

# Introduction to Object-Oriented Programming Using Visual C# Express Edition

by T. Grandon Gill

Copyright © 2008 by T. Grandon Gill

Permission to copy for educational use in non-profit institutions is granted, provided that notification of such use is made to the author at [ggill@coba.usf.edu](mailto:ggill@coba.usf.edu).



# Contents:

## Contents

Contents: .....	3
Module 1: Objects & Classes.....	7
What are Variables? .....	9
The Representation Challenge .....	9
Variable Names.....	11
C# Value Types.....	13
string Variables .....	15
What Are Operators? .....	16
Numeric Operators.....	18
string Operators.....	22
Boolean Operators .....	22
What are Functions? .....	25
Using Functions .....	26
Defining C# Functions .....	29
What Are Objects?.....	32
Spreadsheet Example .....	32
Classes and Namespaces.....	35
Example: Class Definition Files .....	35
Namespaces.....	37
UML Class Diagrams .....	40
Class Definitions .....	41
Property Members.....	44
Interface and Implementation .....	48
Creating Objects.....	49
Object Construction .....	50
Assigning Class Objects .....	52
Using Objects.....	54
Accessing Object Data and Functions .....	55
Static Members .....	56
Inheritance.....	59
Implementing Inheritance .....	60
The object Class.....	64
Console Tour.....	66
Creating a Console Application .....	67
The Main() Function .....	68
Introduction to Debugging.....	69
Windows Form Tour.....	72
A Windows Form.....	73
Adding Form Elements .....	76
Customizing Our Application.....	80

Events.....	82
Handling Events.....	83
Module 2: Control and Collections.....	89
if Constructs.....	91
"If" Family of Constructs.....	93
Nesting Constructs.....	97
Case Construct.....	99
switch...case Construct.....	101
Enumerations.....	105
enum Construct in C#.....	105
while Loops.....	109
The while Loop.....	109
for Loops.....	112
C# for Loop.....	113
Variable Scope.....	115
break And continue.....	116
Dialog Boxes and Forms.....	119
Dialog Boxes.....	121
Message Boxes.....	123
Arrays.....	125
Overview.....	125
The Array Collection.....	127
Generic Collections.....	131
The List<> Generic Collection.....	132
foreach Construct.....	137
foreach Loops.....	138
Examples.....	139
DataGridView Control, Part I.....	141
DataGridView.....	141
Form Inheritance.....	146
Implementing Form Inheritance.....	147
Preparing Base Class Form.....	149
Exception Handling.....	152
Exceptions in C#.....	153
Modules 3 & 4: Special Topics.....	157
Indexers (Module 3).....	159
Defining an Indexer.....	159
Resources and Bitmaps (Module 4).....	164
Introduction to Windows Drawing.....	164
Resources.....	168
Working with Bitmaps.....	171
Polymorphism.....	173
Examples of Polymorphism.....	175
Object Class.....	177
Abstract Classes and Interfaces.....	178
Timers (Module 4).....	183

Timer Control.....	183
Randomization and Hashing .....	188
Random Class .....	189
Hashing .....	191
Form Containers (Module 3) .....	195
TabControl .....	195
Splitter Window .....	196
ToolStripContainer .....	198
Menus.....	200
Menus.....	200
Tool Strips.....	202
DataGridView II .....	204
DataGridView Formatting .....	205
Setting Styles in C# Code .....	208
Serialization .....	210
C# Serialization.....	211
Customized Serialization .....	214
Printing.....	217
C# Form Printing .....	218
Print Preview.....	220
Database Programming (Module 3).....	222
Connecting to a Database.....	224
DataSet objects.....	230



# **Module 1: Objects & Classes**



# What are Variables?

## Learning Objectives

Upon completion of this reading, you should be able to explain:

- How computers represent information in memory
- Why variables must be defined if we are to write large programs
- The difference between giving a variable a name and a value
- The main primitive data types supported by C#
- The properties of different types of numbers
- The nature of Boolean values
- How to declare and initialize text data in the form of **string** objects

## Overview

When you write a program, you must accomplish two basic activities:

1. Representing the information that you are going to work with
2. Instructing the computer regarding how you want to process that information

In this segment, we focus on the representation activity. Specifically, we will examine how information can be represented in its most primitive forms. The forms we will present here are:

- Numbers (integers and real numbers)
- Boolean values (used to represent the logical values **true** and **false**)
- Text values, as stored in the **string** object.

As the course progresses, we will discover that most of the objects that we create are simply highly organized collections of these primitive forms. For now, however, we content ourselves with looking at the primitive data types alone.

## The Representation Challenge

Before we jump into why we need variables, it is useful to look at the problems that we face when trying to store information in a computer.

## The Twin Challenges of Representation

If you were to look inside a computer, you would discover a collection of components--processors, RAM chips, disk drives, etc.--all of which have one thing in common: they represent information as patterns of 0s and 1s, known as **bits**. How they represent these bits depends on the physical device; it might be voltages in a switched circuit, changes in magnetic field on a hard disk, holes in a punch card, and so on. Nonetheless, **digital** devices are nearly always constrained

to representing data using bits. That is what makes them digital. Thus, we face twin challenges in trying to represent information:

- How do we translate our information--whatever it may be--into patterns of 0s and 1s (and then translate the 0s and 1s back into information we can use)
- How do we locate the information we have placed in the computer

## Defining Data Types

In order to deal with the problem of representing information, we need to develop translation schemes for moving between bits and information. Some of the most important of these translation schemes involve **primitive data types**, which is to say, the lowest level of information we typically worry about representing. Examples of primitive data types include:

- *Integers*, or whole numbers. In C#, for example, integers can be represented using clusters of 16 bits (**Int16**), 32 bits (**Int32** or **int**), or 64 bits (**Int64**).
- *Real numbers*. In C#, real numbers can be represented using clusters of 32 bits (**float**) or 64 bits (**double**).
- *Booleans*, or true/false values. In C#, these can be represented with a single bit (**bool**).
- *Text*, also referred to as character strings. In C#, we normally represent this as a sequence of 16 bit clusters (**string**), which are interpreted using the **Unicode** character representation scheme. Sequences of 8 bit clusters, translated using the **ASCII** character representation scheme, may also be used.

We call these data types primitive because--with the some-time exception of **string** objects--we never really need to worry about how these clusters of data actually translate the values we're holding into patterns of 0s and 1s. We simply take it on faith that they are doing so correctly. In database parlance, such values are also referred to as being **atomic**.

As we move towards representing information in more complex forms, such as user-defined objects, we will generally discover that such objects are really just aggregations of primitive data types. For example, a date object might be viewed as consisting of a month, day and year value, all of which are integers. Indeed, in C# (although not in all languages), these primitive types are actually objects themselves. Later in the course, we'll discover what that means.

## Finding Our Information

Given that we (or, at least, our programming tools) know how to translate our information back and forth into bit patterns, the next problem is finding them. For example, a typical PC today might have a Gigabyte of RAM (roughly a billion 8-bit characters, or 8 billion bits). That means our odds of finding a single 32 bit integer by randomly inspecting RAM are considerably worse than our odds of winning the state lottery.

To facilitate the process of organizing digital information, every storage location (e.g., in RAM, in the processor, on a CD, on a hard drive) is assigned an address. Thus, if we know the address of a piece of data, we can retrieve it from digital storage. Indeed, no matter what scheme we use

to organize our data (e.g., folders on a hard drive), by the time it reaches the hardware that information is supplied as an address.

In the very early days of computers, addresses were used exclusively for identifying where information was located. Thus, when a programmer wrote a program, he or she had to keep track of the address of every piece of information that the computer was working with. To put this in context, however, that was in the days when computers might have 2,000 characters of **primary storage** (what we call RAM today). There were a lot fewer things to keep track of.

By the mid-1950s, the hardware storage capacity had increased to the point that keeping track of every piece of information stored by the computer using its physical address placed too great a burden on the programmer. That meant a new approach to assigning and locating information needed to be developed.

## Variable Names

The solution to the problem of keeping track of information in primary storage that replaced direct addressing in the mid- to late-1950s is still in use today. It involves two key elements:

- Allowing memory locations where data is stored to be referred to by a user-defined name, instead of by address.
- Allowing programming tools--such as assemblers or compilers--to determine the actual placement of such information, instead of forcing the programmer to do that task.

As a consequence, when the programmer wants to store information, he or she must normally perform two tasks:

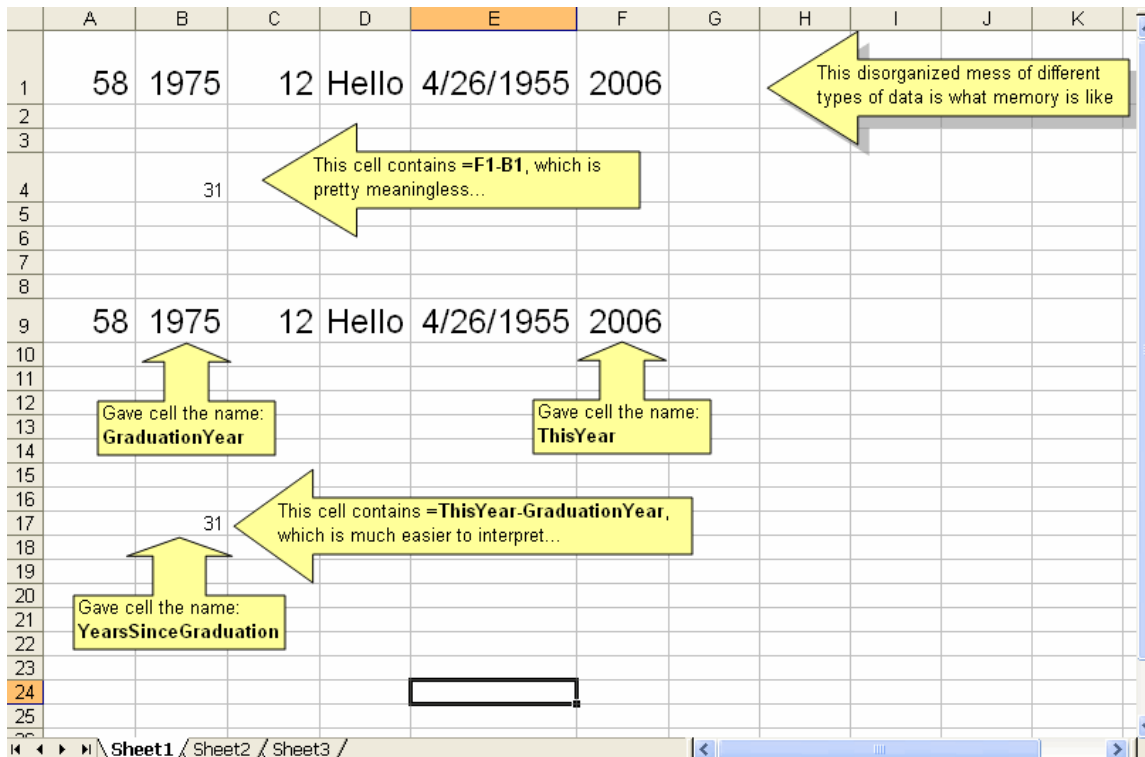
1. Come up with a name and associated value type (real, integer, string, object, etc.) to be used in referring to that information
2. Tell the computer to set up a location where that information will be stored.

For primitive value types, such as presented in this section, these two activities normally occur at the same time.

## Benefits of Naming

The principal benefits of naming are in reducing the mental strains placed on the programmer. This is important because if the programmer is worried about tedious details, he or she cannot focus on the complex tasks that need to be accomplished.

Throughout this module, we will use a familiar tool--the spreadsheet--to illustrate a number of programming concepts. Our first example, presented below, shows how the use of names can clarify what a programmer's intent is.



The example can be interpreted as follows:

- The top line of the spreadsheet represents what "raw" RAM--already organized into primitive data types--might look like. By itself, it makes no sense to the observer whatsoever.
- In cell B4 (referred to by its "address") the value of the formula F1-B1 appears. Once again, this tells us little about what it's supposed to represent
- In line 9, we have a copy of the same data--except we've assigned two range names (Excel's equivalent to variable names): B9 is named *GraduationYear* and F9 is named *ThisYear*
- The formula in B17 is specified using names, instead of cell addresses, as: *ThisYear-GraduationYear*. The cell is then named *YearsSinceGraduation*.

Think how much easier it would be to figure out what the spreadsheet was doing (and if it were properly designed) if you were given the named version.

## Variables in C#

When programming in C#, you can choose almost anything for a variable name, provided:

- It begins with a character or `_` (underscore)
- It consists of letters, numbers and `_` characters

- It does not conflict with existing **reserved words** (e.g., names like **if**, **class** or **break** that already have meaning in C#). This is usually not a problem today (although it used to be, in less advanced days!) since the C# editor colors reserved words--making them easy to identify if you use one by mistake.

In addition, names are **case-sensitive** in C#, which means a variable named `a1` is totally different from one called `A1`.

Variable names are not the only way of referring to information in a program--which is good, since it would take years to come up with enough individual names for a program with a million pieces of information (which isn't that many)! C#, for example, offers a number of **collection** types, such as **arrays** and **lists**, that can hold and provide access to information without requiring individual names for each information element. Thus, we might refer to `MyInfo[1004]` to access the 1005th element of a collection of integers stored in `MyInfo` (since C# numbering schemes always begin at 0).

Collections will be covered in greater detail when we get to Module 2.

## C# Value Types

As mentioned earlier, creating a name for a variable does not necessarily set aside the memory needed to hold the information it represents. For C# objects known as **value types**, however, declaring a name for the variable *does* set aside the storage for it. These value types also include most of our primitive data types. Specifically:

- Integer values (e.g., **int**, **int16**, **int32**, **int 64**), where the XX in `intXX` specifies how many bits of storage are used.
- Real number values (e.g., **float**, **double**). The `double` is most commonly used, and takes 64 bits of storage
- Boolean values (e.g., **bool**), which can hold two values--**true** and **false**.

In addition, text data can be held in the **string** data type. Although it is not actually a true value type, it is set up in such a way that it behaves like a value type in most circumstances, and there is little harm in viewing it as such.

## Declaring Value Types

Before you can use a value type variable--indeed, any variable--you must first declare its name and type. This involves a statement of the form:

```
value-type name; // declaring a variable without initializing it
```

OR

*value-type* name=*value*; // declaring a variable and initializing it

Some actual examples are presented below:

```
int x = 3; // int is the most common integer type
int y = 4;
int z;
uint PosOnly = 47; // unsigned integer values can only be positive
double dx = 3; // double is the most common real type type
double dy = 4;
double dz;
bool b1 = false; // bool can have only two values, true or false
bool b2 = true;
```

Some comments about these declarations:

- Built-in value types are reserved words, and appear in blue
- Anything following // is treated as a comment, and appears in green
- **true** and **false** are reserved words.
- Uninitialized variables are set to 0, 0.0 or false, as appropriate.

For the time being, don't worry about where such declarations are used. Later in this module, we'll discover that they can appear in two places in a C# program--within function definitions and within class definitions.

## Numeric Value Ranges

As you may have noticed, there are a number of different choices available for a given category of data, such as integers. It stands to reason that the more bits available for a variable, the greater the range of values it can hold. In the "bad old days" of programming, we cared about such things a great deal. RAM was in short supply, so we always tried to use the smallest variable type that we could. Today, such issues matter much less, particularly if you are programming for business.

That being said, you should probably be aware of the following limits, however:

- **int**, which is equivalent to **int32**, provides a range of roughly -2 billion to +2 billion. If you think you might exceed these limits, **int64** takes you up into the quintillions.
- **uint**, an **unsigned integer** type that can only hold positive numbers, provides a range of 0 to about +4 billion
- **double** stores numbers in an exponential format and has a range that is too large to worry about. It gives about 15 digits of precision, but is subject to rounding error if used for currency calculations.

- **float** stores numbers in an exponential format and also has a range that is too large to worry about. It gives about 6-7 digits of precision, making it a very bad choice for currency calculations.

Another reason why we aren't spending too much time on these ranges is that .NET will throw an exception that will normally crash your program should you exceed a variable's allowable range. While this may seem like a bad thing, it's actually much better than having the program keep running after a computation has bombed.

## string Variables

C# supports a **string** object type that was designed to behave like a value type, even though it is truly not a value type. The reason it seems like a value type is:

- It can be initialized like a value type, using text quoted in double quotes ("). Double quotes is important to note here, since single quotes can only be used to refer to a single character.
- Once its value has been set, the only way to change it is to reassign its value (i.e., you can't change characters within a string once it has been created).

Examples of actual string declarations are presented below:

```
string s1 = "Hello"; // Strings can be initialized with literals
string s2 = @"World,
it's me"; // @ allows multiline declarations and special characters, like \
string s3;
```

Some comments on these:

- Like a value type, you can initialize to a literal value (the quoted strings, shown in red).
- If you want to put things like line breaks or \ characters in a quoted string, preface it with the @ character. Indeed, it is probably safest to use this character.
- Uninitialized string objects are set to **null**. This keyword is not used for value types.

There are many other interesting things we'll discover about **string** objects as the course progresses. But we now know enough to move on to our next topic.

# What Are Operators?

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what an operator is
- Identify different categories of operators
- Explain the difference between unary and binary operators
- Write simple expressions using numeric operators
- Write simple expressions using string operators
- Write simple expressions using logical and conditional operators
- Identify certain situations where care must be taken in applying operators

## Overview

The operators provided by C# closely resemble the operators supported by other languages and are also similar to symbols that we may have encountered in areas such as mathematics and philosophy. We will focus on 6 different types of operators:

1. *Arithmetic*, used to tell the program to perform computations
2. *Comparison*, used to test the equivalency (==) or order (<, >) of values
3. *Assignment*, used to tell the program to move a value from one place to another (usually a variable or collection element)
4. *Logical*, used to combine Boolean expressions using logical terms (e.g., and, or)
5. *Concatenation*, used to combine string variables into single long string
6. *Conditional*, used to choose between two values

There are additional operator types--e.g., grouping operators such as ( and ) and built-in operators, such as **new**--but these either tend to be easier to use than explain (e.g., parentheses) or will be introduced later.

## What is an Operator?

An operator is usually just a shorthand way of telling the computer to perform some task. What distinguishes operators from other ways of telling the computer to do things, such as functions (covered in the next reading) is mainly a matter of how they are applied and where they are placed in your program. Specifically, most operators fall into one of two categories:

- *Unary operators:* Operators that apply to a single expression. When placed in front of a variable or expression, such as the minus sign in  $-X$ , unary operators are called **prefix operators**. When placed behind a variable or expression, as the plus-plus in  $i++$ , unary operators are called **postfix operators**.
- *Binary operators:* Operators that are placed between two variables or expressions, such as the  $+$  sign in  $X+Y$ .

We normally use operators for convenience, since anywhere we use an operator we could use a function (e.g., the LISP programming language, used in artificial intelligence, has dispensed with operators altogether). They also can be quite intuitive. For example, the precedence rules learned in algebra--in the expression  $x+y*3$  we do the multiplication before the addition, owing to the higher precedence of multiplication--generally apply to the arithmetic expressions we write when programming.

### Examples of Operators

Returning to our use of spreadsheets to illustrate programming concepts, we demonstrate how operators can be used in cell formulas.

	A	B	C	D	E	F
1	<b>Names</b>	<b>Values</b>				
2	X	12				
3	Y	14				
4	Z	16				
5	h	Hello				
6	blank					
7	end	World				
8						
9			12	← Formula: =X		
10			42	← Formula: =X+Y+Z		
11			FALSE	← Formula: =Y>Z		
12			TRUE	← Formula: =Y<Z		
13			Hello World	← Formula: =h&blank&end		
14						

In interpreting the example:

- The first arrow shows the use of the assignment operator to place the value in cell B2 (to which we gave the variable name X) into cell C9.
- The second arrow shows the chaining of  $+$  operators together, also supported in C#.

- The third and fourth arrows demonstrate how comparison operators can be used to produce Boolean expressions. In C#, these would evaluate to **false** and **true** (case sensitive).
- The final arrow demonstrates concatenation of strings. Note that in Excel, this is done with the & operator, while in C# we'd be using the + operator.

If these examples make sense to you, then all we need to do is mention a few details on the specific use of these operators.

## Numeric Operators

The most common use of operators in C# is for numeric expressions. Generally speaking, four categories of operators are most commonly applied to numeric values:

- *Arithmetic operators*, including +, -, / and \*. These are binary operators and use the same precedence rules you learned in algebra, with \* and / being done before + and -. There are also some less familiar operators:
  - ++ and --: These are unary operators that can be applied either prefix or postfix and serve to increment (++) or decrement (--) the variable they are applied to. For example, the expression X++ is equivalent to X = X+1.
  - %: The modulus operator returns the remainder of an integer division.
- *Comparison operators*, including <, >, <=, == and != (not equal): Return the Boolean result for the specified comparison.
- *Assignment*, including = and +=: Takes the value of the RHS (right hand side) and puts it in the variable on the LHS (left hand side). X+=Y is equivalent to X=X+Y.
- *Negation*, the - sign. A prefix operator that takes the expression on its right and multiplies it by -1. For example, -X is equivalent to -1\*X.

## C# Example

Because the numeric operators are so intuitive, explaining them further is less productive than giving an example.

```

int x = 3; // int is the most common integer type
int y = 4;
int z;
int r;
uint PosOnly = 47; // unsigned integer values can only be positive
double dx = 3; // double is the most common real type type
double dy = 4;
double dz;
bool b1 = false; // bool can have only two values, true or false
bool b2 = true;

z = x + y;
dz = dx + dy;
b1 = (x < y);
b2 = (x > y);
r = y % x; // Gives the remainder of y/x, which is 1
x += y; // same as writing x = x + y;

```

## Operator Issues

You need to be aware of certain issues with operators when you start to program. We'll now examine some of these.

### *Assignment Issues*

It is important to note that the assignment operator--a single equal sign--serves to move the value from the RHS into the variable on the LHS. In this respect, it is different from algebra, although the difference is subtle.

In algebra, the expression  $X = Y+3$  is an assertion that the value of  $X$  is equivalent to  $Y+3$ .

In C#,  $X=Y+3$  is a command saying "Add 3 to the value of  $Y$  then place the result in storage location  $X$ ".

The difference becomes obvious when we reverse the expressions, since:

In algebra, the expression  $X = Y+3$  is completely equivalent to  $Y+3 = X$ .

In C#,  $Y+3 = X$  is nonsense, and would not be allowed, since  $Y+3$  is not a valid location in which to store a value.

Thus, in C#, you'll always find variable names (or, as we will see later, array element expressions such as  $Y[3]$ ) on the left hand side of an assignment operation.

Another common problem--one that will produce an error message when you try to prepare your code for running (**compile** it)--is mixing the assignment ( $=$ ) and equality test ( $==$ ) operators. Specifically:

- The assignment (=) attempts to move the RHS value into the LHS variable, changing it in the process.
- The equality test (==) attempts to see if the RHS and LHS values are equal, returning **true** if they are, **false** if they are not--just the way < and > do in the example code.

Since C# will give you an error if you mix these two operators up, you can usually catch such problems early.

### *Integer Division*

When dividing integers, the remainder of the division is thrown away--it is not rounded. Thus:

$17/5 ==> 3$

$19/5 ==> 3$

$4/5 ==> 0$

This can be important because whenever you are dealing with integers, you could run into problems if you are not aware of ordering. For example:

$28*(6/7) ==> 0$  because  $6/7$  is 0.

On the other hand:

$28.0/(6.0/7.0) ==> 24.0$ , since real number divisions hold fractions.

The modulus (%) operator can be used to test the remainder of an integer division operation.

### *Mixing Integers and Real Numbers*

When integers and real numbers are mixed in an arithmetic expression, C# will normally convert all the values to real numbers before performing the computation, since real numbers can hold a much larger range of values than integers.

If necessary, use parentheses to force proper execution. For example:

$0.8*(6/7)$  will give 0, since  $6/7$  is done as integer division.

$(0.8*6)/7$  will give a non-zero value (0.7) since the first expression produces a double, which then becomes a part of the second calculation.

## Typecasting

From time to time, you may need to move numeric information across different types (e.g., from integers to real numbers or vice versa). In general, C# will let you do this automatically if the result is a promotion, e.g.:

```
int X=3;
```

```
double Y;
```

```
Y=X; // This should be allowed, since there's no danger of losing data or exceeding range
```

On the other hand, C# is very picky about assignments where the LHS range could be exceeded. Thus, the following code would produce an error:

```
double X=3.0;
```

```
int Y;
```

```
Y=X; // This won't be allowed, since there is a danger of exceeding Y's range
```

Sometimes, however, we know that our assignment would be okay (as in the case of the 2nd example above, since 3 is certainly within the range of Y). In such circumstances, C# allows us to indicate that we know what we're doing by placing the target type in parentheses in front of the expression we're moving. For example:

```
double X=3.0;
```

```
int Y;
```

```
Y=(int)X; // The (int) in front of X tells the compiler to ignore the error
```

An example of a typecast follows:

```
int x = 3; // int is the most common integer type
int y = 4;
int z;
int r;
uint PosOnly = 47; // unsigned integer values can only be positive
double dx = 3; // double is the most common real type type
double dy = 4;

PosOnly = dx / x; // Generates an error about converting double to uint */
PosOnly = (uint) (dx / x); // Typecast removes error
```

Two comments about typecasting:

- It can only be used between compatible types. For example, it can be used to move between number types but you can't typecast a number to a string.
- It should be used cautiously, especially if you are trying to get rid of a pesky error without really understanding what it's doing.

Typecasting can also be used in a more general context--for example, with objects that inherit from each other--as we shall see in later modules.

## string Operators

A number of operators can be applied to string objects. These include the following:

- *Concatenation*, done with the + operators. For example, if X and Y are strings and X contains "Hello" and Y contains "World" then X+Y would evaluate to "HelloWorld".
- *Comparison*, done with the == and != operators. Somewhat surprisingly, the < and > comparison operators are not supported for **string** objects. The reasoning behind this is probably that C# strings are case sensitive, meaning that "aardvark" would be "Zoo", since the character 'a' (lower case) comes after 'Z' (upper case) in ASCII and Unicode.
- *Assignment*, done with the = operator.

Some examples of string operators follow:

```
string s1 = "Hello"; // Strings can be initialized with literals
string s2 = @"World,
it's me"; // @ allows multiline declarations and special characters, like \
string s3;
string a1 = "aardvark";
string a2 = "Aardvark";

s3 = s1+" "+s2;
bool b3 = (a1 == a2); // This is false! 'a' doesn't equal 'A' in ASCII
```

## Boolean Operators

A number of operators apply specifically to Boolean (true/false) expressions. These can be applied either to variables of type **bool** or, more commonly, to Boolean expressions, such as (X > 3). Common categories include:

- *Logical operators*, && and ||. Specifically, b1 && b2 (read b1 AND b2) returns true only if both b1 and b2 are true. b2 || b2 (read b1 OR b2) returns true if either b1 or b2 or both is true, false otherwise.

- *Comparison operators*, == and !=. In the case of Boolean comparisons, < and > don't make any sense, so this makes sense.
- *Conditional operator*, ? :. This is the only ternary (three part) operator, and works like the Excel =if() function, using the form **(test) ? result-if-true : result-if-false**.
- *Negation operator*, ~. Unary prefix operator that is the logical equivalent of a - sign. Thus, ~ true is false and ~ false is true.

Some examples of Boolean operators in action follow:

```
double dx = 3; // double is the most common real type type
double dy = 4;
bool b1 = false; // bool can have only two values, true or false
bool b2 = true;
bool b3 = !b2; // b3 is false
bool b4 = (b3 == b2); // b4 is false
bool b5 = (b2 || b3); // b5 is true, since b2 is true
bool b6 = (x > y) && (x * y > 0); // b6 is false, since y>x
double dMax = (dx > dy) ? dx : dy; // Conditional operator
```

## Conditional Operator

The conditional operator is probably the least familiar of the Boolean operators, but it is relatively simple to understand if you've ever used the Excel =if() function, illustrated below:

	A	B	C	D	E	F	G	H
1								
2	X	17	Cells have been named using left column					
3	Y	12						
4								
5								
6			X > Y			Formula: =if(X>Y,"X > Y","X <= Y")		
7			Y <= X			Formula: =if(Y>X,"Y > X","Y <= X")		
8								
9			<b>Absolute value example:</b>					
10								
11			5			Formula: =if(X-Y>0,X-Y,Y-X)		
12			5			Formula: =if(Y-X>0,Y-X,X-Y)		
13								
14								

The difference between the C# and Excel versions are that the Excel version is a function, while the C# version is an operator of the following form:

`test ? expression-if-test-is-true : expression-if-test-is-false`

Thus, the top formula might look something like the following in C#:

`int X=17;`

`int Y=12;`

`string result=(X>Y) ? "X > Y" : "X <= Y";`

Similarly, the first absolute value formula might look like:

`int X=17;`

`int Y=12;`

`int result=(X-Y>0) ? X -Y: Y-X;`

Note that the value returned by the conditional operator depends on the type of the two expressions separated by the `:` character (which must match). It does not (necessarily) return a Boolean value.

# What are Functions?

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain the role played by functions in programming
- List the benefits of using functions
- Identify the three elements of a function header line--the name, return type and argument list
- Figure out how a function should be called if presented with its declaration
- Describe the basic structure of a function definition
- Define simple functions

## Overview

Functions are the fundamental building blocks of computer programs, whether they be object-oriented programs or older structured programs (written in languages like Pascal or C). The use of functions provides a number of benefits to the programmer:

- *Functions allow blocks of code that are needed in different parts of a program to be stored in a single location.* This reduces the size of a program and facilitates maintenance, since errors or updates only have to be made in one place.
- *Functions provide a means of breaking programs up into conceptually manageable blocks.* Prior to the widespread adoption of object-oriented programming in the early 1990s, breaking an application into functions was the only way to manage its complexity. Today, the designers use objects and functions together to make code easier to understand and distribute across multiple programmers.
- *Functions provide a means of sharing code between programmers.* Because functions can be used in a program without fully understanding the mechanics of how they work, they provide a convenient means of encouraging code reuse. Purchasing libraries of functions, or obtaining their open-source equivalent, can save huge amounts of programming and debugging time. Once again, functions and objects work together nicely in this context since--as we'll discover--functions are integral elements of most object definitions.

In working with functions, the programmer needs to be aware of two things:

- How to call a function that has already been defined
- How to define a function

In this reading, we provide a broad, mainly qualitative, introduction to both areas.

## Using Functions

We now take a conceptual look at functions, focusing on how functions are declared and how they are used.

### Declaring a Function

Much like variables, functions need to be declared as part of the process of defining them. The process of declaring functions is slightly more complicated than declaring variables, however, since a function typically has three elements that must be specified:

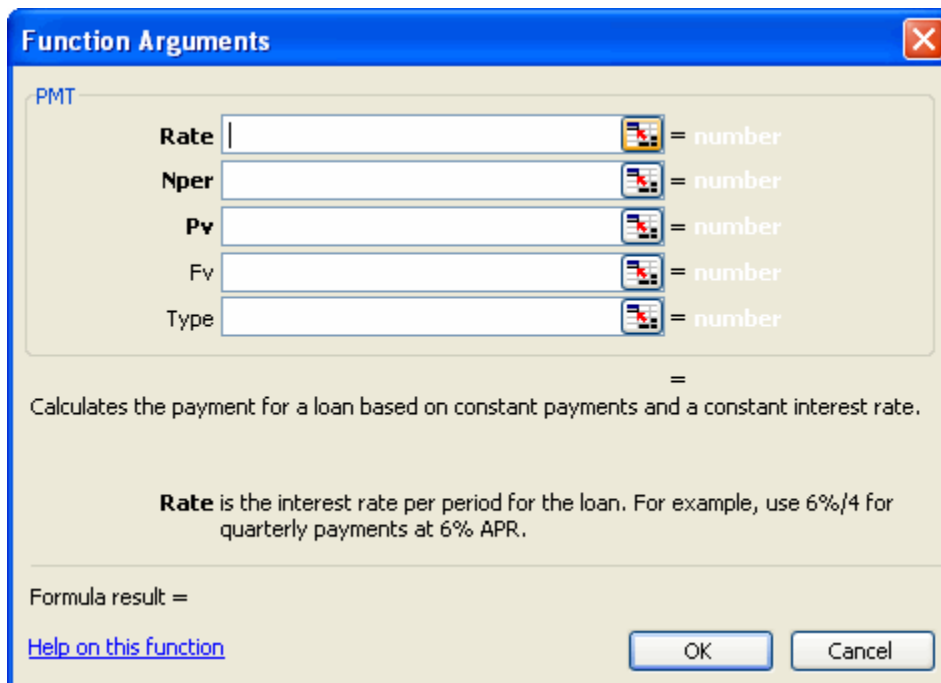
- *Name*: a means of identifying the function. In C#, for example, the rules for naming functions and variables are the same.
- *Arguments*: A mechanism for passing values into the function. Some functions have no arguments. Some functions also change the value of their arguments, as an alternative to returning a value. In C#, function arguments are specified just as if they were variable declarations, except the arguments are separated by commas.
- *Return Type*: The type of data (which may be an object) that is returned once the function has been called. In C#, functions may be defined that don't return a value, in which case their return type is specified to be **void**.

### Spreadsheet Example

Because programming is unfamiliar to many students, it is easy to get confused with respect to their definition and use. Indeed, inability to define and call functions is one of the most common problems introductory programming students encounter. This is unfortunate, since you've probably used functions before--as in spreadsheets.

#### *Function Declaration*

In the example below, we are looking at the box that comes up when you press the **Fx** button next to the formula in Excel for the PMT() function, which calculates a loan payment.



What you see are the same elements you'd see for a C# function. Specifically:

- The function name is PMT
- The function has three required arguments: Rate, Nper, Pv (interest rate, the number of periods of the loan, the present value--a.k.a. amount--of the loan), all of which are numbers. In Excel, there are also some optional arguments. In C#, you could achieve the same thing by defining three versions of the function, one with 3 arguments, one with 4 arguments and one with 5 arguments.
- The return type is number (although not explicitly specified in the dialog).

In C#, the first line of the PMT function definition--the **declaration** or **header** line--might look something like the following:

```
double PMT(double Rate, int Nper, double Pv)
```

Unlike Excel--which is vague about what type of numbers it used--we see that Rate and Pv would be defined as **double** (real numbers), while it would make more sense to specify Nper (number of periods) as an **int** (integer).

### *Calling a Function*

The declaration line, the first part of any function definition, tells us nothing about how the function works. It does, however, tell us all we need to know regarding how it should be called. In the Excel example below, alternative ways of calling the PMT function are presented:

	A	B	C	D	E	F	G	H	I
1									
2	Rate	6.00%	Cells have been named using left column						
3	Periods	30							
4	Amount	\$100,000.00							
5									
6	Annual Payment:			(\$7,264.89)		=PMT(Rate,Periods,Amount)			
7	Annual Payment:			(\$7,264.89)		=PMT(\$B\$2,\$B\$3,\$B\$4)			
8	Annual Payment:			(\$7,264.89)		=PMT(0.06,30,100000)			
9	Monthly Payment:			(\$599.55)		=PMT(Rate/12,Periods*12,Amount)			
10									
11									

Looking at the 4 calls, we see a variety of different ways of making the call can be used:

1. Variable names (range names, in Excel) can be used. To avoid an error, the type of data in the cell must match the argument type--true in all cases.
2. Cell addresses can be used.
3. Literal values can be used (e.g., 0.06, 30, 10000).
4. Expressions can be used (e.g., Rate/12, Periods\*12) as long as the expressions evaluate to the same type of data as the argument requires.

With the exception of the 2nd example (direct use of memory addressed is not normally available in C#--unlike its cousin C++, which uses memory addresses all the time), all of these calls could be made in C#. For example:

```
int Periods=30;
```

```
double Amount=100000.0;
```

```
double Rate=0.06;
```

```
double val1=PMT(Rate,Period,Amount);
```

```
double val2=PMT(0.06,30,100000.00);
```

```
double val3=PMT(Rate/12,Periods*12,Amount);
```

What would not be legal in C# would be a call such as:

```
double val4=PMT(double Rate,int NPer,double Pv); // This is illegal!!!
```

One of the most common errors I see is students confusing the declaration with the call. In the call, you are passing values--it doesn't matter what their names are (and you shouldn't be declaring variables when you call them).

## Defining C# Functions

In C#, all your functions are defined within a class definition--a topic we'll soon be considering. Nonetheless, as was the case for variables, we can examine the mechanics of defining a function before we know exactly where that definition will go in our program.

### How a Function Definition is Structured

In C#, functions are very easy to define. This is good, because most of your programming will revolve around writing and modifying functions. The basic structure of the definition is as follows:

```
Return-type Name(Argument Declarations)  
  
{  
  
    Body...  
  
}
```

The **Body** contains the actual code that implements our function. To write our own version of the PMT function, for example, we'd need to find the formula for computing a loan payment then write the code for it.

Here's a simple example:

```
int Sum(int arg1, int arg2, int arg3)  
  
{  
  
    int arg4=arg1 + arg2 + arg3;  
  
    return arg4;  
  
}
```

This function takes its three arguments, adds them together, then returns the result. The function works as follows:

- We would call the function with appropriate argument values. it might be called in a line such as: **Result=Sum(3,4,5)**
- Upon entering the function, the values for the arguments would be set. In our example, **arg1** would be set to 3, **arg2** to 4, and **arg3** to 5.
- We see a declaration of a new variable--**arg4**--which is then set to the sum of the three arguments, which is 12.

- The **return** statement causes the value of **arg4** to be returned from the function. In our example, this would be placed in the variable **Result**.

On the subject of return statements, we should note the following:

- Any function that returns a value must have one or more **return** statements.
- The expression that follows a return statement must match the return type specified for the function.
- If a function does not return a value, meaning it is declared with the return type **void**, then no return needs to be specified. The function can simply end.

## Example Functions

Two examples of C# function definitions are presented below:

```
void DemoFunction()
{
    int a=4;
    int b=5;
    int c=1;
    int d;
    int e;
    int f;
    int g;
    d = Sum(a, b, c); // d will become 10
    e = Sum(5, 6, 7); // e will become 18
    f = Sum(a / 2, b + c, 4); // f will become 12
    g = Sum(int arg1, int arg2, int arg3); // This produces an error
}
int Sum(int arg1, int arg2, int arg3)
{
    int arg4=arg1 + arg2 + arg3;
    return arg4;
}
```

Some comments:

- The DemoFunction() would not compile as written, since it has a bad line (notice the red underscores, signifying the errors). Here, the programmer made the mistake of confusing the call with the declaration.
- DemoFunction is defined without arguments and without a return value (type **void**). That means two things:

First, when called, the call would look like:

**DemoFunction();**

**// The parentheses are what tells C# you're calling a function, and can't be omitted**

Second, the function cannot be used on the RHS of an assignment or as part of an expression. Since it doesn't return a value, it must be on a line by itself. Also, it doesn't need a return statement, since reaching the end of the function produces an implicit return.

# What Are Objects?

## Learning Objectives

Upon completing this reading, the student should be able to do the following:

- Explain, in broad terms, the nature of an object
- Explain why objects are valuable in programming
- Explain the difference between data and function members of an object
- Relate objects used in programming to familiar spreadsheet concepts

## Overview

The use of objects in programming is a natural extension of the use of functions. In essence, an object is a means of organizing code and data within programs, using a technique called **encapsulation**--the process of making code and data elements more understandable by packaging them into self-contained, nearly independent units.

Objects are collections of **member** elements. Conceptually, we can think of these elements as being of three types:

1. **Data Members:** Variables and other data declared within the object. For example, a "person" object might have name, address, date-of-birth and other pertinent information included within it.
2. **Function Members:** Functions related to the object can also be members--which is key to distinguishing objects from simple collections of data, such as rows in a database table.
3. **Events:** Objects can communicate with other objects by sending **messages**, usually in response to **events** (such as a mouse click). This **event-driven** message passing that drives many object-oriented programs distinguishes it from the more procedural structured programs that used to dominate programming.

## Spreadsheet Example

In this example, we'll consider how a typical spreadsheet incorporates many of the ideas associated with objects. Consider the spreadsheet shown below:

	A	B	C	D	E	F	G	H	I
1									
2									
3									
4	<b>Location</b>	<b>State</b>	<b>Sales</b>	<b>COGS</b>	<b>Administration</b>	<b>Profit</b>	<b>ProfitPct</b>	<b>Reset</b>	
5	Topeka	KS	\$100,000	55%	\$20,000	\$25,000	25.00%	Reset	
6	Chattanooga	TN	\$40,000	60%	\$5,000	\$11,000	27.50%	Reset	
7	Pascagoula	MS	\$230,000	48%	\$64,000	\$55,600	24.17%	Reset	
8	Hoboken	NJ	\$300,000	51%	\$80,000	\$26,000	8.67%	Reset	
9	Totals:		\$670,000	51%	\$169,000	\$117,600	17.55%		
10									
11									
12	<b>Member Data</b>								
13	<b>Member Function</b>								
14	<b>Static Function</b>								
15									
16									
17									

Profit and Profit Percent are computed values, not raw data

Static functions are related to the objects, but do not operate on a specific object

On click event: Sets sales and administration to zero for the row

The table illustrates many object concepts. For example:

- Each row in the table could be interpreted as a BranchOffice object
- Data members for each BranchOffice object include:
  - string** Location
  - string** State
  - double** Sales
  - double** COGS
  - double** Administration
- Function members for each BranchOffice object include:
  - double** Profit (computed as  $(1 - \text{COGS}) * \text{Sales} - \text{Administration}$ )
  - double** ProfitPercent (computed as  $\text{Profit} / \text{Sales}$ )
  - void** Reset (sets values of Sales and Administration for the object to 0)

In addition, we might make the following further interpretations:

- The 4 rows, taken together, might represent a **collection** of objects. How collections are stored is central to Module 3.
- The buttons (which call the Reset function) represent an example of event handling. Later in this module, we'll see how to attach an event handler to a C# button.
- The totaling functions at the bottom might be viewed as examples of **static** functions. Such functions are conceptually associated with a class, but do not operate on an individual object. These functions are commonly encountered in libraries, and the Main() function--used to start every C# program--is another example of a static function.

In the next segment, we'll begin to examine the process by which we can create and use our own C# objects.

# Classes and Namespaces

## Learning Objectives

Upon completing this reading, the student should be able to:

- Explain the differences between objects and classes
- Describe the role namespaces play in C# programming
- Understand the relationship between program files and class definitions
- Explain how the programmer and the Visual C# Express Environment work together to create some classes
- Represent a simple class as a UML object
- Take a simple object and create a corresponding class definition file
- Explain how keywords such as **public** and **private** impact how classes and members can be used
- Understand how properties are defined and the role they play
- Define the terms interface and implementation, in class terms

## Overview

In the four previous readings in this section, we introduced the concepts of variables, operators, functions and, finally, objects. To create your own object-oriented program, you need to combine all these elements. This section begins by exploring some of the practical aspects of C# programming. These include:

- The use of namespaces and how files relate to programs.
- Creating definitions of simple classes, focusing on writing data and member functions.
- Making objects and members accessible or inaccessible to other objects.
- Property members, which mix characteristics of member functions and member data.

We conclude with a more abstract discussion related to program design, and how classes should conceptually be divided into interface and implementation elements.

## Example: Class Definition Files

Before talking about creating objects in an abstract way, it is useful to present an example from a C# project, and see what elements we can identify. Consider the excerpts from the two files presented below:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Assignment1
{
    public partial class MainWindow : Form
    {
        private Student sGrandon;
        private Student sClare;
        private Student sTommy;
        private Student sJonathan;
        private Student sCurrent;
        public MainWindow()
        {
            InitializeComponent();
            sGrandon = new Student("Grandon Gill", 51, pictureGrandon, "Facu

```

The above file, presented first, is part of a completed Assignment1 project. Looking through it, you might notice the following:

- It begins with a series of **using** statements. These statements identify **namespaces** from the .NET framework that may be referenced within the file.
- We see a **namespace** statement specifying Assignment1 as the namespace. As we will discover, every time a project is created, a namespace with that name is created for it.
- We see a line that reads: **public partial class MainWindow : Form**. This is the heart of our definition. As we will discover, **public** means the class can be used by other classes, **partial** means the class definition is spread across more than one file (as opposed to one class per file, which is the normal way of defining a class), **class** tells us we are beginning a class definition, **MainWindow** tells us that we are naming our class MainWindow. The **: Form** tells us that we are not creating this class from scratch but, instead, are using the .NET **Form** class (which is defined by Microsoft, not by us) as a starting point, a process known as inheritance.
- The lines beginning with **private** look like data declarations--which they are. Specifically, we're creating 5 member variables, each of which refers to a **student** object (instead of referring to primitive value types, as we have done before).
- The next line looks a bit like a function declaration, except there's no return type. This is actually the start of a special function, known as a **constructor** function.

Now, we'll look at the contents of a second file:

```

namespace Assignment1
{
    partial class MainWindow
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                //
            }
        }
    }
}

```

As you'll notice, this file also specifies the Assignment1 namespace and includes the **partial class MainWindow** line. The purpose of this file, then, is to continue the definition of the MainWindow class that was started in the previous file. The reason the definition is spread over two files is simple:

- The first file is designed to hold all the code written by the programmer who is creating a form-based application (similar to the one developed for Assignment0).
- The second file contains the code that Visual C# Express Edition generates when the programmer does things like adding pictures, text controls and buttons in the form editor. This file is definitely "look but don't touch", since it is rewritten every time the user makes a change to the form layout.

Our goal in this reading, then, is to give you enough background on namespaces and class definitions so that these explanations start to make sense. A good way of testing your understanding is to go back once you're done reading (and watching the lectures) to see if the explanations are becoming any clearer.

## Namespaces

Namespaces are very simple. The only problem with explaining them to students is that until you've run into problems caused by the absence of namespaces, they seem like more effort than they are worth.

Namespaces are intended to solve problems as follows. As we have seen, when we declare a *variable* or *function*, we give it a name. The same applies to a *class*, which--as we shall see--is used to define an object. The difficulty here is that there are millions of programmers out there, all of whom are giving names to their classes. Suppose you want to use one of their libraries of classes (i.e., collections of code containing classes they've defined). Perhaps you are buying it, or

it has been available open source. When you load it into your existing application, you discover a glitch: one or more of the classes you purchased uses the same names as classes you've already developed. This is no minor problem, by the way. You can spend weeks attempting to change the names in the purchased files or your own. And it's an incredible waste of time.

Defining a namespace accomplishes two things:

- It allows you to disambiguate objects. Specifically, an object called **YourNamespace.CommonClassName** is entirely different from **OtherGuysNamespace.CommonClassName**, even though the class name is the same.
- It can reduce the time your development environment spends trying to find where objects you are using are defined. If you tell it to look in certain namespaces, it can ignore the rest.

Namespaces are so useful in C# that you are basically required to use them. Whenever you create a project (such as Assignment0, in the previous module) a namespace of that name is automatically created, and all your code is then embedded within that namespace, i.e.:

```
namespace ProjectName  
  
{  
  
    ...Your class definitions go in here  
  
}
```

If your project has multiple files--and nearly all of them do--each file will embed its contents within the namespace braces.

C# allows namespaces to be nested, thus you could define:

```
namespace Level1  
  
{  
  
    namespace Level2  
  
    {  
  
        // class definition for MyClass might be located in here  
  
    }  
  
}
```

To refer to a class, MyClass, within the inner level, we'd just use **Level1.Level2.MyClass**.

In the second example file, there was a line:

```
private System.ComponentModel.IContainer components = ...some weird expression... ;
```

Knowing what we now know, and ignoring the private keyword for the moment, we can see this is a declaration:

- of a variable called **components**
- that is an **IContainer** object
- whose class definition happens to be located in the **ComponentModel** namespace
- that is nested within the **System** namespace.

Not very pretty, but then .NET has thousands of built-in definitions in **System**. Namespaces can be very useful in helping you find them.

The main problem with implementing namespaces is that they add a lot of typing and produce very long names. To reduce this burden, the **using** statements at the top of the file can be helpful.

When a statement such as:

```
using System;
```

appears, it means that any class or namespace defined within system can be referenced without the System prefix--so long as it doesn't lead to ambiguity with a similarly named class. Thus:

```
private System.ComponentModel.IContainer components = ...some weird expression... ;
```

could have been rewritten:

```
private ComponentModel.IContainer components = ...some weird expression... ;
```

Similarly, if we'd written:

```
using System.ComponentModel;
```

at the top of the file, we could have referred to it as:

```
private IContainer components = ...some weird expression... ;
```

That's pretty much all you need to know about namespaces.

## UML Class Diagrams

**UML**, the **Universal Modeling Language**, is a graphic specification used for designing programs. Although it consists of dozens of types of diagrams--only a few of which will be considered in this course--the one most frequently used is the **class diagram**. These diagrams are very simple, and can be explained largely through example.

We begin with the spreadsheet example, already used to illustrate the notion of an object.

	A	B	C	D	E	F	G	H
1								
2								
3								
4	<b>Location</b>	<b>State</b>	<b>Sales</b>	<b>COGS</b>	<b>Administration</b>	<b>Profit</b>	<b>ProfitPct</b>	<b>Reset</b>
5	Topeka	KS	\$100,000	55%	\$20,000	\$25,000	25.00%	Reset
6	Chattanooga	TN	\$40,000	60%	\$5,000	\$11,000	27.50%	Reset
7	Pascagoula	MS	\$230,000	48%	\$64,000	\$55,600	24.17%	Reset
8	Hoboken	NJ	\$300,000	51%	\$80,000	\$26,000	8.67%	Reset
9	Totals:		\$670,000	51%	\$169,000	\$117,600	17.55%	
10								
11								
12	<b>Member Data</b>							
13	<b>Member Function</b>							
14	<b>Static Function</b>							
15								
16								
17								

Profit and Profit Percent are computed values, not raw data

Static functions are related to the objects, but do not operate on a specific object

On click event: Sets sales and administration to zero for the row

As you may recall, our BranchOffice object, representing the information contained in one row of the table, included both data elements and functions. This fits nicely into a UML class diagram, which is represented as a rectangle broken into three sections:

- The class name
- The data members (followed by an optional : data-type)
- The function members (with an optional argument list and followed by an optional : data-type)

Thus, the UML class diagram for our BranchOffice would appear as follows:

<b>BranchOffice</b>
+ Location : string + State : string + Sales : double + COGS : double + Administration : double
+ Profit : double + ProfitPercent : double + Reset()

The + sign preceding each member name signifies that the member is available to any program using the class (a.k.a., a **public** member). We'll discover why this is important later. A - sign designates a member as **private**.

## Class Definitions

Within a typical project namespace, you'll have a series of class definitions. Class definitions typically take the following form:

```

public class Class-Name
{
    Data-member-declarations

    Function-member-declarations
}

```

Since we already know how to declare variables and to define functions, it seems as if we're pretty much ready to go. It's worth making a few comments, however:

- Classes are normally given **public** access, meaning they are visible to other classes in the namespace and using the namespace.

- Data and function members can appear in any order. These are also declared **public** or **private** (the default). Members declared **private** can be accessed only within the class itself, meaning they can't be used in other places--such as the project's Main() function.
- Members declared **static** do not apply to individual objects. Most common example: general-purpose functions. For example, the **Convert** class provided by .NET contains tons of **static** functions for converting data from one type to another.

## Example Class Definition

Let's return to the example used to illustrate objects in the previous reading. As you may recall, each row could be viewed as being a BranchOffice object.

	A	B	C	D	E	F	G	H
1								
2								
3								
4	<b>Location</b>	<b>State</b>	<b>Sales</b>	<b>COGS</b>	<b>Administration</b>	<b>Profit</b>	<b>ProfitPct</b>	<b>Reset</b>
5	Topeka	KS	\$100,000	55%	\$20,000	\$25,000	25.00%	Reset
6	Chattanooga	TN	\$40,000	60%	\$5,000	\$11,000	27.50%	Reset
7	Pascagoula	MS	\$230,000	48%	\$64,000	\$55,600	24.17%	Reset
8	Hoboken	NJ	\$300,000	51%	\$80,000	\$26,000	8.67%	Reset
9	Totals:		\$670,000	51%	\$169,000	\$117,600	17.55%	
10								
11								
12	<b>Member Data</b>							
13	<b>Member Function</b>							
14	<b>Static Function</b>							
15								
16								
17								

Profit and Profit Percent are computed values, not raw data

Static functions are related to the objects, but do not operate on a specific object

On click event: Sets sales and administration to zero for the row

Variables / Operators / Functions / **Objects**

If we wanted to create a BranchOffice class in C#, what might it look like? Shown below is how such a class definition might be implemented:

```

namespace DemoFunctions
{
    public class BranchOffice
    {
        // Data members
        public string Location;
        public string State;
        public double Sales;
        public double COGS;
        public double Administration;
        public double Profit
        {
            get
            {
                return Sales * (1 - COGS) - Administration;
            }
        }
        public double ProfitPercent
        {
            get
            {
                return (Sales > 0) ? Profit / Sales : 0.0;
            }
        }
        public void Reset()
        {
            Sales = 0;
            Administration = 0;
        }
    }
}

```

What we see is the following:

- Location and State data would be held in **string** members.
- Sales, COGS and Administration data would be held in **double** members.
- Profit and ProfitPercent are defined as **property** members. They will be explained on the next page.
- Reset() is defined as a member function--no arguments, no return values. When called, it would set Sales and Administration members to 0,0 (as specified in the spreadsheet).

Thus, examining our definition, we seem to have incorporated most of what we talked about in the spreadsheet example--ignoring the **static** totals at the bottom for the time being. All that remains is to explain the two **property** members: Profit and ProfitPercent.

## Property Members

Very often, when you create a class you will want to control access to some or all of the information in the class. One way of doing this is to declare a member **private**, which means it is available only within the class. A programmer who wants to use your class in his or her application is out of luck.

A much more powerful way of controlling access is to define the member as a **property**, which is a member function that looks a lot like a variable.

## Defining Property Members

To define a property member, you begin as if you were declaring a public data member then, instead of ending the declaration with a semicolon, you add braces into which you insert a **get** and/or **set** construct.

- The **get** construct, which always includes a return statement, identifies what happens when someone tries to look at the value of the member, a.k.a. read the member.
- The **set** construct determines what happens when the program attempts to assign a value to the property, a.k.a. writing to the member. It uses the keyword **value** to refer to whatever expression is being assigned.

The easiest way to understand how these work is to look at some examples.

## Read-Only Properties

In our previous code, repeated below, Profit and ProfitPercent are defined as properties. Our reason for doing so is that they are supposed to be computed--as described in the spreadsheet--and should not be set by the program directly.

```

namespace DemoFunctions
{
    public class BranchOffice
    {
        // Data members
        public string Location;
        public string State;
        public double Sales;
        public double COGS;
        public double Administration;
        public double Profit
        {
            get
            {
                return Sales * (1 - COGS) - Administration;
            }
        }
        public double ProfitPercent
        {
            get
            {
                return (Sales > 0) ? Profit / Sales : 0.0;
            }
        }
        public void Reset()
        {
            Sales = 0;
            Administration = 0;
        }
    }
}

```

In looking at the two members, we see that only the **get** construct is present. That makes these properties read-only. In other words, C# will not let you set their values directly. They are defined as follows:

- Profit: computed, using the formula described in the spreadsheet.
- ProfitPercent: One thing you always want to do in your programs is to avoid dividing by 0. In computing profit percent, we use the conditional operator to check if *Sales* is positive (i.e., non-zero). If it is, we can divide profit by sales. If not, we return 0. (You may recall, the conditional operator is like the =IF() function in Excel, which is to say what we've done is like the Excel formula =IF(Sales>0,Profit/Sales,0).

## Read and Write Properties

A lot of the time, we'll want to define properties that both set and get the values of our data. A modified version of the Sales and COGS members, shown below, demonstrates this.

```
// "Typical" property definition
private double m_sales;
public double Sales
{
    get
    {
        return m_sales;
    }
    set
    {
        m_sales = value;
    }
}
// Property definition that controls access
private double m_cogs;
public double COGS
{
    get
    {
        return m_cogs;
    }
    // Prevents COGS greater than 200% from being set
    // using the conditional (? :) operator
    set
    {
        m_cogs = (value >= 0 && value < 2.0) ? value : 0.0;
    }
}
```

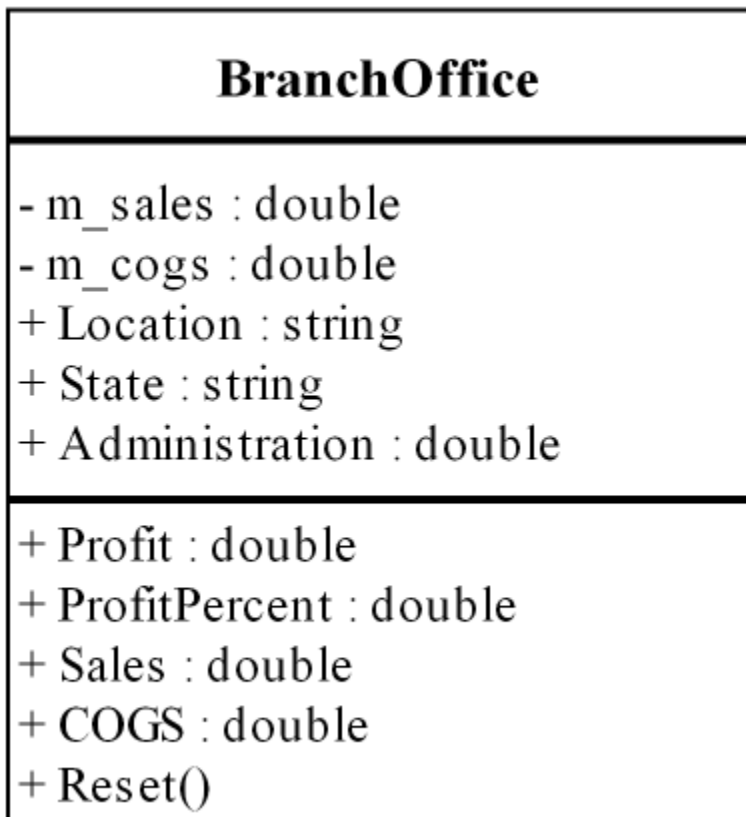
The Sales property represents a typical generic property. The **get** member returns its value and the **set** member assigns it the value passed. To actually store the data, we use a **private double** member (m\_sales). At first glance, this would seem to offer no advantages over using a public data member for Sales, as it was previously defined. There are, however, certain benefits to using public properties even if you are doing nothing but hiding a private member. For example, some Windows form objects will automatically display properties if you assign a variable to their data sources. We'll use this to our advantage in Module 3. On a more personal note, as part of Assignment 1, you'll be required to modify the public data member code and make them properties.

The COGS property implementation is considerably more interesting. Here the get (read) property is a simple return of its private data store (m\_cogs). The set (write) property, on the

other hand, does some validation--making sure the value is between 0.0 and 2.0. We might do this because users are sometimes confused about how to enter a number like 20%--is it 0.2 or 20.0. What we assumed here is that any number greater than 2.0 (200% COGS--which is still frighteningly high) is invalid and so--using the conditional operator--we set the private store (m\_cogs) to 0 instead. (If we knew how, we might also pop up a dialog box telling the user we think it's an error).

### ***Revised UML Diagram***

After changing the class definition in this way, the revised UML class diagram might appear as follows:



The changes from the original are that SALES and COGS have been moved down to the function area, and the internal data elements (m\_sales and m\_cogs) have been added to the member data area, marked as private with the - sign.

## Interface and Implementation

The use of public specifiers, private specifiers and properties actually represents an important aspect of object-oriented design, separating *interface* from *implementation*. The term "user interface" is frequently used in terms of applications to describe how a user interacts with the application. When we're talking about program code, the "user" is likely to be another programmer who wants to use your class definitions in building his or her applications. To make that cost-effective, he or she cannot be expected to examine every line of code (if you provide it--which you may not choose to do) to determine how it works. By defining specific functions, data members and properties as being public, you give that programmer a clear sense of what it is okay to access in a program where the class is used. If you make everything public, however, you provide no insights into what should and should not be touched. Thus, everything that you make public should be considered part of the interface.

The implementation consists of the guts of your class--what makes it work internally. Implementation members are of greatest interest to the class designer. Some programmers using your class may also be interested--just as some drivers are also interested in setting the timing of their spark plugs--but, for the most part, the implementation is best left to the original programmers who built the class. Would you be interested in knowing the mechanics behind all the .NET classes that you'll be using (even if Microsoft would tell you)?

As a general rule, you might consider adopting the following guidelines for C# programming:

- Make all data members private.
- Implement public properties for all data members that users might need access to.
- Make those functions designed to be called by class users public (this will be most functions). If, however, you can't come up with a clean description of what a function does, it should probably be private.

# Creating Objects

## Learning Objectives

Upon completing this reading, the student should be able to:

- Describe the object lifecycle
- Explain how to create an object based upon a specific class definition
- Define constructor functions to initialize data members of the class
- Explain how the assignment operator--applied to most objects--differs from assignment of value types

## Overview

Before we can use objects in our programs, we must create them--acquiring the memory we need to hold the data members that are included within our object definition. This creation process is accomplished using the **new** operator, which is the key to the difference in how normal objects and primitive value types (already discussed) are handled.

When defining a class, the programmer may include one or more constructor functions in the definition. These can then be called, as part of the **new** operator processing sequence, to accomplish specific initializations. Such functions are easily identified, having the same name as the class itself and no specified return type (not void, no type at all).

In this section, we'll look at examples of the object construction process.

## Classes vs. Objects

When you create a class, as was done in the previous reading, *you are **not** creating an object*. To explain this using a number of analogies:

- When you create a database table design, you do not automatically populated it with actual data.
- When you create a blueprint for a house, the house is not automatically built.
- When you acquire a recipe, the meal it describes does not automatically appear.

Thus, once we've created classes, we will still have to write code that creates objects from those classes. Otherwise (with the exception of the already-mentioned **static** elements of a class), we can't do anything with them.

## The Object Lifecycle

Objects in the program follow a specific lifecycle that proceeds as follows:

- *Prerequisite:* Programmer defines class. In C#, any object that we create must begin with a class definition--although this is not true in all object-oriented programming languages.
- Object is created from class, acquiring memory from the operating system (OS). This is accomplished using the **new** operator.
- Object is used within program by variables, etc. that refer to it. How to access elements within an object is the subject of the next reading.
- When all references to the object in the program are gone, the object's memory is marked for garbage collection. C# uses what is called garbage-collected memory. One way to think of this is that each object has an internal counter that keeps track of how many variables and other data elements refer to it. As soon as that number reaches 0, we don't need to hold on to the object's memory any longer.
- When time allows, garbage collection restores memory to OS. Since reclaiming memory from discarded objects can take time, .NET and the operating system attempt to schedule for times when little else is happening, or when available memory is getting scarce.

## Object Construction

To create an object, the **new** operator is used, followed by the class name. Arguments (in parentheses) may also follow that name, signifying the presence of constructor functions.

Example (using BranchOffice class example):

```
BranchOffice myBranch=new BranchOffice();
```

When you declare an object, but don't create one, its value is null, e.g.,

```
BranchOffice aBranch; // aBranch==null
```

```
aBranch=new BranchOffice(); // Object created
```

Only value types (e.g., numbers, Boolean) and strings objects can be created without **new**.

## Constructor Functions

Often, when you create objects, you want to initialize their member values. For this reason, special functions--referred to as constructor functions--may be defined that are automatically called upon object creation. One or more constructor functions may be defined for any object. Such functions are defined as follows:

- Their name is exactly the same as the class name.
- They have no return value declared (not even void). These are the only C# functions that are typically defined in this way.
- If more than one constructor is defined, the functions must differ with respect to their arguments.

To call a constructor with arguments, include those arguments (in parentheses) after class name in new statement.

Some examples of constructor function definitions are presented in the code below, for our example BranchOffice class:

```
public class BranchOffice
{
    // Constructor 1
    public BranchOffice()
    {
        m_cogs = 0.0;
    }
    // Constructor 2
    public BranchOffice(double sales, double cogs, double admin)
    {
        Sales = sales;
        COGS = cogs;
        Administration = admin;
    }
    // etc...
```

If we were creating BranchOffice objects within our program, the code to do so might appear as follows:

**BranchOffice b1=new BranchOffice(); // Calls the 1st constructor**

**BranchOffice b2=new BranchOffice(100000.00,0.55,20000.00); // Calls 2nd**

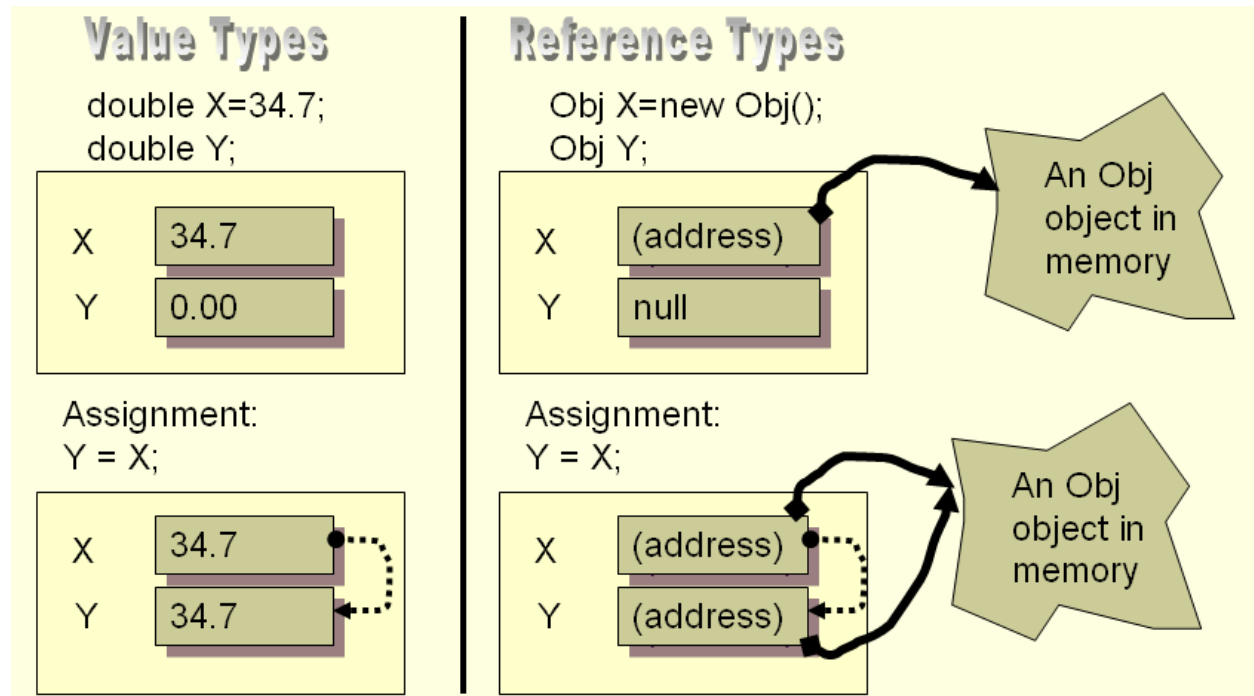
In the second example, the b2 object that was created would have sales of 100000.00, COGS of 55% and Administration of 20000.00, all based on the values we passed in.

## Assigning Class Objects

In addition to the already-mentioned use of the **new** operator, typical classes differ from value-types with respect to what assignment means:

- Since value types are defined with their own value stores, when you assign a value to a value type, the contents of the value store (e.g., the integer or real number) is replaced by what was on the right hand side (\*RHS) of the assignment. In other words, a copy of the RHS is made.
- Class variables, in contrast, can be thought of as pointers to the object created with **new**. That means that when you assign one to another, they both point to the same memory store (i.e., the same object).

Conceptually, **value types** and **reference types** (as most class-created objects are called) can be viewed as being fundamentally different in what a variable holds. For variables, they hold a value. When you assign them, the value is copied. Reference types hold a reference (an address) to an object that is somewhere else in memory--**null** is used to indicate the reference has not yet been initialized. When you assign them, the address is copied (meaning both variables point to the same object). The object itself is not copied. This is illustrated in the diagram below.



As an example, consider the following code fragment:

```
BranchOffice br1=new BranchOffice();
```

```
BranchOffice br2=br1; // No new object created
```

Since br1 and br2 both refer to the same object, any changes to values within object br1 are going to be reflected in br2 and vice versa. For example, if you pass an object into a function as an argument, any changes to the object made within the function would be reflected in the original variable. In this way, functions that don't return a value--or that need to return more than one value--can send additional data back to where they were called from.

# Using Objects

## Learning Objectives

After completing this reading, the student should be able to:

- Write code that accesses public data members within objects
- Write code that accesses public properties within objects
- Write code that calls public member functions within objects
- How to access data, function and property members in objects embedded within another object
- Identify static functions and call them
- Write an elementary console application that interacts with a terminal

## Overview

Now that we have learned to define classes and create objects, the key task we have left to learn is how to actually use those objects within our programs. Naturally, the key to using objects is being able to access the individual members (data, property and function) within an object. Fortunately, this is simply done. In the same way we used it to gain access to classes in namespaces, the . (period) operator allows us to access class members within an object.

In this section we'll also examine more closely the use of **static** members. Many classes contained in the .NET framework provide static member functions that can be extremely useful for accomplishing basic tasks. We will demonstrate some of these in a console application, which allows us to write code that resembles old terminal programs.

## Benefits of Object-Oriented Programming

Before looking at the mechanics of accessing data within objects, it is useful to recall why we chose to program with objects in the first place. Specifically, we learned that the ability to represent related data and functions in the same vicinity--e.g., within a class--provides us a means of reducing the complexity of our programming. If, in addition, we can make these objects as independent of each other as possible, it will simplify our task even further.

This clustering of related data and functions together, combined with making these clusters as independent of each other as possible, form the basis of *encapsulation*--one of the major benefits of object-oriented programming. It is central to this section. The other two beneficial OOP capabilities--the ability to *inherit* code from other objects and the ability to alter behavior in inherited objects (*polymorphism*)--are introduced in later sections.

## Accessing Object Data and Functions

Accessing members of a created object is surprisingly easy. How we do so depends on whether or not the members are being accessed from within another member function.

### From Within Member Functions

Within member functions, we just refer to the data and function elements (which may be public or private) by name--as we have seen in a number of previous examples. We may also preface the name with the **this** keyword. For example, within a BranchOffice member function, we might refer to the Sales property member as either:

**Sales**

**this.Sales**

These two forms are completely equivalent. The main advantage to using the **this** form is that it immediately presents an auto-completion list of member names within the Visual C# Express Edition editor.

### From Outside the Class

Accessing members from objects defined outside of the class employs the same . (period) operator used to navigate through namespaces. Specifically, we get at object members in the following way:

**Object-name . public-data-name**

**Object-name . public-property-name**

**Object-name . public-function-name ( arguments )**

As we've frequently found to be the case, the best way to illustrate this is through examples. Once again, drawing upon our BranchOffice example:

```
BranchOffice branch1 = new BranchOffice(100000.0, 0.55, 20000.0);
double admin = branch1.Administration; // admin is now 20000.00
branch1.Location = "Topeka"; // Assigning values through public members
branch1.State = "KS";
double profitpct = branch1.ProfitPercent;
// Calling member function
branch1.Reset();
```

Had we failed to create a BranchOffice object by calling **new** (first line), the remaining member accesses would have failed, since you cannot access members of a **null** (uninitialized) object.

Doing so will result in an exception being thrown, terminating the program unless you handle it (covered in Module 2).

## Accessing Composed Objects

Many times, your class will include member data elements that are not primitive value types (e.g., numbers) but are, instead, other objects. If you were designing a Car class, for example, it might include an embedded Engine object--to hold data specifically related to the engine. Such **composition**, as it is called, is good practice in object-oriented design, and is central to effective encapsulation. But that leaves the question: how do we access members within composed objects?

The answer is completely intuitive--we just continue to use the . (period) operator as many times as necessary--just as we did when moving into namespaces. For example, the **string** element used to store Location in our BranchOffice example is, itself, an object with many properties. One of these properties, a read-only property named Length, will tell us how many characters are in the string. Thus, the expression:

**branch1.Location.Length**

would refer to the length of the Location string within the branch1 object. In our example above, the value would be 6--the number of characters in "Topeka".

## Static Members

Members declared to be **static**--nearly always member functions or properties if they are public--require a slightly different approach to access. If you want to call a normal member function, you'll always be applying it to some object you've got hanging around. Static members, however, don't relate to a specific object. That means they can be called without one. What we use instead is the class name. Thus, if HomeOffice were declared to be a static property of our BranchOffice class (since all branches share the same home office), we would access it using:

**BranchOffice.HomeOffice**

## Example static Calls

Static functions play an important role in .NET for two reasons:

- Many .NET classes, such as Math, Console and Convert, provide useful collections of static member functions that perform tasks you probably wouldn't want to do yourself (like compute the cosine of an angle)
- Every .NET application has, within it, a **Main()** function that is declared to be a static member of the program class. This is the entry point for your program, as we shall see in upcoming segments.

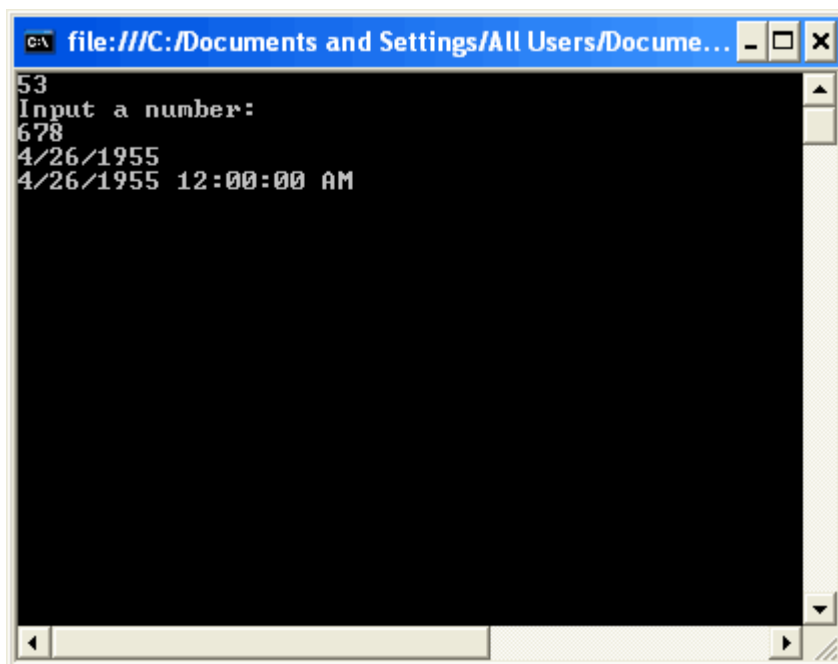
To demonstrate the use of static functions, consider the code below, which makes use of a number of static functions including:

- `Console.Write()`: Displays the text equivalent of 18 different types of data (i.e., 18 versions of the function are defined with different arguments).
- `Console.WriteLine()`: Same as `Console.Write()` except it adds a new line to the end.
- `Console.ReadLine()`: Waits for a line of text to be typed in by the user, followed by a return, then returns it as a string.
- `Convert.ToInt32()`, `Convert.ToDouble()`, etc.: Takes a variety of arguments (e.g., strings, other numbers) and converts them to the target type (specified after the `To`)

For fun, a **Date****Time** object is also included, demonstrating how you can use an object's interface members without knowing how its internal code works.

```
z=53;
Console.Write(z); // writes different data types to the console--18 versions exist
Console.WriteLine(); //writes a carriage return to the console
Console.WriteLine("Input a number:");
string sInput = Console.ReadLine(); // Reads line from keyboard
int nInput = Convert.ToInt32(sInput); // Converts string to integer
double dInput = Convert.ToDouble(nInput); // Returns double with value of nInput
DateTime aDate = Convert.ToDateTime("04/26/1955"); // Converts my birthday to DateTime
Console.Write(aDate.Month);
Console.Write('/'); // Writes a / as a character
Console.Write(aDate.Day);
Console.Write("@"/"); // Writes a / as a string
Console.WriteLine(aDate.Year); // Writes year followed by return
Console.WriteLine(aDate); // Writes date in its Windows-based locale format
```

The output that was produced by this code is as follows:



```
53
Input a number:
678
4/26/1955
4/26/1955 12:00:00 AM
```

By matching the input to the output, you should be able to see what statements produced what output.

# Inheritance

## Learning Objectives

Upon completing this reading, the student should be able to:

- Describe a number of benefits that result from using inheritance in programming
- Use inheritance related terminology, such as base class, child class and sub-class
- Represent an inheritance relationship using a UML class diagram
- Write a class that inherits properties from another class
- Create a constructor that calls a base class constructor
- Identify the role of the C# base class: **object**

## Overview

Inheritance is the ability to define a class that uses a previously defined class as a starting point--referred to as the **base class**-- then extends the capabilities of the base class. Inheritance is one of the three beneficial capabilities offered by object-oriented programming (encapsulation, inheritance and polymorphism), and is closely tied to the third, polymorphism, which works through inheritance.

Inheritance is easy to implement in C#. Indeed, when we created a Windows form in Module 0, we were taking advantage of inheritance. To establish an inheritance relationship, you just add a **: base-class-name** to the top line of the class definition. From that point on, all the capabilities of your base class become available in your new class, sometimes called a **child class** or **sub-class**, with no additional coding.

The only class members that don't inherit are constructor functions which, obviously, must have different names in the **base class** and **child class**. A special keyword, **base**, can be used to call a base class constructor within a child class constructor.

## Benefits of Inheritance

Prior to inheritance, if you wanted to reuse source code and add capabilities, you would typically have to make a copy of the source code and modify the copy. This led to a number of disadvantages:

- It increased the amount of identical or nearly identical code in your project.
- If an error were found in the original code, you would be forced to modify the code in two places--the original and your copy. Things got even worse if the individual working on the original code was not the same person working on the new code.
- Improvements made in the copied code that could be useful in the original code failed to make it back to the original.

When you inherit code, you incorporate your original code into your new class without copying the code. As a result:

- Your new code is much more compact.
- If you find an error in the original code and fix it, your new code is updated automatically when you rebuild the project.
- If you extend or enhance the original code as part of writing your new code, the improvements immediately flow out to any other code that uses the original code (when those other projects are rebuilt).

In addition to reuse and maintainability benefits, inheritance also offers conceptual benefits. Many complex objects that we might represent fit nicely into an inheritance scheme. Nearly all the *Window* objects inherit from key base classes. This means that if you know how to move a button object, you probably also know how to move a picture object--since their move member function is implemented in a common base class.

## Implementing Inheritance

Establishing an inheritance relationship is a fairly trivial matter in C#. You simply modify the new class definition by adding a **:** (colon) **old-class** as follows:

```
public class SubClass : BaseClass
{
    // SubClass definition
}
```

The new class you create will sometimes be called the child class or the sub-class. The class that you are inheriting from is usually referred to as the **base class**, or--less often--as the **parent class**.

## Inheriting Constructors

When you inherit, all data, property and function members become available in the child class, with no further coding. The one exception is constructor functions--which would have different names (e.g., `BaseClass()` and `SubClass()` in the above example). This situation is handled in two ways:

- Normally, the no-argument constructor (if any) of the base class will be called prior to any constructor defined in the child class.
- The **base** keyword can be used, with arguments, to identify a specific base-class constructor to be called prior to calling the child class constructor.

The format for using the **base** keyword is as follows:

```
public ChildClass(arguments...) : base(...base-constructor-arguments, which must be  
included in arguments...)  
  
{  
  
    // body of ChildClass constructor  
  
}
```

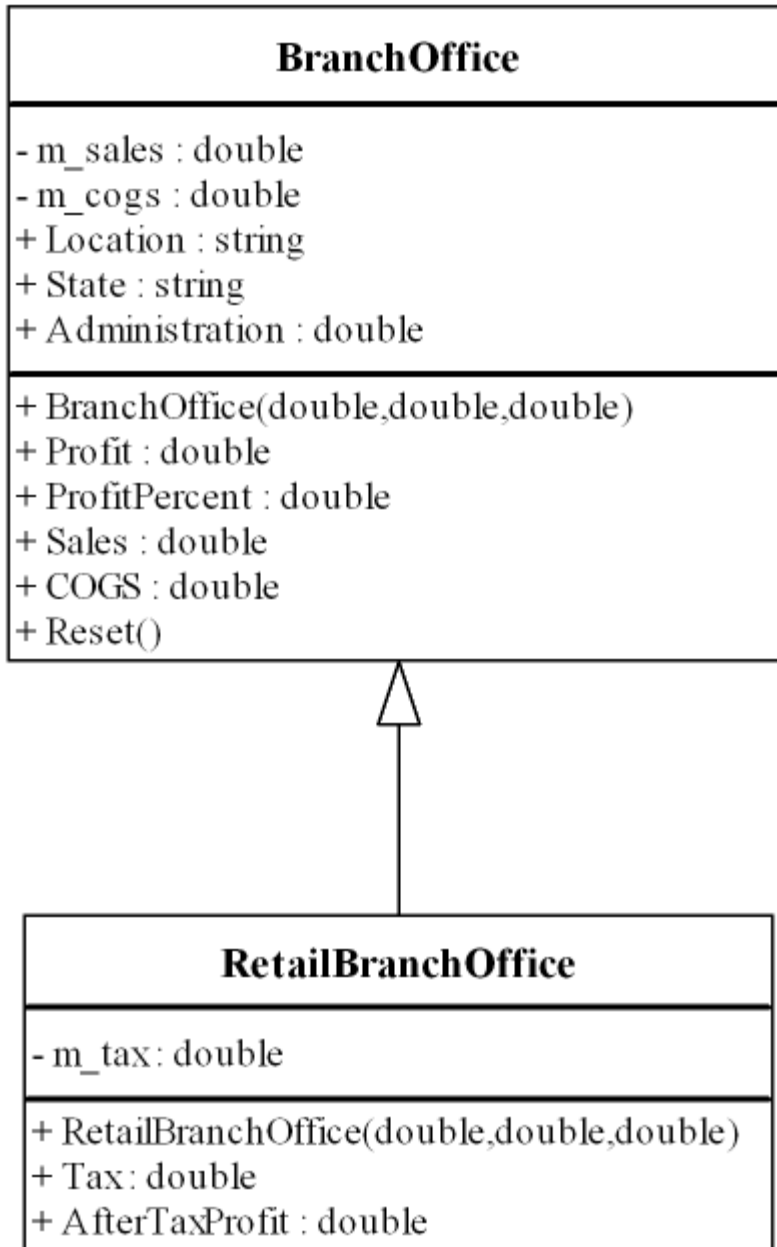
The argument list for the ChildClass constructor must include all values that are necessary to call the base class constructor, but additional arguments (useful only to the child class) may also be supplied.

## **Example of Inheritance**

To illustrate inheritance, we return to our *BranchOffice* scenario. In this case, we find that some of our branches conduct retail sales operations in addition to their other activities, and therefore have sales tax as an additional expense. For these branches, then, we want to define a *RetailBranchOffice* class that inherits from *BranchOffice*, but adds *SalesTax* and *ProfitAfterTax* properties.

### ***UML Representation of Inheritance***

In a UML diagram, an inheritance relationship is represented by showing the child class and connecting it with a line (ending in a triangle-shaped arrow) to the diagram of the base class. For example, our scenario might be diagrammed as follows:



Notice that only the changes to the base class are depicted in the RetailBranchOffice (child) class. Also be aware that the order in which members are presented is up to the designer.

***C# Code to Implement Inheritance***

The changes to the diagram are, pretty much, directly reflected in the new code, presented below.

```

class RetailBranchOffice : BranchOffice
{
    // Constructor 1 does nothing
    public RetailBranchOffice() { }
    // Constructor 2 calls the BranchOffice constructor with 3 arguments
    public RetailBranchOffice(double s, double c, double a) : base(s, c, a)
    {
        // no need to do anything beyond calling the base
    }
    double m_tax; // unspecified visibility defaults to private
    public double Tax
    {
        set
        {
            m_tax = value;
        }
        get
        {
            return m_tax;
        }
    }

    }
    public double AfterTaxProfit
    {
        get
        {
            return Profit - Tax; // Notice call to Base member
        }
    }
    }
}

```

Notes:

- Because inheritance has been used, we can access base class properties, such as Tax, even though they don't appear in our class definition.
- We include a 3 argument constructor that calls the base class constructor (i.e., the three argument constructor defined in BranchOffice) using the **base** keyword.

The power of inheritance can be further illustrated by showing code that uses a RetailBranchOffice object. This code is presented below.

```

RetailBranchOffice branch1 = new RetailBranchOffice(100000.0, 0.55, 20000.0);
double admin = branch1.Administration; // admin is now 20000.00
branch1.Location = "Topeka"; // Assigning values w/ base class public members
branch1.State = "KS";
double profitpct = branch1.ProfitPercent;
Console.WriteLine("After Tax Profit: ");
Console.WriteLine(branch1.AfterTaxProfit); // Using inherited member
// Calling base-class member function
branch1.Reset();

```

Notice how both base class and child class members are applied to our child class object.

## protected Access

When we introduced member access, two options were presented: **public** and **private**. If you are not inheriting from your classes, these two options are all you need. When inheritance is implemented, however, we discover that the two options aren't enough. Specifically:

- **public** inherits as **public**, which is what we'd want it to do. Any **public** members of the base class can be accessed in the child class without any problems.
- **private** members become *inaccessible* in child classes--they are really private! In other words your child classes are treated the same way that unrelated classes are treated. Sometimes, but not always, this is what you want.

To provide better support for inheritance, a third option is available:

- **protected** members are treated like **private** members to unrelated classes--making them inaccessible--but inherit to child classes. Thus, if a member is declared **protected** in your base class, it can be accessed in your child class.

We will see some examples where declaring members **protected** is beneficial when we examine inheriting from classes near the end of this module.

## Modifying Members

In addition to adding members to a child class, it is also possible to change the behavior of base class members by redefining them. This process is known as **overriding** members, and is closely related to polymorphism. For this reason, we will not examine this capability until a later module.

## The object Class

Many object-oriented programming languages, such as C# and Java, provide a fundamental base class that all other objects either directly or indirectly inherit from. In C#, that class is known as **object**.

The practical implications of such an inheritance scheme is that every object--including value types--has a number of built-in members that are implemented in the **object** class. These include:

- *ToString()*: The most commonly used **object** member, it takes the object and represents it in string form. For a complex class object, the representation usually isn't that useful (it is the class name). For many built-in classes, such as numbers and dates, it provides a convenient complement to the Convert library of static functions previously discussed.
- A comparison function, allowing the program to determine if one object is the same as another. (This turns out to be one context in which the **this** keyword can be useful--since an object can determine if it is being passed to itself).
- A hash coding function, convenient for implementing lookup tables (a topic in Module 3)
- And so forth...

It is useful to be aware of inheritance from **object** because it explains certain mysteries--such as why a list of members appear when you auto complete an integer.

# Console Tour

## Learning Objectives

At the end of this reading, you should be able to:

- Describe what a console project is
- Explain why console projects may still be useful
- Locate the Main() function of a console project, and explain its purpose
- Add code to a console project and run the code
- Stop a console project in the debugger and examine its elements

## Overview

This reading takes you on a tour of a console application, as generated by Visual C# .NET 2005. We begin by explaining what console applications are, and their role in programming. We then walk through the creation of a console project and examine the code that is automatically generated. We then discuss the important role of the Main() function in .NET applications and add our own simple code to that function. We conclude by running our program using the debugger.

## What are Console Applications?

Console projects return us to the early days of computing, when interaction with the program took place using a teletype or similar device that could only handle text, and could only scroll in one direction. Despite their primitive appearance, they are still used fairly regularly for a number of purposes, including:

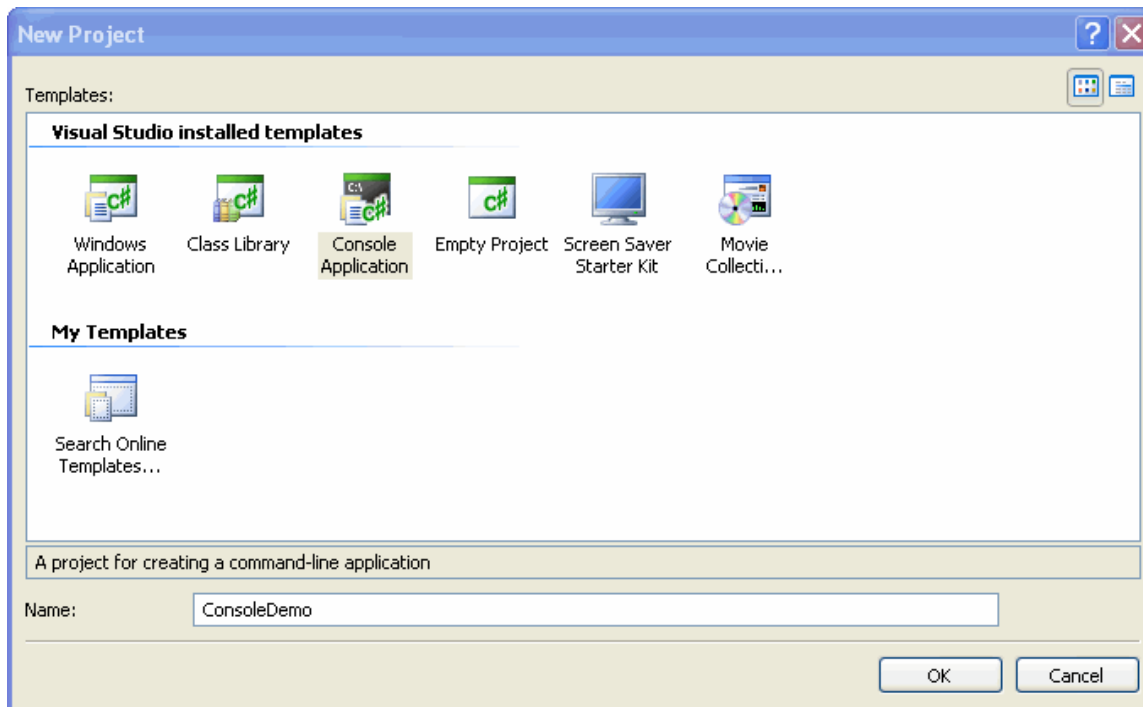
- Creating programs that don't need a graphic user interface, such as file processing programs.
- Writing and testing classes before they are incorporated into other projects, which may have graphic user interfaces.
- Building programs that interact with other programs through standard input and output. For example, CGI scripts--one of the original techniques for interacting with web forms--could be implemented as console programs.

Probably the biggest advantages of console programs are they are simple to write and relatively simple to understand. As the .NET environment has improved over time, however, console programs are less useful. A decade ago, writing a program that opened an empty window involved 5-10 pages of computer-generated code--most of which was difficult, if not impossible, to explain to novices. Today, the code for a Windows application is little more complicated than the code for a console application. Thus, in this course, we'll focus on Windows applications right from the start. Nonetheless, you need to be familiar with console applications, because if

you continue your programming education (e.g., in Java), you will find yourself dealing with the console on a regular basis.

## Creating a Console Application

Because console projects are one of the included project templates in Visual C# Express Edition 2005 (with the 2008 version being nearly identical), creating a console project is simply a matter of making the right selection, as shown below:



## Console Generated Code

Because console projects are so simple, the amount of code automatically generated for such projects is relatively trivial. Indeed, it consists of a single file, as shown below:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleDemo
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

This file contains a single class, **Program**, contained within the **namespace** named after the project (*ProjectDemo*, in this example). The **Program** class, in turn, has a single member--a **static** function called *Main()*. The *Main()* function, in turn, has an argument--**string[] args**--that we will ignore, for now. (It is actually a collection of string objects that would hold any command line arguments that we typed, had we run the program from the Windows "Run" prompt).

## The Main() Function

The *Main()* function generated by VCEE actually plays a crucial role in any runnable .NET application (as opposed to other project types, such as class libraries that can't run on their own). It provides the starting point for the application. In other words, any code that is placed in *Main()* will be executed as soon as the application starts.

## Example Main() Function

Because console applications come with an empty *Main()* function--unlike Windows applications which, as we'll see in the next section, come with code already inserted into the *Main()* function--you must add your own code to *Main()* if you want your console application to do anything.

### Sample Code

For example, consider the following *Main()* function:

```
1     static void Main(string[] args)
      {
          DateTime theDate = DateTime.Today; // Today is static property
          Console.Write(@"Today's date is: ");
          Console.WriteLine(theDate);
          Console.Write("Input your first name: ");
          string sName = Console.ReadLine();
          string uName = sName.ToUpper(); // Using ToUpper() string member
          // Note how the next statement flows across lines
          string sComment =
              (uName == "GRANDON") ? "Your name is the same as mine" :
              "Your name is different from mine";
          Console.WriteLine(sComment);
          Console.Write(@"I'm thinking of a number between 1 and 100.
Type a number: ");
          string sGuess=Console.ReadLine();
          int nGuess=Convert.ToInt32(sGuess);
          bool bRight = (nGuess == 67);
          Console.WriteLine(bRight ? "You got it right!" : "You got it wrong...");
          return;
      }
- }
```

In this case, we declare some local objects (e.g., a **Date****Time** object called *theDate*, **string** objects called *sName* and *uName*, etc.) and then call some **static** functions (as you may recall,

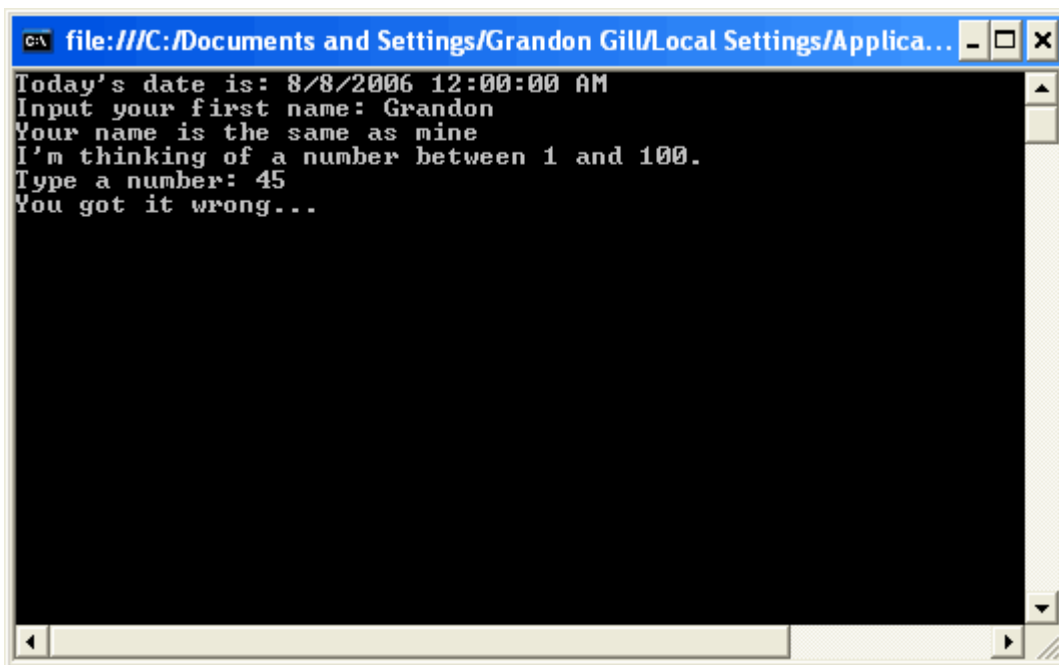
you can tell when **static** functions are called by the fact that a class name, instead of an object name, precedes the call).

To review:

- Console.Write() and Console.WriteLine() each take an argument, represent it in its textform, and display it on the screen. Console.WriteLine() adds a return.
- Console.ReadLine() takes a line of text from the keyboard and displays it on the screen.
- The Convert functions allow us to move between data types.

### *Sample Output*

Rather than explaining the code, the best way to understand it is to compare it to the output it produces, as shown below. Console applications are easy to trace in this manner, since all input and output appear on the screen.



```
file:///C:/Documents and Settings/Grandon Gill/Local Settings/Applica...
Today's date is: 8/8/2006 12:00:00 AM
Input your first name: Grandon
Your name is the same as mine
I'm thinking of a number between 1 and 100.
Type a number: 45
You got it wrong...
```

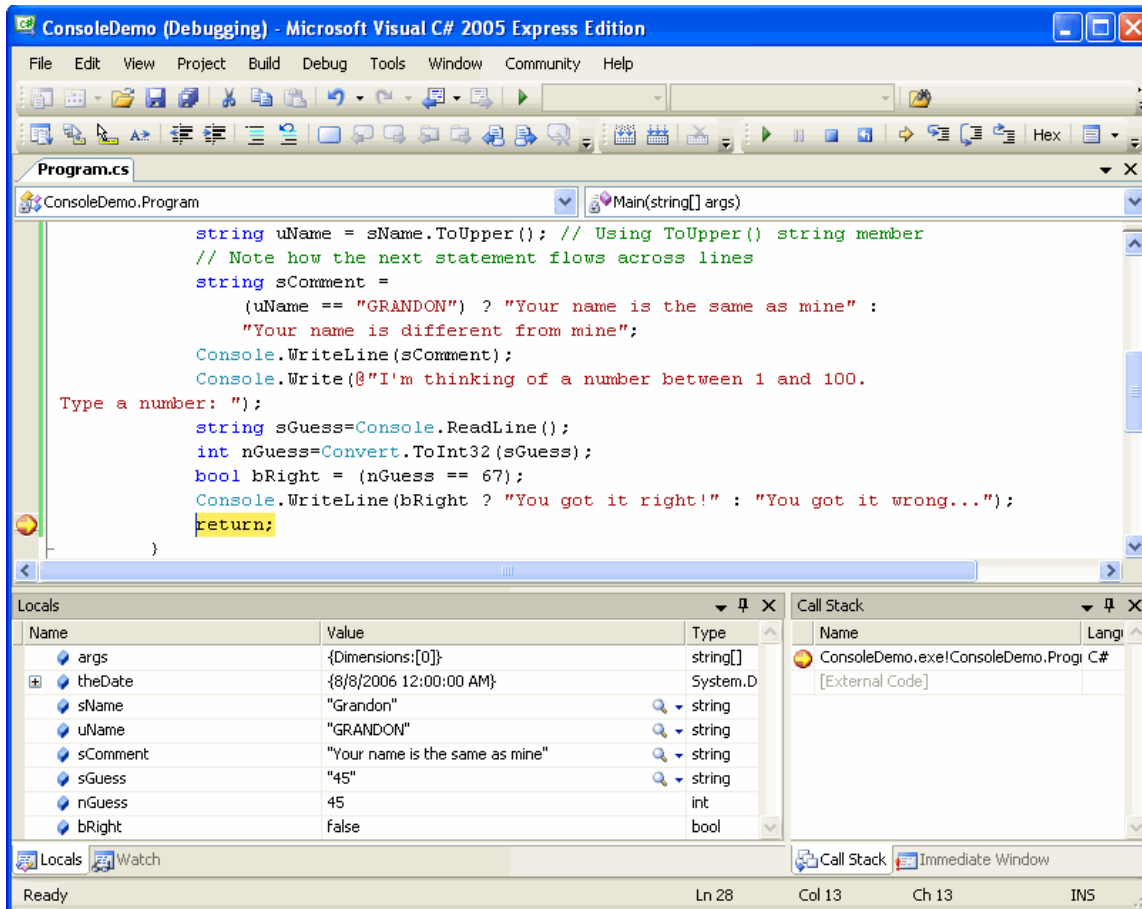
### **Introduction to Debugging**

Debugging features are among the most useful capabilities provided by the VCEE tool. While many advanced capabilities exist, we'll focus on two of them here:

- The ability to pause the program at a specific point and examine values of variables
- The ability to execute code one step at a time

## Breakpoints

The most common use of the debugger is to examine values while the project is running. To do this, we must pause the program at the desired location. This is done by clicking on the bar to the left of the line where we want the program to pause, causing a red dot to be placed there. When the program reaches that line, it pauses, as shown below.



In the example above, the breakpoint was placed at the highlighted return statement (it is highlighted because that is the current line about to be executed, also signified by the yellow arrow over the red dot).

At this point, we can then examine the current values of variables in the window below it--the "Locals" window. You are not limited to primitive objects in this examination. As signified by the (+) next to the `theDate` variable, more complex objects can be "opened up" and the values of their members inspected. Moreover, if the members are, themselves, composite objects, they too can be opened and inspected.

Like virtually every window in the VCEE environment, the `Locals` window can be moved around, undocked or hidden. For this reason, you may need to use the Debug menu or the View menu to find it or make it visible.

## Stepping

The other major debugging capability we will routinely be using is stepping through code. Whether you've built a console or Windows application, watching how variable values change as each line of code executes is a very powerful tool for: a) understanding how your code works, and b) finding logic errors in your code.

While your program is paused, you can access the stepping feature either from the Debug window, or from the debugging toolbar, shown below. (Once again, you may need to go to the View menu to expose missing toolbars).



From left to right, the icons mean the following:

- *Run*: Resume running the application
- *Step Into*: This executes the next line of code. If that code calls a function, it steps into the source code for that function.
- *Step Over*: This executes the next line of code and moves to the next line, regardless of whether or not the line calls functions. This can be a quick way of determining if a function is returning the correct value.
- *Step Out Of*: Run to the next return statement within the current function, then move back out to the line of code that called the function. This can be particularly useful if you inadvertently stepped into a function when you meant to step over it.

In the various demonstrations included in this course, you will see these capabilities used many times.

## Console Debugging

One area where console projects have a big advantage over Windows projects is in ease of debugging for one simple reason. When you pause a Windows program for debugging purposes, the ability to refresh the screen is also stopped (until you start running again), which means that it is hard to examine the output of your code. A console window, on the other hand, is like a scrollable text document and does not require refreshing. Thus, you can watch your output build step by step.

Console applications and the debugger tend to go together because console applications will simply exit when done, and their window will disappear before you can read it. For this reason, you should almost always put a break point in the return statement in your Main() function (as shown above) to give you enough time to look at the console window's contents.

# Windows Form Tour

## Learning Objectives

Upon completing this reading, you should be able to:

- List the key files generated for a Windows Form application by the Visual C# .NET 2005 or 2008 Express Edition
- Identify what code should and should not be edited
- Identify code elements generated by the form designer
- Identify where designer-generated elements are declared, constructed and initialized
- Write your own simple code to work with designer-generated elements

## Overview

When working with VCEE in this course, you will never create an application from a blank project. Instead, you will build applications cooperatively with the VCEE environment, using powerful design tools to increase your productivity. Unfortunately, that does not eliminate your responsibility to understand what the designer is accomplishing. If you are to build your own applications, you will need to have a basic idea of how Windows forms are constructed and how the designer translates your visual objects into C# statements.

This reading focuses on examining the components of a Windows Form project. Although some of the code will remain mysterious (as some lines still remain for your instructor!), you should be surprised by how much you can interpret--provided you have a reasonable understanding of the previous modules in this section. The example shown uses the VCEE 2005 version which is, for the purposes of the current demonstration, essentially identical to the 2008 edition.

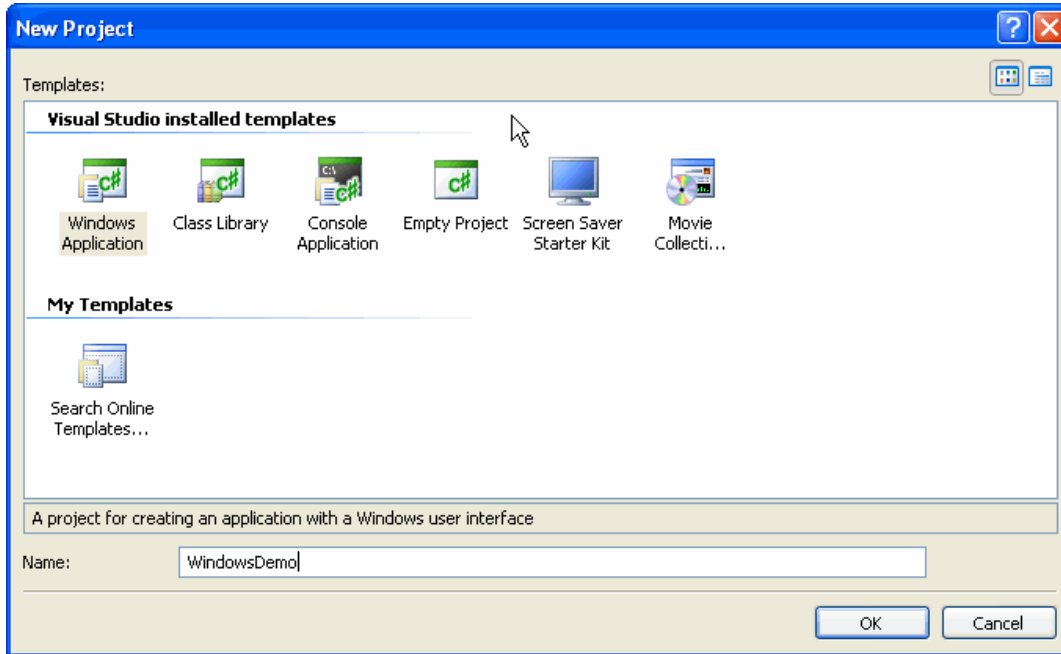
We proceed as follows:

- Create a new Windows application and look at the three key files produced.
- Add a few graphic elements to the form and see how they are translated into C# code by the designer.
- Make a few customizations to the program, using our understanding of the VCEE code and what it does.

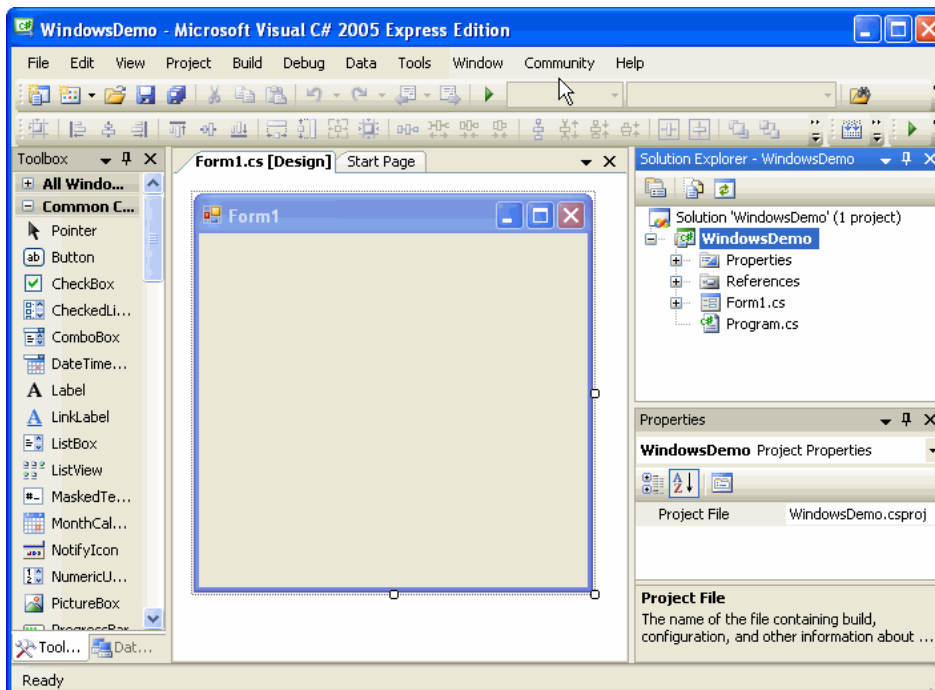
Having done this, we are almost ready to move on to far more advanced applications. All we need is a slightly greater understanding of how to implement event handling--the final topic in this module.

## A Windows Form

As we have already seen in the previous module, it is easy to create a Windows form-based application in VCEE. Just select the Windows Application template when you are creating a new project, as shown below.



In this case, the project we are creating will be called Windows Demo. When we press OK, the form designer opens, as shown below:



To the right of the designer area, we see the Solution Explorer (as always, you may need to open it from the View menu), which already shows that certain files (e.g., Program.cs) have been created. We will now examine some of these files.

## The Main() Function

As stated in the previous section, the starting point for program code in C# is a Main() function. Windows applications differ from console applications significantly in how Main() is implemented, however. In console applications, we need to put our own code in Main(). In Windows applications, Main() is used primarily to load our form (or a designated form, if a project contains more than one form). For this reason, we normally will not need to edit the Main() function--even though it is available to us. The typical Main() function for a form-based project contained in Program.cs is shown below.

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace WindowsDemo
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

As should be evident from the example, the actual launching of our form occurs in the last line, where the **static** function **Application.Run()** is called on a newly created copy of our form object (*Form1()* is the default name for the first form in an application, as shown above, unless you change it). The two functions that precede it are also **static** functions from the **Application** class. Their names suggest that they impact the styling of the form, and you can look them up in the help system to see what they actually do.

As mentioned earlier, although you can change the *Main()* function in a Windows application, you usually will not need to do so.

## Form Code: The Editable Part

If you right click the form designer window, one of the options is likely to be "Show Code". By selecting this option, you open the file containing the portion of the form's code that is intended to be edited. That generated code appears below:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsDemo
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Looking at this code, we see a very simple class definition. You should note the following:

- Because the class is a **partial** class, this is not the whole definition.
- Because the class inherits from **Form**, a .NET built-in class, it is probably a lot more powerful than it appears to be.
- The only member currently defined is a no-argument constructor, which does nothing but call a function--not defined here--called `InitializeComponent()`

In looking at this code, you should be aware that its lack of members is not a problem. Rather, its an indication of what we are currently doing, which is popping up a form with nothing on it. As we add capabilities to our form, we will be:

- Adding data members or properties as needed
- Adding member functions, as needed
- Modifying the constructor to do whatever initializations we require
- Adding event handlers

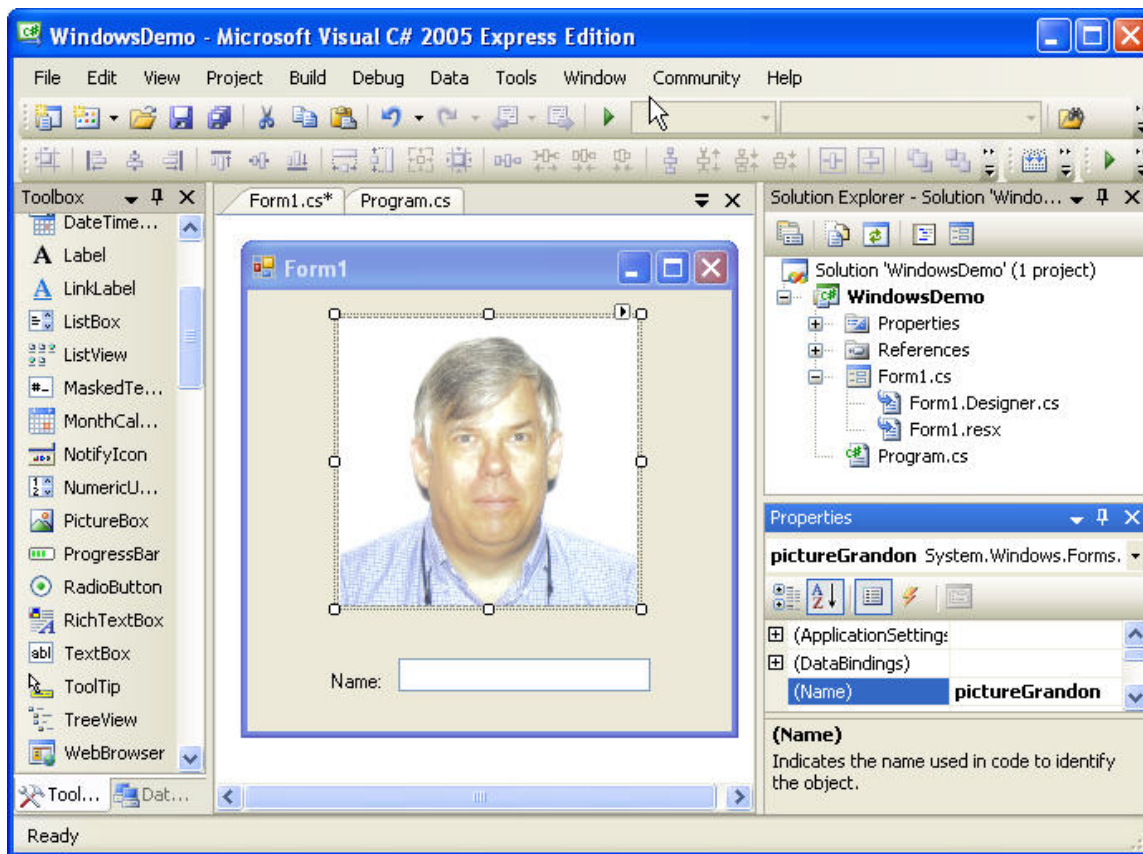
In other words, most of our programming effort--at least in this course--will revolve around customizing the behaviors of the forms we create.

## Form Code: The Generated Part

As noted, the Form1 class that we were looking at is a partial class. To find the remainder of the class definition, we need to go to Form1.Designer.cs (which can be reached by opening the Form1 object in the Solution Explorer). To make sense of what that code does, we'll need to add some elements to the form.

## Adding Form Elements

To understand our generated code, we begin by adding three graphic elements to our form, similar to what we did in Module 0. Specifically, we will be adding a picture, a label and a text editing box, as shown below.



As was also done in Assignment 0, we've changed the default names assigned by the designer to make them more recognizable. For example, the picture is now named **pictureGrandon**, as shown above.

## Generated Control Code

Now that we have something on our form, we can see how that translates to generated code.

```

namespace WindowsDemo
{
    partial class Form1
    {
        /// <summary> ...
        private System.ComponentModel.IContainer components = null;

        /// <summary> ...
        protected override void Dispose(bool disposing) {...}

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent() {...}

        #endregion

        private System.Windows.Forms.PictureBox pictureGrandon;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.TextBox textGrandon;
    }
}

```

When we open the designer code, our attention should be drawn to two areas (which I have highlighted):

- *Green (above):* We see that the `InitializeComponent()` function, called in the other part of the class definition, is defined here.
- *Yellow (below):* We see that three **private** member variables, referring to the controls that we just created, have been added to the class.

## Object Construction

Since the purpose of the visual designer is to generate code, an obvious place to look for it is within the `InitializeComponent()` function. Indeed, the first thing we see that is specific to our controls is the following code:

```

this.pictureGrandon = new System.Windows.Forms.PictureBox();
this.label1 = new System.Windows.Forms.Label();
this.textGrandon = new System.Windows.Forms.TextBox();
((System.ComponentModel.ISupportInitialize)(this.pictureGrandon)).BeginInit();
this.SuspendLayout();
...

```

The highlighted lines of code are just calls to constructor functions to create the actual form objects and assign them to the member variables. You'll see this type of code in almost any constructor for objects that have other objects as members.

The two lines below are less comprehensible. Given the names, however, you might guess that they serve to keep the form from redrawing while we initialize the various components. If you look them up in the help system (as I had to do), you'll find that this is actually what they do.

## Object Initialization

We're helped along with the next blocks of code by the fact that the system put in comments identifying what objects it was working on.

### *Text Initialization*

The two text controls (a label and a text box) map to the following code:

```
//  
// label1  
//  
this.label1.AutoSize = true;  
this.label1.Location = new System.Drawing.Point(47, 229);  
this.label1.Name = "label1";  
this.label1.Size = new System.Drawing.Size(38, 13);  
this.label1.TabIndex = 1;  
this.label1.Text = "Name:";  
//  
// textGrandon  
//  
this.textGrandon.Location = new System.Drawing.Point(91, 222);  
this.textGrandon.Name = "textGrandon";  
this.textGrandon.Size = new System.Drawing.Size(152, 20);  
this.textGrandon.TabIndex = 2;
```

What we see here is the program setting various properties for the control. Indeed, these include the properties that we set when we created the control, such as the textbox name ("textGrandon") and the label text ("Name:"). Similarly, the Size() and Point() member calls reflect where we put the control on the form. Finally, the TabIndex is used to identify the ordering of the controls when the tab key is pressed. Thus, there's really nothing mysterious going on here--just a bunch of initializations.

### *Picture Initialization*

The initialization for the picture object is a bit less obvious, but we can still get a general feel for what it does. That code is shown below.

```

//
// pictureGrandon
//
this.pictureGrandon.Image = ((System.Drawing.Image) (resources.GetObject("pictureGrandon.Image")));
this.pictureGrandon.Location = new System.Drawing.Point(54, 16);
this.pictureGrandon.Name = "pictureGrandon";
this.pictureGrandon.Size = new System.Drawing.Size(182, 175);
this.pictureGrandon.SizeMode = System.Windows.Forms.PictureBoxSizeMode.Zoom;
this.pictureGrandon.TabIndex = 0;
this.pictureGrandon.TabStop = false;

```

In point of fact, only the first line appears to be mysterious. To understand what it is doing, you need to know a bit about resources. Specifically, Windows is able to store different types of data (e.g., text, bitmaps, cursors) within executable files--saving us the trouble of keeping track of separate files. When you add a photo to the designer, as done in the example, it is moved into a file used to hold resources (you can see it in Form1.resx). What that line tells us is to load that image from the resource area. Indeed, the only line of the InitializeComponent() function that we have skipped over so far reads:

```

System.ComponentModel.ComponentResourceManager resources =
new System.ComponentModel.ComponentResourceManager(typeof(Form1));

```

This line basically ties the **resources** local variable (a **ComponentResourceManager** object, whatever that is) to the **Form1** object--telling it where to go for resources.

Thus, even the initialization of a picture isn't totally beyond our comprehension.

### *Form Initialization*

The final block of code in InitializeComponent() initializes the form itself. That code is as follows:

```

//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textGrandon);
this.Controls.Add(this.label1);
this.Controls.Add(this.pictureGrandon);
this.Name = "Form1";
this.Text = "Form1";
({System.ComponentModel.ISupportInitialize} (this.pictureGrandon)).EndInit();
this.ResumeLayout(false);
this.PerformLayout();

```

We can interpret this code as follows:

- *Above the yellow highlight and between the highlights:* we see various properties of the form being set.
- *Yellow highlight:* We are adding our components to a collection property called **Controls**. We'll examine collections in Module 2. It makes sense, however, that a form needs to keep track of the controls that are placed on it. If it did not, for example, when the form was closed, those controls might remain visible--an ugly effect, to be sure.
- *Green highlight:* it tells the form it is done initializing, so it can now draw itself.

## Why Should We Care?

Given that:

- The code we've been looking at is generated for us by the designer, and
- The code we've been looking at should *never* be edited by us

an obvious question becomes: *why should we care?*

There are two reasons. First, this code illustrates that there is nothing that mysterious about what goes on when we create a form. Second, and much more importantly, we will find--from time to time--that we want our programs to create controls for us dynamically--instead of drawing them in the editor. It turns out that the easiest way to do this is to create an empty project, add a control using the designer, then look at the code it generated. Indeed, we can (and do) even cut and paste that code into our own application.

As has been said before, the way to learn C# is to examine code that has been written!

## Customizing Our Application

The last portion of this tour involves using what we have learned to customize our application. Suppose, for example, we wanted to set up the program so that it established a default name (say Grandon Gill) in the edit box. Further suppose that we wanted to do so programmatically (as opposed to editing the `textGrandon.Text` field in the designer, as we would normally do), and store that name in a member variable. What our tour should tell us is the following:

- Adding a member variable should be no problem. We just place it in the Form1 class.
- We can initialize that memory variable in the Form1 constructor. Where doesn't matter.
- We can initialize the `textGrandon.Text` value in the constructor as well. For that control, however, the assignment must be done after the `InitializeComponent ()` function is called--since it is within that function that the textbox object is created.

Thus, the code to create our initialized textbox would look something like the following (highlighted in yellow):

```
namespace WindowsDemo
{
    public partial class Form1 : Form
    {
        private string sName; // Create a member variable
        public Form1()
        {
            // Non-graphic elements can be set before InitializeComponent()
            sName = "Grandon Gill";
            InitializeComponent();
            this.textGrandon.Text = sName;
        }
    }
}
```

# Events

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by events and message passing
- Describe the basic approach to message passing used by .NET
- Create an event handler for a button or other graphic object on a form
- Write simple code to handle the event
- Explain how the VCEE Form Designer creates the code for an event

## Overview

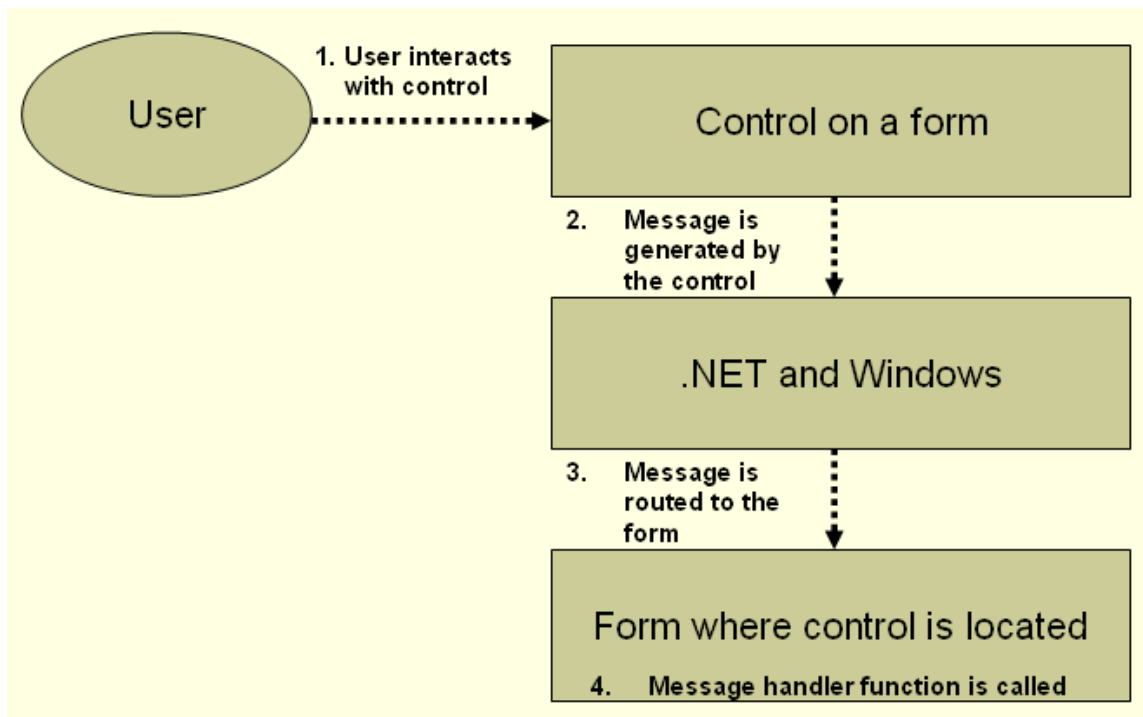
Traditional programming techniques, such as those dominating the industry from the 1950s through the early 1990s, were mainly procedural in nature. This meant that you wrote a program as if it would run from start to finish. You prompted the user for input, the user responded, you took the next step, and so forth.

Object-oriented programming tends to be dominated by a different approach, sometimes referred to as *event-driven programming*. In this approach, objects communicate with each other by passing *messages*. These messages, in turn, tend to be generated in response to *events* (such as mouse clicks, a timer or menu selections). Using this style of programming, you write programs designed to sit passively--waiting for things to happen then taking action--rather than programs written to take control of the computer and proceed until they are done. It is no surprise that event driven programming became popular with the introduction of graphic user interface (GUI) operating systems, such as Windows. In the GUI world, you assume that a lot of different programs are running at the same time. You wouldn't want all of them trying to hog the system at the same time.

In this section, we examine how Windows handles events using the .NET framework and how we implement event handling and message passing in our code. To demonstrate this, we create a button and trace what it does.

## The .NET Event Processing Cycle

A simplified version of the event processing cycle used by .NET is presented below.



The cycle proceeds as follows:

- Some action occurs, such as a user interacting with a control (e.g., clicking a button). This generates an event.
- This event generates a message, which is then passed to Windows or .NET. That event is *queued* (put in line) for processing--important because many different events may be generated concurrently and it is up to the operating system to determine the order in which they proceed.
- The message is then passed down to the appropriate form holding the control (typically via the Application object--which might also be used to respond to the event, though that is not the default behavior).
- A message handler in the form responds to the event, with the message data being passed in as an argument.

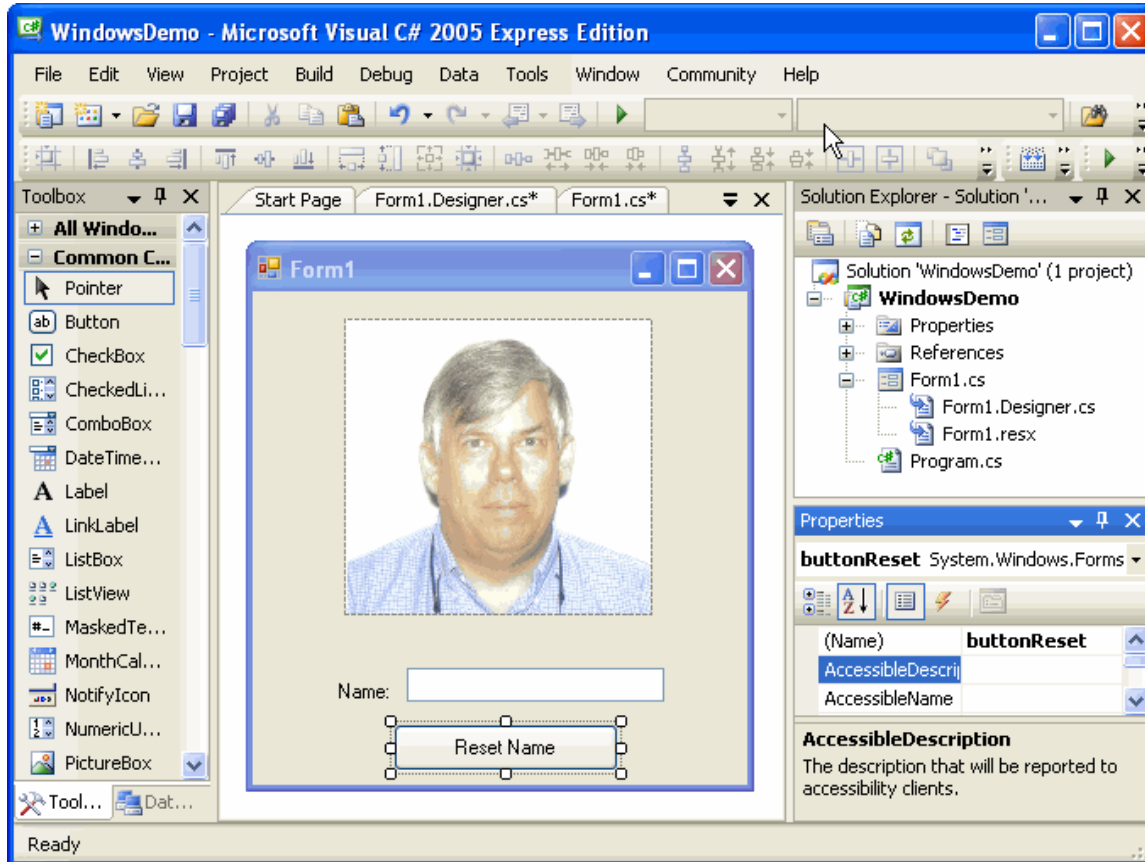
You will notice that event handling takes a rather roundabout route. This is one of the reasons why certain intense applications--such as games--tend to bypass Windows as much as possible, thereby achieving improved performance. But for business applications, the standard routing of events works just fine.

## Handling Events

As previously noted, in a form-based application, events are typically generated by controls and handled by the form on which the control resides. This makes sense because a form is likely to have a lot more information easily accessible to it than a control, such as a button. Because Windows generates events automatically for Window-based objects, what we usually need to do as programmers is to write the handler.

## Creating a Handler for a Button

Continuing with our previous demonstration, we can easily add a button to our form, as shown below:



In this example, we've named the button object **buttonReset**. Let's suppose that we want the button to reset the value of the text box to its original value ("Grandon Gill", from the previous section) every time we click it. That means we'll need to create a handler.

### *Creating Handler Code*

Creating a click handler for a button is trivially easy. Just double click the button in the designer and:

- .NET creates the code for a handler for you, giving you a name that can be changed, and
- Puts you in the code editor right in the middle of the handler.

Since we want to reset the name to the member variable we created in the previous section, the code we might use is highlighted in the following example.

```
namespace WindowsDemo
{
    public partial class Form1 : Form
    {
        private string sName; // Create a member variable
        public Form1()
        {
            // Non-graphic elements can be set before InitializeComponent()
            sName = "Grandon Gill";
            InitializeComponent();
            this.textGrandon.Text = sName;
        }

        private void buttonReset_Click(object sender, EventArgs e)
        {
            // resets text to original value
            textGrandon.Text = sName;
        }
    }
}
```

In this example, all we had to type was:

**textGrandon.Text = sName;**

The designer does the rest.

### ***Handler Arguments***

Two values are passed into every handler. The first is the object that created the event (the button, in this case). We might want this information, for example, in the case that multiple controls use the same handler and we need to find which one generated the event. The second argument contains the actual contents of the message. For a mouse click, this would include the coordinates where the click took place--but each event produces a different set of values in the **EventArgs** argument. This is accomplished by having all the different event argument classes inherit from **EventArgs**. In terms of your code, this means that you may need to typecast the arguments--based on what you know about what causes the event--in order to extract the information you need.

To determine what information the handler arguments supply, put a breakpoint in the handler, then examine the arguments in the debugger when the handler is called.

## Attaching the Handler to the Form

In addition to the code that we edited, the handler also needs to do another important task. Attach the function we want to the button, so Windows can determine where the event will be handled. That occurs in the non-editable portion of the class, as follows:

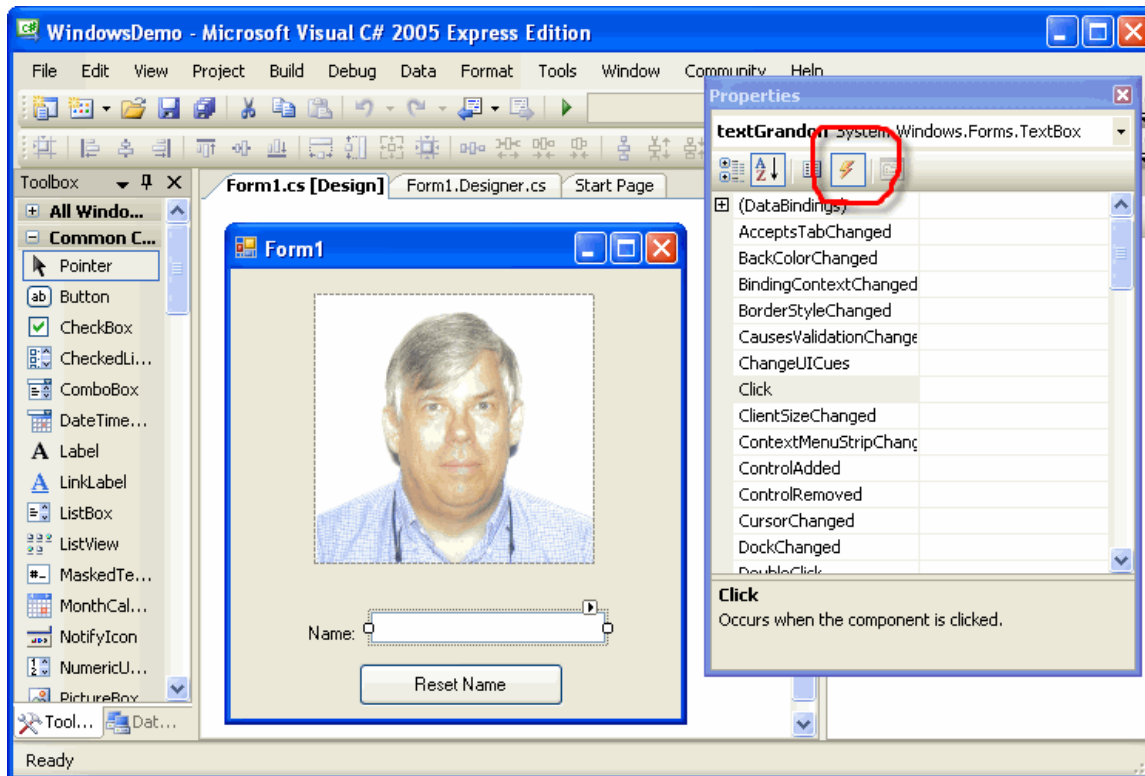
```
//  
// buttonReset  
//  
this.buttonReset.Location = new System.Drawing.Point(83, 255);  
this.buttonReset.Name = "buttonReset";  
this.buttonReset.Size = new System.Drawing.Size(133, 28);  
this.buttonReset.TabIndex = 3;  
this.buttonReset.Text = "Reset Name";  
this.buttonReset.UseVisualStyleBackColor = true;  
this.buttonReset.Click += new System.EventHandler(this.buttonReset_Click);
```

The code above the highlighted area sets the button properties--all of which are straightforward. The highlighted line assigns the function name to the Click property of the button. The fact that += was used (instead of =) suggests--accurately--that it is possible for a given event to be handled in several places.

Once again this code is not to be edited. It can, however, be copied in the event you want to add other handlers to your form from objects that you create while your application is running.

## Handling Other Events

Since buttons naturally respond to clicks--and aren't expected to do much else--it makes sense that double-clicking a button in the editor automatically creates a "Click" handler. There are, however, many other events that user controls (such as buttons, menus, text boxes, etc.) can generate. The process for creating these is nearly identical, except you use the property box, as shown below.



The list of events supported by a control is accessed using the lightning bolt icon (circled). Even a relatively simple control, such as a text box (shown) will have several dozen events that it can generate, such as:

- Mouse clicks
- Control being moved or resized
- Control contents changing
- Control gaining or losing input focus
- etc.

Using these events, you can achieve very fine control over your user interface. Indeed, most programming associated with a form will tend to revolve around event processing.



## **Module 2: Control and Collections**



# if Constructs

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by control in the context of a program
- Define the terms path and branching
- Draw a flowchart of a branch
- Explain what is meant by nesting
- Write **if**, **if...else**, and **if...else if...else** constructs in C#

## Overview

If you are going to write programs that do something of interest, they will have to do more than just execute the same sequence of statements, known as a **path**, every time they start up. The internal structure that allows the same program to execute different paths is called **control**. We have already seen one form of control--writing programs that respond to different types of events. Two other types of control will be examined in this course:

- **Branching:** The program structure that allows different paths of execution to take place depending on conditions. The most widely known branching form is the **if...then...** form--although others exist.
- **Looping:** The program structure that allows blocks of code to be repeated a specified number of times or until some condition is met.

Other control types, most notably *recursion*, exist. These, however, are less commonly used in business programming and, therefore, do not need to be covered in an introductory course.

The control forms we will be examining fall into the language category of **constructs**. A construct is, essentially, a special sequence of keywords, operators and characters that is built into the language itself. We've already seen some constructs in C#, for example (in much simplified form):

- Declaration: **type-name** *variable-name* ;
- *Function: return-type* **function-name** ( *argument-list* ) { *statements* }
- *Class: public class* **class-name** { *declarations-or-functions* }

Like the above, our control forms will require we write our code in a specific way, referred to as **syntax**. It has gotten much easier to write code in the proper syntax now that the editor does things like color coding and autocompletion.


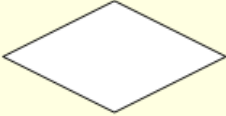


In this section, our focus will be on implementing simple branching, based on the results of a Boolean value (e.g., a test expression or a **bool** variable). We will examine three variations of the

same construct: **if**, **if...else**, and **if...else if...else**. Simple examples of code that employ the **if** construct will be presented.

## Flowcharting

In introducing constructs, we will--from time to time--use diagrams called flowcharts. In the 1960s and 1970s, developers were often required to flowchart entire applications. Over time, however, it was discovered that flowcharts were mainly useful to those learning how to program. For experienced programmers, they were tedious to prepare, huge in size and offered few real benefits for code design. Thus, they fell out of favor, replaced by less detailed diagramming techniques such as UML. Nonetheless, as noted, these charts can be useful in learning to write code and for illustrating constructs.

We will use a tiny fraction of the flowcharting symbols, including those shown in the table below:

	Flow of control
	Decision points (flow comes in and goes out along two or more paths)
	Blocks of code with no control logic
	Flow paths converge or diverge

We'll mainly be explaining what we're doing as we go along, but the basics are as follows:

- Arrows indicate the flow of control. As your code executes, it can be viewed as moving in the direction of the arrows.
- Diamonds indicate a decision point. In general, it will have a question inside, one arrow coming in, and two going out--with labels indicating which outward path is taken for a **true/yes** answer, and which for a **false/no** answer.
- Rectangles indicate individual statements or sequences of statements with no control. They can also be used to summarize self-contained activities, such as function calls.
- Circles indicate paths of flow coming together or (in some cases) breaking apart. They are useful mainly to distinguish actual flow from lines accidentally crossing in the flowchart.

All this will become much clearer when we examine the actual constructs.

## "If" Family of Constructs

The most elementary of the control constructs is the **if** construct. It implements a two-way branch in your code, and comes in three flavors:

- **if**
- **if...else**
- **if...else if...else**

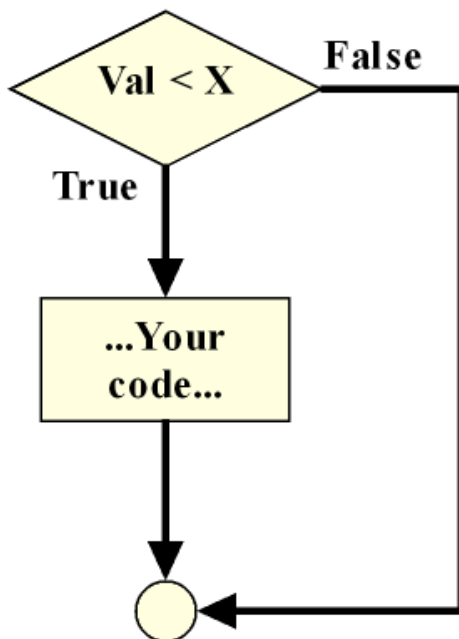
We will now look at each of these.

### if Construct

The **if** construct comes in the following form:

```
if ( Boolean-expression )  
{  
    ...code to be executed if the test is true...  
}
```

In flow chart form, this can be presented as follows:



In essence, the construct simply skips the code in the { } delimited block. In this context, it should also be noted that--within a function--a single statement can be substituted for a block of statements within braces. In general, however, when you are learning it is best to always use braces after your construct.

The highlighted block that follows shows an actual **if** construct in some test code.

```
int X = 10;
Console.WriteLine("Enter a number between 1 and 20: ");
string Num = Console.ReadLine();
int Val = Convert.ToInt32(Num);
if (Val < X)
{
    Console.WriteLine("Your number was less than my number");
}
```

This code:

- Sets the value of an integer variable called X to 10
- Prompts the user for a number
- Reads a line of text to the user and converts it to a number (using the **Convert.ToInt32()** function)
- If the number is less than X (i.e., 10), it displays "Your number was less than my number"

Because the code to be performed based on the test was a single line, the **if** construct could have also been written without braces, as follows:

```
if (Val < X) Console.WriteLine("Your number was less than my number");
```

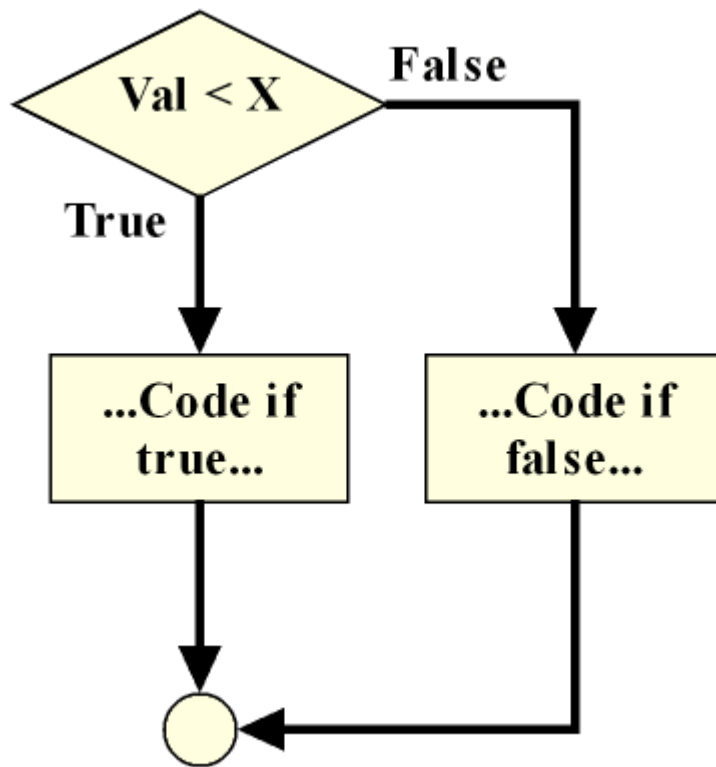
As mentioned earlier, however, when you are learning, it is probably best to use braces consistently to delimit blocks of code--even if they are just a single line.

## **if...else Construct**

Very similar to the **if** construct, the **if...else** version adds a clause to be performed in the event the test is false. It has the following form:

```
if ( Boolean-expression )
{
    ...code to be executed if the test is true...
}
else
{
    ...code to be executed if the test is false...
}
```

The flowchart version of the construct looks as follows:



As the diagram shows, there is now code for both true and false versions of the test--after which the flow of control comes back together (after the construct)

The highlighted block extends our previous example to include an **else** clause:

```
int X = 10;
Console.WriteLine("Enter a number between 1 and 20: ");
string Num = Console.ReadLine();
int Val = Convert.ToInt32(Num);
if (Val < X)
{
    Console.WriteLine("Your number was less than my number");
}
else
{
    Console.WriteLine("Your number was greater than or equal my number");
}
```

The explanation for this code is the same, except that it now displays "Your number was greater than or equal my number"--grammatically incorrect but correct in syntax!

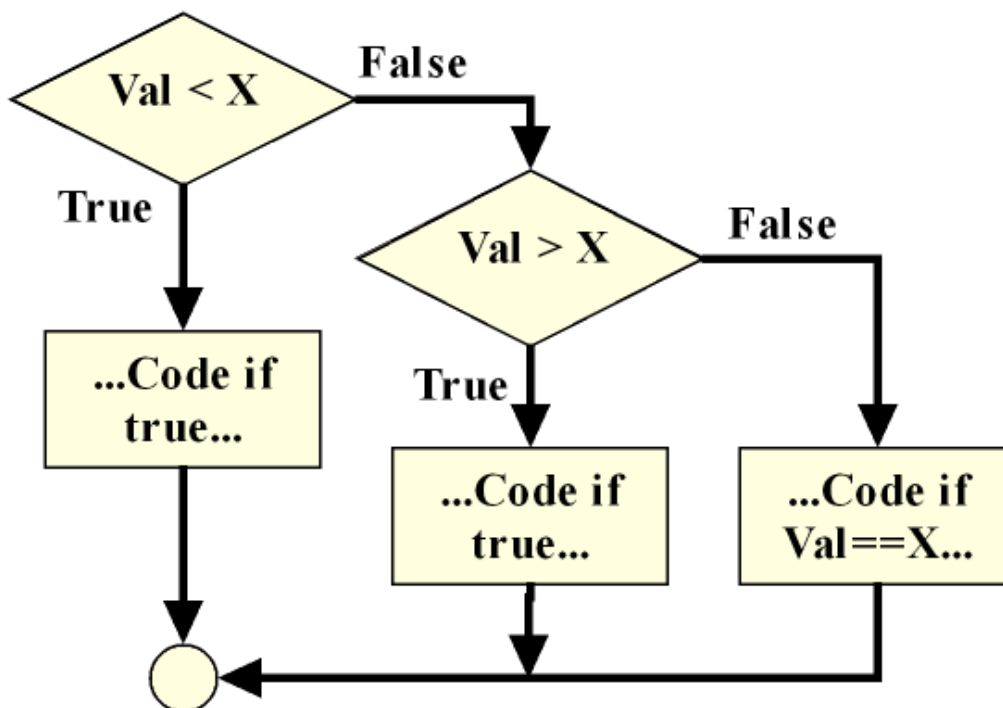
## if...else if...else Construct

Similar to the two previous constructs, the **if...else if...else** version adds a clause to be performed in the event the test is false. It has the following form:

```
if ( Boolean-expression1 )
{
    ...code to be executed if the test is true...
}
else if ( Boolean-expression2 )
{
    ...code to be executed if the test is true...
}
// more else if clauses can be inserted
else
{
    ...This clause is optional, and contains code to be executed if the test is false...
}
```

In this construct, only one block of code will be performed. In other words, the block that follows the first **true** test is performed--even if tests below it are also **true**

The flowchart version of the construct looks as follows:



Note that the rightmost block (the **else** block) implies that  $Val == X$  in our chart, since we've eliminated  $Val < X$  and  $Val > X$ , so equal is all that is left. Extending our code example to correspond to the flowchart:

```
if (Val < X)
{
    Console.WriteLine("Your number was less than my number");
}
else if (Val > X)
{
    Console.WriteLine("Your number was greater than my number");
}
else
{
    Console.WriteLine("Your number was equal to my number");
}
```

This block of code should be self-explanatory by now.

## Nesting Constructs

It is possible--and, indeed, common practice--to place construct code within the code block of another construct. This is referred to as *nesting*. In the case of an if construct, the **if...else if...else** construct can always be replaced by nested versions of the same constructs.

Indenting of code is usually used to indicate nesting has taken place. In the "old days", we used to have to grade students on their use of proper or improper indentation. Today, the punishment the editor will inflict on you if you refuse to nest your code is far worse than anything we could do to you with point deductions.

A code example of the previous **if...else if...else** construct implemented as nested **if...else** constructs is presented below. The area highlighted is the nested portion. The flowcharts are the same!

```
if (Val < X)
{
    Console.WriteLine("Your number was less than my number");
}
else
{
    if (Val > X)
    {
        Console.WriteLine("Your number was greater than my number");
    }
    else
    {
        Console.WriteLine("Your number was equal to my number");
    }
}
}
```

# Case Construct

## Learning Objectives

Upon completing this reading, you should be able to:

- Describe situations where a multi-way branch is appropriate
- Distinguish between an **if** construct and a **switch...case** construct
- Write a C# **switch...case** construct using integer switching
- Write a C# **switch...case** construct using string switching

## Overview

The C# **switch...case** construct, also commonly referred to as the case construct, implements a branch between an arbitrary number of options, known as a multi-way branch. Although, any multi-way branch can be implemented as a series of two-way (if) constructs, the former can be much more efficient and easier to both read and write.

Multi-way branches are implemented by defining a switching expression--evaluating to an **integer** or **string** in our examples--and identifying a set of individual values (constants or literals) that it can take on. These become the individual cases that the construct uses to identify where control flow should go. In some languages (e.g., C++), flow from one case block can fall into another (known as case fall-through). In C#, however, the code block associated with each switching constant is independent.

Case constructs prove to be incredibly useful in programming, since we often find ourselves needing to direct our flow based on a set of values known in advance. The reading will present some examples from the Assignment 2 code.

## Multi-Way Branching

Once you have the ability to implement a two-way branch, you can implement multi-way branches through nesting of the two-way constructs (or through as series of **else if** constructs in C#). The problem is that it can lead to nearly unreadable code. For example, suppose we were implementing a menu with options 1 through 7 and wanted to direct control based on what option was pressed. Implemented using **else if** constructs, it would look something like:

```

if (option==1)
{
    ...option 1 code...
}
else if (option==2)
{
    ...option 2 code...
}
else if (option==3)
{
    ...option 3 code...
}
else if (option==4)
{
    ...option 4 code...
}
else if (option==5)
{
    ...option 5 code...
}
else if (option==6)
{
    ...option 6 code...
}
else
{
    ...option 7 code...
}

```

Not only does this look nasty, it's also computationally wasteful. If the user chooses option 7, six tests need to be performed in order to get to the code.

The problem can also be illustrated with a qualitative example. Suppose a restaurant accepts cash, credit cards, gift cards, traveler's checks and personal checks, and that a different procedure exists for each type of payment. When the server looks at your tray with the payment on it, does he or she say:

- *Is this cash?* No
- *Is this a credit card?* No
- *Is this a gift card?* No
- *Is this a traveler's check?* No
- *Is this a personal check?* Yes--proceed to processing the check...

Although there are probably some psychological syndromes that cause an individual to process information in this manner, it is certainly not the norm. The server looks at the tray, sees what's

on it, then proceeds to the appropriate process. That's what we want our code to do for a multi-way branch.

## switch...case Construct

The C# construct for implementing multi-way branching is the **switch...case** construct, commonly referred to as the case construct. The form of the construct is as follows:

```
switch (expression)
{
    case Const1:
        ...your code...
        break; // return also allowed
    case Const2:
        ...your code...
        break; // return also allowed
    case Const3:
        ...etc...
    default: // optional, like else
        ...your code...
        break;
}
```

The *expression* can be a variable or more complex expression (such as a function that returns a value). The individual case values must be constant values, such as integers (e.g., 2,5,6), characters (e.g., 'a', 'b', 'c') or strings (e.g., "red", "White", "Blue")--they cannot be variables or expressions.

Each case must end with a **break** keyword (although a **return** is also permitted, as is the hated **goto** statement, which will not be mentioned again). This prevents *case fall through*, where statements from one case pass through into another--something that C# does not allow (but other languages, such as C++, do).

Although C# does not allow case fall through, you can have more than one label for a given case. For example, if you were switching on a character, you might want to have two labels--one for uppercase and one for lowercase. e.g.,

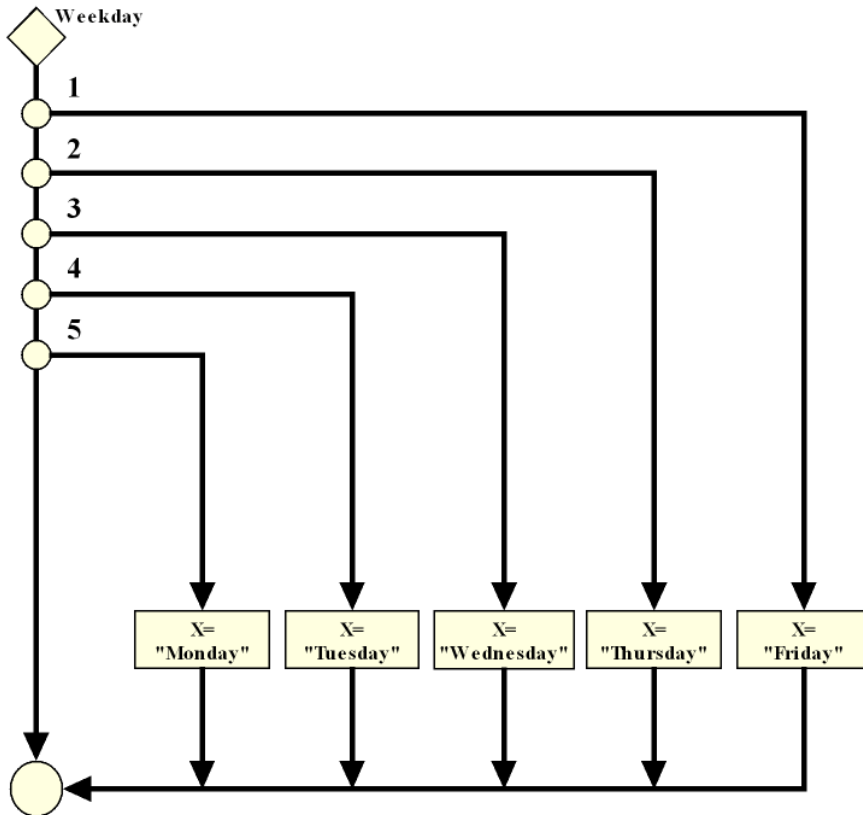
```

switch (myChar)
{
case 'A':
case 'a':
    ...your code...
    break; // return also allowed
case 'B':
case 'b':
    ...your code...
    break; // return also allowed
case 'C':
case 'c':
    ...etc...
default: // optional, like else
    ...your code...
    break;
}

```

## Flowchart Case Structure

Conceptually, a case construct can be flowcharted as a decision point with a series of branches coming out of it, as represented below:



The absence of case fall through is what makes each of the five blocks presented above independent of the other blocks.

## Integer Switching

One of the most common forms of case construct uses integer values to switch between options. For example, the previous flow chart, represented as code, might appear as follows:

```
string GetName (int nDay)
{
    string s;
    switch (nDay)
    {
        case 1:
            s = "Monday";
            break;
        case 2:
            s = "Tuesday";
            break;
        case 3:
            s="Wednesday";
            break;
        case 4:
            s = "Thursday";
            break;
        case 5:
            s = "Friday";
            break;
        default:
            s = "Illegal Day!";
            break;
    }
    return s;
}
```

This function would take an integer value from 1 to 5 and return the corresponding day of the week name as a string. (FYI, If we wanted to represent the default in the flowchart, it might be added as a box on the main line, after all the options).

Although it might be good programming practice to put integer case labels in some sort of order, there is nothing about the construct that requires it. Thus, for example, case 5 may precede case 2 with no impact on how the program runs.

## Switching on Strings

The values of individual cases can also be strings. If, for example, we wanted to create an inverse for the previous function, it could be written as follows:

```

int GetNumber(string sDay)
{
    int nDay;
    switch (sDay.ToUpper())
    {
        case "MONDAY":
        case "MON":
            nDay = 1;
            break;
        case "TUESDAY":
        case "TUE":
            nDay = 2;
            break;
        case "WEDNESDAY":
        case "WED":
            nDay = 3;
            break;
        case "THURSDAY":
        case "THU":
            nDay = 4;
            break;
        case "FRIDAY":
        case "FRI":
            nDay = 5;
            break;
        default:
            nDay = 0;
            break;
    }
    return nDay;
}

```

A couple of comments on this code:

- First, since C# is case sensitive, instead of taking our string argument directly (sDay), we make it upper case as part of the **switch** statement (i.e., **sDay.ToUpper()**, where **ToUpper()** is supported by the **string** class).
- Second, this function was designed to respond to 3-letter abbreviations, as well as full names, using multiple labels per branch.

Consider how much easier this is to look at (and write) than the corresponding set of if statements. In the next section, we'll introduce enumerations, which can be used to add even greater representational power to the case construct.

# Enumerations

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain the purpose of **enum** declarations
- Create a simple integer **enum**
- Use **enum** variables in a case construct
- Explain when an **enum** value may need to be typecast

## Overview

The principal challenge of programming *is not* learning to program--that's the easy part. The real challenge is managing the complexity of programs that are ever growing in size. One way to handle that complexity is through intelligent use of names. We've already tried to demonstrate this in the way we name objects and their members. But what about the constants that we use in our programs?

An enumeration, created with the **enum** keyword in C#, allows you to create a set of defined constants--suitable, for example, for use in a **case** statement. What is particularly useful about **enum** sets is that they can be used as a sort of restricted value integer type. Thus, if you were to declare an enumeration called **Months**, with 12 allowable values, any variable declared to be of type **Months** would only be allowed to take on those 12 values.

In the .NET framework, thousands of different enumerations are declared. For example, if you want to interpret the return value of a dialog box, you must use the **DialogResult** enumeration. With the autocompletion feature in the editor, enumerations have become particularly useful--since you no longer have to go into a manual to find all the allowable values for a particular enumeration. Just type its name, followed by a period, and the list presents itself.

In this section, we examine how to construct enumerations and demonstrate their use code, using Assignment 2 as an example.

## **enum Construct in C#**

In C#, the **enum** construct provides a convenient way of naming and accessing sets of defined integer constants. In addition to providing a means of naming many such constants, an **enum** can be treated as if it were an integer data type with restricted values--meaning you can declare variables using the **enum** name as their type.

## **enum Declarations**

The **enum** declaration, which occurs within a **class** definition, consists of the following:

- The **enum** keyword (preceded by **public**, if public access is desired)
- The name for the enumeration
- A { } delimited list of the names being defined, with or without associated values.

This is most easily demonstrated by example. In creating the **MainWindow** object in Assignment 2, for example, we need to keep track of the 4 players. In the example code, the following enum is declared:

```
public enum Players { Grandon, Clare, Tommy, Jonathan };
```

Since no values were specified, **Grandon** would have the value 0, and each element would be one greater than the previous element (e.g., **Jonathan** would be 3).

In the code we use to implement transcripts later in the assignment, we need to keep track of ISM courses. To do so, we declare the following enumeration:

```
public enum Ism
{
    ism3232=3232, ism3113=3113, ism4212=4212, ism4220=4220, ism4300=4300,
    ism4234=4234
};
```

In this example, we've specified the values for individual constants--namely **ism3232** has the value 3232, **ism4234** has the value 4234, and so forth. When specified values and non-specified values are mixed, element values following a specified value go up by 1 from the previous element (just as in the purely unspecified case).

## Using Enumerations

Once declared, we can use enumerated constants just as if they were constants, and we can also use the enumeration as if it were a variable type. Specifically:

- Enumerations can be treated like data types in declarations—meaning only allowed values can be assigned, e.g.,

```
Players p;
```

- To access an individual element in the enumeration, the form *EnumName* . **ItemName** is used, e.g.,

```
Players p=Players.Grandon;
```

- Enumerated elements can be typecast. Most commonly, this is used to move back and forth between enumerated elements and integers, e.g.,

```
int PlayerVal=(int)Players.Grandon;
```

### *Example 1: Selecting a Student*

Once again, this is most easily illustrated by example from Assignment 2. In the `SelectStudent()` function, shown below, a value from the `Player` enumeration (shown above, consisting of the list of player names) is passed in to the function. That value is then used in a case construct that serves to bring up the appropriate student's data. The basic idea here is fairly similar to what was done in Assignment 1, except that instead of tying each person to his or her own event, we have one function that handles all four persons.

```
public void SelectStudent(Players val)
{
    nSelected = val;
    pictureGrandon.Hide();
    pictureClare.Hide();
    pictureTommy.Hide();
    pictureJonathan.Hide();
    switch (val)
    {
        case Players.Grandon:
            {
                sCurrent = sGrandon;
                pictureGrandon.Show();
                break;
            }
        case Players.Clare:
            {
                sCurrent = sClare;
                pictureClare.Show();
                break;
            }
        case Players.Tommy:
            {
                sCurrent = sTommy;
                pictureTommy.Show();
                break;
            }
        case Players.Jonathan:
            {
                sCurrent = sJonathan;
                pictureJonathan.Show();
                break;
            }
    }
    textName.Text = sCurrent.Name;
    textGPA.Text = sCurrent.GPA.ToString();
    textWealth.Text = sCurrent.WealthPoints.ToString();
    textPride.Text = sCurrent.PridePoints.ToString();
    listPlayers.SelectedIndex = (int)val;
}
```

## *Example 2: Using Typecasting*

Our second example, drawn from the **Transcript** class, implements a function that determines if a particular course--passed in by course number--meets the required prerequisites. Because course numbers are stored as integer values within the **Course** class (an implementation decision), we need to typecast our enumerated values before we can use them. Even so, the basic logic of the function should be understandable, without necessarily understanding the mechanics of how each function works. Once again, this is a matter of making good naming choices and using enumerations whenever possible.

```
public bool MeetsPrerequisites(int c)
{
    bool bMet;
    switch (c)
    {
        case (int) Ism.ism3113:
            bMet= (FindPassing((int) Ism.ism3232) >= 0 ||
                FindEnrolled((int) Ism.ism3232) != null);
            break;
        case (int) Ism.ism3232:
            bMet=true;
            break;
        case (int) Ism.ism4234:
            bMet= (FindPassing((int) Ism.ism3232)>=0);
            break;
        case (int) Ism.ism4220:
        case (int) Ism.ism4212:
            bMet=(FindPassing((int) Ism.ism3113)>=0);
            break;
        case (int) Ism.ism4300:
            bMet= (FindPassing((int) Ism.ism4220)>=0 &&
                FindPassing((int) Ism.ism4212)>=0 &&
                FindPassing((int) Ism.ism3232)>=0);
            break;
        default:
            bMet=true;
            break;
    }
    return bMet;
}
```

# while Loops

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by a loop
- Describe some of the benefits of looping
- Draw a flowchart of a basic while loop
- Write C# code that implements a while loop
- Describe some of the most likely causes of infinite loops

## Overview

Looping involves establishing a program flow that executes the same block of code multiple times in succession. There are numerous programming situations that call for looping--e.g., applying the same code to all the elements in a collection (e.g., computing the grade of each student in a class), performing mathematical algorithms (e.g., computing the balance left on a mortgage at a given period in time), etc. The execution of one pass through a loop construct within a program is often referred to as an **iteration**; programming techniques based on looping are referred to as **iterative**.

C# provides a number of constructs that can be employed for looping. The simplest of these is the **while** loop, which performs a test prior to each iteration in order to determine if the loop should process. In this reading, we examine how to construct **while** loops, and present some code examples.

## The while Loop

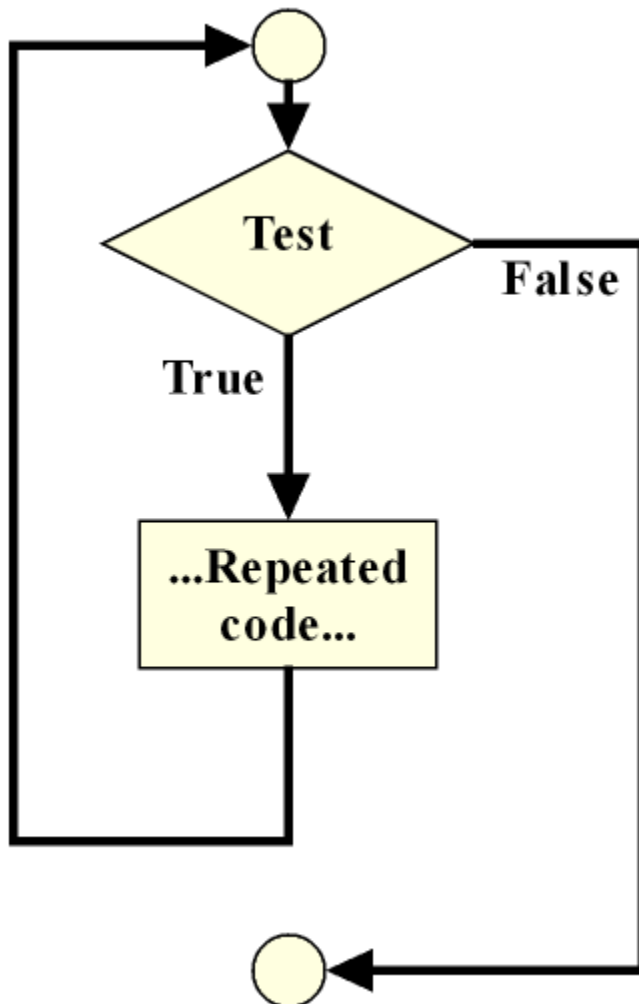
The **while** loop is the simplest of the looping constructs provided by C#. It takes the following form:

```
while (test)
{
    ... code to be repeated...
}
```

In constructing a while loop, the test can be any Boolean expression (e.g.,  $X < 10$  or a **bool** variable). The test is performed before any execution of the code to be repeated--indeed, if the test is false when the loop is encountered, the entire loop is skipped. It is also important to realize that the test is not some psychic construct that continually inspects what's going on in the code inside the loop. Instead, it is performed prior to the start of each iteration. If conditions change within the iteration, these are not tested until the current iteration ends and the next one is about to begin.

## Flowchart of while Loop

The structure of the **while** loop can be illustrated with a flowchart, as shown below. Although similar to the **if** construct, it is different in that the "true" branch returns to the test and the process repeats.



An important point to emphasize about while loops is that something must change within the loop that causes the test to change (or break out of the loop in some other way, such as a **return** or **break** statement, as will be discussed in a later section, on **for** loops). If the test remains the same, the program will continue looping--a condition known as an infinite loop.

## while Loop Example

An example of a **while** loop should clarify the concept. The code below illustrates a simple **while** loop, constructed within a console program:

```

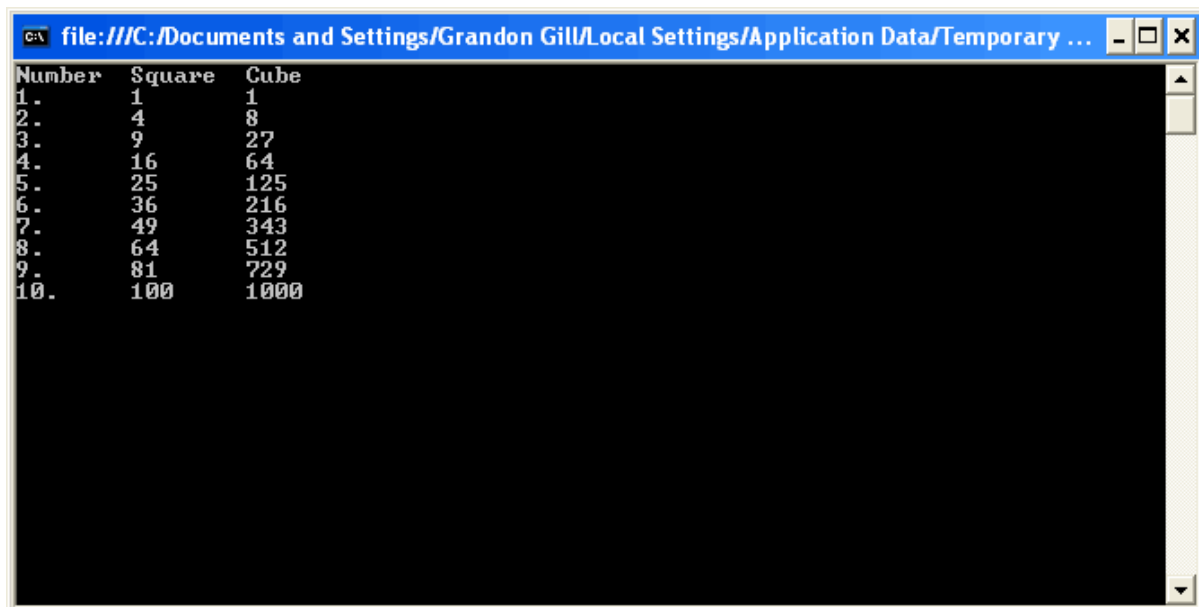
void PrintLoop()
{
    Console.WriteLine("Number\tSquare\tCube");
    int ctr = 1;
    while (ctr <= 10)
    {
        Console.WriteLine("{0}.\t{1}\t{2}", ctr, ctr * ctr, ctr * ctr * ctr);
        ctr++;
    }
}

```

This loop is used to create a table of numbers, their squares and their cubes, from 1 to 10. It works as follows:

- The table headings are displayed (the `\t` character causes a **tab** character to be displayed)
- A counter variable is initialized to 1 (the top highlight)
- We enter a while loop that tests to ensure the counter is `<= 10`
- We display the numbers, squares and cubes (the `{0}`, `{1}` and `{2}` in the string argument identify where the three argument values are to be placed)
- We increment the counter (bottom highlight; recall that `ctr++` is equivalent to writing `ctr=ctr+1`)
- When the `ctr` value reaches 11 (after the 10th pass, since we started at 1), the while test fails and the function ends.

The output produced from running this function is as follows:



```

c:\ file:///C:/Documents and Settings/Grandon Gill/Local Settings/Application Data/Temporary ...
Number  Square  Cube
1.      1       1
2.      4       8
3.      9       27
4.      16      64
5.      25      125
6.      36      216
7.      49      343
8.      64      512
9.      81      729
10.     100     1000

```

# for Loops

## Learning Objectives

Upon completing this reading, you should be able to:

- Identify the syntax of a for loop
- Use for loops and while loops interchangeably
- Describe how **break** and **continue** statements change the flow of control in a loop
- Explain how the concept of *scope* impacts variables in **for** loops

## Overview

A **for** loop is simply a minor variant on the **while** loop. It adds *initialization* and *incrementation* blocks to the **while** construct, making the loop somewhat more self-contained. This proves to be a sufficiently useful variation, however, that many programmers use **for** loops far more often than **while** loops.

In addition to viewing examples of **for** loops, this section examines certain statements--**break** and **continue**--that can be used to interrupt the flow of the loop. We also consider the concept of variable scope--how long variables within code blocks remain available to a program. This has particular implications for **for** loops.

## Initialization and Incrementation Blocks

Consider the while loop code example presented in the previous section:

```
void PrintLoop()  
{  
    Console.WriteLine("Number\tSquare\tCube");  
    int ctr = 1;  
    while (ctr <= 10)  
    {  
        Console.WriteLine("{0}.\t{1}\t{2}", ctr, ctr * ctr, ctr * ctr * ctr);  
        ctr++;  
    }  
}
```

The two highlighted lines perform the key functions:

- **Top:** *Initializes* the counter to keep track of where we are in the loop.
- **Bottom:** *Increments* the counter, so it will eventually break us out of the loop

Although not every loop needs to perform such *initialization* and *incrementation* activities, a surprisingly large number do. The for loop incorporates spaces for these activities. This can

make the code more compact (and reduce the likelihood that the programmer will forget to put them in).

## C# for Loop

The **for** loop in C# is constructed as follows:

```
for (initialization-code; test; increment-code)  
{  
    ...code to be repeated...  
}
```

To interpret this:

- The *initialization-code* is a series of statements separated by **commas** (since the semicolon is used to separate this code from the *test*). It can be empty, although that would be somewhat unusual.
- The *test* is any expression that returns a Boolean value (just like a **while** loop).
- The *increment-code* is a series of statements, separated by commas. It can also be empty.

Because both the initialization and increment blocks can be empty, it is possible to write the construct:

```
for(;test;) {...code...}
```

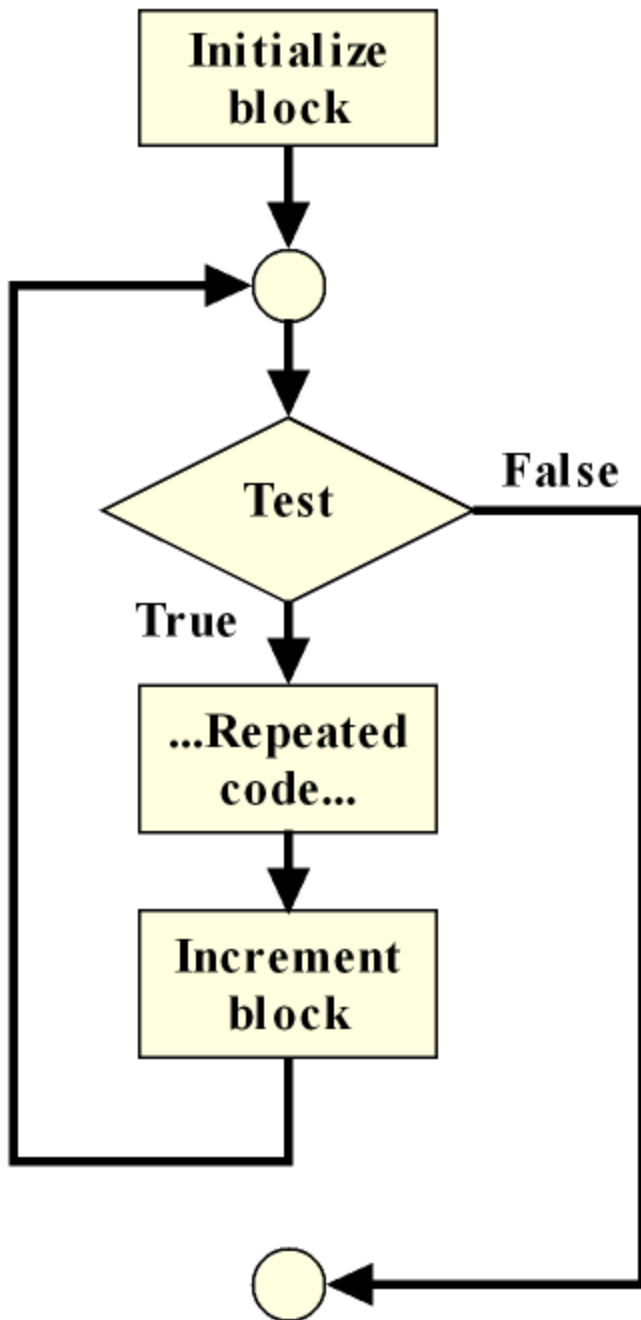
This would be identical to:

```
while(test) {...code...}
```

This would also be a good example of a time when a **while** loop would be preferable to a **for** loop.

## Flowchart of a for Loop

The flowchart of a **for** loop is similar to that of a while loop, except that two blocks (initialize and increment) are embedded within the construct, as shown below:



### for vs. while Loops

In the example below, the **while** loop code already discussed is compared with the **for** loop code that does exactly the same thing:

```

void PrintLoop()
{
    Console.WriteLine("Number\tSquare\tCube");
    int ctr = 1;
    while (ctr <= 10)
    {
        Console.WriteLine("{0}.\t{1}\t{2}", ctr, ctr * ctr, ctr * ctr * ctr);
        ctr++;
    }
}
void PrintForLoop()
{
    Console.WriteLine("Number\tSquare\tCube");
    for (int ctr = 0; ctr <= 10; ctr++ )
    {
        Console.WriteLine("{0}.\t{1}\t{2}", ctr, ctr * ctr, ctr * ctr * ctr);
    }
}

```

The biggest practical difference is that the **for** loop (bottom) is a bit more compact. It is also less likely that you'll forget to put in your initialization or iteration steps when you write a for loop.

## Variable Scope

In the previous example, the **for** loop presented began:

```
for(int ctr = 0; ctr <= 10; ctr ++) { ...etc...}
```

One reasonable question to ask is: can we use the value of counter after the loop ends?

The answer is no, for reasons of variable scope. Basically the rules for scoping are as follows (with for loops being a special case):

- Any variable declared within a code block (i.e., within { and } braces) can only be used from the point it is defined until the end of that code block. This means, for example, that when a function returns, all the variables declared within that function disappear (although not necessarily the objects they refer to). It also means that if you are nesting constructs, variables declared in the inner code blocks cannot be referenced in the outer code blocks.
- Any variable declared in the initializer block of a **for** loop remains in scope until the **for** loop is complete.

You can't describe these scoping rules as good or bad, you just need to be aware of them. For example:

- if you want to use a counter variable after its for loop ends, just declare it before the for loop (just as we did with the while loop). For example:

```
int ctr;

for(ctr = 0; ctr <= 10; ctr ++) { ...etc...}
```

```
// ctr can be used here
```

- Since for loop counters go out of scope once the loop ends, you can keep using the same name over and over again with a function that has a series of for loops, such as:

```
for(int ctr = 0; ctr <= 10; ctr ++) { ...etc...}
```

```
for(int ctr = 0; ctr <= 20; ctr ++) { ...etc...}
```

```
for(int ctr = 0; ctr <= 30; ctr ++) { ...etc...}
```

They won't interfere with each other since they are no longer defined when each loop ends.

## break And continue

The **break** and **continue** statements are special statements that change the flow of control within a loop. Specifically:

- **break**: Ends the loop immediately and moves the control path to the next statement after the loop.
- **continue**: Takes control to the iteration block (in a **for** loop), skipping the remaining steps in the block of repeated statements. In a **while** loop, it sends control up to the test.

The function below illustrates these two statements:

```

void RunningTotal()
{
    int nTotal=0; // We want to use Total after loop
    int Val=0;
    for (bool KeepGoing = true; KeepGoing; nTotal += Val)
    {
        Console.WriteLine("Enter an integer (Quit or Exit to end) ");
        string sVal = Console.ReadLine();
        if (sVal.ToUpper() == "QUIT") break; // One way to end
        else if (sVal.ToUpper() == "EXIT") // Another way to end
        {
            KeepGoing = false;
            continue;
        }
        Val = Convert.ToInt32(sVal);
    }
    Console.WriteLine("Your total is {0}", nTotal);
}

```

The function takes user input of numbers and computes a running total. It ends when the user types "Exit" or "Quit". The explanation is as follows:

- We initialize nTotal (that will keep a running total) and Val (which will hold the value the user types in).
- We enter a **for** loop that begins by initializing a variable, **KeepGoing**, to **true** (highlighted in yellow). Notice that in this loop, we're not using a counter to end the loop, but rather the value of a **bool** variable.
- The program prompts the user to enter an integer, then reads it into **sVal**.
- If (sVal.ToUpper() == "QUIT")--highlighted in green--we break out of the loop using a **break** statement. Control passes to the statement at the bottom, which displays the total.
- If (sVal.ToUpper() == "EXIT") we set KeepGoing to **false**, then **continue**. This actually produces an error in the computation, since control passes to the **nTotal += Val** statement, which would use the previously computed value and add it--for a second time--to the running total.
- When the loop ends, the running total value is output.

The output from running this function--exiting in the manner that does not produce an error--is presented below.

```
file:///C:/Documents and Settings/Grandon Gill/Local Settings/Application Data/Temporary ...
Enter an integer <Quit or Exit to end> 1
Enter an integer <Quit or Exit to end> 2
Enter an integer <Quit or Exit to end> 3
Enter an integer <Quit or Exit to end> 4
Enter an integer <Quit or Exit to end> quit
Your total is 10
```

# Dialog Boxes and Forms

## Learning Objectives

Upon completing this reading, you should be able to:

- Describe the differences in behavior between a number of window objects, including the main window of an application, child windows, dialog boxes and message boxes
- Attach dialog result codes to buttons
- Invoke a dialog box to gather information from the user
- Explain how standard dialog boxes differ from user-created dialog boxes
- Interpret dialog and message box return codes

## Overview

Window objects provide the principal interface between a user and underlying program code for most applications. A particularly common style of window is the **modal dialog box**, used to gather information from the user. These dialog boxes are *modal* in that they prevent the user from accessing other parts of the application until they are dismissed (by answering the questions or canceling). In this fashion, they differ from other child windows, which allow users to move to other application windows at will.

This section focuses on the creation of dialog boxes and their close cousin, message boxes. We begin by exploring how dialog boxes differ from other child windows (i.e., how to make them modal). We then consider how dialog boxes differ in design from other windows. Finally, we examine how quick yes/no questions to users can be answered with message boxes.

## Forms of Windows

Windows come in many different shapes and exhibit many different types of behavior. In this course we will focus on five different types of windows:

1. Main windows of an application
2. Child windows
3. Dialog boxes
4. Message boxes
5. Control windows (e.g., buttons, menus) on a form

Since we've already worked with the last category of windows in our earlier assignments and readings, we'll focus on the first four. Moreover, the first three differ mainly in how they are invoked, while the fourth is just a special class of pre-made dialog box.

## *Main Windows*

The main window of an application is, for the most part, just like any other window--how it differs is the manner in which it is invoked. Specifically, for our applications, the main window will be created and displayed in the **Main()** function, in the **Program.cs** file. The code generated is:

```
Application.Run(new MainWindow());
```

where *MainWindow()* is whatever name you gave your form (**Form1** is the default). Creating a window in this way attaches the window to a new application object, and only one window is attached this way. As a result, closing the window causes the application to shut down (automatically closing any child windows that have been created in the process).

## *Child Windows*

Child windows can be forms, just like the main window. They differ, however, in their association with the application object and how they are created. Specifically, closing a child window does not automatically end an application (as long as the main window is still open). Furthermore, the creation of a child window is usually a two step process, with creating and displaying the window being distinct activities. Specifically:

```
WindowDemo wnd=new WindowDemo();
```

```
wnd.Show();
```

where *WindowDemo()* is the class name of the window being created. Multiple child windows can be open at the same time, and users can move between them.

## *Dialog Boxes*

Dialog boxes are normally forms, just like the main window and child windows. They differ, however, in the function used to display them and in how they behave once displayed. Specifically, although they use a two step create/display process--like a child window--they are displayed using **ShowDialog()** instead of **Show()**, as shown here:

```
WindowDemo wnd=new WindowDemo();
```

```
wnd.ShowDialog();
```

where *WindowDemo()* is the class name of the dialog box being created. Dialog boxes also differ in three other respects:

- They are usually application-modal, meaning that other application windows cannot be accessed until the dialog has been dismissed, usually by pressing a button on the dialog.

- The **ShowDialog()** function returns a result, enumerated in **DialogResult**, that usually signifies what button was used to end the dialog.
- Dialog box objects stick around and are available for access even after they are closed. This can be important, since we often need to extract data from the dialog to determine what data the user input.

## *Message Boxes*

Message boxes are just a standardized form of dialog box--that are customizable mainly with respect to the text they display--that are generally used to pop up messages to the user or to prompt for quick responses (e.g., Yes/No, Ignore/Retry/Fail). They are invoked by calling the static **MessageBox.Show()** function, e.g.,

```
MessageBox.Show("This pops up a message box!");
```

The **MessageBox.Show()** function has numerous options regarding the buttons that appear. It also returns a **DialogResult** value, so the user's response can be obtained.

## **Dialog Boxes**

Once you understand the differences between the window forms, dialog boxes are pretty straightforward to create and use.

## **Invoking a Dialog Box**

A typical call to a user-created dialog box class object is presented below:

```
MultipleChoice dlg = new MultipleChoice(q);
if (dlg.ShowDialog() == DialogResult.OK)
{
    if (dlg.Correct) Score = Score + PointsPerQuestion;
}
else return false;
```

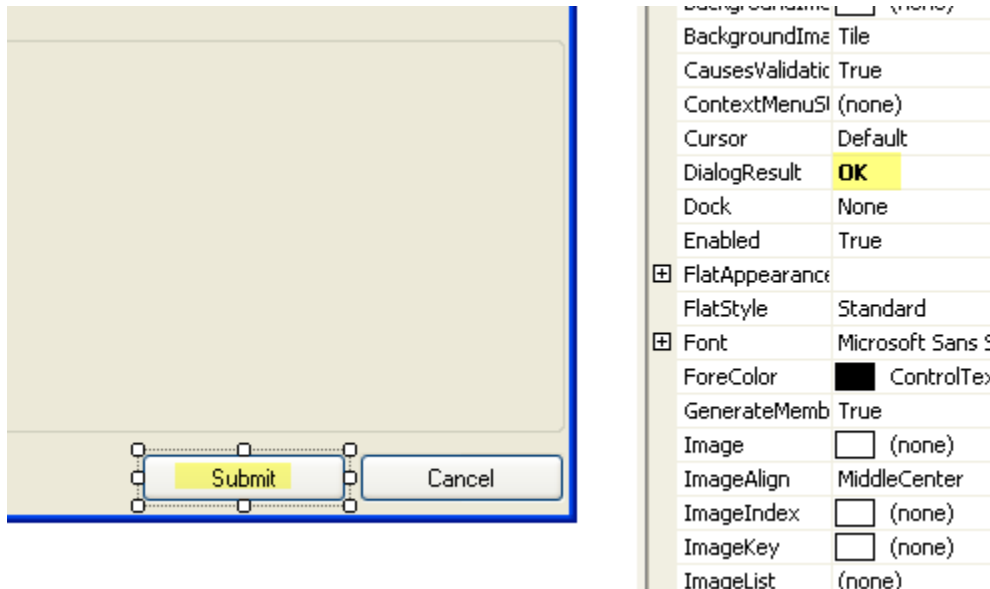
The explanation of the code is as follows:

- In the first line, the dialog box object--a multiple choice question form from Assignment 2--is created. (In this case, the constructor was defined to require a multiple choice question, previously assigned to **q**, as an argument).
- In the second line, we test to see that the user pressed OK to end the dialog (the other common option being **DialogResult.Cancel**).
- If so, within the code block, we extract the **Correct** property from the dialog (which returns **true** if the user selected an answer that was marked correct when the dialog was initialized) and--if the answer was correct--we add the question score to the cumulative score for the test.

Other **DialogResult** enumeration values you can test for include: **Abort**, **Cancel**, **Retry**, **Ignore**, **No**, **None**, **Yes**.

## Creating a Dialog Box

You create a dialog box like any other window. There are just a few key things you need to remember. First, you need to attach **DialogResult** values to some of your buttons, as shown below--where the *Submit* button is set to send **DialogResult.OK** when clicked (highlighted).



In addition to sending the specified result, buttons with the **DialogResult** property set also close the dialog. (If the dialog is closed in some other way, the **DialogResult.Cancel** value is normally returned). The other properties you can set for a button include the same values mentioned previously: **Abort**, **Cancel**, **Retry**, **Ignore**, **No**, **None**, **Yes**.

The other thing to remember when creating a dialog box is to implement properties to get data into and out of the dialog. Your main window and child windows frequently interact directly with the application's data; unless undo capabilities have been implemented, any changes you make have to be reversed by you. Dialogs, on the other hand, usually have a Cancel capability. This means that you are best off holding dialog results in temporary variables--accessible through dialog properties--that can be assigned to application data after you've verified that OK has been pressed.

## Standard Dialog Boxes

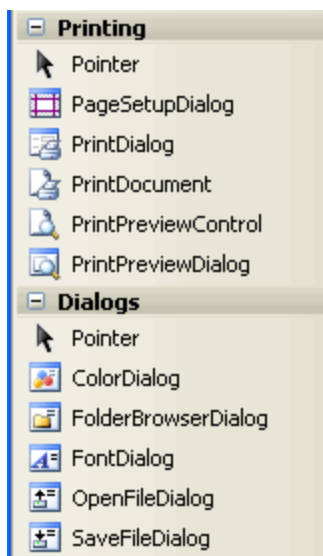
.NET also provides a number of standard dialog boxes. These include:

- Selecting a file for saving
- Selecting a file for opening
- Choosing a color

- Choosing a font
- Establishing printer settings
- Choosing a printer
- Launching a print preview display

Although these behave in the same way as the dialogs you design, they are normally dragged on to a form from the Toolbox (see below). This action has a number of consequences:

1. A member variable is created and placed in the grey non-displaying controls area at the bottom of the form .
2. The code to create the new dialog object and assign it to the member variable is placed in the form's InitializeComponent().



The practical result of having a member variable is that the dialog box can be launched without the object-creation step, as shown below in code that would be typical for a *File/Open* menu handle (*openFile* is the name chosen by the programmer for the **OpenFileDialog** member variable):

```
private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (this.openFile.ShowDialog() == DialogResult.OK)
    {
        FileName = openFile.FileName;
    }
}
```

## Message Boxes

Message boxes are a convenient form of modal dialog box that you can pop up at any time, usually to tell the user something or to get back a quick response. They are invoked by a call to the static **MessageBox.Show()** function, which has numerous overloads. Two of the most commonly used are presented in the code below:

```

static void TestMessageBox ()
{
    switch (MessageBox.Show("Click a button", "Message box test",
        MessageBoxButtons.YesNoCancel))
    {
        case DialogResult.Yes:
            MessageBox.Show("You pressed Yes!");
            break;
        case DialogResult.No:
            MessageBox.Show("You pressed No!");
            break;
        case DialogResult.Cancel:
            MessageBox.Show("You pressed Cancel!");
            break;
    }
}

```

The top call, a 3 argument version, allows the programmer to specify:

1. The message inside the window (i.e., "Click a Button")
2. The message box's title bar (i.e., "Message box test")
3. The buttons to be displayed. The **MessageBoxButtons** enumeration has numerous combinations of possible buttons listed as constants. In this case, the box would show the Yes, No and Cancel buttons.

The function takes the return value of the call--which is a **DialogResult** enumerated value--and uses it in a case construct. Each case, in turn, calls the single argument version that pops up a message with an OK button, used to alert the user of something.

# Arrays

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by the term collection
- Describe the basic structure of an array
- Access elements within an array object using array notation
- Create an array object

## Overview

The big picture: A typical computer has about a gigabyte of RAM installed. If a programmer had to come up with names for every variable in memory, he or she would spend an entire lifetime creating names. Obviously, then, we can't be required to name every object we create. But how, then, do we keep track of them? The answer is, we use collections.

A collection class is just a class that can be used to hold multiple objects, without requiring that each be named separately. Every such class needs two capabilities:

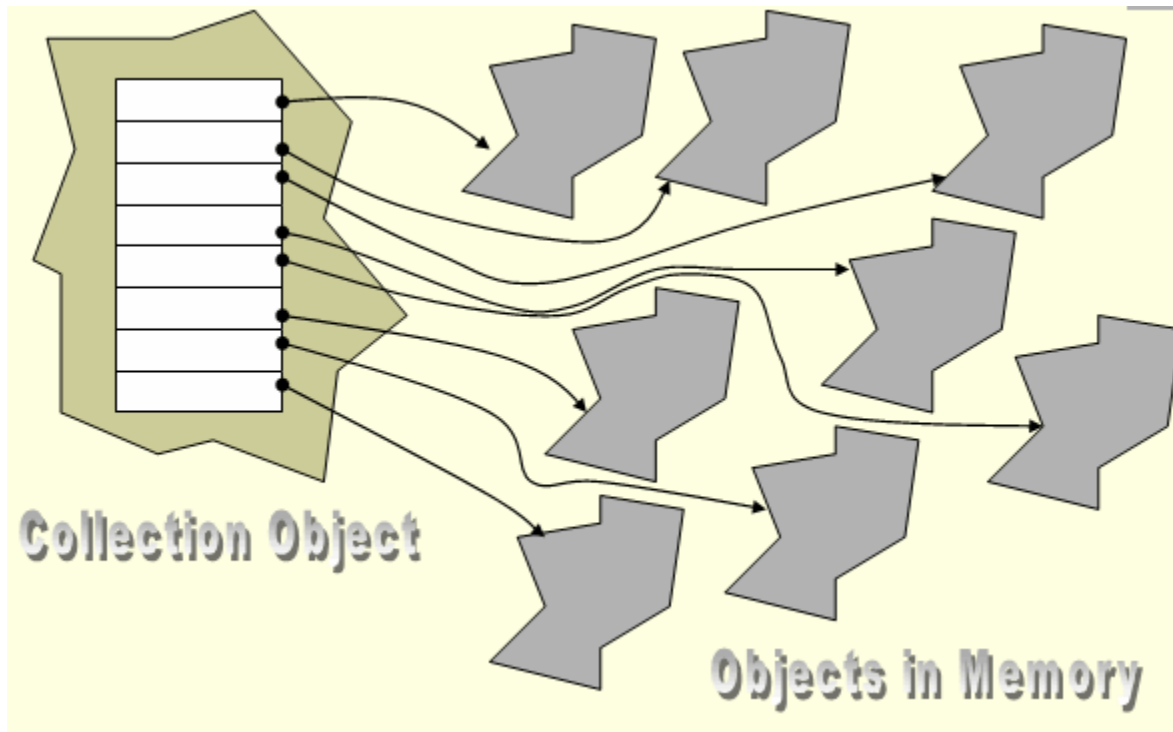
1. A means of getting objects into the collection
2. A means of accessing the objects in the collection

A common way of accessing objects in a collection is to use **collection-object-name[index]** notation (e.g., Quiz[3]). This notation is often called **array notation**, in honor of the first collection class: the **array**.

The array is the simplest form of collection, consisting of a pre-specified number of objects that can be individually accessed using **[index]** notation. In this section, we'll explore the nature of arrays, learn how to create them, and demonstrate how they can be used.

## C# Collections

Before turning to arrays, it is useful to look at what C# collection objects actually are. This is important, since this understanding will help to clarify how they are initialized and used. The diagram below conveys the basic idea, and probably needs some explanation.

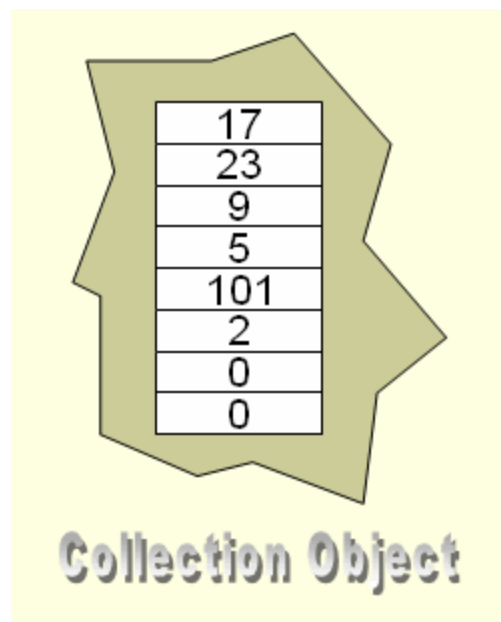


Objects--any objects--occupy some memory needed to store their member data and other odds-and-ends that don't presently concern us. Object collections, like any other object, also take up memory. But within the object collection, there is some structure used to keep track of the collected object. In the diagram above, the collection consists of 8 pointer elements (just like variables) that each point to a different object in memory. Thus, the collection object itself doesn't hold the object data--it simply keeps track of it (the way a table of contents or index keeps track of the actual text in a book).

From a practical perspective, this means that any time we create a variable that represents a collection object, there are three things that must take place:

1. We must declare the variable, just like any variable.
2. We must allocate memory for the collection object (the left object in the previous diagram).
3. We must allocate memory for the individual objects in the collection (unless they've already been created, and we're just collecting them).

For **value type** arrays, the third step can be omitted, since value type objects store actual values, not references to objects. This is illustrated by the following diagram:



In this case, an `int[8]` value array has been created and at least 6 of those values were initialized with integer values.

With this basic understanding, we can turn to arrays.

## The Array Collection

The array is the simplest C# collection. Conceptually, you can think of it as a one column table (as in the prior diagram) where each cell refers to an object somewhere else in memory. Some other useful details:

- The first element can be accessed at `array-name[0]`, the second at `array-name[1]`, the third at `array-name[2]`, etc.
- Once created, arrays cannot normally be resized (unlike other types of collections).
- The `Count` property identifies how many elements are in a particular array.

## Defining an Array

The variable to reference an array for some arbitrary class, we'll call it *ObjectType*, can be defined in three ways:

1. `ObjectType[] variableName1 = {element-list}; // Creates collection & elements`
2. `ObjectType[] variableName2 = new ObjectType[integer-val]; // Creates collection`
3. `ObjectType[] variableName3; // This array == null until initialized`

The first form performs all three collection creation activities: 1) it declares a variable name (`variableName1`), 2) it allocates a collection object (implicitly) and it attaches the objects to that

collection. The *object-list* between the braces is either a comma-separated list of previously created objects, e.g.,

```
PictureBox[] myPictures={picture1, picture2, picture3, picture 4, picture5};
```

or a series of object creations, such as:

```
PictureBox[] myPictures={new PictureBox(), new PictureBox(), new PictureBox(),  
new PictureBox(), new PictureBox()};
```

or a mix of the two. In both cases, a 5 element collection object would be created.

The second form declares the variable and creates the collection object. It does not, however, attach actual objects to the collection. As a result, each element of the array (which would have *integer-val* elements) would have a value of **null**--signaling that it references nothing.

The third form only declares the variable, whose value will be **null** until a collection object is assigned.

In all three cases, the fact that an array--not a simple *ObjectType* object--is being defined is signaled by the [] after *ObjectType* in the declaration. In reality, *ObjectType[]* is really an entirely different type of class from *ObjectType*.

Some examples of actual declarations are presented below:

```
// Collections created and initialized with elements  
int[] myInts = { 3, 4, 7, 2, 5, 1 };  
Form[] myForms = { new Form(), new Form(), new Form() };  
// Collections created, but no element objects created, must be added  
int[] myNewInts = new int[7];  
Form[] myNewForms = new Form[4];  
// Names for the collections, but no collection object created  
int[] intNoInts;  
Form[] formNoForms;
```

In the case of the second group, for the *myNewInts* array the construction process is completed, since the array will be populated with 0 values for elements *myNewInts[0]* through *myNewInts[6]*. For *myNewForms*, on the other hand, elements *myNewForms[0]* through *myNewForms[3]* all contain **null**, so additional assignments will be necessary to get the array elements to refer to **Form** objects.

## Iterating Through an Array

In addition to providing a method of accessing elements without giving them names, the fact that indexes can be used to access individual array elements also provides a convenient way to iterate through all the array's elements in a loop. This is further facilitated by the **Length** property of

every array object which identifies how many elements are in the array. This is the same property we used for **string** objects, which are basically arrays of characters.

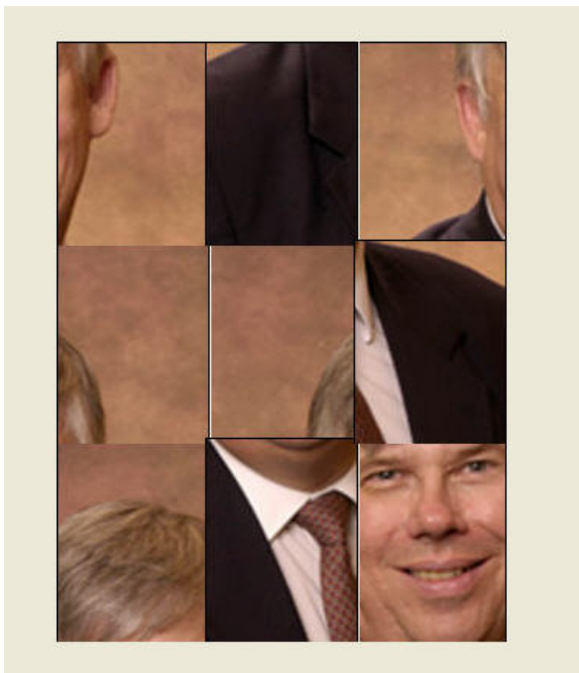
In the following example, we show how simple it is to access all the elements in an array (and total them up) using a simple **for** loop:

```
static void DemoArraySum()
{
    int[] myData={6,5,4,3,2,1};
    int nTotal=0;
    for (int i = 0; i < myData.Length; i++)
    {
        nTotal = nTotal + myData[i];
    }
    Console.WriteLine("The total is " + nTotal.ToString());
}
```

In this case, we make our loop variable (i) go through the array coefficients 0 through 5--since **myData.Length** is 6. As a result, the function returns 21 (6+5+4+3+2+1).

## Sample Code

In general, there are usually better collection choices than an array available in C#--its main deficiency being its limited ability to resize. If, however, you know in advance how many elements you'll have in a collection, then creating an array is a very viable alternative. One example in Assignment 2 involves a set of 9 pictures that must be unscrambled as part of the **Ism4300** class implementation. To "pass" that particular form, the user keeps swapping panes in the scrambled photograph, such as the one shown below:



To determine if the photograph has been assembled properly, before we scramble the photograph we need to store the correct position of each picture. To do this, we use a 9-element **Point** array (a Point contains an x,y position) and initialize it with the correct positions of each PictureBox, as shown below in the SetCorrect() function:

```
Point[] correct;
void SetCorrect()
{
    correct = new Point[9];
    correct[0] = new Point (pictureBox1.Location.X, pictureBox1.Location.Y);
    correct[1] = new Point (pictureBox2.Location.X, pictureBox2.Location.Y);
    correct[2] = new Point (pictureBox3.Location.X, pictureBox3.Location.Y);
    correct[3] = new Point (pictureBox4.Location.X, pictureBox4.Location.Y);
    correct[4] = new Point (pictureBox5.Location.X, pictureBox5.Location.Y);
    correct[5] = new Point (pictureBox6.Location.X, pictureBox6.Location.Y);
    correct[6] = new Point (pictureBox7.Location.X, pictureBox7.Location.Y);
    correct[7] = new Point (pictureBox8.Location.X, pictureBox8.Location.Y);
    correct[8] = new Point (pictureBox9.Location.X, pictureBox9.Location.Y);
}
```

The code can be explained as follows, according to the highlighted color:

- *Yellow*: we create a member variable named *correct* to hold the collection, but don't initialize it.
- *Green*: We create the 9-element collection object.
- *Light Blue*: In this function, we create new **Point** objects (using each picture's location) because we specifically don't want the array element pointing to the **PictureBox**'s object in memory--which would change each time the picture was moved by the user.

The CheckValid() function uses 9 **if** constructs to test the position of each picture--seeing if it is in the right position. If all 9 tests pass, the function returns **true**.

```
bool CheckValid()
{
    if (correct[0] != pictureBox1.Location) return false;
    if (correct[1] != pictureBox2.Location) return false;
    if (correct[2] != pictureBox3.Location) return false;
    if (correct[3] != pictureBox4.Location) return false;
    if (correct[4] != pictureBox5.Location) return false;
    if (correct[5] != pictureBox6.Location) return false;
    if (correct[6] != pictureBox7.Location) return false;
    if (correct[7] != pictureBox8.Location) return false;
    if (correct[8] != pictureBox9.Location) return false;
    return true;
}
```

# Generic Collections

## Learning Objectives

Upon completing this lesson, you should be able to:

- Distinguish between arrays, all-purpose collection classes and generic collection classes
- Declare a generic to hold a collection of a particular type of object
- Access objects in the collection
- Add and remove objects from that collection

## Overview

Prior to .NET 2.0, if you needed to hold a collection of objects, you had two alternative paths you could take:

- *You could define an array.* As we have seen, arrays are type-safe--meaning that you can only use an array to hold the type of objects it is declared to hold (e.g., `MultipleChoice[]` can only hold `MultipleChoice` objects). On the other hand, they are not easily resized and support only one means of access--using an index.
- *You could use one of .NET's all-purpose collection classes.* These classes--the most common example being `ArrayList`--could be automatically resized and offered variations that supported, for example, key-based access (e.g., if you were storing person objects, you might access them by integer index, just like an array, and by some string key you defined, such as `LastName + FirstName`.) The principal drawback of these classes was that all the elements in the collection were treated as **Object** objects. Since everything in .NET inherits from **Object**, this did not limit what you could store. It did, however, prevent the compiler from warning you if you placed the wrong type of object into a collection. To use the proper terminology, the collection was not **type-safe**.

When .NET 2.0 was introduced, it included **generics**. Rather than being a typical class, a generic is a class that allows parameters to be specified (within `<` and `>` delimiters) that specialize the class being created. For example, a `List<MultipleChoice>` class would define a `List<>` class that could only hold **MultipleChoice** objects, whereas a `List<Course>` class would have the same structure, but would be specialized for holding **Course** objects. What this gives us, then, is a type-safe, flexible collection.

In this section, we will introduce generic collections by examining the most commonly used of these collections, the `List<>` generic. This particular collection offers all the benefits of an array while simultaneously offering the flexibility of an all-purpose class such as the `ArrayList`. Indeed, generic collections are so powerful that programmers are being strongly encouraged to stop using the all-purpose collection classes and to use generics instead. For this reason, we won't even bother covering the all-purpose classes. Fortunately, if you understand the behavior of the generic classes, you pretty much know how to use the all-purpose classes. For example, the `ArrayList` class is virtually identical to the generic collection `List<Object>`--not that one would ever want to declare such a type-unsafe class.

## Collection Shapes

Before proceeding to the generic `List<>`, it is useful to introduce the concept of a **collection shape**. Conceptual, rather than physical, in nature, a collection's shape helps to determine the types of activities it is useful for. The types of properties influenced by a collection's shape include:

- Ease of insertion and deletion (at the beginning, end and in the middle)
- Ease of resizing
- How elements can be accessed
- Difficulty of finding something

For example, the array shape that we have already seen has the following shape properties:

- Elements are accessed by a position index
- Insertion and deletion of intermediate elements is difficult
- Adding elements is difficult
- Search is difficult--in other words, in an unsorted array, you need to go through the elements one-by-one to find a particular element

.NET supports many other collection shapes, including:

- *Sorted array*, which is like a regular array except searching is faster.
- *List*, which allows for easy insertion, deletion and resizing.
- *Dictionary*, which allows direct lookup of individual elements using a key that must match the element's key exactly.
- *Sorted dictionary*, which combines a dictionary with a sorted array--making it possible to find nearby keys if an exact match is not found (e.g., like a regular dictionary, which can be used to figure out how to spell a word).

Our focus in this reading will be on the `List<>` generic collection. A few other collections will be introduced in later modules.

## The `List<>` Generic Collection

The `List<>` generic collection has been modeled after the old `ArrayList` all-purpose collection which, as the name suggests, combines the characteristics of an array (fast access to elements by index) with those of a list (easy resizing, insertion and deletion). Once you have declared a `List<>` generic object--and populated it with the objects you're holding in the collection--you can use it just as if it were an array. What you can also do, however, is use a number of other members that aren't available for arrays, including:

- **Add(*element*)**: adds element to end
- **AddRange(*collection*)**: Adds a collection of elements, such as an array or another `List<>` object, to the end of the collection
- **InsertAt(*index,element*)**: Inserts element at specified position (*index*)

- **Remove(*element*)**: Determines if matching element exists and removes it
- **RemoveAt(*index*)**: Removes element at index

In addition, it has a **Count** property that returns the number of elements. This is equivalent to the **Length** property of an array, discussed in the previous reading.

The easiest way to understand the way the **List<>** works is to look at some examples. We begin with a comparison of a simple **int[]** array and a **List<int>**.

```
static void DemoArraySum()
{
    // int[] version
    int[] myData={6,5,4,3,2,1};
    int nTotal=0;
    for (int i = 0; i < myData.Length; i++)
    {
        nTotal = nTotal + myData[i];
    }
    Console.WriteLine("The total is " + nTotal.ToString());
    // List<int> version
    List<int> myList =new List<int>();
    myList.AddRange(myData);
    myList.Add(7);
    nTotal = 0;
    for (int i = 0; i < myList.Count; i++)
    {
        nTotal = nTotal + myList[i];
    }
    Console.WriteLine("The total is " + nTotal.ToString());
}
```

The key differences between the two collections are highlighted. These include the following:

- **List<int>** objects can't be initialized with brace-enclosed values in .NET 2005. (In .NET 2008, additional syntax has been added to allow brace-delimited initialization using a syntax similar to that of arrays). Thus we have to use **new** to create a new object.
- **AddRange()** can be used to add the elements in a collection to the end of the **List<>** collection. There is minimal restriction on the collection type of the argument, so we use our previously created array in the example.
- **Add()** can be used to add a single element.
- Virtually no changes need to be made to the loop itself--the only exception being that generic collections use **Count** instead of **Length** (which is reserved for arrays).

Thus, the choice between **List<>** and array collections tends to be based on how they are to be initialized (and will the number of elements in the collection be changing as the program runs--which would make **List<>** a no-brainer) and not based on ease of access once the collection has been constructed. Iteration and access tend to be nearly identical in difficulty.

## Example: Transcript class

In this example, we demonstrate how access to elements in a **List<>** object is essentially the same as it is for an array.

In Assignment 2, a transcript consists of a collection of course records (**Course** objects). This being the class, it makes sense to create a Transcript class that inherits from **List<Course>**. By doing this, we get all the nice collection properties built into the class, as demonstrated below:

```
public class Transcript : List<Course>
{
    public enum Ism
    {
        ism3232=3232, ism3113=3113, ism4212=4212, ism4220=4220, ism4300=4300, ism4234=4234
    };
    // Looks for a specified course number (Number) starting at position nStart;
    public int FindCourse(int Number, int nStart)
    {
        int i;
        for (i = nStart; i < this.Count; i++)
        {
            Course c = this[i];
            if (c.Number == Number) return i;
        }
        return -1; // Signifies no course was found
    }
}
```

Because Transcript inherits from List<Course> (top blue highlight), we can access **List<Course>** properties within member functions. For example:

- In the second blue highlight, we access the **Count** member (this could also have been written **Count**, instead of using **this.Count**).
- In the bottom blue highlight, we access elements by array index. Since [] brackets need an argument to the left, we have to use **this** here.

The FindCourse() function iterates through the **Course** elements in the collection (starting at index nStart, which would be 0 if we wanted to start at the beginning) looking for one that matches a particular course number. If we find a match, we break out of the for loop by returning the position (**return i**);. If we reach the end of the array (**i==Count**), we return -1 to signify that no match was found.

## Example: Quiz Creation

In this example, we demonstrate adding elements to a **List<>** collection.

In Assignment 2, a **Quiz** object is a collection of **Question** objects. For this reason, the class inherits from **List<Question>** as follows:

```

public class Quiz : List<Question>
{
    public double PointsPerQuestion = 0.0;
    public double Score = 0.0;
}

```

As part of the assignment, we need to create **Quiz** objects from time-to-time. This can be done by adding questions to a **Quiz** object, as follows:

```

Quiz myQuiz = new Quiz();
myQuiz.Add(new Question("What is the color of code comments?",
    new Answer("Red", false),
    new Answer("Green", true),
    new Answer("Blue", false),
    new Answer("Purple", false),
    new Answer("Yellow", false)
));
myQuiz.Add(new Question("What is the color of code literals?",
    new Answer("Red", true),
    new Answer("Green", false),
    new Answer("Blue", false),
    new Answer("Purple", false),
    new Answer("Yellow", false)
));

```

The top line creates the **Quiz** object. The next two constructs are calls to the **List<Question>.Add()** function, which **Quiz** inherits. Notice that we don't have to declare variable names for the two **Question** objects we are adding (within the **myQuiz.Add()** calls). That is because we'll be able to get at them by index, namely **myQuiz[0]** (the "code comments" question) and **myQuiz[1]** (the "code literals" question).

### *Example: Generating multiple choice dialogs*

In this example, we illustrate accessing elements from outside the collection class.

```

private void pictureKnight_Click(object sender, EventArgs e)
{
    AskQuestion();
}
int nQuestionCount = 0;
int nCorrectCount = 0;
PictureBox m_current;
Quiz testQuiz;
void AskQuestion()
{
    if (nQuestionCount >= 5) return;
    if (testQuiz == null)
    {
        testQuiz = CreateQuiz();
        m_current = pictureBox1;
    }
    MultipleChoice question = new MultipleChoice(testQuiz[nQuestionCount]);
    nQuestionCount++;
}

```

In the **Ism4234** class, each time the picture of a knight is clicked, the **AskQuestion()** function is called, involving the creation of a **MultipleChoice** dialog from a **Question** object in a **Quiz** collection. This code works as follows (see highlights):

- *Yellow*: **AskQuestion()** called when the user clicks the knight's picture
- *Light Blue*: **nQuestionCount** tracks what question we're on
- *Green*: A **MultipleChoice** dialog is created based on the question within the **Quiz** collection (a 5 question quiz initialized in **CreateQuiz()** static function). Here, the **testQuiz** collection (inheriting from **List<Question>** as we have seen) is accessed by index (i.e., **testQuiz[nQuestionCount]**)

# foreach Construct

## Learning Objectives

Upon completing this reading, you should be able to:

- Use a **foreach** construct to iterate through a loop
- Explain how alternative looping constructs can be used to accomplish what a **foreach** loop does

## Overview

One of the most common activities of loops involves iterating through all the elements of a collection. We have already seen an example of this, as follows, using a **for** loop:

```
public class Transcript : List<Course>
{
    public enum Ism
    {
        ism3232=3232, ism3113=3113, ism4212=4212, ism4220=4220, ism4300=4300, ism4234=4234
    };
    // Looks for a specified course number (Number) starting at position nStart;
    public int FindCourse(int Number, int nStart)
    {
        int i;
        for (i = nStart; i < this.Count; i++)
        {
            Course c = this[i];
            if (c.Number == Number) return i;
        }
        return -1; // Signifies no course was found
    }
}
```

To accomplish this iteration through the **Transcript** class--inheriting from the **List<Course>** generic collection (top highlight)--elements, we needed to:

- establish a counter variable, **i**
- test its value against the number of elements in the collection (middle highlight)
- access each element by using array notation (bottom highlight)

Given that we typically spend a lot of time iterating through collections in our projects, it would be nice to use a less tedious method for accomplishing such an iteration. In this section, we introduce the **foreach** construct, a C# innovation that allows just such an iteration.

## foreach Loops

The **foreach** construct is so simple--and easy to use--there is not a lot that can be said about it. Its form is as follows:

```
foreach (Object-Type variable-name in collection-name)
{
    ...Code to be repeated...
}
```

## Loop Comparisons

We substitute **foreach** loops for **for** loops in the two examples presented below, which iterate through an array and a **List<>**:

```
int nTotal = 0;
// for loop
for (int i = 0; i < myData.Length; i++)
{
    nTotal = nTotal + myData[i];
}
Console.WriteLine("The total is " + nTotal.ToString());
nTotal = 0;
// foreach loop
foreach (int k in myData)
{
    nTotal = nTotal + k;
}
Console.WriteLine("The total is " + nTotal.ToString());
```

In the array case, the use of **foreach** allows us to iterate without worrying about two things:

- How to determine the size of the array (i.e., the **Length** property)
- How individual elements are accessed (i.e., the use of *array-name[index]* notation)

These can be quite convenient, as **foreach** can be used for virtually any collection shape. The versatility of the construct can be illustrated by the iteration through the **List<int>** collection, shown as follows:

```

List<int> myList = new List<int>();
myList.AddRange(myData);
myList.Add(7);
nTotal = 0;
// for loop
for (int i = 0; i < myList.Count; i++)
{
    nTotal = nTotal + myList[i];
}
Console.WriteLine("The total is " + nTotal.ToString());
nTotal = 0;
// foreach loop
foreach (int k in myList)
{
    nTotal = nTotal + k;
}
Console.WriteLine("The total is " + nTotal.ToString());

```

Of particular interest in this example is the use of **foreach** led to a **List<int>** loop that was exactly the same as the loop used for the **int[]** array--the only exception being the name of the collection.

## foreach Limitations

There are certain limitations that you need to be aware of in using **foreach** loops. Specifically, the construct assumes that the collection does not change during the iteration process. That means that your loop should not:

- Add elements to the collection
- Remove elements from the collection
- Reorder elements in the collection

The compiler will identify obvious violations of this. More subtle instances--such as calling a function that adds a collection element--are likely to be identified through a runtime error.

## Examples

Some samples can be used to explain the concept.

```

foreach (Question q in myQuiz) {...code...}

```

In this example, **myQuiz** would need to be an object that contains a collection of **Question** objects, such as a **Question[]** array, a **List<Question>** generic collection, or a class inheriting from a collection, such as the **Quiz** object of Assignment 2, which inherits from **List<Question>** (see code example appearing shortly). Within the loop, each **Question** object could be accessed through the variable **q**, a much cleaner approach than using an element reference like **myQuiz[i]**.

Another example:

```
foreach (string s in checkedCourses.CheckedItems) {...code...}
```

In this example, drawn from the final part of Assignment 2, a checklist box holds a list of course names (**string** objects) and also has a collection member property (**CheckedItems**) that contains the names of all the elements checked by the user. This allows us to loop through them (used in the **EnrollmentForm** class). Very commonly, windows controls--such as menus, lists and grids--have collection properties that allow internal data to be accessed.

## Within-Class Example

**foreach** constructs can also be used within objects that inherit from collection classes, as shown in the Assignment 2 example below:

```
public class Quiz : List<Question>
{
    public double PointsPerQuestion = 0.0;
    public double Score = 0.0;
    public bool AdministerQuiz()
    {
        Score = 0.0;
        foreach (Question q in this)
        {
            MultipleChoice dlg = new MultipleChoice(q);
            if (dlg.ShowDialog() == DialogResult.OK)
            {
                if (dlg.Correct) Score = Score + PointsPerQuestion;
            }
            else return false;
        }
        return true;
    }
}
```

To identify the collection in the **foreach** loop, we use the **this** keyword--since our **Quiz** class is, itself, a collection by virtue of its inheritance. Within the loop, the individual elements are then referred to by **q** (the variable name identified in the **foreach** construct).

# DataGridView Control, Part I

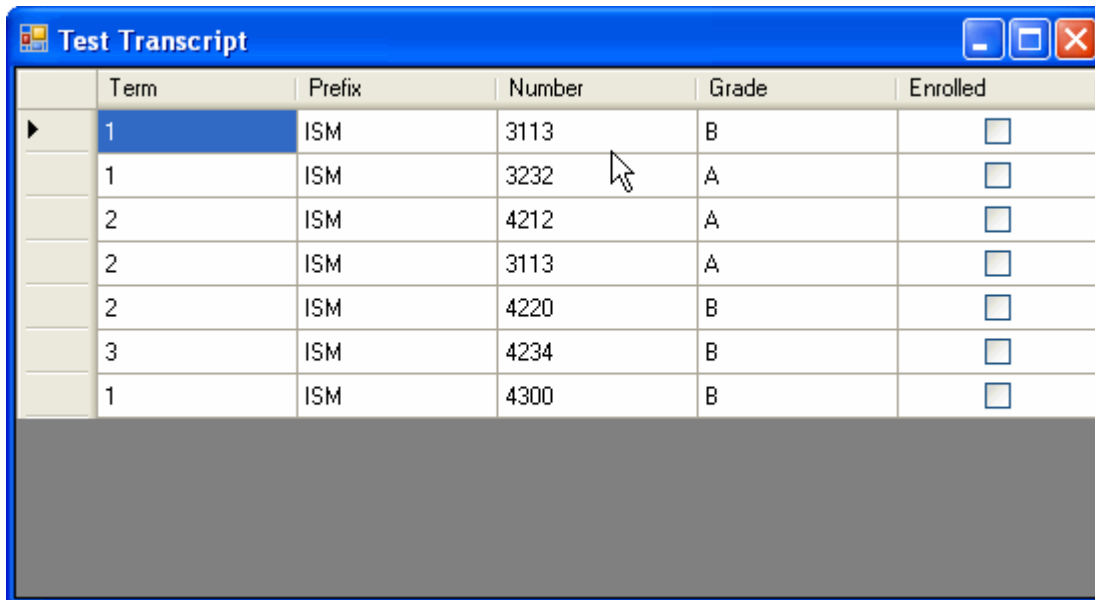
## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what a **DataGridView** control is
- List some of the key purposes of **DataGridView** controls
- Prepare a collection to make it appropriate for a **DataGridView** control
- Attach a collection to a **DataGridView** control

## Overview

The **DataGridView** control is a remarkable class. With a minimal amount of programming effort, you can create tabular displays of data, such as the **Transcript** object displayed below:



	Term	Prefix	Number	Grade	Enrolled
▶	1	ISM	3113	B	<input type="checkbox"/>
	1	ISM	3232	A	<input type="checkbox"/>
	2	ISM	4212	A	<input type="checkbox"/>
	2	ISM	3113	A	<input type="checkbox"/>
	2	ISM	4220	B	<input type="checkbox"/>
	3	ISM	4234	B	<input type="checkbox"/>
	1	ISM	4300	B	<input type="checkbox"/>

Although designed to be used principally with an external data source (i.e., a database), it can also be used to display the contents of a collection. To do so, however, the objects being collected (i.e., **Course** objects in the above illustration) must have the members to be displayed implemented as properties. In this reading, we'll examine this process.

## DataGridView

The **DataGridView** control was introduced in .NET 2.0. Although particularly useful as a means of quickly attaching objects to tables in a database (the subject of a *DataGridView, Part II* lecture in later modules), the control is not limited to displaying databases. Specifically, it can be used to:

- *Display **DataSet** objects.* This is a .NET class that can be used to store relational data in memory and can be populated from a number of sources, such as databases and XML.
- *Display collection objects.* Any .NET collection object can be attached to a **DataGridView**. When this is done, each object in the collection is given a row and columns are defined based on the collected object's public properties.

## Preparing Objects for DataGridView

Prior to attaching an object to a DataGridView, some preparations are appropriate. Specifically, objects in the collection should meet the following characteristics:

- All members to be displayed in the grid must be defined as **public** properties. That is how the grid knows how to display them.
- Each property must be of a type that has a reasonable **ToString()** translation.

The second item requires some further explanation. Since the **DataViewGrid** needs to put data in each cell, it faces a bit of a problem when a property is not something with an obvious string display, such as a **string**, number, **bool** or date--all of which have self-evident string displays (e.g., "17", "21 January 2006"). To deal with this, the control simply displays the class name for any object it doesn't know how to display--which is the default behavior of the **ToString()** member of the **Object** base class. Thus if you have a **MyClass** property member in your object named **PropVal**, the column header will be *PropVal* and the data in each row will be something like *YourProject.MyClass*--not very informative.

There are two ways of addressing columns that don't display properly in grids:

1. You can hide that particular property, using the column editor (as shown later in this section).
2. You can override the **ToString()** property of the class being displayed as a property. This is an example of the use of polymorphism and is covered in a later module.

For our example and in Assignment 2, we avoid this problem by attaching a collection of **Course** objects to our grid. The public properties of a Course object are:

- **int** Term
- **string** Prefix
- **int** Number
- **string** Grade
- **bool** Enrolled

Since **string**, **int** and **bool** all have appropriate **ToString()** members, the problem doesn't come up.

## Implementing DataGridView

There are a number of steps required to implement a DataGridView control attached to a collection class in a form. These involve:

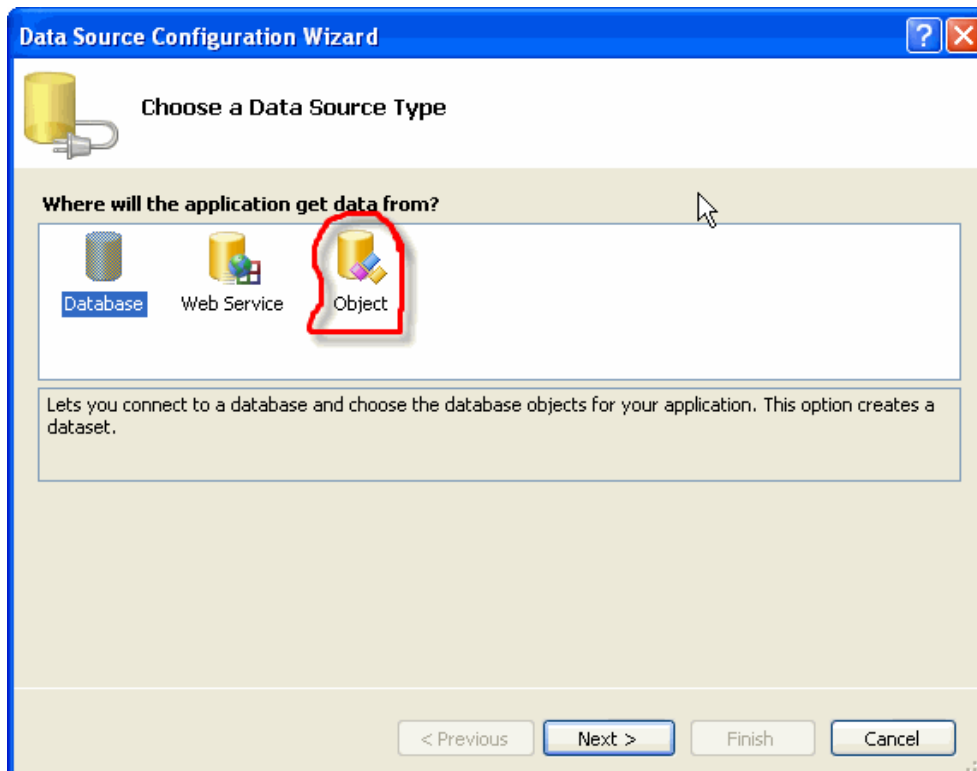
- Placing the control on the form
- Establishing the collection class as a project data source
- Associating the class with the DataGridView control
- Customizing the display, as needed
- Assigning a collection object to the control in the program code

We will now demonstrate this with the **Transcript** class developed in Assignment 2. These steps essentially repeat the same steps detailed in the assignment itself.

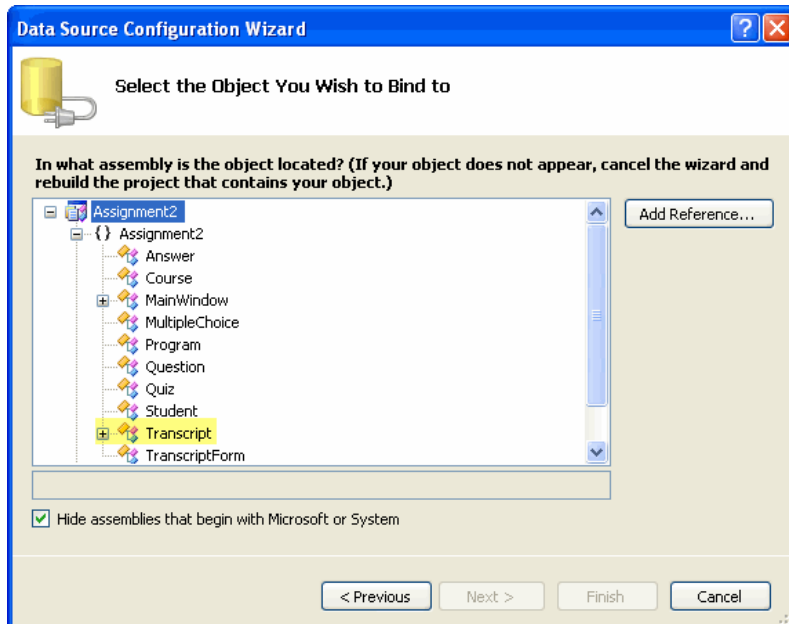
### *Setting up the DataViewGrid control*

Assuming that you have a Windows Form ready for use, setting up the DataGridView object involves the following steps:

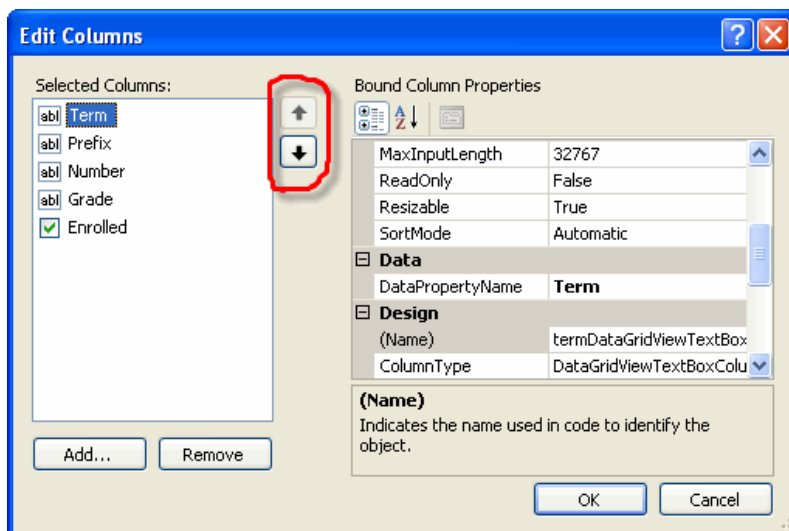
- Add a DataGridView object to the form.
- Using the control's common task menu (or the Solution Explorer), select "Add a Project Data Source" to open the *Data Source Configuration Wizard*, where you will choose "Object" as circled below.



- Click the "Next" button and open up the project and select the appropriate collection class--in this case, **Transcript**, as highlighted below.



- Click "Finish" and the data source will be attached to the grid using a **transcriptBindingSource** object that the wizard automatically creates. The purpose of the binding source object is to make the translation between your object's internal data and what appears on the grid. It can, for example, translate integers so they are rendered as strings on the grid, but when the user edits the strings in the grid, it translates them back to integers and updates the appropriate properties of the object in the collection.
- When you look at the grid, you'll probably notice that the properties aren't listed in the order you'd like. Right click the column headers then select "Edit Columns." The circled buttons on the column editor dialog, shown below, can be used to reorder the columns, and other column properties can be edited as well in the properties box.



When completed, your form should look something like the form presented earlier in the overview.

### *Assigning Object to Control*

The actions just described in the Form Designer prepared your control to display the data for any Transcript object. In your code, however, you need to attach the specific object that you want displayed. That will require a step such as the following (highlighted):

```
public partial class TranscriptForm : Form
{
    public TranscriptForm(string Title, Transcript trans)
    {
        InitializeComponent();
        dataTranscript.DataSource = trans;
        Text = Title;
    }
}
```

It should be noted that a step such as this is not unique to the **DataGridView** control. In many cases, however, such statements are embedded in the designer-generated code.

# Form Inheritance

## Learning Objectives

Upon completing this reading, you should be able to:

- Create a form that inherits from another form
- Identify activities that you must perform to make a form suitable as a base class for other forms

## Overview

Prior to the introduction of .NET, it was nearly impossible to inherit from a form class. With the introduction of C# and .NET, however, inheriting from a form became little different from inheriting from any other class. Form inheritance offers many potential benefits, including:

- The ability to create a common look-and-feel in a base class form that can be incorporated into child forms with no additional coding (similar to the way slide designs can be used in PowerPoint).
- The ability to incorporate repetitive interface code into a base class form so it does not need to be repeated in each child form. For example, common menu options might be handled in the base class.
- The ability to change the look and feel of an entire set of child forms by making changes to a base class form.

In this section, we look at how we can inherit forms, and what design guidelines should be applied to the base classes for our forms.

## Obstacles to Form Inheritance

Although form inheritance is straightforward, there are certain things about forms that are different from inheriting data-centric classes, as we have done up to now. These include the following issues, some serious and some easily overcome:

- The **Form** class, from which all designer-generated forms inherit, handles a number of events as part of its base class implementation--many of which should only be done once. When responding to these events in normal forms, this fact is taken into account (beneath the surface). Should you override these in both your base class form and in a child form, however, some events may be done twice. Often, doing so leads to "unpredictable" results (the euphemism used by programmers to describe "very bad" happenings). The best solution here is probably to avoid handling events--as much as possible--in the base class form that you might conceivably need handled in the child forms. Also, setting event access to **protected**, rather than **private**, is generally preferable.
- The designer-generated code in a form you create, by default, includes a lot of **private** members (e.g., all the controls on the form). If you inherit from such a form, you won't be

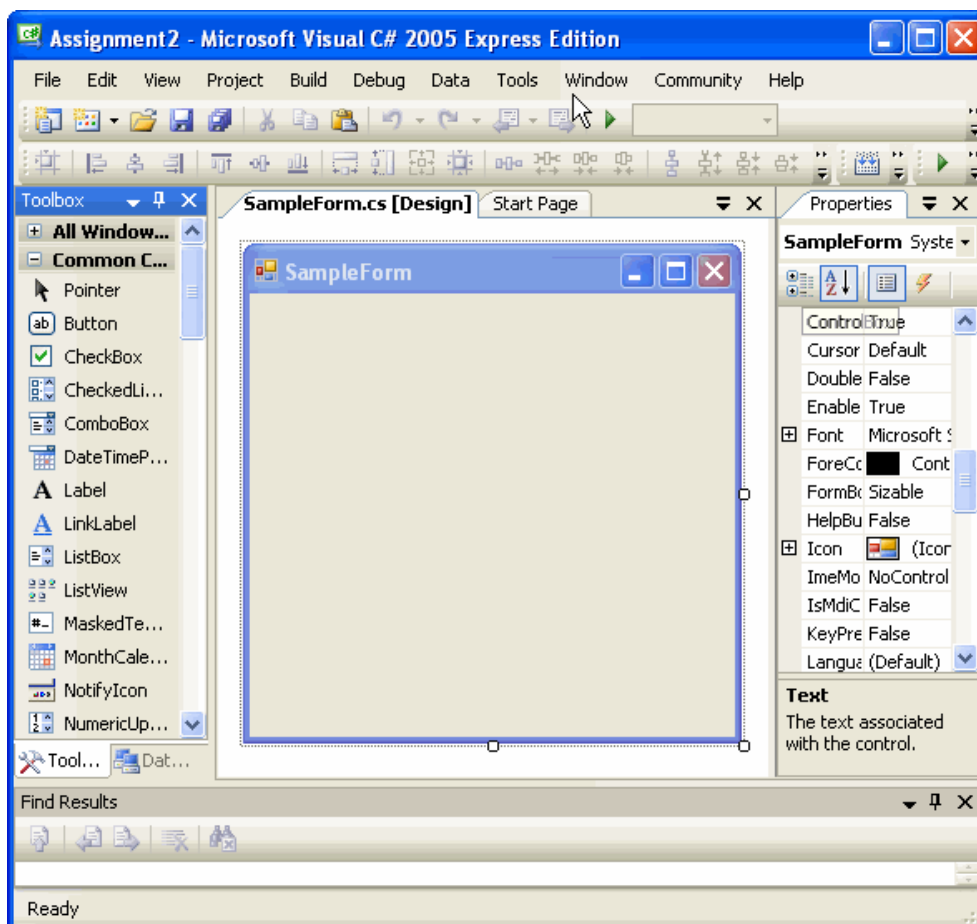
able to access these controls in your code. This problem is a relatively small one, once you recognize it, since you can define **public** or **protected** properties in your base class form to provide access to base class data that you need. You can also change the access level of base class controls, using the control properties window.

- When the designer brings up a form, it actually calls some of the event functions you've written to establish the layout. If you've overridden the wrong events, this can crash **Visual C# 2005 Express Edition**. (Interestingly, the forms may run perfectly well when your program itself is run). My solution to this problem--sad to say--has been to stop overriding any function that repeatedly crashes VCEE.

Microsoft has a tool, the *Inheritance Picker*, which is specifically designed to help implement form inheritance. Another obstacle, specific to VCEE, is that the *Express Edition* product line doesn't include this particular feature. Fortunately, we don't seem to need it.

## Implementing Form Inheritance

When form inheritance works (which is most of the time) it is quite stunning in its impact. Assume you've already prepared a base class form, such as **IsmClassForm**--used in Assignment 2. You just add a new form to your project:

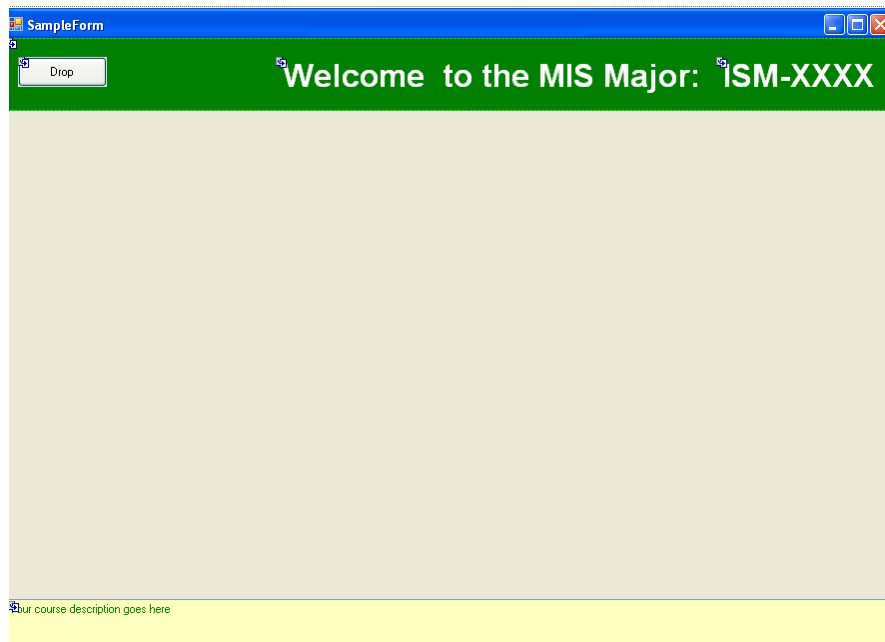


You then go to that's form's class code and change **Form** to your base class, in this case **IsmClassForm**.

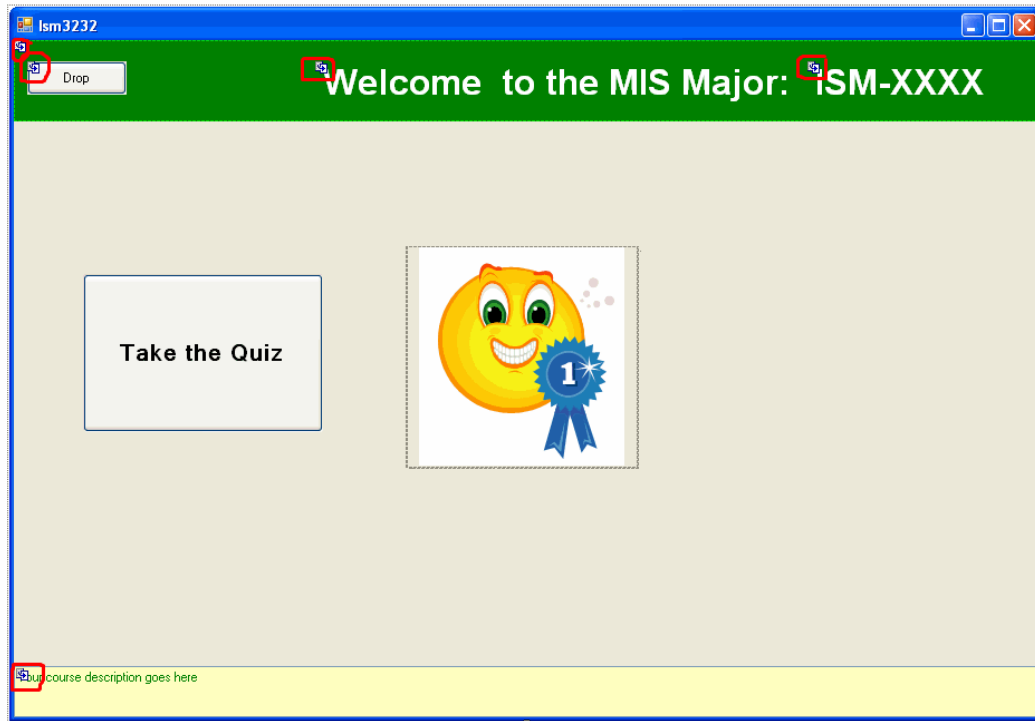
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Assignment2
{
    public partial class SampleForm : IsmClassForm
    {
        public SampleForm()
        {
            InitializeComponent();
        }
    }
}
```

Then you return to the form designer and, poof, your form has completely changed to mimic the base class form, as shown:



You are now ready to begin adding your own controls to the child class form. As you do so, you may notice that the base class controls all have little arrows associated with them (circled below). These indicate that the controls are inherited. You may also find that they cannot be edited--although this is something that you can change.



Obviously, the process of inheriting a form is easy. Preparing the base class requires a bit more thought.

## Preparing Base Class Form

The basic mechanism of form inheritance is identical to normal C# inheritance. It is a two step process:

1. When a child form is created, a call to the default (no argument) base class constructor is made--just as it would be for any C# object (where no other specific base constructor is specified, via **base(...args...)** in the child constructor). The base form constructor calls **InitializeComponent()** for the base form, creating and initializing the base form controls.
2. Then the child form constructor begins, calling the child version of **InitializeComponent()** to add any additional controls specified in the child form.

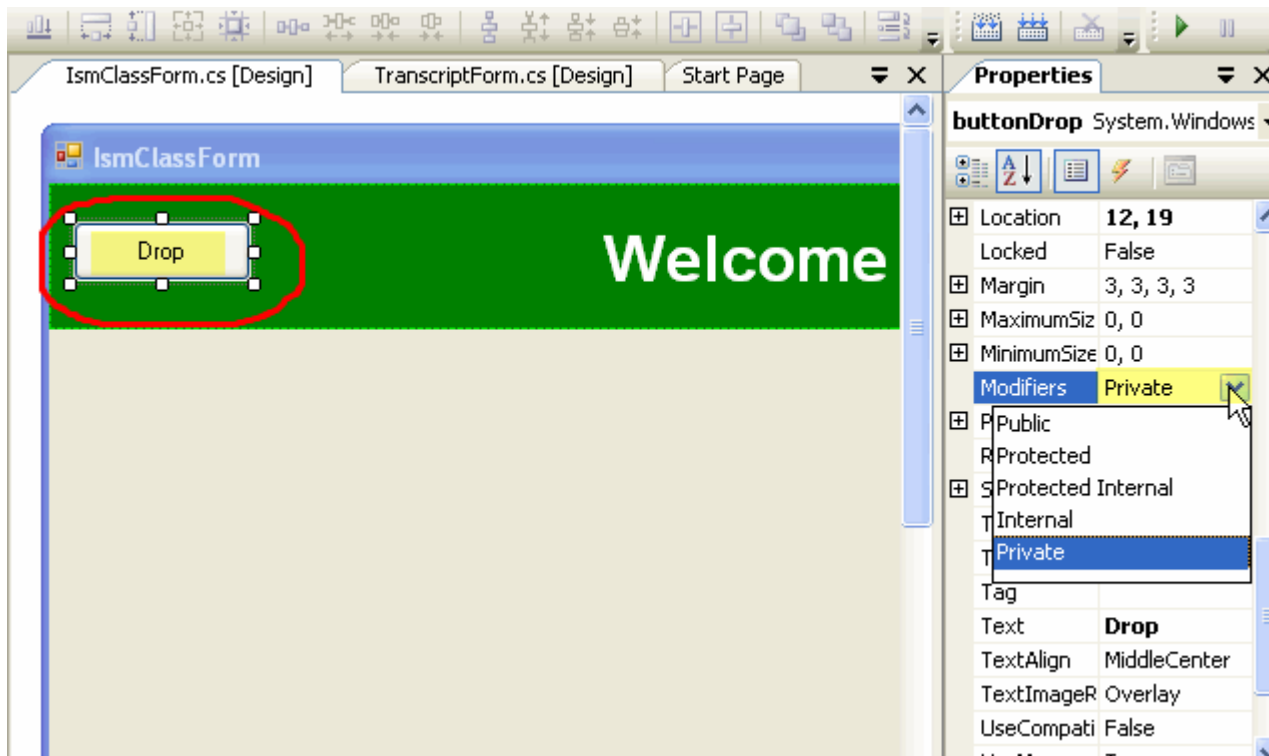
To facilitate this process, some issues are involved in preparing a base class form so that it works well when it is inherited. Since the challenges to inheriting forms have been presented earlier, we'll now focus on solutions. Specifically:

- Minimize the definition of form-level event handlers in the base class. This is particularly true of event handlers that you might need to implement in child classes (I had particular problem with the **Show** handler, which kept crashing my form designer when I tried to display the base class).
- Ensure that all base class form data and controls that you may need in the child form are declared to be **protected** (or **public**), since **private** base class members are inaccessible in the child class. This can be done in two ways:

1. Define properties to access base class form data or change handler access, as shown below

```
// Base class access
public string Description
{
    get
    {
        return textDescription.Text;
    }
    set
    {
        textDescription.Text = value;
    }
}
public string Title
{
    get
    {
        return this.Text;
    }
    set
    {
        this.Text = value;
    }
}
public string Welcome
{
    get
    {
        return labelWelcome.Text;
    }
    set
    {
        labelWelcome.Text = value;
    }
}
protected Student m_student;
protected Course m_course;
protected IsmClassForm m_next;
protected EnrollmentForm m_home;
```

2. Change the control **Modifiers** property in the base class to an appropriate level, as shown below. If the button is set to **protected** or **public**, for example, you can access it in your child form code and set its properties in the child form designer. Indeed the only property you can't adjust in the child form is the **Modifiers** property itself.



# Exception Handling

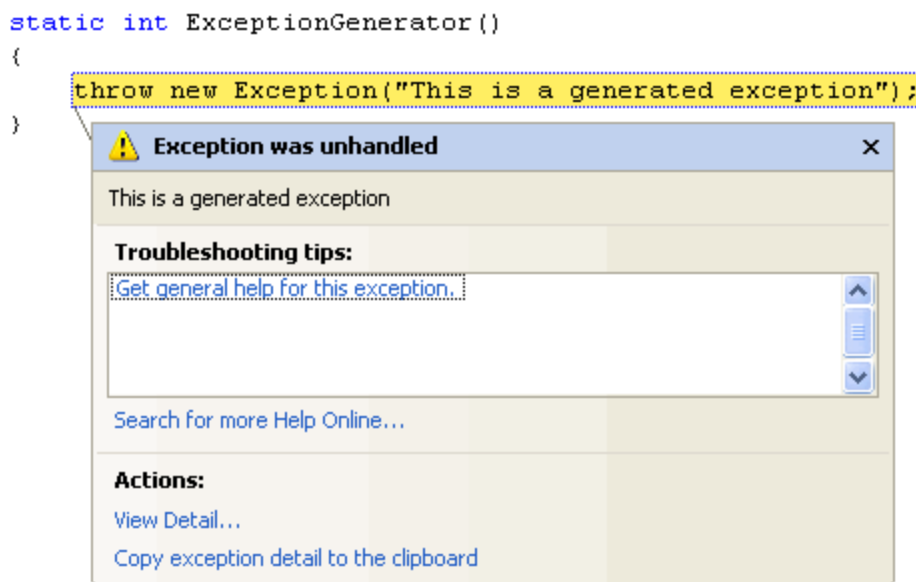
## Learning Objectives

Upon completing this reading, you should be able to:

- Explain the purpose of exceptions, and the role they play in programming
- Generate a simple exception, using the **throw** keyword
- Handle an exception using the **try...catch...finally** construct

## Overview

The scenario is a common one--and one you have likely encountered already. You are running your program under the debugger and suddenly you find yourself facing a screen like the following:



Your program has encountered an exception!

Exceptions and exception handling are actually important tools in the programmer's repertoire. Many things can cause a program to stop working. These include:

- *Programmer error*, such as failing to check the denominator for 0 before performing a division or trying to access property values of a **null** object.
- *Interaction with system resources*, for example the operating system doesn't have enough memory to acquire a needed object for your program or the printer is tied up when you try to print.

- *User error*, such as trying to save to a read-only CD or looking for a file in a path that doesn't exist.

The worst thing you could do in situations like this would be to let a program keep running after an error occurs that could corrupt data or code. It is, for example, far better to let your word processor crash than to allow you to save a file that is so corrupted that it can't be reopened. On the other hand, crashing programs every time something small goes wrong can be very annoying to the user, and can lead to substantial declines in productivity.

Thus, a far better solution is to provide a mechanism for trapping errors within a program in such a way that the programmer can determine whether or not recovery is possible. That is the goal of exception handling.

## Exceptions in C#

There are two pieces to the exception topic: *generating exceptions* and *handling exceptions*. Since the first, generating, is nearly trivial to accomplish, we'll consider it first.

### Generating Exceptions

The **throw** statement is used to generate an exception in C#. It is followed by an **Exception** object (or one that inherits from **Exception**, which is more common). An example of such code was already presented in the overview:

```
throw new Exception("This is a generated exception");
```

Upon encountering this code, the program will commence unraveling until exception handling code is encountered. If no such handling is present in the application, the program will terminate and the operating system will pop up one of those "Program has terminated unexpectedly" boxes that offers to let you send information to Microsoft. Thus, you want to handle exceptions rather than ignoring them, whenever possible.

### Handling Exceptions

In C#, exceptions are handled using a **try...catch...finally** construct that allows you to trap the exception and resume program execution, if that is desirable. There are four different types of code block within the construct:

- **try { ... }**: Contains code that might generate an exception. It is important to realize that even if you don't plan to **throw** exceptions in your code, plenty of .NET functions do. If you anticipate possible errors and attempt to catch these, your program is likely to be very fragile.
- **catch (ExceptionClass var) {...}**: This code gets executed if an exception is encountered in the **try {...}** block and the exception's type matches the class (e.g., *ExceptionClass*, in the example). The variable (e.g., *var*) can be examined in the block for more information.

- **catch { ... }** : Any exception not caught by previous catch blocks gets caught here. Since no variable is declared, no information about what caused the exception can be determined.
- **finally {...}** : Code that **always** gets executed, whether an exception was generated or not. This block is often used to do things like close an open file.

A typical complex form of the **try...catch...finally** construct might appear as follows:

```

try
{
    // Your code that might generate exception
}
catch (System.DivideByZeroException ex1)
{
    // Example: this block would handle divide by zeros. ex1 could be examined for more
    information
}
catch (System.DllNotFoundException ex2)
{
    // Example: this block would handle missing dll files. ex2 could be examined for more
    information
}
catch (System.Exception ex3)
{
    // catches most remaining exceptions, since all C# exceptions inherit from System.Exception
}
catch
{
    // Catches anything that's left. Since there's no variable provided, its cause can't be analyzed
}
finally
{
    // Optional and always done, whether or not there's an exception
}

```

## Example

A reasonable example of when it is appropriate to handle exceptions can be found in the **Ism4234** class of Assignment 2. That class includes a *Windows Media Player* control that plays recorded segments as the user clicks on pictures of a knight. Unfortunately, there are a lot of things that can go wrong with the control--the most common of which is that you supply it the name of a file that it can't find. (You don't need to ask how I know this...) Since a control problem like a missing file automatically throws an exception, it is a good idea to assign the **URL** property--which is what would generate a missing file exception--of the control within a **try** block. To accomplish this within the **Ism4234** class, the following function was used to play a specific sound file:

```

void PlayKnight(string sFile, string sText)
{
    try
    {
        axWindowsMediaPlayer1.URL = sFile;
    }
    catch
    {
        textComments.Text = "The knight says: \"" + sText + "\"";
    }
}

```

Passed into the function are **sFile**, the name of the .wav file, and **sText**, the alternative text to display if the media player doesn't work. Thus, we try to play the file within the **try** block. If no exception occurs, the function returns normally. If assigning the **URL** property generates an exception, on the other hand, we fall into the **catch** block and the text comment is assigned to a text box control (**textComments**) already displayed on the screen. Since we have a perfectly good alternative action to perform if an exception is generated, we don't bother capturing the exception property--e.g., **catch (Exception ex)**--since we don't really care what caused the problem (moreover, we already have a pretty good idea!)



## **Modules 3 & 4: Special Topics**



## Indexers (Module 3)

### Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by the term indexer
- Describe possible situations where indexers might be useful
- List similarities and differences between indexers and properties
- Define single argument indexers, using both integers and strings
- Define multiple argument indexers

### Overview

An *indexer* is a special form, closely related to the property, which permits array-style bracket notation to be used to access information in a class that is not necessarily an array. Indexers extend the array notation in two ways, however:

- They permit non-integer coefficients to be defined, such as **string** objects.
- They permit multiple coefficients to be defined, allowing comma-separated access, such as **myIndexedObj[4,2]**.

These properties make indexers invaluable in certain circumstances. For example:

- Non-integer coefficients can be used to implement a lookup interface. For example, in a **DataSet** object, **Table** object can be accessed by sequence number (i.e., integer index, such as **Tables[0]**) or by name (e.g., **Tables["Games"]**).
- Multiple coefficients can be used to simulate multi-dimensional arrays, where doing so adds to program clarity.
- The indexer can be recognized by some controls (e.g., the **DataViewGrid**), allowing automatic display of data stored in non-standard collections

### Defining an Indexer

The basic form for defining an indexer within a class looks as follows:

```

public return-type this[...arguments...]
{
    get
    {
        ...get code, if readable...
    }
    set
    {
        ...set code, if writeable...
    }
}

```

The construct is similar to a property definition in three ways:

1. **get** and **set** are used to move values in and out
2. **get** must return an object matching the return type
3. **set** can use the **value** keyword to access the value on the right hand side of an assignment

There are, however, three key differences between indexers and properties:

1. **this**[...argument-list...] is used to identify the comma-separated list of variables that will be used when the index is applied. e.g., the argument **this**[**int** val] in an indexer within a class would signify that the myObj[2] would access the integer. Similarly, the argument **this**[**string** a1,**int** a3] would signify access using myObj["Hello",3].
2. The **this** keyword is used in place of the property name since indexers defined within a class can only be defined for the class itself, not for members. (Within member class definitions, of course, indexers can be defined).
3. Arguments declared within the argument list can be accessed within the **get** and **set** constructs.

The easiest way to understand these rules is to see some indexers in action.

### Example 1: Simple integer indexer

Within the **Bingo** class, rows on the bingo card (**Row** objects) are generally accessed using the **B**, **I**, **N**, **G** and **O** properties, although the underlying data is stored in a protected 5-element **Cell** array called **m\_cells**. Because it would also be convenient to access the cells by numeric position (e.g., to move between the **Card** object and a **DataGridView** control), we define the following indexer:

```

// Declaring an indexer
public Cell this[int i]
{
    get
    {
        if (i < 0 || i > 4)
        {
            throw new Exception("Illegal cell index being accessed in Row object");
        }
        return m_cells[i];
    }
    set
    {
        if (i < 0 || i > 4)
        {
            throw new Exception("Illegal cell index being set in Row object");
        }
        m_cells[i] = value;
    }
}

```

The code is immediately identifiable as an indexer because it uses **this** in the declaration (yellow highlight) followed by the [...arguments...] (green highlight). In addition to moving data back and forth between the underlying **private Cell[] m\_cell** array, it also validates the incoming value **i**, making sure it is between 0 and 4, throwing an exception if it isn't.

## Example 2: Simple string indexer

Another variation on the integer-indexed indexer is to allow access by cell name, e.g., `myRow["N"]` or `myRow["B"]`. This indexer is defined below:

```

public Cell this[string ltr]
{
    get
    {
        switch(ltr.ToUpper())
        {
            case "B":
                return m_cells[0];
            case "I":
                return m_cells[1];
            case "N":
                return m_cells[2];
            case "G":
                return m_cells[3];
            case "O":
                return m_cells[4];
            default:
                throw new Exception("Illegal cell letter being accessed in Row object");
        }
    }
    set
    {
        switch (ltr.ToUpper())
        {
            case "B":
                m_cells[0]=value;
                break;
            case "I":
                m_cells[1]=value;
                break;
            case "N":
                m_cells[2]=value;
                break;
            case "G":
                m_cells[3]=value;
                break;
            case "O":
                m_cells[4]=value;
                break;
            default:
                throw new Exception("Illegal cell letter being set in Row object");
        }
    }
}

```

In this case, a case construct is used to validate the input string, but the same basic access to the underlying `m_cells` array is provided, and an exception is thrown if the column does not match B, I, N, G or O. For user convenience (and programmer convenience), the input string is capitalized in the `switch` statement using `ToUpper()`, meaning we only have to check for uppercase letters.

### Example 3: Multiple coefficient indexers

The multiple coefficient indexer is defined exactly as the single argument version. Such an indexer is defined in the **Card** class (which inherits from **List<Row>**), where it is used to access **Cell** objects using *column,row* notation. The code is as follows:

```
Cell this[int col, int row]
{
    get
    {
        if (row >= 5 || row < 0)
        {
            throw new Exception("Invalid row column combination");
        }
        return this[row][col];
    }
    set
    {
        if (row >= 5 || row < 0)
        {
            throw new Exception("Invalid row column combination");
        }
        this[row][col] = value;
    }
}
```

Here, the indexer has two arguments (yellow highlight). How cells are accessed is also of interest, since it combines array access with the use of our **Row** integer indexer. It works as follows:

- **this[row]** (green highlight) returns a **Row** object from the **List<Row>** collection that **Card** inherits from.
- We apply the **Row** indexer--[col]--to that Row object to get the **Cell**.

Also useful to note, if we had a **Card** object **myCard**, we could access a given cell in two ways:

1. **myCard[col,row]** -- using our indexer
2. **myCard[row][col]** -- accessing the collection row, then using the row indexer to get the column

These would access the same cell. The programmer would do well to remember, however, that the arguments come in a different order.

# Resources and Bitmaps (Module 4)

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain the process drawing that occurs in .NET applications
- Describe the uses of resources within a program
- Load images into your project as resources and access them in your programs
- Create Bitmap objects from files and resources
- Modify Bitmap properties to set a transparent color

## Overview of Windows Display

There are two common ways to present an interface to the user:

- Create an interface using forms and controls, as we have done throughout this course
- Draw directly to the window

The two approaches can, of course, be mixed. You could, for example, create a form with an empty picture box, then draw to that picture box. Alternatively, you could draw on your form, then place controls over it.

Bitmaps provide a nice interface alternative to drawing. You can add them as controls to get the styling you want, but you don't have to go to the effort of drawing them. (Or, at least, you don't have to write program code to draw them). To access bitmaps in your code, you have two alternatives: add them as **resources** to your program files or load them as separate files.

In this reading, we will provide a brief introduction to Windows drawing, then we will turn to working with bitmaps--both as resources and as files. With the capabilities introduced here, you can create quite creative user interfaces, as demonstrated by the Aquarium project.

## Introduction to Windows Drawing

Although this course has focused on developing form-based applications, if you are interested in developing interesting user interfaces, you will eventually need to learn something about drawing in Windows. There are three reasons that drawing is important:

1. There are some application types, such as mapping applications, where drawing is a critical component of the display process.
2. Any time you want your application to support printing, you are likely to need to draw the output you want.
3. Using drawing, you can often substantially improve program performance (as opposed to having Windows do all the work for you).

## Form Drawing Cycle

Windows takes the following basic approach to draw a form:

- The background of the form is drawn. This is accompanied by a **Paint** event.
- The individual controls on the form (in the **Form.Controls** collection) are drawn according to their Z-order.

Because the default form drawing is accompanied by a **Paint** event, that is where custom drawing normally occurs.

## Drawing Process

We now turn to the elements of the .NET drawing process.

### *Graphic Devices*

The process of drawing is made more complicated because there are many places to draw and many different devices to draw to (e.g., the screen, printers, bitmaps). For this reason, before we draw we need to acquire a **Graphics** object that contains all the information Windows needs to know in order to properly carry out drawing activities.

Graphic objects can be acquired in a number of ways:

- They are passed in as arguments for an event, such as **Paint** and **PagePrint**.
- They can be acquired for any Window (including controls) using the **CreateGraphics()** member. This allows us to draw on a control or our form at any time.
- They can be acquired from a **Bitmap** object using the **Graphics.FromImage()** member. This allows drawing to be done to the bitmap, rather than the screen.

Once you have a **Graphics** object, sometimes referred to as a graphics device, a huge realm of drawing possibilities become available. Shape drawing members alone include:

- DrawArc
- DrawBezier
- DrawBeziers
- DrawClosedCurve
- DrawCurve
- DrawEllipse
- DrawIcon
- DrawIconUnstretched
- DrawImage
- DrawImageUnscaled
- DrawImageUnscaledAndClipped
- DrawLine
- DrawLines

- DrawPath
- DrawPie
- DrawPolygon
- DrawRectangle
- DrawRectangles
- DrawString
- FillClosedCurve
- FillEllipse
- FillPath
- FillPie
- FillPolygon
- FillRectangle
- FillRectangles
- FillRegion

Using these, you can construct virtually any image in your code--provided you have the patience.

### *Example 1: Simple Drawing*

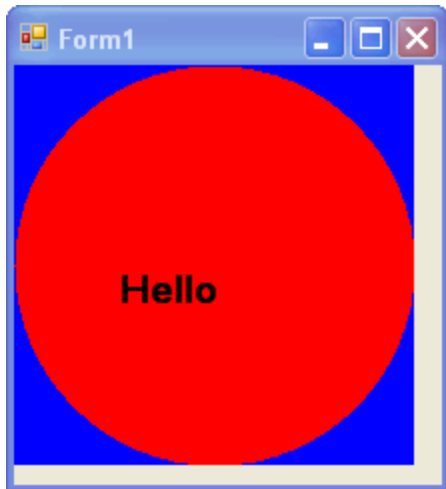
The use of graphic devices can be illustrated with some simple code. On an empty form, the Paint message was handled with the following code:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics client=e.Graphics;
    Pen p=new Pen(Color.Blue);
    SolidBrush b = new SolidBrush(Color.Blue); ;
    client.FillRectangle(b, 0, 0, 200, 200);
    b = new SolidBrush(Color.Red);
    client.FillEllipse(b, 0, 0, 200, 200);
    b = new SolidBrush(Color.Black);
    client.DrawString("Hello",
        new Font(FontFamily.GenericSansSerif, 0.2f, FontStyle.Bold, GraphicsUnit.Inch),
        b, new PointF(50, 100));
    client.Dispose();
}
```

The function gets a graphics device from the **Paint** handler argument. It then proceeds to:

- Draw a filled blue rectangle
- Draw a filled red ellipse within that rectangle
- Draw a black text string, using a generic font, saying "Hello"

The resulting output appears below:



Since the dimensions given for the ellipse (200 X 200) were equal, the resulting ellipse is actually a circle.

### ***Example 2: CaptureScreen function for printing***

The code for performing the screen capture used for printing and print preview in both the Aquarium and Bingo projects is presented below as an illustration of a drawing application related to printing:

```
private Bitmap memoryImage;
private void CaptureScreen()
{
    Graphics mygraphics = this.CreateGraphics(); 1
    Size s = this.ClientSize;
2 memoryImage = new Bitmap(s.Width, s.Height, mygraphics);
    Graphics memoryGraphics = Graphics.FromImage(memoryImage);
    memoryGraphics.CopyFromScreen(PointToScreen(new Point(0, 0)),
        new Point(0, 0), s); 3
    mygraphics.Dispose();
    memoryGraphics.Dispose();
}
```

The basic operation of the code is as follows (using the numbers):

1. *Creates a .NET graphics device from the form.* A graphics device encapsulates all the information necessary to take graphic commands (such as FillRectangle) and directs them to the appropriate display device--which could be the screen, a printer, a fax machine, etc. It can even be an area of memory that simulates a device.
2. *Creates an empty bitmap consistent with the form's graphic parameters, such as color depth.* We now have a place to hold our screen capture, once we make it. We then create a graphics device that maps to the bitmap. This gives us a way to draw to that empty bitmap, once we are ready to do so.

3. *Copies the bits from the client region of the form into our new bitmap.* `CaptureScreen()` copies bits from the screen to the graphics device it is applied to. Since it uses screen coordinates (not the form's internal coordinates, which is what we usually use), it needs to convert the upper left hand corner of the client (0,0) to screen coordinates using the `PointToScreen()` member function.

At the end of the function, the two graphics devices are released with a call to **Graphics.Dispose()**. You are advised to make a practice of doing this whenever you are done working with graphics devices you create.

The image created in this code is then fed into the `PrintPage` handler, where we see how to draw a bitmap:

```
private void printDocument1_PrintPage(System.Object sender,
    System.Drawing.Printing.PrintPageEventArgs e)
{
    e.Graphics.DrawImage(memoryImage, 100, 100);
}
```

In this case, the 100, 100 represent 1/100s of an inch, so the image is placed with 1 inch top and left margins. In this case, we don't call the `Dispose()` method on the graphics device, since that is the responsibility of the calling function (that actually created the graphics device).

## Resources

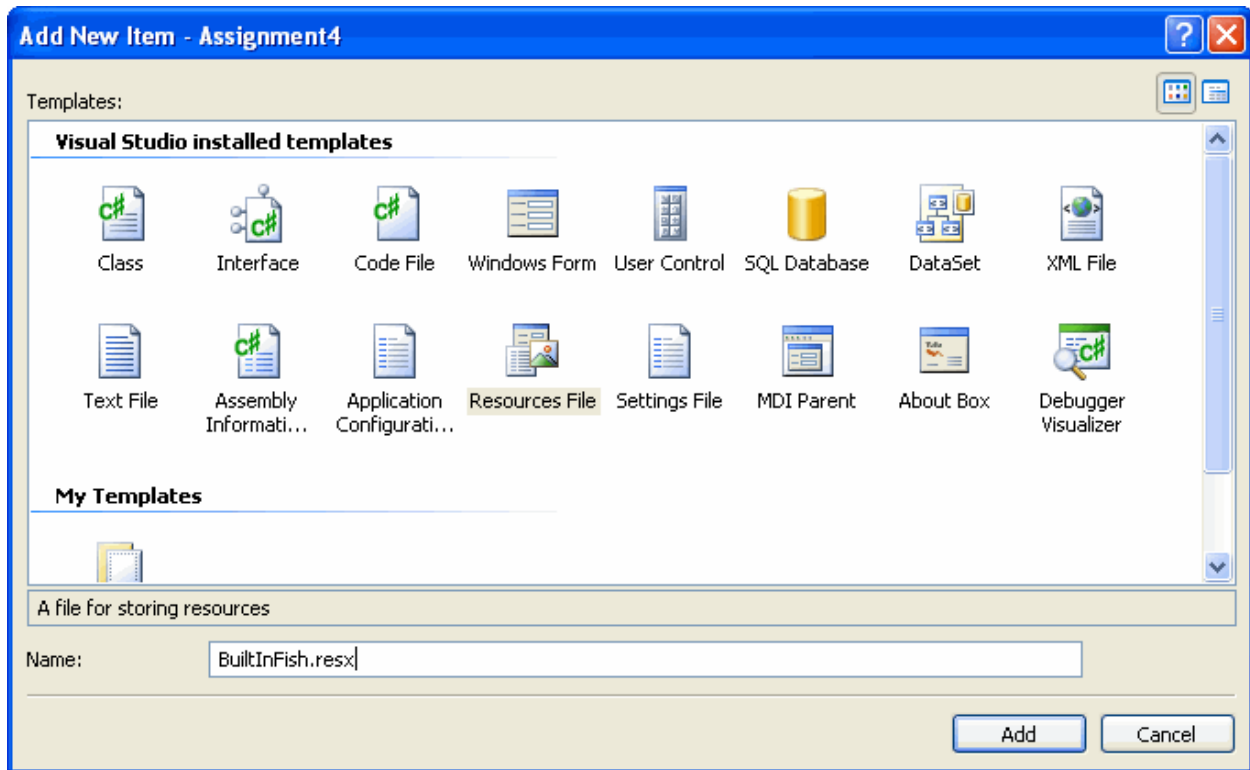
Frequently, a program needs to have access to display elements that aren't really integral to the code, but nonetheless must be present. Two examples of this type of element are:

- Bitmaps that are integral to the program, such as icons and interface elements
- Text strings--such as menu item names--that may need to be localized (e.g., translated to another language)

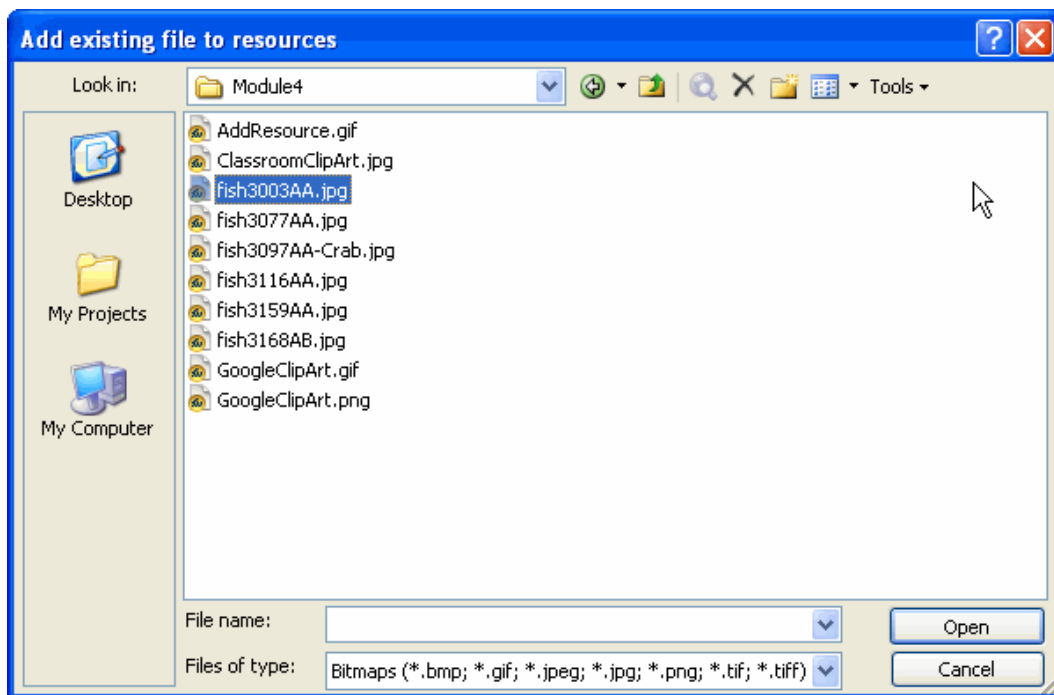
Although such elements can be included as accompanying files, this complicates the installation process, increases the number of things that could go wrong when running the program and may slow program loading. Another approach is to compile the elements into the program file itself. Such elements are referred to as **resources**.

## Creating a Resource File

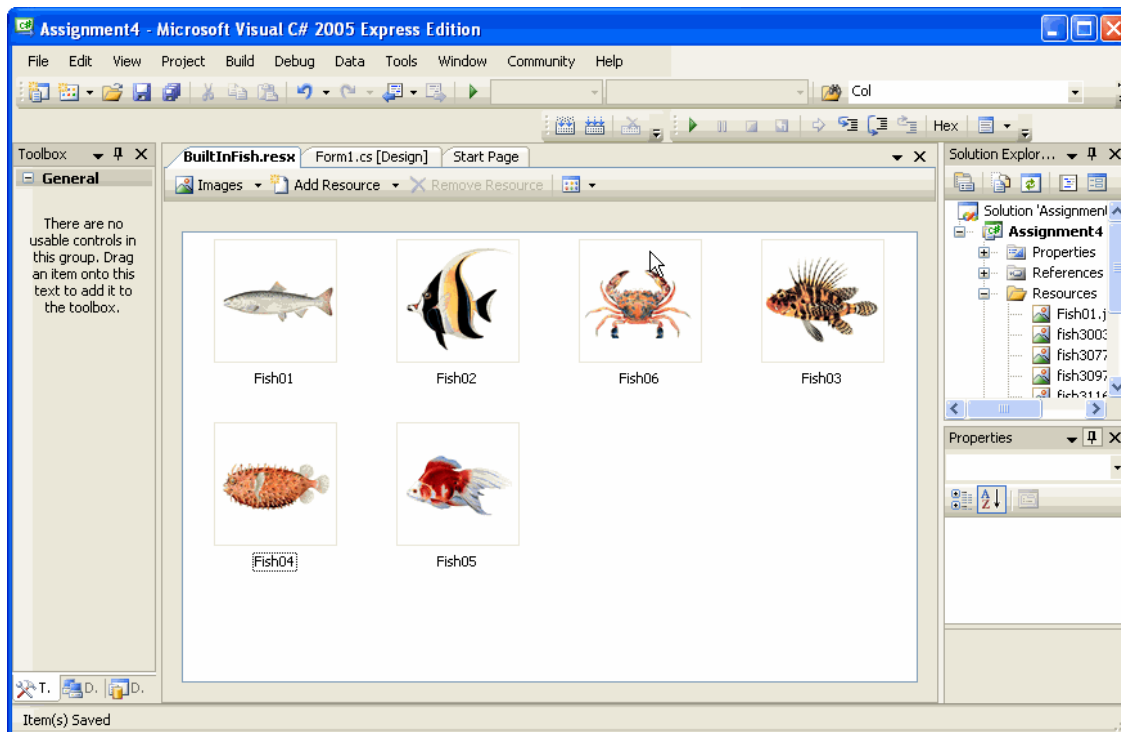
We have actually been creating resources throughout the course--every time we load a picture into a picture box or a form, it is included in the form's resource area. Sometimes, however, we want to create a special area for resources--especially if we have a lot of them. This is very easy to do. First, right click the project and `Add | New Item...`, then select `Resources File`, as shown below:



Once you've got the resource editor open, you can choose *Add existing file to resources...* and select your image.



You can add as many resources as you care to. Bitmaps are displayed in the resource area as follows:



When your code is compiled, they are included in the .exe file produced by your project.

## Using Resources in Your Code

Using the resources you've imported into your code is very simple. As shown in the code from the Aquarium project below, to use a resource you just need to supply the resource area name (the name of the resource file--in this case **BuiltInFish**) followed by the name you gave the resource when you imported it.

```
public partial class Aquarium : Form
{
    public Aquarium()
    {
        m_game = new GameSpecs();
        InitializeComponent();
        InitializeFish();
    }
    GameSpecs m_game;

    Bitmap[] builtin = {(Bitmap)BuiltInFish.Fish01,
        (Bitmap)BuiltInFish.Fish02,
        (Bitmap)BuiltInFish.Fish03,
        (Bitmap)BuiltInFish.Fish04,
        (Bitmap)BuiltInFish.Fish05,
        null, // Break in pattern signifies start of crabs
        (Bitmap)BuiltInFish.Fish06};
}
```

This particular code places our resources in an array that is later used to create **Fish** and **Crab** objects. Typecasts are required since resources are not necessarily bitmaps--they could be text strings--thus, we need to tell the compiler what they are.

## Working with Bitmaps

When we use bitmaps in our projects, we will normally be attaching them to the **Image** property of a PictureBox object, meaning we will not have to worry about the actual mechanics of drawing them (see earlier PrintPage function for an example of how to draw a bitmap directly). We need to load a bitmap from a file and copy a bitmap from another bitmap. In addition, our Aquarium project needs to establish transparency in the fish it loads.

## Creating and Copying Bitmaps

The two versions of the Fish.LoadPicture() function illustrate loading a bitmap from a file (using the Bitmap constructor function with a file name as an argument) and copying a bitmap using the Clone() member:

```
public void LoadPicture(string sFile)
{
    Bitmap bmp = new Bitmap(sFile);
    Picture = bmp;
}
public void LoadPicture(Bitmap b)
{
    Bitmap bmp = (Bitmap)b.Clone();
    Picture = bmp;
}
```

The assignment to the Picture property of the Fish object leads to the bitmap being attached to a PictureBox, using its **Image** property, as highlighted below:

```
public Bitmap Picture
{
    get
    {
        return m_picture;
    }
    set
    {
        m_picture = (Bitmap)value;
        Color bkg = m_picture.GetPixel(1, 1);
        m_picture.MakeTransparent(bkg);
        Size = ComputeSize();
        Box.Image = (Image)m_picture;
    }
}
```

## Creating Transparency

The FishPicture property code also demonstrates how to make part of a bitmap transparent. Transparency works as follows:

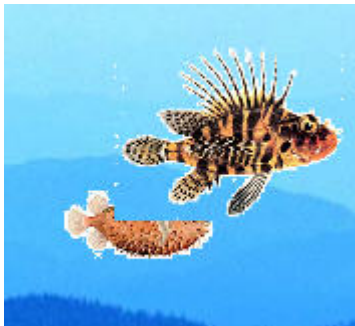
- *You define a particular color in the bitmap to be transparent.* Knowing that the fish images we are using always come on a plain background, we use a pixel in the bitmap's upper left corner `m_picture.GetPixel(1,1)` on the assumption that is it likely to be background. We then set that color to be transparent with:

```
m_picture.MakeTransparent(bkg);
```

*If that pixel is not in the background area, or if we are using a photograph which does not have a plain background, this will not work.*

- *When the form draws, it replaces the background color in the bitmap with the form background color or the appropriate portion of the form's background image.* This works okay as long as something besides form background hasn't already been drawn under the bitmap--in which case the bitmap will replace the portion of that control or image it covers with form background.

To clarify the second point, consider the following image, taken from the Aquarium project:



The puffer fish was drawn first. When the other fish (perhaps a scorpion fish?) was drawn over it, the white area from the rectangular bitmap--instead of being truly transparent--causes the form background to be drawn over the first fish.

# Polymorphism

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain the difference between polymorphic and non-polymorphic behavior
- Define a polymorphic function using the **virtual** keyword
- Define an override of a polymorphic function using the **override** keyword
- Identify the **virtual** members of the **object** class
- Explain what is meant by an **abstract** class
- Explain what is meant by an **interface**

## Overview

We have repeatedly stated that object-oriented programming languages provide three key capabilities, and have so far explored two of these:

1. *Encapsulation*: The ability to bundle related functions and data together into a single object, and control access by users of that object (e.g., through **public** and **private** keywords).
2. *Inheritance*: The ability to use one class (*base class*) as the starting point for creating another class (*child class*).

The third capability, polymorphism, is inheritance-related and revolves around our ability to override member functions and--more importantly--change what happens as a given member (e.g., function or property) is applied to a child class object.

Perhaps the easiest way to illustrate polymorphic and non-polymorphic behavior is through an example, drawn from the Aquarium assignment. In this project, two key objects are defined:

- **Fish**: An object with an associated picture that swims horizontally back and forth across the tank. Each time the fish hits the side of the tank, it reverses direction. This involves two key changes: its speed is reversed (multiplied by -1) and its picture is flipped horizontally, so that the fish picture is facing in the direction of motion.
- **Crab**: An object that inherits from fish. Its swimming pattern is vertical, however, moving from the top of the tank to the bottom and back. The reversal of direction is also accomplished by multiplying its speed by -1. A crab, however, is always upright--so it doesn't need to be flipped when it hits an edge.

A typical aquarium scene is presented below:



The Aquarium class, which inherits from Form, is where fish are "told" to swim. It includes a **FishSet** collection, **m\_fish**, which inherits from **List<Fish>** that holds all the **Fish** and **Crab** objects. The code to invoke the actual swimming is shown below:

```
foreach (Fish f in m_fish)
{
    // ...
    f.Swim(nTicks,m_game);
    // ...
}
```

Because **m\_fish** is, essentially, a **List<Fish>** object, every **f** in the loop is a **Fish** (which is okay, since a **Crab** is a **Fish** through inheritance).

Now, when we tell **Fish f** to swim, two possible behaviors are possible:

1. **Non-polymorphic**: Because the collection is declared to have **Fish** objects, all fish in the collection (whether they are **Fish** or **Crab** objects) swim like **Fish**—back and forth. *Implication*: compiler can determine what function to call, since it will always be the **Fish.Swim()** version of the function.

2. **Polymorphic:** **Fish.Swim()** is called on **Fish** objects, **Crab.Swim()** is called on **Crab** objects. *Implication:* which version of the function to call **cannot** be determined until program is running, since collection contents aren't known until then.

The compiler implications are important, because if we want to make a function polymorphic, we need to include a special hidden pointer, sometimes called a **thunk**, in each object we create that will point the program--while it is running--in the direction of the proper version of the function.

Because implementing polymorphism involves some extra data (i.e., the **thunk**) and a small performance hit (to look up the function to be called), it is not the default behavior for object members. Instead, we need to tell the compiler what we are doing. This is accomplished through two keywords:

- **virtual:** Used in a base class to specify a function member or property is polymorphic, and will be in any class that inherits from it, either directly or through its child classes (i.e., including grandchild classes, great-grandchild classes, etc.)
- **override:** Used in a child class to signify that the polymorphic version of the function/property is being overridden (e.g., the way **Swim()** was overridden in the **Crab** class). The function declaration in the child class should exactly match the return type and arguments of the base class virtual function or else a compiler error will result.

## Examples of Polymorphism

The easiest way to explain how to create polymorphic functions is to look at some code that implements them.

### Example: Fish and Crab

The code for the **Aquarium** example, just presented, is a good place to start in illustrating the use of **virtual** and **override** keywords. In this case, two functions needed to be polymorphic: **Swim()** (already discussed) and **Reverse()**--which changes the sign of the fish speed and reverses the image (**Fish**) or leaves it as is (**Crab**). Since the internals of these functions are not currently of interest, we compress the display to show the declarations only. For **Fish**, we see the two occurrences of the **virtual** keyword (highlighted).

```

namespace Assignment4
{
    [Serializable]
    public class Fish
    {
        public Fish()...
        void InitializePicture()...
        // Virtual functions
        virtual public void Swim(int nTicks, GameSpecs g)...
        virtual public void Reverse()...

        [NonSerialized]
        FishPicture m_box;
        public FishPicture Box...
        Bitmap m_picture;
        public Bitmap Picture...
    }
}

```

For **Crab**, we see the inheritance from **Fish** (circled in red) and the two corresponding occurrences of the override keyword (highlighted):

```

namespace Assignment4
{
    [Serializable]
    class Crab : Fish
    {
        public override void Swim(int nTicks, GameSpecs g)...
        public override void Reverse()...
    }
}

```

## Base class call

All the programmer action required to implement polymorphism is in these declarations lines--the code within the functions is written normally. The only exception is when the child function wants to call the base class version of the function within the child class. This is a relatively common occurrence, since child class functions often serve to enhance base class functions, rather than merely replace them (as was done for the Swim() function).

To accomplish this, the base keyword, previously seen in constructor functions, can be used. Suppose, for example, we want to define a Dolphin class that swims like a fish but, from time to time, needs to go to the surface (since dolphins, the mammal type--not mahi-mahi--need to breath). The class might look something like the following:

```

class Dolphin : Fish
{
    public override Swim()
    {
        base.Swim();
        // Additional code for occasional trips to the surface goes here
    }
}

```

Don't go ballistic yet about the fact that a dolphin isn't a fish, it is a mammal. We'll return to this example later.

## Object Class

Polymorphism is frequently encountered related to the **object** class. As we already know, every .NET object inherits from this class. And, as it turns out, two of its functions are polymorphic (i.e., declared to be **virtual**):

1. **ToString()**: Renders the object as a string. For basic types (and many common types), this function is overridden to provide a meaningful display. For example, if **pt** is a **Point** object, **pt.ToString()** might display as "(7,23)", where 7 is the **X** coordinate, and 23 is the **Y** coordinate. For many of the classes you define, there is no obvious ToString() display that makes sense. Thus, you leave the default--which displays "*Namespace.ClassName*" (e.g., "**Assignment3.Bingo**") for every object, no matter what its values are.
2. **HashCode()**: Generates a code between 0 and ~2 billion for every object, used in an algorithm called hashing--which is very useful for implementing lookup tables (more on hashing is presented in another reading). The integer HashCode() function has two useful properties: a) objects with the same value generate the same HashCode() and b) if objects are not identical, the odds are very low (about 1 in 2 billion) that their hash codes are the same.

## Example: Overriding ToString()

The **Bingo** assignment presents a scenario where overriding the **object.ToString()** member comes in very handy. Specifically:

- A **Card** object is a **List<Rows>** collection by inheritance. As we already know, such collections can be attached directly to a **DataGridView** control--eliminating days of complex interface coding.
- The **Row** object has 5 public properties--named **B, I, N, G, O**. The default behavior of the DataGridView would be to use these as column headers. This behavior is perfect from our perspective, since a bingo card has columns labeled B, I, N, G, O.
- The **Cell** object has two public properties--**Value** and **Selected**. **Value** indicates the numeric value of the cell (i.e., a number between 1 and 75). **Selected** indicates whether or not a cell has been called.

The **Cell** structure is less than ideal, since if we use it as is, when we attach a card we'll get the proper column headers but every cell will contain the text "Assignment3.Cell", which is not very useful.

We can eliminate this problem by overriding **ToString()**. A simple override might return the cell number. But we can do better. Specifically, we can change the function so it does the following:

- Displays unselected cells using their cell number, e.g., "19"
- Displays selected cells in brackets, e.g., "[73]"
- Displays a cell with the value 0 as "Free". This is nice because every Bingo board has a free cell at position 2,2 (the middle of the card).

The actual **Cell.ToString()** override is as follows:

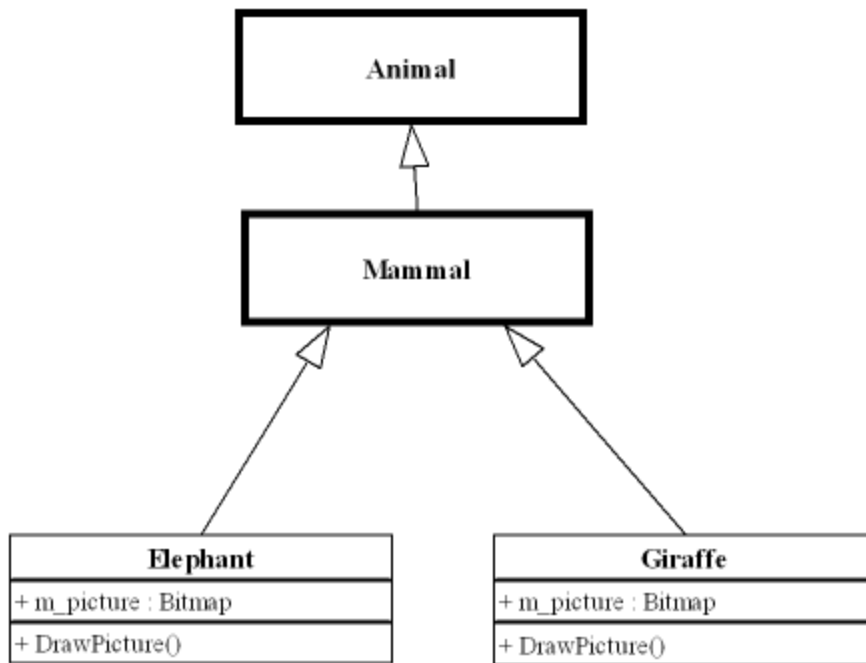
```
public override string ToString()
{
    if (Value == 0)
    {
        return "Free";
    }
    else if (Selected)
    {
        return "[" + m_val.ToString() + "]";
    }
    else return m_val.ToString();
}
```

## Abstract Classes and Interfaces

The final topic of the section deals with two important polymorphic capabilities that aren't used directly in the assignments, but are worth knowing for future programming. Both allow the programmer to develop considerably more sophisticated object hierarchies.

### Abstract Classes

Many times, we want to define a base class for different objects, but never intend to create an actual object of that type. For example, suppose we wanted to create a veldt version of the Aquarium application populated with African animals. Our UML-version of the inheritance hierarchy might include the following:



Further suppose that we wanted to draw the animals on the veldt image, the way we did with the fish swimming. The code might look something like the following:

```

List<Animal> animals=new List<Animal>();
// initialization code omitted
foreach(Animal a in animals)
{
    // code to position animal omitted
    a.DrawPicture()
}
  
```

As we design the application, we discover practical and conceptual problems. First, the code as written won't compile because the DrawPicture() function is not defined at the Animal level. So, we could move the function up to the Animal class, which solves the practical problem. But this leads to a conceptual problem: what does a picture of an animal look like? Stated another way, is there any picture I could show you that would make you point and say--"hey, that's an animal!"?

What we'd really like to do, in this case, is to define the Animal class so that it:

1. Guarantees that any Animal object we create will have a DrawPicture() function
2. Doesn't force us to define the function for the Animal class itself

This exactly describes the case where we would use an abstract class. An abstract class consists of two things:

- Member data and functions that are defined like any normal class
- **abstract** functions and properties that are declared, but not defined. These functions are implicitly **virtual** (since they will be overridden) and, if present, require the class itself be declared **abstract**.

To illustrate this, we take a simpler variation of our code above, where the Name() member is abstract, instead of the DrawPicture():

```

abstract class Animal
{
    abstract public string Name();
}
abstract class Mammal
{
}
class Elephant : Mammal
{
// Not providing an Name method results
// in a compile-time error.
    public override string Name()
    {
        return "Elephant";
    }
}

```

In this case, both **Animal** and **Mammal** are abstract--we cannot create actual **Animal** or **Mammal** objects--but we can use them to define collections (e.g., **List<Animal>**) and can call Name() on the objects in those collections, since the **abstract** declaration of Name() in the **Animal** class guarantees that any **Animal** object we *can* create *will* have a Name() member defined.

Abstract classes can be very powerful tools for organizing code. In some applications, you may have three or four layers of abstract inheritance before you ever get to a class that you can actually create.

## Interfaces

C# does not permit the object-oriented capability called multiple inheritance, which is where a child class inherits from two or more base classes. There are a lot of good reasons for leaving multiple inheritance out (it causes all sorts of problems when the two base classes happen to inherit from the same object--which they do in C# by definition). But there are times when it might be really useful.

As an example, consider our aquarium situation. Earlier in this reading, we gave an example where a Dolphin object inherited from Fish. As far as swimming behavior is concerned, it is a reasonable approximation. As far as many other elements are concerned (e.g., how it breathes,

blood temperature, giving live birth), it is much better modeled as a Mammal object. Thus, what we might like is for Dolphin to inherit from two classes, such as:

- Mammal
- SwimmingSeaDweller

Meanwhile, a Shark might inherit from:

- Fish
- SwimmingSeaDweller

This would solve our problem nicely except, as noted earlier, C# doesn't support multiple inheritance.

C# does support, however, the notion of an interface. An interface object is, in essence, an abstract class that consists of nothing but abstract functions and properties. For example, our SwimmingSeaDweller class might be redefined as an interface along the following lines, using the **interface** keyword:

```
interface SwimmingSeaDweller
{
    void Swim();
}
```

By their very nature, all **interface** functions and properties are both virtual and abstract, so we don't need to specify that. Since you can inherit from a single base class and as many interfaces as you want (just list the base class first, and the interfaces afterwards, separated by commas), we could now define Dolphin and Shark as follows:

```
class Dolphin : Mammal, SwimmingSeaDweller {...}
class Shark : Fish, SwimmingSeaDweller {...}
```

This solves our problem--so long as both Dolphin and Shark implement their own version of the Swim() function (just as they would have to do when inheriting from an **abstract** class).

With the interface thus defined we could, for example, create a **List<SwimmingSeaDweller>** collection and call Swim() on every object.

Within the .NET framework, interfaces are used extensively in controls and collections. These can easily be identified because Microsoft always begins their name with a capital I. For example, **List<>** template implements the **IList** and **ICollection** interfaces, among others.

Similarly, the VCEE help system describes classes that can be bound to a **DataGridView** control as follows:

The **DataGridView** control supports the standard Windows Forms data binding model, so it will bind to instances of classes described in the following list:

- Any class that implements the [IList](#) interface, including one-dimensional arrays.
- Any class that implements the [IListSource](#) interface, such as the [DataTable](#) and [DataSet](#) classes.
- Any class that implements the [IBindingList](#) interface, such as the [BindingList](#) class.
- Any class that implements the [IBindingListView](#) interface, such as the [BindingSource](#) class.

Knowing these two facts, we can deduce that a **List<>** object can be attached to a **DataGridView** control.

# Timers (Module 4)

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain why a timer may be needed
- Add a timer control to a form and set its event interval
- Start and stop the timer
- Respond to timer events

## Overview

Event-driven programming is just that: writing code that responds to events by handling them. Unfortunately, this style of programming makes doing something at regular intervals difficult. Unless we make the user click the mouse once a second, how do we write code that does something every second?

The obvious solution is to use a **Timer** control. Such a control generates events at regular intervals that we can then respond to in our code with an event handler.

Actually, the timer concept is not new to object-oriented programming. Computers have employed clock-generated **interrupts**, the low level equivalent of an event, for five decades or more--as long as there have been operating systems. These events force the computer to do things like scan the keyboard and check for data in input-output ports. On a PC, these clock interrupts occur every 1/18th of a second. This figure is relevant since these hardware clock interrupt events also trigger timer events.

In this reading, we look at how to use a timer control to animate activities on the screen, as implemented in the Aquarium project.

## Timer Control

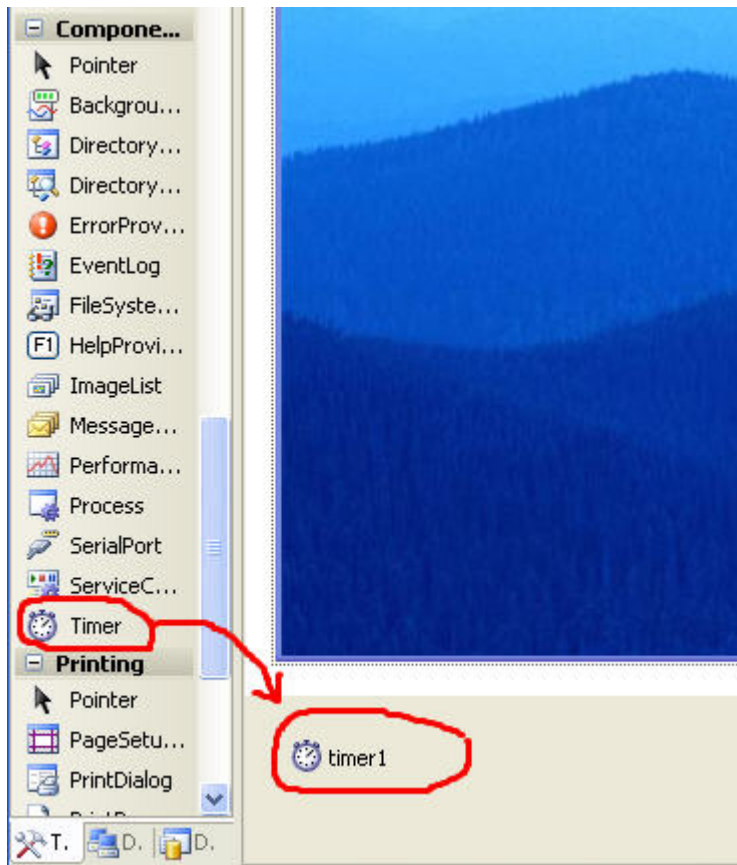
.NET supports the Timer control as a means of generating regular events.

## Setting Up a Timer Control

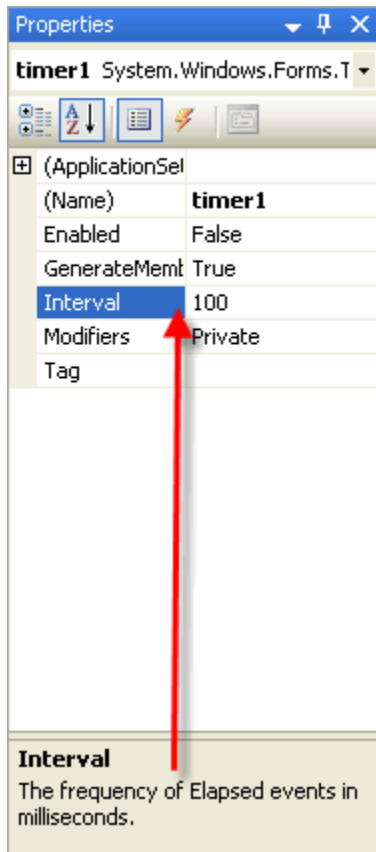
The timer control is the simplest control we've encountered in the entire course. First, we're only interested in three of its members:

- **Interval:** A property defining time between **Tick** events, specified in milliseconds (1/1000s of a second).
- **Start():** Starts the timer running.
- **Stop():** Stops the timer from generating events.

Secondly, it is simple in that you always do the same thing with it. You begin by dragging it on to the form, as shown below. Since it is a non-displaying control, it appears in the grey box under the form.



Next, you set its interval--which is measured in 1/1000 second intervals. The precision available is somewhat misleading, however, since it is triggered by the PC clock every 1/18th of a second--so every interval you specify is going to be a rounded multiple of 1/18. Setting the property is shown below:



In this case, the interval is 1/10th of a second, which will probably end up being 1/9th of a second for most ticks.

Finally, you double click the control to create the handler for the Tick event.

## Example Timer Handler

Handling the timer event is at the heart of the Aquarium code. The function is presented below:

```

private void timer1_Tick(object sender, EventArgs e)
{
    if (nTicks >= m_game.GameLength)
    {
        1 timer1.Stop();
        MessageBox.Show("Your final score is " + m_game.TotalPoints.ToString());
        return;
    }
    2 bool bIncrease=false;
    nTicks++;
    3 if (nTicks > 0 && nTicks % m_game.SpeedIncreaseInterval == 0)
    {
        bIncrease = true;
    }
    SuspendLayout();
    4 foreach (Fish f in m_fish)
    {
        if (f.Visible) m_game.TotalPoints=m_game.TotalPoints+m_game.PointsPerTick;
        f.Tank = this.ClientRectangle;
        f.Swim(nTicks,m_game);
        if (bIncrease) f.Speed = (int) (f.Speed *(1+ m_game.SpeedIncreasePercent));
    }
    ResumeLayout();
}

```

In this function, we use a member variable, **nTicks**, to keep track of how many times the event has been generated. This gets incremented each time the handler is called. The code works as follows:

1. If the number of ticks has reached the number specified to end a game, we stop the timer and pop up a message box with a score.
2. We increment the number of ticks, so we can keep track of how long the game has been going on.
3. The **GameSpecs** class has a parameter that allows Fish speed to be increased after a specified number of ticks (the **SpeedIncreaseInterval**). If we reached an even multiple of that interval--found by taking the remainder of a division by that interval with the % operator, we set a flag variable (**bIncrease**) to true.
4. For every visible fish in our collection, we reset the tank size (in case the user resized the screen), tell the fish to Swim() (which will move it in the proper direction by **f.Speed** units), and--if it's time to increase speed--we multiply the **Speed** property of each fish by a speed increase percent--also a **GameSpecs** value--then assign that back as the new **Speed**.

## Stopping and Starting Timer

The code for stopping and starting the timer is placed in a menu item handler. If the game is currently running, it stops the timer. If it is not running, it starts the timer.

```

bool m_running;
private void button1_Click(object sender, EventArgs e)
{
    m_running = !m_running;
    if (m_running)
    {
        if (timer1.Interval != m_game.TickDuration)
            timer1.Interval = m_game.TickDuration;
        timer1.Start();
        ((ToolStripItem)sender).Text="&Pause";
    }
    else
    {
        timer1.Stop();
        ((ToolStripItem)sender).Text = "&Play";
        MessageBox.Show("Your current score is " + m_game.TotalPoints.ToString());
    }
}
}

```

Perhaps most interestingly, this code changes the text of the menu item that generated the event. Thus, if the timer has been stopped by the event, the item becomes "Play". If the timer has been started by the event, the menu item becomes "Pause".

# Randomization and Hashing

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by pseudorandom numbers
- Use the Random() class to generate different types of random numbers
- Describe the role played by the seed in randomization
- Explain the basic idea behind the hashing algorithm
- Use hash codes to generate random number seeds

## Overview

Random numbers are numbers whose values fall within a specified range without any underlying pattern. As such, they are generally impossible to obtain, although some approaches (e.g., timing the frequency of atomic decay) may come pretty close. We can, however, come up with *pseudorandom* numbers fairly easily. These are numbers generated by some algorithm--hence they can't be truly random--that have no discernable patterns. For most purposes, pseudorandom is random enough--and we'll refer to them as random for the rest of the reading.

Random numbers have many uses in programming. They are, for example, used extensively in:

- Simulations, where they can be used to model situations too complex to solve with equations.
- Games, where they add an element of chance and produce game experiences that vary each time a game is played.
- System level programming, where they can be used to avoid useless cycling (e.g., if two packets collide in a network and need to be resent, you don't want both senders to try again at exactly the same interval--otherwise they will keep colliding forever. Instead, you can add a random element to the retry interval).

Random numbers are typically generated in two forms, *seeded* and *unseeded*. By way of explanation, pseudorandom generation algorithms typically need a value to get them started--known as the **seed**.

- **Seeded.** Every time the same seed is supplied, the same sequence of random numbers is generated. In many cases, we want the same sequence, so we supply the seed. This would, for example, allow us to compare different strategies in a business simulation against the same simulated environment.
- **Unseeded.** Since our algorithm needs a seed to start, there is no such thing as an unseeded sequence of numbers. We can, however, base our seed on something that is continuously changing, such as the system clock. We call such a sequence unseeded, since every time we generate a sequence, it is likely to be different.

Although not directly related, an algorithm called *hashing* provides a nice complement to our discussion of randomization. A central aspect of the hashing algorithm is the generation of a hash code, which is very similar to a random number in its properties. A hash code is always seeded, however. The seed can be any object--not just a number. Although we'll discuss the hashing algorithm in this reading--since it is very important in programming--our principal use of hash codes in this course will be to generate appropriate random number seeds.

## Random Class

In .NET, random numbers are generated using the **Random** class. It can be used to generate seeded or unseeded sequences of random integers, bytes or double objects (with values between 0 and 1.0). The interface of the class is relatively trivial in nature, and includes the following:

- Two constructor overloads
  1. `Random()`: Generates a sequence of values using the system clock to supply a seed (essentially unseeded)
  2. `Random(int seed)`: Generates a sequence with a user-supplied seed
- A variety of overloads to extract data. These include:
  1. `int Next()`: Returns a random integer between 0 and ~2 billion
  2. `int Next(int ub)`: Returns a random integer between 0 and *ub*-1
  3. `int Next(int lb,int ub)`: Returns a random integer between *lb* and *ub*-1.
  4. `double NextDouble()`: Returns a random real number between 0.00 and 1.00

For example:

```
Random r1=new Random(); // Generates unseeded sequence  
Random r2=new Random(1256); // Generates sequence seeded with 1256  
int val1 = r1.Next(); // Generates random number between 0 and ~2 billion  
int val2 = r2.Next(100); // Generates random number between 0 and 99  
r1=new Random();  
r2=new Random(1256);  
int val3=r1.Next() // Almost certainly different from val1  
int val4=r2.Next(100); // Same as val2, since same seed, same position (1st)
```

The assignments provide some more useful examples.

### Example 1: Generating Bingo Card

To implement our Bingo application, we need to be able to generate random cards for players to use at will. This needs to be done on a column by column basis, with the following constraints:

- Each column is generated from a set of 15 values {1..15}, {16...30}, {31...45}, {46...60}, {61...75}

- 5 values are needed for each column (the middle column only really needs 4, since the middle value is "Free", but we can generate 5 then overwrite the middle element)
- Values within a column cannot be duplicated--meaning that we are dealing with a situation that can be called *sampling without replacement*.

We can simplify this generation somewhat by noticing that if we can generate values for column 1, we can make these suitable for other columns by adding an appropriate amount (15 for I, 30 for N, 45 for G and 60 for O). Thus, we'll focus on writing code to generate values in the 1 to 15 range.

The non-duplication problem is harder to address, since there's a reasonable chance that if you generate 5 random numbers in a row, you'll get at least one duplication. One way we can deal with this is to put the possible values {1...15} in a collection and then, every time we choose a value, remove the corresponding element from the collection. Moreover, using **Random.Next(int ub)** overload, we can use the **Count** of the collection to determine our upper bound (*ub*), then treat the random number as a coefficient, instead of a value. That is the approach taken in the code below:

```
static Cell[] GenerateCol(Random rand)
{
    Cell[] col=new Cell[5];
    List<int> available=new List<int>();
    // Create an array with elements 1 through 15
    for (int i = 1; i <= 15; i++)
    {
        available.Add(i);
    }
    for (int i = 0; i < 5; i++)
    {
        int Max = available.Count;
        int coeff = rand.Next(Max);
        col[i] = new Cell(available[coeff]);
        available.RemoveAt(coeff);
    }
    return col;
}
```

In that code:

- The yellow highlight is where we generate our set of cell values {1...15}.
- In the green highlight, we generate a random number that will be used as an index (**coeff**).
- In the next line, we assign the value from our collection at that index (i.e., **available[coeff]**) to our next column position.
- The light blue highlight is where we remove the element from the collection. This has the effect of making Max smaller by 1 the next time the loop executes.

## Example 2: Generating Fish Speed

In the Aquarium assignment, we randomize the speeds at which our fish initially swim. The challenge here is that we want ranges from negative top speed to positive top speed. Since the **Random.Next(int ub)** value is limited to positive numbers, we take twice the top speed as our upper bound, leading to a value between 0 and 2\*top speed-1, then subtract top speed, leading to a range from - top speed to + top speed -1. The code is as follows:

```
int speed = rand.Next(m_game.TopSpeed * 2) - m_game.TopSpeed;  
if (speed == 0) speed = 10; // We don't want dead fish...
```

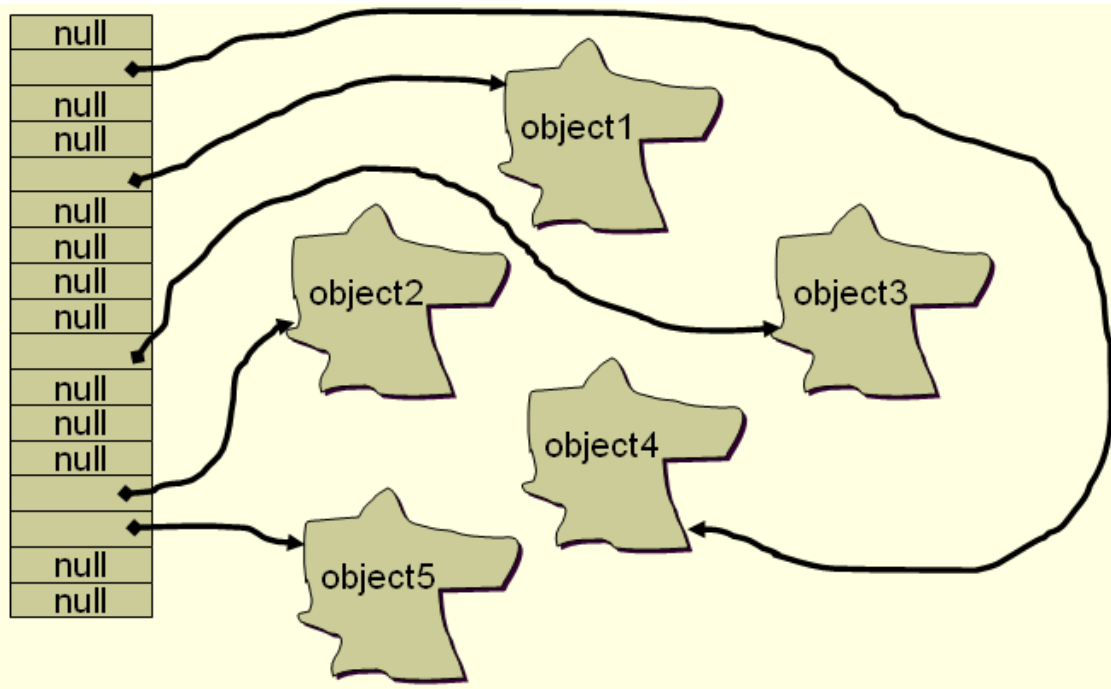
## Hashing

Hashing describes an algorithm frequently used in lookup functions. In its simplest version, using a string as a lookup key, the algorithm works along the following lines:

- An array is used to store the objects in the collection. In general, the size of the array is substantially larger than the collection it will be used to store, so there are many empty cells.
- Object placement is determined by the hash code of the lookup key string, which is very much like the first value coming out of a seeded random number sequence. The difference here is that the key itself--not an integer value--is used to generate the code. (In .NET, a virtual **Object.GetHashCode()** function will generate an appropriate value for any type of object, not just strings). The hash code can then be adjusted for table size by dividing the hash code by table size and taking the remainder (i.e., code % table-size).
- To retrieve an object, we use the same lookup key (e.g., the key string) to generate the same hash code. By looking at that location in the table, we can determine if the object is there and, if it is, we can retrieve it.

## Illustration

Conceptually, you can think of a hash table as shown in the following diagram:



As you can see, the hash table has many empty spaces (null values) and objects referenced by locations that have been determined by hashing the lookup key associated with each object. If these were student data objects, for example, we might use the NETID as our lookup key--as long as it is unique.

The hashing algorithms used in the real world have a lot of added twists--mainly dealing with the fact that in any reasonably sized table, some objects are going to produce the same hash code (remember, table size determines how many possible codes there can be), referred to as collisions. In this course, however, we'll be using hash codes for another purpose, so we'll focus on the codes, not the algorithm. This does not mean, however, that you can't implement a lookup table in your programs. Many .NET collections (such as **Dictionary**) use hashing to implement their collection shape.

### **object.GetHashCode()**

The GetHashCode() function is one of two virtual functions--the other being ToString()--that is implemented in the **object** base class. Thus, every .NET object can be used to generate a hash code.

To review, the GetHashCode() function has the following characteristics:

- Produces a value between 0 and ~2 billion for any object
- Although many objects "can" have same code, same object (e.g., a particular string) always produces same code
- Chances of two different objects having the same code are roughly 1 in 2 billion

The characteristics of the `HashCode()` function make it nearly ideal for generating Bingo cards. The scenario is this:

- We want (nearly) every player to have a different card
- We want a player to be able to recreate his/her card without having to store it in a database
- We want the caller to be able to verify a "Bingo" claim by a particular player

Naturally, we could do this by assigning each player integer seeds for each card. This would involve everybody carrying around ten digit integers for each card. Moreover, the caller would have to keep track of who was given each seed.

Much more elegant is to use the `HashCode()` function to generate a seed from a particular seed. To generate the `HashCode()` we can use some easily recreateable key, such as:

- Blackboard ID "-" Game Name
- Blackboard ID "-" Game Name + "-" Board Name (allowing a player to have multiple boards in the same game)

Based on the `HashCode()` properties, such keys will mean:

- Every time the game changes, the boards generated will be different for every player (OK, 1 in 2 billion chance two players get the same board)
- The caller can verify a bingo by knowing the player's ID and the name of the game
- We can recreate cards at will

This is built into the **Card** class using the constructor functions, as follows:

```
public Card() 1
{
    Random rand = new Random();
    GenerateCard(rand.Next());
}
public Card(int seed) 2
{
    GenerateCard(seed);
}
public Card(string key) 3
{
    int seed = key.GetHashCode();
    GenerateCard(seed);
}
void GenerateCard(int seed)
{
    Random rand = new Random(seed);
    Row[] rows = new Row[5];
```

The constructors are as follows:

1. The first constructor overload is our "unseeded" version, since it uses the **Random()** overload to produce a clock-seeded number that we use as our seed
2. A seeded constructor, where we supply the seed
3. A constructor which uses our key string to generate a `HashCode()`, which is then used to seed the random number generator

In all three cases, the seed generated in the constructor is passed into `GenerateCard()`, where (highlighted) it is used to start the random number generator we'll be using to create cards.

# Form Containers (Module 3)

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by a container control
- Use a tab control on a form
- Use a splitter window on a form
- Use a toolbar container on a form

## Overview

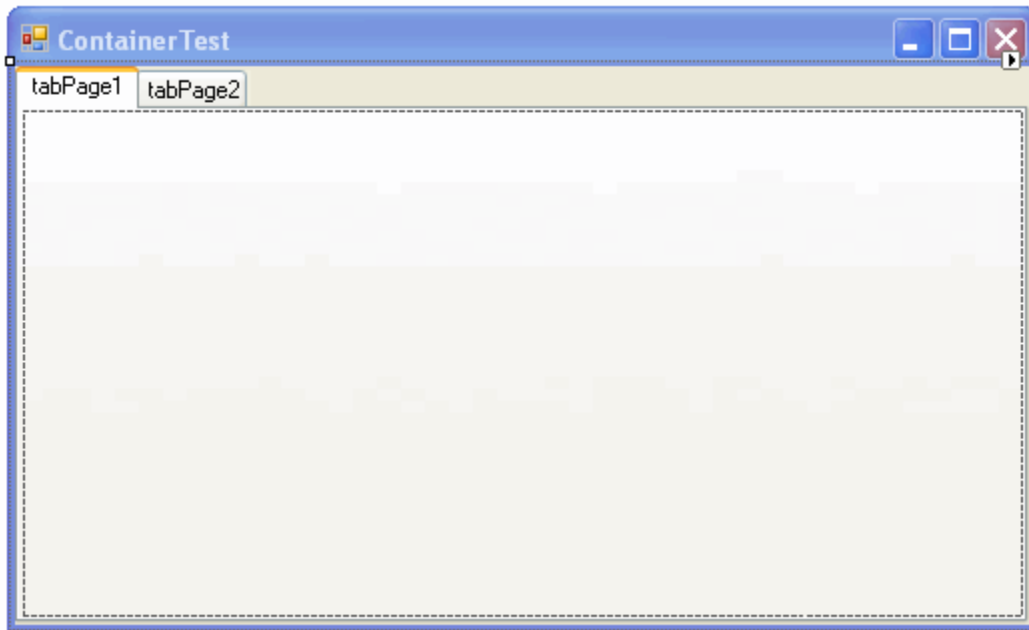
Form containers are very simple controls that can be very powerful in organizing forms. Conceptually, you can think of a form as being a drawing area (known as the client area) and a collection of controls (the **Form.Controls** collection) that reside on top of the drawing area. Most containers provide an alternative area on which to put controls. The benefits of adding a container between the form and other controls can include the abilities to:

- Hold multiple layers of controls that can be accessed like separate forms but which share the same data members and interrupts (e.g., the **TabControl**)
- Establish separate docking areas for ease of size management (e.g., **SplitContainer**)
- Rearrange controls as a window resizes, the way web pages often do (e.g., **FlowLayoutPanel**, **TableLayoutPanel**)
- Establish docking areas for menus and toolbars (e.g., **ToolStripContainer**)

All of these containers are very flexible and can be used in many ways. We'll focus on three simple ones: the **TabControl**, **SplitContainer** and **ToolStripContainer**.

## TabControl

The **TabControl**, shown below, allows multiple control layers to be established in the same form. These layers can be selected by the user within the program.



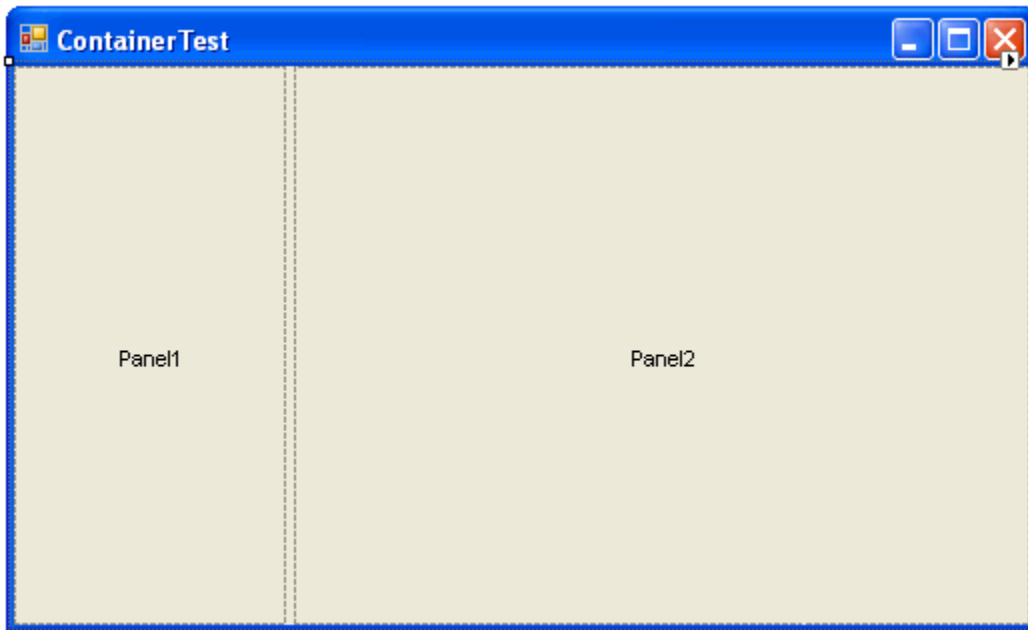
A **TabControl** is different from separate forms in that all the layers of the control share the same access to form data and other controls. This can be a major advantage over moving data back and forth from multiple forms.

Nearly every setting in a **TabControl** can be adjusted either through the properties menu or within the program. Tabs can be added and deleted and their properties changed. The two most commonly used members within a program are:

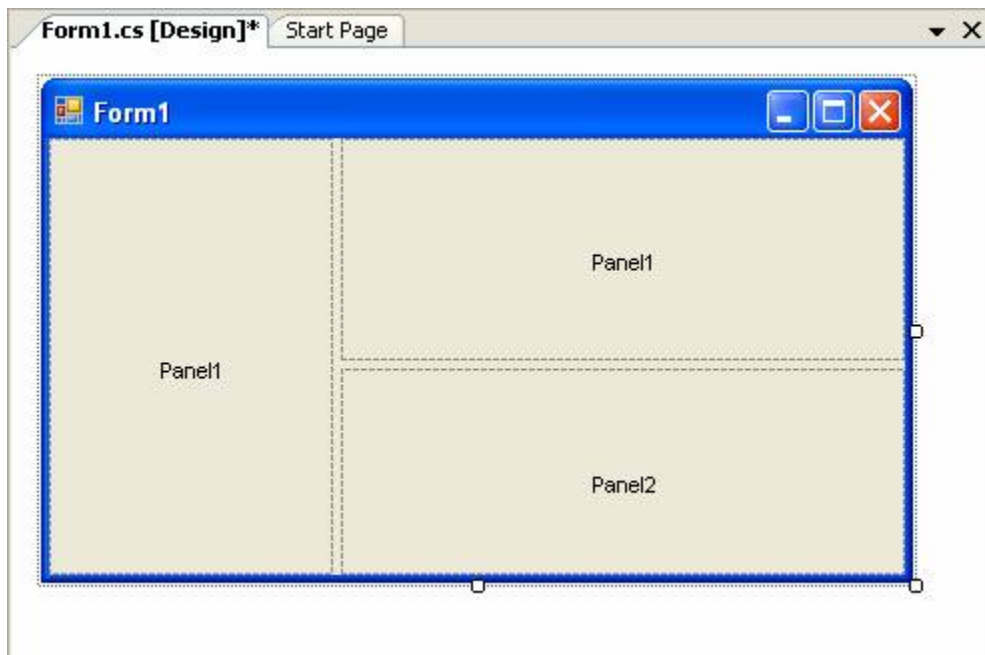
- **SelectedIndex**: read-only identifies active tab
- **SelectIndex(int)**: selects active tab

## Splitter Window

The **SplitContainer** control (also known as the splitter window) is the easiest possible control to use. As shown below, it has a vertical or horizontal splitter bar that, effectively, makes each side of the bar act like a separate form.



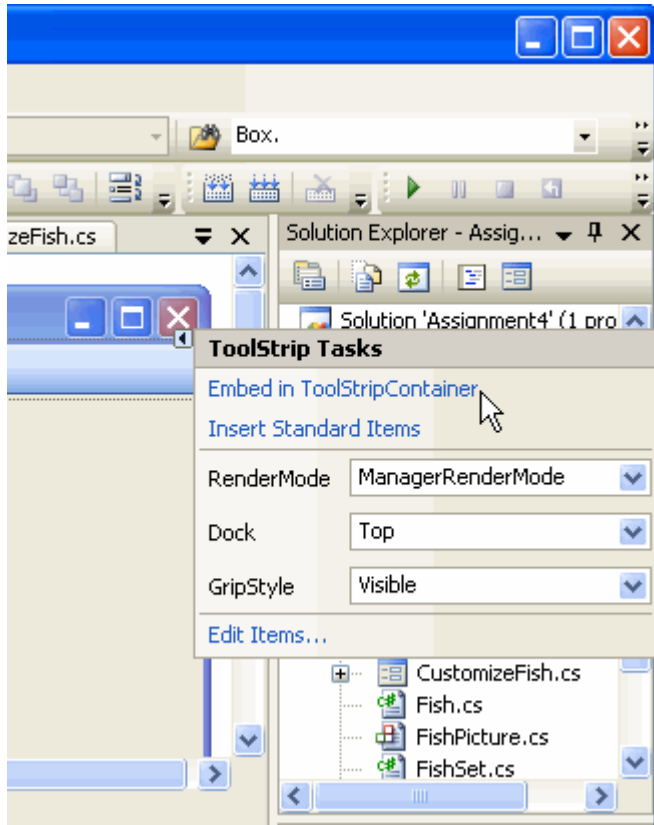
**SplitContainer** controls tend to be particularly useful in situations where you want to dock multiple controls within the form, but don't want the whole client area taken up. They can also be nested--embedded within each other--if more elaborate splits are desired, as shown below:



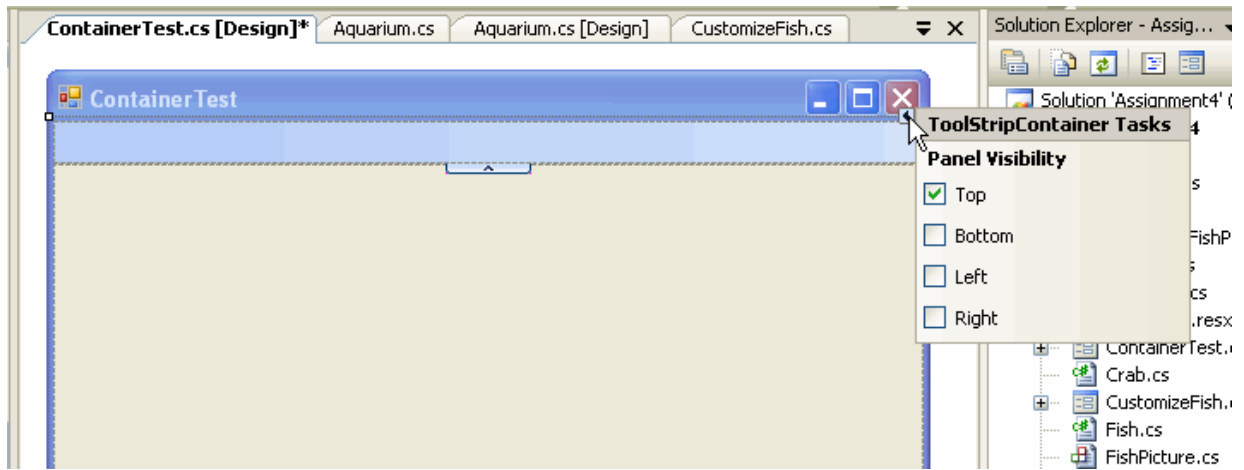
**Orientation** is the only splitter property typically used. It determines if the bar is horizontal or vertical.

## ToolStripContainer

The toolstrip container provides an area onto which dockable toolbars and menus can be dropped. Many new Microsoft applications (such as PowerPoint) make extensive use of this control to allow separate controls or views to exist around the edge of the form). The control is most commonly encountered when adding a menu or toolbar, as shown below:



For typical menus, you probably want to leave only the Top option selected in the dialog that follows. Leaving all the options checked means that dockable controls can end up on any side of the form. Unless your application is highly complex, this is probably overkill.



When a ToolStripContainer has been inserted on your form, it has an annoying habit of disappearing and becoming very hard to delete in the designer. It is important to be aware of the handles (at the bottom edge of the form strip above) that can be used to hide and expose the container. Once a container is in place, multiple tool strips can be added. When your program is running, the container will expand--as necessary--to hold the controls (e.g., tool strips, menus) that you place in it.

# Menus

## Learning Objectives

Upon completing this reading, you should be able to:

- Add a menu to a form
- Populate the menu with standard items
- Create a dockable toolstrip

## Overview

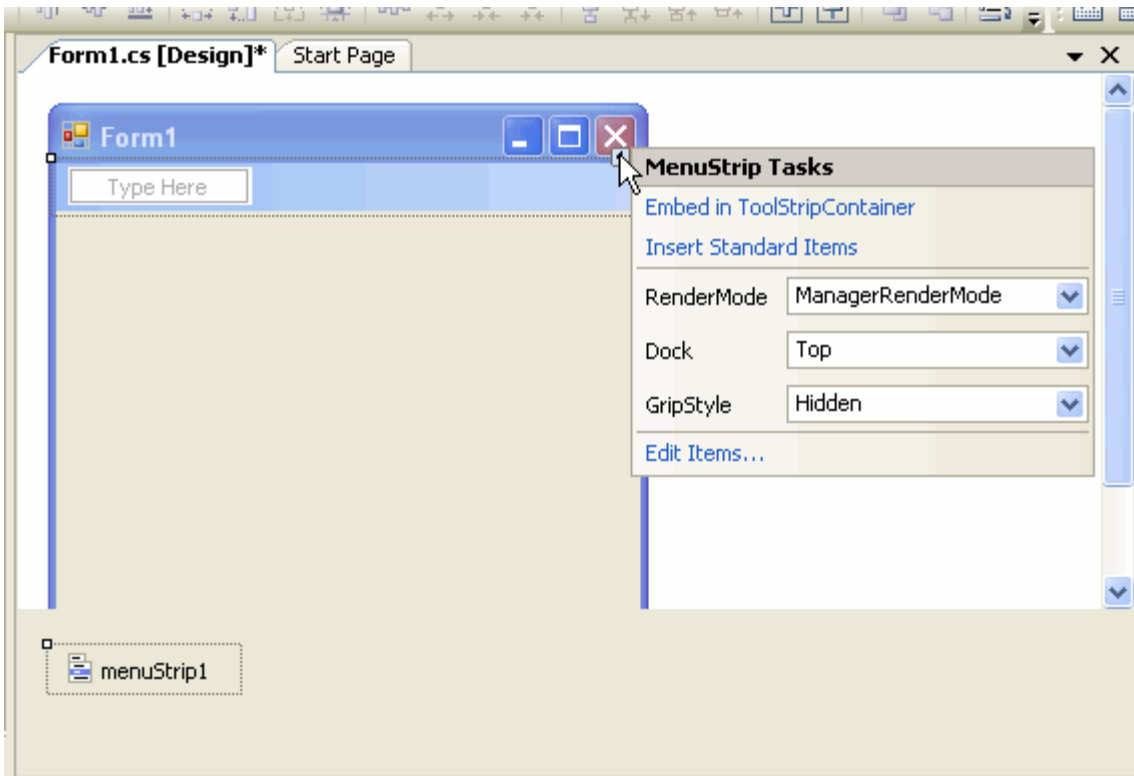
As your applications become larger, you will typically want to use menus to organize your forms, since using buttons and other controls tends to clutter up the forms. It is also important for ease of use to create an interface with which users are familiar. For most windows applications, this means conforming to the standard user interface, which usually includes the following menu items:

- **File:** Contains major functionalities for moving documents to and from serialization and to the printer. Includes options such as **New, Open, Save, Save As, Print, Print Preview** and **Exit**.
- **Edit:** Contains options for cutting, copying, and pasting data within the application.
- **Tools:** Contains application-specific options, as well as a place to customize application settings.
- **Help:** Includes Topics (table of contents-style access), index, search and **About** functionality (to display version and serial numbers).

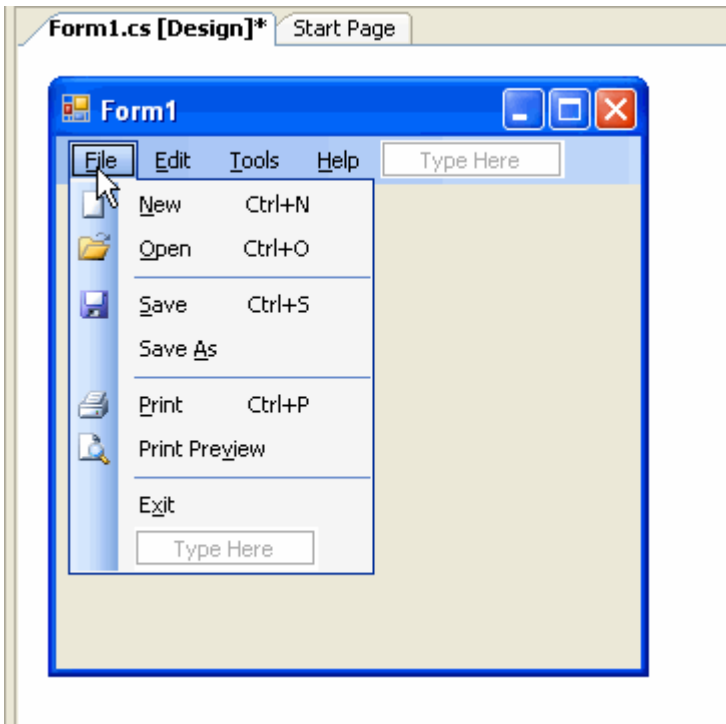
The .NET framework 2.0 provides some very nice tools for generating menus. In addition, the toolstrip functionality--used to accomplish many of the same things previously done with menus, allows a clean, modern interface to be developed relatively painlessly.

## Menus

Not a great deal can be said about menus. You drag them on to your form, as shown below, then add items as needed.



The common tasks allow you to embed your menu in a toolstrip container automatically so it can be docked and undocked. A real time saver is the "Insert Standard Items" task, which populates the menu with the standard File|Edit|Tools|Help options used by most applications, as shown below:



By double clicking a menu item, you can create a handler. Sometimes menu items share handlers with other controls or menu items. In this case, you can go to the properties menu to get a list of available handlers to choose from. This was done in the code below, used in the Aquarium project, which was originally attached to a button and then later attached to the "Play" menu item.

```
bool m_running;
private void button1_Click(object sender, EventArgs e)
{
    m_running = !m_running;
    if (m_running)
    {
        if (timer1.Interval != m_game.TickDuration)
            timer1.Interval = m_game.TickDuration;
        timer1.Start();
        ((ToolStripItem) sender).Text = "&Pause";
    }
    else
    {
        timer1.Stop();
        ((ToolStripItem) sender).Text = "&Play";
        MessageBox.Show("Your current score is " + m_game.TotalPoints.ToString());
    }
}
```

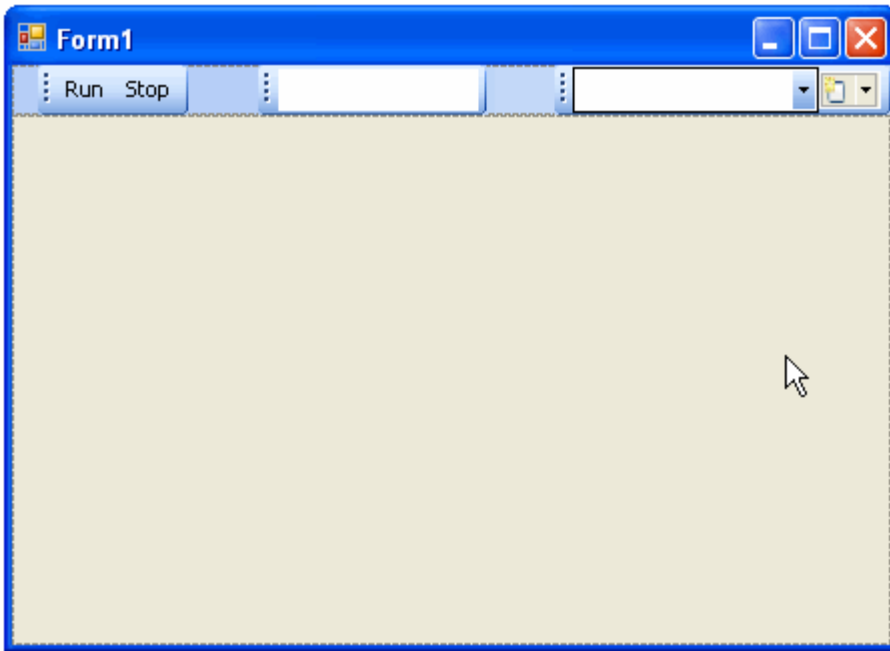
This code also indicates how menu item text can be changed. When the game is running in the above example, the menu item's text is set to "Pause". When the game is not running, the item is set to "Play". The & produces an underscore in the designer, which is used to signify a speed key to access the option.

For more advanced menu control, handlers can be established for the event of opening a menu display. Then, before the display appears, you can add, remove, rename, enable or disable the items--as appropriate for the application. This code, while relatively straightforward, is beyond the scope of this course.

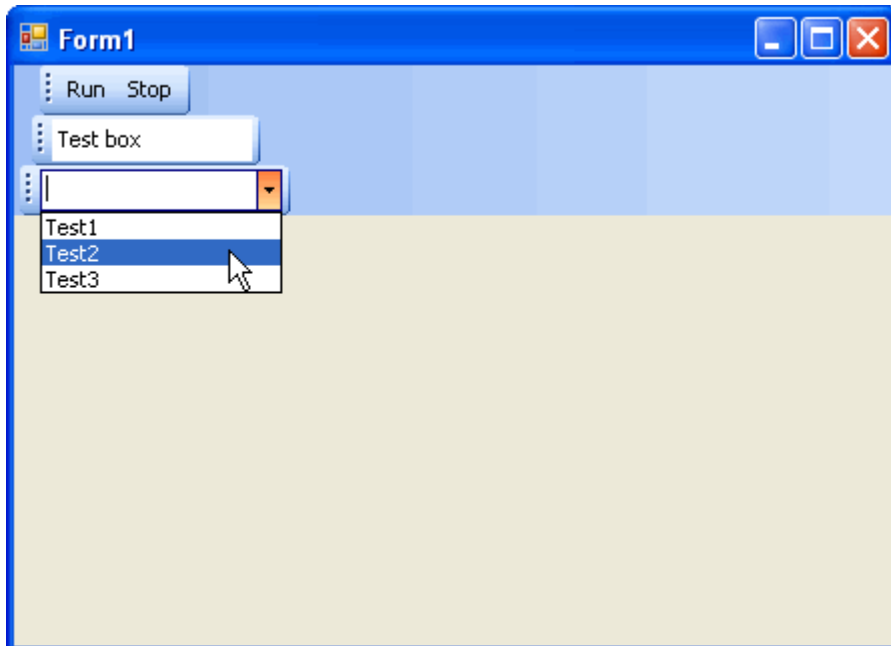
## Tool Strips

ToolStrip controls are similar to menus, except they don't open up the same way and are more flexible in terms of the types of controls that can be easily inserted (easily is the operative word here, since a good programmer with enough time can make a control do almost anything).

Toolstrips, combined with ToolStripContainer controls, provide extraordinary ability to dock and rearrange controls while a program is running. For example, the designer view of the control, shown below, places three separate toolstrips in one container:



When the form is run, these controls can be totally rearranged by simple user dragging, as shown below:



It is most impressive that these capabilities required no programming whatsoever to implement. Writing the code to respond to the controls, on the other hand, will (naturally) involve the programmer...

# DataGridView II

## Learning Objectives

Upon completing this reading, you should be able to:

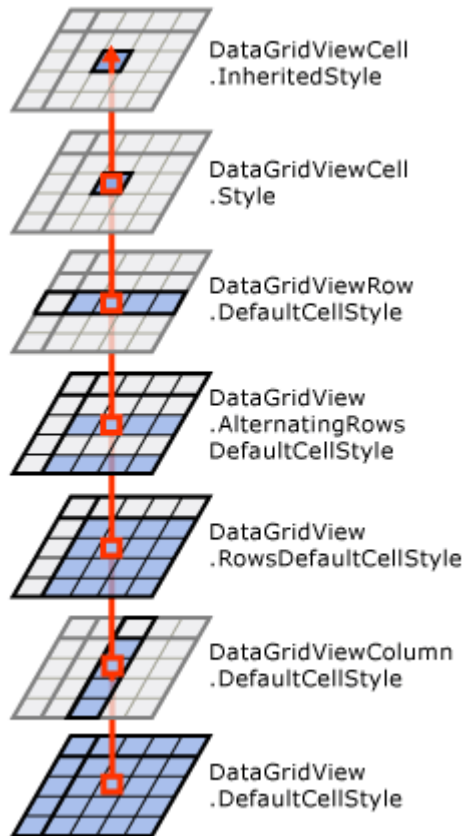
- Explain the process by which styles are set in a **DataGridView** control.
- Attach an object to a **DataGridView** control that includes pictures
- Modify row formatting for a **DataGridView** control
- Access individual cell data and formatting within a **DataGridView** control

## Cell Styling in a DataGridView

The styling for a **DataViewGrid** can be set in a lot of places. In fact, the styling for an individual cell can be established in six different places:

1. Grid's default cell style
2. Default style for the column
3. Default style for all rows
4. Default style for alternating rows
5. Default style for specific row
6. Style set for the specific cell

This process is managed through an inheritance-style process. The actual styling of a given cell is determined by its **InheritedStyle** property, which is private and can't be touched. The settings for the 6 styles listed above determine the value for **InheritedStyle**. The order of precedence is shown below (in a diagram taken from .NET Help):



This can be interpreted as follows:

- If the default cell style for the form is set (which it is, by default) and no other styles are set, that becomes the cell's style.
- If a default column style is set (e.g., in the column editor), it overrides the default cell style.
- If a default row style is set--which applies to all rows--it overrides the column style.
- If an alternating row pattern is set, alternating row styles override general row styles.
- If a specific row's default style is set, it overrides any previously set row styles.
- If a specific cell's style is set, it overrides all other styles that have been set above.

If you are planning to do much styling of a **DataGridView** control, you would do well to be aware of this hierarchy, since you might not expect that, for example, the cell styles you set for a row make any column style settings irrelevant.

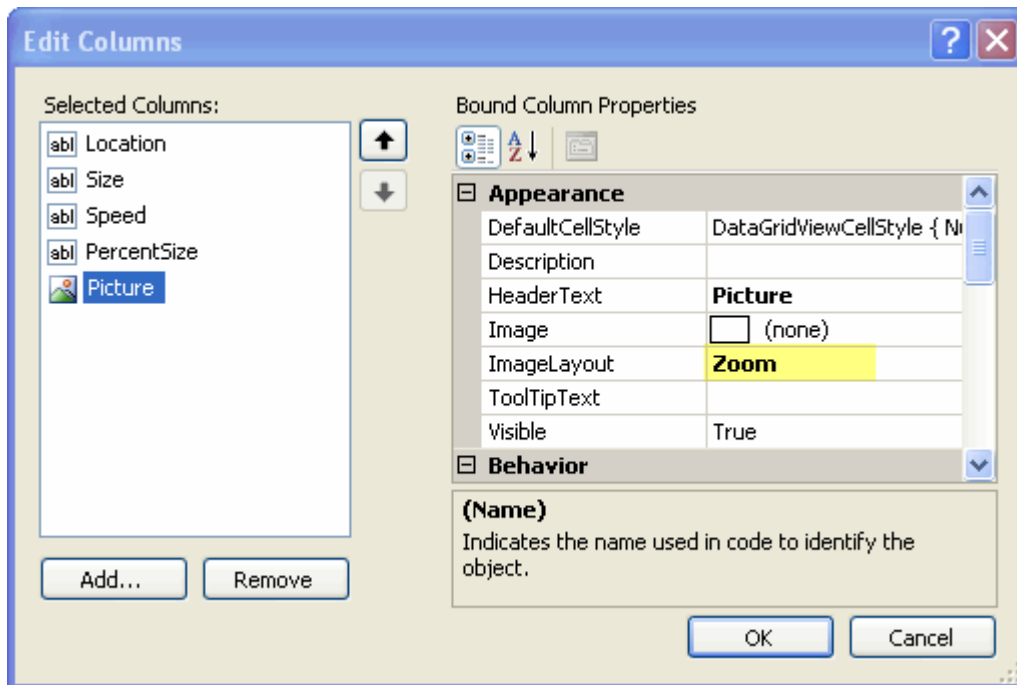
## DataGridView Formatting

To illustrate **DataGridView** formatting, we'll consider how formatting can be set through the designer for columns and default row. Then, on the next page, we'll look at setting styles in the C# code.

## Column Styling

We're already used to the "Edit Columns" common task for the **DataGridView** control (also accessible with a right mouse click) to change column order. It can also be used to change column styling. A good example of this can be found in the **Aquarium** project, where we need to edit our list of fish, including a picture box.







Some important settings--most notably image mode--can be set for a column to make our picture display more appropriately. In the column editor--which adjusts the available properties based upon the type of data displayed in the column--this appears as follows:



By setting this value to Zoom, we see the entire picture, as shown below:

CustomizeFish

Add Fish Delete Fish

	Location	Size	Speed	PercentSize	Picture
▶	333, 335	115, 57	-6	0.2	
	183, 164	98, 90	1	0.2	
	133, 27	115, 86	32	0.2	
	502, 264	76, 34	16	0.2	
	592, 219	115, 68	-43	0.2	
	357, 206	115, 75	1	0.2	

Naturally, many column properties--such as text styling--can be edited in this manner. You should remember, however, that column styling sits near the top of the inheritance chain. This means that your settings could easily be overridden by other styling settings.

## Row Styling

Row styling can also be edited in the designer. This is done in the **RowTemplate** property of the **DataGridView**. This, for example, would be a good place to set row height--which is useful for both the Aquarium and Bingo projects.

RowHeadersWidth	41
RowHeadersWidthSizeMode	EnableResizing
RowsDefaultCellStyle	DataGridViewCellStyle { }
RowTemplate	<b>DataGridViewRow { Index=-1 }</b>
ContextMenuStrip	(none)
DefaultCellStyle	<b>DataGridViewCellStyle { }</b>
DividerHeight	0
ErrorText	
Height	<b>60</b>
ReadOnly	False
Resizable	NotSet
ScrollBars	Both
SelectionMode	RowHeaderSelect
ShowCellErrors	True
ShowCellToolTips	True
ShowEditingIcons	True

Alternating row styles can also be set in the designer. For more specific styles (e.g., specific row, cell styles), styling will normally have to be changed in program code--since the specific data (i.e., rows, cells) that you will be displaying is established when the program is running, not at design time.

## Setting Styles in C# Code

The process of setting a style property in C# is pretty much identical to setting any other property. The key issues the programmer faces are:

- Getting access to the specific column/row/cell for which the styling is to be set
- Accessing the specific style property

In many cases, its easier to use the autocompletion feature to determine what is available than to look it up in the help system.

## Example: Setting Selected Cell Colors

Although not actually required by the assignment itself, the Bingo project provides a nice scenario to illustrate the use of cell styles. In that project, selected pieces on a Bingo card are indicated by placing brackets around the piece (e.g., 19 displays as [19] when selected). There is no reason, however, that we couldn't also change the cell color when the user clicks on it, to highlight the selection more visibly. The code to accomplish this is written as follows:

```

private void dataCardView_CellContentClick(object sender,
    DataGridViewCellEventArgs e)
{
    int row = e.RowIndex;
    int col = e.ColumnIndex;
    DataGridViewCell cell = dataCardView.Rows[e.RowIndex].Cells[e.ColumnIndex];
    if (current.Select(col, row))
    {
        cell.Style.BackColor=Color.Maroon;
    }
    else cell.Style.BackColor = Color.White;
    dataCardView.Update();
    if (current.Bingo()) MessageBox.Show("Bingo!");
}

```

In this code, we first get the value of the **cell** variable by using the **row** and **col** of the user's click, then accessing the cell by choosing the selected row in the **dataCardView.Rows** collection, then selecting the appropriate column within the **Cells** collection of that row.

We next call the **Select()** member of the bingo **Card** class, which toggles the selection. (For this demonstration, the **Select()** function was slightly modified so, instead of returning void, it returns the cell's selection setting). Using this, we access the background color for the cell through **cell.Style.BackColor** and set it to **Maroon** (an element in the **Color** enumeration) if selected, **White** if not (highlighted). The resulting card appears as follows:

Player	Caller	B	I	N	G	O
		[7]	21	35	48	73
		1	[19]	37	58	69
▶		2	29	Free	[59]	64
		4	22	43	57	63
		5	24	34	52	75

Note how the selected cells (indicated with []'s around the number) are now also colored in.

# Serialization

## Learning Objectives

Upon completing this reading, you should be able to:

- Explain what is meant by serialization
- Identify how serialization differs from accessing a database
- Differentiate between binary and XML serialization
- Prepare a class for opening and saving using the file menu
- Identify key issues relating to binary serialization and describe how to overcome them

## Overview

**Serialization** is the process of moving application data to persistent storage, such as a hard disk. The "serial" in the term is intended to convey a sequential process of moving a data stream in large blocks (e.g., saving an entire document) as opposed to saving chunks of data when needed (e.g., writing records to a database). When used more generally, serialization can also refer to movement in both directions (e.g., opening and saving documents). **Deserialization** is normally reserved for reading from data streams (opening).

Serialization, when implemented in most applications today, uses a **binary** format for saving and loading data. Such a format stores data in a sequence of bytes very similar to the manner in which it is held in memory. This both minimizes the processing involved in saving and loading and reduces the size of the resulting files. That situation is starting to change, however. **XML**, a text-based standard similar to HTML in format but specifically intended for transferring data (as opposed to displaying documents), is increasingly being used for serialization purposes. Although larger and more processing intensive, XML documents are much more transportable between systems, and more suited to open source applications by virtue of their readability.

.NET offers built-in serialization capability for many objects. As a general rule, getting an object to serialize/deserialize is relatively trivial provided it meets two criteria:

1. All of its members are serializable
2. Its constructor function is not too complex

The first of these stems from the fact that .NET knows that to serialize an object to binary format, you must serialize all of its member data (**public** and **private**). As a result, it has the built-in ability to construct the code to do just that. The second is a result of the fact that binary deserialization bypasses the constructor functions when creating objects from storage. If the constructor function is needed to do important things, the deserialized object may not work properly. Both of these issues can be addressed with a certain amount of extra programming work. They are also not as applicable to XML serialization, which only serializes public members and does not bypass the constructor.

The process for serializing an object involves three basic steps:

- A special formatting object (e.g., BinaryFormatter, SOAPFormatter) must be created, based on what type of serialization is intended.
- A file stream must be opened suitable for writing (serialize) or reading (deserialize).
- The data must be transferred using the formatting object to move data between the object being saved/opened and the file stream being written/read.

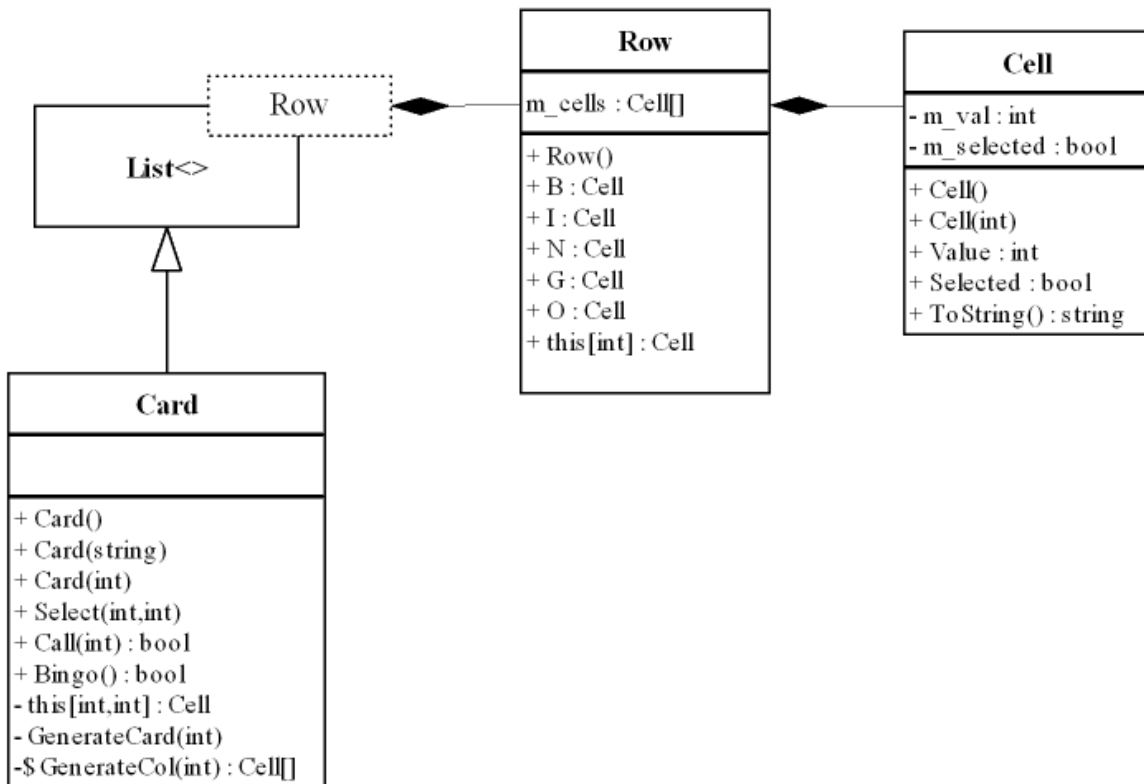
This reading will now present some examples of how this can be implemented in C#.

## C# Serialization

The only way to learn serialization is to look at some code examples, since no one would be able to come up with the specific sequence of classes and actions required by guessing.

### Bingo Card Serialization

The Bingo project provides a good example of "plain vanilla" serialization. Although the application uses a database to store caller-related data, the **File | Open** and **File | Save** options are specifically provided to load/save a single card. To do this, two objects need to be saved: a **Game** object which holds the game name and user name (which can be used to recreate an empty version of the card) and a **Card** object, which includes the collection of **Rows**, each of which, in turn, contains a collection of **Cells**--each of which has a **int Value** (the number in the cell) and a **bool Selected** property (identifying if it has been called). The UML diagram for these classes is shown below:



### Preparation

Identifying all the classes involved is important as *each* must be marked with the [Serializable] attribute, as illustrated below (highlighted) for the Cell class:

```

namespace Assignment3
{
    [Serializable]
    public class Cell
    {
        public Cell() { }
        public Cell(int v) ...
        int m_val;
        bool m_selected;
        public int Value ...
        public bool Selected ...
        public override string ToString() ...
    }
}
  
```

## Serialization

Once all four classes--**Game**, **Cell**, **Row**, **Card**--have been marked serializable, the next step is to implement the **File | Save** and **File | SaveAs** functions. Since **Save** and **Save As** can use the same serialization code, we'll only discuss the latter:

```
private void saveAsToolStripMenuItem_Click(object sender, EventArgs e)
{
    FileName = null;
    saveToolStripMenuItem_Click(sender, e);
}

private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (FileName == null)
    {
        if (this.saveFile.ShowDialog() == DialogResult.OK)
        {
            FileName = saveFile.FileName;
        }
    }
    if (FileName != null)
    {
        // IFormatter is in System.Runtime.Serialization
        // BinaryFormatter() is in System.Runtime.Serialization.Formatters.Binary
        1 System.Runtime.Serialization.IFormatter fmt =
            new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
        2 // Stream and FileStream are in System.IO
        System.IO.FileStream strm = new FileStream(FileName, FileMode.Create,
            FileAccess.Write, FileShare.None);
        3 fmt.Serialize(strm, m_game);
        fmt.Serialize(strm, current);
        strm.Close(); 4
    }
}
```

The *File | Save* handler (**saveToolStripMenuItem\_Click**) begins by opening a **SaveFileDialog** object to get the file name, if one is not available. Otherwise it moves right into serialization, by the numbers:

1. Creates a **BinaryFormatter()** object. In truth, finding the namespace where the class was contained (not shown in the .NET sample code) was probably the hardest thing about writing the function--which is why it is fully specified in the example code.
2. Opens a file stream suitable for writing (**FileMode.Create** and **FileAccess.Write** combine to open a file for writing, erasing any existing file that is there).
3. The **fmt** (formatter) object transfers data first from **m\_game** (the **Game** object) to **strm** (the file stream object), then from **current** (the **Card** object) to **strm**.
4. The file stream is closed, to keep from locking out other users or other uses of the file in the program.

## Deserialization

Implementing deserialization (the **File | Open** menu item) is very similar to serialization. This can be seen in the code for deserializing the game information and card, shown below:

```
private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (this.openFile.ShowDialog() == DialogResult.OK)
    {
        FileName = openFile.FileName;
        // IFormatter is in System.Runtime.Serialization
        // BinaryFormatter() is in System.Runtime.Serialization.Formatters.Binary
        System.Runtime.Serialization.IFormatter fmt =
            new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
        // Stream and FileStream are in System.IO
        System.IO.FileStream strm = new FileStream(FileName, FileMode.Open,
            FileAccess.Read, FileShare.None);
        Game = (Game)fmt.Deserialize(strm);
        Current = (Card)fmt.Deserialize(strm);
        strm.Close();
    }
}
```

Significant differences between the serialize and deserialize code are highlighted. These can be explained as follows:

- **FileMode.Open** and **FileAccess.Read** open a file--which must exist--for reading. Otherwise the file opening code is the same.
- **fmt.Deserialize** is used instead of **fmt.Serialize**, consistent with the objective of the function. Since we need to get an object out of this action instead of sending one into the file, the function has only one argument--the file stream.
- A typecast is needed to move the deserialized object into the memory variables. The **fmt.Deserialize()** returns an **Object**, which can be typecast to anything. Based on the order we used to serialize, however, we know the object we just opened is first a **Game** object and second a **Card** object.

It is critical that the data in Save/Open handlers be serialized and deserialized in the same order. Remember, the file contains binary data. This data has no distinctive markings or tags (like XML) to tell the program what is being loaded during an open operation. That means it is critical that the data be positioned as expected.

## Customized Serialization

If you try to serialize an object containing (or inheriting from) a non-serializable object, you will get an exception. This proves to be a problem in the Aquarium project, because the **Fish** object has a composed **PictureBox** object and graphic controls are not serializable. Moreover, important initializations of that **PictureBox** occur in the **Fish** constructor--which is bypassed during serialization. Thus, we have two problems to address.

## Problem 1: Non-serializable members

.NET Binary serialization writes every data member to the stream, **public**, **protected** or **private**. If an individual member is not to be serialized, a **[NonSerialized]** attribute can be placed over its declaration in the class, as shown in green below. This has the effect of skipping it during the serialization process.

```
[Serializable]
public class Fish
{
    public Fish()...
    void InitializePicture()...
    // Virtual functions
    virtual public void Swim(int nTicks, GameSpecs g)...
    virtual public void Reverse()...

    [NonSerialized]
    FishPicture m_box;
    public FishPicture Box...
    Bitmap m_picture;
    public Bitmap Picture...
    double m_percent;
    public double PercentSize...
```

## Problem 2: Bypassed constructor function

The **Fish** constructor's normal behavior is to create a new picture box (**FishPicture** inherits from **PictureBox**) and initialize it. When we open a file using binary serialization, that constructor is bypassed. That means we have a **null m\_box** member sitting around after we open the file. This can't be good.

A solution to this, shown below, uses the **Box** property to control access to **m\_box**. Since we've made sure that our code always accesses the picture box by using **Box**, it follows that we can initialize the object the first time it is accessed--through the **get** construct. Thus, as highlighted in the code, we put the same constructor code in the **Box** property code as we had in the constructor. If **Box** is accessed and **m\_box** is null, we construct and initialize the box before we return it!

```

public Fish()
{
    m_box=new FishPicture(this);
    InitializePicture();
}
void InitializePicture()...
// Virtual functions
virtual public void Swim(int nTicks, GameSpecs g)...
virtual public void Reverse()...

[NonSerialized]
FishPicture m_box;
public FishPicture Box
{
    get
    {
        if (m_box == null)
        {
            m_box = new FishPicture(this);
            InitializePicture();
            if (m_picture!=null)
            {
                Picture=m_picture;
            }
            m_box.Location = Location;
        }
        return m_box;
    }
}
}

```

These two modifications solve the two major serialization obstacles presented by the **Fish** class. Other issues had to be addressed (e.g., removing old fish objects and adding new fish objects to the **Aquarium** form when a file is opened), but these are application-specific. We now have our data in a state such that essentially the same code used for the bingo card can be used to perform the actual load and save of our fish collection.

# Printing

## Learning Objectives

Upon completing this reading, you should be able to:

- Identify the obstacles to printing
- List the key events in the .NET printing process
- Create the handlers used to implement printing
- Create a print preview of a document for which printing has been implemented

## Overview

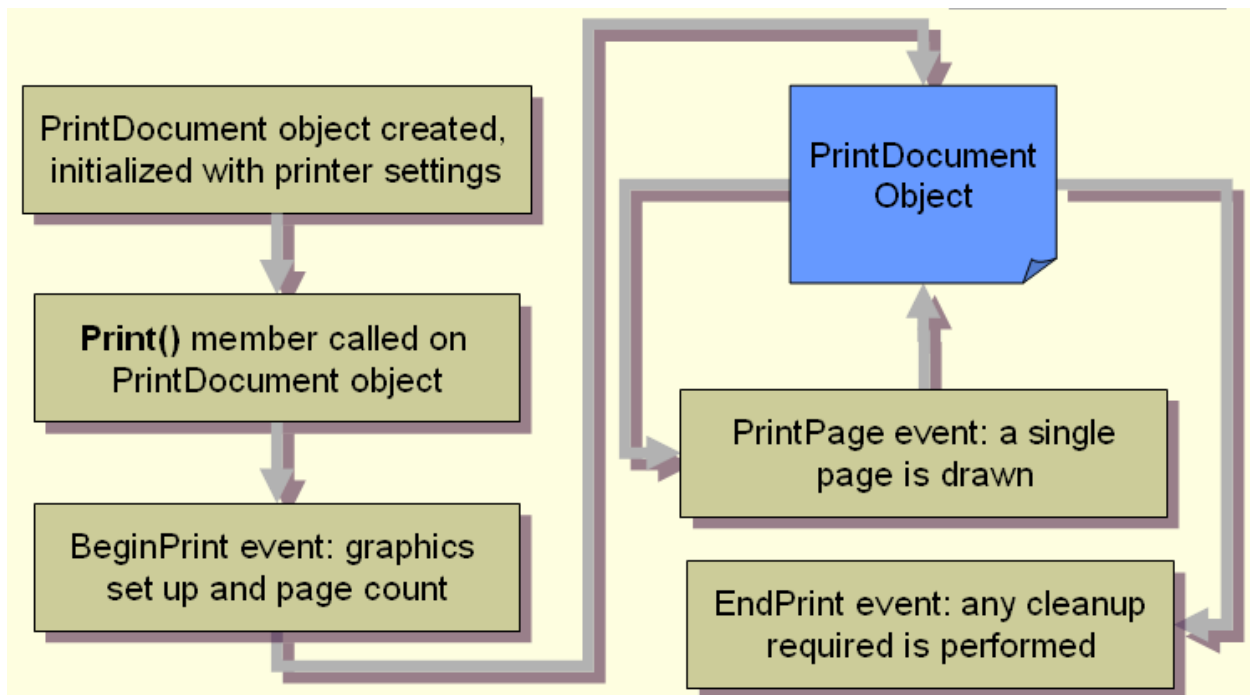
Printing can be a lot of work; there's no point in sugarcoating it. The printing process presents a number of obstacles:

1. *Printers require drawing.* Since no built-in controls get sent to the printer, everything has to be drawn. This is a lot harder than dropping a control on a form.
2. *Printers paginate.* When a screen display gets too large, scroll bars appear--as if by magic. In printing, it is up to you to compute where pages start and end and come up with some reasonable method of handling jobs too large for a single page.

A lot of the time, you can bypass the need to support application printing using a number of tricks, such as: a) using third party controls that handle printing well, such as Crystal Report Writer or the Spread control, b) sending your data to applications that do printing well, such as MS-Access, or c) writing your data out in a format (such as HTML or rich text format) that is easy to edit and reformat in another application. If (a) through (c) aren't feasible, then you are stuck with it--you'll need to implement printing.

## .NET Print Cycle

When you print in .NET, you are not entirely unsupported. Specifically, .NET offers a number of objects (e.g., **PrintDocument**, **PrintDialog**, **PrintPreviewDialog**) and events that can be helpful in the printing process. The overall approach used by .NET is diagrammed below, in no particular standard (i.e., it's not UML!):



This can be explained as follows:

- Before printing, a **PrintDocument** object should be created. It should then be formatted and associated with a particular printer. This is most easily done by attaching it to a **PrintDialog** object, which allows the user to make selections that are set within the print document.
- The actual printing sequence occurs when the `Print()` member of the **PrintDocument** object is called. This begins the generation of print events (which the print document object also handles).
- The first event in the printing sequence is a **BeginInit** event, which can be used for things like computing page count and initializing graphic settings.
- A sequence of **PrintPage** events is then generated. A **PrintPage** handler must be associated with the document. This handler should check the page number being printed, then draw the appropriate information using the printer device. This loop stops when the final page is reached.
- An **EndInit** event is called when the page limit has been reached. The **PrintDocument** object can handle this to do cleanup, if required.

If this seems complicated, it is. Fortunately, all you really need to do--to produce a one page, thoroughly unprofessional document--is to handle the **PagePrint** method. That will be our limited objective.

## C# Form Printing

To implement a basic print of the form from our applications (the same for both Aquarium and Bingo), we will take an approach based on code provided in the .NET samples. Since there is no

obvious way to print .NET controls available, we make a screen capture of the form, then send that capture to the printer using the normal printing sequence. We begin with the screen capture.

## ScreenCapture

The code for performing the screen capture is presented below:

```
private Bitmap memoryImage;
private void CaptureScreen()
{
    Graphics mygraphics = this.CreateGraphics(); 1
    Size s = this.ClientSize;
2 memoryImage = new Bitmap(s.Width, s.Height, mygraphics);
    Graphics memoryGraphics = Graphics.FromImage(memoryImage);
    memoryGraphics.CopyFromScreen(PointToScreen(new Point(0, 0)),
        new Point(0, 0), s); 3
    mygraphics.Dispose();
    memoryGraphics.Dispose();
}
```

The code's basic operation is as follows (using the numbers):

1. *Creates a .NET graphics device from the form.* A graphics device encapsulates all the information necessary to take graphic commands (such as FillRectangle) and directs them to the appropriate display device--which could be the screen, a printer, a fax machine, etc. It can even be an area of memory that simulates a device.
2. *Creates an empty bitmap consistent with the form's graphic parameters, such as color depth.* We now have a place where we can hold our screen capture, once we make it. We then create a graphics device that maps to the bitmap. This will give us a way to draw to that empty bitmap, once we are ready to do so.
3. *Copies the bits from the client region of the form into our new bitmap.* CaptureScreen() copies bits from the screen to the graphics device it is applied to. Since it uses screen coordinates (not the form's internal coordinates, which is what we usually use), it needs to convert the upper left hand corner of the client (0,0) to screen coordinates using the PointToScreen() member function.

At the end of the function, the two graphic devices are released with a call to **Graphics.Dispose()**. You are advised to make a practice of doing this whenever you are done working with graphic devices you create.

An alternative to a screen capture of the complete form might be to use bitmaps of individual controls, then draw these individually to the printer device. Since most controls have a **DrawToBitmap()** member implemented, this task would not be unreasonable. The challenge then becomes to decide how to implement layout (matching the screen layout or specific to the printer) and pagination. One approach might be to create a page collection--which could be established in the **BeginPrint** handler--then use that collection to render pages individually in the **PrintPage** handler.

## Printing the Capture

Once we've captured our screen object, the remaining printing tasks are less intimidating, and more like a normal printing job. As we mentioned, there are two handlers we need to write: one to respond to **PrintPage** events (attached to the **PrintDocument** object) and another to handle our *File / Print* menu item. To help us with this, .NET allows us to drag a **PrintDocument** control and various print dialog controls on to the form, so those objects (**printDocument1** and **printDialog1**) are already available for us.

The code for the two handlers is shown below:

```
private void printDocument1_PrintPage(System.Object sender,
    System.Drawing.Printing.PrintPageEventArgs e)
{
    e.Graphics.DrawImage(memoryImage, 100, 100); 1
}
// End of code copied from samples

private void printToolStripMenuItem_Click(object sender, EventArgs e)
{
    Update(); 2
    printDialog1.Document = printDocument1; 3
    CaptureScreen();
    if (printDialog1.ShowDialog() == DialogResult.Cancel) return; 4
    printDocument1.Print(); 5
}
```

This code can be explained as follows, by the numbers:

1. Calls **DrawImage()** within the **PrintPage** handler to write bitmap to the graphics device (which is the printer, based on the way the event is generated). 100, 100 gives ~an inch from top-left.
2. **Update()** call forces the File menu to close up before **ScreenCapture()** is called to capture the screen. When it wasn't there, the output included the menu. This was not detected in the sample code, which invoked printing from a button, not from a menu.
3. Assigns the **PrintDocument** object to the **PrintDialog** object, allowing it to assign the document to a printer, etc.
4. If the user cancels we return.
5. **Print()** method called. This will lead to unhandled **BeginPrint** event, then a **PagePrint** event which is handled by (1), assuming we set it as the handler.

Using this code, you can reliably produce an image of the current display, whatever it is.

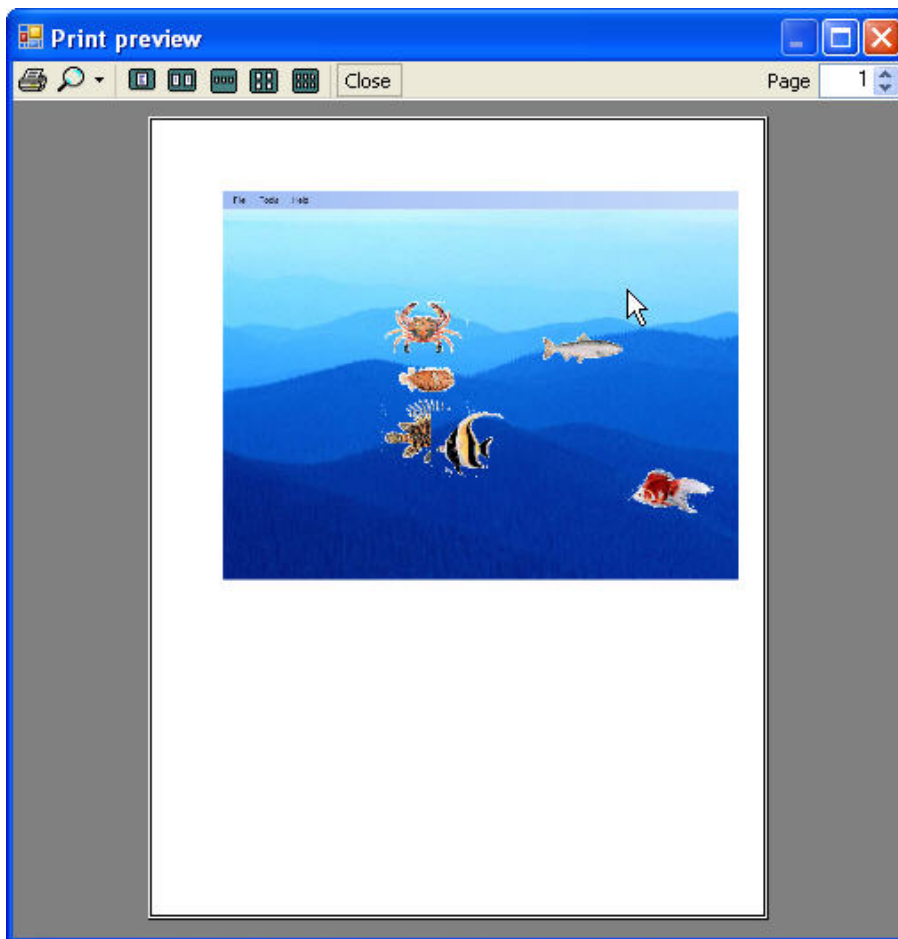
## Print Preview

Print preview is one of those nice things that you get--almost for free--once you've implemented printing. The code to implement the *File / Print Preview* menu item is as follows:

```
private void printPreviewToolStripMenuItem_Click(object sender, EventArgs e)
{
    Update();
    printPreviewDialog1.Document = printDocument1;
    CaptureScreen();
    printPreviewDialog1.ShowDialog();
}
```

The code is very simple because print preview works just like printing, except it sends a different graphics device (one attached to the print preview screen) to the **PrintPage** method. Also, **PrintDocument.Print()** is not required, since that is handled within the print preview dialog.

The resulting display, from the Aquarium project, appears as follows:



# Database Programming (Module 3)

## Learning Objectives

Upon completing this reading, you should be able to:

- Identify the key classes involved in connecting an application to a database
- Incorporate a database into your project
- Explain the purpose of the **DataSet**
- Acquire table from a database for use in a **DataSet**
- Update a database with data contained in a **DataSet**

## Overview

Connecting to a database in a VCEE .NET project is a relatively trivial matter. After identifying the database as a project data source, nearly all the necessary classes are established for you. If you initiate the process from a **DataViewGrid** control, the process goes even further--making the connection to the user interface.

Once the connection infrastructure has been established, using the data effectively in your program becomes somewhat more challenging. For example:

- Decisions need to be made about when updates back to the database need to be made, and the code to implement these updates needs to be written.
- A working knowledge of SQL is beneficial if advanced accesses--such as queries to specific data in the database--are to be made.
- Choices regarding what should be done locally (for later update) versus directly to the database need to be continually made.

Of course, these challenges can be accompanied by substantial benefits. The control structure of C# makes it much easier to implement complex database logic than trying to code all that logic in SQL. Moreover, in a common business scenario, you can implement your program as a middle tier in a multi-level architecture, instead of as a standalone application. (In this course, however, we'll limit ourselves to the standalone case.)

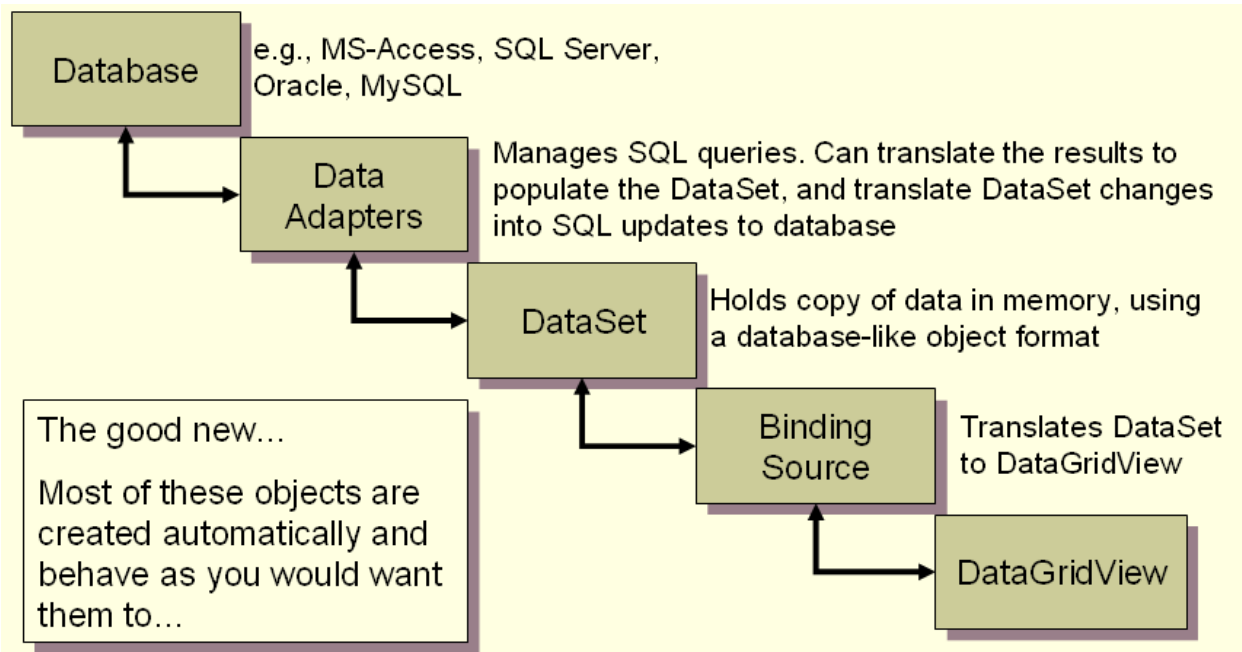
This reading covers the following topics:

- The nature of the database connection in C# and the objects involved
- The process of establishing the connection to the database
- The nature of the **DataSet** object, which holds data locally, inside the application
- Moving data and updates back and forth across the connection

These only scratch the surface of using databases. But they are enough to get started.

## The .NET Database Architecture

The basic architecture used to move data back and forth in .NET is shown below, which shows the components chain from database to output form:



Why does it need to be so complicated? Because .NET tries to support as many data access scenarios as possible. Specifically, not all these objects are required in a program, depending upon what the user needs to do. For example:

- If the program simply makes updates to a database and doesn't need an internal copy of the data, one could probably make due with *data adapters* that connect to the database and give us a way to issue SQL queries.
- If we want to manipulate data in memory using a database format, but don't care about a user interface, we only need a database, data adapters and a **DataSet** object.
- If the program creates its own local, in-memory database from external sources (e.g., text files) and displays it in a **DataGridView** object, we would just need the **DataSet**, binding source objects and **DataGridView** controls.
- If we don't use a database format at all--such as when we connect a **DataGridView** to a collection, as we saw in the **Transcript** class and both the **Aquarium** and **Bingo** projects--we just need binding sources and the **DataGridView** object.

The different layers allow us to pick and choose how we implement our program. Furthermore, to reduce the drudgery of programming the connection, the VCEE tool provides wizards that create many of these objects without programming. For example, when you create a new data source and attach it to a **DataGridView** (all done in one wizard accessed from the

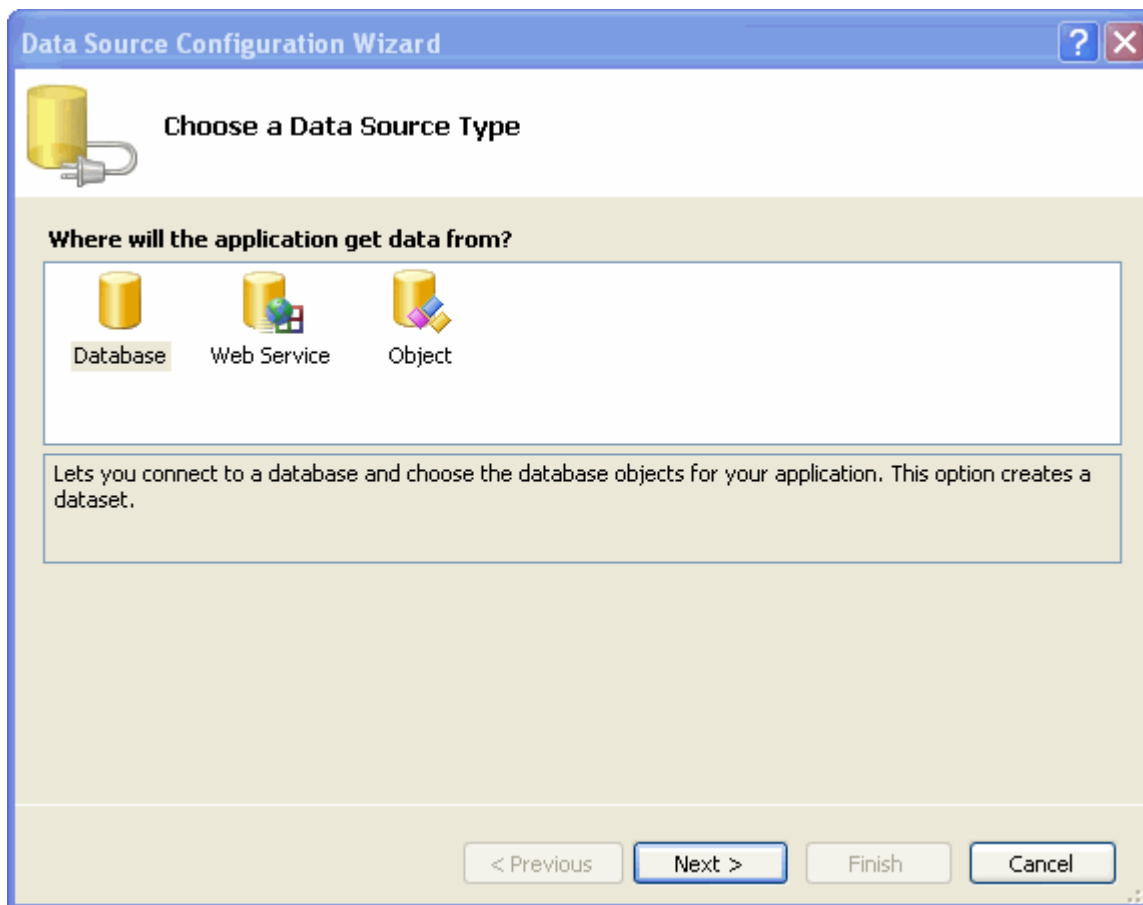
**DataGridView** common task shortcut), the data adapters, **DataSet** and binding sources are all created automatically and configured to move data from the database all the way to the control.

## Connecting to a Database

To illustrate the connection process, we use the Bingo project as an example. Here, we need to build the connection from a local MS-Access database all the way to a database control. From a programming perspective, this is one of the easiest connections to make, since the wizard does pretty much everything for you--and prompts you for what it doesn't know (e.g., database names, table to be connected, variable names).

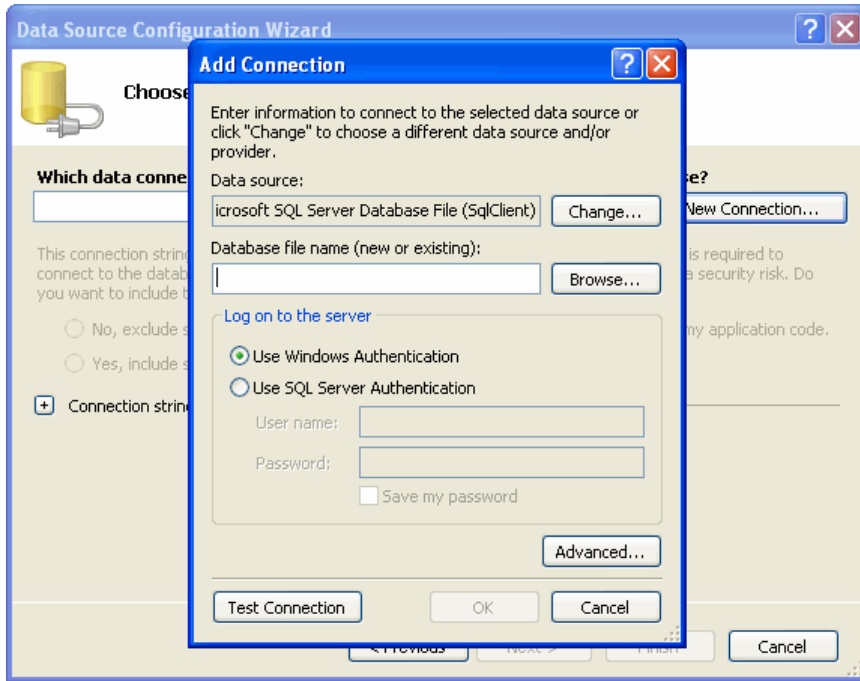
## Identify the Database

First, we need to identify an MS-Access database that we've already created, named *Caller.mdb*, as our data source. This can be done by choosing *Data / Add new data source...* from the main menu, or as part of the common tasks if you are doing it from a **DataViewGrid**. You only need to add a data source to the project once--it will be shared between grids. The wizard appears as follows:

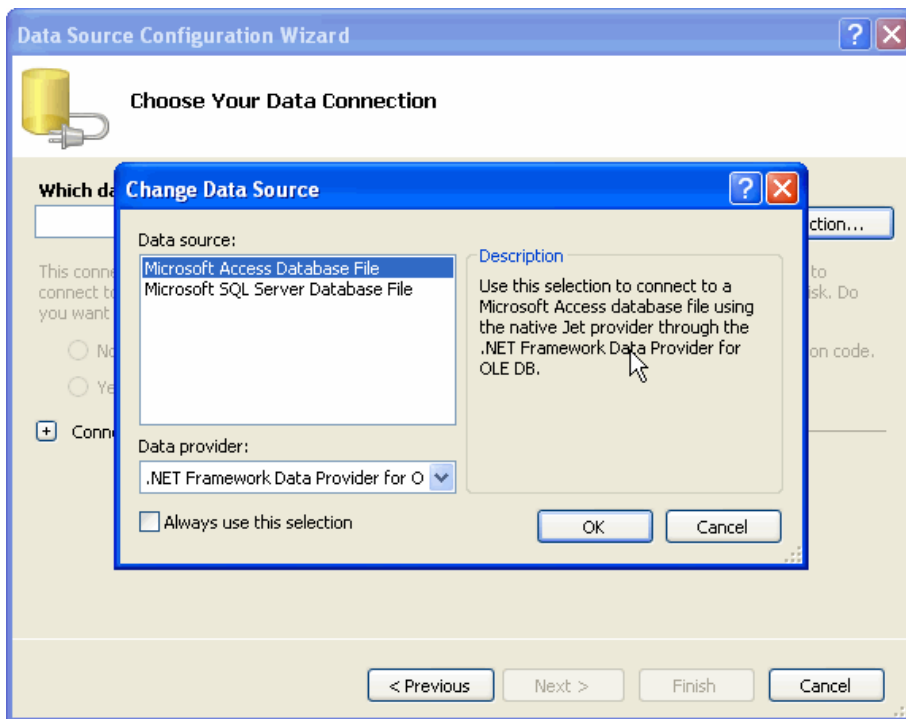


Choose database, then click next. On the next screen, click the "New Connection" button. In the resulting dialog (below), first click the "Change" button. The default is SQL server, but we want

an MS-Access database, since explaining how to set up a SQL server database is beyond the scope of this course.



This will open a dialog that allows you to select MS-Access as your data source, as shown below:



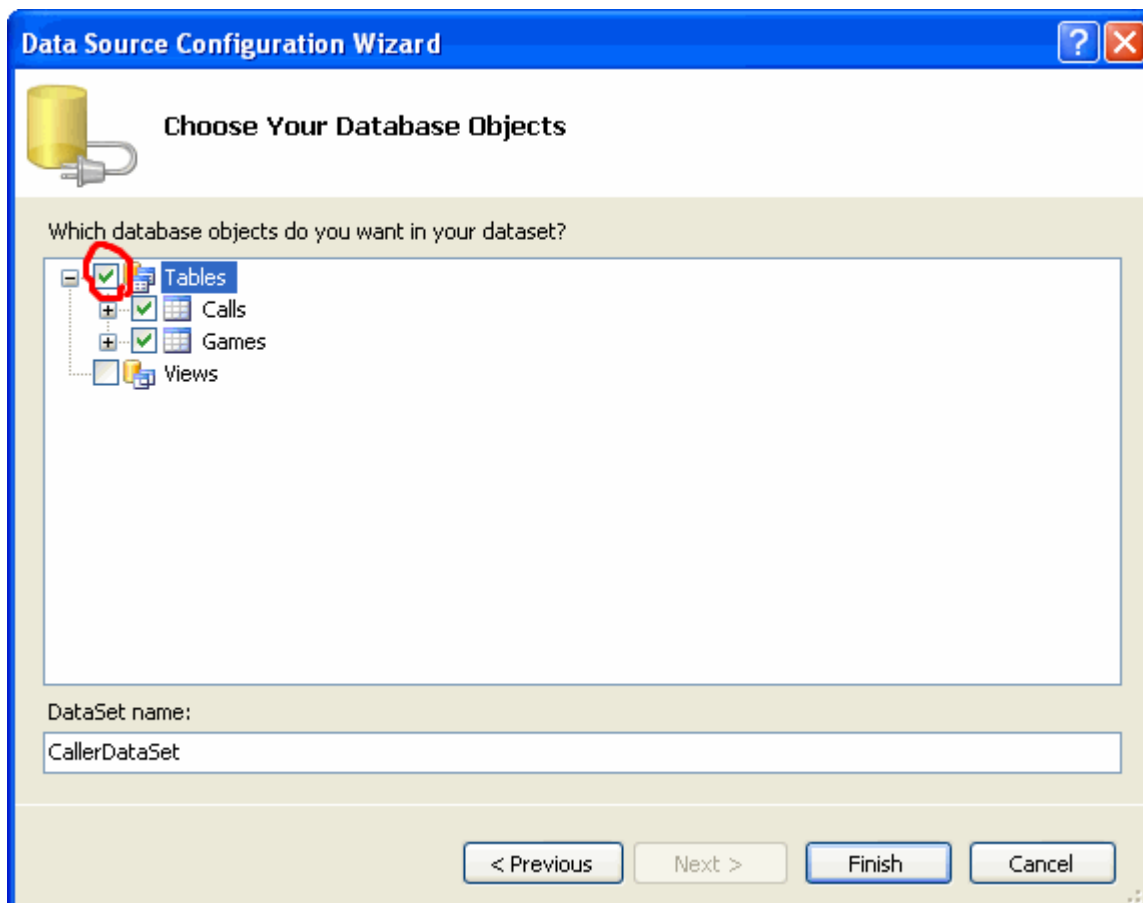
You can then OK the dialog. Upon returning to the "New Connection" dialog, you can browse for the *Caller.mdb* database file by clicking the "Browse" button next to the *Database file name* (shown previously). You can just ignore the User Name and Password options and click OK. Then click the Next button. It will pop up a message about copying the file into your project.

In answering the file copying question, say yes. This creates a local copy, held in your project area, which holds any initial data you've put in it. *Be aware of the following:*

Every time you compile your project, that local file is then copied into the program directory (...*project-folder\bin\Debug* below your project folder). This, of course, erases any previous changes you might have made to the database the last time you ran your project. It also means that if you want to see if your program is actually updating the database, you need to look at the ...*project-folder\bin\Debug* before you compile again to see these changes.

## Identify the Tables to be Used

Once you've identified your database, you can then click next on the screen asking you about saving the connection string. Now you are on the final screen. Check the Tables box, circled below, to indicate you want to be able to access the *Games* and *Calls* tables in the application. You can modify this later, should you need to, to add additional tables from a larger database.



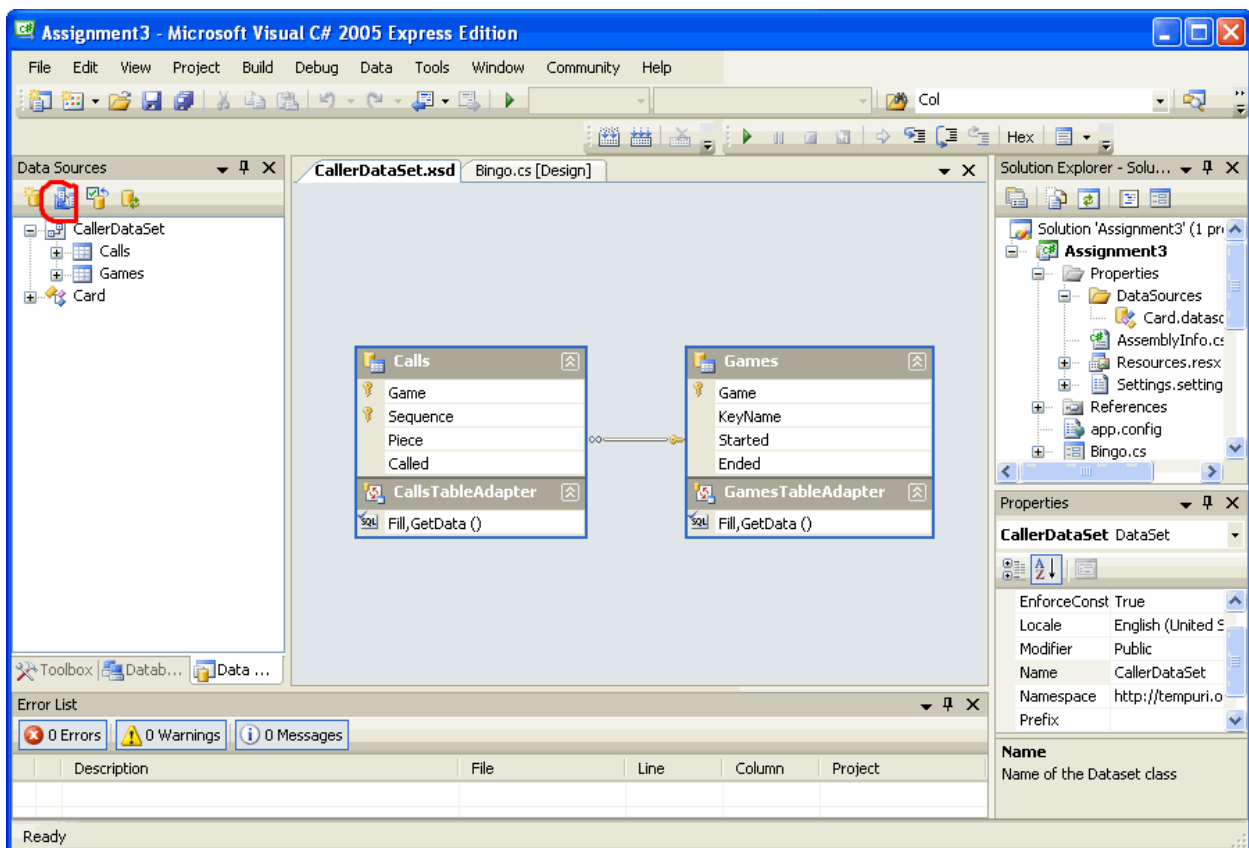
This particular step is the key to creating the classes we need to implement our connection. Specifically, when you identify tables, you are telling the wizard to create a **TableAdapter** object for each table. In this case, it calls them:

- **gamesTableAdapter**
- **callsTableAdapter**

In addition, the name you place in the bottom textbox (the default is *your-data-baseDataSet*) serves to create the **DataSet** object that your application moves data in and out of. In addition, if you are creating this within a **DataGridView** control, the wizard will do two additional things:

- Add a **Load** method handler to the form that automatically populates the **DataSet** object from the table adapters.
- Creates the binding source object to attach the **DataSet** to the **DataGridView** control.

Once you've completed entering the information in this form, you can then click "Finish". If you then choose *Data / Show Data Sources* from the VCEE menu, then Edit DataSet with designer (button circled below), you can see the table structure of the database.



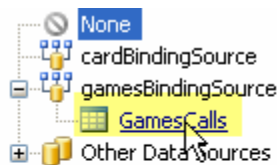
In this case, we see two tables:

- Games, containing the Game name, a KeyName and Started and Ended dates.

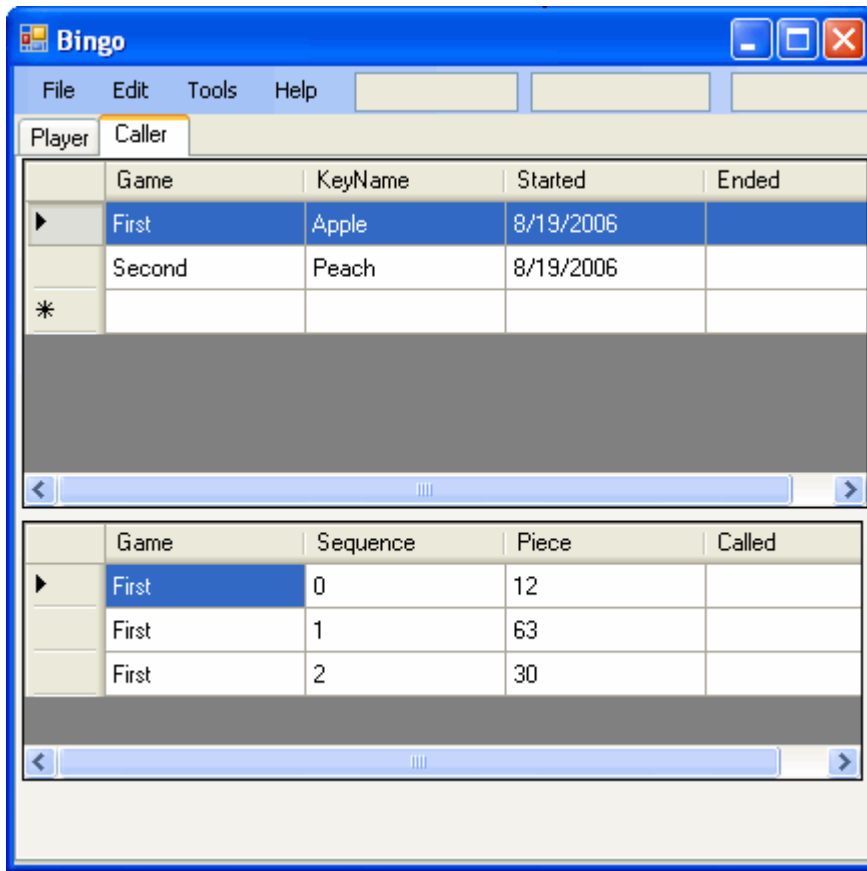
- Calls, a related table that shows the Sequence (a counter), Piece and Called time for the particular game.

The bottom portion of each table display identifies members of the table adapter classes created for your database. The member **Fill()** is used to populate a particular **DataSet** table with records from the database. It is possible to add members to the table adapter that are based on SQL queries. These methods can either return data (SQL **Select** statements) or modify database data (e.g., delete, update, insert). Custom data adapters (e.g., supporting multi-table queries) can also be added, allowing virtually any database action through function calls within your program. Unless you are reasonably conversant with SQL, however, you will probably not be able to make full use of these advanced features.

In our particular example, there is a many-to-one relationship between calls and games--a given game can have many calls (up to 75, in fact). This relationship in the database is also reflected in the newly created **DataSet** object and proves to be important because it impacts the binding object created when you add the parent side (the "one" side in a many-to-one relationship-- Games in this example) to a **DataGridView**. Specifically, the binding object for the parent table includes a **GamesCalls** sub-binding object, as shown below:

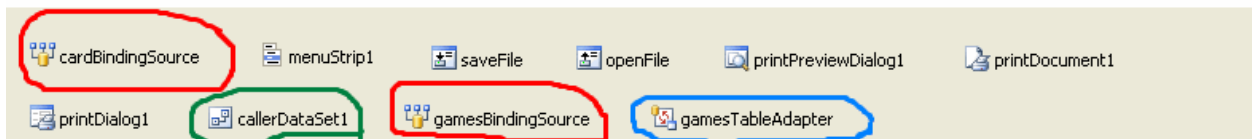
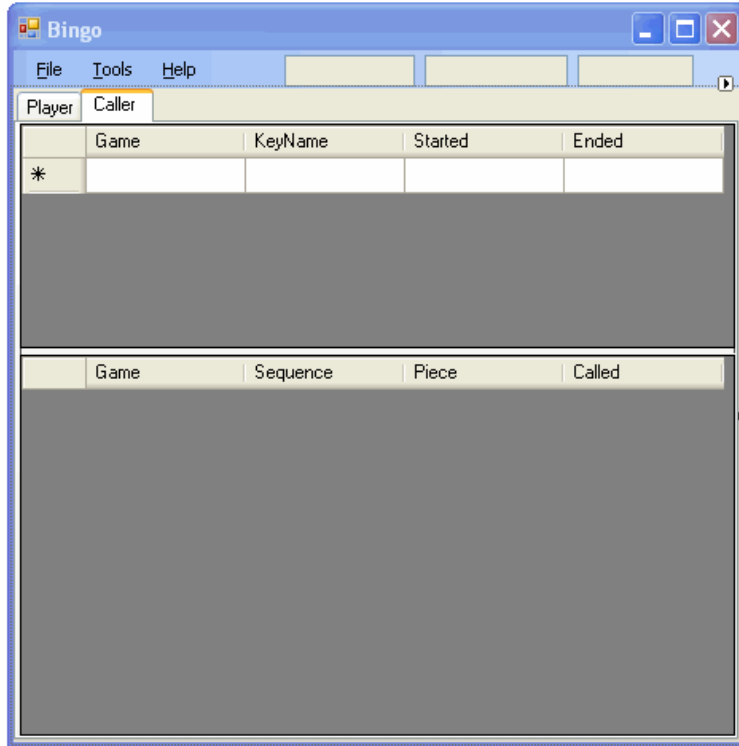


A **DataGridView** can also attach to this sub-binding object. When this is done, the row selected in the parent table grid (e.g., Games) controls what displays in the second grid (a.k.a., the child grid, in this case Calls). This means that only rows (e.g., Calls) related to the selected parent row (e.g., a particular Game) are displayed in the child table. This is referred to as a parent-child relationship. Implemented with a splitter container, it might appear as follows:



## Form Result

After going through the procedure described, many objects will be added to your form (assuming this was done starting from a **DataGridView** control). These appear as follows, circled according to their type:



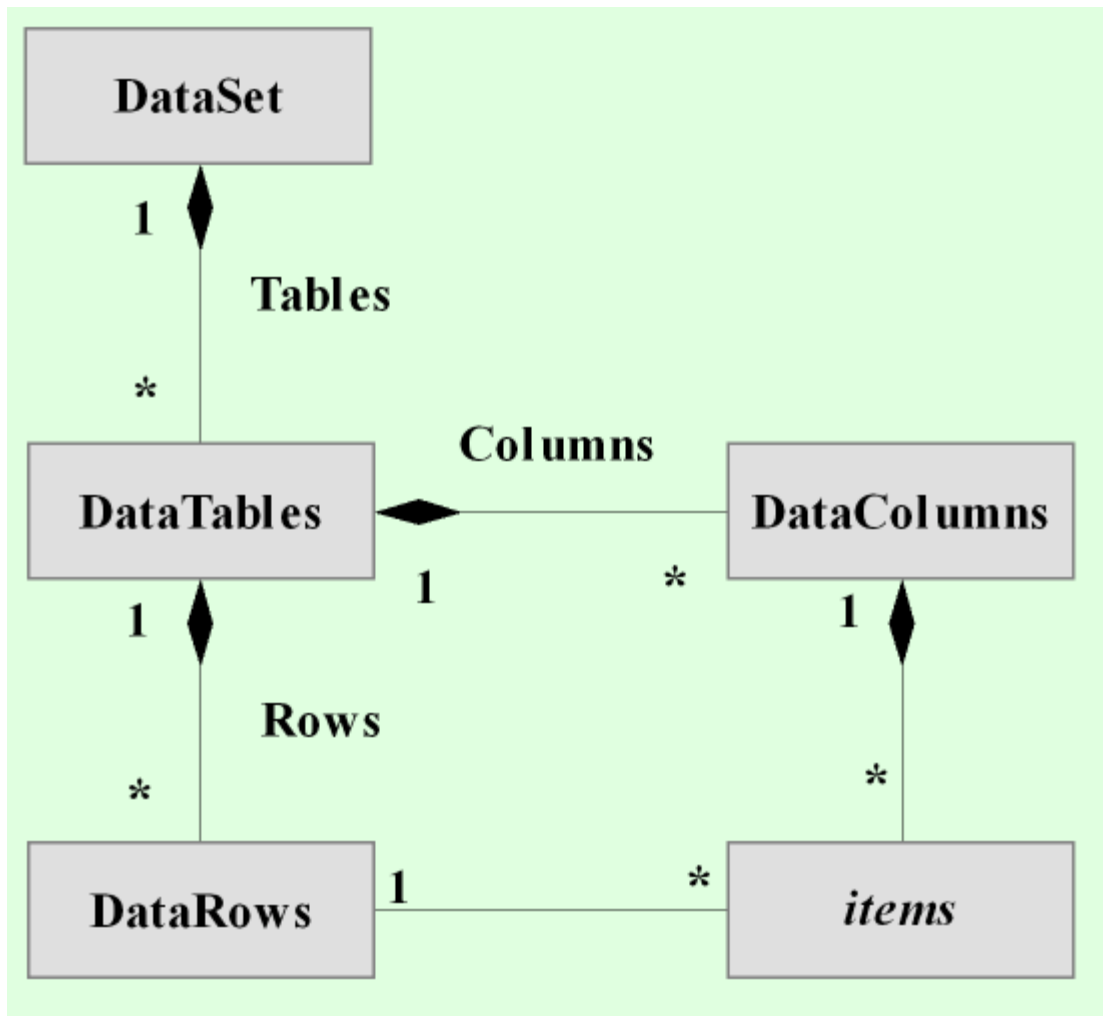
- *Red:* Binding source
- *Green:* DataSet
- *Blue:* Table adapter

## DataSet objects

The DataSet is a very nice .NET innovation that allows you to create an object that functions like an in-memory database. Specifically, a DataSet is a complex object composed of many other objects, including:

- **Tables, Relations** to establish the table structure and relationships. **Tables** contains a collection of **DataTable** objects, **Relations** contains a collection of **DataRelation** objects, defining individual relationships between tables (which can be used, for example, to implement parent-child forms).
- **DataTable** objects contain **Rows, Columns**. **Columns** is a collection of **DataColumn** objects that allow the table structure to be established. **Rows** is a collection of **DataRow** objects that allow data to be accessed, added and edited.

Conceptually, the overall structure of a DataSet object (excluding relationships) is shown below:



The actual data can be viewed as the intersection of a given **DataColumn** and **DataRow** object. Indexers are provided within the **DataSet** to allow tables and columns to be accessed by name as well as by position (index), e.g.,

```
DataTable myTable=callerDataSet.Tables["Games"];
```

When **DataSet** objects are created using the wizard, this style of access is not normally required, however, since additional members, named after the actual tables, are also created automatically, e.g.,

```
DataTable myTable=callerDataSet.Games;
```

**DataSet** objects can also be serialized directly and can easily be exported to XML--which can then be imported by databases such as MS-Access. For this reason, you may decide to use a **DataSet** to store application data even if no databases are involved.

## Bringing Database Data into a DataSet

Despite their versatility, our primary purpose in using **DataSet** objects in this course is as a way station for data between a database and a **DataViewGrid** object. The code to populate the **DataSet** is accomplished through table adapter **Fill()** statements. These are automatically generated by VCEE when a database is attached to a **DataGridView** using the wizard and appear as follows:

```
private void Bingo_Load(object sender, EventArgs e)
{
    // TODO: This line of code loads data into the 'callerDataSet1.Calls' table.
    //You can move, or remove it, as needed.
    this.callsTableAdapter.Fill(this.callerDataSet1.Calls);
    // TODO: This line of code loads data into the 'callerDataSet1.Games' table.
    //You can move, or remove it, as needed.
    this.gamesTableAdapter.Fill(this.callerDataSet1.Games);
}
```

This code supplies the **DataSet** table (using the automatically generated name property for the table) as an argument to the table adapter's **Fill()** member. Naturally, the **Fill()** member can also be used to populate other appropriately structured data set objects. Specialized **Fill()** functions can also be generated based on SQL queries.

## Updating Database Data from a DataSet

To move data from the **DataSet** back to the database, we use the table adapter's **Update()** method. This works in the opposite direction from the **Fill()** member, as shown below:

```
private void Bingo_FormClosing(object sender, FormClosingEventArgs e)
{
    gamesTableAdapter.Update((CallerDataSet.GamesDataTable) callerDataSet1.Games);
    callsTableAdapter.Update((CallerDataSet.CallsDataTable) callerDataSet1.Calls);
}
```

The **Update()** member usually requires a typecast, as shown, since it is very specifically tied to a given table.

**Update()** works based on the way a **DataSet** handles editing. Specifically, the **DataSet** records each change made by the user since the most recent **Fill()**, then issues a stream of SQL commands reflecting these changes within the **Update()** logic. There is a temptation after editing your data set to call the **AcceptChanges()** member, just because it sounds reassuring. This turns out to be a very bad idea, since it changes the data in the data set and then erases the entire record of changes. With this record erased, the **Update()** member will do nothing--since it doesn't see any changes. Why I felt this was worth mentioning is left to the reader's imagination.