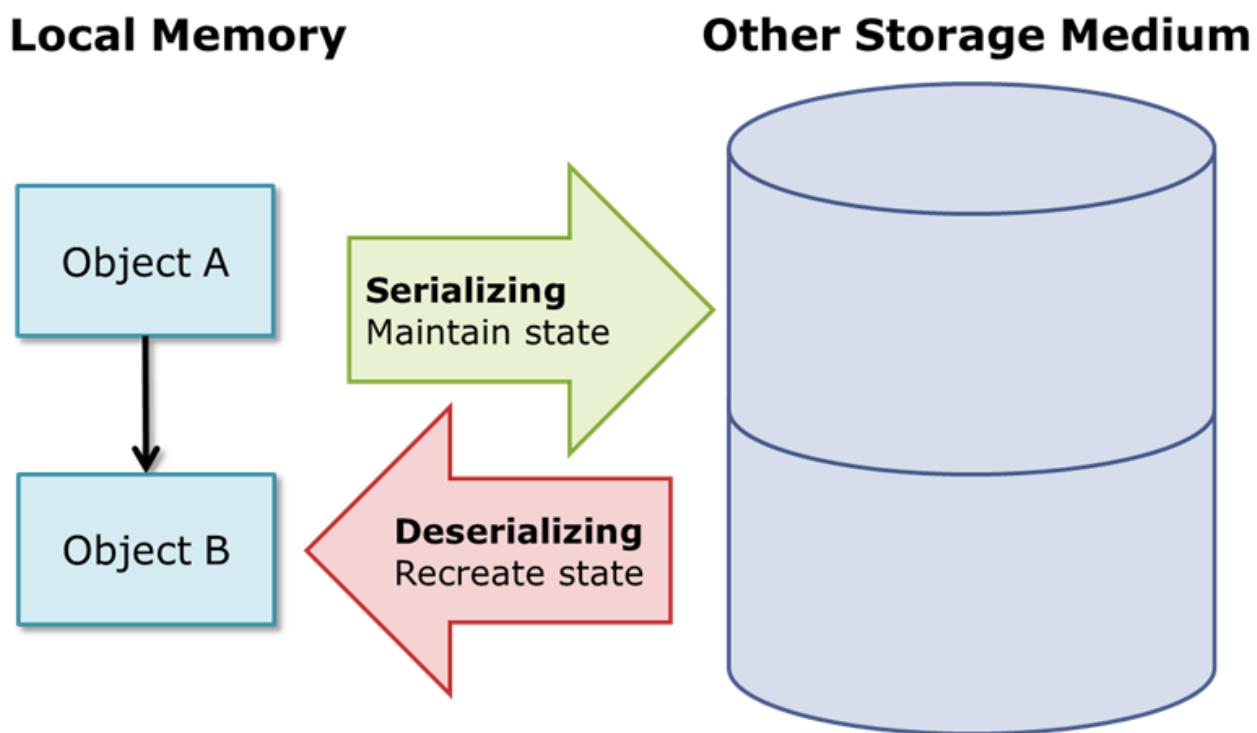
Activity 17: Streams

- Create a console application that uses the FileSystemWatcher object to watch a directory for changes.
- Log these changes to files.
 - When a file is renamed it should be logged to a file called rename.log
 - All other changes should be logged to a file called filechange.log
- Watch for changes in LastAccess and LastWrite times, and the renaming of files or directories.
 - Hint: Use the NotifyFilter property of the FileSystemWatcher class.
 - To use more than one filter use the bitwise or operator | to "or" the filters together
- To catch the changes as they happen you will have to subscribe to some events of the FileSystemWatcher class.
- Test the file I/O

Saving and Loading Objects

Streams also play a vital role in saving object state to a storage medium, a process known as **serialization**. Serialization is handled by the CLR, and accessible via formatter classes.

The Big Picture



- Serializing
 - Memory to somewhere else
 - Maintain object state
- Deserializing
 - Somewhere to memory
 - Recreate object state

The Process of Serialization

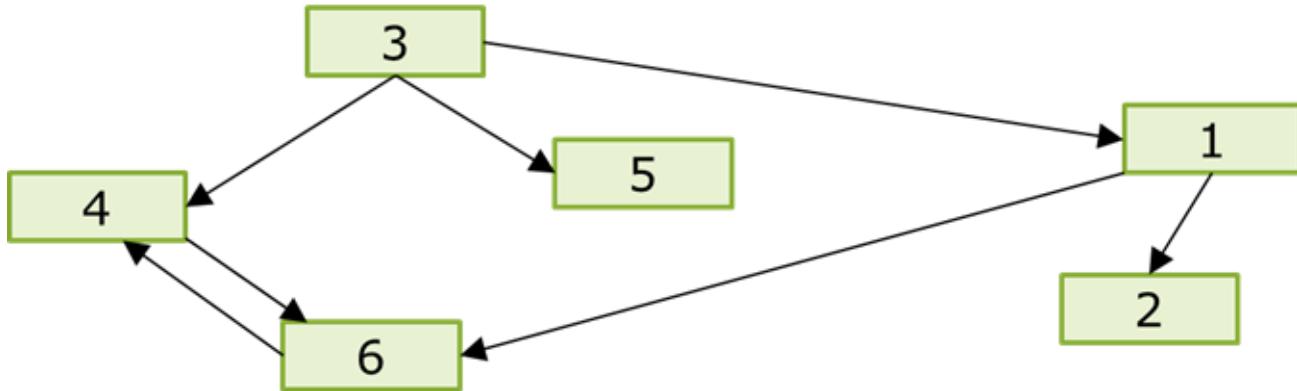
Things to figure out:

- What _____ and their _____?
- What _____ of objects?
- What _____ to serialize in?

Relationships are critically important as pointers between objects must persist to preserve the state of the objects.

Object Graph

The diagram below represents a simple graph of a set of objects and their references to other objects.



- Serialization allows you to save off parts of an object
- Use attributes to mark fields/classes with metadata for serialization

Attributes

Attributes add _____ to classes. Code can then examine values of attributes and act on them.

Applying Attributes

Attributes may be placed in front of:

- _____
- _____
- _____
- _____
- _____

Syntax

```
[attribute(positional-parameters, named-params=value,...)] [...]element
```

Example (built-in)

```
[Serializable]
public class MyClass {}  
  
[NonSerialized]
int _size;
```

Epic Written Example

```
[DataSynchronizationContract]
public class MyClass  
  
[DataSynchronizationMember]
int Size {get; set;}
```

Predefined and Custom Attributes

There are a number of predefined attributes. For example, `Conditional`, `Localizable`, `STAThread`, `Serializable`, `NonSerialized` to name a few.

Custom attributes may also be defined by creating a class which inherits from `System.Attribute`.

Querying Metadata Using Reflection



Querying metadata can be done through `Reflection` which is in the `System.Reflection` namespace. It enables an application to determine class information. Specifically we can use `System.Reflection.MemberInfo` to query the metadata of a class.

Reflector is a class browser, decompiler, and XML documentation browser for .NET components. It is a third party tool that is available for free. It is not a part of Visual Studio but does require some license to run.

Serialization Attributes

You may find the following attributes useful when working with serialization:

Attribute	Applies to	Purpose
<code>[Serializable]</code>	Classes (required) or fields (optional)	Indicates the class can be serialized. Can optionally mark specific fields to serialize along with the class. However, fields not marked as <code>[Serializable]</code> are serialized by

Attribute	Applies to	Purpose
		default.
[NonSerialized]	Fields (optional)	Mark specific fields that should not be serialized along with the instance.
[OnSerializing]	Methods - The method must have one parameter of type StreamingContext	Indicate a method that runs immediately before serialization takes place.
[OnSerialized]	Methods - The method must have one parameter of type StreamingContext	Indicate a method that runs immediately after serialization takes place.
[OnDeserializing]	Methods - The method must have one parameter of type StreamingContext	Indicate a method that runs immediately before deserialization takes place.
[OnDeserialized]	Methods - The method must have one parameter of type StreamingContext	Indicate a method that runs immediately after deserialization takes place.



What is StreamingContext?

It describes the source and destination of a given serialized stream, and provides additional caller-defined context. See:

<http://msdn.microsoft.com/en-us/library/system.runtime.serialization.streamingcontext.aspx>

```
[Serializable]
public class Item
{
    // Serialized by default
    private int _serializedField;

    [NonSerialized]
    private int _calculatedField;
}
```

```
[OnSerializing]
private void OnSerializing(StreamingContext context)
{
    // Unsubscribe events, other cleanup...
}

[OnDeserialized]
private void OnDeserialized(StreamingContext context)
{
    _calculatedField = _serializedField + 5;
}
```



The [Serializable] attribute is not inherited

Even if you derive from a serializable class, you still need to mark your class as [Serializable].

Serialization Classes

2 classes needed

- Stream class to read/write from/to
 - FileStream
 - MemoryStream
 - NetStream
- Formatter class - to structure properly
 - Binary
 - SOAP - Simple Object Access Protocol

Example: Serialization

```
class SerializeExample
{
    public static void Main(String[] args)
    {
        // create the object graph
        List<int> l = new List<int>();
        for (int x=0; x<100; x++)
        {
            l.Add (x);
```

```
}

// create the filestream
FileStream s = File.Create("c:\\temp\\foo.bin");
// create the Binary Formatter
BinaryFormatter b = new BinaryFormatter();
// serialize the graph to the stream
b.Serialize(s, l);
s.Close();
Console.ReadLine();
}

}
```

Example: Deserialization

```
class DeSerialize
{
    public static void Main(String[] args)
    {
        // open the filestream
        FileStream s = File.OpenRead("c:\\temp\\foo.bin");

        // create the formatter
        BinaryFormatter b = new BinaryFormatter();
        // deserialize
        List<int> p = (List<int>) b.Deserialize(s);
        s.Close();
        // print out the new object graph
        PrintValues(p);
        Console.ReadLine();
    }

    public static void PrintValues(IEnumerable myList)
    {
        foreach (object o in myList)
        {
            Console.WriteLine("{0}", o.ToString());
        }
    }
}
```

Custom Serialization

- More control over what is serialized
- Implement ISerializable
 - To serialize: GetObjectData
 - To deserialize: Populate a SerializationInfo object and pass it to the constructor

Example: Custom Serialization



In order to run this sample you must include a reference to:
System.Runtime.Serialization.Formatters.Soap

```
public class Program
{
    public static void Main()
    {
       ToFile();
       FromFile();
       Console.ReadLine();
    }

    public static void ToFile()
    {
        Console.WriteLine("ToFile");
        ExampleObject fooOut = new ExampleObject();
        fooOut.i = 1;
        fooOut.j = 20;
        fooOut.k = 50;

        Console.WriteLine("i: {0}", fooOut.i);
        Console.WriteLine("j: {0}", fooOut.j);
        Console.WriteLine("k: {0}", fooOut.k);

        IFormatter objFormatterToStream = new SoapFormatter();
        Stream toStream = new FileStream(
            "myFoo.xml", FileMode.Create, FileAccess.Write, FileShare.None);
        objFormatterToStream.Serialize(toStream, fooOut);
        toStream.Close();
    }

    public static void FromFile()
    {
        Console.WriteLine("FromFile");
        IFormatter objFormatterFromStream = new SoapFormatter();
```

```
Stream fromStream = new FileStream(
    "myFoo.xml", FileMode.Open, FileAccess.Read, FileShare.Read);
ExampleObject fooIn =
    (ExampleObject) objFormatterFromStream.Deserialize(fromStream);
fromStream.Close();

Console.WriteLine("i: {0}", fooIn.i);
Console.WriteLine("j: {0}", fooIn.j);
Console.WriteLine("k: {0}", fooIn.k);
}

}

[Serializable()]
public class ExampleObject : ISerializable
{
    public int i { get; set; }
    public int j { get; set; }
    public int k { get; set; }

    public ExampleObject()
    {

    }

    internal ExampleObject(SerializationInfo si, StreamingContext c)
    {
        // Restore our scalar values.
        i = si.GetInt32("i");
        j = si.GetInt32("j");
        k = si.GetInt32("k");
    }

    public void GetObjectData(SerializationInfo si, StreamingContext c)
    {
        si.AddValue("i", i);
        si.AddValue("j", j);
        si.AddValue("k", k);
    }
}
```

Activity 18: Serialization

- Provide the CompanySim solution with load/save functionality using serialization.
- Use the formatter of your choice.

Lesson 10: Communicating with the Database

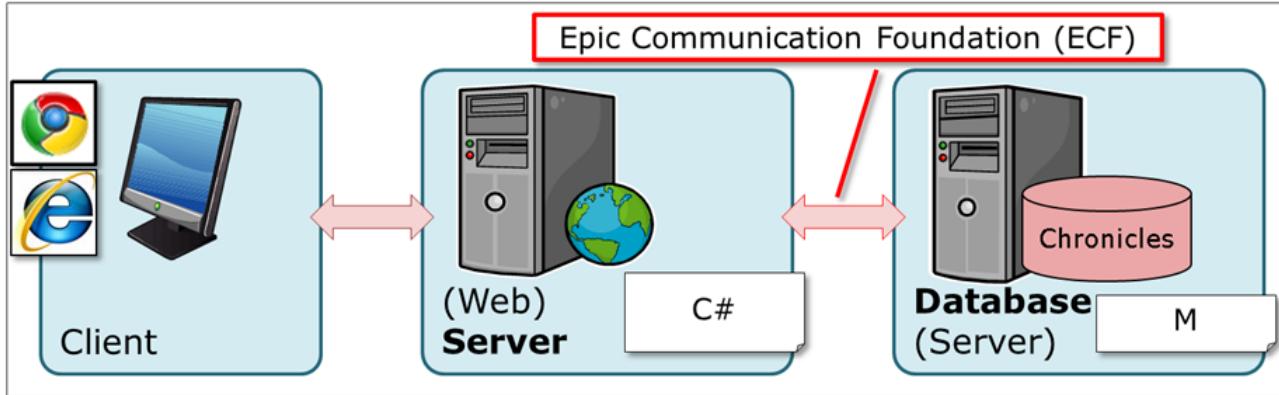
The Big Picture.....	10•3
Scenario.....	10•3
Using Basic ECF API.....	10•5
ECF Basics	10•5
Bulk Name Change.....	10•6
Dictionary of Values.....	10•7
Returning an Array of Objects.....	10•8
Exercise: Experimenting with ECF.....	10•9
Part 1: Connect to the Environment	10•9
Part 2: Create an ECF Command.....	10•10
Part 3: Prepare a new Console Application.....	10•11
Part 4: Setup your request and response classes	10•13
Part 5: Finish Your M Routine.....	10•15
Wrap up	10•16
If you have time	10•16
The DateOnly Attribute	10•17
Using the BulkRPC Wrapper.....	10•18
Executing a BulkRPC	10•18
Converting the Result to Objects	10•20
Exercise: Using BulkRPCs over ECF	10•21
Part 1: Classes	10•21
Part 2: Get the Cheeses	10•22
If you have time	10•23
Using Generated Code	10•24
Overview.....	10•24
How to Use.....	10•24
Basic Examples	10•24
Get Employee Name.....	10•24
Bulk Name Change	10•26
Syntax.....	10•27
Overall Workflow	10•28

Command / INI Level	10•29
Questions	10•33
ID / DAT Level	10•33
Workflow Summary.....	10•34
Activity: String Search.....	10•36
Sample Solution	10•37
Item Level.....	10•38
Workflow Summary.....	10•39
Basic Items.....	10•40
Related Groups	10•42
Property Groups	10•43
Fields.....	10•45
Example Routines.....	10•46
Template routines.....	10•46
Small data sets in Chronicles	10•47
Large data sets in Chronicles.....	10•47
Data from Globals.....	10•48
Exercise: Working with ECF CDO	10•49
Part 1: Read notes on programming points	10•49
Part 2: Use an ECF CDO	10•51
Part 3: Modify your C# Class.....	10•52
Part 4: Load the remaining items.....	10•52
Working with Chronicles	10•54
Access Manager	10•54
User Settings	10•54
Database Operations.....	10•54
Chronicles Database Definitions.....	10•54

Communicating with the Database

The Big Picture

VB Hyperspace uses EpicComm over ECF (Epic Communication Foundation) to communicate with the database. Hyperspace Web will use pure ECF.



ECF is used for server-database communication

ECF is more modern, secure, portable and is faster than EpicComm, so ECF should be used for all new development on the web. When transitioning existing activities, RPCs and BulkRPCs may be reused by creating a wrapper class, but transactions must be completely replaced by ECF.

For more information on the ECF protocol, see:
[http://wiki.epic.com/main/Coding_Standards/Epic_Communication_Foundation_\(ECF\)/Protocol](http://wiki.epic.com/main/Coding_Standards/Epic_Communication_Foundation_(ECF)/Protocol)

Scenario

We will use a big-mouse cheese factory as a scenario in this part of class. There is a Chronicles database that is used to store cheese information, XCH:

Item #	Item Name	Data	Add	Resp	Idx	Packed
1000	COUNTRY	C	N	S	Y	1000;1
1010	WEIGHT	N	N	S	N	1000;2

Item #	Item Name	Data	Add	Resp	Idx	Packed
1020	PRICE	N	N	S	N	1000;3
2000	DESCRIPTION	S	N	M	N	2000;1;1
3000	IMAGE PATH SMALL	S	N	S	N	3000;1
3010	IMAGE PATH LARGE	S	N	S	N	3000;2

The goal of this scenario will be to load only cheeses of a particular country, as listed in the category item XCH - 1000. The three most common values for this category are:

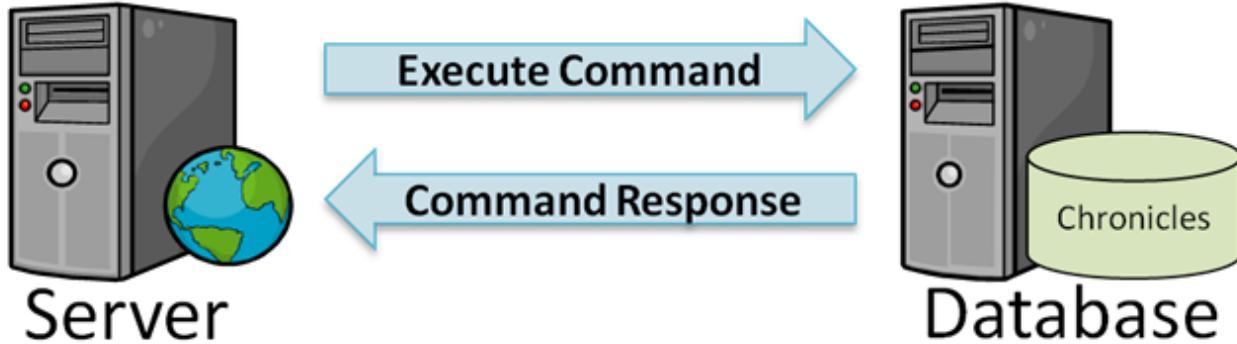
- United States of America - 1
- France - 61
- United Kingdom - 182

The regular item index will be used to look up the appropriate cheese records.

Using Basic ECF API

ECF is based on the serialization of .NET classes to and from an M environment. An ECF command is a pre-determined tag^routine used to save data to or load data from the database.

ECF Basics



The basic unit of ECF communication is the command

Below is a simple example of an ECF request to look up the name of an employee record:

Request Class:

```
class Request
{
    public string Id { get; set; }
}
```

Command Code:

```
GetName n id, name
s id=$$zECFGet("Id") ;Request object sent as part of command
i id="" q
s id=$$getidin(id,"EMP")
s name=$$znam("EMP",id)
s %=$$zECFSet("Name",name) ;Response object populated
q
```



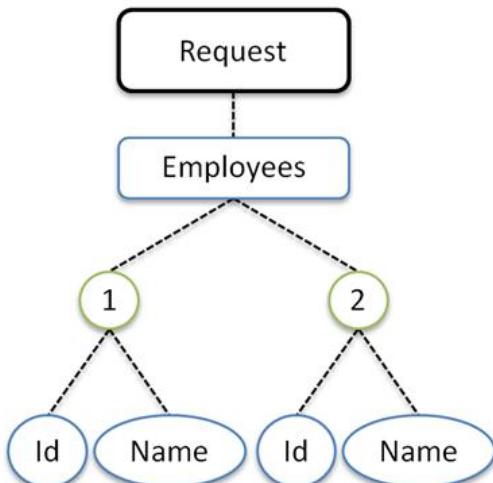
\$\$zECFGet returns "" both when there is a null value for a property, AND when the property is omitted from a request. To distinguish between the two cases, \$\$zECFGetWasSent returns 0 when a property is omitted and 1 when the property is included with the value of null.

Response Class:

```
class Response
{
    public string Name { get; set; }
}
```

Bulk Name Change

Command requests can be considerably more complex than the previous example. For example, a bulk name change. In this case, the request is a list of employees, instead of a single employee. The request could appear like this:



Request object for a bulk name change

```
class Request
{
    public List<Employee> Employees { get; set; }
}

class Employee
{
    public string Id { get; set; }
    public string Name { get; set; }
}
```

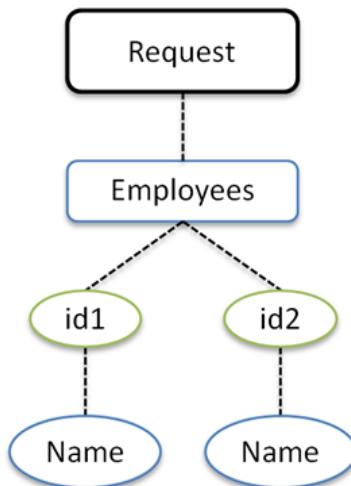
Command Code:

```
UpdateNames n ln,employee,id,name
```

```
f ln=1:1:$zECFNumElmts ("Employees") d
. s employee=$zECFGetElmt ("Employees", "", ln)
. s id=$zECFGet ("Id", employee)
. s name=$zECFGet ("Name", employee)
. ; Code to change record names...
```

Dictionary of Values

Instead of a list of objects, another approach is to use a dictionary of key-value pairs. In this case, the key is the EMP record ID, and the value is the new employee name.



Example of a dictionary request

Request class:

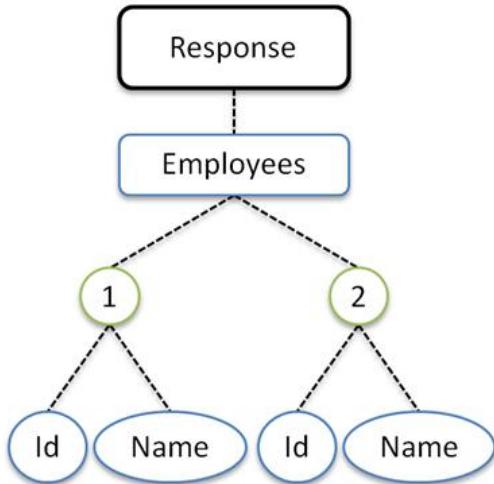
```
class Request
{
    public Dictionary<string, string> Employees { get; set; }
}
```

Command code:

```
UpdateNames n id, name
f s id=$zECFDctNxKey ("Employees", "", id) q:id="" d
. s name=$zECFGetElmt ("Employees", "", id)
. ; Code to change record names...
```

Returning an Array of Objects

The response to a command often needs to be more than a single object. For example, returning a list of records that match a query. In this case, you need to create a new collection and new objects within that collection.



Sending a complex response

```

class Response
{
    public List<Employee> Employees { get; set; }
}

class Employee
{
    public string Id { get; set; }
    public string Name { get; set; }
}
  
```

Command code:

```

GetEmployees n ln,employeeList,id,name,employee
  s employeeList=$$zECFNew("Employees","","","A")
  f  s id=$$zoID("EMP",id) q:id="" d
  . s employee=$$zECFNewElmtObj(employeeList)
  . s %=$$zECFSet ("Id",id,employee)
  . s %=$$zECFSet ("Name",$$znam("EMP",id),employee)
  
```



To return into an array a set of primitive types, like `int` or `string`, do not use `zECFNewElmtObj`. Instead use the `zECFSetElmt` function and omit the key parameter. This will cause the function to assume that the property it is setting into is a list and create a new element with the value specified.

Similarly if the response is a dictionary of primitives (for example `Dictionary<string, string>`) include the key parameter as the key for the dictionary element.

Exercise: Experimenting with ECF

The goal of this activity is to create a console application that communicates with Chronicles via ECF. The application will load records from the Cheese master file (XCH).

Part 1: Connect to the Environment

In this section, you'll verify that you can access the environment and edit code.



If you have not taken tech camp, then skip these and any future steps that have you edit an M routine. We'll provide you with an alternative routine that you can use.

1. Connect a text session to the environment that HyperspaceWeb connects to:
 - a. Host: **epic-fndlabs2**
 - b. Log in with your Windows username and password for the first set of login prompts.
 - c. Type **epicmenu** and choose either **DVA** or **DVB** corresponding to Foundations Lab Development A or B.
 - d. Use "dev" as the username and password for the second set of login prompts.
 - e. Use "1" for the user ID and "epic" for the user password for the third set of login prompts.
2. Create a new routine called X<TLG>, where <TLG> is your EMP record ID.



Make sure to create the routine in the correct environment. It **should** be in **Foundations Lab Dev A (FNDLABDVA)** or **Foundations Lab Dev B (FNDLABDVB)**.

This is not the same as the exercise environments you are familiar with tech camp. You **should not** use **Foundations Exercises A (FNDEXA)** or **Foundations Exercises B (FNDEXB)**.

3. Add a tag called GetCheeses.

4. For now, just add the following lines:

- GetCheeses
 - k ^X<TLG>
 - s ^X<TLG>=1
 - q

Part 2: Create an ECF Command

In this part you will create a command record that points to the routine you created in the previous step.

1. Copy the following folder:

- I:\Internal\Advanced\C Sharp\CommandBuilder

2. Paste the folder somewhere convenient, such as:

- C:\EpicSource\

3. Run is script to mark your machine as an Epic development computer:

- M:\fnd\FndGUI\Reg-EpicDeveloper.bat



For more details on getting the Command Builder, see the wiki:

[http://wiki.epic.com/main/Coding_Standards/Epic_Communication_Foundation_\(ECF\)/CommandBuilder](http://wiki.epic.com/main/Coding_Standards/Epic_Communication_Foundation_(ECF)/CommandBuilder)

4. Run the command builder. It will be located here:

- CommandBuilder\CommandBuilder.exe

5. Connect to fndladv<A/B>

6. Log in using "1" for the user ID and "epic" for the user code

7. Left-click **New Command**

8. Choose **ECF Command (Strongly-Typed)**



If you don't get the option for **ECF Command (Strongly-Typed)**, this means you did not properly register your computer as an Epic development machine. Attempt to run this script again:

M:\fnd\FndGUI\Reg-EpicDeveloper.bat

9. Call the command Train.<username>.GetCheeses, where <username> is your windows user name

- Be sure to replace <username> with your username.

10. Set **Entry point:** to GetCheeses^X<TLG>
11. Add a **Description**
12. If it is blank, select any **Owning Application**



If you did not create a routine in the previous step, use GetCheeses^XCH instead.

13. Click the **Save** button
14. Close the command builder.

Part 3: Prepare a new Console Application

Don't worry about the path or namespace of this project.

1. Start Visual Studio
2. New > Project > Visual C# > Console Application
 - Name: "SimpleEcf<TLG>"
3. Right click on the **References** folder of your solution
4. Click **Add Reference**
5. Add the following Epic references using the Browse button on the bottom right of the dialog window:

Assembly	Path to binary	Purpose
Epic.Core	CommandBuilder \	Contains the custom Epic serialization attributes used with ECF.
Epic.Core. Communication	CommandBuilder \	Contains the components necessary to establish ECF connections and execute commands.

6. Add three new class files to your project (For now, don't worry about namespaces):
 - Request.cs
 - CheeseList.cs
 - Cheese.cs
7. In the main method of Program.cs, add the following code:

Note: Be sure to use the keyboard shortcut **CTRL + .** to resolve unknown classes and attributes. This

will automatically pull in the required Using statements, as long as the necessary references are present.

Also, ensure you choose the correct environment below (FNDLABDVA or FNDLABDVB):

```
Connection connection = Connection.Create("FNDLABDV<A/B>",
CacheLicensingModel.User);
connection.Login("1", "epic");

// Add code to call command here

connection.Release();
Console.WriteLine("<Press any key to continue>");
Console.ReadKey();
```

8. Compile and run your application. You should see the following Exception:

Epic.Core.Communication.ConfigurationException:

"Unable to locate the connection configuration file at ..."

9. Stop the application

10. Create a new folder called Config in your project.

11. Add all config files from the CommandBuilder to your application:

- Solution Explorer > SimpleEcf<TLG> > right click > Add > Existing Item
- They are located in:
 - CommandBuilder\Config\
 - When looking for these files, be sure to set the filter to All Files (*.*)

12. The following steps will ensure that the configuration files are available to the compiled version of the console application:

- a. Right-click AppConnection.config in the solution explorer and select **properties**.
- b. Change the **Copy to Output Directory** property to **Copy always**.
- c. Repeat the previous two steps for all remaining config files.



This ensures that the configuration files are moved to the location where the executable is produced, so when it runs it will find those files. If you skip this step, then you'll keep getting the "Unable to locate configuration file at..." message.

13. Run your application again. You should not get an exception this time.

14. Add the following class attribute to the Request, CheeseList and Cheese classes:

```
[DataSynchronizationContract(  
    ClientSynchronizationMode.OnServerOnly,  
    DatabaseSynchronizationMode.Automatic)]
```

The first argument of this new attribute sets the **default** client-to-web-server synchronization mode, while the second does the same for the web-server-to-database synchronization mode. All properties marked with DatabaseSynchronizationMember will use these database/client synchronization modes unless a different one is specified.

Since we are creating a console application, the class is setup not to go to the client (ClientSynchronizationMode.OnServerOnly), but it will synchronize to the database (DatabaseSynchronizationMode.Automatic).



Synchronization modes will be covered in more detail during week 4 of Web Tech Camp.

15. Add the following to Main, after the connection is established, but before it is released:

```
// Add code to call command here  
Command<Request, CheeseList> cmd = new Command<Request, CheeseList>(  
    "Train.<your username>.GetCheeses", connection);  
Request req = new Request();  
cmd.Request = req;  
cmd.Execute();
```

16. Compile and run your project.

17. Verify that your global was set by the command:

```
FNDLABDV>w ^X<TLG>
```

```
1
```



If you are using GetCheeses^XCH instead of your own routine, then don't worry about checking the global.

Part 4: Setup your request and response classes

In this part you will setup the business objects so that they can communicate the required information.

1. In VisualStudio, edit the Request class:
 - Add a string property called **CountryValue**.
 - This will be used to search the item index of XCH for cheeses that are available in a given country.
2. Mark the property with **DataSynchronizationMember**

```
[DataSynchronizationMember]
public string CountryValue { get; set; }
```

[DataSynchronizationMember] is Epic's replacement for Microsoft's [DataMember] attribute. It has several advantages, including enhanced serialization performance and tighter property encapsulation.

3. Add a property to CheeseList of type ObservableCollection<Cheese>:

```
[DataSynchronizationMember]
public ObservableCollection<Cheese> Cheeses { get; internal set; }
```

4. Add string properties to Cheese for Id and Name. Also add a ToString method to display its properties:

```
[DataSynchronizationMember]
public string Name { get; internal set; }

[DataSynchronizationMember]
public string Id { get; internal set; }

public override string ToString()
{
    return Name + ", " + Id;
}
```

5. In the Main method, prompt the user for the country, then set that value to CountryValue property of the command's request object, req.
6. After executing the command, iterate over the returned Cheese objects and output them to the console:

```
foreach (Cheese cheese in cmd.Response.Cheeses)
{
```

```
Console.WriteLine(cheese.ToString());  
}
```



Before moving on to the next part, double check that all properties that you want to communicate between the server and database are:

- Marked with [DataSynchronizationMember]
- Contain get and set accessors

Part 5: Finish Your M Routine



- If you get stuck while working on your routine, take a look at GetCheeses^XCH, but don't modify it.
- If you are not creating your own routine, then you should be ready to test your code.

1. Remove the line that writes 1 to your global. You can continue to use your global to debug/trace your routine as necessary.
2. In GetCheeses^X<TLG>, get the CountryValue property passed with the Request class.
3. Use the regular item index to identify all matching cheese records. You may find the following functions useful (the country item is XCH-1000):
 - \$\$znxIxID(ini, item, val, id) ; loop on ids with the given item value
4. Add matching cheeses to the response, along with their Ids and names.
 - Ask the instructor if you are unsure how to do this.
5. Run your console application. Try the following values

Country	Value
United States of America	1
France	61
United Kingdom	182

6. Your output should resemble the following:

```
Enter a country (1-US, 61-FR, 182-UK): 1
```

```
BLUE US, 175
```

```
CHEDDAR US, 225
SWISS US, 325
AMERICAN US, 449
FETA US, 594
LIMBURGER US, 670
GOUDA US, 797
MOZZARELLA US, 857
...

```

<Press any key to continue>

7. Answer the wrap-up questions.

Wrap up

1. True or False: It is important that you limit request classes to only include properties that are necessary for the request.
 - _____
2. True or False: It is important that you limit response classes to only include properties that are necessary for the response.
 - _____
3. What is the significance of having to create a command before it can be used?
 - _____

If you have time

1. Change your routine so that if no country is provided, all cheeses are displayed.
 - You can use `$$zOID(ini, id)` to loop over all cheeses.
2. Rather than providing a user id and password in your console application, require the user to log in.
 - The Login method returns a LoginResult object. You can check the Result property against `LoginStatus.Success`

- The Errors property contains a message that you can display if the login fails.

The DateOnly Attribute

In addition to using [DataSynchronizationMember] on properties of your class that you intend to serialize to/from the database, there is an additional attribute for Date items: [DateOnly]. This attribute is used to correctly serialize the days since 12/31/1840 to a DateTime property (which normally expects both a date and time component).

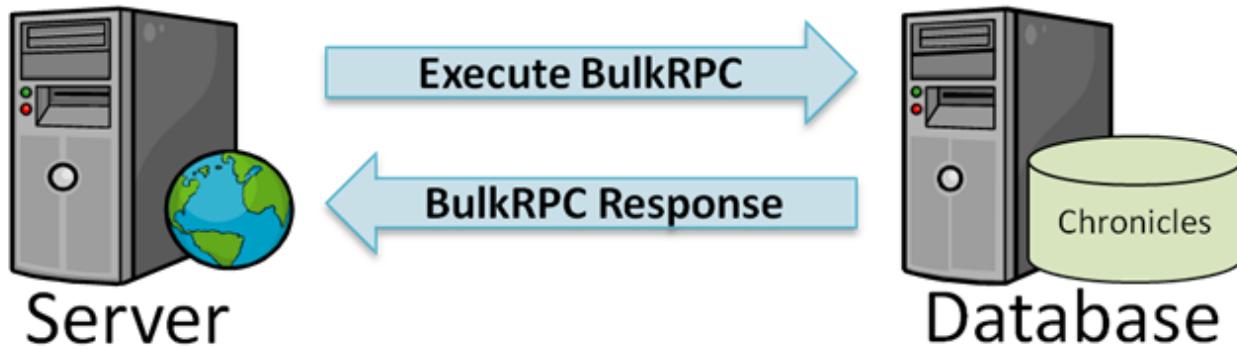


If you fail to use [DateOnly], ECF will convert the DateTime object to UTC when sending to the server (and back from UTC when coming from the server). This can lead to dates displaying a day off on the client, if the local time zone is to the west of UTC. Accidentally leaving this attribute off when saving a date to Chronicles can introduce a data integrity issue, as you will be saving an instant into a Chronicles date item.

Using the BulkRPC Wrapper

During the transition from VB to the Web, the primary goal is to complete the transition without losing any functionality or introducing additional bugs. To this end, the BulkRPC wrapper is provided so that a majority of the Server code can be reused.

This is not to say that no additional code is required. You will still need to write the web-server code to execute the BulkRPC and consume the resulting data.



BulkRPCs are also used for server-to-database communication

Executing a BulkRPC

A BulkRPC request consists of a delimited string of fields, and possibly sub-fields and custom fields. When sending .NET classes to the database, this amounts to requiring the developer to write the code to serialize the class to a delimited string manually (unlike ECF commands, which handle the serialization automatically).

Suppose you want to send the following message to the server:

- Field 1, standard subfield delimiter of \$C (6)
 - Subfield: "1"
 - Subfield: "2"
 - Subfield: "3"
- Field 2, custom subfield delimiter of " | "
 - Custom field: "X"
 - Custom field: "Y"
 - Custom field: "Z"

The following code will generate the desired fields and execute the BulkRPC:

```

string result;
FieldList fl = new FieldList() {
    new SubFieldList() { "1", "2", "3" },
    new CustomFieldList(" | ") {"X","Y","Z" } };
connection.BulkRpc("D tag^Routine", out result, fl);

```

Once the message is sent to the database, the same code that was used previously can be used to split up the various kinds of fields, as long as the message format matches the same request that VB would send.

```

tag n sfl,cfl
s sfl=$$zBulkNxF() ;1_C6_2_C6_3
s cfl=$$zBulkNxF() ;X|Y|Z
s %=$$zBulkRep("I" _C6_ "II" _C5_ "a|b")
q

```

The result returned from the database is also a delimited string. It can be parsed using the `ResultList` class:

```

ResultList fields = new ResultList(result, ResultType.Field);
ResultList subFields =
    new ResultList(fields.Piece(1), ResultType.Subfield);
ResultList customFields = new ResultList(fields.Piece(2), " | ");
foreach (Result customField in customFields)
{
    Console.WriteLine(customField.Value);
}

```

<code>ResultList(result, ResultType.Field)</code>	Splits on \$C(5)
<code>ResultList(result, ResultType.Subfield)</code>	Splits on \$C(6)
<code>ResultList(result, " ")</code>	Splits on " "



For bulk RPCs that have numerous sub-levels, the best way to get a unique delimiter is to use:

```
s delim=$$zBulkDlm(level) ;Get delimiter character(s) for given level
```

To determine what that character is on the web server, use:

```
string delimiter = Connection.GetBulkDelimiter(level);
```

Converting the Result to Objects

The result of a BulkRPC request is, again, a delimited string. To make use of this result, the string must be converted to one or more instances of a .NET class. This can be accomplished by creating a factory class to handle the conversion.

In the following example, a list of employee IDs is sent to the database, and the result is a delimited string of employee information. An employee factory class is used to convert the resulting delimited string into a collection of employee objects. It works by first implementing the `IResultListClassFactory<Employee>` interface, which has a public method called `Create`. This method accepts a result (one entry of the delimited list) and returns a corresponding instance of type `Employee` based on that result:

```
public class EmployeeFactory: IResultListClassFactory<Employee>
{
    public Employee Create(Result r)
    {
        Employee anEmployee = new Employee();

        ResultList employeeItems =
            new ResultList(r.Value, ResultType.Subfield);
        anEmployee.Id = employeeItems.NextPiece();
        anEmployee.Name = employeeItems.NextPiece();

        return anEmployee;
    }
}
```

Each result corresponds to a field from the response. Within the result, the ID and name of each employee are stored in subfields. The `ResultList` class can make use of the new factory class with the `ToObservableCollection<TResultType, TResultTypeFactory>` method:

```
connection.BulkRpc("d GetEmployees^Example", out result,
```

```
employeeIdFields);  
ResultList resultList = new ResultList(result, ResultType.Field);  
  
ObservableCollection<Employee> employees;  
employees = resultList.ToObservableCollection<Employee, EmployeeFactory>()  
;  
  
Console.WriteLine("Employees:");  
foreach (Employee e in employees)  
{  
    Console.WriteLine(e.ToString());  
}
```

ToObservableCollection uses the factory to create an employee object for each result and returns a collection of the employees.



Why create a factory class?

Why create a factory class, rather than adding a constructor to Employee?

-

Exercise: Using BulkRPCs over ECF

In this exercise you will reuse an existing BulkRPC to load all cheeses in a given price range. Use the same solution from the previous exercise.

Part 1: Classes

In this section you will modify the Cheese class to add the Price property. Additionally, you will create the CheeseFactory class to convert a delimited list of subfields into a Cheese object.

1. Add a public decimal property named Price to the Cheese class.
 - Be sure to mark it with the attribute DataSynchronizationMember.
2. Modify ToString to also display the price if it is not zero.
 - Remember, the command you wrote in the previous exercise doesn't return Price, so you won't see it in the output of your previous solution. If you would like to add it to your code, it is item 1020 (though this is not required for this exercise).
3. Create a new class named CheeseFactory that implements the interface IResultListClassFactory<Cheese>

4. Implement the interface (ctrl + "."). It should create the following unimplemented method:

```
public Cheese Create(Result r)
{
    throw new NotImplementedException();
}
```

5. The Result passed to Create will be a list of three sub fields. Use the ResultList class of ResultType.SubField to break them up.
6. Create a new cheese object and assign the pieces to the correct properties, then return the new object. The pieces are as follows:

Piece 1	Piece 2	Piece 3
Id	Name	Price

Part 2: Get the Cheeses

- Comment out the code that displays all cheeses from a selected country.
- Prompt the user for a lower and upper price range.
- If the range is valid, call the BulkRPC. The request should be a FieldList with the first field being the lower price bound and the second being the upper price bound:

```
FieldList request = new FieldList() {
    lower.ToString(),
    upper.ToString()
};

string result;

connection.BulkRpc(
    "D GetCheesesInPriceRange^XCH",
    out result,
    request
);
```

4. Convert the result to an ObservableCollection<Cheese>

```
ResultList resultList = new ResultList(result);
ObservableCollection<Cheese> cheesesInRange;
cheesesInRange = resultList.ToObservableCollection<Cheese,
CheeseFactory>();
```

5. Iterate over the cheeses, writing them to the console.
6. Test your program using a range that contains cheeses. For example:
 - 1 - 100
 - 60 -80
7. Test your program using a range that contains no cheeses (1-5).
 - Fix any bugs that you find.
 - Hint: What happens when the variable `result` is empty?

If you have time

Read about the additional API available for the BulkRPCwrapper on the wiki:

[http://wiki.epic.com/main/Coding_Standards/Epic_Communication_Foundation_\(ECF\)/Wrapper_Classes_for_BulkRPC](http://wiki.epic.com/main/Coding_Standards/Epic_Communication_Foundation_(ECF)/Wrapper_Classes_for_BulkRPC)

Using Generated Code

In this section, you will learn how to use create Chronicles Data Operations (CDOs) to handle ECF commands.

Overview

There are several advantages to using generated code:



Why use generated code?

-
-
-

How to Use

As a programmer writing an ECF-CDO, you are responsible for specifying what data you want to save or load on the database (INIs, IDs, DATs, items, etc.) and how that data maps to objects and properties on the web server. You also need to create the ECF command and point it to the right routine and tag.

What you do not need to do is manually load the data from or store it to Chronicles. Also, you will not need to use much of the ECF API yourself, though you will need to use it from time-to-time for certain scenarios.

Basic Examples

The following examples illustrate how common tasks can be accomplished using an ECFCDO.

Get Employee Name

Suppose we want to load the name of an employee record given its ID. The request and response classes are the same as when wrote the M code was written by hand:

Request Class:

```
class Request
{
    public string Id { get; set; }
}
```

Response Class:

```
class Response
{
    public string Name { get; set; }
}
```

The difference with an ECF CDO is that you don't write any of the actual M code yourself. Instead, you need to provide a specially formatted comment block that is interpreted by the M compiler:

```
; ;#ECFGENERATE#
; Type: Load
; Tag: lemp
;INI: EMP
; ID: |Id
; Items:
;     .2|Name
; EndItems
; EndINI
; ;#ENDGEN#
```

; Type:	<p>Required.</p> <p>Either Load to get data from Chronicles or Save to store data in Chronicles.</p>
; Tag:	<p>Required.</p> <p>The name of the auto-generated entry point for the ECF command. It will also be a prefix to numerous auto-generated helper subroutines. Must be <= 4 characters in length because it is used as a prefix to various generated tags.</p>

When the routine is compiled, code similar to the following is generated within the intermediate routine:

```

ltemp ;
N ID
S ID=$$zECFGet("Id")
I ID' ]"" D ltempCLEAR G ltempDON
S %==$zECFSet ("Name", ^ER("EMP", ID))

ltempDON ;
Q

ltempCLEAR ;
S %==$zECFSet ("Name", "")
Q

```



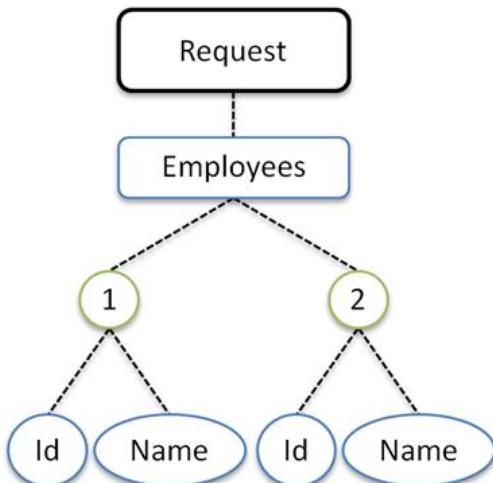
The generated M code has been simplified to illustrate how the ECFCDO functions

The actual generated code is more complex because it includes many additional features. For example, the actual generated code also includes:

- Checks to enforce EMFI item protection
- Generation of a tag that allows you to test the generated code from the M prompt
- NEWing of additional variables that are used as more INIs and items are processed

Bulk Name Change

It is also possible to process multiple records. Take the bulk name change example from earlier in the lesson. In this case we need to change the name of a set of employees. The request classes appear as follows:



Request object for a bulk name change

```
class Request
{
    public List<Employee> Employees { get; set; }
}

class Employee
{
    public string Id { get; set; }
    public string Name { get; set; }
}
```

The ECFCDO comment block to perform this task is as follows:

```
; ; #ECFGENERATE#
; Type: Save
; Tag: sEMP
; INI: EMP/M|Employees
;   ID: |Id
;   Items:
;     .2|Name
;   EndItems
; EndINI
; ; #ENDGEN#
```

Notice the /M flag following EMP. This directs the generated code to process multiple records. There are other flags available in the ECFCDO which will be covered later.

When multiple records are processed, the generated code expects the name of a collection following the pipe character. In this case, the collection name is Employees.

In the collection, each object has an Id property and a Name property.

Syntax

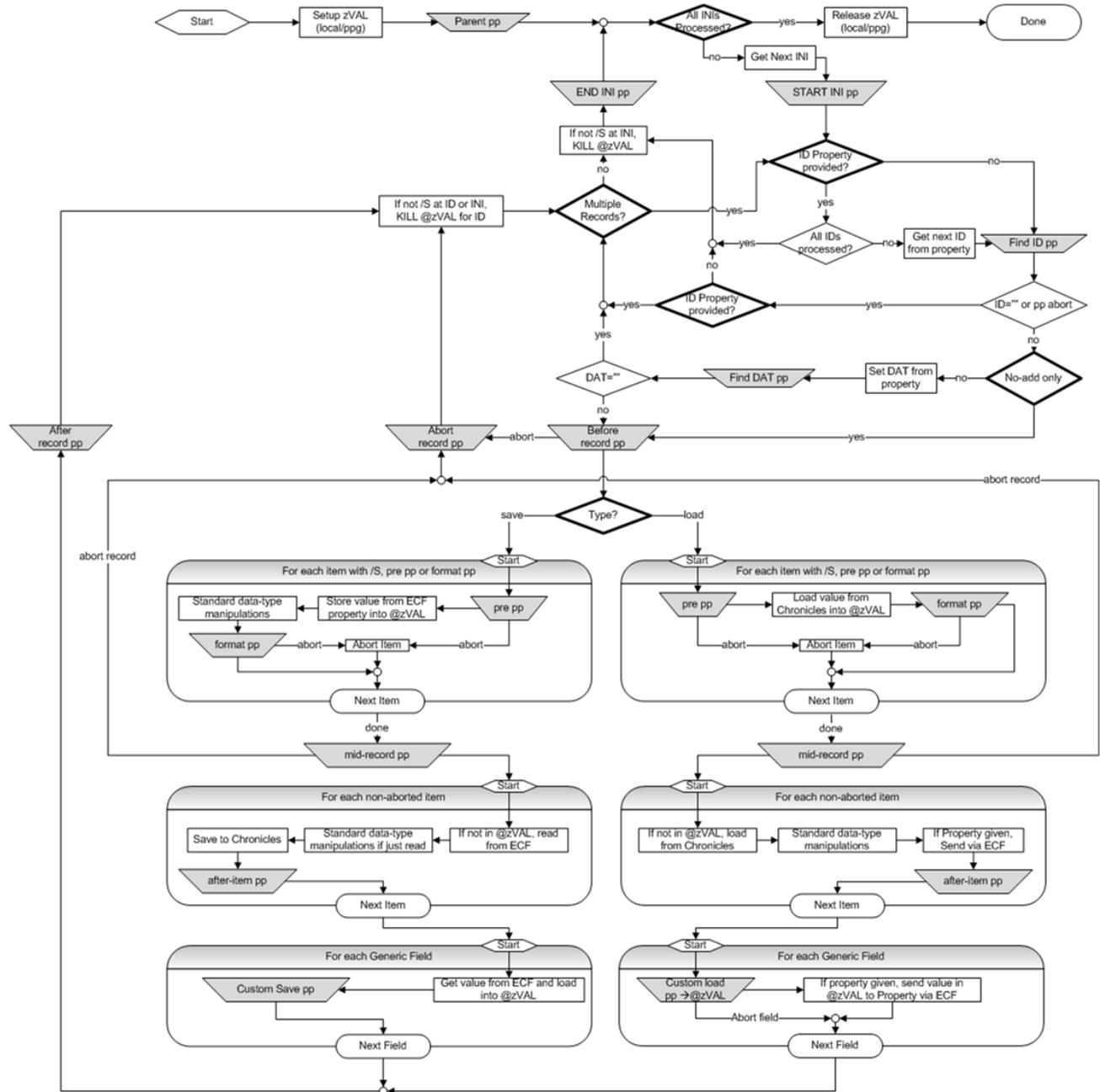
An illustration of the complete syntax is listed below, followed by a work-flow diagram. In the next several sections, each part will be explained in more detail.

```
; #ECFGENERATE#
; Command: commandName
; Type: Load/Save
; Tag: TagName
; Storage: Local
```

```
; Protection: No
; Parent: pp
;INI: XXX	flags|Property|START:pp~END:pp
;    ID: /flags|RequestProperty~ResponseProperty|FindIDpp
;    DAT: /flags|RequestProperty~ResponseProperty|FindDATpp
;    Rec: BEFORE:pp~AFTER:pp~MID:pp~ABORT:pp
;    Items:
;        Item#/flags|Property~commentProperty|PRE:pp~FRMT:pp~AFT:pp
;    EndItems
;    Fields:
;        #/flags|Property|Fieldpp|otherInfo|associatedItem#
;    EndFields
; EndINI
;#ENDGEN#
```

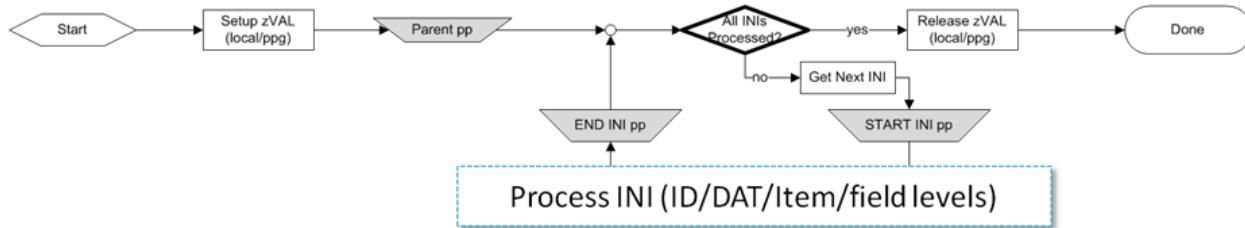
- Anything marked with "pp" in the above syntax represents a "programming point". In this case, a programming point is simply a location where you can specify a tag^routine to execute when a certain point during the ECFCDO is reached.
 - The tag^routine is called like a function, so be sure to include () next to the tag and quit with a value, even if no parameters are passed and the return value is not used.
- Any section marked as a Property refers to a property marked with [DataSynchronizationMember] on a .NET class.
- Flags are single-characters that add a specific feature at the level indicated.
 - For example, you can add the "C" class at the item level to indicate that the item is a category.

Overall Workflow



- Gray trapezoids are programming points, where the programmer can insert hooks to database code
- Bold diamonds are decisions made at compile time
- Regular diamonds are decisions made at run time

Command /INI Level



Comments to load a set of employees, followed by a set of providers:

```

;#ECFGENERATE#
; Command: Train.Instructor.ExampleEcfCdo
; Type: Load
; Tag: ld
; Storage: Local
; Protection: No
; Masking: No
; INI: EMP/MLS | Employees | START:startEmp~END:endEmp

... EMP comments (IDs/DATs/Items/Fields) ...

; EndINI
; INI: SER/M| Providers

... SER Comments (IDs/DATs/Items/Fields) ...

; EndINI
;#ENDGEN#

```

Code generated by the previous comments:

```

ld ;
    N ID, DAT, INI, ...
    N zVAL, tmpVAL S zVAL="tmpVAL"
ld1 ;
    S INI="EMP"
    S %=$$startEmp()
    S recsID=$$zECFNew("Employees","","L")
    ;
    ; iterate over records / items...
ld1DON ;
    S %=$$endEmp()
ld2 ;
    S INI="SER"
    S recsID=$$zECFNew("Providers","","L")
    ;
    ; iterate over records / items ...

```

```

K @zVAL@(INI, ID, DAT)
G ld2ID
ld2DON ;
ldDON ;
Q

```

;<#ECFGENERATE#>	Required. Begins a ECF CDO comment block. Can appear anywhere within the routine before ; ; #eor#
;<#ENDGEN#>	Required. Terminates an ECF CDO comment block.
; Command:	Optional. Indicates which E4C record (command) is used to call the code generated by this block. This is used for documentation purposes only.
; Type:	Required. Either Load to get data from Chronicles or Save to store data in Chronicles.
; Tag:	Required. The name of the auto-generated entry point for the ECF command. It will also be a prefix to numerous auto-generated helper subroutines. Must be <= 4 characters in length because it is used as a prefix to various generated tags.
; Storage: Local	Optional. Specify to use a local variable to load the data. Only use when the amount of data will not overflow the symbol table. Omitting this will cause data to be loaded into a process-private temporary global.
; Protection: No	Optional. (Yes by default to prevent items from being changed based on ownership rules)

	<p>If set to no, deactivates EMFI item protection and allows items to be changed.</p> <p>Use when the purpose of the load is display-only (e.g., reports), since the items will not be changed anyway and enforcing item protection can be expensive.</p> <p>To receive this information the C# class must implement the Epic.Code.Database.IReadOnlySupport interface.</p>
; Masking: No	<p>Optional</p> <p>Should only be used if you will not be displaying the data to the end-user. When data masking is turned off, the masked version of the data will not be sent to the client to use.</p>
; Parent: pp	<p>Optional.</p> <p>Programming point used to create an ECF parent ID by hand (using \$\$zECFNew). This is only required when the default root parent object doesn't directly contain properties set later. Specify the tag without parameters (none may be passed). Return the id of the created parent object.</p>
; INI	<p>Required.</p> <p>Specify the database initials being accessed. If only fields are loaded, XXX may be specified instead of an actual INI, in which case only generic fields (not actual items) may be processed.</p> <p>Syntax:</p> <pre>; INI: XXX flags Property START:pp~END:pp</pre> <p>Flags:</p> <ul style="list-style-type: none"> • L: Perform Lookback when reading data • M: Process multiple records. The property will automatically be changed to a list. • S: Save temporary data in zVAL between INIs. Various programming points can access zVAL using subscript indirection. <p>Property: The .NET property containing information for this INI. If the</p>

M flag was used, this will be a list.

START: The tag for the programming point fired prior to processing this INI block.

END: The tag for the programming point fired after processing this INI block.

Questions



Why is zVAL set to a string?

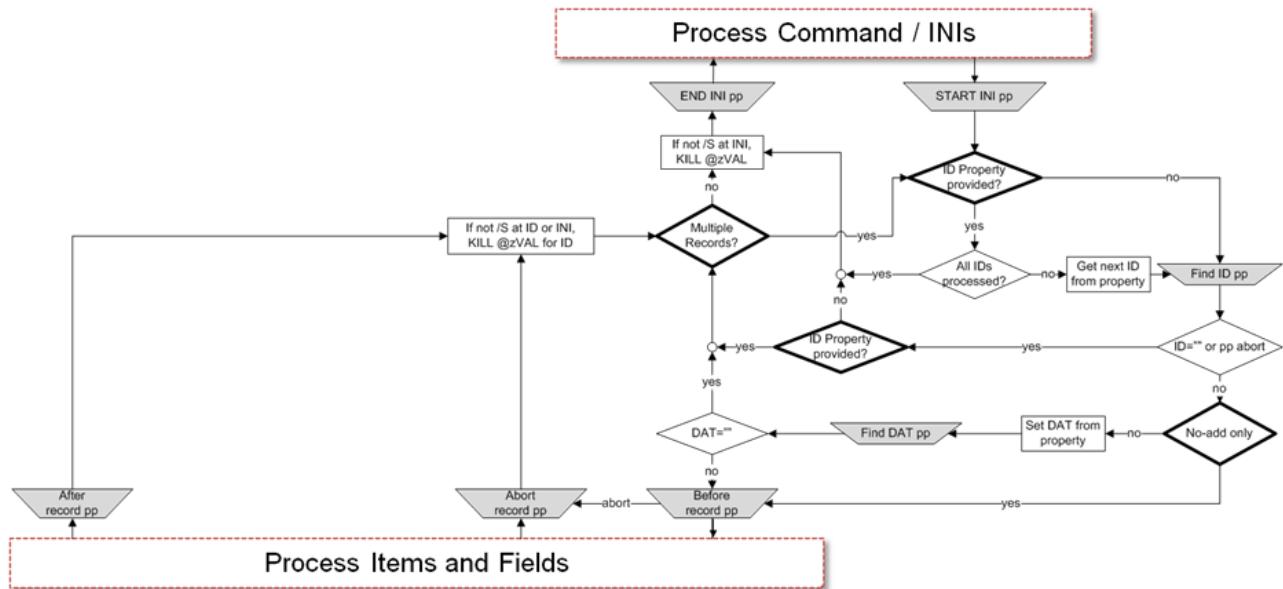


Why is "L" (instead of "S") used when initializing the Employees property?



Why is zVAL killed after SER, but not after EMP?

ID / DAT Level



Workflow Summary

1. Find the Record ID
2. Find the DAT
3. Load the items
4. Clean up and move on to the next record

Comments to indicate which EMP records to load:

```

; INI: EMP/MLS|Employees|START:startEmp~END:endEmp
; ID: /S|IdIn~IdOut|findID
; DAT: |DatIn~DatOut|findDAT
; Rec: BEFORE:beforeEMP~AFTER:afterEMP~ABORT:abortTEMP

```

Code generated by the previous comments:

```

S %=$$startEmp()
S recsID=$$zECFNew("Employees", "", "L")
S numIDs=$$zECFNumElmts("IdIn"), IDCnt=0
ld1ID ;
S IDCnt=IDCn+1
I IDCnt>numIDs G ld1DON
S ID=$$zECFGetElmt("IdIn", "", IDCnt)
I '$$findID(.ID) G ld1ID
I ID="" G ld1ID
S DAT=$$zECFGetElmt("DatIn", "", IDCnt)
I '$$findDAT(ID,.DAT) G ld1ID
I DAT="" G ld1ID

```

```
I '$$beforeEMP() S %=$$abortEMP() G ld1ID
;
; Load Items/Fields for EMP
;
S idID=$$zECFNewElmtObj (recsID)
S %=$$zECFSet ("IdOut", ID, idID)
S %=$$zECFSet ("DatOut", DAT, idID)
S %=$$afterEMP()
G ld1ID
ld1DON ;
S %=$$endEmp()
```

; ID:

Required.

Used to indicate which record(s) to process as part of this INI block.

Syntax:

```
; ID: /flags|RequestProp~ResponseProp|FindIDpp
```

Flags:

- **S:** Save temporary data in zVAL between records

RequestProp: The property containing the ID of the record to process. Will be a list of IDs if the M flag was applied to INI. If omitted, the programming point will be used to get the ID or iterate over the IDs.

ResponseProp: The property that will contain the ID of each record processed. If omitted, then the record ID is not sent back to the Server.

FindIDpp: The tag^routine used to find the next ID. The incoming ID parameter will be "" if an incoming ID property was not provided. Assign ID the value of the next ID to process. Return 1 to continue iterating, or "" when finished.

; DAT:

Optional.

Used to indicate which contact to process. Only required if over-time items are accessed.

Syntax:

```
; DAT:/flags|RequestProp~ResponseProp|FindDATpp
```

	<p>Flags: There are no DAT flags at this time, but they may be added later</p> <p>RequestProp: The property containing the DAT of the contact to process.</p> <p>ResponseProp: The property that will contain the DAT of the contact processed. If omitted, then the DAT is not sent back to the Server.</p> <p>FindDATpp: The tag^routine used to find the DAT to process. The incoming DAT parameter will be "" if the incoming DAT property was not provided. Assign DAT the value corresponding to the contact to process. Return 1 to continue processing the record and contact, or "" to abort.</p>
; Rec:	<p>Optional.</p> <p>Allows the programmer to specify tags that are executed before, during, and after a record is processed, or when the record processing is aborted.</p> <p>Syntax:</p> <pre>; Rec: BEFORE:pp~AFTER:pp~MID:pp~ABORT:pp</pre> <p>BEFORE: Specify a tag^routine to execute before each record is processed. Return 1 to continue processing that record, or "" to abort.</p> <p>MID: Specify a tag to execute after all items with the /S flag or with pre or format programming points assigned have been processed and loaded into @zVAL. Return 1 to continue processing the remaining items, or "" to abort the record.</p> <p>AFTER: Specify a tag^routine to execute after the record has been successfully processed.</p> <p>ABORT: Specify a tag^routine to execute when the record has been aborted.</p>

Activity: String Search

Describe how you would load all EMP records matching a search string using an ECF CDO. Start by answering the following questions:

1. How do you get the search string on the database?

2. How do you iterate?

3. What programming points would you use?

Sample Solution

The following is the M code that solves this problem:

```
; ;#ECFGENERATE#
; Type: Load
; Tag: lemp
; INI: EMP/M|Employees|start:startEmp
;       ID: |~Id|findEmpID
;       Items:
;             .2|Name
;       EndItems
; EndINI
; ;#ENDGEN#  
  
startEmp() ;Get the search string and initialize
s %ySearch=$$zECFGet("SearchString")
s %yName=$$znxNm("EMP",%ySearch,-1)
s %yName=$$znxNm("EMP",%yName)
s %yID=""
q 1  
  
findEmpID(ID) ;Return IDs until there are none
i $E(%yName,1,$L(%ySearch))'=%ySearch q 0
s %yID=$$znxNmID("EMP",%yName,%yID)
i %yID="" d ;No more IDs in current name
. s %yName=$$znxNm("EMP",%yName)
. i $E(%yName,1,$L(%ySearch))=%ySearch d
. . s %yID=$$znxNmID("EMP",%yName,%yID)
s ID=%yID
q ID'=""
```

Use %y* variables to maintain information between programming points. Note that you should not NEW them within the programming points, because doing so would prevent other programming points from seeing those values.



If you want to new your %y* variables prior to starting your CDO, you can define a separate entry point, new those variables, and then call the ECFCDO tag. For example:

```
EmployeeSearch ;  
    N %ySearch, %yName, %yID  
    D ltemp  
    Q
```

In your ECF command, use EmployeeSearch as the entry point rather than directly referencing ltemp.

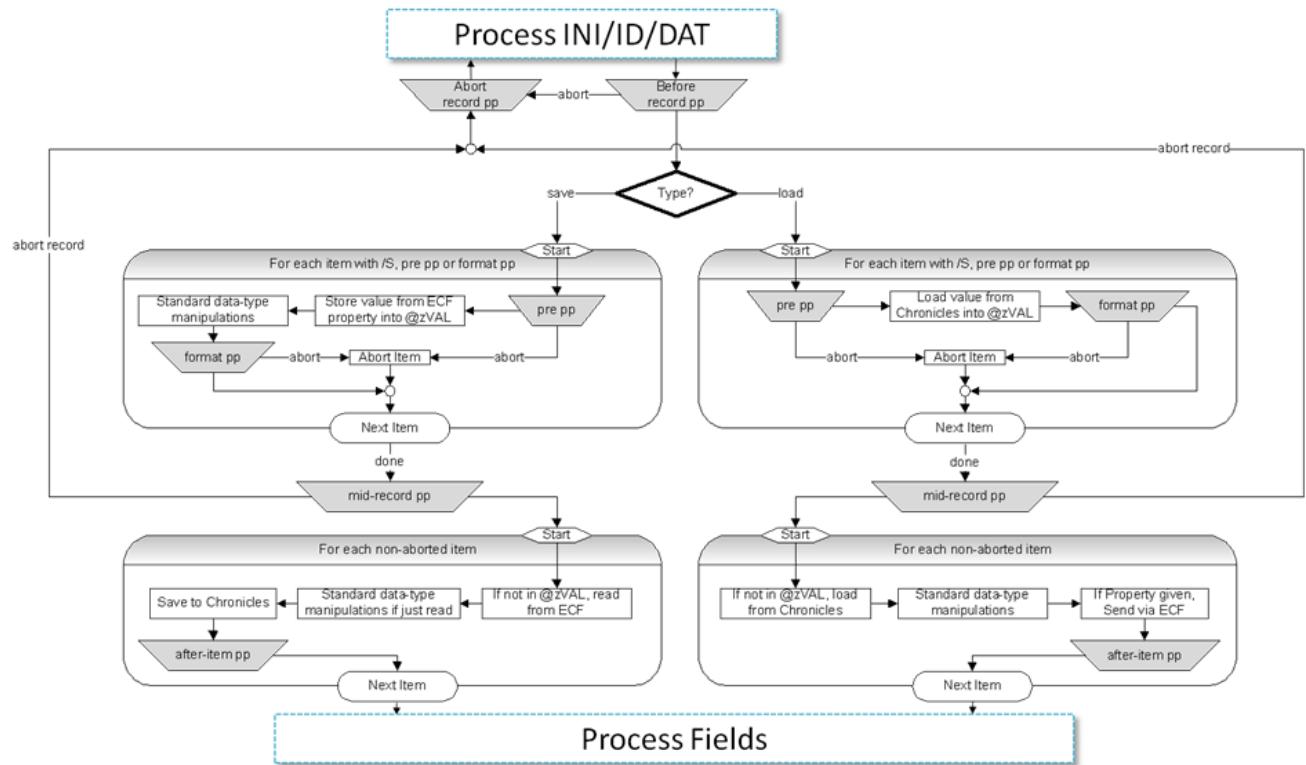


Are programming points called as functions or subroutines?



What would happen if you write them as if they were subroutines and not functions?

Item Level



Workflow Summary

1. Process items that
 - Require special code / formatting
 - Could cause an abort of the record
2. Chance to abort record based on step 1
3. Finish processing items
 - Apply standard formatting
 - Save to Chronicles (if Save) / Send via ECF to web server (if Load)

Example item-level comments:

```

; Items:
;   50/C|Status|PRE:before50~FRMT:format50~AFT:after50
;   RelGroup: LocaleInfo
;     3400|Locale
;     3402|LoadCdaAtStartup
;   EndRelGroup
;   PropGroup: Demographics
;     .2|Name
;   100/T|Address
;   14700/C|Sex
;   EndPropGroup
  
```

```
; EndItems
```

Basic Items

Comments to load I EMP 50, Status:

```
; 50/C|Status |PRE:before50~FRMT:format50~AFT:after50
```

Corresponding generated code:

```
S abortLst=", "
I '$$before50() S abortLst=abortLst_50_'," I 1
E D
. S @zVAL@(INI, ID, DAT, 50)=^ER1("EMP", ID, 50, 99999, 1)
. I '$$format50() S abortLst=abortLst_50_,"
I '$$midEMP() S %=$$abortEMP() G ld1ID
S idID=$$zECFNewElmtObj (recsID)
S %=$$zECFSet ("IdOut", ID, idID)
S %=$$zECFSet ("DatOut", DAT, idID)
I abortLst[, 50, " D
. S %=$$zECFSet ("Status", "", idID)
E D
. S %=$$zECFSet ("Status", "", idID)
. S val=@zVAL@(INI, ID, DAT, 50)
. I val'="" D
.. S catID=$$zECFNew ("Status", idID, "S")
.. S cmt=$P(val, "[", 2, 99999), val=$P(val, "[", 1)
.. S %=$$zECFSet ("Number", val, catID)
.. S %=$$zECFSet ("Comment", cmt, catID)
.. S cat=^ED("EMP", 50, "N", val)
.. S %=$$zECFSet ("Title", $P(cat, "^", 1), catID)
.. S %=$$zECFSet ("Abbr", $P(cat, "^", 2), catID)
. S %=$$after50()
```

; Items:

Optional.

Begin listing Chronicles Items

Syntax:

```
; Items:
;
```

```
Item#/flags|Property~commentProperty|PRE:pp~FRMT:pp~AFT:pp  
;EndItems
```

Flags:

- **R:** Item networks to a record. The corresponding C# data type will be Epic.Core.Database.Record
- **C:** Item is a category The corresponding C# data type will be Epic.Core.Database.Category
- **S:** Save the item value in @zVAL before final processing
- **P:** Do not run standard conversions (such as converting dates to/from internal Caché format).
- **T:** Item is a text property. In C# it is one long, delimited string. In Chronicles it is stored in multiple lines.

Property: The C# property name associated with this item

commentProperty: The C# property that holds the comment, the portion of stored data to the right of the "[" delimiter.

PRE: Programming point called right before processing this item. Return 1 when finished, or "" to abort the item.

FRMT: Programming point called to manually format the item. Return 1 to continue or "" to abort the item. The item value will be in @zVAL, using the following pattern if only one INI is used:

```
@zVAL@ ( ID, DAT, <ItemNum>) =value
```

If multiple INIs are used, the following pattern is used:

```
@zVAL@ (INI, ID, DAT, <ItemNum>) =value
```

INI, ID and DAT are assumed variables that are populated before the programming point is called.

AFT: Called after the item has been processed.



Why is item 50 loaded before the mid-record programming point?



Why is Status an object with sub properties rather than just one value?

Related Groups

Related groups correspond to the Epic.Code.Database.RelatedTable<T> on the web server, where T is a class with one property for each item in the related group.

Related-group comments:

```
;     RelGroup: LocaleInfo
;       3400|Locale
;       3402|LoadCdaAtStartup
;   EndRelGroup
```

Corresponding generated code:

```
S %=$$zECFSet ("LocaleInfo","",idID)
S relTblID=$$zECFNew ("LocaleInfo",idID,"S")
S lstID=$$zECFNew ("Rows",relTblID,"L")
S totalMin=1E25,totalMax=0
S total(3400)=^ER1("EMP",ID,3400,99999,0)
I totalMin>total(3400) S totalMin=total(3400)
I totalMax<total(3400) S totalMax=total(3400)
S total(3402)=^ER1("EMP",ID,3402,99999,0)
I totalMin>total(3402) S totalMin=total(3402)
I totalMax<total(3402) S totalMax=total(3402)
I totalMin=1E25 S totalMin=0
F lineCnt=1:1:totalMin D
. S lineID=$$zECFNewElmtObj(lstID)
. S %=$$zECFSet ("Locale",^ER1("EMP",ID,3400,99999,lineCnt),lineID)
```

```

. S %=$$zECFSet("LoadCdaAtStartup",^ER1("EMP",ID,3402,99999,lineCnt),line
ID)
F lineCnt=totalMin+1:1:totalMax D
. S lineID=$$zECFNewElmtObj(lstID)
. I total(3400)'<lineCnt D
.. S %=$$zECFSet("Locale",^ER1("EMP",ID,3400,99999,lineCnt),lineID)
. I total(3402)'<lineCnt D
.. S %=$$zECFSet("LoadCdaAtStartup",^ER1("EMP",ID,3402,99999,lineCnt),lin
eID)
K total,siCnt,siData,datLB

```

; RelGroup:

Optional.

Specify items that should be loaded/saved together as a related group. The related group may not contain nested related groups or property groups.

Syntax:

```

;Items:
; RelGroup:Property
;
Item#/flags|ItemProp~commentProperty|PRE:pp~FRMT:pp~AFT:pp
; EndRelGroup
;EndItems

```

Property: The C# property name associated with this related group. The data type will be Epic.Code.Database.RelatedTable<T>, where T is a class containing a property for each item in the related group.

Property Groups

Property groups are used when a class has a property that is itself an object with its own properties.

Property-group comments:

```

; PropGroup: Demographics
; .2|Name
; 100/T|Address
; 14700/C|Sex
; EndPropGroup

```

Corresponding generated code:

```

S %=$$zECFSet ("Demographics","",idID)
S grpID=$$zECFNew ("Demographics",idID,"S")
S %=$$zECFSet ("Name",^ER("EMP",ID),grpID)
S propID=$$zECFNew ("Address",grpID,"S")
S %=$$zECFSet (propID,"")
S crlf=$C(13)_$C(10)
S total=^ER1("EMP",ID,100,99999,0)
F lineCnt=1:1:total D
. I lineCnt=total S crlf=""
. S %=$$zECFSet (propID,^ER1("EMP",ID,100,99999,lineCnt)_crlf)
S %=$$zECFSet ("Sex","",grpID)
S val=$p($p(^EN("EMP",ID,14700),$c(1),1),$c(2),2)
I val'="" D
. S catID=$$zECFNew ("Sex",grpID,"S")
. S cmt=$P(val,",2,99999),val=$P(val,",1)
. S %=$$zECFSet ("Number",val,catID)
. S %=$$zECFSet ("Comment",cmt,catID)
. S cat=^ED("EMP",14700,"N",val)
. S %=$$zECFSet ("Title",$P(cat,"^",1),catID)
. S %=$$zECFSet ("Abbr",$P(cat,"^",2),catID)

```

; PropGroup:

Specify items that should be loaded/saved together as a sub property of the parent. Property groups may contain nested property groups or related groups.

Syntax:

```

;Items:
; PropGroup:Property
;
Item#/flags|ItemProp~commentProperty|PRE:pp~FRMT:pp~AFT:pp
; EndPropGroup
;EndItems

```

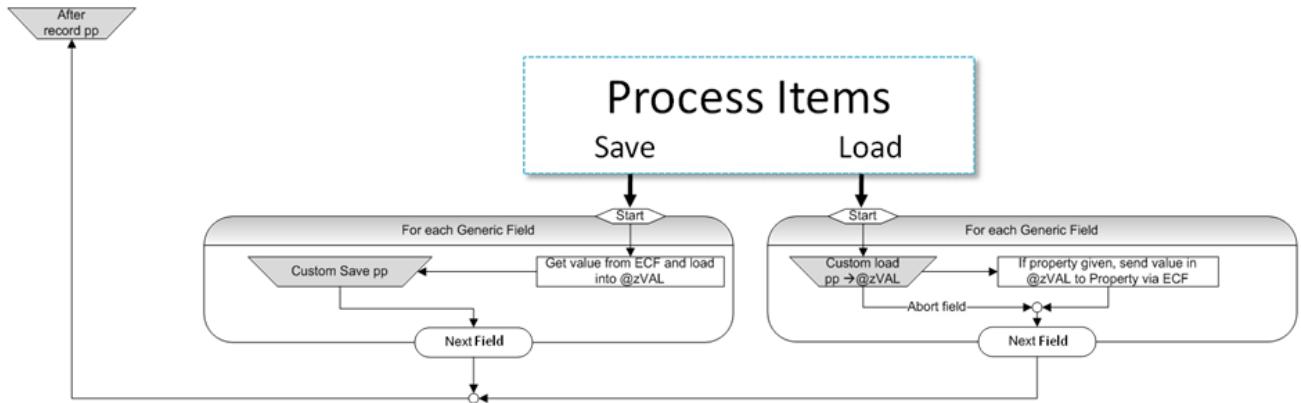
Property: The C# property name associated with this property group. The data type will be T, where T is a class containing a property for each ItemProp in the property group.



Why split properties into groups?

Fields

Use fields when the data you are working with is stored in globals outside of Chronicles.



Comments to generate code:

```

;Fields:
; 1/R|Record|GetRecordPp
; 2/CM|Categories|GetCategoryPp
; 3/T|Text|GetTextPp|80
;EndFields
  
```

Because data for fields could be stored anywhere, the programming points are actually required in order to load or store the data. Only the code used to call the programming points at the right time is created for you.

; Fields:

Optional.

Save/Load data to/from the database that is not specifically tied to Chronicles. For example, the data may be stored in other Caché globals.

Syntax:

```

; Fields:
; #/flags|Property|pp|otherInfo|associatedItem#
; EndFields
  
```

#: The number associated with this field. The value is accessed in zVAL using @zVAL@("F#")=value, or @zVAL@("F#",line)=value for multi-line values.

Flags:

- **R:** Value represents a record. The corresponding C# data type will be Epic.Core.Database.Record
- **C:** Value represents a category The corresponding C# data type will be Epic.Core.Database.Category
- **M:** Value has multiple lines
- **T:** Value is a text property. In this case, otherInfo is the maximum number of characters per line.

Property: The C# property associated with this value. If omitted, then the value is not automatically read from or sent to ECF.

Pp: The programming point used to process this field, either to save to or load from the database (not necessarily Chronicles).

associatedItem#: only necessary when a property name was given and there is a Chronicles item which can be associated with this information for the purposes of determining item protection

Example Routines

See the following routines for more detailed examples of the various options in ECFCDO. These examples will not be covered in class.

- Location: epic-fnd > epicmenu > FD

Template routines

These are generation test routines. No commands are defined for these.

Description	Name
Play around with generating load code	XBSBECFPB
Play around with generating save code	XBSBECFPS

Small data sets in Chronicles

Description	Name
Load one record from one INI	XBSBECFP1
Save one record from one INI	XBSBECFP2
Load one record from one INI into a parent object	XBSBECFP1P
Save one record from one INI into a parent object	XBSBECFP2P
Load one record from one INI and a second from a different INI	XBSBECFP3
Save one record from one INI and a second from a different INI	XBSBECFP4
Load multiple records from one INI	XBSBECFP5
Save multiple records from one INI	XBSBECFP6
Load multiple records from two INIs	XBSBECFP7
Save multiple records from two INIs	XBSBECFP8
Load and saves one record from one INI with groups within groups	XBSBECFPG

Large data sets in Chronicles

These use the full suite of 82 items.

Description	Name
Load record with record level pps	XBSBECFPP0L

Description	Name
Save record with record level pps	XBSBECFPP0S
Load record with pre item pps	XBSBECFPP1L
Save record with pre item pps	XBSBECFPP1S
Load record with format item pps	XBSBECFPP2L
Save record with format item pps	XBSBECFPP2S
Load record with after item pps	XBSBECFPP3L
Save record with after item pps	XBSBECFPP3S
Load record with pre & format item pps	XBSBECFPP4L
Save record with pre & format item pps	XBSBECFPP4S
Load record with pre & after item pps	XBSBECFPP5L
Save record with pre & after item pps	XBSBECFPP5S
Load record with format & after item pps	XBSBECFPP6L
Save record with format & after item pps	XBSBECFPP6S
Load record with pre & format & after item pps	XBSBECFPP7L
Save record with pre & format & after item pps	XBSBECFPP7S
Load record with /S	XBSBECFPP8L
Save record with /S	XBSBECFPP8S

Data from Globals

These use generic data from ^XBSBADATA.

Description	Name
Load & save generic fields for a single record, single INI	XBSBECFPF
Load & save generic fields for a multiple records, single INI	XBSBECFPF2
Load & save generic fields for a single record, multiple INIs	XBSBECFPF3
Load & save generic fields for a multiple records, multiple INIs	XBSBECFPF4
Load & save generic fields for a single record, single INI into a parent object	XBSBECFPF5

Exercise: Working with ECF CDO

In this exercise you will replace your code that loads cheeses from a certain country with a comparable ECF CDO routine.

Part 1: Read notes on programming points

As part of this exercise, you will need to write several programming points that fit into the ECF CDO. To ensure that you understand how they should be structured, read the following notes:

- For a programming point only specify the tag or tag^routine. The programming point is called as a function using "`$$_tag_()`". Parameters may not be specified with the tag name and none are sent with the exception of the Find ID and Find DAT programming points. All programming points should return 1 for success and 0 for failure and the record or item will be aborted if appropriate. An exception is the parent programming point, which returns the ID of the parent object, created using `$$zECFNew`.
- Even if the code for the programming point exists in the same routine as where the ECFGGENERATE comment block is defined, it may be necessary to attach the routine name to the tag when specifying it as a programming point. It is possible for the generated code to be too large to fit into the initial routine, causing the code to be split to another routine. Therefore the call to the programming point may no longer be in the same routine.



If your command initially works, but starts throwing <NOLINE> errors after you add additional code, it is likely because it is being split into separate routines. Fix this by adding the routine name to your programming points

- The following programming points do not abort anything:INI start,INI end, After Record, After item
- When multiple programming points are specified for an INI, record or item in the format: type:code~type:code, the order in which the programming points are listed does not matter
- The following variables are available to programming points:

Variable Name	Description
INI	The current INI
ID	The current record once determined
DAT	The current DAT once determined
zVAL	When temporary storage is used, <i>zVAL</i> contains the name of that storage location. Use @ <i>zVAL</i> @ to access the temporary storage.
zREC	When temporary storage is used, <i>zREC</i> contains the name of the contact level subscript for easy access to items. Use @ <i>zRec@</i> (item#) to access an item in temporary storage. This can be used in any programming point where the ID and DAT are already determined. <i>Available in 2012</i>
zITM	When temporary storage is used, <i>zITM</i> contains the name of the item level subscript for easy access to lines. Use @ <i>zITM@</i> (line#) to access an item in temporary storage. This can be used in item level programming points. <i>Available in 2012</i>
zPARENT	<i>zParent</i> contains the ECF ID of the object that represents the top level of the record. It can be passed as the parent argument to ECF functions. This can be used in After Record and Generic Field programming points. <i>Available in 2014</i>
abortLst	This variable is used to keep track of the items which have been aborted by their pre or format programming points.

Variable Name	Description
wasSent	Used when saving data to indicate if an item or generic field was sent to the server. Format: wasSent(item#)=1 if the property for the item was sent or wasSent("F#")=1 if the property for the generic field was sent

- If a programming point wishes for a variable to persist between programming points (other than those listed above), use a variable starting with %y.

Part 2: Use an ECF CDO

In this section you will add an ECF CDO comment block to load a list of cheese records that are sold in the specified country. The comment block will be provided to you, but you will be responsible for writing the programming point code.

1. Add the following comment block to your X<TLG> routine:

```
;#ECFGENERATE#
; Command: Train.<username>.GetCheeses
; Type: Load
; Tag: ld
; Storage: Local
;INI: XCH/M|Cheeses|START:start
;    ID: |~Id|nextId
;    Items:
;        .2|Name
;        1000/C|Country
;    EndItems
; EndINI
;#ENDGEN#
```

2. Add a tag called start to your routine. This will be the programming point executed before processing the XCH portion of the CDO.
3. Within start: get CountryValue using zECFGet and store it in a variable prefixed with %y.



Do not NEW %y variables. If you do, they will not persist between programming points.

4. Add a tag called nextId. This tag will be passed the variable ID by reference, so include it in your parameter list.

5. Write code in nextId that will find the next record ID from the regular item index. Keep the following in mind:
 - The incoming ID will always be "" because you are not bringing in an ID list from the server. Instead, maintain the previous ID examined using a %y variable.
 - You should return 1 if there is another ID, or 0 if there is not.
 - Remember that ID is an output parameter and must be assigned the ID of the next matching record.
 - Structure this method so that if CountryValue was passed as a null string, all records are loaded using \$\$zOID(INI, ID) rather than just those that match a particular country.
6. Edit your ECF command and change the entry point from GetCheeses^X<TLG> to Id^X<TLG>.

Part 3: Modify your C# Class

Notice that the ECFGENERATE block specifies a new item, 1000, which is a category (using the /C flag). This means that you need a new Country property in the Cheese class that is of type Epic.Core.Database.Category.

1. Add a reference to *Epic.Core.Database*. You will find the DLL in the following path:
 - CommandBuilder\Epic.Core.Database.dll
2. Add a new property named Country of type Category to *Cheese*.
3. In the ToString method of Cheese, be sure to call Country.ToString() and concatenate it to the result.
 - Remember that Country could be null.
4. Run your project and verify that it works. Fix any errors that you encounter.

Part 4: Load the remaining items

1. Modify your routine so that it loads the cheese items from the following table that you are not already loading. Then modify your project appropriately.

Item #	Item Name	Data	Add	Resp	Indx	Packed
1000	COUNTRY	C	N	S	Y	1000;1
1010	WEIGHT	N	N	S	N	1000;2
1020	PRICE	N	N	S	N	1000;3

Item #	Item Name	Data	Add	Resp	Idx	Packed
2000	DESCRIPTION	S	N	M	N	2000;1;1
3000	IMAGE PATH SMALL	S	N	S	N	3000;1
3010	IMAGE PATH LARGE	S	N	S	N	3000;2



Be sure to include the /T option on DESCRIPTION (XCH2000) since it is a multiple-response string item. This will delimit the lines with a carriage return, line feed.

Working with Chronicles

The following components are available for interacting with Chronicles from C#.

Access Manager

- Check License and Security
- Hyperspace Web [wiki > Database > AccessManager](#)

User Settings

- Replaces `EUserSettings` in VB
- [Hyperspace Web wiki > Database > User Settings](#)

Database Operations

- Create, duplicate and delete records, create contacts, etc.
- [Hyperspace Web Wiki > Database > Database Objects > Chronicles Operations](#)

Chronicles Database Definitions

- Replaces `EChronDBInfo` and `EChronItemInfo` in VB
- [HyperspaceWeb wiki > Database > Database Definitions](#)

Appendix A: Name Conventions Cheat Sheet

Naming Conventions Cheat Sheet**A•3**

Appendix A: Name Conventions Cheat Sheet

Naming Conventions Cheat Sheet

Identifiers

Type of Identifier	Style	Example
Parameter	camelCase	myParameter
Local Variable	camelCase	myString
Property	PascalCase	FileName, FirstName
Constant*	PascalCase	MaxValue
Private Instance Member Variable	camelCase with "_" prefix	_instanceVariable
Private Static Member Variable	camelCase with "s_" prefix	s_staticVariable
Public Member Variable	Use a property	

*- Sometimes values that are semantically constants, can't be declared with the const keyword. For example, private static readonly TimeSpan ShortTimeSpan = new TimeSpan(0, 0, 5). To these values, the naming convention for constants applies rather than the naming convention for static values.

Types

Type	Style	Type Name Example	Instance Example
Enum, Enum Values	PascalCase	ErrorLevel	errorLevel
Class	PascalCase	Patient	myPatient

Type	Style	Type Name Example	Instance Example
Method	PascalCase	ToString	
Struct	PascalCase	Point	myPoint
Interface	PascalCase, prefix with "I"	IDisposable	
Custom Exception	PascalCase, postfix with "Exception"	MessageException	ex
Event	PascalCase	DataUpdated	
Event Delegate	PascalCase, postfix with "EventHandler"	DataUpdatedEventHandler	
Collection	PascalCase, postfix with "Collection"	StateCollection	states
Attribute	PascalCase, postfix with "Attribute"	SyncAttribute	
Namespace	PascalCase	AppDomain, System.IO	

© 2016 Epic Systems Corporation. All rights reserved. PROPRIETARY INFORMATION - This item and its contents may not be accessed, used, modified, reproduced, released, performed, displayed, loaded, stored, distributed, shared, or disclosed except as expressly provided in the Epic agreement pursuant to which you are permitted (if you are permitted) to do so. This item contains trade secrets and commercial information that are privileged, confidential, and exempt from disclosure under the Freedom of Information Act and prohibited from disclosure under the Trade Secrets Act. This item and its contents are "Commercial Items," as that term is defined at 48 C.F.R. § 2.101. After Visit Summary, Analyst, ASAP, Beaker, BedTime, Break-the-Glass, Breeze, Cadence, Canto, Care Elsewhere, Care Everywhere, Charge Router, Chronicles, Clarity, Cogito ergo sum, Cohort, Colleague, Comfort, Community Connect, Country Connect, Cupid, Epic, EpicCare, EpicCare Link, Epicenter, Epic Earth, EpicLink, EpicOnHand, EpicWeb, Good Better Best, Grand Central, Haiku, Healthy People, Healthy Planet, Hyperspace, Identity, IntraConnect, Kaleidoscope, Light Mode, Lucy, MyChart, MyEpic, OpTime, OutReach, Patients Like Mine, Phoenix, Powered by Epic, Prelude, Radar, RedAlert, Region Connect, Resolute, Revenue Guardian, Rover, SmartForms, Sonnet, Stork, Tapestry, Trove, Trusted Partners, Welcome, Willow, Wisdom, With the Patient at Heart and World Connect are registered trademarks, trademarks or service marks of Epic Systems Corporation in the United States and/or in other countries. Other product or company names referenced herein may be trademarks of their respective owners. U.S. and international patents issued and pending.