

Coalesce

Chapter 1

Variables

Overview

Definition

One of the jobs of a program is to request information from the user who types it using the keyboard. In some cases, the user provides information using the mouse. In the computer world, a piece of information that is part of a program is called datum and the plural is data. Sometimes the word data is used both for singular and plural items.

Imagine you write a program as an employment application. A person would use it to enter different names of various job applicants as they apply for the same job. Because the names change from one job applicant to another, it (the job applicant's name) is called a variable.

When the user enters data in a program, the computer receives it and must store it somewhere to eventually make it available to the program as needed. Referring to the employment application analogy, the types of information a user would enter into the program are the name, the residence, the desired salary, years of experience, education level, etc. Because there can be so much information for the same program, you must specify to the computer what information you are referring to and when. To do this, each category of piece of information must have a name.

Names in C#

To name the variables of your program, you must follow strict rules. In fact, everything else in your program must have a name. C# uses a series of words, called keywords, for its internal use. This means that you must avoid naming your objects using one of these keywords. They are:

abstract	const	extern	int	out	short	typeof
as	continue	false	interface	override	sizeof	uint
base	decimal	finally	internal	params	stackalloc	ulong
bool	default	fixed	is	private	static	unchecked
break	delegate	float	lock	protected	string	unsafe
byte	Do	for	long	public	struct	ushort
case	double	foreach	namespace	readonly	switch	using
catch	Else	goto	new	ref	this	virtual
char	enum	if	null	return	throw	void
checked	event	implicit	object	sbyte	true	volatile
class	explicit	in	operator	sealed	try	while

Coalesce

Once you avoid these words, there are rules you must follow when naming your objects. On this site, here are the rules we will follow:

- The name must start with a letter or an underscore
- After the first letter or underscore, the name can have letters, digits, and/or underscores
- The name must not have any special characters other than the underscore
- The name cannot have a space

Besides these rules, you can also create your own but that abide by the above.

C# is case-sensitive. This means that the names Case, case, and CASE are completely different. For example, the main function is always written Main.

Data Types

Before using a variable in your program, you must let the computer (in reality the compiler) know about it. Once the compiler knows about a variable, it would reserve an amount of memory space for that variable. Using its name, you can refer to a particular variable when necessary. Because there are various types of variables a program can use, such as the employee's name, the residence, the desired salary, years of experience, education level, etc for our employment application analogy, the compiler needs a second piece of information for each variable you intend to use. This piece of information specifies the amount of space that a variable needs. You can see that, to store a character, such as an employee's gender (M or F) or an answer as Y or N to a question, the compiler would certainly not need the same amount of space to store the name of the last school attended by an employee.

A data type is an amount of space needed to store the information related to a particular variable.

The name of a variable allows you and the compiler to refer to a particular category of information in your program. The data type allows the compiler to reserve an adequate amount of memory space for a variable. Because you are the one who writes a program, you also tell the computer the amount of memory space each particular variable will need. Based on this, the C# language provides categories of data types used to specify this amount of space needed for a variable.

As stated already, in order to use a variable, the compiler must be aware of it. Making the compiler aware is referred to as declaring the variable. To declare a variable, provide the data type followed by the name of the variable. Therefore, the syntax used to declare a variable is:

DataType VariableName;

After declaring a variable, you can ask the compiler to store some value into its reserved memory space. This is done using the assignment operator represented with =. Of course, throughout your program, you can assign another value to the variable as you judge necessary. Giving its first value to a variable is referred to as initializing it.

If you plan to use various variables of the same type, you can declare them and specify their names on the same line, sharing the common data type. The syntax used would be:

DataType Variable1, Variable2, Variable_n;

The variables declared like this must be of the same type.

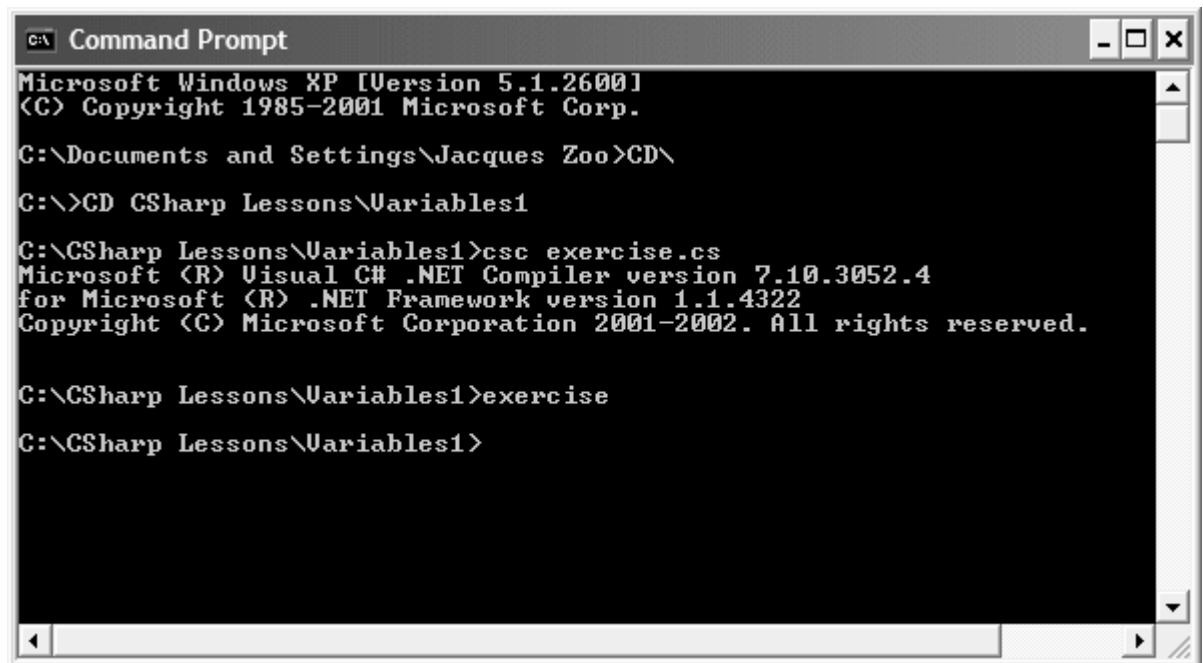
Coalesce

Practical Learning: Using Variables

1. Start Notepad and, in the empty file, type the following:

```
using System;  
  
// A Car class  
class Car  
{  
    static void Main()  
    {  
    }  
}
```

2. Save it in a new folder called **Variables1**
3. Save the file as **exercise.cs**
4. Open the Command Prompt and change the directory to the folder that contains the current exercise
5. Compile it by typing **csc exercise.cs**
6. Execute it by typing **exercise**



```
C:\> Command Prompt  
Microsoft Windows XP [Version 5.1.2600]  
<C> Copyright 1985-2001 Microsoft Corp.  
  
C:\Documents and Settings\Jacques Zoo>CD\  
  
C:\>CD CSharp Lessons\Variables1  
  
C:\CSharp Lessons\Variables1>csc exercise.cs  
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4  
for Microsoft (R) .NET Framework version 1.1.4322  
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.  
  
C:\CSharp Lessons\Variables1>exercise  
  
C:\CSharp Lessons\Variables1>
```

7. Return to Notepad

Coalesce

Primary Types

Strings

A string is an empty space, a character, a word, or a group of words that you want the compiler to consider "as is", that is, not to pay too much attention to what the string is made of, unless you explicitly ask it to. This means that, in the strict sense, you can put in a string anything you want.

Primarily, the value of a string starts with a double quote and ends with a double-quote. An example of a string is "Welcome to the World of C# Programming!". You can include such a string in the **Console.Write()** method to display it on the console. Here is an example:

```
using System;

class BookClub
{
    static void Main()
    {
        Console.WriteLine("Welcome to the World of C# Programming!");
    }
}
```

This would produce:

Welcome to the World of C# Programming!

Sometimes, you will need to use a string whose value is not known in advance. Therefore, you can first declare a string variable. To do this, use the string keyword (in fact, it is a class) followed by a name for the variable. The name will follow the rules we defined above. An example of a string declaration is:

```
string Msg;
```

After declaring a string, you can give it a primary value by assigning it an empty space, a character, a symbol, a word, or a group of words. The value given to a string must be included in double-quotes. Here are examples of string variables declared and initialized:

```
string Empty = "";
string Gender = "F";
string FName = "Nelson Mandela";
string Msg = "Welcome to the World of C# Programming!";
```

After initializing a string variable, you can use its name in any valid operation or expression. For example, you can display its value on the console using the **Console.Write()** or the **Console.WriteLine()** methods. Here is an example:

```
using System;

class BookClub
{
    static void Main()
```

Coalesce

```
{
    string Msg = "Welcome to the World of C# Programming! ";
    Console.WriteLine(Msg);
}
```

Practical Learning: Using Strings

1. To declare a string, in Notepad, change the file as follows:

```
using System;

// A Car class
class Car
{
    static void Main()
    {
        // Specify the name of car
        string Make, Model;
        // Specify the name of car
        Make = "Ford";
        Model = "Taurus";

        // Display information about this car
        Console.Write("Car Name: ");
        Console.Write(Make);
        Console.Write(" ");
        Console.WriteLine(Model);
        Console.WriteLine();
    }
}
```

2. Save the file
3. In the Command Prompt, compile it with **csc exercise.cs**
4. Execute it with **exercise**. This would produce:

Car Name: Ford Taurus

5. Return to Notepad

Enumerators

An enumerator is a list of natural numeric values with each value represented by a name. To create an enumerator, use the **enum** keyword. The values, called members of the enumerator, are included between an opening curly bracket and the list ends with a closing curly bracket.

Coalesce

Because the enumerator can be used as a data type, it must have a name. Here is an example:

```
enum MemberCategories { mcTeen, mcAdult, mcSenior };
```

In C#, an enumerator must be created outside of a function, that is, in a class.

As stated already, each member of the enumerator holds a value of a natural number, such as 0, 4, 12, 25, etc. In C#, an enumerator cannot hold character values (of type **char**).

After creating an enumerator, you can declare a variable from it. For example, you can declare a variable of a MemberCategories type as follows:

```
using System;

class BookClub
{
    enum MemberCategories { mcTeen, mcAdult, mcSenior };

    static void Main()
    {
        MemberCategories MbrCat;
    }
}
```

After declaring such a variable, to initialize it, specify which member of the enumerator would be given to it. You should only assign a known member of the enumerator. To do this, on the right side of the assignment operator, type the name of the enumerator, followed by the period operator, and followed by the member whose value you want to assign. Here is an example:

```
using System;

class BookClub
{
    enum MemberCategories { mcTeen, mcAdult, mcSenior };

    static void Main()
    {
        MemberCategories MbrCat = MemberCategories.mcAdult;
    }
}
```

You can also find out what value the declared variable is currently holding. For example, you can display it on the console using the Write() method. Here is an example:

```
using System;
```

Coalesce

```
class BookClub
{
    enum MemberCategories { mcTeen, mcAdult, mcSenior };

    static void Main()
    {
        MemberCategories MbrCat = MemberCategories.mcAdult;
        Console.WriteLine(MbrCat);
    }
}
```

This would produce:

mcAdult

As stated already, an enumerator is in fact a list of numbers where each member of the list is identified with a name. The first item of the list has a value of 0, the second has a value of 1, etc. For example, on the MemberCategories enumerator, mcTeen has a value of 0 while mcSenior has a value of 2. These are the default values. If you don't want these values, you can explicitly define the value of one or each member of the list. Suppose you want the mcTeen in the above enumerator to have a value of 5. To do this, use the assignment operator, =, to give the desired value. The enumerator would be:

```
enum MemberCategories { mcTeen=5, mcAdult, mcSenior };
```

In this case, mcTeen now would have a value of 5, mcAdult would have a value of 6, and mcSenior would have a value of 7. You can also assign a value to more than one member of an enumerator. Here is an example:

```
enum MemberCategories { mcTeen=3, mcAdult=8, mcSenior };
```

In this case, mcSenior would have a value of 9.

Practical Learning: Using an Enumerator

1. To use an enumerator, in Notepad, change the contents of the file as follows:

```
using System;

// A Car class
class Car
{
    enum TransmissionType { Manual, Automatic, Unspecified };

    static void Main()
    {
        // Specify the name of car
        string Make="Ford", Model="Taurus";
        TransmissionType TransType = TransmissionType.Manual;

        // Display information about this car
        Console.Write("Car Name:  ");
        Console.Write(Make);
    }
}
```

Coalesce

```
        Console.Write(" ");  
        Console.WriteLine(Model);  
        Console.Write("Transmission: ");  
        Console.WriteLine(TransType);  
        Console.WriteLine();  
    }  
}
```

2. Save the file
3. Compile and execute it. This would produce:

```
Car Name:   Ford Taurus  
Transmission: Manual
```

4. Return to Notepad

Boolean Variables

A variable is referred to as Boolean if it is meant to carry only one of two values stated as true or false. To declare a Boolean variable, use the bool keyword.

Here is an example of declaring a Boolean variable:

```
bool TheStudentIsHungry;
```

After declaring a Boolean variable, you can give it a value by assigning it a true or false value. Here is an example:

```
using System;  
  
class ObjectName  
{  
    static void Main()  
    {  
        bool TheStudentIsHungry;  
  
        TheStudentIsHungry = true;  
        Console.Write("The Student Is Hungry expression is: ");  
        Console.WriteLine(TheStudentIsHungry);  
        TheStudentIsHungry = false;  
        Console.Write("The Student Is Hungry expression is: ");  
        Console.WriteLine(TheStudentIsHungry);  
    }  
}
```

This would produce:

```
The Student Is Hungry expression is: True  
The Student Is Hungry expression is: False
```


Coalesce

You can also give its first value to a variable when declaring it. In this case, the above variable can be declared and initialized as follows:

```
bool TheStudentIsHungry = true;
```

Practical Learning: Using a Boolean Variable

1. To use an example of a Boolean variable, in the Notepad file, declare the following:

```
using System;

// A Car class
class Car
{
    enum TransmissionType { Manual, Automatic, Unspecified };

    static void Main()
    {
        // Specify the name of car
        string Make = "Ford", Model = "Taurus";
        TransmissionType TransType = TransmissionType.Manual;
        bool HasAirCondition = true;

        // Display information about this car
        Console.Write("Car Name:  ");
        Console.Write(Make);
        Console.Write(" ");
        Console.WriteLine(Model);
        Console.WriteLine("Transmission: ");
        Console.WriteLine(TransType);
        Console.WriteLine("Air Condition? ");
        Console.WriteLine(HasAirCondition);

        Console.WriteLine();
    }
}
```

2. Save the file
3. Compiler and execute it. This would produce:

```
Car Name:  Ford Taurus
Transmission: Manual
Air Condition? True
```

4. Return to Notepad

Characters

Coalesce

In the English alphabet, a character is one of the following symbols: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, and Z. Besides these readable characters, the following symbols are called digits and they are used to represent numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. In addition, some symbols on the (US QWERTY) keyboard are also called characters or symbols. They are ` ~ ! @ # \$ % ^ & * () - _ = + [{] } \ | ; : ' < ? . / , > "

Besides the English language, other languages use others or additional characters that represent verbal or written expressions.

C# recognizes that everything that can be displayed as a symbol is called a character. To declare a variable whose value would be a character, use the **char** keyword. Here is an example:

```
char Gender;
```

To initialize a character variable, include its value in single quotes. Here is an example:

```
using System;

class ObjectName
{
    static void Main()
    {
        char Gender = 'M';

        Console.Write("Student Gender: ");
        Console.WriteLine(Gender);
    }
}
```

This would produce:

Student Gender: M

Natural Numbers

Byte

A byte is a small positive number whose value can range from 0 to 255. You can use it when you know a variable would hold a relatively small value such as people's age, the number of children of one mother, etc.

To declare a variable that would hold a small natural number, use the Byte keyword. Here is an example:

```
Byte Age;
```

You can initialize a Byte variable when declaring it or afterwards. Here is an example that uses the Byte data type:

Coalesce

```
using System;

class ObjectName
{
    static void Main()
    {
        Byte Age = 14;

        Console.Write("Student Age: ");
        Console.WriteLine(Age);
        Age = 12;
        Console.Write("Student Age: ");
        Console.WriteLine(Age);
    }
}
```

Make sure you do not use a value that is higher than 255 for a Byte variable, you would receive an error. When in doubt, or when you think that there is a possibility a variable would hold a bigger value, don't use the Byte data type as it doesn't like exceeding the 255 value limit.

Signed Byte

A byte number is referred to as signed if it can hold a value that ranges from -128 to 127. To declare a variable for that kind of value, use the **sbyte** keyword.

Short Numbers

A natural number is short if its value is between -32768 and 32767. A variable for such a number is declared using the **short** data type.

Positive Short Numbers

If a short number must always be positive, it is called unsigned. A variable for this type is declared using the **ushort** keyword.

Integers

An integer is a natural number, that is, a number that doesn't hold a decimal value or that is not a fraction. Its value can range from -2,147,483,648 to 2,147,483,647.

To declare a variable for an integer type, use the **int** keyword.

Unsigned Integers

When an integer must always have a positive value, it is referred to as unsigned. To declare a variable for an unsigned integer, use the **uint** data type.

Long Integers

Coalesce

A natural number that can hold a very large value qualifies as a long integer. The value of a long integer can range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. A variable for such a value can be declared using the **long** keyword.

Unsigned Long Integers

If a long integer must always assume a positive value, such as ranging from 0 to 18,446,744,073,709,551,615, it is called long integer. Such a variable can be declared with the **ulong** keyword.

Practical Learning: Using an Natural Number

1. To use a natural number, declare, initialize, and display the following unsigned long integer:

```
using System;

// A Car class
class Car
{
    enum TransmissionType { Manual, Automatic, Unspecified };

    static void Main()
    {
        // Specify the name of car
        string Make = "Ford", Model = "Taurus";
        TransmissionType TransType = TransmissionType.Manual;
        bool HasAirCondition = true;
        ulong Mileage = 72866;

        // Display information about this car
        Console.WriteLine("\n=====");
        Console.Write("Car Name:   ");
        Console.Write(Make);
        Console.Write(" ");
        Console.WriteLine(Model);
        Console.Write("Transmission: ");
        Console.WriteLine(TransType);
        Console.Write("Air Condition? ");
        Console.WriteLine(HasAirCondition);
        Console.Write("Mileage:   ");
        Console.Write(Mileage);
        Console.WriteLine(" miles");
        Console.WriteLine("=====");

        Console.WriteLine();
    }
}
```

2. Save the file

Coalesce

3. Compile and execute it. This would produce:

```
Car Name:   Ford Taurus
Transmission: Manual
Air Condition? True
Mileage:    72866 miles
```

4. To finish, type Exit and press Enter

Floating-Point Values

Float Values

A number is referred to as floating it can assume a decimal fraction, that is, a number that includes a period. To declare a variable that can hold such a value, use the `float` keyword. Here is an example:

```
float ShoeSize;
```

Double-Precision

A number is referred to as double-precision when its value requires significantly more precision than a regular decimal value. To declare a double-precision value in C#, use the **`double`** keyword. Here is an example:

```
double Distance;
```

Use a double data type not only if the variable would need precision but also if its value would range from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$ with up to 16 digits precision.

Here is an example:

```
using System;

class ObjectName
{
    static void Main()
    {
        double Distance = 224.62;

        Console.Write("Distance = ");
        Console.WriteLine(Distance);
    }
}
```

This would produce:

```
Distance = 224.62
```

Decimal

Coalesce

The Decimal keyword is used for a variable that would hold significant large values that can range from $\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$.

Details on Using Variables

Value Casting

Value casting consists of converting a value of one type into a value of another type. For example, if you want a decimal value and you may want that value in an expression that expects an integer.

There are various ways to cast a value from one type to another but C# is particularly flexible on this issue. Here is an example:

```
using System;

class Exercise
{
    static void Main()
    {
        double number1 = 22.285;
        // Cast a decimal number into a natural number
        int number2 = (int)number1;
    }
}
```

Value Types Methods

In C# (unlike many other languages such as C/C++, Pascal, etc), all of the data types we have used so far are in fact complete classes. This means that they are equipped to handle their own assignments called functions as we will learn in subsequent lessons. When a data type (in this case a class) has the ability to perform assignments through its functions, it is able to better communicate with other variables, functions, or classes (the fact that the other languages, namely C/C++, Pascal, etc, use data types that are not considered classes doesn't mean they are less efficient or that they have an anomaly: it is simply the construct of the language!). These classes are defined in the **System** namespace. The classes of these data types are defined as:

Data Type	Class	Data Type	Class
bool	Boolean	char	Char
byte	Byte	sbyte	SByte
short	Int16	ushort	UInt16
int	Int32	uint	UInt32
long	Int64	ulong	UInt64
float	Single	double	Double
decimal	Decimal		

This means that, if you don't want to use the data types we have reviewed so far, you can use the class that is defined in the **System** namespace. To use one of these types, type the System

Coalesce

namespace followed by a period. Then type the equivalent class you want to use. Here is an example:

```
class Operations
{
    public double Addition()
    {
        System.Double a;
        System.Double b;
        System.Double c;

        a = 128.76;
        b = 5044.52;
        c = a + b;

        return c;
    }
}
```

Because the regular names of data types we introduced in the previous lesson are more known and familiar, we will mostly use them.

Because the data type are defined as classes, they are equipped with methods. One of the methods that each one them has is called **ToString**. As it s name implies, it is used to convert a value to a string.
Data Reading and Formatting

Coalesce

Chapter 2 Data Handling

Introduction

In the previous lesson, we saw that the `Console` class allows using the **Write()** and the **WriteLine()** functions to display things on the screen.

While the **Console.Write()** method is used to display something on the screen, the **Console** class provides the **Read()** method to get a value from the user. To use it, the name of a variable can be assigned to it. The syntax used is:

```
VariableName = Console.Read();
```

This simply means that, when the user types something and presses Enter, what the user had typed would be given (the word is *assigned*) to the variable specified on the left side of the assignment operator.

Read() doesn't always have to assign its value to a variable. For example, it can be used on its own line, which simply means that the user is expected to type something but the value typed by the user would not be used for any significant purpose. For example some versions of C# (even including Microsoft's C# and Borland C#Builder) would display the DOS window briefly and disappear. You can use the **Read()** function to wait for the user to press any key in order to close the DOS window.

Besides **Read()**, the **Console** class also provides the **ReadLine()** method. Like the **WriteLine()** member function, after performing its assignment, the **ReadLine()** method sends the caret to the next line. Otherwise, it plays the same role as the **Read()** function.

Practical Learning: Introducing Data Reading

1. Start Notepad to create a new file and type the following:

```
using System;

class Payroll
{
    static void Main()
    {
        string firstName;
        string lastName;
        int    dayHired;
        int    monthHired;
        int    yearHired;
        double hourlySalary;

        firstName = "Alexander";
        lastName  = "Baugh";
    }
}
```


Coalesce

```
        dayHired = 12;
        monthHired = 6;
        yearHired = 1998;
        hourlySalary = 14.58;

        Console.WriteLine("\nEmployee Payroll");
        Console.Write("Full Name: ");
        Console.Write(firstName);
        Console.Write(" ");
        Console.WriteLine(lastName);
        Console.Write("Date Hired: ");
        Console.Write(dayHired);
        Console.Write("/");
        Console.Write(monthHired);
        Console.Write("/");
        Console.WriteLine(yearHired);
        Console.Write("Hourly Salary: ");
        Console.WriteLine(hourlySalary);
    }
}
```

2. Save the file as **exercise.cs** in a new folder called **Requests** created inside your CSharp Lessons folder we created in the **previous lesson**
3. Compile the program with **csc exercise.cs**
4. Execute it by typing exercise

```
C:\CSharp Lessons\Requests>csc exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.
```

```
C:\CSharp Lessons\Requests>exercise
```

```
Employee Payroll
Full Name: Alexander Baugh
Date Hired: 12/6/1998
Hourly Salary: 14.58
```

```
C:\CSharp Lessons\Requests>
```

5. After viewing the result, return to Notepad

Requesting String Values

In most assignments of your programs, you will not know the value of a string when writing your application. For example, you may want the user to provide such a string. To request a string (or any of the variables we will see in this lesson), you can call the **Console.Read()** or the **Console.ReadLine()** function and assign it to the name of the variable whose value you want to retrieve. Here is an example:

Coalesce

```
string FirstName;  
Console.Write("Enter First Name: ");  
FirstName = Console.ReadLine();
```

Practical Learning: Reading String Values

1. To request strings from the user, change the file as follows:

```
using System;  
  
class Payroll  
{  
    static void Main(string[] args)  
    {  
        string firstName;  
        string lastName;  
  
        Console.WriteLine();  
        Console.Write("Enter First Name: ");  
        firstName = Console.ReadLine();  
        Console.Write("Enter Last Name: ");  
        lastName = Console.ReadLine();  
  
        Console.WriteLine("\nEmployee Payroll");  
        Console.Write("Full Name: ");  
        Console.Write(firstName);  
        Console.Write(" ");  
        Console.WriteLine(lastName);  
    }  
}
```

2. Compile and execute the program. Here is an example:

```
Enter First Name: Hermine  
Enter Last Name: Toussaing  
  
Employee Payroll  
Full Name:  Hermine Toussaing
```

3. Return to Notepad

Requesting Numbers

In C#, everything the user types is a string and the compiler would hardly analyze it without your explicit asking it to do so. Therefore, if you want to get a number from the user, first request a string, then convert that string into a number. To perform this conversion, the integer data types are equipped with a member function called **Parse**. Therefore, after retrieving a string from the user, you can use the **Parse()** method of the data type you are using to convert the string to it.

Coalesce

Practical Learning: Reading Numeric Values

1. To retrieve various values from the user, change the file as follows:

```
using System;

class Payroll
{
    static void Main(string[] args)
    {
        string firstName;
        string lastName;
        int dayHired;
        int monthHired;
        int yearHired;
        bool isMarried;
        double hourlySalary;

        Console.WriteLine();
        Console.WriteLine("Enter the following pieces of information");
        Console.Write("First Name: ");
        firstName = Console.ReadLine();
        Console.Write("Last Name: ");
        lastName = Console.ReadLine();
        Console.Write("Day Hired: ");
        string d = Console.ReadLine();
        Console.Write("Month Hired: ");
        string m = Console.ReadLine();
        Console.Write("Year Hired: ");
        string y = Console.ReadLine();
        Console.WriteLine("Is the employee married?");
        Console.Write("If Yes, type True. Otherwise, type False: ");
        string married = Console.ReadLine();
        Console.Write("Hourly Salary: ");
        string s = Console.ReadLine();

        dayHired = int.Parse(d);
        monthHired = int.Parse(m);
        yearHired = int.Parse(y);
        isMarried = bool.Parse(married);
        hourlySalary = double.Parse(s);

        Console.WriteLine("\nEmployee Payroll");
        Console.Write("Full Name: ");
        Console.Write(firstName);
        Console.Write(" ");
        Console.WriteLine(lastName);
        Console.Write("Date Hired: ");
        Console.Write(dayHired);
```

Coalesce

```
        Console.Write("/");  
        Console.Write(monthHired);  
        Console.Write("/");  
        Console.WriteLine(yearHired);  
        Console.Write("Married? ");  
        Console.WriteLine(isMarried);  
        Console.Write("Hourly Salary: ");  
        Console.WriteLine(hourlySalary);  
    }  
}
```

2. Save, compile, and execute the exercise. Here is an example:

Enter the following pieces of information
First Name: Youry
Last Name: Bamoch
Day Hired: 15
Month Hired: 11
Year Hired: 2002
Is the employee married?
If Yes, type True. Otherwise, type False: True
Hourly Salary: 22.08

Employee Payroll
Full Name: Youry Bamoch
Date Hired: 15/11/2002
Married? True
Hourly Salary: 22.08

3. Return to Notepad

Formatting Data Display

Introduction

Instead of using two **Write()** functions or a combination of **Write()** and **WriteLine()** function to display data, you can convert a value to a string and display it directly. To do this, you can provide two strings to the **Write()** or **WriteLine()** function:

1. The first part of the string provided to the **Write()** or **WriteLine()** function is the complete string that would display to the user. This first string itself can be made of different sections:
 - a. One section is a string in any way you want it to display
 - b. Another section is a number included between an opening curly bracket "{" and a closing curly bracket "}".

You can put the curly brackets anywhere inside of the string. The first combination of curly brackets must have number 0. The second must have number 1, etc. With this technique, you can create the string anyway you like and use the curly brackets as placeholders anywhere inside of the string. As placeholders, the curly brackets would be used by the next part

Coalesce

2. The second part of the string provided to the **Write()** or **WriteLine()** function is the value that you want to display. It can be one value if you used only one combination of curly brackets with 0 in the first string. If you used different combinations of curly brackets, you can then provide a different value for each one of them in this second part, separating the values with a comma

Here are examples:

```
using System;
```

```
// An Exercise class
class Exercise
{
    static void Main()
    {
        String FullName = "Anselme Bogos";
        int Age = 15;
        double HSalary = 22.74;

        Console.WriteLine("Full Name: {0}", FullName);
        Console.WriteLine("Age: {0}", Age.ToString());
        Console.WriteLine("Distance: {0}", HSalary.ToString());

        Console.WriteLine();
    }
}
```

This would produce:

```
Full Name: Anselme Bogos
Age: 15
Distance: 22.74
```

As mentioned already, the numeric value typed in the curly brackets of the first part is an ordered number. If you want to display more than one value, provide each incremental value in its curly brackets. The syntax used is:

```
Write("To Display {0} {1} {2} {n}", First, Second, Third, nth);
```

You can use the sections between a closing curly bracket and an opening curly bracket to create meaningful sentence. Here is an example:

Practical Learning: Displaying Data

1. To use curly brackets to display data, change the file as follows:

```
using System;
```

Coalesce

```
class Payroll
{
    static void Main(string[] args)
    {
        string firstName;
        string lastName;
        int dayHired;
        int monthHired;
        int yearHired;
        bool isMarried;
        double hourlySalary;

        Console.WriteLine();
        Console.WriteLine("Enter the following pieces of information");
        Console.Write("First Name: ");
        firstName = Console.ReadLine();
        Console.Write("Last Name: ");
        lastName = Console.ReadLine();
        Console.Write("Day Hired: ");
        string d = Console.ReadLine();
        Console.Write("Month Hired: ");
        string m = Console.ReadLine();
        Console.Write("Year Hired: ");
        string y = Console.ReadLine();
        Console.WriteLine("Is the employee married?");
        Console.Write("If Yes, type True. Otherwise, type False: ");
        string married = Console.ReadLine();
        Console.Write("Hourly Salary: ");
        string s = Console.ReadLine();

        dayHired = int.Parse(d);
        monthHired = int.Parse(m);
        yearHired = int.Parse(y);
        isMarried = bool.Parse(married);
        hourlySalary = double.Parse(s);

        Console.WriteLine("\nEmployee Payroll");
        Console.WriteLine("Full Name: {0} {1}", firstName, lastName);
        Console.WriteLine("Married? {0}", isMarried);
        Console.WriteLine("Date Hired: {0}/{1}/{2}",
            dayHired, monthHired, yearHired);
        Console.WriteLine("Hourly Salary: {0}", hourlySalary);
    }
}
```

2. Save, compile, and execute the program. This would produce:

```
Enter the following pieces of information
First Name: James
Last Name: Wallers
Day Hired: 18
```

Coalesce

Month Hired: 2
Year Hired: 1996
Is the employee married?
If Yes, type True. Otherwise, type False: False
Hourly Salary: 18.12

Employee Payroll
Full Name: James Wallers
Married? False
Date Hired: 18/2/1996
Hourly Salary: 18.12

3. Return to Notepad

Type Formatting

To properly display data in a friendly and most familiar way, you can format it. Formatting tells the compiler what kind of data you are using and how you want the compiler to display it to the user. As it happens, you can display a natural number in a common value or, depending on the circumstance, you may prefer to show it as a hexadecimal value. When it comes to double-precision numbers, you may want to display a distance with three values on the right side of the decimal separator and in some cases, you may want to display a salary with only 2 decimal places.

The **System** namespace provides a specific letter that you can pass to the **ToString()** method for each category of data to display. To format a value, you can type the name of the variable followed by a period followed by the **ToString()** method. In the parentheses of **ToString()**, you can include a specific letter or combination inside double-quotes. The letters and their meanings are:

Character		Used For
c	C	Currency values
d	D	Decimal numbers
e	E	Scientific numeric display such as 1.45e ⁵
f	F	Fixed decimal numbers
g	G	General and most common type of numbers
n	N	Natural numbers
r	R	Roundtrip formatting
x	X	Hexadecimal formatting

The necessary letter is passed to the **ToString()** method and is included between double-quotes. Here is an example:

```
using System;

// An Exercise class
class Exercise
{
    static void Main()
    {
        double Distance = 248.38782;
        int Age = 15;
        int NewColor = 3478;
```

Coalesce

```
double HSalary = 22.74, HoursWorked = 35.5018473;
double WeeklySalary = HSalary * HoursWorked;

Console.WriteLine("Distance: {0}", Distance.ToString("E"));
Console.WriteLine("Age: {0}", Age.ToString());
Console.WriteLine("Color: {0}", NewColor.ToString("X"));
Console.WriteLine("Weekly Salary: {0} for {1} hours",
    WeeklySalary.ToString("c"), HoursWorked.ToString("F"));

Console.WriteLine();
}
}
```

This would produce:

Distance: 2.483878E+002
Age: 15
Color: D96
Weekly Salary: \$807.31 for 35.50 hours

As you may have noticed, if you leave the parentheses of **ToString()** empty, the compiler would use a default formatting to display the value.

As opposed to calling **ToString()**, you can use the above letters in the curly brackets of the first part of **Write()** or **WriteLine()**. In this case, after the number in the curly brackets, type the colon operator followed by the letter.

Practical Learning: Formatting Data Display

1. In Notepad, change the file as follows:

```
using System;

class Payroll
{
    static void Main(string[] args)
    {
        ... No Change

        Console.WriteLine("\nEmployee Payroll");
        Console.WriteLine("Full Name: {0} {1}", firstName, lastName);
        Console.WriteLine("Married? {0}", isMarried);
        Console.WriteLine("Date Hired: {0}/{1}/{2}",
            dayHired, monthHired, yearHired);
        Console.WriteLine("Hourly Salary: {0:C}", hourlySalary);
    }
}
```

2. Save it and switch to the Command Prompt. Then compile the file and execute the application
3. Type exit to close

Chapter 3

Conditional Operators

Logical Operations

Introduction

A program is a series of instructions that ask the computer (actually the compiler) to check some situations and to act accordingly. To check such situations, the computer spends a great deal of its time performing comparisons between values. A comparison is a Boolean operation that produces a true or a false result, depending on the values on which the comparison is performed.

A comparison is performed between two values of the same type; for example, you can compare two numbers, two characters, or the names of two cities. On the other hand, a comparison between two disparate value doesn't bear any meaning. For example, it is difficult to compare a telephone number and somebody's age, or a music category and the distance between two points. Like the binary arithmetic operations, the comparison operations are performed on two values. Unlike arithmetic operations where results are varied, a comparison produces only one of two results. The result can be a logical true or false. When a comparison is true, it has an integral value of 1 or positive; that is, a value greater than 0. If the comparison is not true, it is considered false and carries an integral value of 0.

The C# language is equipped with various operators used to perform any type of comparison between similar values. The values could be numeric, strings, or objects (operations on objects are customized in a process referred to as Operator Overloading).

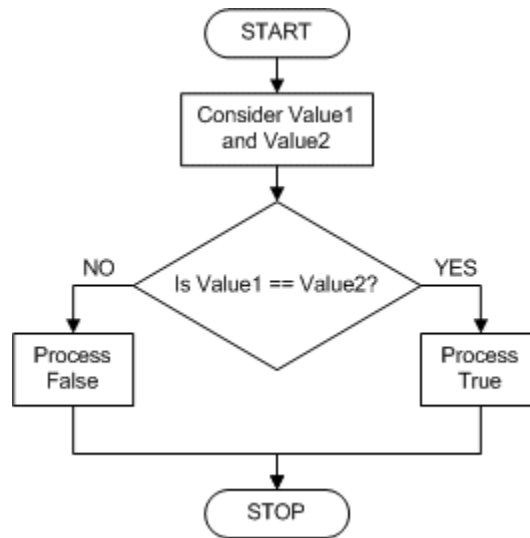
The Equality Operator ==

To compare two variables for equality, C# uses the == operator. Its syntax is:

Value1 == Value2

The equality operation is used to find out whether two variables (or one variable and a constant) hold the same value. From our syntax, the compiler would compare the value of Value1 with that of Value2. If Value1 and Value2 hold the same value, the comparison produces a true result. If they are different, the comparison renders false or 0.

Coalesce



Most of the comparisons performed in C# will be applied to conditional statements; but because a comparison operation produces an integral result, the result of the comparison can be displayed on the monitor screen using a cout extractor. Here is an example:

```
using System;
```

```
class NewProject
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        int Value = 15;
```

```
        Console.Write("Comparison of Value == 32 produces ");
```

```
        Console.WriteLine(Value == 32);
```

```
        Console.Write("\n");
```

```
    }
```

```
}
```

The result of a comparison can also be assigned to a variable. As done with the cout extractor, to store the result of a comparison, you should include the comparison operation between parentheses. Here is an example:

```
using System;
```

```
class NewProject
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        int Value1 = 15;
```

```
        int Value2 = 24;
```

```
        Console.Write("Value 1 = ");
```

```
        Console.WriteLine(Value1);
```

```
        Console.Write("Value 2 = ");
```

Coalesce

```
        Console.WriteLine(Value2);
        Console.WriteLine("Comparison of Value1 == 15 produces ");
        Console.WriteLine(Value1 == 15);
    }
}
```

This would produce:

Value 1 = 15

Value 2 = 24

Comparison of Value1 == 15 produces True

The Logical Not Operator !

When a variable is declared and receives a value (this could be done through initialization or a change of value) in a program, it becomes alive. It can then participate in any necessary operation. The compiler keeps track of every variable that exists in the program being processed. When a variable is not being used or is not available for processing (in visual programming, it would be considered as disabled) to make a variable (temporarily) unusable, you can nullify its value. C# considers that a variable whose value is null is stern. To render a variable unavailable during the evolution of a program, apply the logical not operator which is !. Its syntax is:

!Value

There are two main ways you can use the logical not operator. As we will learn when studying conditional statements, the most classic way of using the logical not operator is to check the state of a variable.

To nullify a variable, you can write the exclamation point to its left. When used like that, you can display its value using the cout extractor. You can even assign it to another variable. Here is an example:

```
using System;

class NewProject
{
    static void Main()
    {
        bool HasAirCondition = true;
        bool DoesIt;

        Console.WriteLine("HasAirCondition = ");
        Console.WriteLine(HasAirCondition);
        DoesIt = !HasAirCondition;
        Console.WriteLine("DoesIt      = ");
        Console.WriteLine(DoesIt);
    }
}
```

This would produce:

Coalesce

HasAirCondition = True
DoesIt = False

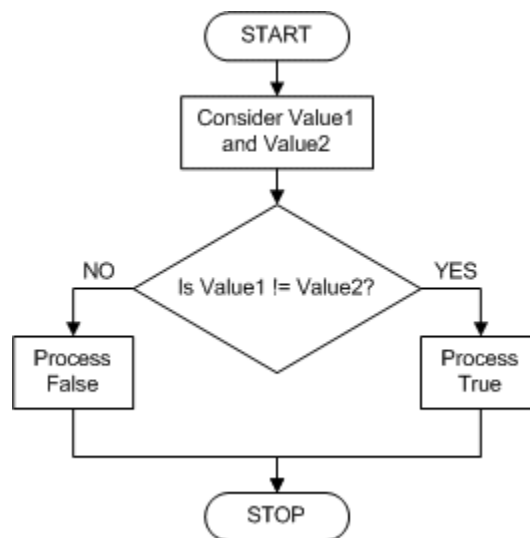
When a variable holds a value, it is "alive". To make it not available, you can "not" it. When a variable has been "notted", its logical value has changed. If the logical value was true, which is 1, it would be changed to false, which is 0. Therefore, you can inverse the logical value of a variable by "notting" or not "notting" it.

The Inequality Operator !=

As opposed to Equality, C# provides another operator used to compare two values for inequality. This operation uses a combination of equality and logical not operators. It combines the logical not ! and a simplified == to produce !=. Its syntax is:

Value1 != Value2

The != is a binary operator (like all logical operator except the logical not, which is a unary operator) that is used to compare two values. The values can come from two variables as in `Variable1 != Variable2`. Upon comparing the values, if both variables hold different values, the comparison produces a true or positive value. Otherwise, the comparison renders false or a null value.



Here is an example:

```
using System;

class NewProject
{
    static void Main()
    {
        int Value1 = 212;
        int Value2 = -46;
        bool Value3 = (Value1 != Value2);
    }
}
```

Coalesce

```
        Console.Write("Value1 = ");  
        Console.WriteLine(Value1);  
        Console.Write("Value2 = ");  
        Console.WriteLine(Value2);  
        Console.Write("Value3 = ");  
        Console.Write(Value3);  
        Console.WriteLine();  
    }  
}
```

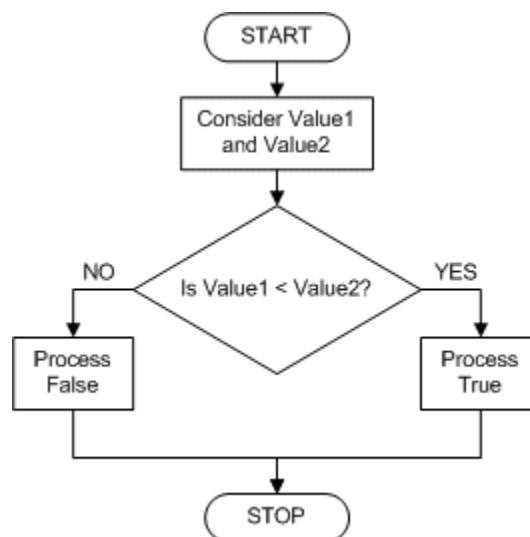
The inequality is obviously the opposite of the equality.

The Comparison for a Lower Value <

To find out whether one value is lower than another, use the < operator. Its syntax is:

Value1 < Value2

The value held by Value1 is compared to that of Value2. As it would be done with other operations, the comparison can be made between two variables, as in Variable1 < Variable2. If the value held by Variable1 is lower than that of Variable2, the comparison produces a true or positive result.



Here is an example:

```
using System;
```

```
class NewProject  
{  
    static void Main()
```

Coalesce

```
{  
    int Value1 = 15;  
    bool Value2 = (Value1 < 24);  
  
    Console.Write("Value 1 = ");  
    Console.WriteLine(Value1);  
    Console.Write("Value 2 = ");  
    Console.WriteLine(Value2);  
    Console.WriteLine();  
}
```

This would produce:

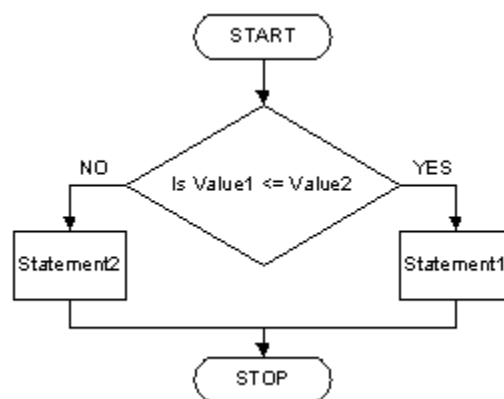
```
Value 1 = 15  
Value 2 = True
```

Combining Equality and Lower Value <=

The previous two operations can be combined to compare two values. This allows you to know if two values are the same or if the first is less than the second. The operator used is <= and its syntax is:

Value1 <= *Value2*

The <= operation performs a comparison as any of the last two. If both *Value1* and *VValue2* hold the same value, result is true or positive. If the left operand, in this case *Value1*, holds a value lower than the second operand, in this case *Value2*, the result is still true.



Here is an example:

```
using System;
```

```
class NewProject  
{  
    static void Main()  
    {
```

Coalesce

```
int Value1 = 15;  
bool Value2 = (Value1 <= 24);  
  
Console.Write("Value 1 = ");  
Console.WriteLine(Value1);  
Console.Write("Value 2 = ");  
Console.WriteLine(Value2);  
Console.WriteLine();  
    }  
}
```

This would produce:

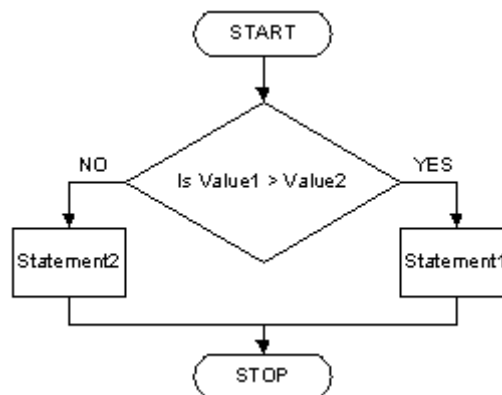
```
Value 1 = 15  
Value 2 = True
```

The Comparison for a Greater Value >

When two values of the same type are distinct, one of them is usually higher than the other. C# provides a logical operator that allows you to find out if one of two values is greater than the other. The operator used for this operation uses the > symbol. Its syntax is:

```
Value1 > Value2
```

Both operands, in this case Value1 and Value2, can be variables or the left operand can be a variable while the right operand is a constant. If the value on the left of the > operator is greater than the value on the right side or a constant, the comparison produces a true or positive value . Otherwise, the comparison renders false or null.



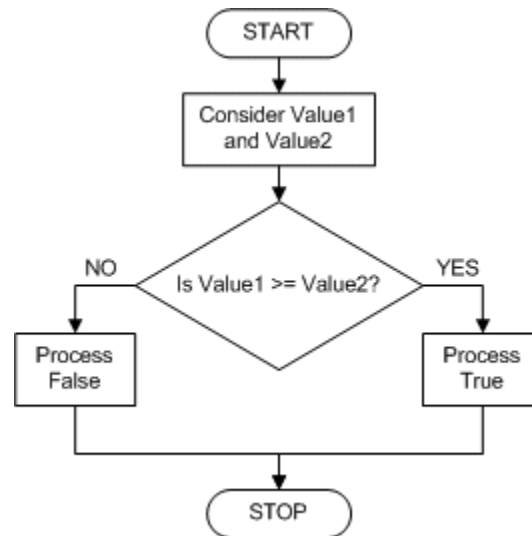
The Greater Than or Equal Operator >=

The greater than or the equality operators can be combined to produce an operator as follows: >=. This is the "greater than or equal to" operator. Its syntax is:

```
Value1 >= Value2
```

Coalesce

A comparison is performed on both operands: Value1 and Value2. If the value of Value1 and that of Value2 are the same, the comparison produces a true or positive value. If the value of the left operand is greater than that of the right operand,, the comparison produces true or positive also. If the value of the left operand is strictly less than the value of the right operand, the comparison produces a false or null result.



Here is a summary table of the logical operators we have studied:

Operator	Meaning	Example	Opposite
==	Equality to	a == b	!=
!=	Not equal to	12 != 7	==
<	Less than	25 < 84	>=
<=	Less than or equal to	Cab <= Tab	>
>	Greater than	248 > 55	<=
>=	Greater than or equal to	Val1 >= Val2	<

Conditional Statements

Conditions

When programming, you will ask the computer to check various kinds of situations and to act accordingly. The computer performs various comparisons of various kinds of statements. These statements come either from you or from the computer itself, while it is processing internal assignments.

Coalesce

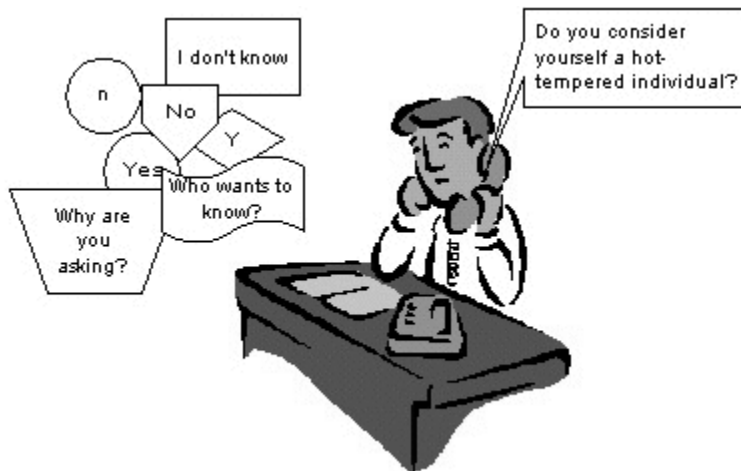
Let's imagine you are writing an employment application and one question would be, "Do you consider yourself a hot-tempered individual?" The source file of such a program would look like this:

```
using System;

class NewProject
{
    static void Main()
    {
        char Answer;
        string Ans;

        Console.WriteLine("Do you consider yourself a hot-tempered individual? ");
        Ans = Console.ReadLine();
        Answer = char.Parse(Ans);
    }
}
```

Some of the answers a user would type are y, yes, Y, Yes, YES, n, N, no, No, NO, I don't know, Sometimes, Why are you asking?, and What do you mean? The variety of these different answers means that you should pay attention to how you structure your programs, you should be clear to the users.



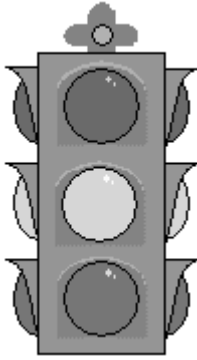
A better version of the line that asks the question would be:

```
Console.WriteLine("Do you consider yourself a hot-tempered individual? (y=Yes/n=No)");
```

This time, although the user can still type anything, at least you have specified the expected answers.

Introduction to Conditional Statements

Coalesce



There are three entities that participate on a traffic light: the lights, the human beings who interact with the light, and the law. The road provides a platform on which these components come together.

The Traffic Light

Everything taken into consideration, a traffic light is made of three light colors: Green – Yellow/Orange – Red. When the light is green, the road is clear for moving in. The red light signals to stop and wait. A yellow light means, “Be careful, it is not safe to proceed right now. Maybe you should wait.” When it is not blinking, the yellow light usually serves as a transition period from green to red. There is no transition from red to green.

The Drivers

There are two main categories of people who deal with the traffic light: the drivers and the walkers. To make our discussion a little simpler, we will consider only the driver. When the light is green, a driver can drive through. When the light is red, the driver is required to stop and wait.

The Law

Rules and regulations dictate that when a driver does not obey the law by stopping to a red light, he is considered to have broken the law and there is a consequence.

The most independent of the three entities is the traffic light. It does not think, therefore it does not make mistakes. It is programmed with a timer or counter that directs it when to act, that is, when to change lights. The second entity, the driver, is a human being who can think and make decisions based on circumstances that are beyond human understanding. A driver can decide to stop at a green light or drive through a red light...

A driver who proceeds through a red light can get a ticket depending on one of two circumstances: either a police officer caught him “hand-in-the-basket” or a special camera took a picture. Worse, if an accident happens, this becomes another story.

The traffic light is sometimes equipped with a timer or counter. We will call it Timer T . It is equipped with three lights: Green, Yellow, and Red. Let's suppose that the light stays green for 45 seconds, then it turns and stays yellow for 5 seconds, and finally it turns and stays red for 1 minute = 60 seconds. At one moment in the day, the timer is set at the beginning or is reset and the light is green: $T = 0$. Since the timer is working fine, it starts counting the seconds 1, 2, 3, 4, ... 45. The light will stay green from $T = 0$ to $T = 45$. When the timer reaches 45, the timer is reset to 0 and starts counting from 0 until it reaches 5; meanwhile, Color = Yellow.

Coalesce

If a Condition is True

In C#, comparisons are made from a statement. Examples of statements are:


- "You are 12 years old"
- "It is raining outside"
- You live in Sydney"

When a driver comes to a traffic light, the first thing he does is to examine the light's color. There are two values the driver would put together: The current light of the traffic and the desired light of the traffic.

Upon coming to the traffic light, the driver would have to compare the traffic light variable with a color he desires the traffic light to have, namely the green light (because if the light is green, then the driver can drive through). The comparison is performed by the driver making a statement such as "The light is green".

After making a statement, the driver evaluates it and compares it to what must be true.

When a driver comes to a traffic light, he would likely expect the light to be green. Therefore, if the light is green (because that is what he is expecting), the result of his examination would receive the Boolean value of TRUE. This produces the following table:

Color	Statement	Boolean Value
	The light is green	true

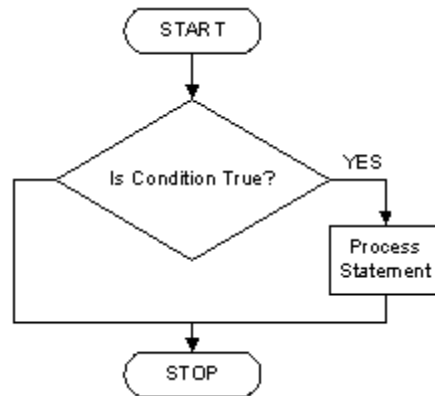
One of the comparisons the computer performs is to find out if a statement is true (in reality, programmers (like you) write these statements and the computer only follows your logic). If a statement is true, the computer acts on a subsequent instruction.

The comparison using the **if** statement is used to check whether a condition is true or false. The syntax to use it is:

if(*Condition*) *Statement*;

If the *Condition* is true, then the compiler would execute the *Statement*. The compiler ignores anything else:

Coalesce



If the statement to execute is (very) short, you can write it on the same line with the condition that is being checked.

Consider a program that is asking a user to answer Yes or No to a question such as "Are you ready to provide your credit card number?". A source file of such a program could look like this:

```
using System;

class NewProject
{
    static void Main()
    {
        char Answer;
        string Ans;

        // Request the availability of a credit card from the user
        Console.Write("Are you ready to provide your credit card number(1=Yes/0=No)? ");
        Ans = Console.ReadLine();
        Answer = char.Parse(Ans);

        // Since the user is ready, let's process the credit card transaction
        if(Answer == '1') Console.WriteLine("\nNow we will get your credit card information.");

    }
}
```

You can write the **if** condition and the statement on different lines; this makes your program easier to read. The above code could be written as follows:

```
using System;

class NewProject
{
    static void Main()
    {
        char Answer;
        string Ans;
```

Coalesce

```
        // Request the availability of a credit card from the user
        Console.WriteLine("Are you ready to provide your credit card
number(1=Yes/0=No)? ");
        Ans = Console.ReadLine();
        Answer = char.Parse(Ans);

        // Since the user is ready, let's process the credit card transaction
        if(Answer == '1')
            Console.WriteLine("\nNow we will get your credit card
information.");
    }
}
```

You can also write the statement on its own line if the statement is too long to fit on the same line with the condition.

Although the (simple) **if** statement is used to check one condition, it can lead to executing multiple dependent statements. If that is the case, enclose the group of statements between an opening curly bracket “{” and a closing curly bracket “}”. Here is an example:

```
using System;

class NewProject
{
    static void Main()
    {
        char Answer;
        string Ans, CreditCardNumber;

        // Request the availability of a credit card from the user
        Console.WriteLine("Are you ready to provide your credit card
number(1=Yes/0=No)? ");
        Ans = Console.ReadLine();
        Answer = char.Parse(Ans);



        // Since the user is ready, let's process the credit card transaction
        if(Answer == '1')
        {
            Console.WriteLine("\nNow we will get your credit card
information.");
            Console.WriteLine("\nPlease enter your credit card number
without spaces: ");
            CreditCardNumber = Console.ReadLine();
        }
    }
}
```

If you omit the brackets, only the statement that immediately follows the condition would be executed.



Coalesce

Using the Logical Not



When a driver comes to a light that he expects to be green, we saw that he would use a statement such as, "The light is green". If in fact the light is green, we saw that the statement would lead to a true result. If the light is not green, the "The light is green" statement produces a false result. This is shown in the following table:

Color	Statement	Boolean Value
	The light is green	true
	The light is green	false

As you may realize already, in Boolean algebra, the result of performing a comparison depends on how the *Condition* is formulated. If the driver is approaching a light that he is expecting to display any color other than green, he would start from a statement such as "The light is not green". If the light IS NOT green, the expression "The light is not green" is true (very important). This is illustrated in the following table:

Color	Statement	Boolean Value
	The light is green	true
	The light is not green	true



The "The light is not green" statement is expressed in Boolean algebra as "Not the light is green". Instead of writing "Not the light is green", in C#, using the logical Not operator, you would formulate the statement as, !"The light is green". Therefore, if P means "The light is green", you can express the negativity of P as !P. The Boolean table produced is:

Color	Statement	Boolean Value	Symbol
	The light is green	true	P
	The light is not green	false	!P

When a statement is true, its Boolean value is equivalent to a non-zero integer such as 1. Otherwise, if a statement produces a false result, it is given a 0 value. Therefore, our table would be:

Color	Statement	Boolean Value	Integer Value
-------	-----------	---------------	---------------

Coalesce

	The light is green	true	1
	The light is not green	false	0

Otherwise: if...else

The **if** condition is used to check one possibility and ignore anything else. Usually, other conditions should be considered. In this case, you can use more than one if statement. For example, on a program that asks a user to answer Yes or No, although the positive answer is the most expected, it is important to offer an alternate statement in case the user provides another answer. Here is an example:

```
using System;
```

```
class NewProject
{
    static void Main()
    {
        char Answer;
        string Ans;

        // Request the availability of a credit card from the user
        Console.Write("Do you consider yourself a hot-tempered individual(y=Yes/n=No)? ");
        Ans = Console.ReadLine();
        Answer = char.Parse(Ans);

        if( Answer == 'y' ) // First Condition
        {
            Console.WriteLine("\nThis job involves a high level of self-control.");
            Console.WriteLine("We will get back to you.\n");
        }
        if( Answer == 'n' ) // Second Condition
            Console.WriteLine("\nYou are hired!\n");
    }
}
```

Here is an example of running the program:

```
Do you consider yourself a hot-tempered individual(y=Yes/n=No)? y
```

```
This job involves a high level of self-control.
We will get back to you.
```

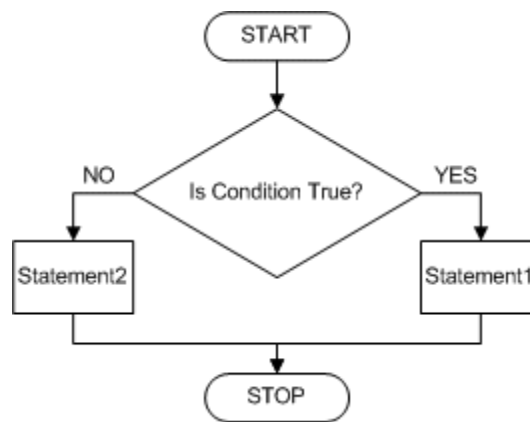
```
Press any key to continue
```

Coalesce

The problem with the above program is that the second **if** is not an alternative to the first, it is just another condition that the program has to check and execute after executing the first. On that program, if the user provides y as the answer to the question, the compiler would execute the content of its statement and the compiler would execute the second **if** condition.

You can also ask the compiler to check a condition; if that condition is true, the compiler will execute the intended statement. Otherwise, the compiler would execute alternate statement. This is performed using the syntax:

```
if(Condition)
    Statement1;
else
    Statement2;
```



The above program would better be written as:

```
using System;

class NewProject
{
    static void Main()
    {
        char Answer;
        string Ans;

        // Request the availability of a credit card from the user
        Console.Write("Do you consider yourself a hot-tempered
individual(y=Yes/n=No)? ");
        Ans = Console.ReadLine();
        Answer = char.Parse(Ans);

        if( Answer == 'y' ) // First Condition
        {
            Console.WriteLine("\nThis job involves a high level of self-
control.");

            Console.WriteLine("We will get back to you.\n");
        }
        else // Second Condition
```


Coalesce

```
        Console.WriteLine("\nYou are hired!\n");
    }
}
```

The Conditional Operator (?:)

The conditional operator behaves like a simple **if...else** statement. Its syntax is:

Condition ? *Statement1* : *Statement2*;

The compiler would first test the *Condition*. If the *Condition* is true, then it would execute *Statement1*, otherwise it would execute *Statement2*. When you request two numbers from the user and would like to compare them, the following program would do find out which one of both numbers is higher. The comparison is performed using the conditional operator:

```
using System;

class NewProject
{
    static void Main()
    {
        int Number1, Number2, Maximum;
        string Num1, Num2;

        Console.WriteLine("Enter first numbers: ");
        Num1 = Console.ReadLine();
        Console.WriteLine("Enter second numbers: ");
        Num2 = Console.ReadLine();

        Number1 = int.Parse(Num1);
        Number2 = int.Parse(Num2);

        Maximum = (Number1 < Number2) ? Number2 : Number1;

        Console.WriteLine("\nThe maximum of ");
        Console.WriteLine(Number1);
        Console.WriteLine(" and ");
        Console.WriteLine(Number2);
        Console.WriteLine(" is ");
        Console.WriteLine(Maximum);

        Console.WriteLine();
    }
}
```

Here is an example of running the program;

```
Enter first numbers: 244
Enter second numbers: 68
```

Coalesce

The maximum of 244 and 68 is 244

Conditional Statements: if...else if and if...else if...else

The previous conditional formula is used to execute one of two alternatives. Sometimes, your program will need to check many more than that. The syntax for such a situation is:

```
if(Condition1)
    Statement1;
else if(Condition2)
    Statement2;
```

An alternative syntax would add the last else as follows:

<pre>if(Condition1) Statement1; else if(Condition2) Statement2; else Statement-n;</pre>	<pre>if(Condition1) Statement1; else if(Condition2) Statement2; else if(Condition3) Statement3; else Statement-n;</pre>
---	---

The compiler will check the first condition. If Condition1 is true, it will execute Statement1. If Condition1 is false, then the compiler will check the second condition. If Condition2 is true, it will execute Statement2. When the compiler finds a Condition-n to be true, it will execute its corresponding statement. If that Condition-n is false, the compiler will check the subsequent condition. This means you can include as many conditions as you see fit using the else if statement. If after examining all the known possible conditions you still think that there might be an unexpected condition, you can use the optional single else.

A program we previously wrote was considering that any answer other than y was negative. It would be more professional to consider a negative answer because the program anticipated one. Therefore, here is a better version of the program:

```
using System;

class NewProject
{
    static void Main()
    {
        char Answer;
        string Ans;

        // Request the availability of a credit card from the user
        Console.WriteLine("Do you consider yourself a hot-tempered
individual(y=Yes/n=No)? ");
        Ans = Console.ReadLine();
        Answer = char.Parse(Ans);
```

Coalesce

```
        if( Answer == 'y' ) // First Condition
        {
            Console.WriteLine("\nThis job involves a high level of self-
control.");
            Console.WriteLine("We will get back to you.\n");
        }
        else if( Answer == 'n' ) // Alternative
            Console.Write("\nYou are hired!\n");
        else
            Console.Write("\nThat's not a valid answer!\n");
    }
}
```

The switch Statement

When defining an expression whose result would lead to a specific program execution, the switch statement considers that result and executes a statement based on the possible outcome of that expression, this possible outcome is called a case. The different outcomes are listed in the body of the switch statement and each case has its own execution, if necessary. The body of a switch statement is delimited from an opening to a closing curly brackets: “{“ to “}”. The syntax of the switch statement is:

```
switch(Expression)
{
    case Choice1:
        Statement1;
    case Choice2:
        Statement2;
    case Choice-n:
        Statement-n;
}
```

The expression to examine is an integer. Since an enumeration (enum) and the character (char) data types are just other forms of integers, they can be used too. Here is an example of using the switch statement:

```
using System;

class NewProject
{
    static void Main()
    {
        int Number;
        string Nbr;

        Console.Write("Type a number between 1 and 3: ");
        Nbr = Console.ReadLine();
        Number = int.Parse(Nbr);
    }
}
```

Coalesce

```
switch(Number)
{
    case 1:
        Console.WriteLine("\nYou typed 1.");
        break;
    case 2:
        Console.WriteLine("\nYou typed 2.");
        break;
    case 3:
        Console.WriteLine("\nYou typed 3.");
        break;
}

Console.WriteLine();
}
```

The program above would request a number from the user. If the user types 1, it would execute the first, the second, and the third cases. If she types 2, the program would execute the second and third cases. If she supplies 3, only the third case would be considered. If the user types any other number, no case would execute.

When establishing the possible outcomes that the **switch** statement should consider, at times there will be other possibilities other than those listed and you will be likely to consider them. This special case is handled by the **default** keyword. The **default** case would be considered if none of the listed cases matches the supplied answer. The syntax of the **switch** statement that considers the default case would be:

```
switch(Expression)
{
    case Choice1:
        Statement1;
    case Choice2:
        Statement2;
    case Choice-n:
        Statement-n;
    default:
        Other-Possibility;
}
```

Therefore another version of the program above would be

using System;

```
class NewProject
{
    static void Main()
    {
        int Number;
        string Nbr;
```

Coalesce

```
Console.Write("Type a number between 1 and 3: ");
Nbr = Console.ReadLine();
Number = int.Parse(Nbr);

switch(Number)
{
    case 1:
        Console.Write("\nYou typed 1.");
        break;
    case 2:
        Console.Write("\nYou typed 2.");
        break;
    case 3:
        Console.Write("\nYou typed 3.");
        break;
    default:
        Console.Write(Number);
        Console.WriteLine(" is out of the requested range.");
        break;
}

Console.WriteLine();
}
```

Here is an example of running the program:

```
Type a number between 1 and 3: 8
8 is out of the requested range.
```

Coalesce

Chapter 4

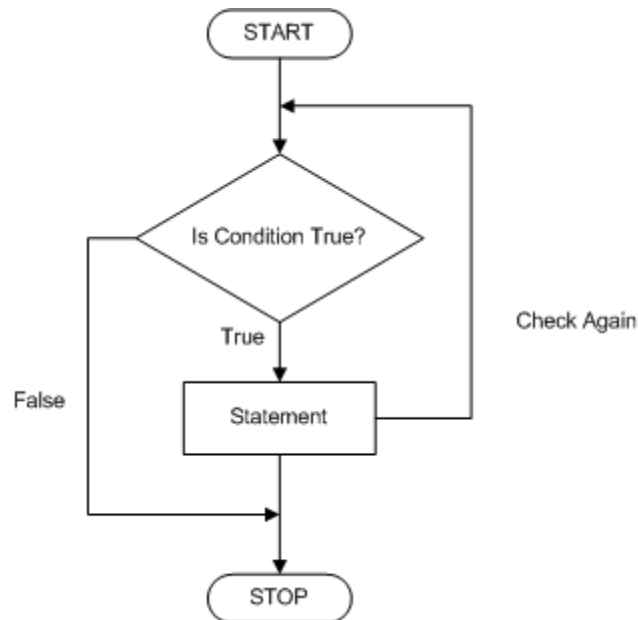
Counting and Looping

The while Statement

The C# language provides a set of control statements that allows you to conditionally control data input and output. These controls are referred to as loops.

The **while** statement examines or evaluates a condition. The syntax of the **while** statement is:

while(*Condition*) *Statement*;



To execute this expression, the compiler first examines the *Condition*. If the *Condition* is true, then it executes the *Statement*. After executing the *Statement*, the *Condition* is checked again. AS LONG AS the *Condition* is true, it will keep executing the *Statement*. When or once the *Condition* becomes false, it exits the loop.

Here is an example:

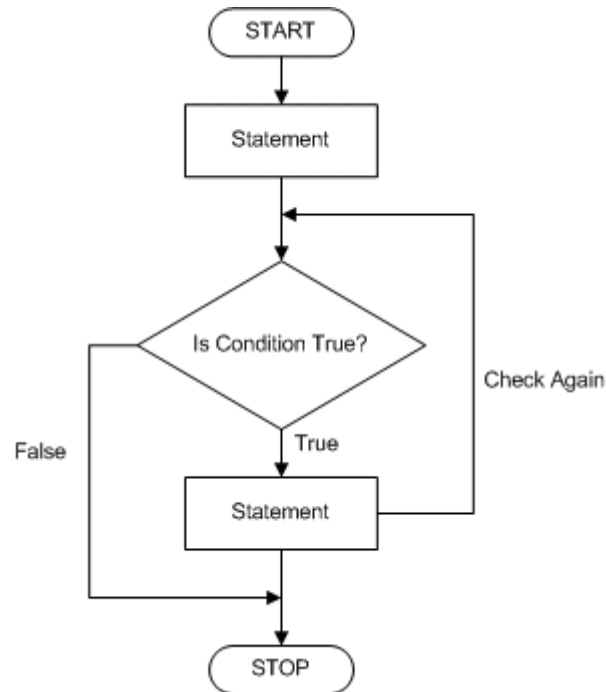
```
int Number;

while( Number <= 12 )
{
    Console.Write("Number ");
    Console.WriteLine(Number);
    Number++;
}
```

Coalesce

}

To effectively execute a **while** condition, you should make sure you provide a mechanism for the compiler to use a get a reference value for the condition, variable, or expression being checked. This is sometimes in the form of a variable being initialized although it could be some other expression. Such a while condition could be illustrated as follows:



An example would be:

using System;

```
class NewProject
{
    static void Main()
    {
        int Number = 0;

        while( Number <= 12 )
        {
            Console.Write("Number ");
            Console.WriteLine(Number);
            Number++;
        }

        Console.WriteLine();
    }
}
```

This would produce:

Coalesce

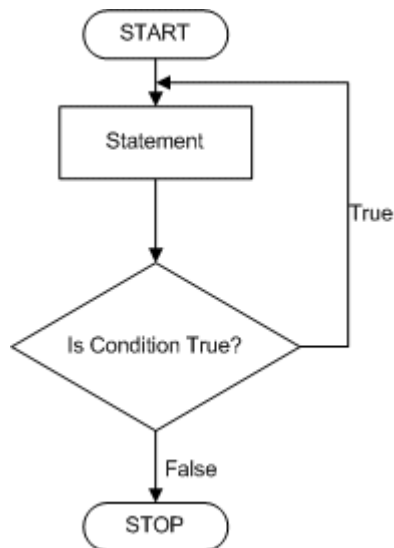
Number 0
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
Number 10
Number 11
Number 12

Press any key to continue

The do...while Statement

The **do...while** statement uses the following syntax:

do *Statement* **while** (*Condition*);



The **do...while** condition executes a *Statement* first. After the first execution of the *Statement*, it examines the *Condition*. If the *Condition* is true, then it executes the *Statement* again. It will keep executing the *Statement* AS LONG AS the *Condition* is true. Once the *Condition* becomes false, the looping (the execution of the *Statement*) would stop.

If the *Statement* is a short one, such as made of one line, simply write it after the do keyword. Like the if and the while statements, the *Condition* being checked must be included between parentheses. The whole do...while statement must end with a semicolon.

Another version of the counting program seen previously would be:

using System;

Coalesce

```
class NewProject
{
    static void Main()
    {
        int Number = 0;

        do {
            Console.Write("Number ");
            Console.WriteLine(Number);
            Number++;
        } while( Number <= 12 );

        Console.WriteLine();
    }
}
```

If the Statement is long and should span more than one line, start it with an opening curly bracket and end it with a closing curly bracket.

The **do...while** statement can be used to insist on getting a specific value from the user. For example, since our ergonomic program would like the user to sit down for the subsequent exercise, you can modify your program to continue only once she is sitting down. Here is an example on how you would accomplish that:

```
using System;

class NewProject
{
    static void Main()
    {
        char SittingDown;
        string SitDown;

        Console.Write("For the next exercise, you need to be sitting down\n");

        do {
            Console.Write("Are you sitting down now(y/n)? ");
            SitDown = Console.ReadLine();
            SittingDown = char.Parse(SitDown);
        } while( !(SittingDown == 'y') );

        Console.WriteLine();
    }
}
```

Here is an example of running the program:

```
For the next exercise, you need to be sitting down
Are you sitting down now(y/n)? t
Are you sitting down now(y/n)? h
Are you sitting down now(y/n)? G
```

Coalesce

Are you sitting down now(y/n)? y

Press any key to continue

The for Statement

The **for** statement is typically used to count a number of items. At its regular structure, it is divided in three parts. The first section specifies the starting point for the count. The second section sets the counting limit. The last section determines the counting frequency. The syntax of the for statement is:

for(*Start*; *End*; *Frequency*) *Statement*;

The Start expression is a variable assigned the starting value. This could be Count = 0; The End expression sets the criteria for ending the counting. An example would be Count < 24; this means the counting would continue as long as the Count variable is less than 24. When the count is about to reach 24, because in this case 24 is excluded, the counting would stop. To include the counting limit, use the <= or >= comparison operators depending on how you are counting. The Frequency expression would let the compiler know how many numbers to add or subtract before continuing with the loop. This expression could be an increment operation such as ++Count.

Here is an example that applies the for statement:

```
using System;
```

```
class NewProject
{
    static void Main()
    {
        for(int Count = 0; Count <= 12; Count++)
        {
            Console.WriteLine("Number ");
            Console.WriteLine(Count);
        }

        Console.WriteLine();
    }
}
```

The C# compiler recognizes that a variable declared as the counter of a for loop is available only in that for loop. This means the scope of the counting variable is confined only to the for loop. This allows different for loops to use the same counter variable. Here is an example:

```
using System;
```

```
class NewProject
{
    static void Main()
    {
        for(int Count = 0; Count <= 12; Count++)
```

Coalesce

```
        {
            Console.Write("Number ");
            Console.WriteLine(Count);
        }

        for(int Count = 10; Count >= 2; Count--)
        {
            Console.Write("Number ");
            Console.WriteLine(Count);
        }

        Console.WriteLine();
    }
}
```

Creating Expressions

Accessories for Conditional Statements

Nesting Conditions

A condition can be created inside of another to write a more effective statement. This is referred to as nesting conditions. Almost any condition can be part of another and multiple conditions can be included inside of others.

As we have learned, different conditional statements are applied in specific circumstances. In some situations, they are interchangeable or one can be applied just like another, which becomes a matter of choice. Statements can be combined to render a better result with each playing an appropriate role.

To continue with our ergonomic program, imagine that you would really like the user to sit down and your program would continue only once she answers that she is sitting down, you can use the do...while statement to wait for the user to sit down; but as the do...while is checking the condition, you can insert an if statement to enforce your request. Here is an example of how you can do it:

```
using System;
```

```
class NewProject
{
    static void Main()
    {
        char SittingDown;
        string SitDown;

        do {
            Console.Write("Are you sitting down now(y/n)? ");
            SitDown = Console.ReadLine();
            SittingDown = char.Parse(SitDown);
        }
```

Coalesce

```
        if( SittingDown != 'y' )
            Console.WriteLine("Could you please sit down for the next exercise? ");
    } while( !(SittingDown == 'y') );

    Console.WriteLine();
}
}
```

Here is an example of running the program:

```
Are you sitting down now(y/n)? n
Could you please sit down for the next exercise?
Are you sitting down now(y/n)? g
Could you please sit down for the next exercise?
Are you sitting down now(y/n)? y
```

Press any key to continue

One of the reasons you would need to nest conditions is because one would lead to another. Sometimes, before checking one condition, another primary condition would have to be met. The ergonomic program we have been simulating so far is asking the user whether she is sitting down. Once the user is sitting down, you would write an exercise she would perform. Depending on her strength, at a certain time, one user will be tired and want to stop while for the same amount of previous exercises, another user would like to continue. Before continuing with a subsequent exercise, you may want to check whether the user would like to continue. Of course, this would be easily done with:

```
using System;

class NewProject
{
    static void Main()
    {
        char SittingDown;
        string SitDown;

        do {
            Console.Write("Are you sitting down now(y/n)? ");
            SitDown = Console.ReadLine();
            SittingDown = char.Parse(SitDown);

            if( SittingDown != 'y' )
                Console.WriteLine("Could you please sit down for the next exercise? ");
        } while( !(SittingDown == 'y') );

        Console.Write("Wonderful. Now we will continue today's exercise...");
        Console.WriteLine("\n...\nEnd of exercise");

        char WantToContinue;

        Console.Write("\nDo you want to continue(y=Yes/n=No)? ");
        string ToContinue = Console.ReadLine();
    }
}
```

Coalesce

```
        WantToContinue = char.Parse(ToContinue);

        Console.WriteLine();
    }
}
```

If the user answers No, you can stop the program. If she answers Yes, you would need to continue the program with another exercise. Because the user answered Yes, the subsequent exercise would be included in the previous condition because it does not apply for a user who wants to stop. In this case, one “if” could be inserted inside of another. Here is an example:

```
using System;

class NewProject
{
    static void Main()
    {
        char SittingDown;
        string SitDown;

        do
        {
            Console.Write("Are you sitting down now(y/n)? ");
            SitDown = Console.ReadLine();
            SittingDown = char.Parse(SitDown);

            if( SittingDown != 'y')
                Console.WriteLine("Could you please sit down for the
next exercise? ");
        }while( SittingDown != 'y' );

        Console.WriteLine("Wonderful. Now we will continue today's exercise...");
        Console.WriteLine("\n...\nEnd of exercise\n");

        char WantToContinue;

        Console.Write("\nDo you want to continue(1=Yes/0=No)? ");
        string ToContinue = Console.ReadLine();
        WantToContinue = char.Parse(ToContinue);

        if(WantToContinue == '1')
        {
            char LayOnBack;

            Console.WriteLine("Good. For the next exercise, you should lay on
your back");

            Console.Write("Are you laying on your back(1=Yes/0=No)? ");
            string Lay = Console.ReadLine();
            LayOnBack = char.Parse(Lay);

            if(LayOnBack == '1')
```

Coalesce

```
exercise.");
        Console.WriteLine("Great.\nNow we will start the next
        else
        Console.WriteLine("\nWell, it looks like you are getting
tired...");
    }
    else
        Console.WriteLine("We had enough today");
        Console.WriteLine("We will stop the session now\nThanks.\n");
    }
}
```

In the same way, you can nest statements as you see fit. The goal is to provide an efficient and friendly application. You can insert and nest statements that provide valuable feedback to the user while minimizing boredom. The above version of the program can be improved as followed:

```
using System;
```

```
class NewProject
{
```

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

```
        char SittingDown;
        string SitDown;
```

```
        do
        {
```

```
            Console.Write("Are you sitting down now(y/n)? ");
            SitDown = Console.ReadLine();
            SittingDown = char.Parse(SitDown);
```

```
            if( SittingDown != 'y' )
```

```
                Console.WriteLine("Could you please sit down for the next exercise?");
```

```
        } while( SittingDown != 'y' );
```

```
        Console.WriteLine("Wonderful. Now we will continue today's exercise...");
```

```
        Console.WriteLine("\n...\nEnd of exercise\n");
```

```
        char WantToContinue;
```

```
        Console.Write("\nDo you want to continue(1=Yes/0=No)? ");
```

```
        string ToContinue = Console.ReadLine();
```

```
        WantToContinue = char.Parse(ToContinue);
```

```
        if(WantToContinue == '1')
```

```
        {
```

```
            char LayOnBack;
```

```
            Console.WriteLine("Good. For the next exercise, you should lay on your back");
```

```
            Console.Write("Are you laying on your back(1=Yes/0=No)? ");
```

Coalesce

```
        string Lay = Console.ReadLine();
        LayOnBack = char.Parse(Lay);

        if(LayOnBack == '0')
        {
            char Ready;

            do
            {
                Console.WriteLine("Please lay on your back");
                Console.Write("\nAre you ready(1=Yes/0=No)? ");
                string R = Console.ReadLine();

                Ready = char.Parse(R);
            }while(Ready == '0');
        }
        else if(LayOnBack == '1')
            Console.WriteLine("\nGreat.\nNow we will start the next exercise.");
        else
            Console.WriteLine("\nWell, it looks like you are getting tired...");
    }
    else
        Console.WriteLine("\nWe had enough today");

    Console.WriteLine("\nWe will stop the session now\nThanks.\n");
}
}
```

The break Statement

The break statement is used to stop a loop for any reason or condition the programmer sees considers fit. The break statement can be used in a while condition to stop an ongoing action. The syntax of the break statement is simply:

break;

Although made of only one word, the break statement is a complete statement; therefore, it can (and should always) stay on its own line (this makes the program easy to read).

The break statement applies to the most previous conditional statement to it; provided that previous statement is applicable.

The following program would display the letter d continuously unless something or somebody stops it. A break statement is inserted to stop this ever looping process:

```
using System;

class NewProject
{
    static void Main()
    {
        char Letter = 'd';
```

Coalesce

```
        while( Letter <= 'n' )
        {
            Console.Write("Letter ");
            Console.WriteLine(Letter);
            break;
        }
    }
```

The **break** statement can also be used in a do...while or a for loop the same way.

The break statement is typically used to handle the cases in a switch statement. We saw earlier that all cases in a switch would execute starting where a valid statement is found.

Consider the program we used earlier to request a number from 1 to 3, a better version that involves a break in each case would allow the switch to stop once the right case is found. Here is a new version of that program:

```
using System;

class NewProject
{
    static void Main()
    {
        int Number;

        Console.Write("Type a number between 1 and 3: ");
        string Nbr = Console.ReadLine();
        Number = int.Parse(Nbr);

        switch(Number)
        {
            case 1:
                Console.WriteLine("\nYou typed 1.");
                break;
            case 2:
                Console.WriteLine("\nYou typed 2.");
                break;
            case 3:
                Console.WriteLine("\nYou typed 3.");
                break;
            default:
                Console.Write(Number);
                Console.WriteLine(" is out of the requested range.");
                break;
        }
    }
}
```

Even when using the break statement, the switch allows many case to execute as one. To do this, as we saw when not using the break, type two cases together. This technique is useful when

Coalesce

validating letters because the letters could be in uppercase or lowercase. This illustrated in the following program:

```
using System;

class NewProject
{
    static void Main()
    {
        char Letter;

        Console.Write("Type a letter: ");
        string Ltr = Console.ReadLine();
        Letter = char.Parse(Ltr);

        switch( Letter )
        {
            case 'a':
            case 'A':
            case 'e':
            case 'E':
            case 'i':
            case 'I':
            case 'o':
            case 'O':
            case 'u':
            case 'U':
                Console.Write("The letter you typed, ");
                Console.Write(Letter);
                Console.WriteLine(", is a vowel\n");
                break;

            case 'b':case 'c':case 'd':case 'f':case 'g':case 'h':case 'j':
            case 'k':case 'l':case 'm':case 'n':case 'p':case 'q':case 'r':
            case 's':case 't':case 'v':case 'w':case 'x':case 'y':case 'z':
                Console.Write(Letter);
                Console.WriteLine(" is a lowercase consonant\n");
                break;

            case 'B':case 'C':case 'D':case 'F':case 'G':case 'H':case 'J':
            case 'K':case 'L':case 'M':case 'N':case 'P':case 'Q':case 'R':
            case 'S':case 'T':case 'V':case 'W':case 'X':case 'Y':case 'Z':
                Console.Write(Letter);
                Console.WriteLine(" is a consonant in uppercase\n");
                break;

            default:
                Console.Write("The symbol ");
                Console.Write(Letter);
                Console.WriteLine(" is not an alphabetical letter\n");
                break;
        }
    }
}
```

Coalesce

```
}  
}
```

This would produce:

Type a letter: g
g is a lowercase consonant

Press any key to continue

The **switch** statement is also used with an enumerator that controls cases. This is also a good place to use the break statement to decide which case applies. An advantage of using an enumerator is its ability to be more explicit than a regular integer.

To use an enumerator, define it and list each one of its members for the case that applies. Remember that, by default, the members of an enumerator are counted with the first member having a value of 0, the second is 1, etc. Here is an example of a switch statement that uses an enumerator.

```
using System;  
  
class NewProject  
{  
    enum TEmploymentStatus { esFullTime, esPartTime, esContractor, esNS };  
  
    static void Main()  
    {  
        int EmplStatus;  
  
        Console.WriteLine("Employee's Contract Status: ");  
        Console.WriteLine("0 - Full Time | 1 - Part Time");  
        Console.WriteLine("2 - Contractor | 3 - Other");  
        Console.Write("Status: ");  
        string Status = Console.ReadLine();  
        EmplStatus = int.Parse(Status);  
  
        Console.WriteLine();  
  
        switch( EmplStatus )  
        {  
            case (int)TEmploymentStatus.esFullTime:  
                Console.WriteLine("Employment Status: Full Time");  
                Console.WriteLine("Employee's Benefits: Medical  
Insurance");  
  
                Console.WriteLine(" Sick Leave");  
                Console.WriteLine(" Maternal Leave");  
                Console.WriteLine(" Vacation Time");  
                Console.WriteLine(" 401K");  
                break;
```

Coalesce

```
        case (int)TEmploymentStatus.esPartTime:
            Console.WriteLine("Employment Status: Part Time");
            Console.WriteLine("Employee's Benefits: Sick Leave");
            Console.WriteLine(" Maternal Leave");
            break;

        case (int)TEmploymentStatus.esContractor:
            Console.WriteLine("Employment Status: Contractor");
            Console.WriteLine("Employee's Benefits: None");
            break;

        case (int)TEmploymentStatus.esNS:
            Console.WriteLine("Employment Status: Other");
            Console.WriteLine("Status Not Specified");
            break;

        default:
            Console.WriteLine("Unknown Status\n");
            break;
    }
}
```

The continue Statement

The continue statement uses the following syntax:

continue;

When processing a loop, if the statement finds a false value, you can use the continue statement inside of a while, do...while or a for conditional statements to ignore the subsequent statement or to jump from a false Boolean value to the subsequent valid value, unlike the break statement that would exit the loop. Like the break statement, the continue keyword applies to the most previous conditional statement and should stay on its own line.

The following programs asks the user to type 4 positive numbers and calculates the sum of the numbers by considering only the positive ones. If the user types a negative number, the program manages to ignore the numbers that do not fit in the specified category:

```
using System;

class NewProject
{
    static void Main()
    {
        // Declare necessary variables
        int posNumber, Sum = 0;
        string Nbr;

        // Request 4 positive numbers from the user
        Console.WriteLine("Type 4 positive numbers.");
```

Coalesce

```
// Make sure the user types 4 positive numbers
for( int Count = 1; Count <= 4; Count++ )
{
    Console.Write("Number: ");
    Nbr = Console.ReadLine();
    posNumber = int.Parse(Nbr);

    // If the number typed is not positive, ignore it
    if( posNumber < 0 )
        continue;

    // Add each number to the sum
    Sum += posNumber;
}

// Display the sum
Console.Write("Sum of the numbers you entered = ");
Console.WriteLine(Sum);
}
```

This would produce:

```
Type 4 positive numbers.
Number: 3
Number: 178
Number: 2394
Number: 52
Sum of the numbers you entered = 2627
Press any key to continue
```

The goto Statement

The **goto** statement allows a program execution to jump to another section of the function in which it is being used.

In order to use the **goto** statement, insert a name on a particular section of your function so you can refer to that name. The name, also called a label, is made of one word and follows the rules we have learned about C++ names (the name can be anything), then followed by a colon. The following program uses a for loop to count from 0 to 12, but when it encounters 5, it jumps to a designated section of the program:

```
using System;
```

```
class NewProject
{
    static void Main()
    {
        for(int Count = 0; Count <= 12; ++Count)
        {
            Console.Write("Count ");
```

Coalesce

```
        Console.WriteLine(Count);

        if( Count == 5 )
            goto MamaMia;
    }

    MamaMia:
        Console.Write("Stopped at 5");

    Console.WriteLine();
}
}
```

This would produce:

```
Count 0
Count 1
Count 2
Count 3
Count 4
Count 5
Stopped at 5
Press any key to continue
```

Logical Operations on Statements

The conditional Statements we have used so far were applied to single situations. You can combine statements using techniques of logical thinking to create more complex and complete expressions. One way to do this is by making sure that two conditions are met for the whole expression to be true. On the other hand, one or the other of two conditions can produce a true condition, as long as one of them is true. This is done with logical conjunction or disjunction.

Logical Conjunction: AND

The law of the traffic light states that if a driver drives through a red light, he or she has broken the law. Three things happen here:

1. The traffic light is red
2. The driver is driving
3. The law is broken

Let's segment these expressions and give each a name. The first statement will be called L. Therefore,

$L \iff$ The traffic light is red

The second statement will be called D. This means

$D \iff$ The driver is driving through the light

Coalesce

The last statement will be called B, which means

$B \iff \text{The law is broken}$

Whenever the traffic light is red, the “The traffic light is red” statement is true. Whenever a driver is driving, the “The driver is driving” statement is true, which means D is true. Whenever the law is broken, the “The law is broken” statement is true. When a statement is true, it receives a Boolean value of true:

L	D	B
true	true	true

These three statements are completely independent when each is stated in its own sentence. The third bears any consideration only when the first two are combined. Therefore, the third statement is a consequence or a result. The fact that a driver is driving and/or a light is red or displays any color, does not make a law broken. The law is broken only when or IF a driver drives through a red light. This means L and D have to be combined to produce B.

A combination of the first two statements means you need Statement1 AND Statement2. Combining Statement1 AND Statement2 means L AND D that produces

“The traffic light is red” AND “The driver is driving through the light”

In C++, the AND keyword is called an operator because it applies for one or more variable. The AND operator is specifically called a binary operator because it is used on two variables. The AND operator is used to concatenate or add two statements or expressions. It is represented by &&. Therefore, a concatenation of L and D would be written as L && D. Logically, what does the combination mean?

When the traffic light is red, L is true. If a driver is driving through the light, D is true. If the driver is driving through the light that is red, this means L && D. Then the law is broken:

L	D	L && D	B
true	true	true	TRUE

When the traffic light is not red, regardless of the light’s color, L is false. If a driver drives through it, no law is broken. Remember, not only should you drive through a green light, but also you are allowed to drive through a yellow light. Therefore, B is false:

L	D	L && D	B
false	true	false	FALSE

If the traffic light is red, L is true. If no driver drives through it, D is false, and no law is broken. When no law is broken, B, which is the result of L && D, is false:

L	D	L && D	B
true	false	false	FALSE

Coalesce

If the light is not red, L is false. If no driver drives through it, D is false. Consequently, no law is broken. B, which is the result of L && D, is still false:

L	D	L && D	B
false	false	false	FALSE

From our tables, the law is broken only when the light is red AND a driver drives through it. This produces:

L	D	L && D	B
true	true	true	TRUE
false	true	false	FALSE
true	false	false	FALSE
false	false	false	FALSE

The logical conjunction operator **&&** is used to check that the combination of two statements results in a true condition. This is used when one condition cannot satisfy the intended result. Consider a pizza application whose valid sizes are 1 for small, 2 for medium, 3 for large, and 4 for jumbo. When a clerk uses this application, you would usually want to make sure that only a valid size is selected to process an order. After the clerk has selected a size, you can use a logical conjunction to validate the range of the item's size. Such a program could be written (or started) as follows:

```
using System;

class NewProject
{
    static void Main()
    {
        int PizzaSize;

        Console.WriteLine("Select your pizza size");
        Console.WriteLine("1=Small | 2=Medium");
        Console.WriteLine("3=Large | 4=Jumbo");
        Console.Write("Your Choice: ");
        string PSize = Console.ReadLine();
        PizzaSize = int.Parse(PSize);

        if(PizzaSize >= 0 && PizzaSize <= 4)
            Console.WriteLine("Good Choice. Now we will proceed with the
toppings\n");
        else
            Console.WriteLine("Invalid Choice\n");
    }
}
```

Here is an example of running the program:

Coalesce

```
Select your pizza size
1=Small | 2=Medium
3=Large | 4=Jumbo
Your Choice: 2
Good Choice. Now we will proceed with the toppings

Press any key to continue
```

When a program asks a question to the user who must answer by typing a letter, there is a chance that the user would type the answer in uppercase or lowercase. Since we know that C++ is case-sensitive, you can use a combined conditional statement to find out what answer or letter the user would have typed.

We saw that the truthfulness of a statement depends on how the statement is structured. In some and various cases, instead of checking that a statement is true, you can validate only negative values. This can be done on single or combined statements. For example, if a program is asking a question that requires a Yes or No answer, you can make sure the program gets a valid answer before continuing. Once again, you can use a logical conjunction to test the valid answers. Here is an example:

```
using System;

class NewProject
{
    static void Main()
    {
        char SittingDown;
        string SitDown;

        do
        {
            Console.WriteLine("Are you sitting down now(y/n)? ");
            SitDown = Console.ReadLine();
            SittingDown = char.Parse(SitDown);

            if( SittingDown != 'y' && SittingDown != 'Y' )
                Console.WriteLine("Could you please sit down for the
next exercise?");
        } while( SittingDown != 'y' && SittingDown != 'Y');

        Console.WriteLine("Wonderful!!!\n");
    }
}
```

Here is an example from running the program:

```
Are you sitting down now(y/n)? w
Could you please sit down for the next exercise?
Are you sitting down now(y/n)? h
```


Coalesce

Could you please sit down for the next exercise?

Are you sitting down now(y/n)? p

Could you please sit down for the next exercise?

Are you sitting down now(y/n)? y

Wonderful!!!

Press any key to continue

Logical Disjunction: OR

Let's assume that a driver has broken the law by driving through a red traffic light but there was no accident (to make our discussion simpler). There are two ways he can get a ticket: a police officer saw him, a special camera took a picture. This time again, we have three statements to make:

$S \Leftrightarrow$ A police officer saw the driver

$H \Leftrightarrow$ A camera took a picture of the action

$T \Leftrightarrow$ The driver got a ticket

If a police officer saw the driver breaking the law, the "A police officer saw the driver" statement is true. Consequently, S is true.

If a (specially installed) camera took the picture (of the scene), the "A camera took the picture of the action" statement is true. This means H is true.

If the driver gets a ticket, the "The driver gets a ticket" statement is true, which means T is true.

S
True

H
true

T
true

Once again, the third statement has no bearing unless you consider the first two. Last time, we saw that if the first two statements were combined, only then the result would produce the third statement. Let's consider in which case the driver would get a ticket.

If a police officer saw the driver, would he get a ticket? Yes, because on many traffic lights there is no camera but a police officer has authority to hand an infraction. This means if S is true, then T also is true. This produces:

S	H	T
true	Don't Care	true

Imagine a traffic light is equipped with a camera. If the driver breaks the law, the camera would take a picture, which means the driver would get a ticket. Therefore, if a camera takes a picture (H is true), the driver gets a ticket (T is true):

S	H	T
Don't Care	true	true

What if a police officer catches the action and a camera takes a picture. This means the driver will still get a ticket, even if one of both the police officer and the camera does not act but the other does. If both the police officer and the camera catch the action and act accordingly, the driver would get only one ticket (even if the driver receives two tickets, only one would be considered).

Coalesce

Therefore, whether the first statement OR the second statement is true, the resulting third statement T is still true:

S	H	T
true	true	true

The only time the driver would not get a ticket is when no police officer catches him and no camera takes a picture. In other words, only when both of the first two statements are false can the third statement be false.

Since T is the result of S and H combined, we have seen that T is true whenever either S is true OR H is true. The OR logical disjunction is expressed in C++ with the `||` operator. Here is the resulting table:

S	H	S H	T
true	true	true	TRUE
false	true	true	TRUE
true	false	true	TRUE
false	false	false	FALSE

Consider a program that asks a question and expects a yes or no answer in the form of y or n. Besides y for yes, you can also allow the user to type Y as a valid yes. To do this, you would let the compiler check that either y or Y was typed. In the same way, either n or N would be valid negations. Any of the other characters would fall outside the valid characters. Our hot-tempered program can be restructured as follows:

```
using System;
```

```
class NewProject
{
    static void Main()
    {
        char Answer;

        Console.WriteLine("Do you consider yourself a hot-tempered individual(y=Yes/n=No)? ");
        string Ans = Console.ReadLine();
        Answer = char.Parse(Ans);

        if( Answer == 'y' || Answer == 'Y' ) // Unique Condition
        {
            Console.WriteLine("\nThis job involves a high level of self-control.");
            Console.WriteLine("\nWe will get back to you.\n");
        }
        else if( Answer == 'n' || Answer == 'N' ) // Alternative
            Console.WriteLine("\nYou are hired!\n");
        else
            Console.WriteLine("\nThat was not a valid answer!\n");
    }
}
```

Coalesce

Chapter 5

Introduction to Classes

A Re-Introduction to Classes

Overview

So far, we have been using classes in a very introductory manner, based on brief definition we had in the first lesson. This was necessary because everything in C# is built around a concept of class. A class can be more elaborate than we have come accustomed. A class is a technique of creating one or a group of variables to be used as a foundation for a more elaborate variable. A class must be created in a computer file. You can create such a file with only one class or you can create many classes inside on one file.

Class Creation

To create a class, you start with the **class** keyword followed by a name and its body delimited by curly brackets. Here is an example of a class called Book:

```
class Book
{
}
```

The section between the curly brackets of a class is referred to as its body. In the body of a class, you can create a list of the parts that make up the object. Each of these parts must be a complete variable with a name and a data type. For example, here are the characteristics that make up a book, declared as the parts of the above Book class:

```
class Book
{
    string Title;
    string Author;
    short YearPublished;
    int NumberOfPages;
    char CoverType;
}
```

The variables declared in the body of a class are referred to as its member variables. The member variables can be any type we have seen in the previous lesson. Based on this, when creating a class, it is your job to decide what your object is made of.

Coalesce

Practical Learning: Introducing Class Members

Imagine you want to write a (console-based) program for a department store and the customer has given you a preliminary catalog as follows:

Stock #: 437876 Women Spring Coat Classy Unit: \$145.55	Stock #: 79475 Women Bathing Suit Sandstone \$75.55	Stock #: 74797 Women Suit Exchange \$225.75
Stock: 68432 Men Jacket \$115.85	Stock #: 75947 Children Summer Hat \$17.95	Stock #: 48746 Children Playground Dress \$24.55

Each item in this catalog is represented by its Stock number, its name or description, and its price. Based on this, you can create a class that represents each item.

1. Start Notepad
2. In the empty file, type the following:

```
using System;

class StoreItem
{
    long ItemNumber;
    string ItemName;
    double UnitPrice;
}
```

3. Save the file in a new folder named **DeptStore1**
4. Save the file itself as **item.cs**
5. Without closing Notepad, open another instance of Notepad (start "another" Notepad) and type the following in it:

```
using System;

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

6. Save the new file as **exercise.cs** in the same DeptStore1 folder

Class Variable Declaration

After creating a class and its members, like any normal variable, to use a class in your program, you must first declare it where you would need it. In C#, as well as Visual Basic, if you create a class in

Coalesce

any of the files that belong to the same project, the class is made available to all other parts of the same project.

Like the variables we introduced in the previous lesson, to declare a variable of a class, type its name followed by a name for the variable. For example, to declare a variable of the above Book in another class called NewProject, you could type the following:

```
using System;

class Book
{
}

class NewProject
{
    static void Main()
    {
        Book ToRead;
    }
}
```

In C++ and Pascal, if you create a class CA in a certain file FA and want to access that class in another file FB, you must type **#include** or **uses** followed by the name of the file FA somewhere in the top section of the file FB.

The variables we have declared so far are called value variable. This is because the variable holds its value. The C# language supports another type of variable. This time, when you declare the variable, its names doesn't hold the value of the variable, it holds a reference to the address where the actual variable is stored. This reference type is the kind of variable used to declare a variable for a class.

To use a variable as reference, you must initialize it using an operator called **new** (later on, we will learn that the **new** operator is in fact used to access a constructor of the class). Here is an example:

```
using System;

class Book
{
}

class NewProject
{
    static void Main()
    {
        Book ToRead;

        ToRead = new Book();
    }
}
```

Coalesce

You can also use the **new** operator directly when declaring the variable as follows:

```
Book ToRead = new Book();
```

Practical Learning: Declaring a Class Variable

1. To use the StoreItem Car class in the main class of the project, access the exercise.cs file declare the variable as follows:

```
using System;

class DeptStore
{
    Static void Main()
    {
        DeptStore dptStore = new DeptStore();
    }
}
```

2. Save the exercise.cs file

Access to Class Members

The characteristics of an object usually fall into two categories: those you can touch and those you don't have access to. For example, for a car parked at the mall, you can see or touch its doors and its tires but you don't see its engine or the spare tire in its trunk. The parts of an object that you have access to are referred to as public. Those you can't see or touch are referred to as private.

A C# class also recognizes that some parts of a class can be made available to other classes and some other parts can be hidden from other classes. A part that must be hidden from other classes is **private** and it can be declared starting with the **private** keyword. If you declare a member variable and want to make it available to other classes, you must start its name with the **public** keyword. The **public** and **private** keywords are referred to as access level.

By default, if you declare a member variable (or anything else) in a class but don't specify its access level, the member is considered private and cannot be accessed from outside, that is by a non-member, of that class. Therefore, to make a member accessible by other classes, you must declare it as public.

You can use a mix of public and private members in a class and there is no rule on which access level should be listed first or last. Here is an example:

```
class Book
{
    public string Title;
    public string Author";
    short YearPublished;
    private int NumberOfPages;
    char CoverType;
}
```

Coalesce

Just keep in mind that if you omit or forget the access level of a member of a class, the member is automatically made private.

After declaring a member of a class, to access it from another class, first declare a variable from its class as we saw earlier. To actually access the method, use the period operator as follows:

```
using System;

class Book
{
    public string Title;
    public string Author;
    short YearPublished;
    private int NumberOfPages;
    char CoverType;
}

class BookClub
{
    static void Main()
    {
        Book ToRead = new Book();
        ToRead.Author = "Francis Patrick Kouma";

        Console.WriteLine(ToRead.Author);
    }
}
```

To reduce confusion as to what member is public or private, we will always specify the access level of a member variable.

Practical Learning: Using a Class' Member Variables

1. Access the item.cs file and make public all members of the StoreItem class:

```
using System;

class StoreItem
{
    public long ItemNumber;
    public string ItemName;
    public double UnitPrice;
}
```

2. Save the item.cs file
3. Access the exercise.cs file
To access the members of the StoreItem class, change the exercise.cs file as follows:

```
using System;
```

Coalesce

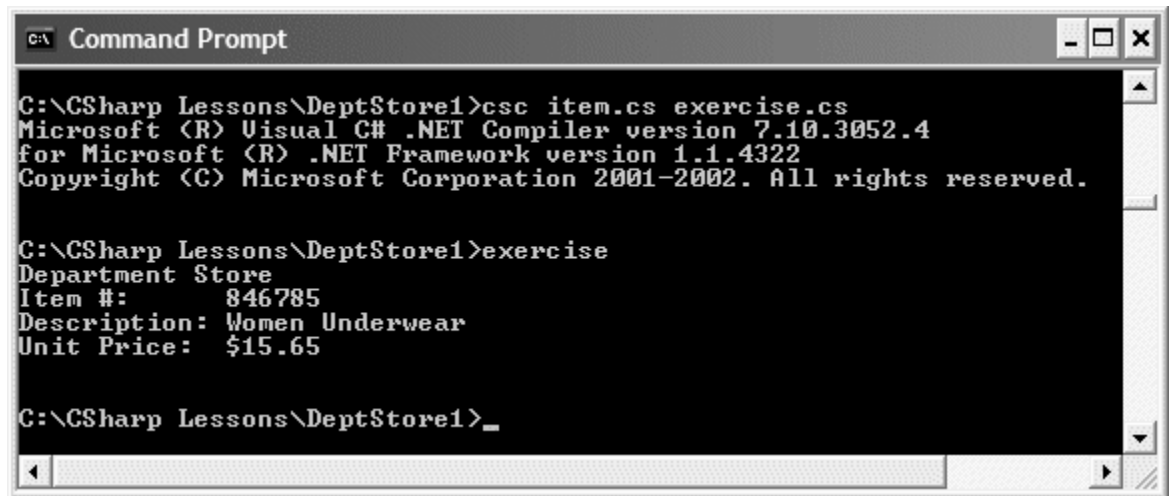
```
class DeptStore
{
    static void Main()
    {
        StoreItem dptStore = new StoreItem();

        dptStore.ItemNumber = 846785;
        dptStore.ItemName = "Women Underwear";
        dptStore.UnitPrice = 15.65;

        Console.WriteLine("Department Store");
        Console.WriteLine("Item #: {0}", dptStore.ItemNumber);
        Console.WriteLine("Description: {0}", dptStore.ItemName);
        Console.WriteLine("Unit Price: {0:C}", dptStore.UnitPrice);

        Console.WriteLine();
    }
}
```

4. Save the exercise.cs file
5. Open the Command Prompt and switch to the DeptStore1 folder that contains the current exercise
6. To compile the project, type **csc item.cs exercise.cs**
7. To execute the application, type **exercise**
8. and test it. This would produce:



```
C:\> Command Prompt

C:\CSharp Lessons\DeptStore1>csc item.cs exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

C:\CSharp Lessons\DeptStore1>exercise
Department Store
Item #:      846785
Description: Women Underwear
Unit Price:  $15.65

C:\CSharp Lessons\DeptStore1>_
```

Static Member Variables

Imagine you can a class called Book. To access it in the Main() function, you can declare its variable, as we have done so far. A variable you have declared of a class is also called an instance of the class. In the same way, you can declare various instances of the same class as necessary:

Coalesce

```
using System;

class Book
{
    public string Title;
    public string Author;
    public short YearPublished;
    public int NumberOfPages;
    public char CoverType;
}

class NewProject
{
    static void Main()
    {
        Book written = new Book();
        Book bought = new Book();
    }
}
```

Each one of these instances gives you access to the members of the class but each instance holds the particular values of the members of its instance. Consider the results of the following program:

```
using System;

class Book
{
    public string Title;
    public string Author;
    public short YearPublished;
    public int NumberOfPages;
    public char CoverType;
}

class Program
{
    static void Main(string[] args)
    {
        Book First = new Book();

        First.Title = "Psychology and Human Evolution";
        First.Author = "Jeannot Lamm";
        First.YearPublished = 1996;
        First.NumberOfPages = 872;
        First.CoverType = 'H';

        Console.WriteLine("Book Characteristics");
        Console.WriteLine("Title: {0}", First.Title);
        Console.WriteLine("Author: {0}", First.Author);
        Console.WriteLine("Year: {0}", First.YearPublished);
        Console.WriteLine("Pages: {0}", First.NumberOfPages);
        Console.WriteLine("Cover: {0}\n", First.CoverType);
    }
}
```

Coalesce

```
        Book Second = new Book();

        Second.Title = "C# First Step";
        Second.Author = "Alexandra Nyango";
        Second.YearPublished = 2004;
        Second.NumberOfPages = 604;
        Second.CoverType = 'P';

        Console.WriteLine("Book Characteristics");
        Console.WriteLine("Title: {0}", Second.Title);
        Console.WriteLine("Author: {0}", Second.Author);
        Console.WriteLine("Year: {0}", Second.YearPublished);
        Console.WriteLine("Pages: {0}", Second.NumberOfPages);
        Console.WriteLine("Cover: {0}\n", Second.CoverType);
    }
}
```

This would produce:

```
Book Characteristics
Title: Psychology and Human Evolution
Author: Jeannot Lamm
Year: 1996
Pages: 872
Cover: H
```

```
Book Characteristics
Title: C# First Step
Author: Alexandra Nyango
Year: 2004
Pages: 604
Cover: P
```

All of the member variables and methods of classes we have used so far are referred to as *instance members* because, in order to access them, you must have an instance of a class declared in another class in which you want to access them.

C# allows you to declare a class member and refer to it regardless of which instance of an object you are using. Such a member variable is called static. To declare a member variable of a class as static, type the **static** keyword on its left. Whenever you have a static member, in order to refer to it, you must "qualify" it in the class in which you want to call it. Qualifying a member means you must give its complete location, including the name of its class and the namespace (if any) in which its class was created. Here is an example:

```
using System;

class Book
{
    public static string Title;
    public static string Author;
    public short YearPublished;
```

Coalesce

```
    public int   NumberOfPages;
    public char  CoverType;
}

class Program
{
    static void Main(string[] args)
    {
        Book First = new Book();

        Book.Title = "Psychology and Human Evolution";
        Book.Author = "Jeannot Lamm";
        First.YearPublished = 1996;
        First.NumberOfPages = 872;
        First.CoverType = 'H';

        Console.WriteLine("Book Characteristics");
        Console.WriteLine("Title: {0}", Book.Title);
        Console.WriteLine("Author: {0}", Book.Author);
        Console.WriteLine("Year: {0}", First.YearPublished);
        Console.WriteLine("Pages: {0}", First.NumberOfPages);
        Console.WriteLine("Cover: {0}\n", First.CoverType);

        Book Second = new Book();

        Book.Title = "C# First Step";
        Book.Author = "Alexandra Nyango";
        Second.YearPublished = 2004;
        Second.NumberOfPages = 604;
        Second.CoverType = 'P';

        Console.WriteLine("Book Characteristics");
        Console.WriteLine("Title: {0}", Book.Title);
        Console.WriteLine("Author: {0}", Book.Author);
        Console.WriteLine("Year: {0}", Second.YearPublished);
        Console.WriteLine("Pages: {0}", Second.NumberOfPages);
        Console.WriteLine("Cover: {0}\n", Second.CoverType);

        Console.ReadLine();
    }
}
```

Notice that when a member variable has been declared as static, you don't need an instance of the class to access that member variable outside of the class. Based on this, if you declare all members of a class as static, you don't need to declare a variable of their class in order to access them:

```
using System;

class Book
{
    public static string Title;
    public static string Author;
```

Coalesce

```
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        Book.Title = "Psychology and Human Evolution";  
        Book.Author = "Jeannot Lamm";  
  
        Console.WriteLine("Book Characteristics");  
        Console.WriteLine("Title: {0}", Book.Title);  
        Console.WriteLine("Author: {0}\n", Book.Author);  
  
        Book.Title = "C# First Step";  
        Book.Author = "Alexandra Nyango";  
  
        Console.WriteLine("Book Characteristics");  
        Console.WriteLine("Title: {0}", Book.Title);  
        Console.WriteLine("Author: {0}\n", Book.Author);  
  
        Console.ReadLine();  
    }  
}
```

If the member variable is declared with an access level, you can position the **static** keyword before or after the access level keyword:

```
class Book  
{  
    public static string Title;  
    static public string Author;  
}
```

If you are referring to a static member variable in the same class in which it was declared, you don't have to qualify it. Here is an example:

```
using System;  
  
class NewProject  
{  
    static double Length;  
    static double Width;  
  
    static void Main()  
    {  
        Console.WriteLine("Rectangles Characteristics");  
  
        Length = 22.55;  
        Width = 20.25;  
  
        Console.WriteLine("\nRectangle 1");  
        Console.Write("Length: ");  
        Console.WriteLine(Length);  
    }  
}
```

Coalesce

```
        Console.Write("Width: ");  
        Console.WriteLine(Width);  
  
        Length = 254.04;  
        Width = 408.62;  
  
        Console.WriteLine("\nRectangle 2");  
        Console.Write("Length: ");  
        Console.WriteLine(Length);  
        Console.Write("Width: ");  
        Console.WriteLine(Width);  
  
        Console.WriteLine();  
    }  
}
```

Coalesce

Chapter 6

Class Methods

Introduction to Methods

Definition

Besides its characteristics, a class can also perform assignments. For example, a book as an object can display its pages; that's one of its assignments. A car as an object can move forward; that also is an assignment. An assignment performed by a class is called a function. When a function belongs to a class (in C# this is not an issue since everything that is not a class must belong to a class), the function is called a member function or a method. On this site, the word method will always refer to a function that is a member of a class.

A method must have a name. We will follow the same **rules and conventions** we defined for all names. An example of a method would be *DisplayInformation*. To distinguish a method from a variable, it is always followed by parentheses, sometimes empty.

Since a method is used to perform an assignment, its job is included between an opening curly bracket "{" and a closing curly bracket "}". Here is an example:

```
class Book
{
    DisplayInformation() { }
}
```

If the assignment performed by a method is long, you can use many lines to define its behavior. For this reason, each bracket can be typed on its own line.

After a method has performed its assignment, it may provide a result. The result of a method is called a return value. Each return value follows a particular type based on one of the data types we reviewed in the previous lesson. Any time you create a function, the compiler needs to know the type of value that the method would return. The type of value that a method would return is specified on the left side of the method's name.

Some, even many, of the methods used in your programs will not return a value after they have performed an assignment. When a method doesn't return a value, it is considered as void. The return type of such a method is **void**. Here is an example:

```
class Book
{
    void DisplayInformation()
    {
    }
}
```

If you create a method and want to make it available to other classes, you must start its name with the **public** keyword, otherwise, as mentioned for the member variables, the method would be considered private and inaccessible to other classes. Of course, it is your responsibility to decide whether a member of a class must be public or private.

Coalesce

Here is an example:

```
class Book
{
    public void DisplayInformation()
    {
    }
}
```

Like member variables, a class can have as many methods as you judge necessary and each method can have the access level of your choice. Remember that if you omit or forget to specify the access level, the method is considered private. Therefore, to reduce confusion as to what method is public or private, we will always specify the access level of each method of a class.

After creating a method, to access it from another class, first declare a variable from its class as we saw earlier. To actually access the method, use the period operator as follows:

```
using System;

class Book
{
    public void DisplayInformation()
    {
    }
}

class NewProject
{
    static void Main()
    {
        Book ToRead = new Book();
        ToRead.DisplayInformation();
    }
}
```

Practical Learning: Creating Methods

1. Start Notepad
2. In the empty file, type the following:

```
using System;

class StoreItem
{
    long ItemNumber;
    string ItemName;
    double UnitPrice;
}
```

3. Save the file in a new folder named **DeptStore2**

Coalesce

4. Save the file itself as **item.cs**
5. Without closing Notepad, open another instance of Notepad (start "another" Notepad) and type the following in it:

```
using System;

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

6. Save the new file as **exercise.cs** in the same DeptStore1 folder
7. To create methods for a class, open the item.cs file change the StoreItem class as follows:

```
using System;

public class StoreItem
{
    public long  ItemNumber;
    public string ItemName;
    public double UnitPrice;

    public void RegisterCar()
    {
    }
}
```

8. Save the file

The Method's Body

As mentioned above, the behavior of a method is stated (or defined) between its delimiting curly brackets. The section inside the curly brackets is called the body of a method. In the body of the method, you can simply display a sentence. You can use a method to initialize an object, that is, to provide default values for its member variables. You can also use a method to display the characteristics of an object. Here is an example:

```
using System;

class Book
{
    private string Title = "Basic Business Mathematics";
    private string Author = "Eugene Don - Joel Lerner";
    private short  YearPublished = 2000;
    private int    NumberOfPages = 249;
    private char   CoverType = 'P';
    public void DisplayInformation()
```


Coalesce

```
        {
            Console.WriteLine("Book Information");
            Console.Write("Title: ");
            Console.WriteLine(Title);
            Console.Write("Author: ");
            Console.WriteLine(Author);
            Console.Write("Year: ");
            Console.WriteLine(YearPublished);
            Console.Write("Pages: ");
            Console.WriteLine(NumberOfPages);
            Console.Write("Cover: ");
            Console.WriteLine(CoverType);
        }
    }

class BookClub
{
    static void Main()
    {
        Book ToRead = new Book();

        ToRead.DisplayInformation();
        Console.WriteLine();
    }
}
```

Practical Learning: Declaring a Class Variable

1. To use a method of a class, change the new method as follows:

```
using System;

public class StoreItem
{
    public long ItemNumber;
    public string ItemName;
    public double UnitPrice;

    public void RegisterItem()
    {
        ItemNumber = 0;
        ItemName = "Unknown";
        UnitPrice = 0.00;
    }
}
```

2. Open the exercise.cs file and change the Main() function as follows:

```
using System;
```

Coalesce

```
class DeptStore
{
    static void Main()
    {
        StoreItem dptStore = new StoreItem();

        dptStore.RegisterItem();

        Console.WriteLine("Department Store");
        Console.WriteLine("Item #: {0}", dptStore.ItemNumber);
        Console.WriteLine("Description: {0}", dptStore.ItemName);
        Console.WriteLine("Unit Price: {0:C}", dptStore.UnitPrice);

        Console.WriteLine();
    }
}
```

3. In the Command Prompt, compile the application with **csc item.cs exercise.cs**
4. Execute it by typing **exercise**

```
Department Store
Item #: 0
Description: Unknown
Unit Price: $0.00
```

The Constructor of a Class

When you declare a variable for a class, a special method must be called to initialize the members of that class. This method is automatically provided for every class and it is called a constructor. Whenever you can create a new class, a constructor is automatically created for it. Although a constructor is created for your class, you can customize its behavior or change it tremendously.

A constructor is a method that servers as a primary helper of a class. It holds the same name as the class and doesn't return any value, not even **void**. Here is an example:

```
class Book
{
    Book()
}
```

Like every method, a constructor can be equipped with a body. In this body, you can access any of the member variables (or method(s)) of the same class. Consider the following program:

```
using System;

class Exercise
{
    public void Welcome()
    {
        Console.WriteLine("The wonderful world of C# programming");
    }
}
```

Coalesce

```
public Exercise()
{
    Console.WriteLine("The Exercise class is now available");
}

class Class1
{
    static void Main()
    {
        Exercise exo = new Exercise();
    }
}
```

When executed, it would produce:

The Exercise class is now available

This shows that, when a class has been instantiated, its constructor is the first method to be called. For this reason, you can use a constructor to initialize a class, that is, to assign default values to its member variables.

1. Access the item.cs file. To use a constructor, change the existing method as follows:

```
using System;

public class StoreItem
{
    public long  ItemNumber;
    public string ItemName;
    public double UnitPrice;

    public StoreItem()
    {
        ItemNumber = 0;
        ItemName  = "Unknown";
        UnitPrice  = 0.00;
    }
}
```

2. Save the file and open the exercise.cs file
3. Change it as follows:

```
using System;

class DeptStore
{
    static void Main()
    {
        StoreItem dptStore = new StoreItem();
    }
}
```

Coalesce

```
        Console.WriteLine("Department Store");
        Console.WriteLine("Item #: {0}", dptStore.ItemNumber);
        Console.WriteLine("Description: {0}", dptStore.ItemName);
        Console.WriteLine("Unit Price: {0:C}", dptStore.UnitPrice);

        Console.WriteLine();
    }
}
```

4. Save the exercise.cs file then compile the program at the Command Prompt with **csc item.cs exercise.cs** and execute it with **exercise**

Local Variables

In the body of a method, you can also declare variables that would be used internally by the member function. A variable declared in the body is referred to as a local variable. It cannot be accessed outside of the method as it "belongs" to the method.

After declaring a local variable, it is made available to the method and you can use it as you see fit, for example, you can assign it a value prior to using it.

Methods and their Return Type

The void Return

After a method has carried its assignment, it may or may not return a value. Fortunately, when a method is created, it has direct access to all member variables of a class. This means that you can use a method to perform any type of operation of calculation that involves the members of a class without re-declaring them. This also means that such a method doesn't need to return a particular value.

As used so far, when a method doesn't return a value, it must be declared as **void**. Here is an example:

using System;

```
namespace ConsoleApplication1
{
    class Book
    {
        public static string Title;
        static public string Author;

        public void ShowCharacteristics()
        {
            Console.WriteLine("Book Characteristics");
            Console.WriteLine("Title: {0}", Title);
            Console.WriteLine("Author: {0}", Author);
        }
    }
}
class Program
{
```

Coalesce

```
static void Main()
{
    Book First = new Book();

    Book.Title = "Psychology and Human Evolution";
    Book.Author = "Jeannot Lamm";

    First.ShowCharacteristics();

    Book.Title = "C# First Step";
    Book.Author = "Alexandra Nyango";

    First.ShowCharacteristics();
}
}
```

This would produce:

```
Book Characteristics
Title: Psychology and Human Evolution
Author: Jeannot Lamm
Book Characteristics
Title: C# First Step
Author: Alexandra Nyango
```

Primitive Return Types

The methods we have used so far did not return any value necessary for other methods or classes. If a method has carried an assignment and must make its result available to other methods or other classes, the method must return a value and cannot be **void**. To declare a method that returns a value, provide its return type to its left. If you are also specifying the access level, provide the return type between the access level and the method name. Here is an example:

```
using System;

class Operations
{
    public double Addition()
    {
    }
}
```

After a method has performed its assignment, it must clearly demonstrate that it is returning a value. To do this, you use the **return** keyword followed by the value that the method is returning. The value returned must be the same type specified as the return type. The Area method could be implemented as follows:

```
using System;

class Operations
```

Coalesce

```
{
    public double Addition()
    {
        double a, b, c;

        a = 128.76;
        b = 5044.52;
        c = a + b;

        return c;
    }
}
```

When a method returns a value, the compiler considers such a method as if it were a regular value. This means that you can use the **Console.Write()** of the **Console.WriteLine()** methods to display its value. To do this, simply type the name of the method and its parentheses in the **Console.Write()** of the **Console.WriteLine()** methods' parentheses. Here is an example:

```
using System;

class Operations
{
    public double Addition()
    {
        double a, b, c;

        a = 128.76;
        b = 5044.52;
        c = a + b;

        return c;
    }
}

class Methods101
{
    static void Main()
    {
        Operations Oper = new Operations();
        Console.Write("Result: ");
        Console.WriteLine(Oper.Addition());
    }
}
```

In the same way, a method returns a value can be assigned to a variable of the same type.

Static Methods

Like a member variable, a member function, called a method, of a class can be define as static. This means that this particular method can access any member of the class regardless of the instance if there are many instances of the class declared.

Coalesce

To define a method as static, type the **static** keyword to its left. This time, the **static** keyword must be the first on the line. Here is an example:

```
using System;

class NewProject
{
    static double Length;
    static double Width;

    static double Perimeter()
    {
        return 2 * (Length + Width);
    }
    static double Area()
    {
        return Length * Width;
    }
    static void Main()
    {
        Console.WriteLine("Rectangles Characteristics");

        Length = 22.55;
        Width = 20.25;

        Console.WriteLine("\nRectangle 1");
        Console.Write("Length: ");
        Console.WriteLine(Length);
        Console.Write("Width: ");
        Console.WriteLine(Width);
        Console.Write("Perimeter: ");
        Console.WriteLine(Perimeter());
        Console.Write("Area: ");
        Console.WriteLine(Area());

        Length = 254.04;
        Width = 408.62;

        Console.WriteLine("\nRectangle 2");
        Console.Write("Length: ");
        Console.WriteLine(Length);
        Console.Write("Width: ");
        Console.WriteLine(Width);
        Console.Write("Perimeter: ");
        Console.WriteLine(Perimeter());
        Console.Write("Area: ");
        Console.WriteLine(Area());

        Console.WriteLine();
    }
}
```

Coalesce

This would produce:

Rectangles Characteristics

Rectangle 1

Length: 22.55

Width: 20.25

Perimeter: 85.6

Area: 456.6375

Rectangle 2

Length: 254.04

Width: 408.62

Perimeter: 1325.32

Area: 103805.8248

Practical Learning: Introducing C#

1. Start a new instance of Notepad and type the following in it:

```
using System;

class CarRental
{
    static private string Make;
    static private string Model;
    static private uint CarYear;
    static private byte NumberOfDoors;
    static private bool HasAirCondition;
    static private bool HasCDPplayer;
    static private ulong Mileage;

    // This method is used to initialize a Car object
    static private void RegisterCar()
    {
        Make = "Lincoln";
        Model = "Blackwood";
        NumberOfDoors = 4;
        CarYear = 2002;
        HasAirCondition = true;
        HasCDPplayer = false;
        Mileage = 24778;
    }

    // Used to display the characteristics of a Car variable
    static private void ShowCarCharacteristics()
    {
        // Display information about this car
        Console.WriteLine("Car Characteristics");
    }
}
```


Coalesce

```
        Console.Write("Car Name:   ");
        Console.Write(Make);
        Console.Write(" ");
        Console.WriteLine(Model);
        Console.Write("Year:      ");
        Console.WriteLine(CarYear);
        Console.Write("Nbr of Doors: ");
        Console.WriteLine(NumberOfDoors);
        Console.Write("Has Air Cond.? ");
        Console.WriteLine(HasAirCondition);
        Console.Write("Has CD Player? ");
        Console.WriteLine(HasCDPplayer);
        Console.Write("Mileage:    ");
        Console.Write(Mileage);
        Console.WriteLine(" miles");
        Console.WriteLine("\n");
    }

    static void Main()
    {
        RegisterCar();
        ShowCarCharacteristics();
    }
}
```

2. Save the file in a new folder named Methods1
3. Save the file as exercise.cs
4. Compile it at the Command Prompt with `csc exercise.cs`
5. Execute it with `exercise`

Conditional Statements and Class Methods

Using Conditions in Functions

The use of functions in a program allows you to isolate assignments and confine them to appropriate entities. While the functions take care of specific requests, you should provide them with conditional statements to validate what these functions are supposed to do. There are no set rules to the techniques involved; everything depends on the tasks at hand. Once again, you will have to choose the right tools for the right job. To make effective use of functions, you should be very familiar with different data types because you will need to return the right value.

The ergonomic program we have been writing so far needs to check different things including answers from the user in order to proceed. These various assignments can be given to functions that would simply hand the results to the `main()` function that can, in turn, send these results to other functions for further processing. Here is an example:

```
using System;

class NewProject
{
    static char GetPosition()
```

Coalesce

```
{
    char Position;
    string Pos;

    do {
        Console.WriteLine("Are you sitting down now(y/n)? ");
        Pos = Console.ReadLine();
        Position = char.Parse(Pos);

        if( Position != 'y' && Position != 'Y' &&
            Position != 'n' && Position != 'N' )
            Console.WriteLine("Invalid Answer\n");
    } while( Position != 'y' && Position != 'Y' &&
            Position != 'n' && Position != 'N' );

    return Position;
}

static void Main()
{
    char Position;

    Position = GetPosition();

    if( Position == 'n' || Position == 'N' )
        Console.WriteLine("\nCould you please sit down for the next
exercise?\n");
    else
        Console.WriteLine("\nWonderfull!!!\n\n");
}
}
```

Here is an example of running the program:

```
Are you sitting down now(y/n)? V
Invalid Answer
```

```
Are you sitting down now(y/n)? z
Invalid Answer
```

```
Are you sitting down now(y/n)? n
```

```
Could you please sit down for the next exercise?
```

```
Press any key to continue
```

Functions do not have to return a value in order to be involved with conditional statements. In fact, both issues are fairly independent. This means, void and non-void functions can manipulate values based on conditions internal to the functions. This is illustrated in the following program that is an enhancement to an earlier ergonomic program:

Coalesce

```
using System;

class NewProject
{
    private static char GetPosition()
    {
        char Position;
        string Pos;

        do {
            Console.WriteLine("Are you sitting down now(y/n)? ");
            Pos = Console.ReadLine();
            Position = char.Parse(Pos);

            if( Position != 'y' && Position != 'Y' &&
                Position != 'n' && Position != 'N' )
                Console.WriteLine("Invalid Answer\n");
        } while(Position != 'y' && Position != 'Y' &&
            Position != 'n' && Position != 'N');

        return Position;
    }

    private static void NextExercise()
    {
        char LayOnBack;

        Console.WriteLine("Good. For the next exercise, you should lay on your
back");

        Console.WriteLine("\nAre you laying on your back(1=Yes/0=No)? ");
        string Lay = Console.ReadLine();
        LayOnBack = char.Parse(Lay);

        if(LayOnBack == '0')
        {
            char Ready;
            string Rd;

            do
            {
                Console.WriteLine("Please lay on your back");
                Console.WriteLine("Are you ready(1=Yes/0=No)? ");
                Rd = Console.ReadLine();
            } while(Ready == '0');

            Ready = char.Parse(Rd);
        }
        else if(LayOnBack == '1')
            Console.WriteLine("\nGreat.\nNow we will start the next
exercise.\n");
        else
            Console.WriteLine("\nWell, it looks like you are getting tired...");
    }
}
```

Coalesce

```
static void Main()
{
    char Position, WantToContinue;

    Position = GetPosition();

    if( Position == 'n' || Position == 'N' )
        Console.WriteLine("\nCould you please sit down for the next
exercise?\n");
    else
    {
        Console.WriteLine("\nWonderful!\nNow we will continue today's
exercise...\n");
        Console.WriteLine("\n...\n\nEnd of exercise\n");
    }

    Console.Write("\nDo you want to continue(1=Yes/0=No)? ");
    string ToContinue = Console.ReadLine();
    WantToContinue = char.Parse(ToContinue);

    if( WantToContinue == '1' )
        NextExercise();
    else if( WantToContinue == '0' )
        Console.WriteLine("\nWell, it looks like you are getting tired...\n");
    else
    {
        Console.WriteLine("\nConsidering your invalid answer...");
        Console.WriteLine("\nWe had enough today\n");
    }
    Console.WriteLine("We will stop the session now\nThanks.\n");
}
}
```

Here is an example of running the program:

Are you sitting down now(y/n)? a
Invalid Answer

Are you sitting down now(y/n)? p
Invalid Answer

Are you sitting down now(y/n)? y

Wonderful!
Now we will continue today's exercise...

...

End of exercise

Coalesce

Do you want to continue(1=Yes/0=No)? 6

Considering your invalid answer...

We had enough today

We will stop the session now

Thanks.

Press any key to continue

Conditional Returns

A function defined other than void must always return a value. Sometimes, a function will perform some tasks whose results would lead to different consequences. A function can return only one value (this is true for this context, but we know that there are ways to pass arguments so that a function can return more than one value) but you can make it render a result depending on a particular behavior. Image that a function is requesting an answer from the user. Since the user can provide different answers, you can treat each result differently.

In the previous section, we saw an example of returning a value from a function. Following our employment application, here is an example of a program that performs a conditional return:

```
using System;

class NewProject
{
    private static bool GetAnswer()
    {
        char Answer;
        string Response;

        Console.WriteLine("Do you consider yourself a hot-tempered
individual(y=Yes/n=No)? ");
        string Ans = Console.ReadLine();
        Answer = char.Parse(Ans);

        if( Answer == 'y' )
            return true;
        else
            return false;
    }

    static void Main()
    {
        bool Answer;

        Answer = GetAnswer();

        if( Answer == true )
        {
```

Coalesce

```
        Console.WriteLine("\nThis job involves a high level of self-control.");
        Console.WriteLine("\nWe will get back to you.\n");
    }
    else
        Console.WriteLine("\nYou are hired!\n");
}
}
```

Imagine you write the following function:

```
using System;

class NewProject
{
    private static string GetPosition()
    {
        char Position;

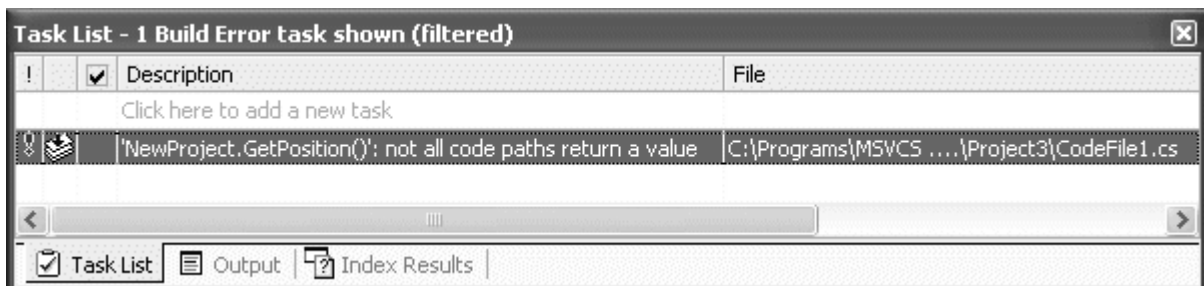
        Console.WriteLine("Are you sitting down now(y/n)? ");
        string Pos = Console.ReadLine();
        Position = char.Parse(Pos);

        if( Position == 'y' || Position == 'Y' )
            return "Yes";
        else if( Position == 'n' || Position == 'N' )
            return "No";
    }

    static void Main()
    {
        string Answer;

        Answer = GetPosition();
        Console.WriteLine("\nAnswer = ");
        Console.WriteLine(Answer);
    }
}
```

When running this program, you would receive the following error:



Coalesce

On paper, the function looks fine. If the user answers with y or Y, the function returns the string Yes. If the user answer with n or N, the function returns the string No. Unfortunately, this function has a problem: what if there is an answer that does not fit those we are expecting? In reality the values that we have returned in the function conform only to the conditional statements and not to the function. Remember that in `if(Condition)Statement`, the `Statement` executes only if the `Condition` is true. Here is what will happen. If the user answers y or Y, the function returns Yes and stops; fine, it has returned something, we are happy. If the user answers n or N, the function returns No, which also is a valid value: wonderful. If the user enters another value (other than y, Y, n, or N), the execution of the function will not execute any of the return statements and will not exit. This means the execution will reach the closing curly bracket without encountering a return value. Therefore, the compiler will issue a warning. Although the warning looks like not a big deal, you should take care of it: never neglect warnings. The solution is to provide a return value so that, if the execution reaches the end of the function, it would still return something. Here is a solution to the problem:

```
using System;

class NewProject
{
    private static string GetPosition()
    {
        char Position;

        Console.Write("Are you sitting down now(y/n)? ");
        string Pos = Console.ReadLine();
        Position = char.Parse(Pos);

        if( Position == 'y' || Position == 'Y' )
            return "Yes";
        else if( Position == 'n' || Position == 'N' )
            return "No";
        return "Invalid Answer";
    }

    static void Main()
    {
        string Answer;

        Answer = GetPosition();
        Console.Write("\nAnswer = ");
        Console.WriteLine(Answer);
    }
}
```

This is illustrated in the following program that has two functions with conditional returns:

```
using System;

class NewProject
{
    private static char GetPosition()
    {
```

Coalesce

```
char Position;
string Pos;

do
{
    Console.WriteLine("Are you sitting down(y/n)? ");
    Pos = Console.ReadLine();
    Position = char.Parse(Pos);

    if( Position != 'y' &&
        Position != 'Y' &&
        Position != 'n' &&
        Position != 'N' )
        Console.WriteLine("Invalid Answer\n");
} while(Position != 'y' &&
        Position != 'Y' &&
        Position != 'n' &&
        Position != 'N' );

if( Position == 'y' || Position == 'Y' )
    return 'y';
else if( Position == 'n' || Position == 'N' )
    return 'n';

// If you reach this point, none of the answers was valid
return Position;
}

private static void NextExercise()
{
    char LayOnBack;

    Console.WriteLine("Good. For the next exercise, you should lay on your back");
    Console.WriteLine("\nAre you laying on your back(1=Yes/0=No)? ");
    string Lay = Console.ReadLine();
    LayOnBack = char.Parse(Lay);

    if(LayOnBack == '0')
    {
        char Ready;
        string Rd;

        do
        {
            Console.WriteLine("Please lay on your back");
            Console.WriteLine("\nAre you ready(1=Yes/0=No)? ");
            Rd = Console.ReadLine();
            Ready = char.Parse(Rd);
        } while(Ready == '0');
    }
    else if(LayOnBack == '1')
        Console.WriteLine("\nGreat.\nNow we will start the next exercise.");
}
```


Coalesce

```
        else
            Console.WriteLine("\nWell, it looks like you are getting tired...");
    }

    private static bool ValidatePosition(char Pos)
    {
        if( Pos == 'y' || Pos == 'Y' )
            return true;
        else if( Pos == 'n' || Pos == 'N' )
            return false;

        // If you reached this point, we need to prevent a warning
        return false;
    }

    static void Main()
    {
        char Position, WantToContinue;
        bool SittingDown;

        Position = GetPosition();
        SittingDown = ValidatePosition(Position);

        if( SittingDown == false )
            Console.WriteLine("\nCould you please sit down for the next
exercise?");
        else
        {
            Console.WriteLine("\nWonderful!\nNow we will continue today's
exercise...\n");

            Console.WriteLine("\n...\n\nEnd of exercise\n");
        }

        Console.WriteLine("\nDo you want to continue(1=Yes/0=No)? ");
        string ToContinue = Console.ReadLine();
        WantToContinue = char.Parse(ToContinue);

        if( WantToContinue == '1' )
            NextExercise();
        else if( WantToContinue == '0' )
            Console.WriteLine("\nWell, it looks like you are getting tired...");
        else
        {
            Console.WriteLine("\nConsidering your invalid answer...");
            Console.WriteLine("\nWe had enough today");
        }

        Console.WriteLine("\nWe will stop the session now\nThanks.\n");
        Console.WriteLine();
    }
}
```

Coalesce

Here is an example of running the program:

Are you sitting down now(y/n)? q

Invalid Answer

Are you sitting down now(y/n)? b

Invalid Answer

Are you sitting down now(y/n)? k

Invalid Answer

Are you sitting down now(y/n)? n

Could you please sit down for the next exercise?

Do you want to continue(1=Yes/0=No)? 1

Good. For the next exercise, you should lay on your back

Are you laying on your back(1=Yes/0=No)? 0

Please lay on your back

Are you ready(1=Yes/0=No)? 0

Please lay on your back

Are you ready(1=Yes/0=No)? 1

We will stop the session now

Thanks.

Press any key to continue

Coalesce

Chapter 7

Methods and Their Parameters

Arguments

Overview of an Argument

As you may have realized already, when a member variable and a method are declared in a class, the method has access to that member variable and you don't have to declare a variable in a method to "represent" and have a copy of an existing member variable. This is a tremendous advantage of object oriented programming (OOP). Still, in some cases, a method may need to have its own variables in order to carry an assignment.

As we saw already, a function is a small assignment that a section of code is asked to carry so other parts of a program can reliably use its result. Sometimes, a function would need one or more values in order to carry its assignment. The particularity of such a value or such values is that the function or object that calls this function must supply the needed values. In other words, the member variables of a function cannot fulfill these needed. When a method needs a value to complete its assignment, such a value is called an argument.

Introduction to Arguments

Like a variable, an argument is represented by its type of value. For example, one function may need a character while another function would need a string. Yet another function may require a decimal number. This means that the function or class that calls a method is responsible for supplying the right value, even though a method may have an internal mechanism of checking the validity of such a value.

The value supplied to a method is typed in the parentheses of the method and its called an argument. In order to declare a method that takes an argument, you must specify its name and the argument between its parentheses. Because a method must specify the type of value it would need, the argument is represented by its data type and a name. If the method would not return a value, it can be declared as void.

Suppose you want to define a method that displays the side length of a square. Since you would have to supply the length, you can define such a method as follows:

```
using System;

class NewProject
{
    void DisplaySide(double Length)
    {
    }
    static void Main()
    {
    }
}
```

Coalesce

In the body of the method, you may or may not use the value of the argument. Otherwise, you can manipulate the supplied value as you see fit. In this example, you can display the value of the argument as follows:

```
using System;

class NewProject
{
    void DisplaySide(double Length)
    {
        Console.Write("Length: ");
        Console.WriteLine(Length);
    }
    static void Main()
    {
    }
}
```

In this case, remember to define the method as static if you plan to access it from Main().

In order to call a method that takes an argument, you must supply a value for the argument when calling the method; otherwise you would receive an error. Also, you should/must supply the right value; otherwise, the method may not work as expected and produce an unreliable result. Here is an example:

```
using System;

class NewProject
{
    static void DisplaySide(double Length)
    {
        Console.Write("Length: ");
        Console.WriteLine(Length);
    }
    static void Main()
    {
        DisplaySide(35.55);

        Console.WriteLine();
    }
}
```

As mentioned already, a method that takes an argument can also declared its own variable. Such variables are referred to as local.

Practical Learning: Introducing Arguments

1. Start Notepad and, in the empty file, type the following:

```
using System;

class Payroll
```

Coalesce

```
{
    static string GetName()
    {
        string FirstName, LastName, FN;

        Console.Write("Employee's First Name: ");
        FirstName = Console.ReadLine();
        Console.Write("Employee's Last Name: ");
        LastName = Console.ReadLine();

        FN = FirstName + " " + LastName;
        return FN;
    }

    static double GetHours(string FullName)
    {
        string Mon, Tue, Wed, Thu, Fri;
        double Monday, Tuesday, Wednesday, Thursday, Friday, TotalHours;

        Console.Write("Enter ");
        Console.Write(FullName);
        Console.WriteLine("'s Weekly Hours\n");
        Console.Write("Monday: ");
        Mon = Console.ReadLine();
        Console.Write("Tuesday: ");
        Tue = Console.ReadLine();
        Console.Write("Wednesday: ");
        Wed = Console.ReadLine();
        Console.Write("Thursday: ");
        Thu = Console.ReadLine();
        Console.Write("Friday: ");
        Fri = Console.ReadLine();

        Monday = double.Parse(Mon);
        Tuesday = double.Parse(Tue);
        Wednesday = double.Parse(Wed);
        Thursday = double.Parse(Thu);
        Friday = double.Parse(Fri);

        TotalHours = Monday + Tuesday + Wednesday + Thursday + Friday;
        return TotalHours;
    }

    static void Main()
    {
        string FullName;
        double Hours;

        FullName = GetName();
        Hours = GetHours(FullName);

        Console.WriteLine("\nEmployee's Name: ");
    }
}
```

Coalesce

```
        Console.WriteLine(FullName);
        Console.Write("Weekly Hours: ");
        Console.Write(Hours);
        Console.WriteLine(" hours\n");
    }
}
```

2. Save the file in a new folder named **Payroll1**
3. Save the file itself as **exercise.cs**
4. Open the Command Prompt and switch to the folder that contains the current project
5. Compile the program by typing **csc exercise.cs**
6. Execute it by typing **exercise**
Here is an example of executing the program:

```
Employee's First Name: Antoinette
Employee's Last Name: Malhoum
Enter Antoinette Malhoum's Weekly Hours
```

```
Monday: 8
Tuesday: 9.50
Wednesday: 8.50
Thursday: 8
Friday: 8.50
```

```
Employee's Name: Antoinette Malhoum
Weekly Hours: 42.5 hours
```

7. Return to Notepad
8. A method can take more than one argument. When defining such a method, provide each argument with its data type and a name. The arguments are separated by a comma.
As an example, change the file as follows:

```
using System;
```

```
class Payroll
{
    static double WeeklyWage(double WeeklyHours, double HourlySalary)
    {
        double Total = WeeklyHours * HourlySalary;
        return Total;
    }

    static string GetName()
    {
        string FirstName, LastName, FN;

        FirstName = "Georgette";
        LastName = "Samsons";
    }
}
```

Coalesce

```
        FN = FirstName + " " + LastName;
        return FN;
    }

    static double GetHours(string FullName)
    {
        double Monday, Tuesday, Wednesday, Thursday, Friday, TotalHours;
        Monday = 8.00;
        Tuesday = 8.50;
        Wednesday = 8.50;
        Thursday = 7.50;
        Friday = 9;

        TotalHours = Monday + Tuesday + Wednesday + Thursday + Friday;
        return TotalHours;
    }

    static void Main()
    {
        string FullName;
        double Hours, Weekly;

        FullName = GetName();
        Hours = GetHours(FullName);
        Weekly = WeeklyWage(Hours, 12.55);

        Console.WriteLine("Employee's Payroll");
        Console.Write("Full Name:   ");
        Console.WriteLine(FullName);
        Console.Write("Weekly Hours:  ");
        Console.Write(Hours);
        Console.WriteLine(" hours");
        Console.Write("Weekly Salary: ");
        Console.WriteLine(Weekly);

        Console.WriteLine("\n");
    }
}
```

9. Compile and execute the program. This would produce:

```
Employee's Payroll
Full Name:   Georgette Samsons
Weekly Hours: 41.5 hours
Weekly Salary: 520.825
```

10. Return to Notepad
11. In the same way, a method can take as many arguments as you see fit. As an example, change the file as follows:

Coalesce

```
using System;

class Payroll
{
    static double WeeklyWage(double WeeklyHours, double HourlySalary)
    {
        double Total = WeeklyHours * HourlySalary;
        return Total;
    }

    static string GetName()
    {
        string FirstName, LastName, FN;

        FirstName = "Georgette";
        LastName = "Samsons";

        FN = FirstName + " " + LastName;
        return FN;
    }

    static double GetHours(string FullName)
    {
        double Monday, Tuesday, Wednesday, Thursday, Friday, TotalHours;
        Monday = 8.00;
        Tuesday = 8.50;
        Wednesday = 8.50;
        Thursday = 7.50;
        Friday = 9;

        TotalHours = Monday + Tuesday + Wednesday + Thursday + Friday;
        return TotalHours;
    }

    static double GetHourlySalary()
    {
        double HSalary;
        /*
        string SalValue;

        Console.Write("Enter Hourly Salary: ");
        SalValue = Console.ReadLine();
        HSalary = double.Parse(SalValue);
        */
        HSalary = 14.55;
        return HSalary;
    }

    static void DisplayPayroll(string Name, double Hours, double HSalary, double WSalary)
    {
        Console.WriteLine("Employee's Payroll");
        Console.WriteLine("Full Name:   ");
        Console.WriteLine(Name);
    }
}
```


Coalesce

```
        Console.WriteLine("Weekly Hours: ");
        Console.WriteLine(Hours);
        Console.WriteLine(" hours");
        Console.WriteLine("Hourly Salary: ");
        Console.WriteLine(HSalary);
        Console.WriteLine("Weekly Salary: ");
        Console.WriteLine(WSalary);
    }

    static void Main()
    {
        string FullName;
        double Hours, Hourly, Weekly;

        FullName = GetName();
        Hours = GetHours(FullName);
        Hourly = GetHourlySalary();
        Weekly = WeeklyWage(Hours, Hourly);

        DisplayPayroll(FullName, Hours, Hourly, Weekly);

        Console.WriteLine("\n");
    }
}
```

12. Compile and execute the program. This would produce:

```
Employee's Payroll
Full Name:   Georgette Samsons
Weekly Hours: 41.5 hours
Hourly Salary: 14.55
Weekly Salary: 603.825
```

13. Return to Notepad

Techniques of Passing Arguments

Passing Arguments by Value

When calling the methods that take one or more arguments, we made sure we provided the needed argument. This is because an argument is always required and the calling function or class must provided a valid value when calling such a method.

Practical Learning: Passing Arguments By Value

1. To start a new project, on the main menu of Notepad, click File -> New
2. In the empty file, type the following:

```
using System;
```

Coalesce

```
class Payroll
{
    static void Earnings(double ThisWeek, double Salary)
    {
        Console.WriteLine("\nIn the Earnings() function,");
        Console.Write("Weekly Hours  = ");
        Console.WriteLine(ThisWeek);
        Console.Write("Salary      = ");
        Console.WriteLine(Salary);
    }

    static void Main()
    {
        double Hours, Rate, Wage;
        string H, R, W;

        Console.Write("Enter the total Weekly hours:  ");
        H = Console.ReadLine();
        Console.Write("Enter the employee's hourly rate: ");
        R = Console.ReadLine();
        Rate = double.Parse(R);
        Hours = double.Parse(H);

        Console.WriteLine("\nIn the Main() function,");
        Console.Write("\nWeekly Hours  = ");
        Console.WriteLine(Hours);
        Console.Write("\nSalary      = ");
        Console.WriteLine(Rate);
        Console.Write("Weekly Salary  = ");
        Console.WriteLine(Hours * Rate);

        Console.WriteLine("\nCalling the Earnings() function");

        Earnings(Hours, Rate);

        Console.Write("\nAfter calling the Earnings() function, ");
        Console.WriteLine("\nin the Main() function,");
        Console.Write("\nWeekly Hours  = ");
        Console.WriteLine(Hours);
        Console.Write("\nSalary      = ");
        Console.WriteLine(Rate);
        Console.Write("Weekly Salary  = ");
        Console.WriteLine(Hours * Rate);

        Console.WriteLine("\n");
    }
}
```

3. Save the file in a new folder named **Payroll2**
4. Save the file itself as **exercise.cs**
5. Open the Command Prompt and switch to the folder that contains the current project

Coalesce

6. Compile the program by typing **csc exercise.cs**
7. Execute it by typing **exercise**
Here is an example of executing the program:

Enter the total Weekly hours: 35.50
Enter the employee's hourly rate: 12.42

In the Main() function,

Weekly Hours = 35.5
Salary = 12.42
Weekly Salary = 440.91

Calling the Earnings() function

In the Earnings() function,
Weekly Hours = 35.5
Salary = 12.42

After calling the Earnings() function,
in the main() function,

Weekly Hours = 35.5
Salary = 12.42
Weekly Salary = 440.91

8. Return to Notepad

Passing an Argument by Reference

Consider the following program:

```
using System;
```

```
class Payroll
{
    static void Earnings(double ThisWeek, double Salary)
    {
        ThisWeek = 42.50;

        Console.WriteLine("\nIn the Earnings() function,");
        Console.Write("Weekly Hours  = ");
        Console.WriteLine(ThisWeek);
        Console.Write("Salary      = ");
        Console.WriteLine(Salary);
        Console.Write("Weekly Salary: = ");
        Console.WriteLine(ThisWeek * Salary);
    }

    static void Main()
```

Coalesce

```
{
    double Hours, Rate;

    Rate = 15.58;
    Hours = 26.00;

    Console.WriteLine("In the Main() function,");
    Console.Write("\nWeekly Hours  = ");
    Console.Write(Hours);
    Console.Write("\nSalary      = ");
    Console.WriteLine(Rate);
    Console.Write("Weekly Salary  = ");
    Console.WriteLine(Hours * Rate);

    Console.WriteLine("\nCalling the Earnings() function");

    Earnings(Hours, Rate);

    Console.Write("\nAfter calling the Earnings() function, ");
    Console.WriteLine("\nin the Main() function,");
    Console.Write("\nWeekly Hours  = ");
    Console.Write(Hours);
    Console.Write("\nSalary      = ");
    Console.WriteLine(Rate);
    Console.Write("Weekly Salary  = ");
    Console.WriteLine(Hours * Rate);

    Console.Write("\n");
}
}
```

This would produce:

In the Main() function,

Weekly Hours = 26
Salary = 15.58
Weekly Salary = 405.08

Calling the Earnings() function

In the Earnings() function,
Weekly Hours = 42.5
Salary = 15.58
Weekly Salary: = 662.15

After calling the Earnings() function,
in the Main() function,

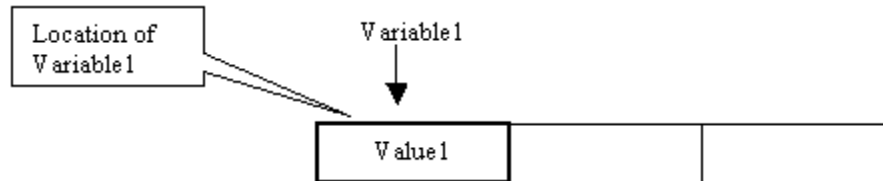
Weekly Hours = 26
Salary = 15.58
Weekly Salary = 405.08

Coalesce

Press any key to continue

Notice that the weekly hours and salary values are the same before and after calling the Earnings() method.

When you declare a variable in a program, the compiler reserves an amount of space for that variable. If you need to use that variable somewhere in your program, you call it and make use of its value. There are two major issues related to a variable: its value and its location in the memory.



The location of a variable in memory is referred to as its address.

If you supply the argument using its name, the compiler only makes a copy of the argument's value and gives it to the calling method. Although the calling method receives the argument's value and can use in any way, it cannot (permanently) alter it. C# allows a calling method to modify the value of a passed argument if you find it necessary. If you want the calling method to modify the value of a supplied argument and return the modified value, you should pass the argument using its reference.

To pass an argument as a reference, when defining and when calling the method, precede the argument's data type with the **ref** keyword. You can pass 0, one, or more arguments as reference in the program or pass all arguments as reference. The decision as to which argument(s) should be passed by value or by reference is based on whether or not you want the called method to modify the argument and permanently change its value.

Here are examples of passing some arguments by reference:

```
void Area(ref double Side); // The argument is passed by reference
bool Decision(ref char Answer, int Age); // One argument is passed by reference
// All arguments are passed by reference
float Purchase(ref float DiscountPrice, ref float NewDiscount, ref char Commission);
```

You add the ampersand when defining when calling the method. The above would be called with:

```
Area(ref Side);
Decision(ref Answer, Age);
Purchase(ref DiscountPrice, ref NewDiscount, ref Commission);
```

If you want a calling function to modify the value of an argument, you should supply its reference and not its value, using the **ref** keyword.

Coalesce

Practical Learning: Passing Arguments By Reference

1. To pass arguments by reference, change the file as follows:

```
using System;

class Payroll
{
    static void Earnings(ref double ThisWeek, double Salary)
    {
        ThisWeek = 42.50;

        Console.WriteLine("\nIn the Earnings() function,");
        Console.Write("Weekly Hours  = ");
        Console.WriteLine(ThisWeek);
        Console.Write("Salary      = ");
        Console.WriteLine(Salary);
        Console.Write("Weekly Salary: = ");
        Console.WriteLine(ThisWeek * Salary);
    }

    static void Main()
    {
        double Hours, Rate;

        Rate = 15.58;
        Hours = 26.00;

        Console.WriteLine("In the Main() function,");
        Console.Write("\nWeekly Hours  = ");
        Console.Write(Hours);
        Console.Write("\nSalary      = ");
        Console.WriteLine(Rate);
        Console.Write("Weekly Salary  = ");
        Console.WriteLine(Hours * Rate);

        Console.WriteLine("\nCalling the Earnings() function");

        Earnings(ref Hours, Rate);

        Console.Write("\nAfter calling the Earnings() function, ");
        Console.WriteLine("\nin the Main() function,");
        Console.Write("\nWeekly Hours  = ");
        Console.Write(Hours);
        Console.Write("\nSalary      = ");
        Console.WriteLine(Rate);
        Console.Write("Weekly Salary  = ");
        Console.WriteLine(Hours * Rate);

        Console.WriteLine("\n");
    }
}
```

Coalesce

}

2. Compile and execute it. This would produce:

In the `Main()` function,

Weekly Hours = 26
Salary = 15.58
Weekly Salary = 405.08

Calling the `Earnings()` function

In the `Earnings()` function,
Weekly Hours = 42.5
Salary = 15.58
Weekly Salary: = 662.15

After calling the `Earnings()` function,
in the `Main()` function,

Weekly Hours = 42.5
Salary = 15.58
Weekly Salary = 662.15

3. Notice that, this time, when the weekly hours and the weekly salary are accessed in the **Main()** function the second time, they have been permanently modified.
4. To finish, type Exit and press Enter

Coalesce

Chapter 8

The Properties of a Class

Overview of Properties

Introduction

In C++ and Java, when creating member variables of a class, programmers usually "hide" these members in private sections (C++) or create them as private (Java). This technique makes sure that a member variable is not accessed outside the class so that the clients of the class cannot directly influence the value of the member variable. If you create a member variable as private but still want other classes or functions to access or get the value of such a member variable, you must then create one or two functions used as "accessories", like a door in which the external functions or classes must pass through to access the member variable.

Accessories for Properties

C# provides a feature that was available so far to only a few languages (Visual Basic) or libraries (Borland's VCL). A property is a member of a class that plays as an intermediary to a member variable of the class. For example, if you have a member variable of class and that member represents the salary of an employee, a property can be the "door" that other functions or classes that need the salary must present their requests to. As such, these external functions and class cannot just change the salary or retrieve it as they wish. A property can be used to validate their request, to reject or to accept them.

As mentioned already, a property is used to "filter" access to a member variable of a class. Therefore, you start by declaring a (private (if you don't make it private, you may be deceiving the purpose of creating a property)) member variable:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;
    }

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


Coalesce

Obviously this private member variable cannot be accessed by a function or class outside of its class. Therefore, to let outside classes access this variable, you would/can create a property. Unlike **Managed C++** and the VCL (Borland **C++ Builder** and Delphi) where you use a keyword to indicate that you are creating a property, in C#, to create a property, there is a syntax you must follow. To start, you must create a member whose formula resembles a function without the parentheses. Therefore, you would start a property as follows:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;

        // This is a new property
        public double Side
        {
        }
    }
}
```

With regards to their role, there are two types of properties.

Practical Learning: Introducing Properties

1. Start a new file in Notepad
2. From what we know so far, type the following:

```
using System;

namespace CSharpLessons
{
    public class DepartmentStore
    {
        private string itemNo;
        private string cat;
        private string name;
        private string size;
        private double price;
    }

    public class Exercise
    {
        static void Main()
        {
        }
    }
}
```

Coalesce

3. Save it as **exercise.cs** in a new folder named **DeptStore1** inside of your CSharp Lessons folder
4. To create a property for each member variable, change the DepartmentStore class as follows:

```
using System;

namespace CSharpLessons
{
    public class DepartmentStore
    {
        private string itemNo;
        private string cat;
        private string name;
        private string size;
        private double price;

        // A property for the stock number of an item
        public string ItemNumber
        {
        }

        // A property for the category of item
        public string Category
        {
        }

        // A property for an item's name of an item
        public string ItemName
        {
        }

        // A property for size of a merchandise
        public string Size
        {
        }

        // A property for the marked price of an item
        public double UnitPrice
        {
        }
    }

    public class Exercise
    {
        static void Main()
        {
        }
    }
}
```

Coalesce

5. Save the file

Types of Properties

Property Readers

A property is referred to as *read* if its role is only to make available the value of the member variable it represents. To create a read property, in the body of the property, type the **get** keyword and create a body for the keyword, using the traditional curly brackets that delimit a section of code. Here is an example:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;

        // This is a new property
        public double Side
        {
            get
            {
            }
        }
    }
}
```

In the body of the **get** clause, you can implement the behavior that would be used to make the member variable's value available outside. The simplest way consists of just returning the corresponding member variable. Here is an example:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;

        // This is a new property
        public double Side
        {
            get
            {
                return _side;
            }
        }
    }
}
```

Coalesce

```
}
```

A read property is also referred to as read-only property because the clients of the class can only retrieve the value of the property but they cannot change it. Therefore, if you create (only) a read property, you should provide the users with the ability to primarily specify the value of the member variable. To do this, you can create an accessory method or a constructor for the class . Here is an example of such a constructor:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;

        // This is a new property
        public double Side
        {
            get
            {
                return _side;
            }
        }

        public Square(double s)
        {
            _side = s;
        }
    }
}
```

Once a read property has been created, other classes or functions can access it, for example they read its value as follows:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;

        // This is a new property
        public double Side
        {
            get
            {
                return _side;
            }
        }
    }
}
```

Coalesce

```
public Square(double s)
{
    _side = s;
}

class Exercise
{
    static void Main()
    {
        Square sq = new Square(-25.55);

        Console.WriteLine("Square Side: {0}", sq.Side);
    }
}
```

This would produce:

Square Side: -25.55

Press any key to continue

We described a property as serving as a door from outside to its corresponding member variable, preventing those outside classes or function to mess with the member variable. Notice that the Square class was given a negative value for the member variable, which is usually unrealistic for the side of a square. In this case and others, while still protecting the member variable as private, you can use the read property to reset the value of the member variable or even to reject it. To provide this functionality, you can create a conditional statement in the read property to perform a checking process. Here is an example:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;

        // This is a new property
        public double Side
        {
            get
            {
                if(_side < 0)
                    return 0;

                // else is implied
                return _side;
            }
        }
    }
}
```

Coalesce

```
    }

    public Square(double s)
    {
        _side = s;
    }
}

class Exercise
{
    static void Main()
    {
        Square sq1 = new Square(-12.48);
        Square sq2 = new Square(25.55);

        Console.WriteLine("First Square Characteristics");
        Console.WriteLine("Side:    {0}\n", sq1.Side);

        Console.WriteLine("Second Square Characteristics");
        Console.WriteLine("Side:    {0}\n", sq2.Side);
    }
}
```

This would produce:

First Square Characteristics
Side: 0

Second Square Characteristics
Side: 25.55

Press any key to continue

Practical Learning: Creating Property Readers

1. To create read properties, change the contents of the file as follows:

```
using System;

namespace CSharpLessons
{
    public class DepartmentStore
    {
        private string itemNo;
        private string cat;
        private string name;
        private string size;
        private double price;

        // A property for the stock number of an item
```

Coalesce

```
public string ItemNumber
{
    get
    {
        if( itemNo == "" )
            return "Invalid Item";
        else
            return itemNo;
    }
}

// A property for the category of item
public string Category
{
    get
    {
        if( cat == "" )
            return "Unknown Category";
        else
            return cat;
    }
}

// A property for an item's name of an item
public string ItemName
{
    get
    {
        if( name == "" )
            return "Item no Description";
        else
            return name;
    }
}

// A property for size of a merchandise
public string Size
{
    get
    {
        if( size == "" )
            return "Unknown Size or Fits All";
        else
            return size;
    }
}

// A property for the marked price of an item
public double UnitPrice
{
    get
    {
```

Coalesce

```
        if( price == 0 )
            return 0.00;
        else
            return price;
    }
}

public DepartmentStore(string nbr,
                        string ctg,
                        string nme,
                        string siz,
                        double prc)
{
    itemNo = nbr;
    cat    = ctg;
    name   = nme;
    size   = siz;
    price  = prc;
}

}

public class Exercise
{
    static void Main()
    {
        DepartmentStore store = new DepartmentStore("53564",
                                                    "Men",
                                                    "Khaki Pants Sahara",
                                                    "34",
                                                    24.95);

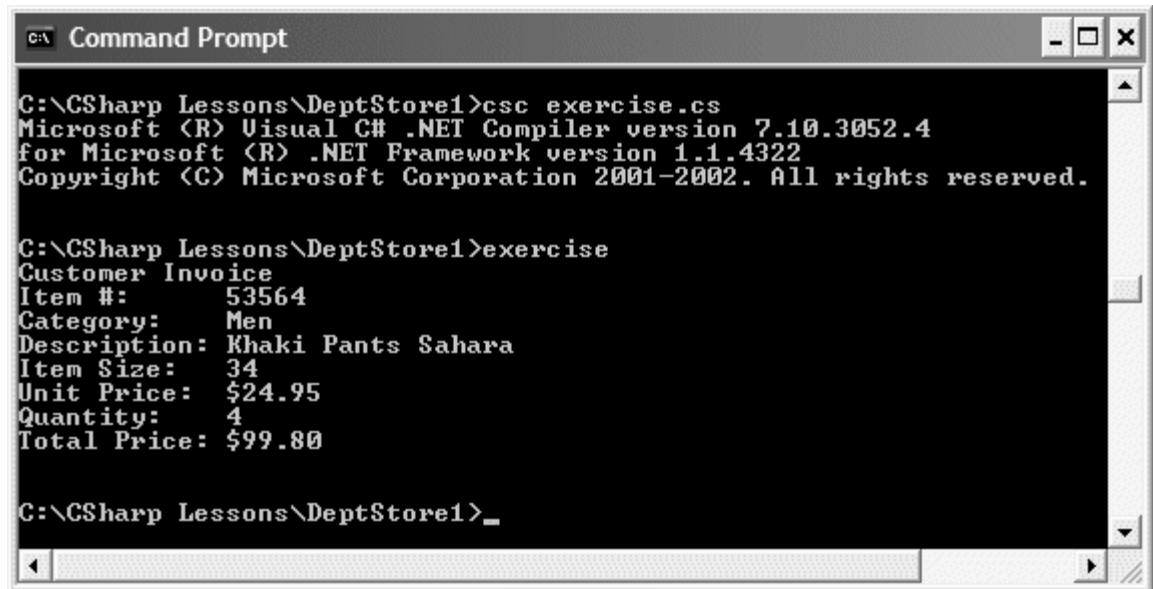
        int quantity = 4;
        double totalPrice = store.UnitPrice * quantity;

        Console.WriteLine("Customer Invoice");
        Console.WriteLine("Item #:    {0}", store.ItemNumber);
        Console.WriteLine("Category:  {0}", store.Category);
        Console.WriteLine("Description: {0}", store.ItemName);
        Console.WriteLine("Item Size:  {0}", store.Size);
        Console.WriteLine("Unit Price: {0}", store.UnitPrice.ToString("C"));
        Console.WriteLine("Quantity:  {0}", quantity);
        Console.WriteLine("Total Price: {0}\n", totalPrice.ToString("C"));
    }
}
}
```

2. Save the file and open the Command Prompt to the folder that contains the current exercise

Coalesce

3. Compile and execute the program:



```
C:\CSharp Lessons\DeptStore1>csc exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

C:\CSharp Lessons\DeptStore1>exercise
Customer Invoice
Item #:      53564
Category:    Men
Description:  Khaki Pants Sahara
Item Size:   34
Unit Price:  $24.95
Quantity:    4
Total Price: $99.80

C:\CSharp Lessons\DeptStore1>_
```

4. Return to your text editor

Property Writers

In our Square class so far, we were using a constructor to create a value for each of the necessary member variables. This meant that we had to always make sure that we knew the value of the member variable when we declared an instance of the class. Sometimes, this is not effective. For example, you cannot just call a constructor in the middle of the program, that is after the object has been declared, to assign a new value to the member variable. To solve this kind of problem, you must provide another means of accessing the member variable any time to change its value.

Besides, or instead of, retrieving the value of a member variable of a class, you may want external classes to be able to change the value of that member. Continuing with our policy to hide a member variable as private, you can create another type of property. A property is referred to as write if it can change (or write) the value of its corresponding member variable.

To create a write property, type the **set** keyword followed by the curly bracket delimiters. Here is an example:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;

        // This is a new property
        public double Side
        {
            set
            {
```

Coalesce

```
    }  
  }  
}
```

The minimum assignment you can perform with a write property is to assign it a value that would be provided by the outside world. To support this, C# provides the **value** keyword. Here is an example:

```
using System;  
  
namespace CSharpLessons  
{  
    public class Square  
    {  
        private double _side;  
  
        // This is a new property  
        public double Side  
        {  
            set  
            {  
                _side = value  
            }  
        }  
    }  
}
```

As you see, clients of a class can change the corresponding member variable of a member variable through the property writer. Based on this relationship, it is not unusual for a client of a class to make an attempt to "mess" with a member variable. For example, an external object can assign an invalid value to a member variable. Consider the following program:

```
using System;  
  
namespace CSharpLessons  
{  
    public class Square  
    {  
        private double _side;  
  
        // This is a new property  
        public double Side  
        {  
            get  
            {  
                return _side;  
            }  
  
            set  
            {  

```

Coalesce

```
        _side = value;
    }
}

public Square(double s)
{
    _side = s;
}

public double Perimeter()
{
    return Side * 4;
}

    public double Area()
    {
        return Side * Side;
    }
}

class Exercise
{
    static void Main()
    {
        Square sq1 = new Square();
        Square sq2 = new Square();

        sq1.Side = -12.48;
        sq2.Side = 25.55;

        Console.WriteLine("First Square Characteristics");
        Console.WriteLine("Side:    {0}", sq1.Side);
        Console.WriteLine("Perimeter: {0}", sq1.Perimeter());
        Console.WriteLine("Area:    {0}\n", sq1.Area());

        Console.WriteLine("Second Square Characteristics");
        Console.WriteLine("Side:    {0}", sq2.Side);
        Console.WriteLine("Perimeter: {0}", sq2.Perimeter());
        Console.WriteLine("Area:    {0}", sq2.Area());
    }
}
```

This would produce:

First Square Characteristics

Side: -12.48

Perimeter: -49.92

Area: 155.7504

Second Square Characteristics

Side: 25.55

Coalesce

Perimeter: 102.2
Area: 652.8025

Press any key to continue

Because of this, and since it is through the writer that the external objects would change the value of the member variable, you can use the write property, rather than the reader, to validate or reject a new value assigned to the member variable. Remember that the client objects of the class can only read the value of the member variable through the read property. Therefore, there may be only little concern on that side.

If you create a property that has only a **set** section, the property is referred to as write-only because the other classes or functions can only assign it a value (or write a value to it). If you create a property that has both a **get** and a **set** sections, its corresponding member variable can receive new values from outside the class and the member variable can provide its values to clients of the class.

Practical Learning: Creating Property Writers

1. To create property writers and complete the program, change the content of the file as follows:

```
using System;

namespace CSharpLessons
{
    public class DepartmentStore
    {
        private string itemNo;
        private string cat;
        private string name;
        private string size;
        private double price;

        // A property for the stock number of an item
        public string ItemNumber
        {
            get
            {
                return itemNo;
            }

            set
            {
                if( itemNo == "" )
                    itemNo = "Invalid Item";
                else
                    itemNo = value;
            }
        }

        // A property for the category of item
```

Coalesce

```
public string Category
{
    get
    {
        return cat;
    }

    set
    {
        if( cat == "" )
            cat = "Unknown Category";
        else
            cat = value;
    }
}

// A property for an item's name of an item
public string ItemName
{
    get
    {
        return name;
    }

    set
    {
        if( name == null )
            name = "Item no Description";
        else
            name = value;
    }
}

// A property for size of a merchandise
public string Size
{
    get
    {
        return size;
    }

    set
    {
        if( size == null )
            size = "Unknown Size or Fits All";
        else
            size = value;
    }
}

// A property for the marked price of an item
public double UnitPrice
```

Coalesce

```
    {
    get
    {
        return price;
    }

    set
    {
        if( price < 0 )
            price = 0.00;
        else
            price = value;
    }
    }
}

public class Exercise
{
    static void Main()
    {
        int quantity = 0;
        DepartmentStore item1 = new DepartmentStore();
        double totalPrice = 0;

        item1.ItemNumber = "G76-85";
        item1.ItemName   = "Men Something";
        item1.Size       = "36L28W";
        quantity         = 6;
        totalPrice = item1.UnitPrice * quantity;

        Console.WriteLine("Customer Invoice - Bad Receipt");
        Console.WriteLine("Item #:    {0}", item1.ItemNumber);
        Console.WriteLine("Category:  {0}", item1.Category);
        Console.WriteLine("Description: {0}", item1.ItemName);
        Console.WriteLine("Item Size:  {0}", item1.Size);
        Console.WriteLine("Unit Price: {0}", item1.UnitPrice.ToString("C"));
        Console.WriteLine("Quantity:  {0}", quantity);
        Console.WriteLine("Total Price: {0}\n", totalPrice.ToString("C"));

        item1.ItemNumber = "74797";
        item1.Category    = "Women";
        item1.ItemName    = "Suit Gallantry";
        item1.Size        = "10-1/4";
        item1.UnitPrice   = 225.75;
        quantity          = 2;
        totalPrice = item1.UnitPrice * quantity;

        Console.WriteLine("Customer Invoice - 12/08/2002");
        Console.WriteLine("Item #:    {0}", item1.ItemNumber);
        Console.WriteLine("Category:  {0}", item1.Category);
        Console.WriteLine("Description: {0}", item1.ItemName);
        Console.WriteLine("Item Size:  {0}", item1.Size);
    }
}
```

Coalesce

```
        Console.WriteLine("Unit Price: {0}", item1.UnitPrice.ToString("C"));
        Console.WriteLine("Quantity: {0}", quantity);
        Console.WriteLine("Total Price: {0}\n", totalPrice.ToString("C"));
    }
}
```

2. Close the file. When asked whether you want to save it, click Yes
3. Switch to the Command Prompt
4. Compile and execute the program:

```
C:\CSharp Lessons\DeptStore1>csc exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.
```

```
C:\CSharp Lessons\DeptStore1>exercise
Customer Invoice - Bad Receipt
Item #:    G76-85
Category:
Description: Item no Description
Item Size: Unknown Size or Fits All
Unit Price: $0.00
Quantity:  6
Total Price: $0.00
```

```
Customer Invoice - 12/08/2002
Item #:    74797
Category:  Women
Description: Suit Gallantry
Item Size: 10-1/4
Unit Price: $225.75
Quantity:  2
Total Price: $451.50
```

```
C:\CSharp Lessons\DeptStore1>
```

5. Close the Command Prompt

Coalesce

Chapter 9

Interactions With Other Classes

Overview

A Class as a Member Variable

Just like any of the variables we have used so far, you can make any normal class a member variable of another class. To use a class in your own class, of course you must have that class. You can use one of the classes already available in C#, as we will see in the next lesson, or you can first create your own class. Here is an example of a class:

```
class MVADate
{
    private int DayHired;
    private int MonthHired;
    private int YearHired;
}
```

To use a class as a member of your class, simply declare its variable as you would proceed with any of the member variables we have declared so far. Here is an example:

```
using System;
```

```
class MVADate
{
    private int DayHired;
    private int MonthHired;
    private int YearHired;
}
```

```
class MotorVehicleAdministration
{
    MVADate MDate;

    static int Main()
    {
        return 0;
    }
}
```

After a class has been declared as a member variable of another class, it can be used regularly. Because the member is a class, declared as a reference (in (Managed) C++, it would be declared as a pointer) instead of a value type, there are some rules you must follow to use it. After declaring the member variable, you must make sure you have allocated memory for it on the heap. You must also make sure that the variable is initialized appropriately before it can be used; otherwise you would receive an error when compiling the file.

Coalesce

Imagine you declare a variable of class A as a member of class B. If you declare a variable of class B in an external function, as we have done so far in the **Main()** function, to access a member of class A, you must fully qualify it.

Practical Learning: Declaring a Class as a Member Variable

1. Start Notepad and, in the empty file, type the following:

```
using System;

class MVADate
{
    private int dayOfBirth;
    private int monthOfBirth;
    private int yearOfBirth;

    public void SetDate(int d, int m, int y)
    {
        dayOfBirth = d;
        monthOfBirth = m;
        yearOfBirth = y;
    }

    public string ProduceDate()
    {
        string result = dayOfBirth + "/" + monthOfBirth + "/" + yearOfBirth;

        return result;
    }
}

class MotorVehicleAdministration
{
    public MotorVehicleAdministration()
    {
        birthdate = new MVADate();
    }

    private string fullName;
    private MVADate birthdate;
    private bool isAnOrganDonor;

    static int Main()
    {
        MotorVehicleAdministration MVA = new MotorVehicleAdministration();

        Console.WriteLine("To process a registration, enter the information");
        Console.Write("Full Name: ");
        MVA.fullName = Console.ReadLine();
        Console.Write("Day of Birth: ");
        int d = int.Parse(Console.ReadLine());
```

Coalesce

```
        Console.WriteLine("Month of Birth: ");
        int m = int.Parse(Console.ReadLine());
        Console.WriteLine("Year of Birth: ");
        int y = int.Parse(Console.ReadLine());
        Console.WriteLine("Is the application an organ donor (0=No/1=Yes)? ");
        string ans = Console.ReadLine();

        if( ans == "0" )
            MVA.isAnOrganDonor = false;
        else
            MVA.isAnOrganDonor = true;

        MVA.birthdate.SetDate(d, m, y);

        Console.WriteLine("\n ==- Motor Vehicle Administration ==-");
        Console.WriteLine(" ==- Driver's License Application ==-");
        Console.WriteLine("Full Name:   {0}", MVA.fullName);
        Console.WriteLine("Dateof Birth: {0}", MVA.birthdate.ProduceDate());
        Console.WriteLine("Organ Donor? {0}", MVA.isAnOrganDonor);

        Console.WriteLine();
        return 0;
    }
}
```

2. Save the file as **exercise.cs** in a new folder called **Members1** created inside your CSharp Lessons folder
3. Open the Command Prompt and switch to the folder that contains the current exercise
4. Compile the program with **csc exercise.cs**
5. Execute it by typing exercise
Here is an example:

```
C:\CSharp Lessons\Members1>csc exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.
```

```
C:\CSharp Lessons\Members1>exercise
To process a registration, enter the information
Full Name:   Ernestine Mvouama
Day of Birth: 8
Month of Birth: 14
Year of Birth: 1984
Is the application an organ donor (0=No/1=Yes)? 1
```

```
==- Motor Vehicle Administration ==-
==- Driver's License Application ==-
Full Name:   Ernestine Mvouama
Dateof Birth: 8/14/1984
```

Coalesce

Organ Donor? True

6. Return to Notepad

Returning a Class From a Class' Method

Like a value from regular type, you can return a class value from a method of a function. To do this, you can first declare the member function and specify the class as the return type. Here is an example:

```
using System;

class MVADate
{
}

class MotorVehicleAdministration
{
    private static MVADate RequestDateOfBirth()
    {
    }

    static int Main()
    {
        MotorVehicleAdministration MVA = new MotorVehicleAdministration();

        Console.WriteLine();
        return 0;
    }
}
```

After implementing the function, you must return a value that is conform to the class, otherwise you would receive an error when compiling the application. You can proceed by declaring a variable of the class in the body of the function, initializing the variable, and then returning it.

Once a function has returned a value of a class, the value can be used as normally as possible.

Practical Learning: Returning a Class From a Method

1. Change the file as follows:

```
using System;

class MVADate
{
    private int dayOfBirth;
    private int monthOfBirth;
    private int yearOfBirth;

    public void SetDate(int d, int m, int y)
    {
    }
}
```

Coalesce

```
        dayOfBirth = d;
        monthOfBirth = m;
        yearOfBirth = y;
    }

    public string ProduceDate()
    {
        string result = dayOfBirth + "/" + monthOfBirth + "/" + yearOfBirth;

        return result;
    }
}

class MotorVehicleAdministration
{
    public MotorVehicleAdministration()
    {
        birthdate = new MVADate();
    }

    private string fullName;
    private MVADate RequestDateOfBirth();
    private bool isAnOrganDonor;

    private static MVADate RequestDateOfBirth()
    {
        MVADate date = new MVADate();

        Console.Write("Day of Birth: ");
        int d = int.Parse(Console.ReadLine());
        Console.Write("Month of Birth: ");
        int m = int.Parse(Console.ReadLine());
        Console.Write("Year of Birth: ");
        int y = int.Parse(Console.ReadLine());

        date.SetDate(d, m, y);
        return date;
    }

    static int Main()
    {
        MotorVehicleAdministration MVA = new MotorVehicleAdministration();

        Console.WriteLine("To process a registration, enter the information");
        Console.Write("Full Name: ");
        MVA.fullName = Console.ReadLine();

        MVA.birthdate = RequestDateOfBirth();

        Console.Write("Is the application an organ donor (0=No/1=Yes)? ");
        string ans = Console.ReadLine();
    }
}
```

Coalesce

```
        if( ans == "0" )
            MVA.isAnOrganDonor = false;
        else
            MVA.isAnOrganDonor = true;

        Console.WriteLine("\n == Motor Vehicle Administration ==");
        Console.WriteLine(" == Driver's License Application ==");
        Console.WriteLine("Full Name: {0}", MVA.fullName);
        Console.WriteLine("Dateof Birth: {0}", MVA.birthdate.ProduceDate());
        Console.WriteLine("Organ Donor? {0}", MVA.isAnOrganDonor);

        Console.WriteLine();
        return 0;
    }
}
```

2. Save it and switch to the Command Prompt
3. Compile and execute the application

A Class as Argument

Passing a Class as Argument

Once a class has been created, it can be used like any other variable. For example, its variable can be passed as argument to a method of another class. When a class is passed as argument, its public members are available to the function that uses it. Here is an example:

```
class CarRegistration
{
    public void DisplayDriversLicense(MotorVehicleAdministration mva)
    {
        Console.WriteLine("Full Name: {0}", mva.fullName);
        Console.WriteLine("Dateof Birth: {0}", mva.birthdate.ProduceDate());
        Console.WriteLine("Organ Donor? {0}", mva.isAnOrganDonor);
    }
}

class MotorVehicleAdministration
{
    public string fullName;
    public MVADate birthdate;
    public bool isAnOrganDonor;
}
```

In the same way, you can pass more than one class as arguments to a function. Because classes are always used by reference, when passing a class as argument, it is implied to be passed by reference. To reinforce this, you can type the ref keyword to the left of the argument. Here is an example:

```
class CarRegistration
{
```

Coalesce

```
public void DisplayDriversLicense(ref MotorVehicleAdministration mva)
{
    Console.WriteLine("Full Name:  {0}", mva.fullName);
    Console.WriteLine("Dateof Birth: {0}", mva.birthdate.ProduceDate());
    Console.WriteLine("Organ Donor? {0}", mva.isAnOrganDonor);
}
}
```

Practical Learning: Passing a Class as Argument

1. To pass a class as argument, change the file as follows:

```
using System;

namespace ConsoleApplication1
{
    class MVADate
    {
        private int dayOfBirth;
        private int monthOfBirth;
        private int yearOfBirth;

        public void SetDate(int d, int m, int y)
        {
            dayOfBirth  = d;
            monthOfBirth = m;
            yearOfBirth  = y;
        }

        public string ProduceDate()
        {
            string result = dayOfBirth + "/" + monthOfBirth + "/" + yearOfBirth;

            return result;
        }
    }

    class Car
    {
        public string Make;
        public string Model;
        public int   CarYear;
    }

    class CarRegistration
    {
        public void DisplayDriversLicense(MotorVehicleAdministration mva)
        {
            Console.WriteLine("\n == Motor Vehicle Administration ==");
            Console.WriteLine(" == Car Registration ==");
            Console.WriteLine("Full Name:  {0}", mva.fullName);
        }
    }
}
```

Coalesce

```
        Console.WriteLine("Dateof Birth: {0}", mva.birthdate.ProduceDate());
        Console.WriteLine("Organ Donor? {0}", mva.isAnOrganDonor);
    }

    public void DisplayCarInformation(Car v)
    {
        Console.WriteLine("Car:      {0} {1}", v.Make, v.Model);
        Console.WriteLine("Year:    {0}", v.CarYear);
    }
}

class MotorVehicleAdministration
{
    public MotorVehicleAdministration()
    {
        birthdate = new MVADate();
    }

    public string  fullName;
    public MVADate birthdate;
    public bool   isAnOrganDonor;
    public Car    NewCar;

    private static MVADate RequestDateOfBirth()
    {
        MVADate date = new MVADate();

        Console.Write("Day of Birth: ");
        int d = int.Parse(Console.ReadLine());
        Console.Write("Month of Birth: ");
        int m = int.Parse(Console.ReadLine());
        Console.Write("Year of Birth: ");
        int y = int.Parse(Console.ReadLine());

        date.SetDate(d, m, y);
        return date;
    }

    private static Car RegisterCar()
    {
        Car c = new Car();

        Console.Write("Make: ");
        c.Make = Console.ReadLine();
        Console.Write("Model: ");
        c.Model = Console.ReadLine();
        Console.Write("Year: ");
        c.CarYear = int.Parse(Console.ReadLine());

        return c;
    }
}
```

Coalesce

```
static void Main()
{
    MotorVehicleAdministration MVA = new MotorVehicleAdministration();
    CarRegistration regist = new CarRegistration();
    Car vehicle = new Car();

    Console.WriteLine("To process a registration, enter the information");
    Console.Write("Full Name:  ");
    MVA.fullName = Console.ReadLine();
    MVA.birthdate = RequestDateOfBirth();
    Console.Write("Is the application an organ donor (0=No/1=Yes)? ");
    string ans = Console.ReadLine();

    if( ans == "0" )
        MVA.isAnOrganDonor = false;
    else
        MVA.isAnOrganDonor = true;

    Console.WriteLine("Enter the following information for Car Registration");
    MVA.NewCar = RegisterCar();

    regist.DisplayDriversLicense(MVA);
    regist.DisplayCarInformation(MVA.NewCar);

    Console.WriteLine();
}
}
```

2. Save it and switch to the Command Prompt
3. Compile and execute the application. Here is an example:

```
To process a registration, enter the information
Full Name:  Alan Scott
Day of Birth:  2
Month of Birth: 10
Year of Birth: 1974
Is the application an organ donor (0=No/1=Yes)? 0
Enter the following information for Car Registration
Make: Ford
Model: Crown Victoria
Year: 1998
```

```
-- Motor Vehicle Administration --
-- Car Registration --
Full Name:  Alan Scott
Dateof Birth: 2/10/1974
Organ Donor? False
Car:      Ford Crown Victoria
Year:     1998
```







Coalesce

Chapter 10

Inheritance

Introduction to Inheritance

Definition

				
Volley Ball	Football	Basketball	Handball	Golf

The primary characteristic of the objects on the above pictures is that they are balls used in different sports. Another characteristic they share is that they are round. On the other hand, although these balls are used in sport, one made for one sport cannot (or should not) be used in another sport (of course, it is not unusual for a footballer to mess with a basketball on a lawn but it is not appropriate). The common characteristics of these objects can be listed in a group like a C# class. The class would appear as:

```
class Ball
{
    TypeOfSport;
    Size;
}
```

If you were asked to create a class to represent these balls, you may be tempted to implement a general class that defines each ball. This may be a bad idea because, despite their round resemblance, there are many internal differences among these balls. Programming languages like C# provide an alternate solution to this type of situation.

Inheritance consists of creating a class whose primary definition or behavior is based on another class. In other words, inheritance starts by having a class that can provide behavior that other classes can improve on.

Class Derivation

As you may have guess, in order to implement inheritance, you must first have a class that provides the fundamental definition or behavior you need. There is nothing magical about such a class. It could appear exactly like any of the classes we have used so far. Here is an example:

```
using System;

class Circle
{
    private double _radius;
```

Coalesce

```
public double Radius
{
    get
    {
        if( _radius < 0 )
            return 0.00;
        else
            return _radius;
    }
    set
    {
        _radius = value;
    }
}

public double Diameter
{
    get
    {
        return Radius * 2;
    }
}

public double Circumference
{
    get
    {
        return Diameter * 3.14159;
    }
}

public double Area
{
    get
    {
        return Radius * Radius * 3.14159;
    }
}
}

class Exercise
{
    public static int Main()
    {
        Circle c = new Circle();
        c.Radius = 25.55;

        Console.WriteLine("Circle Characteristics");
        Console.WriteLine("Side:   {0}", c.Radius);
        Console.WriteLine("Diameter: {0}", c.Diameter);
        Console.WriteLine("Circumference: {0}", c.Circumference);
        Console.WriteLine("Area:   {0}", c.Area);

        return 0;
    }
}
```

Coalesce

```
}  
}
```

This would produce:

```
Circle Characteristics  
Side: 25.55  
Diameter: 51.1  
Circumference: 160.535249  
Area: 2050.837805975  
Press any key to continue
```

The above class is used to process a circle. It can request or provide a radius. It can also calculate the circumference and the area of a circle. Now, suppose you want to create a class for a sphere. You could start from scratch as we have done so far. On the other hand, since a sphere is primarily a 3-dimensional circle, and if you have a class for a circle already, you can simply create your sphere class that uses the already implemented behavior of a circle class.

Creating a class that is based on another class is also referred to as deriving a class from another. The first class serves as parent or base. The class that is based on another class is also referred to as child or derived. To create a class based on another, you use the following formula:

```
class NewChild : Base  
{  
    // Body of the new class  
}
```

In this formula, you start with the class keyword followed by a name from your class. On the right side of the name of your class, you must type the : operator, followed by the name of the class that will serve as parent. Of course, the *Base* class must have been defined; that is, the compiler must be able to find its definition. Based on the above formula, you can create a sphere class based on the earlier mentioned Circle class as follows:

```
class Sphere : Circle  
{  
    // The class is ready  
}
```

After deriving a class, it becomes available and you can use it just as you would any other class. Here is an example:

```
using System;  
  
class Circle  
{  
    private double _radius;  
  
    public double Radius  
    {  
        get  
        {  
            if( _radius < 0 )  
                return 0.00;  
            else
```

Coalesce

```
        return _radius;
    }
    set
    {
        _radius = value;
    }
}
public double Diameter
{
    get
    {
        return Radius * 2;
    }
}
public double Circumference
{
    get
    {
        return Diameter * 3.14159;
    }
}
public double Area
{
    get
    {
        return Radius * Radius * 3.14159;
    }
}
}

class Sphere : Circle
{
}

class Exercise
{
    public static int Main()
    {
        Circle c = new Circle();
        c.Radius = 25.55;

        Console.WriteLine("Circle Characteristics");
        Console.WriteLine("Side:   {0}", c.Radius);
        Console.WriteLine("Diameter: {0}", c.Diameter);
        Console.WriteLine("Circumference: {0}", c.Circumference);
        Console.WriteLine("Area:   {0}", c.Area);

        Sphere s = new Sphere();
        s.Radius = 25.55;

        Console.WriteLine("\nSphere Characteristics");
        Console.WriteLine("Side:   {0}", s.Radius);
    }
}
```

Coalesce

```
        Console.WriteLine("Diameter: {0}", s.Diameter);
        Console.WriteLine("Circumference: {0}", s.Circumference);
        Console.WriteLine("Area:    {0}", s.Area);

        return 0;
    }
}
```

This would produce:

```
Circle Characteristics
Side:   25.55
Diameter: 51.1
Circumference: 160.535249
Area:   2050.837805975

Sphere Characteristics
Side:   25.55
Diameter: 51.1
Circumference: 160.535249
Area:   2050.837805975
Press any key to continue
```

When a class is based on another class, all public (we will also introduce another inheritance-oriented keyword for this issue) members (member variables and methods) of the parent class are made available to the derived class that can use them as easily. While other functions and classes can also use the public members of a class, the difference is that the derived class can call the public members of the parent as if they belonged to the derived class. That is, the child class doesn't have to "qualify" the public members of the parent class when these public members are used in the body of the derived class. This is illustrated in the following program:

```
using System;

class Circle
{
    private double _radius;

    public double Radius
    {
        get
        {
            if( _radius < 0 )
                return 0.00;
            else
                return _radius;
        }
        set
        {
            _radius = value;
        }
    }

    public double Diameter
    {
```

Coalesce

```
        get
        {
            return Radius * 2;
        }
    }
    public double Circumference
    {
        get
        {
            return Diameter * 3.14159;
        }
    }
    public double Area
    {
        get
        {
            return Radius * Radius * 3.14159;
        }
    }

    public void ShowCharacteristics()
    {
        Console.WriteLine("Circle Characteristics");
        Console.WriteLine("Side:  {0}", Radius);
        Console.WriteLine("Diameter: {0}", Diameter);
        Console.WriteLine("Circumference: {0}", Circumference);
        Console.WriteLine("Area:   {0}", Area);
    }
}

class Sphere : Circle
{
    public void ShowCharacteristics()
    {
        // Because Sphere is based on Circle, you can access
        // any public member(s) of Circle without qualifying it(them)
        Console.WriteLine("\nSphere Characteristics");
        Console.WriteLine("Side:  {0}", Radius);
        Console.WriteLine("Diameter: {0}", Diameter);
        Console.WriteLine("Circumference: {0}", Circumference);
        Console.WriteLine("Area:   {0}\n", Area);
    }
}

class Exercise
{
    public static int Main()
    {
        Circle c = new Circle();

        c.Radius = 25.55;
        c.ShowCharacteristics();
    }
}
```

Coalesce

```
        Sphere s = new Sphere();

        s.Radius = 25.55;
        s.ShowCharacteristics();

        return 0;
    }
}
```

This would produce the same result

Implementation of Derived Members

You can notice in the above example that the derived class produce the same results as the base class. In reality, inheritance is used to solve various Object-Oriented Programming (OOP) problems. One of them consists of customizing, adapting, or improving the behavior of a feature (property or method, etc) of the parent class. For example, although both the circle and the sphere have an area, their areas are not the same. A circle is a flat surface but a sphere is a volume, which makes its are very much higher. Therefore, since they use different formulas for their respective area, you should implement a new version of the area in the sphere. Based on this, when deriving your class from another class, you should aware of the properties and methods of the base class so that, if you know that the parent class has a certain behavior or a characteristic that is not conform to the new derived class, you can do something about that (this is not always possible). A new version of the area in the sphere can be calculated as follows:

```
using System;

class Circle
{
    private double _radius;

    public double Radius
    {
        get
        {
            if( _radius < 0 )
                return 0.00;
            else
                return _radius;
        }
        set
        {
            _radius = value;
        }
    }
    public double Diameter
    {
        get
        {
            return Radius * 2;
        }
    }
}
```

Coalesce

```
    }
    public double Circumference
    {
        get
        {
            return Diameter * 3.14159;
        }
    }
    public double Area
    {
        get
        {
            return Radius * Radius * 3.14159;
        }
    }
}

class Sphere : Circle
{
    public double Area
    {
        get
        {
            return 4 * Radius * Radius * 3.14159;
        }
    }
}

class Exercise
{
    public static int Main()
    {
        Circle c = new Circle();
        c.Radius = 25.55;

        Console.WriteLine("Circle Characteristics");
        Console.WriteLine("Side: {0}", c.Radius);
        Console.WriteLine("Diameter: {0}", c.Diameter);
        Console.WriteLine("Circumference: {0}", c.Circumference);
        Console.WriteLine("Area: {0}", c.Area);

        Sphere s = new Sphere();
        s.Radius = 25.55;

        Console.WriteLine("\nSphere Characteristics");
        Console.WriteLine("Side: {0}", s.Radius);
        Console.WriteLine("Diameter: {0}", s.Diameter);
        Console.WriteLine("Circumference: {0}", s.Circumference);
        Console.WriteLine("Area: {0}", s.Area);

        return 0;
    }
}
```


Coalesce

```
}

    This would produce:

Circle Characteristics
Side:   25.55
Diameter: 51.1
Circumference: 160.535249
Area:   2050.837805975

Sphere Characteristics
Side:   25.55
Diameter: 51.1
Circumference: 160.535249
Area:   8203.3512239

Press any key to continue
```

Notice that, this time, the areas of both figures are not the same even though their radii are similar.

Besides customizing member variables and methods of a parent class, you can add new members as you wish. This is another valuable feature of inheritance. In our example, while the is a flat shape, a sphere has a volume. In this case, you may need to calculate the volume of a sphere as a new method or property of the derived class. Here is an example:

```
using System;

class Circle
{
    private double _radius;

    public double Radius
    {
        get
        {
            if( _radius < 0 )
                return 0.00;
            else
                return _radius;
        }
        set
        {
            _radius = value;
        }
    }

    public double Diameter
    {
        get
        {
            return Radius * 2;
        }
    }

    public double Circumference
```

Coalesce

```
{
    get
    {
        return Diameter * 3.14159;
    }
}
public double Area
{
    get
    {
        return Radius * Radius * 3.14159;
    }
}
}

class Sphere : Circle
{
    public double Area
    {
        get
        {
            return 4 * Radius * Radius * 3.14159;
        }
    }

    public double Volume
    {
        get
        {
            return 4 * 3.14159 * Radius * Radius * Radius;
        }
    }
}

class Exercise
{
    public static int Main()
    {
        Circle c = new Circle();
        c.Radius = 25.55;

        Console.WriteLine("Circle Characteristics");
        Console.WriteLine("Side: {0}", c.Radius);
        Console.WriteLine("Diameter: {0}", c.Diameter);
        Console.WriteLine("Circumference: {0}", c.Circumference);
        Console.WriteLine("Area: {0}", c.Area);

        Sphere s = new Sphere();
        s.Radius = 25.55;

        Console.WriteLine("\nSphere Characteristics");
        Console.WriteLine("Side: {0}", s.Radius);
```

Coalesce

```
        Console.WriteLine("Diameter: {0}", s.Diameter);
        Console.WriteLine("Circumference: {0}", s.Circumference);
        Console.WriteLine("Area:    {0}", s.Area);
        Console.WriteLine("Volume:  {0}\n", s.Volume);

        return 0;
    }
}
```

This would produce:

Circle Characteristics
Side: 25.55
Diameter: 51.1
Circumference: 160.535249
Area: 2050.837805975

Sphere Characteristics
Side: 25.55
Diameter: 51.1
Circumference: 160.535249
Area: 8203.3512239
Volume: 209595.623770645

Press any key to continue

In the same way, you can add as many new members as you judge necessary for the derived class.

Protected Members

To maintain a privileged relationship with its children, a parent class can make a set list of members available only to classes derived from it. To do this, some members of a parent class have a protected access level. Of course, as the class creator, it is your job to specify this relationship.

To create a member (property or method) that only derived classes can access, type the protected keyword to its left. Here is an example:

```
class Circle
{
    protected double _radius;

    ...
}
```

Virtual Members

We have just mentioned that you can create a new version of a member in a derived class for a member that already exists in the parent class. After doing this, when you call that member in your program, you need to make sure that the right member gets called, the member in the base class or the equivalent member in the derived class.

Coalesce

When you create a base class, if you anticipate that a certain property or method would need to be redefined in the derived class, you can indicate this to the compiler. On the other hand, while creating your classes, if you find out that you are customizing a property or method that already existed in the base class, you should let the compiler that you are providing a new version. In both cases, the common member should be created as virtual. To do this, in the base class, type the **virtual** keyword to the left of the property or method. Based on this, the Area property of our Circle class can be created as follows:

```
class Circle
{
    ...

    public virtual double Area
    {
        get
        {
            return Radius * Radius * 3.14159;
        }
    }
}
```

In fact, in C#, unlike C++, if you omit the virtual keyword, the compiler would display a warning. When customizing virtual members in the derived class, to indicate that a member is already virtual in the base class and that you are defining a new version, type the **override** keyword to the left of its declaration. For example, the Area property in our Sphere class can be created as follows:

```
class Sphere : Circle
{
    public override double Area
    {
        get
        {
            return 4 * Radius * Radius * 3.14159;
        }
    }

    public double Volume
    {
        get
        {
            return 4 * 3.14159 * Radius * Radius * Radius;
        }
    }
}
```

Coalesce

Characteristics of Inheritance

Abstract Classes

In C#, you can create a class whose role is only meant to provide fundamental characteristics for other classes. This type of class cannot be used to declare a variable of the object. Such a class is referred to as abstract. Therefore, an abstract class can be created only to serve as a parent class for others. To create an abstract, type the abstract keyword to the left of its name. Here is an example:

```
public abstract class Ball
{
    protected int TypeOfSport;
    protected string Dimensions;
}
```

Even if you create a normal property or method, you can also make it abstract. In this case, you would not need to define the value or behavior of the member. The derived class would implement it with the new behavior as they wish.

Sealed Classes

Any of the classes we have used so far in our lessons can be inherited from. If you create a certain and don't want anybody to derive another class from it, you can mark it as sealed. In other words, a sealed class is one that cannot serve as base for another class.

To mark a class as sealed, type the sealed keyword to its left. Here is an example:

```
public sealed class Ball
{
    public int TypeOfSport;
    public string Dimensions;
}
```

Coalesce

Chapter 11

Built-In Classes

C# was clearly created to improve on C++ and possibly offer a new alternative. To achieve this goal, Microsoft created a huge library to accompany the language. The Microsoft .NET Framework is a big library made of various classes and constants you can directly use in your C# application without necessarily explicitly loading an external library. To start, this main library of C# provides a class called **Object**.

As you may have realized by now, everything variable or function in C# (as in Java) must belong to a class, unlike C/C++ where you can have global variables or functions. Therefore, you always have to create at least one class for your application. As such, when you create a class, it automatically inherits its primary characteristics from the parent of all classes: **Object**.

Practical Learning: Introducing Ancestor Classes

1. Start Notepad and, in the empty file, type the following:

```
using System;

class Sport
{
    private double _ballWeight;
    private int    _players;
    private double _courtLength;
    private double _courtWidth;

    public double BallWeight
    {
        get { return _ballWeight; }
        set { _ballWeight = value; }
    }

    public int NumberOfPlayers
    {
        get { return _players; }
        set { _players = value; }
    }

    public double CourtLength
    {
        get { return _courtLength; }
        set { _courtLength = value; }
    }

    public double CourtWidth
    {
        get { return _courtWidth; }
        set { _courtWidth = value; }
    }
}
```

Coalesce

```
    }  
}  
  
class Exercise  
{  
    static int Main()  
    {  
        Sport tennis = new Sport();  
  
        tennis.BallWeight    = 57.50; // grams  
        tennis.NumberOfPlayers = 1;   // Singles game  
        tennis.CourtLength   = 23.70; // meters  
        tennis.CourtWidth    = 8.23; // meters;  
  
        Console.WriteLine("\nGame Characteristics");  
        Console.WriteLine("Ball Weight:      {0} grams", tennis.BallWeight);  
        Console.WriteLine("Players on each side: {0}", tennis.NumberOfPlayers);  
        Console.WriteLine("Court Dimensions(LxW): {0}m X {1}m\n",  
                           tennis.CourtLength, tennis.CourtWidth);  
  
        return 0;  
    }  
}
```

2. Save the file as **exercise.cs** in a new folder called **Inherited** created inside your CSharp Lessons folder
3. Open the Command Prompt and switch to the folder that contains the current exercise
4. Compile the program with **csc exercise.cs**
5. Execute it by typing exercise
Here is an example:

```
C:\CSharp Lessons\Inherited>csc exercise.cs  
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4  
for Microsoft (R) .NET Framework version 1.1.4322  
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.
```

```
C:\CSharp Lessons\Inherited>exercise
```

```
Game Characteristics  
Ball Weight:      57.5 grams  
Players on each side: 1  
Court Dimensions(LxW): 23.7m X 8.23m
```

```
C:\CSharp Lessons\Inherited>
```

6. Return to Notepad

Coalesce

Equality of Two Class Variables

When you declare and initialize two variables, one of the operations you may want to subsequently perform is to compare their value. To support this operation, the **Object** class provides its children with a method called **Equals**. The **Equals()** method comes in two versions. The first has the following syntax:

```
public virtual bool Equals(object obj);
```

This version allows you to call the **Equals()** method on a declared variable and pass the other variable as argument. Here is an example:

```
using System;

class BookCollection
{
    static int Main()
    {
        // First book
        int NumberOfPages1 = 422;
        // Second book
        int NumberOfPages2 = 858;
        // Third book
        int NumberOfPages3 = 422;

        if( NumberOfPages1.Equals(NumberOfPages2) == true )
            Console.WriteLine("The first and the second books have the same number of pages");
        else
            Console.WriteLine("The first and the second books have different number of pages");

        if( NumberOfPages1.Equals(NumberOfPages3) == true )
            Console.WriteLine("The first and the third books have the same number of pages");
        else
            Console.WriteLine("The first and the third books have different number of pages");
    }
}
```

This would produce:

```
The first and the second books have different number of pages
The first and the third books have the same number of pages
```

The first version of the **Object.Equals** method is declared as **virtual**, which means you can override it if you create your own class. The second version of the **Object.Equals()** method is:

```
public static bool Equals(object obj2, object obj2);
```

As a **static** method, to use it, you can pass the variables of the two classes whose values you want to compare.

In both cases, if the values of the variables are similar, the **Equals()** method returns true. If they are different, the method returns false. If you are using the **Equals()** method to compare the variables of two primitive types, the comparison should be straight forward. If you want to use this methods on

Coalesce

variables declared from your own class, you should provide your own implementation of this method.

Practical Learning: Implementing Equality

1. To create your own implementation of the **Equals()** method, change the file as follows:

```
using System;

class Sport
{
    private double _ballWeight;
    private int    _players;
    private double _courtLength;
    private double _courtWidth;

    public double BallWeight
    {
        get { return _ballWeight; }
        set { _ballWeight = value; }
    }

    public int NumberOfPlayers
    {
        get { return _players; }
        set { _players = value; }
    }

    public double CourtLength
    {
        get { return _courtLength; }
        set { _courtLength = value; }
    }

    public double CourtWidth
    {
        get { return _courtWidth; }
        set { _courtWidth = value; }
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }

    public override bool Equals(Object obj)
    {
        Sport sp = (Sport)obj;

        if( (_ballWeight == sp._ballWeight) &&
            (_players    == sp._players) &&
```

Coalesce

```
        (_courtLength == sp._courtLength) &&
        (_courtWidth == sp._courtWidth) )
    return true;

    return false;
}
}

class Exercise
{
    static int Main()
    {
        Sport Euro2002 = new Sport();
        Sport CAN2004 = new Sport();
        Sport tennis = new Sport();

        Euro2002.BallWeight = 435; // grams
        Euro2002.NumberOfPlayers = 11; // persons for each team
        Euro2002.CourtLength = 100; // meters
        Euro2002.CourtWidth = 60; // meters

        tennis.BallWeight = 57.50; // grams
        tennis.NumberOfPlayers = 1; // Singles game
        tennis.CourtLength = 23.70; // meters
        tennis.CourtWidth = 8.23; // meters;

        CAN2004.BallWeight = 435; // grams
        CAN2004.NumberOfPlayers = 11; // persons for each team
        CAN2004.CourtLength = 100; // meters
        CAN2004.CourtWidth = 60; // meters

        if( CAN2004.Equals(tennis) == true )
            Console.WriteLine("The CAN2004 and the tennis variables are equal");
        else
            Console.WriteLine("The Euro2002 and the tennis variables are not equal");

        if( Euro2002.Equals(CAN2004) == true )
            Console.WriteLine("The Euro2002 and CAN2004 variables are equal");
        else
            Console.WriteLine("The Euro2002 and CAN2004 variables are not equal");

        return 0;
    }
}
```

2. Save the file and switch to the Command Prompt
3. Compile and test the application. This would produce:

```
The Euro2002 and the tennis variables are not equal
The Euro2002 and CAN2004 variables are equal
```

Coalesce

4. Return to Notepad

Stringing a Class

In previous lessons, we learned that, to convert the value of a variable declared from a primitive type to a string, you could call the **ToString()** function. Here is an example:

```
using System;

class BookCollection
{
    static int Main()
    {
        int NumberOfPages = 422;

        Console.WriteLine("Number of Pages: {0}", NumberOfPages.ToString());
    }
}
```

In many programming languages such as C++, programmers usually have to overload an (extractor) operator to display the value(s) of class' variable to the screen. The **Object** class provides an alternative to this somewhat complicated solution, through the **ToString()** method. Its syntax is:

```
public virtual string ToString();
```

Although the **Object** class provides this method as non abstract, its implemented version is more useful if you use a primitive type such as int, double and their variances or a string variable. The best way to rely on it consists of overriding it in your own class if you desired to use its role.

Practical Learning: Converting to String

1. To implement and use a **ToString()** method, change the file as follows:

```
using System;

class Sport
{
    private double _ballWeight;
    private int _players;
    private double _courtLength;
    private double _courtWidth;

    public double BallWeight
    {
        get { return _ballWeight; }
        set { _ballWeight = value; }
    }

    public int NumberOfPlayers
    {
        get { return _players; }
        set { _players = value; }
    }
}
```

Coalesce

```
    }

    public double CourtLength
    {
        get { return _courtLength; }
        set { _courtLength = value; }
    }

    public double CourtWidth
    {
        get { return _courtWidth; }
        set { _courtWidth = value; }
    }

    public override int GetHashCode()
    {
        return base.GetHashCode();
    }

    public override bool Equals(Object obj)
    {
        Sport sp = (Sport)obj;

        if( (_ballWeight == sp._ballWeight) &&
            (_players == sp._players) &&
            (_courtLength == sp._courtLength) &&
            (_courtWidth == sp._courtWidth) )
            return true;

        return false;
    }

    public override string ToString()
    {
        string person = null;

        if( NumberOfPlayers.Equals(1) )
            person = " person";
        else
            person = " persons";

        string result = "\nBall Weight:      " + BallWeight + " grams" +
            "\nPlayers on each side: " + NumberOfPlayers + person +
            "\nCourt Dimensions(LxW): " +
            CourtLength + "m X " + CourtWidth + "m";

        return result;
    }
}

class Exercise
{
```

Coalesce

```
static int Main()
{
    Sport CAN2004 = new Sport();
    Sport tennis = new Sport();

    tennis.BallWeight = 57.50; // grams
    tennis.NumberOfPlayers = 1; // Singles game
    tennis.CourtLength = 23.70; // meters
    tennis.CourtWidth = 8.23; // meters

    CAN2004.BallWeight = 435; // grams
    CAN2004.NumberOfPlayers = 11; // persons for each team
    CAN2004.CourtLength = 100; // meters
    CAN2004.CourtWidth = 60; // meters

    Console.WriteLine("\nCup Game Characteristics");
    Console.WriteLine(CAN2004);

    Console.WriteLine();

    Console.WriteLine("\nTennis Game Characteristics");
    Console.WriteLine(tennis);

    return 0;
}
```

2. Save the file and switch to the Command Prompt
3. Compile and test the application. This would produce:

```
Cup Game Characteristics
Ball Weight:      435 grams
Players on each side: 11 persons
Court Dimensions(LxW): 100m X 60m
```

```
Tennis Game Characteristics
Ball Weight:      57.5 grams
Players on each side: 1 person
Court Dimensions(LxW): 23.7m X 8.23m
```

4. Return to Notepad

Finalizing a Variable

While a constructor, created for each class, is used to instantiate a class. The Object class provides the **Finalize()** method as a type of destructor.

Coalesce

Chapter 12

Introduction to Arrays

Introduction

Imagine you want to create a program that would use a series of numbers. In math, we represent such a series as follows: X_1, X_2, X_3, X_4, X_5 . In computer programming, as done in C#, it is better to consider such a list vertically as representing items of the same category. For example, if the series were made of names, you would illustrate it as follows:

Alex
Gaston
Hermine
Jerry

In C#, so far, to use a series of items, we were declaring a variable for each item. If the list was made of numbers, we would declare variables for such numbers as follows:

```
using System;
```

```
namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            double number1 = 12.44,
                number2 = 525.38,
                number3 = 6.28,
                number4 = 2448.32,
                number5 = 632.04;
        }
    }
}
```

Instead of using individual variables that share the same characteristics, you can group them in an entity like a regular variable but called an array. An array is a series of items of the same kind. It could be a group of numbers, a group of cars, a group of words, etc.

Practical Learning: Introducing Arrays


1. Start Notepad
2. In the empty file, type the following:

```
using System;
```

```
namespace CSharpLessons
```

Coalesce

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

3. To save the file, on the main menu, click File -> Save
4. Select the **CSharp Lessons** folder if you had created it in the first lesson. Otherwise select a folder of your choice or a drive letter and display it in the Save In combo box
5. Click the Create New Folder button  again
6. Type **Arrays1** and press Enter twice or display the new folder in the Save In combo box
7. Save the file as **exercise.cs** and click Save
8. To test the application, open the Command Prompt and change to the folder in which you created the C# file
9. Type **csc exercise.cs** and press Enter

Array Creation

Before creating an array, you must first decide the type its items will be made of. Is it a group of numbers, a group of chairs, a group of buttons on a remote controls? This information allows the compiler to know how much space each item of the group will require. This is because each item of the group will occupy its own memory space, just like any of the variables we have used so far.

After deciding about the type of data of the items that make up the series, you must use a common name to identify them. The name is simply the same type you would use for a variable as we have used so far. The name allows you and the compiler to identify the area in memory where the items are located.

Thirdly, you must specify the number of items that will constitute the group. For the compiler to be able to allocate an adequate amount of space for the items of the list, once it knows how much space each item will require, it needs to know the number of items so an appropriate and large enough amount of space can be reserved. The number of items of an array is included in square brackets, as in [5].

In C# (unlike some other languages like C/C++ or Pascal), an array is considered a reference type. Therefore, an array requests its memory using the **new** operator (like a pointer in C/C++). Based on this, the basic formula to declare an array is:

```
DataType[] VariableName = new DataType[Number];
```

In this formula, the *DataType* factor can be one of the types we have used so far. It can also be the name of a class. The square brackets on the left of the assignment operator are used to let the compiler know that you are declaring an array instead of a regular variable. The new operator allows the compiler to reserve memory in the heap. The *Number* factor is used to specify the number of items of the list.

Based on the above formula, here is an example of an array variable:

Coalesce

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            double[] number = new double[5];

            Console.WriteLine();
        }
    }
}
```

Practical Learning: Creating an Array

1. To create an array, declare the following variable:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            string[] deptStoreItemName = new string[5];

            Console.WriteLine();
        }
    }
}
```

2. Save the file

Using Arrays

Array Initialization

When creating an array, you can specify the number of items that make up its series. Each item of the series is referred to as a member of the array. Once the array has been created, each one of its members is initialized with a 0 value. Most, if not all, of the time, you will need to change the value of each member to a value of your choice. This is referred to as initializing the array.

An array is primarily a variable; it is simply meant to carry more than one value. Like every other variable, an array can be initialized. There are two main techniques you can use to initialize an array. If you have declared an array as done above, to initialize it, you can access each one of its members and assign it a desired but appropriate value.

In math, if you create a series of values as X_1 , X_2 , X_3 , X_4 , and X_5 , each member of this series can be identified by its subscript number. In this case the subscripts are 1, 2, 3, 4, and 5. This subscript

Coalesce

number is also called an index. In the case of an array also, each member of an array can be referred to by an incremental number called an index. A C# (like a C/C++) array is zero-based. This means that the first member of the series has an index of 0, the second has an index of 1. In math, the series would be represented as X_0 , X_1 , X_2 , X_3 , and X_4 . In C#, the index of a member of an array is written in its own square brackets. This is the notation you would use to locate each member. One of the actions you can take would consist of assigning it a value. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            double[] number = new double[5];

            number[0] = 12.44;
            number[1] = 525.38;
            number[2] = 6.28;
            number[3] = 2448.32;
            number[4] = 632.04;

            Console.WriteLine();
        }
    }
}
```

Besides this technique, you can also initialize the array as a whole when declaring it. To do this, on the right side of the declaration, before the closing semi-colon, type the values of the array members between curly brackets and separated by a comma. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            double[] number = new double[5]{ 12.44, 525.38, 6.28, 2448.32, 632.04 };

            Console.WriteLine();
        }
    }
}
```

If you use this second technique, you don't have to specify the number of items in the series. If you can leave all square brackets empty. If you do, the compiler will figure out the number of items.

Coalesce

Practical Learning: Using the Members of an Array

1. To initialize the array and use its members, change the program as follows:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            string[] deptStoreItemName = new string[5];

            deptStoreItemName[0] = "Women's Khaki Pants";
            deptStoreItemName[1] = "Diagonal-stripped Dress";
            deptStoreItemName[2] = "Teen Ultimate Cargo Short";
            deptStoreItemName[3] = "Women Swirl Slingback Shoes";
            deptStoreItemName[4] = "Athletic Long Bra";

            Console.WriteLine("\n - Department Store -");
            Console.WriteLine("Item 1: {0}", deptStoreItemName[0]);
            Console.WriteLine("Item 2: {0}", deptStoreItemName[1]);
            Console.WriteLine("Item 3: {0}", deptStoreItemName[2]);
            Console.WriteLine("Item 4: {0}", deptStoreItemName[3]);
            Console.WriteLine("Item 5: {0}", deptStoreItemName[4]);

            Console.WriteLine();
        }
    }
}
```

2. Save it and switch to the Command Prompt
3. Type **csc exercise.cs** to compile the program
4. After compiling, type exercise to execute

```
- Department Store -
Item 1: Women's Khaki Pants
Item 2: Diagonal-stripped Dress
Item 3: Teen Ultimate Cargo Short
Item 4: Women Swirl Slingback Shoes
Item 5: Athletic Long Bra
```

5. Return to Notepad

The Length of an Array

Once an array has been created, you can use it as you see fit. We saw that, when creating an array, you can specify the number of its members, and you must do this if you are not initializing the array. As we saw already, if you are initializing an array when creating it, you don't have to specify its dimension. Here is an example:

Coalesce

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            double[] number = new double[] { 12.44, 525.38, 6.28, 2448.32, 632.04,
                                              -378, 48, 6348, 762, 83, 64, -7, 86,
                                              534, -6, 386, 73, 5, -284, 3654, 671, 34, 693};

            Console.WriteLine();
        }
    }
}
```

Some time in your program, you may want to know how many members are in this type of array. You could get dizzy trying to count them and you may make a mistake with the count. All of the arrays used in C# derive from a class called **Array**. This class provides a property called **Length**. To find out how many members an array has, you can access its Length property. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            double[] number = new double[] { 12.44, 525.38, 6.28, 2448.32, 632.04,
                                              -378, 48, 6348, 762, 83, 64, -7, 86,
                                              534, -6, 386, 73, 5, -284, 3654, 671, 34, 693};

            Console.WriteLine("Number of items {0}", number.Length);

            Console.WriteLine();
        }
    }

    This would produce:

    Number of items 23

    Press any key to continue
```

Practical Learning: Using the Length of an Array

1. To use the length of an array, change the program as follows:

```
using System;
```

Coalesce

```
namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            string[] deptStoreItemName = new string[] { "Women's Khaki Pants",
                                                        "Diagonal-stripped Dress",
                                                        "Teen Ultimate Cargo Short",
                                                        "Women Swirl Slingback Shoes",
                                                        "Athletic Long Bra",
                                                        "Men's Trooper Cargo Short",
                                                        "Women's Hydro Training Shoes",
                                                        "Teen Strapless Dress" };

            Console.WriteLine("\n - Department Store -");
            for(int i = 0; i < deptStoreItemName.Length; i++)
                Console.WriteLine("Item {0}: {1}", i+1, deptStoreItemName[i]);

            Console.WriteLine();
        }
    }
}
```

2. Save it and switch to the Command Prompt
3. Type **csc exercise.cs** to compile the program
4. After compiling, type exercise to execute

```
- Department Store -
Item 1: Women's Khaki Pants
Item 2: Diagonal-stripped Dress
Item 3: Teen Ultimate Cargo Short
Item 4: Women Swirl Slingback Shoes
Item 5: Athletic Long Bra
Item 6: Men's Trooper Cargo Short
Item 7: Women's Hydro Training Shoes
Item 8: Teen Strapless Dress
```

5. Return to Notepad

Arrays of Arrays

A Two-Dimensional Array

The arrays we used in the above sections were made of a uniform series, where all members constituted a simple list, like a column of names on a piece of paper. Also, all items fit in one list. In some cases, you may want to divide the list in delimited sections. For example, if you create a list of names, you may want part of the list to include family members and another part of the list to include friends. Instead of creating a second list, you can add a second dimension to the list. In other

Coalesce

words, you would like to create a list of a list, or one list inside of another list, although the list is still made of items with common characteristics.

A multi-dimensional array is a series of lists so that each list contains its own list. For example, if you create two lists of names, you would have an array of two lists. Each array or list would have its own list or array. The most basic multi-dimensional array is an array of an array, also referred to as two-dimensional.

To create a two-dimensional array, declare the array variable as we did earlier but add a comma in the square brackets. The formula you would use is:

```
DataType[,] VariableName = new DataType[Number1,Number2];
```

The left pair of brackets remains empty but must contain a comma. There are two ways you can use the right square brackets. If you are declaring a two-dimensional array variable but are not ready to initialize it, type two integers separated by a comma in the right square brackets. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            String[,] members = new String[2,4];

            Console.WriteLine();
        }
    }
}
```

In this declaration, the members variable contains two lists. Each of the two lists contains 4 items. This means that the first list contains 4 items, the second list contains 4 items also. Therefore, the whole list is made of 8 items ($2 \times 4 = 8$). Because the variable is declared as String, each of the 8 items is a string.

To initialize a two-dimensional array, one option is to access each member of the array and assign it a value. The external list is zero-based. In other words, the first list has an index of 0, the second is 1, etc. Internally, each list is zero-based and behaves exactly like a one-dimensional array. To access a member of the list, type the name of the variable followed by its square brackets. In the brackets, type the index of the list, a comma, and the internal index of the member whose access you need. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            String[,] members = new String[2,4];

            members[0,0] = "Celeste";
        }
    }
}
```

Coalesce

```
members[0,1] = "Mathurin";
members[0,2] = "Alex";
members[0,3] = "Germain";
members[1,0] = "Jeremy";
members[1,1] = "Mathew";
members[1,2] = "Anselme";
members[1,3] = "Frederique";

Console.WriteLine();
}
}
```

As opposed to initializing an array by accessing each individual member, you can initialize an array when declaring it. To do this, on the right side of the declaration, before the closing semi-colon, type an opening and a closing curly bracket. Inside of the bracket, include a pair of an opening and a closing curly bracket for each external list of the array. Then, inside of a pair of curly brackets, provide a list of the values of the internal array, just as you would do for a one-dimensional array. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            String[,] members = new String[2,4]{
                {"Celeste", "Mathurin", "Alex", "Germain"},
                {"Jeremy", "Mathew", "Anselme",
"Frederique"}
            };

            Console.WriteLine();
        }
    }
}
```

If you use this technique to initialize an array, you can omit specifying the dimension of the array.

The Dimensions of an Array

We have seen that the square brackets are used to specify to the compiler that you are declaring an array. If you are creating a one-dimensional array, we saw that you can type a number in the square bracket. If you are creating a two-dimensional array, you type two numbers separated by a comma in the second pair of square brackets. Each number, whether it is one, two, or more is a place holder for what is referred to a dimension. In other words, a one dimensional array has a dimension of one. A two-dimensional array has a dimension of 2.

To find out the dimension of an array, the `Array` class, which is the parent of all arrays, provides the **Rank** property. Therefore, to know the dimension of an existing array, you can access its **Rank**

Coalesce

A Jagged Array

A jagged array is just an array of array or an array of arrays of arrays, etc. To create a jagged array, use the square brackets as we did for the two-dimensional array and include as many comma as you judge necessary to separate the dimensions. Here is an example of a three-dimensional array initialized:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
            double[, ] number = new double[2,3,5] { { { 12.44, 525.38, -6.28, 2448.32, 632.04},
                                                       {-378.05, 48.14, 634.18, 762.48, 83.02},
                                                       { 64.92, -7.44, 86.74, -534.60, 386.73}
            },
            { { 48.02, 120.44, 38.62, 526.82, 1704.62},
              { 56.85, 105.48, 363.31, 172.62,
128.48},
              { 906.68, 47.12, -166.07, 4444.26,
408.62} },
            };

            Console.WriteLine("Number of items {0}", number.Length);
            Console.WriteLine("Number of items {0}", number.Rank);

            Console.WriteLine();
        }
    }
}
```

In this example, we are creating 2 groups of items. Each of the two groups is made of three lists. Each list contains 5 numbers. This would produce:

Number of items 30

Number of items 3

Press any key to continue

To locate each member of the array, type the name of the array followed by the opening square bracket, followed by the 0-based first dimension, followed by a comma. Continue with each dimension and end with the closing square bracket. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Exercise
    {
        static void Main()
        {
```

Coalesce

```
double[,] number = new double[2,3,5] { { { 12.44, 525.38, -6.28, 2448.32, 632.04},
                                           {-378.05, 48.14, 634.18, 762.48, 83.02},
                                           { 64.92, -7.44, 86.74, -534.60, 386.73}
},
                                           { { 48.02, 120.44, 38.62, 526.82, 1704.62},
                                           { 56.85, 105.48, 363.31, 172.62,
128.48},
                                           { 906.68, 47.12, -166.07, 4444.26,
408.62} },
                                           };

Console.WriteLine("Number[0][0][0] = {0}", number[0,0,0]);
Console.WriteLine("Number[0][0][1] = {0}", number[0,0,1]);
Console.WriteLine("Number[0][0][2] = {0}", number[0,0,2]);
Console.WriteLine("Number[0][0][3] = {0}", number[0,0,3]);
Console.WriteLine("Number[0][0][4] = {0}\n", number[0,0,4]);

Console.WriteLine("Number[0][1][0] = {0}", number[0,1,0]);
Console.WriteLine("Number[0][1][1] = {0}", number[0,1,1]);
Console.WriteLine("Number[0][1][2] = {0}", number[0,1,2]);
Console.WriteLine("Number[0][1][3] = {0}", number[0,1,3]);
Console.WriteLine("Number[0][1][4] = {0}\n", number[0,1,4]);

Console.WriteLine("Number[0][2][0] = {0}", number[0,2,0]);
Console.WriteLine("Number[0][2][1] = {0}", number[0,2,1]);
Console.WriteLine("Number[0][2][2] = {0}", number[0,2,2]);
Console.WriteLine("Number[0][2][3] = {0}", number[0,2,3]);
Console.WriteLine("Number[0][2][4] = {0}\n", number[0,2,4]);

Console.WriteLine("Number[1][0][0] = {0}", number[1,0,0]);
Console.WriteLine("Number[1][0][1] = {0}", number[1,0,1]);
Console.WriteLine("Number[1][0][2] = {0}", number[1,0,2]);
Console.WriteLine("Number[1][0][3] = {0}", number[1,0,3]);
Console.WriteLine("Number[1][0][4] = {0}\n", number[1,0,4]);

Console.WriteLine("Number[1][1][0] = {0}", number[1,1,0]);
Console.WriteLine("Number[1][1][1] = {0}", number[1,1,1]);
Console.WriteLine("Number[1][1][2] = {0}", number[1,1,2]);
Console.WriteLine("Number[1][1][3] = {0}", number[1,1,3]);
Console.WriteLine("Number[1][1][4] = {0}\n", number[1,1,4]);

Console.WriteLine("Number[1][2][0] = {0}", number[1,2,0]);
Console.WriteLine("Number[1][2][1] = {0}", number[1,2,1]);
Console.WriteLine("Number[1][2][2] = {0}", number[1,2,2]);
Console.WriteLine("Number[1][2][3] = {0}", number[1,2,3]);
Console.WriteLine("Number[1][2][4] = {0}\n", number[1,2,4]);

Console.WriteLine("Number of items {0}", number.Length);
Console.WriteLine("Number of items {0}", number.Rank);

Console.WriteLine();
}
```


Coalesce

```
}
```

This would produce:

```
Number[0][0][0] = 12.44  
Number[0][0][1] = 525.38  
Number[0][0][2] = -6.28  
Number[0][0][3] = 2448.32  
Number[0][0][4] = 632.04
```

```
Number[0][1][0] = -378.05  
Number[0][1][1] = 48.14  
Number[0][1][2] = 634.18  
Number[0][1][3] = 762.48  
Number[0][1][4] = 83.02
```

```
Number[0][2][0] = 64.92  
Number[0][2][1] = -7.44  
Number[0][2][2] = 86.74  
Number[0][2][3] = -534.6  
Number[0][2][4] = 386.73
```

```
Number[1][0][0] = 48.02  
Number[1][0][1] = 120.44  
Number[1][0][2] = 38.62  
Number[1][0][3] = 526.82  
Number[1][0][4] = 1704.62
```

```
Number[1][1][0] = 56.85  
Number[1][1][1] = 105.48  
Number[1][1][2] = 363.31  
Number[1][1][3] = 172.62  
Number[1][1][4] = 128.48
```

```
Number[1][2][0] = 906.68  
Number[1][2][1] = 47.12  
Number[1][2][2] = -166.07  
Number[1][2][3] = 4444.26  
Number[1][2][4] = 408.62
```

```
Number of items 30  
Number of items 3
```

```
Press any key to continue
```

Coalesce

Arrays on Functions and Classes

A Primitive Array Passed as Argument

The main purpose of using an array is to use various pseudo-variables grouped under one name. Still, an array is primarily a variable. As such, it can be passed to a function and it can be returned from a function

Like a regular variable, an array can be passed as argument. To do this, in the parentheses of the function, provide the data type, the empty square brackets, and the name of the argument. Here is an example:

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
        }

        static void DisplayNumber(Double[] n)
        {
        }
    }
}
```

When an array has been passed to a function, it can be used in the body of the function as an array can be, following the rules of array variables. For example, you can display its values. Because an array is derived from the Array class of the System namespace, an array passed as argument carries its number of members. This means that you don't have to pass an additional argument, as done in C++.

The simplest way you can use an array is to display the values of its members. Here is an example:

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
        }

        static void DisplayNumber(Double[] n)
        {
            for (int i = 0; i < n.Length; i++)
                Console.WriteLine("Number {0}: {1}", i, n[i]);
        }
    }
}
```

Coalesce

```
}
```

To call a function that takes an array as argument, simply type the name of the argument in the parentheses of the called function. Here is an example:

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            Double[] number = new Double[5];

            number[0] = 12.44;
            number[1] = 525.38;
            number[2] = 6.28;
            number[3] = 2448.32;
            number[4] = 632.04;

            DisplayNumber(number);
            Console.ReadLine();
        }

        static void DisplayNumber(Double[] n)
        {
            for (int i = 0; i < n.Length; i++)
                Console.WriteLine("Number {0}: {1}", i, n[i]);
        }
    }
}
```

This would produce:

```
Number 0: 12.44
Number 1: 525.38
Number 2: 6.28
Number 3: 2448.32
Number 4: 632.04
```

When an array is passed as argument to a function, the array is passed by reference. This means that, if the function makes any change to the array, the change would be kept when the function returns. You can use this characteristic to initialize an array from a function. Here is an example:

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
```

Coalesce

```
Double[] number = new Double[5];

    InitializeList(number);
    DisplayNumber(number);
    Console.ReadLine();
}

static void InitializeList(Double[] nbr)
{
    nbr[0] = 12.44;
    nbr[1] = 525.38;
    nbr[2] = 6.28;
    nbr[3] = 2448.32;
    nbr[4] = 632.04;
}

static void DisplayNumber(Double[] n)
{
    for (int i = 0; i < n.Length; i++)
        Console.WriteLine("Number {0}: {1}", i, n[i]);
}
}
```

Notice that the `InitializeList()` function receives an un-initialized array but returns it with new values.

To enforce the concept of passing a variable by reference, you can also accompany an array argument with the `ref` keyword, both when defining the function and when calling it. Here is an example:

```
#region Using directives

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

#endregion

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Double[] number = new Double[5];

            InitializeList(ref number);
            DisplayNumber(ref number);
            Console.ReadLine();
        }
        static void InitializeList(ref Double[] nbr)
        {
```

Coalesce

```
        nbr[0] = 12.44;
        nbr[1] = 525.38;
        nbr[2] = 6.28;
        nbr[3] = 2448.32;
        nbr[4] = 632.04;
    }

    static void DisplayNumber(ref Double[] n)
    {
        for (int i = 0; i < n.Length; i++)
            Console.WriteLine("Number {0}: {1}", i, n[i]);
    }
}
```

Returning a Primitive Array From a Function

Like a normal variable, an array can be returned from a function. This means that the function would return a variable that carries various values. When declaring or defining the function, you must specify its data type. When the function ends, it would return an array represented by the name of its variable.

To declare a function that returns an array, on the left of the function name, provide the type of value that the returned array will be made of, followed by empty square brackets. Here is an example:

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            Double[] number = new Double[5];

            DisplayNumber(number);

            static Double[] InitializeList()
            {
            }

            static void DisplayNumber(Double[] n)
            {
                for (int i = 0; i < n.Length; i++)
                    Console.WriteLine("Number {0}: {1}", i, n[i]);
            }
        }
    }
}
```

Coalesce

Remember that a function must always return an appropriate value depending on how it was declared. In this case, if it was specified as returning an array, then make sure it returns an array and not a regular variable. One way you can do this is to declare and possibly initialize a local array variable. After using the local array, you return only its name (without the square brackets). Here is an example:

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            Double[] number = new Double[5];

            DisplayNumber(number);
            Console.ReadLine();
        }

        static Double[] InitializeList()
        {
            Double[] nbr = new Double[5];

            nbr[0] = 12.44;
            nbr[1] = 525.38;
            nbr[2] = 6.28;
            nbr[3] = 2448.32;
            nbr[4] = 632.04;

            return nbr;
        }

        static void DisplayNumber(Double[] n)
        {
            for (int i = 0; i < n.Length; i++)
                Console.WriteLine("Number {0}: {1}", i, n[i]);
        }
    }
}
```

When a function returns an array, that function can be assigned to an array declared locally when you want to use it. Remember to initialize such a function only with an array variable. If you initialize it with a regular variable, you would receive an error.

Here is an example:

```
#region Using directives

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

Coalesce

```
using System.Text;

#endregion

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Double[] number = new Double[5];

            number = InitializeList();
            DisplayNumber(number);

            Console.ReadLine();
        }

        static Double[] InitializeList()
        {
            Double[] nbr = new Double[5];

            nbr[0] = 12.44;
            nbr[1] = 525.38;
            nbr[2] = 6.28;
            nbr[3] = 2448.32;
            nbr[4] = 632.04;

            return nbr;
        }

        static void DisplayNumber(Double[] n)
        {
            Console.WriteLine("List of Numbers");
            for (int i = 0; i < n.Length; i++)
                Console.WriteLine("Number {0}: {1}", i, n[i]);
        }
    }
}
```

Arrays and Classes

Introduction

Some languages, such as C++, possess the notion of global variable, C# doesn't. In C#, every variable, array, or method can only be declared inside of a class. This is also valid for the **Main()** function that must also be created inside of a class. This theory of C# can be applied to create an illusion of a global variable. Therefore, if you create a program that has only one class, you can declare an array somewhere in the body of that class and such an array becomes accessible to all other members of the same class.

Coalesce

Practical Learning: Introducing Arrays and Classes

1. Start Notepad
2. In the empty file, type the following:

```
using System;

namespace DepartmentStore
{
    class StoreItem
    {
        public long ItemNumber;
        public string ItemName;
        public double UnitPrice;
    }
}
```

3. Save the file in a new folder named **DeptStore3**
4. Save the file itself as **item.cs**
5. Without closing Notepad, open another instance of Notepad (start "another" Notepad) and type the following in it:

```
using System;

namespace DepartmentStore
{
    class Sales
    {
        static void Main()
        {
        }
    }
}
```

6. Save the new file as **exercise.cs** in the same DeptStore3 folder

A Class as an Array Member Variable

Besides a variable, a class can be declared as an array member variable of another class. To do this, declare the array as we have used them so far but make it static. For example, imagine you have created a class as follows:

```
using System;

namespace CSharpLessons
{
    public class Square
    {
        private double _side;
```


Coalesce

```
        public Square()
        {
            _side = 0;
        }
        public double Side
        {
            get
            {
                return _side;
            }
            set
            {
                _side = value;
            }
        }
        public Double Perimeter
        {
            get
            {
                return _side * 4;
            }
        }
        public double Area
        {
            get
            {
                return _side * _side;
            }
        }
    }
}
```

Here is how you can declare an array variable from the above class:

```
using System;
```

```
namespace CSharpLessons
{
    class Class1
    {
        static Square[] Sq = new Square[2];

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

After declaring the array, you can initialize it using the **Main()** function. You should first allocate memory using the new operator for each member of the array. Each member of the array is

Coalesce

accessed using its index. Once the array exists, you can access each member of the class indirectly from the index of the variable. Here is an example:

```
using System;

namespace CSharpLessons
{
    class Class1
    {
        static Square[] Sq = new Square[2];

        static void Main(string[] args)
        {
            Sq[0] = new Square();
            Sq[0].Side = 18.84;

            Sq[1] = new Square();
            Sq[1].Side = 25.62;

            Console.WriteLine("Squares Characteristics");
            Console.WriteLine("\nFirst Square");
            Console.WriteLine("Side:    {0}", Sq[0].Side);
            Console.WriteLine("Perimeter: {0}", Sq[0].Perimeter);
            Console.WriteLine("Area:     {0}", Sq[0].Area);

            Console.WriteLine("\nSecond Square");
            Console.WriteLine("Side:    {0}", Sq[1].Side);
            Console.WriteLine("Perimeter: {0}", Sq[1].Perimeter);
            Console.WriteLine("Area:     {0}", Sq[1].Area);
        }
    }
}
```

This would produce:

Squares Characteristics

First Square

Side: 18.84

Perimeter: 75.36

Area: 354.9456

Second Square

Side: 25.62

Perimeter: 102.48

Area: 656.3844

Practical Learning: Declaring an Array Variable of a Class

Coalesce

1. To declare and use an array of StoreItem variables, change the contents of the **Main()** function of the exercise.cs as follows:

```
using System;

namespace DepartmentStore
{
    class Sales
    {
        static void Main()
        {
            StoreItem[] anItem = new StoreItem[5];

            anItem[0] = new StoreItem();
            anItem[0].ItemNumber = 79576;
            anItem[0].ItemName = "Women Skirt";
            anItem[0].UnitPrice = 34.55;

            anItem[1] = new StoreItem();
            anItem[1].ItemNumber = 305417;
            anItem[1].ItemName = "60\" Gold Chain";
            anItem[1].UnitPrice = 224.95;

            anItem[2] = new StoreItem();
            anItem[2].ItemNumber = 58257;
            anItem[2].ItemName = "Men Khaki Pants";
            anItem[2].UnitPrice = 32.90;

            anItem[3] = new StoreItem();
            anItem[3].ItemNumber = 84026;
            anItem[3].ItemName = "Children Summer Hat";
            anItem[3].UnitPrice = 18.75;

            anItem[4] = new StoreItem();
            anItem[4].ItemNumber = 44285;
            anItem[4].ItemName = "Women Croco-Belt";
            anItem[4].UnitPrice = 22.45;

            Console.WriteLine("Store Inventory");
            Console.WriteLine("Index: 1");
            Console.WriteLine("Item Stock #: {0}", anItem[0].ItemNumber);
            Console.WriteLine("Description: {0}", anItem[0].ItemName);
            Console.WriteLine("Unit Price: {0:C}\n", anItem[0].UnitPrice);

            Console.WriteLine("Index: 2");
            Console.WriteLine("Item Stock #: {0}", anItem[1].ItemNumber);
            Console.WriteLine("Description: {0}", anItem[1].ItemName);
            Console.WriteLine("Unit Price: {0:C}\n", anItem[1].UnitPrice);

            Console.WriteLine("Index: 3");
            Console.WriteLine("Item Stock #: {0}", anItem[2].ItemNumber);
```

Coalesce

```
        Console.WriteLine("Description: {0}", anItem[2].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[2].UnitPrice);

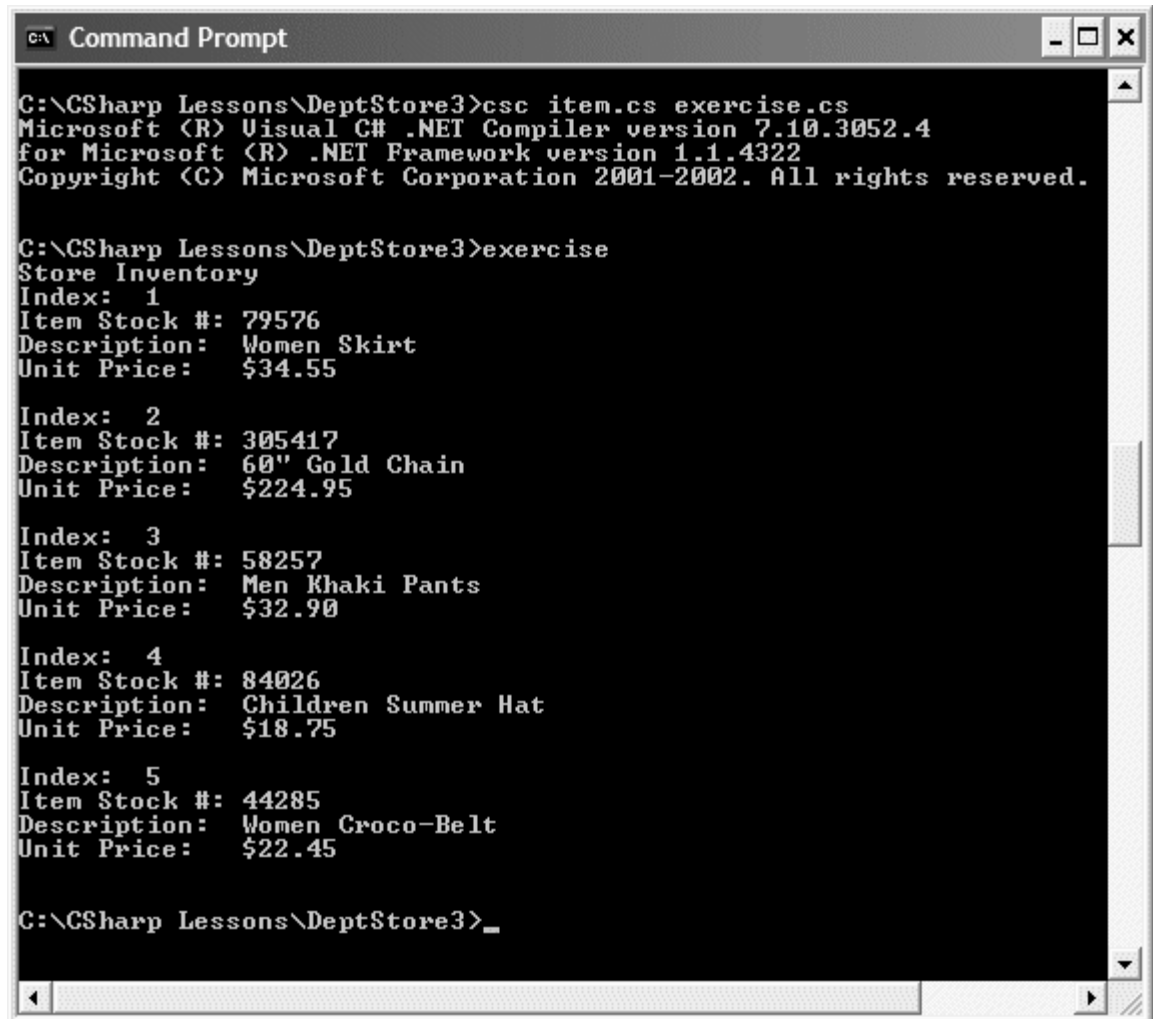
        Console.WriteLine("Index: 4");
        Console.WriteLine("Item Stock #: {0}", anItem[3].ItemNumber);
        Console.WriteLine("Description: {0}", anItem[3].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[3].UnitPrice);

        Console.WriteLine("Index: 5");
        Console.WriteLine("Item Stock #: {0}", anItem[4].ItemNumber);
        Console.WriteLine("Description: {0}", anItem[4].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[4].UnitPrice);
    }
}
```

2. Save the exercise.cs file
3. Open the Command Prompt and switch to the folder created for the current project
4. Compile it by typing `csc item.cs exercise.cs`

Coalesce

5. Execute it by typing exercise



```
C:\CSharp Lessons\DeptStore3>csc item.cs exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

C:\CSharp Lessons\DeptStore3>exercise
Store Inventory
Index: 1
Item Stock #: 79576
Description: Women Skirt
Unit Price: $34.55

Index: 2
Item Stock #: 305417
Description: 60" Gold Chain
Unit Price: $224.95

Index: 3
Item Stock #: 58257
Description: Men Khaki Pants
Unit Price: $32.90

Index: 4
Item Stock #: 84026
Description: Children Summer Hat
Unit Price: $18.75

Index: 5
Item Stock #: 44285
Description: Women Croco-Belt
Unit Price: $22.45

C:\CSharp Lessons\DeptStore3>_
```

A Class Array Passed as Argument

Like a primitive type, a class can be passed to a function as argument. Everything is primarily done the same except that, since the argument is an array, each member of the array is a variable in its own right, with a value for each member variable of the class. Based on this, you when treating the argument, you can access each of its members individually using its index.

Practical Learning: Passing a Class as Argument

1. Access the exercise.cs file
To pass an array of a class to a function as argument, change the file as follows:

using System;

namespace DepartmentStore

Coalesce

```
{
class Sales
{
    static void Main()
    {
        StoreItem[] itmStore = new StoreItem[5];

        CreateInventory(itmStore);
        ShowInventory(ref itmStore);
    }

    static void CreateInventory(StoreItem[] anItem)
    {
        anItem[0] = new StoreItem();
        anItem[0].ItemNumber = 79576;
        anItem[0].ItemName = "Women Skirt";
        anItem[0].UnitPrice = 34.55;

        anItem[1] = new StoreItem();
        anItem[1].ItemNumber = 305417;
        anItem[1].ItemName = "60\" Gold Chain";
        anItem[1].UnitPrice = 224.95;

        anItem[2] = new StoreItem();
        anItem[2].ItemNumber = 58257;
        anItem[2].ItemName = "Men Khaki Pants";
        anItem[2].UnitPrice = 32.90;

        anItem[3] = new StoreItem();
        anItem[3].ItemNumber = 84026;
        anItem[3].ItemName = "Children Summer Hat";
        anItem[3].UnitPrice = 18.75;

        anItem[4] = new StoreItem();
        anItem[4].ItemNumber = 44285;
        anItem[4].ItemName = "Women Croco-Belt";
        anItem[4].UnitPrice = 22.45;
    }

    static void ShowInventory(ref StoreItem[] anItem)
    {
        Console.WriteLine("Store Inventory");
        Console.WriteLine("Index: 1");
        Console.WriteLine("Item Stock #: {0}", anItem[0].ItemNumber);
        Console.WriteLine("Description: {0}", anItem[0].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[0].UnitPrice);

        Console.WriteLine("Index: 2");
        Console.WriteLine("Item Stock #: {0}", anItem[1].ItemNumber);
        Console.WriteLine("Description: {0}", anItem[1].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[1].UnitPrice);
    }
}
```

Coalesce

```
        Console.WriteLine("Index: 3");
        Console.WriteLine("Item Stock #: {0}", anItem[2].ItemNumber);
        Console.WriteLine("Description: {0}", anItem[2].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[2].UnitPrice);

        Console.WriteLine("Index: 4");
        Console.WriteLine("Item Stock #: {0}", anItem[3].ItemNumber);
        Console.WriteLine("Description: {0}", anItem[3].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[3].UnitPrice);

        Console.WriteLine("Index: 5");
        Console.WriteLine("Item Stock #: {0}", anItem[4].ItemNumber);
        Console.WriteLine("Description: {0}", anItem[4].ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem[4].UnitPrice);
    }
}
```

2. Compile the program with **csc item.cs exercise.cs**
3. Execute it with exercise

Returning a Class Array From a Function

Once again, like a regular variable, an array can be returned from a function. The syntax also follows that of a primitive type. When the function ends, you must make sure it returns an array and not a regular variable.

Practical Learning: Return a Class Array From a Function

1. To return an array from a function, change the file as follows:

```
using System;

namespace DepartmentStore
{
    class Sales
    {
        static void Main()
        {
            StoreItem[] itmStore = new StoreItem[5];

            itmStore = CreateInventory();
            ShowInventory(ref itmStore);
        }

        static StoreItem[] CreateInventory()
        {
            StoreItem[] NewInventory = new StoreItem[5];

            NewInventory[0] = new StoreItem();
```

Coalesce

```
NewInventory[0].ItemNumber = 79576;
NewInventory[0].ItemName  = "Women Skirt";
NewInventory[0].UnitPrice = 34.55;

NewInventory[1] = new StoreItem();
NewInventory[1].ItemNumber = 305417;
NewInventory[1].ItemName  = "60\" Gold Chain";
NewInventory[1].UnitPrice = 224.95;

NewInventory[2] = new StoreItem();
NewInventory[2].ItemNumber = 58257;
NewInventory[2].ItemName  = "Men Khaki Pants";
NewInventory[2].UnitPrice = 32.90;

NewInventory[3] = new StoreItem();
NewInventory[3].ItemNumber = 84026;
NewInventory[3].ItemName  = "Children Summer Hat";
NewInventory[3].UnitPrice = 18.75;

NewInventory[4] = new StoreItem();
NewInventory[4].ItemNumber = 44285;
NewInventory[4].ItemName  = "Women Croco-Belt";
NewInventory[4].UnitPrice = 22.45;

return NewInventory;
}

static void ShowInventory(ref StoreItem[] curInventory)
{
Console.WriteLine("Store Inventory");
Console.WriteLine("Index: 1");
Console.WriteLine("Item Stock #: {0}", curInventory[0].ItemNumber);
Console.WriteLine("Description: {0}", curInventory[0].ItemName);
Console.WriteLine("Unit Price: {0:C}\n", curInventory[0].UnitPrice);

Console.WriteLine("Index: 2");
Console.WriteLine("Item Stock #: {0}", curInventory[1].ItemNumber);
Console.WriteLine("Description: {0}", curInventory[1].ItemName);
Console.WriteLine("Unit Price: {0:C}\n", curInventory[1].UnitPrice);

Console.WriteLine("Index: 3");
Console.WriteLine("Item Stock #: {0}", curInventory[2].ItemNumber);
Console.WriteLine("Description: {0}", curInventory[2].ItemName);
Console.WriteLine("Unit Price: {0:C}\n", curInventory[2].UnitPrice);

Console.WriteLine("Index: 4");
Console.WriteLine("Item Stock #: {0}", curInventory[3].ItemNumber);
Console.WriteLine("Description: {0}", curInventory[3].ItemName);
Console.WriteLine("Unit Price: {0:C}\n", curInventory[3].UnitPrice);

Console.WriteLine("Index: 5");
Console.WriteLine("Item Stock #: {0}", curInventory[4].ItemNumber);
```


Coalesce

```
        Console.WriteLine("Description: {0}", curInventory[4].ItemName);  
        Console.WriteLine("Unit Price: {0:C}\n", curInventory[4].UnitPrice);  
    }  
}  
}
```

2. Compile and execute the application

Coalesce

Chapter 13

The Command Line

Command Line Fundamentals

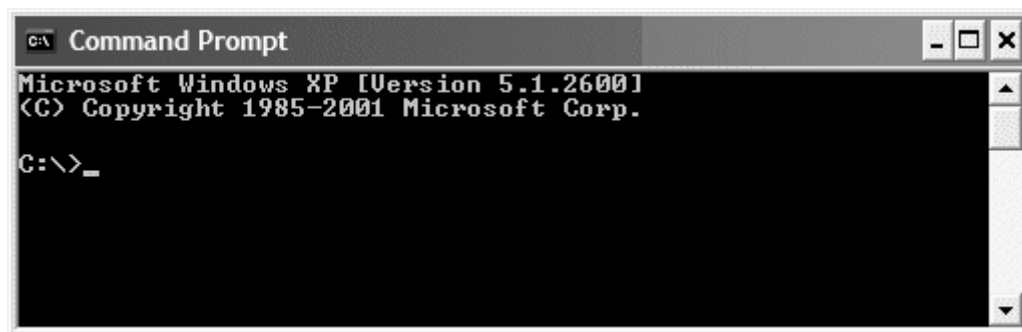
Introduction

When the computer starts, there is no primary visual or graphical effect. The programs must first be loaded to memory. In fact, in the beginning, the Microsoft operating system started as text-based with DOS 2.2, 3.3, 5.0, and 6.2 (6.2 was never a big hit like 5.0 because Windows 3.1, Windows 3.2, and Windows 3.3 was released at the same time, bringing the now attractive "Windows" display). To create a program, a programmer entered the source code using a text editor (there was a text editor called EDIT; honestly it was very cute, very basic, and very wonderful; in fact, up to Windows 3.3, it was still the text editor of choice of most programmers and it was used as the editor for QBasic, the parent of Visual Basic). After creating the source file, the programmer had to use a black (or "amber" screen that displayed a blinking caret whose role was to let the user that it was waiting for a command. A command was a word or a group of words that the programmer typed and pressed Enter (or Return). This caused the computer to produce an action. The command was type on a line. For this reason, the area that displayed the was called the Command Line. Most low-level (low-level doesn't mean weak, to the contrary, it may even mean complicated; have you ever heard of Assembly?) programs are still using this concept. This means that, behind the scenes, computer languages such as C/C++, Pascal, Java, C#, etc, are still created and executed at if everything were done at the Command Line. In fact, if you use Visual Studio .NET to create and compile your program, everything you see is done at the level of Command Line and transported to the front so you can work "visually".

Command Line Understanding

To be an effective programmer, it is important to know how the Command Line works because, as mentioned already, C# programmers are compiled at the Command Prompt. If you know how the Command Line works, you can have a better understanding of things that are going on behind the scenes.

As its name indicates, the Command Line is a blank line with a blinking caret that expects you to type a command:



Coalesce

There are various types of commands used on the command prompt. Some of them are as old as the operating system. Such is the case for commands used to create files or folders, copy files or folders, delete files or folders, format a drive, etc. Some other commands are created by a person like you who needs it for a particular application. For example, the C# language provides a command called **csc** to compile a program at the command line, as we have done so far. In the same way, when you create a program, particularly when you create a source file, you can use some options that would be used at the command prompt.

The Main() Function

Introduction

When a program starts, it looks for an entry point as the area of entrance. This is the role of the `Main()` function. In fact, a program, that is an executable programs, starts by, and stops with, the `Main()` function. The way this works is that, at the beginning, the compiler enters he `Main` function in a top-down approach, starting just after the opening curly bracket. If it finds a problem and judges that it is not worth continuing, it stops and lets you know. If or as long as it doesn't find a problem, it continues line after line, with the option to even call or execute a function in the same file or in another file. This process continues to the closing curly bracket `}"`. Once the compiler finds the closing bracket, the whole program has ended and stops.

If you want the user to provide additional information when executing your program, you can take care of this in the **Main()** function as this is the entry point of your program.

Practical Learning: Introducing the Command Line

1. Start Notepad
2. In the empty file, type the following:

```
using System;


namespace CSharpLessons
{
    class Exercise
    {
        static int Main()
        {
            string FirstName = "James";
            string LastName = "Weinberg";
            Double WeeklyHours = 36.50;
            Double HourlySalary = 12.58;

            string FullName = LastName + ", " + FirstName;
            Double WeeklySalary = WeeklyHours * HourlySalary;

            Console.WriteLine("Employee Payroll");
            Console.WriteLine("Full Name: {0}", FullName);
            Console.WriteLine("WeeklySalary: {0}", WeeklySalary.ToString("C"));
        }
    }
}
```

Coalesce

```
        return 0;
    }
}
```

3. To save the file, on the main menu, click File -> Save
4. Select the **CSharp Lessons** folder if you had created it in the first lesson
5. Click the Create New Folder button  again
6. Type **CommandLine1** and press Enter twice or display the new folder in the Save In combo box
7. Save the file as **Exercise.cs** and click Save
8. To test the application, open the Command Prompt and change to the folder in which you created the C# file
9. Type **csc Exercise.cs** and press Enter
10. To execute the program, type the name **Exercise** and press Enter. This would produce:

```
C:\CSharp Lessons\CommandLine1>csc Exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.
```

```
C:\CSharp Lessons\CommandLine1>Exercise
Employee Payroll
Full Name: Weinberg, James
WeeklySalary: $459.17
```

```
C:\CSharp Lessons\CommandLine1>
```

11. Return to your text editor

Command Request from Main()

So far, to compile a program, we simply typed the **csc** command at the command prompt. Then, to execute a program, we simply typed its name at the prompt. If we distributed one of these programs, we would simply tell the user to type the name of the program at the command prompt. In some cases, you may want the user to type additional information besides the name of the program. To request additional information from the user, pass a string argument to the **Main()** function. The argument should be passed as an array and make sure you provide a name for the argument. Here is an example:

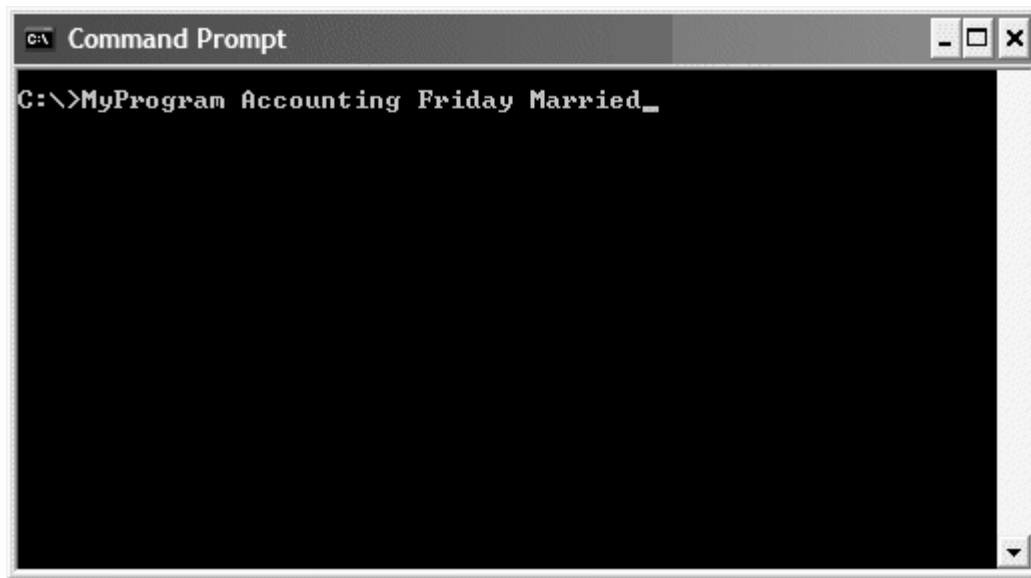
```
using System;
```

```
class ObjectName
{
    static int Main(string[] args)
    {
        return 0;
    }
}
```

Coalesce

```
}  
}
```

The reason you pass the argument as an array is so you can use as many values as you judge necessary. To provide values at the command prompt, the user types the name of the program followed by each necessary value. Here is an example:



The values the user would provide are stored in the zero-based argument array without considering the name of the program. The first value (that is, after the name of the program) is stored at index 0, the second at index 1, etc. Based on this, the first argument is represented by `args[0]`, the second is represented by `args[1]`, etc.

Since the array argument (like all C# arrays) is based on the **Array** class of the **System** namespace, if you want to find out how many values the user supplied, you can call the **Array.Length** property.

Each of the values the user types is a string. If any one of them is not a string, you should/must convert its string first to the appropriate value.

Practical Learning: Passing an Argument to `Main()`

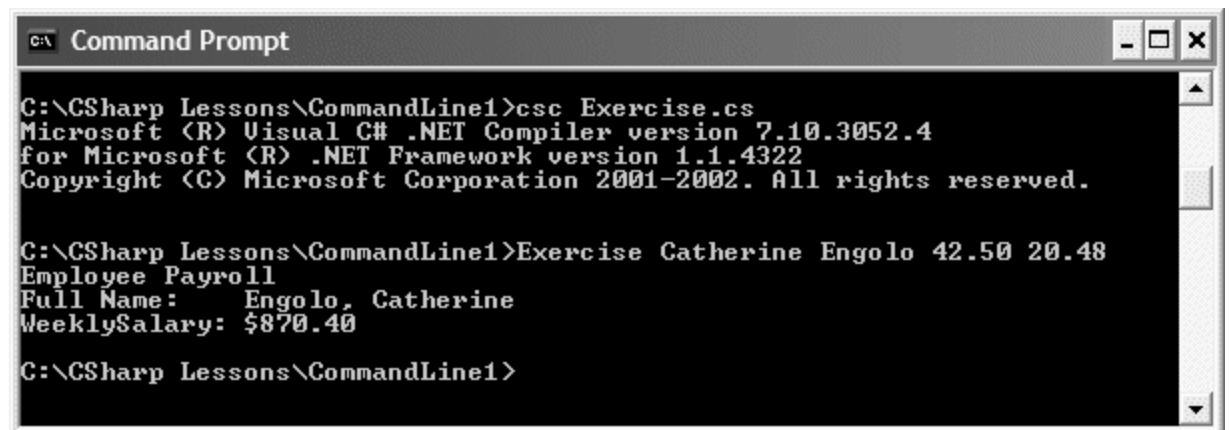
1. To pass an argument to **Main()**, change the method as follows:

```
using System;  
  
namespace CSharpLessons  
{  
    class Exercise  
    {  
        static int Main(string[] Argument)  
        {  
            string FirstName;  
            string LastName;
```

Coalesce

```
    Double WeeklyHours;  
    Double HourlySalary;  
  
    FirstName = Argument[0];  
    LastName = Argument[1];  
    WeeklyHours = Double.Parse(Argument[2]);  
    HourlySalary = Double.Parse(Argument[3]);  
  
    string FullName = LastName + ", " + FirstName;  
    Double WeeklySalary = WeeklyHours * HourlySalary;  
  
    Console.WriteLine("Employee Payroll");  
    Console.WriteLine("Full Name: {0}", FullName);  
    Console.WriteLine("WeeklySalary: {0}", WeeklySalary.ToString("C"));  
  
    return 0;  
    }  
}
```

2. Save the file and switch to the command prompt
3. To compile the application, type **csc Exercise.cs** and press Enter
4. To execute the program, type **Exercise** followed by a first name, a last name, and two decimal values. An example would be **Exercise Catherine Engolo 42.50 20.48**



```
C:\> Command Prompt

C:\CSharp Lessons\CommandLine1>csc Exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

C:\CSharp Lessons\CommandLine1>Exercise Catherine Engolo 42.50 20.48
Employee Payroll
Full Name:      Engolo, Catherine
WeeklySalary:  $870.40

C:\CSharp Lessons\CommandLine1>
```

5. Return to your text editor

Coalesce

Chapter 14

Custom Libraries

Introduction

A library is a program that contains classes and/or methods that other programs can use. Such a program is created with the same approach as the programs we have used so far. Because a library is not an executable, it doesn't need the **Main()** function. A library can have the extension .dll or .lib or other.

Creating a Library

A library can be made of a single file or as many files as necessary. A file that is part of a library can contain one or more classes. Each class should implement a behavior that can eventually be useful and accessible to other classes. The classes in a library are created exactly like those we have used so far.

To create a library, start with a normal C# file with a .cs extension. To get the library, you must compile the file(s) but inform the compiler that you want to get a library rather than a regular executable application. To compile the file(s), you would use the following formula:

```
csc /target:library File.cs
```

As always, you start by invoking the C# compiler. To indicate that you are creating a library, you include the **/target:library** expression. As normally one, you must specify the file that contains the code that needs to be executed. Here is an example:

```
csc /target:library Mine.cs
```

In this case, the compiler would create a library called Mine.cs. If you have many files to put into the library, when compiling, type the name of each file at the end of the compilation line. Here is an example:

```
csc /target:library Mine.cs, Yours.cs Everyone.cs
```

In this case, a library with the name of the first file would be created. You can specify the name you want the library to have, instead of using the same name as the file or the name of the first file in the group. The formula you would use is:

```
csc /target:library /out:DesiredName.dll File.cs
```

The compiler would create a library with the *DesiredName*. If you have more than one file to involve in the library, type them at the end of the compilation line. Here is an example:

```
csc /target:library /out:Great.dll Mine.cs, Yours.cs Everyone.cs
```

Coalesce

Using a Library

After creating a library, you can use its contents in any code of your choice. The most important thing is to know what is in a library you want to use. When writing your code, you can use any class that is part of the library. When compiling your application, you must remember to add a reference to the library. You would use the following formula:

```
csc /reference:LibraryName.dll File.cs
```

Here is an example:

```
csc /reference:Great.dll Exercise.cs
```


Chapter 15

Lists and Collections

Introduction to Collections

Like an array, a collection is a series of items of the same type. The problem with an array is that you must know in advance the number of items that will make up the list. There are cases you don't know, you can't know, or you can't predict the number of items of the list. The solution is to create a linked list. A linked list is a list in which you don't specify the maximum number of items but you allow the user of the list to add, locate, or remove items at will. Traditionally, it has never been easy to create a linked list, especially for beginning programmers. Aware of this, many programming languages ship with one or more libraries that can be used directly to create a list of almost any kind.

The Microsoft .NET Framework provides various classes that can be used to create different types of lists. Most of the classes are easily to use once you get used to them. The idea is to know what classes are available, what each class does, and what class to use for a particular assignment

Practical Learning: Introducing Collections

1. Start Notepad
2. In the empty file, type the following:

```
using System;

namespace BusinessApplication
{
    public class ConvenienceStore
    {
        public long  StockNumber;
        public string Description;
        public double UnitPrice;

        public ConvenienceStore()
        {
            this.StockNumber = 0;
            this.Description = "N/A";
            this.UnitPrice  = 0.00;
        }

        public ConvenienceStore(long nbr, string name, double price)
        {
            this.StockNumber = nbr;
            this.Description = name;
            this.UnitPrice  = price;
        }
    }
}
```

Coalesce

3. Save the file in a new folder named **ConvStore1**
4. Save the file itself as **StoreItem.cs**
5. Without closing Notepad, open another instance of Notepad (start "another" Notepad) and type the following in it:

```
using System;

namespace BusinessApplication
{
    class Exercise
    {
        static void Main()
        {
            ConvenienceStore[] store = new ConvenienceStore[4];

            CreateStoreStock(store);

            Console.WriteLine();
        }

        static void CreateStoreStock(ConvenienceStore[] cnvStore)
        {
            cnvStore[0] = new ConvenienceStore();
            cnvStore[0].StockNumber = 42576;
            cnvStore[0].Description = "Soda 2-Litter";
            cnvStore[0].UnitPrice = 1.45;

            cnvStore[1] = new ConvenienceStore(64836, "150-Sheet Notebook", 1.75);

            cnvStore[2] = new ConvenienceStore();
            cnvStore[2].StockNumber = 83570;
            cnvStore[2].Description = "Rubber Transparent Tape";
            cnvStore[2].UnitPrice = 3.15;

            cnvStore[3] = new ConvenienceStore(48631, "Multicolor Pencils", 1.95);
            cnvStore[3] = new ConvenienceStore(28460, "Ball Point Pens", 2.25);
        }
    }
}
```

6. Save the new file as **exercise.cs** in the same ConvStore1 folder

Foreach Item in the Collection

In the previous lessons, we used the traditional techniques of locating each item in an array. Here is an example:

```
class Exercise
{
    static int Main()
    {
```

Coalesce

```
string[] EmployeeName = new string[4];

EmployeeName[0] = "Joan Fuller";
EmployeeName[1] = "Barbara Boxen";
EmployeeName[2] = "Paul Kumar";
EmployeeName[3] = "Bertrand Entire";

Console.WriteLine("Employees Records");
Console.WriteLine("Employee 1: {0}", EmployeeName[0]);
Console.WriteLine("Employee 2: {0}", EmployeeName[1]);
Console.WriteLine("Employee 3: {0}", EmployeeName[2]);
Console.WriteLine("Employee 4: {0}", EmployeeName[3]);

return 0;
}
}
```

To scan a list, the C# language provides two keyword, **foreach** and **in**, with the following syntax:

foreach (type identifier in expression) statement

The **foreach** and the **in** keywords are required.

The first factor of this syntax, *type*, must be a data type (**int**, **short**, **Byte**, **Double**, etc) or the name of a class that either exists in the language or that you have created.

The *identifier* factor is a name you specify to represent an item of the collection.

The *expression* factor is the name of the array or collection.

The statement to execute while the list is scanned is the *statement* factor in our syntax.

Here is an example:

```
class Exercise
{
    static int Main()
    {
        string[] EmployeeName = new string[4];

        EmployeeName[0] = "Joan Fuller";
        EmployeeName[1] = "Barbara Boxen";
        EmployeeName[2] = "Paul Kumar";
        EmployeeName[3] = "Bertrand Entire";

        Console.WriteLine("Employees Records");
        foreach(string strName in EmployeeName)
            Console.WriteLine("Employee Name: {0}", strName);

        return 0;
    }
}
```

Coalesce

This would produce:

```
Employees Records
Employee Name: Joan Fuller
Employee Name: Barbara Boxen
Employee Name: Paul Kumar
Employee Name: Bertrand Entire
Press any key to continue
```

Practical Learning: Using foreach

1. To use the foreach keyword, change Main() function as follows:

```
using System;

namespace BusinessApplication
{
    class Exercise
    {
        static void Main()
        {
            ConvenienceStore[] store = new ConvenienceStore[5];

            CreateStoreStock(store);

            foreach(ConvenienceStore item in store)
            {
                Console.WriteLine("Item #: {0}", item.StockNumber);
                Console.WriteLine("Description: {0}", item.Description);
                Console.WriteLine("Unit Price: {0}\n", item.UnitPrice);
            }

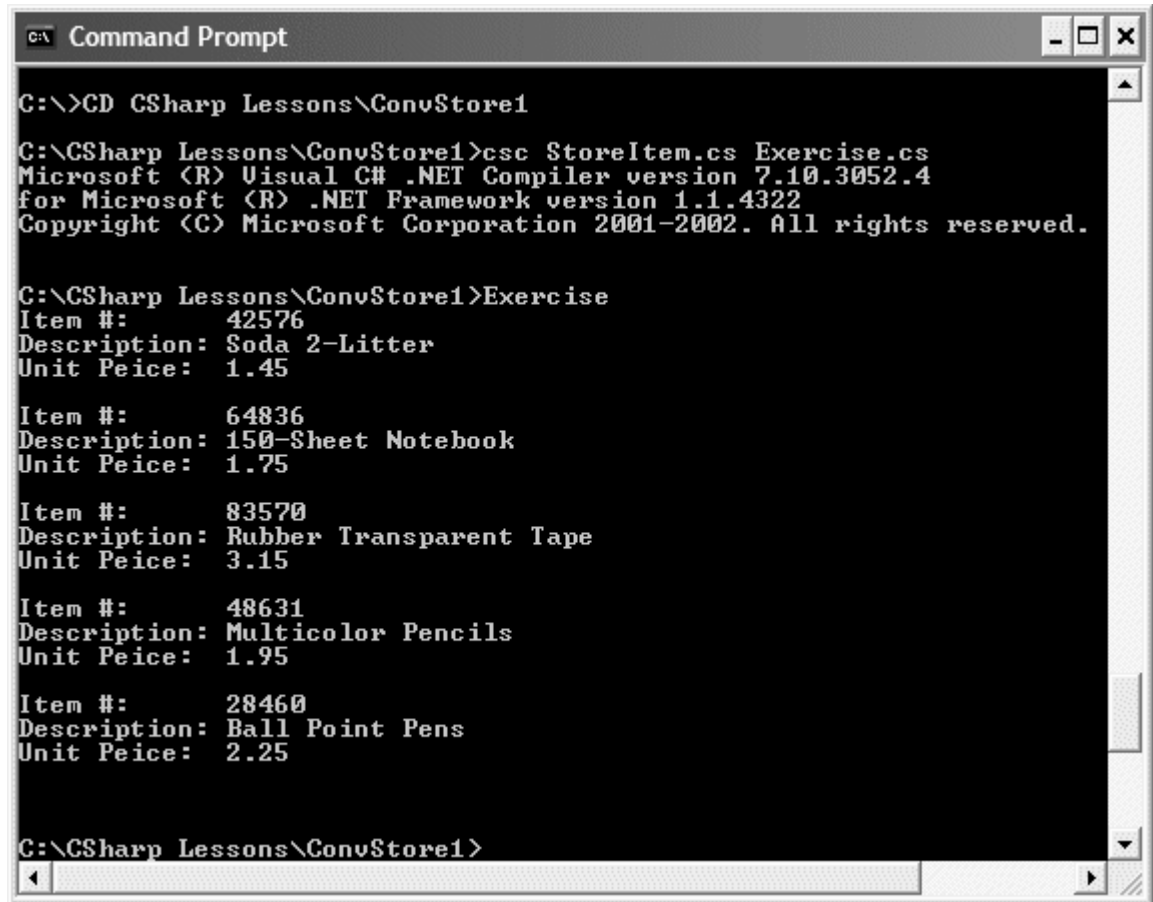
            Console.WriteLine();
        }

        static void CreateStoreStock(ConvenienceStore[] cnvStore)
        {
            ... No Change
        }
    }
}
```

2. Save the Exercise.cs file
3. Open the Command Prompt and switch to the folder where the current project is located
4. Compile the project with **csc StoreItem.cs Exercise.cs**

Coalesce

5. Execute it with **Exercise**



```
C:\>CD CSharp Lessons\ConvStore1

C:\CSharp Lessons\ConvStore1>csc StoreItem.cs Exercise.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.

C:\CSharp Lessons\ConvStore1>Exercise
Item #:      42576
Description: Soda 2-Litter
Unit Peice:  1.45

Item #:      64836
Description: 150-Sheet Notebook
Unit Peice:  1.75

Item #:      83570
Description: Rubber Transparent Tape
Unit Peice:  3.15

Item #:      48631
Description: Multicolor Pencils
Unit Peice:  1.95

Item #:      28460
Description: Ball Point Pens
Unit Peice:  2.25

C:\CSharp Lessons\ConvStore1>
```

C# Arrays

The technique we used to create arrays in previous lessons was the traditional routine used in many languages. The Microsoft .NET Framework goes further than that. Every time you declare an array, it inherits from a class called `Array`. The `Array` provides support for any array with many issues. After creating an array as we have done so far, you may want to rearrange the list based on one of the items. You may want to look for a particular item to find if it exists; then, if it exists, where it is located. The `Array` class provides many other routines to perform such operations

Item Addition

As done so far, to add an item to a list, you specify its index and assign the desired value. Here is an example:

```
class Exercise
{
    static void Main()
    {
        string[] EmployeeName = new string[4];

        EmployeeName[0] = "Joan Fuller";
```

Coalesce

```
        EmployeeName[1] = "Barbara Boxen";
        EmployeeName[2] = "Paul Kumar";
        EmployeeName[3] = "Bertrand Entire";
    }
}
```

Alternatively, you can use the **SetValue()** method to add a new item to the list. This method has 8 versions. The first argument is the value that needs to be added to the list, the other argument(s) is(are) used for the index. Here is an example:

```
class Exercise
{
    static void Main()
    {
        string[] EmployeeName = new string[4];

        EmployeeName.SetValue("Joan Fuller", 0);
        EmployeeName.SetValue("Barbara Boxen", 1);
        EmployeeName.SetValue("Paul Kumar", 2);
        EmployeeName.SetValue("Bertrand Entire", 3);
    }
}
```

Item Retrieval

To locate an item of an array, we were using its index. Here is an example:

```
class Exercise
{
    static void Main()
    {
        string[] EmployeeName = new string[4];

        EmployeeName[0] = "Joan Fuller";
        EmployeeName[1] = "Barbara Boxen";
        EmployeeName[2] = "Paul Kouma";
        EmployeeName[3] = "Bertand Entire";

        Console.WriteLine("Employees Records");
        Console.WriteLine("Employee 1: {0}", EmployeeName[0]);
        Console.WriteLine("Employee 2: {0}", EmployeeName[1]);
        Console.WriteLine("Employee 3: {0}", EmployeeName[2]);
        Console.WriteLine("Employee 4: {0}", EmployeeName[3]);
    }
}
```

Alternatively, the Array class provides the **GetValue()** method, also loaded with 8 versions. Here is an example:

```
class Exercise
{
```

Coalesce

```
static void Main()
{
    string[] EmployeeName = new string[4];

    EmployeeName.SetValue("Joan Fuller", 0);
    EmployeeName.SetValue("Barbara Boxen", 1);
    EmployeeName.SetValue("Paul Kumar", 2);
    EmployeeName.SetValue("Bertrand Entire", 3);

    Console.WriteLine("Employees Records");
    Console.WriteLine("Employee 1: {0}", EmployeeName.GetValue(0));
    Console.WriteLine("Employee 2: {0}", EmployeeName.GetValue(1));
    Console.WriteLine("Employee 3: {0}", EmployeeName.GetValue(2));
    Console.WriteLine("Employee 4: {0}", EmployeeName.GetValue(3));
}
}
```

The Length of an Array

After creating an array and while using it, it is always important to know the number of items in the collection. This number is known as the length of the collection.

To know the number of items in an array or a collection, you can access its **Length** property.

The ArrayList Class

To support arrays of any kind, the Microsoft .NET Framework provides the **ArrayList** class. This class can be used to add, locate, or remove an item from a list. The class provides many other valuable operations routinely done on a list.

Practical Learning: Introducing the ArrayList Class

1. Start a new file in Notepad
2. In the empty file, type the following:

```
using System;

class StoreItem
{
    long  ItemNumber;
    string ItemName;
    double UnitPrice;
}
```

3. Save the file in a new folder named **DeptStore5**
4. Save the file itself as **item.cs**
5. Without closing Notepad, open another instance of Notepad (start "another" Notepad) and type the following in it:

Coalesce

```
using System;

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

6. Save the new file as **exercise.cs** in the same DeptStore1 folder

Item Addition

The primary operation performed on a list is to create one. After declaring a variable of type **ArrayList**, you can call the **Add()** method every time you need to add a new item to the list.

Practical Learning: Introducing the ArrayList Class

1. To create a new list and initialize it, change the exercise.cs file as follows:

```
using System;

namespace DepartmentStore
{
    class Sales
    {
        static void Main()
        {
            ArrayList lstItems = new ArrayList();
            StoreItem anItem;

            anItem = new StoreItem();
            anItem.ItemNumber = 31882;
            anItem.ItemName = "Zembla Sling Shoe - Women";
            anItem.UnitPrice = 255.75;
            lstItems.Add(anItem);

            anItem = new StoreItem();
            anItem.ItemNumber = 70517;
            anItem.ItemName = "14-Karat Rain Diamond Earrings";
            anItem.UnitPrice = 1250.55;
            lstItems.Add(anItem);

            anItem = new StoreItem();
            anItem.ItemNumber = 36308;
            anItem.ItemName = "Men Khaki Pants";
            anItem.UnitPrice = 32.90;
            lstItems.Add(anItem);

            anItem = new StoreItem();
            anItem.ItemNumber = 11582;
```


Coalesce

```
        anItem.ItemName = "Children Summer Hat";
        anItem.UnitPrice = 18.75;
        lstItems.Add(anItem);

        anItem = new StoreItem();
        anItem.ItemNumber = 44005;
        anItem.ItemName = "Men Black Leather Shoe";
        anItem.UnitPrice = 225.45;
        lstItems.Add(anItem);

        Console.WriteLine("Store Inventory");
        for(int i = 0; i < lstItems.Count; i++)
        {
            Console.WriteLine("Index: {0}", i);
            anItem = (StoreItem)lstItems[i];
            Console.WriteLine("Item Stock #: {0}", anItem.ItemNumber);
            Console.WriteLine("Description: {0}", anItem.ItemName);
            Console.WriteLine("Unit Price: {0:C}\n", anItem.UnitPrice);
        }
    }
}
```

2. Open the Command Prompt and switch to the folder that contains the files of the current project
3. Compile it with **csc item.cs exercise.cs**
4. Exercise it with exercise

Chapter 16

Windows Application

Introduction

Microsoft Visual C# is a programming environment used to create applications for the Microsoft Windows family of operating systems.

To use the lessons on this site, you must know a little bit of the **C# language** as we cover it on this site

The Visual Studio .Net IDE presents an impressive interface that allows you to start creating applications. It doesn't start with an application so you can decide what you want to do. This is because there are various types of applications you can create. The screen presents what is referred to as an Integrated Development Environment or IDE.

The top section of the IDE is made of the title bar. You can use the title bar to close, move, minimize or maximize the application.

Under the title bar, the main menu presents a list of actions you can perform during the design of an application. The actions are organized in categories and each category is represented with a word. Examples are File, Edit, Tools, etc.

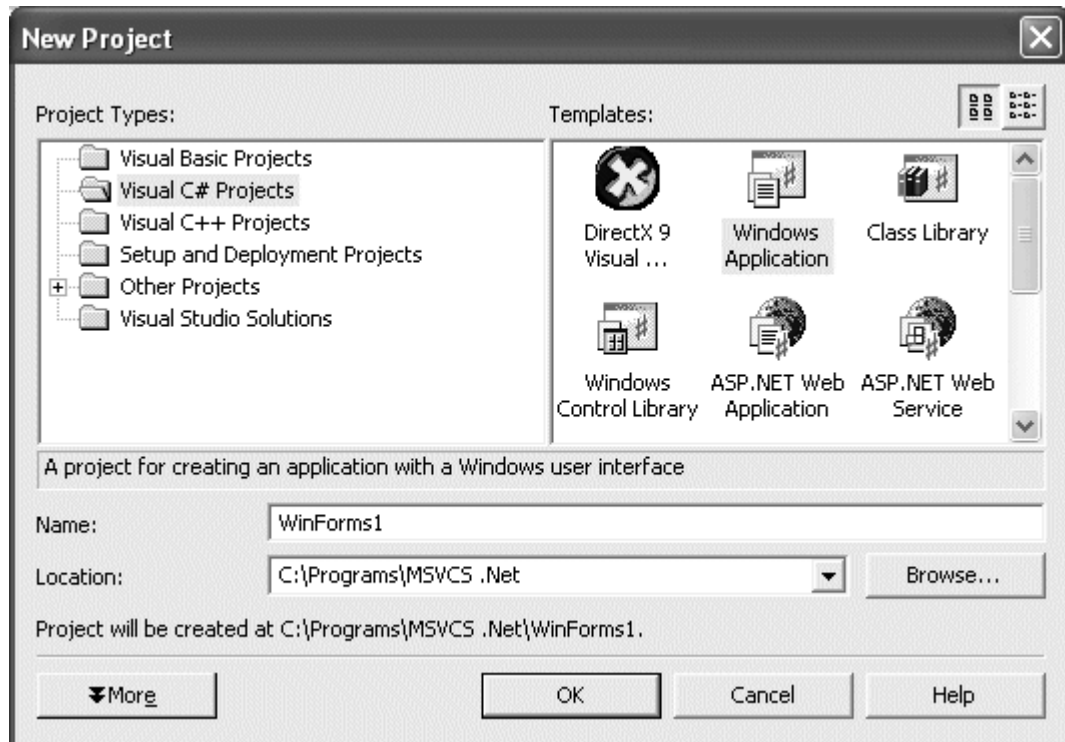
On this site, the menu on top of the IDE is always called the main menu

Creating a Project

An example of using the main menu consists of creating a new project. To do this:

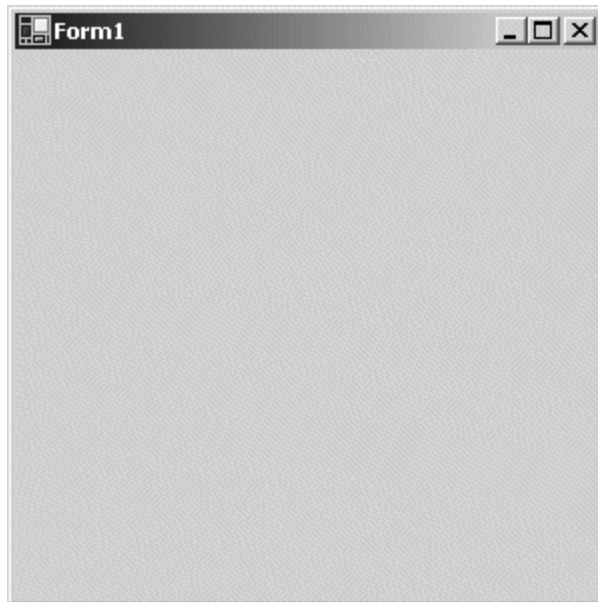
1. On the main menu, click File -> New -> Project...
2. On the New Project dialog box, click the Visual C# Projects node in the Project Types tree list
3. On the Templates list, click Windows Application
4. In the Name edit box, you can type a name for the project or accept the one suggested. An example of a name would be WinForms1


Coalesce



5. After doing this, click OK. This creates a new application with a default form.

6. To execute the application, on the main menu, click Debug -> Start Without Debugging



7. After using the form, close it by clicking its system close button  and return to your programming environment.

Under the main menu, a long bar made of small pictures displays. This is a toolbar. This particular object is called the Standard toolbar.

Coalesce

On the left side of the screen, there is a bar with the word **Toolbox** in a vertical direction. If you position the mouse on it, it extracts to display items arranged by categories. Once you position your mouse away from it, the bar goes back to the way it previously was.

At first glance, the main and middle area of the screen appears blank. It will eventually be used to display code or a list of previous projects.

On the right side, the screen is divided in two sections. The top section includes various tabs such as the **Class View**, the **Resource View**, and the **Solution Explorer**. Once you start using **Help**, other tabs will be added.

In the lower section of the right side of the screen, there is window that displays things depending on what is selected in the top section.

The description we have just givens assumes that you are using the default arrangement of the **Visual Studio IDE**.

Coalesce

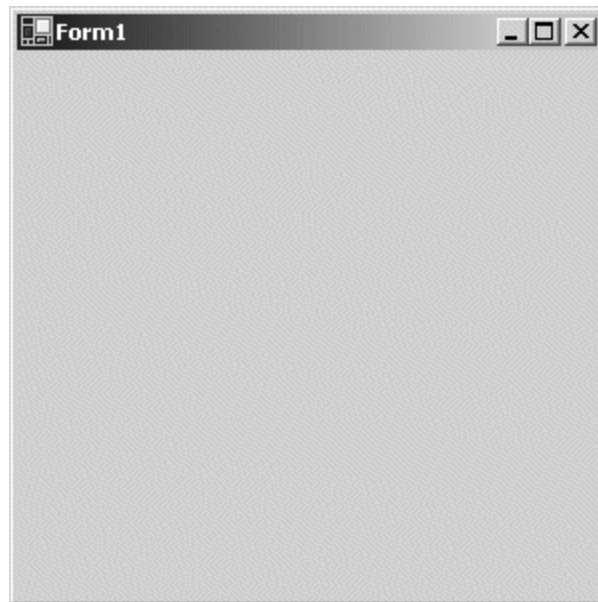
Chapter 17

Form Element

Characteristics of a Form

Introduction

The Form is the most fundamental objects used in an application. By itself, a form does nothing. It's main role is to host other objects that the user uses to interact with the computer:




There are two main ways you get a form to your application. If you create a Windows Forms Application, it creates a starting form for you. Otherwise, the other technique consists of explicitly adding a form to your application.

1. Start Microsoft Visual Studio .NET
2. On the Start Page, click New Project (alternatively, on the main menu, you can click File -> New -> Project...)
3. On the New Project dialog box, in the Project Types tree list, click **Visual C# Projects**
4. In the Templates list, click **Windows Application (.Net)**
5. In the Name edit box, replace the <Enter name> content with WinForms2
6. In the Location combo box, accept the suggestion or type your own. Otherwise, type C:\Programs\MSVCS .NET
7. Click OK
8. Press F5 to test the program
9. Close it and return to Visual Studio

Coalesce

The Title Bar

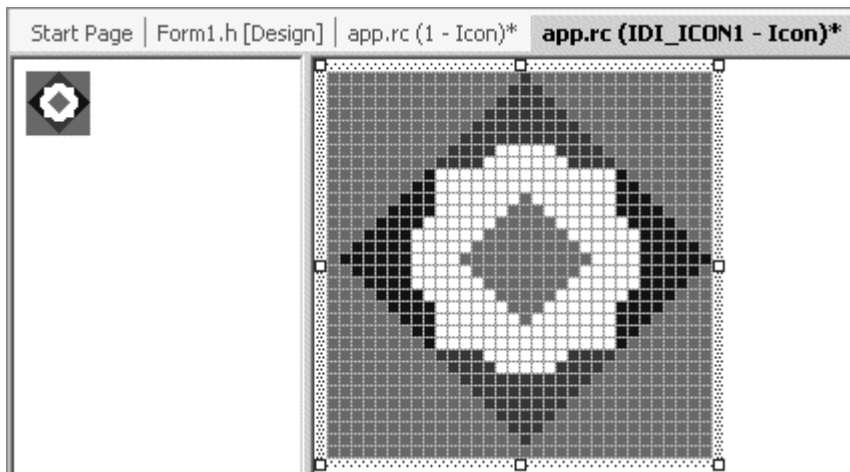
A form is made of various sections that allow its control as a Windows object and other aspects that play a valuable part as a host of other objects. The top section of a form is made of a long portion called the title bar.

On the left part of the title bar, the form displays a small picture called an icon or the system icon. Microsoft Visual Studio provides a default icon for all forms. If you want to use a different icon, while the form is selected, on the Properties window, you can click the Icon field and then click its ellipsis button . This would launch the Open dialog box from where you can locate an icon and open it.

To change the icon programmatically, declare a variable of the Icon class of the **System.Drawing** namespace and initialize it with the name of an icon file using the **new** operator. After initializing the icon, assign it to the form's Icon property. Here is an example:

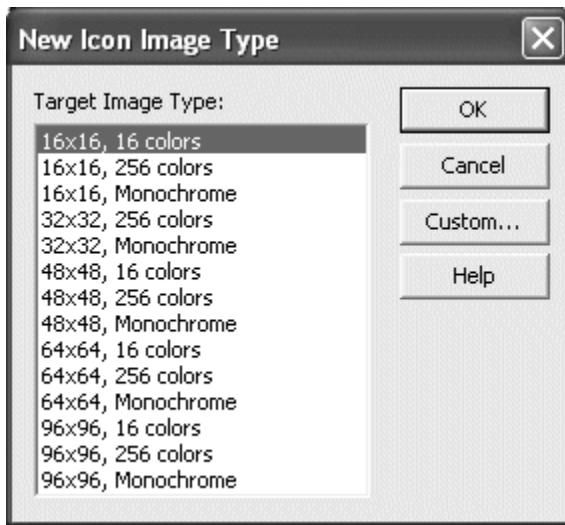
```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Drawing.Icon IC = new System.Drawing.Icon("C:\\Programs\\alarm1.ico");
    Icon = IC;
}
```

1. On the main menu, click Project -> Add Resource...
2. Click Icon and click New
3. On the main menu, click Image -> Show Colors Window
4. Using the colors from the Colors palette and the tools from the Image -> Tools menu, design the icon as follows:

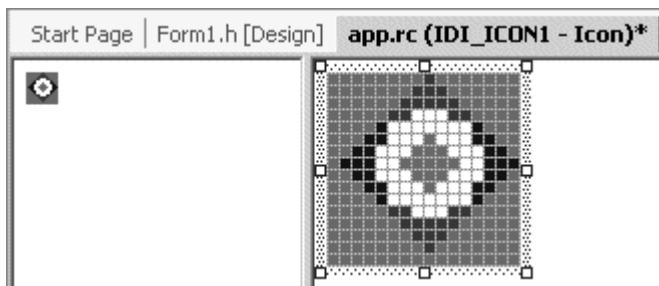



5. To continue with the icon design, on the main menu, click Image -> New Image Type... On the New Icon Image Type, make sure the first 16x16, 16 colors item is selected:

Coalesce

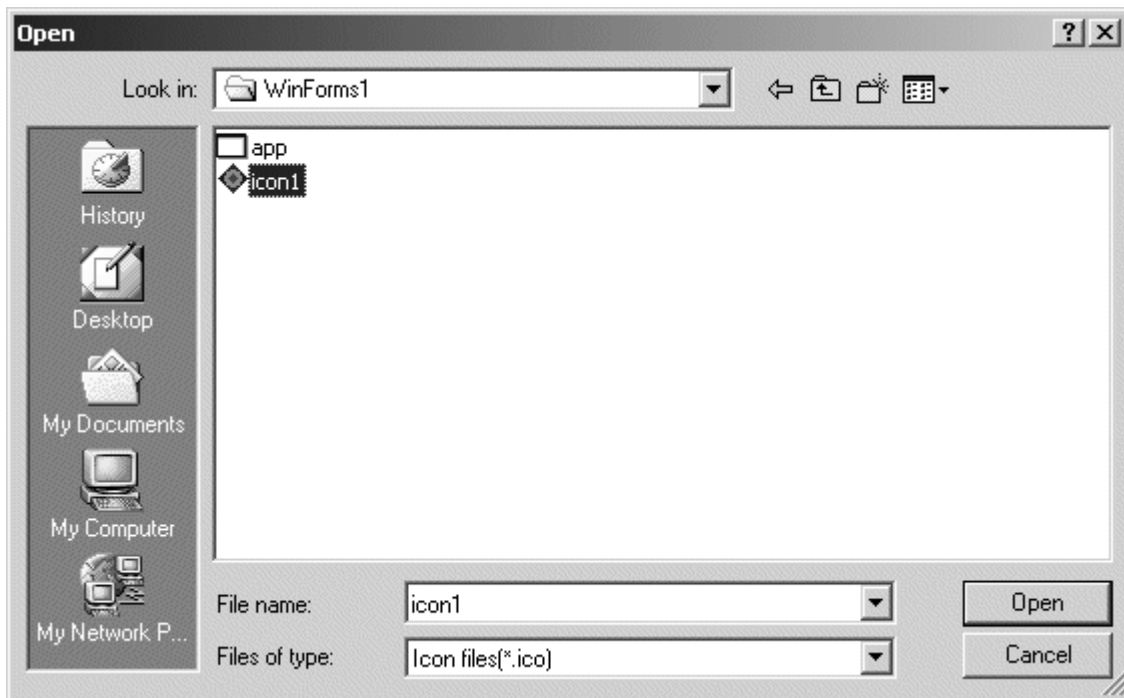


6. Click OK.
7. Design it like the other. To make it transparent, make sure you right-click the color that looks like a monitor and apply it to the external areas:



8. In the Workspace, click the Resource View tab and expand everything. Right-click **IDI_ICON1** and click Properties. In the Properties window, click ID, type **IDI_TARGET** and press Enter
9. Display the form and, on the Properties window, click Icon and click its ellipsis button 
10. On the Open dialog box, located the folder in which you had saved the project and select the icon1 icon

Coalesce



11. Click Open
12. Press Ctrl + F5 to test the program
13. Close it and return to MSVC

The Form's Caption

On the right side of the system icon, there is a word or a group of words called the caption. By default, the caption displays the name of the form. If you want to change the caption, while the form is selected, on the Properties window, click the Text field and type anything you want. After typing the text, press Enter to display it on the form. At design time, the caption is made of text that you type "as is". At run time, you can change the caption to display a more complex text that could be a changing time, the result of a calculation, etc.






1. While the form is selected, on the Properties window, click Text and type

Form Implementation


2. Press Enter

The System Buttons

The system buttons are located on the right side of a title bar. They are used to minimize, maximize, restore, or close a window.

On the right side of the caption, there are three small buttons called the system buttons, made of the Minimize , Maximize , and Close buttons . The Minimize  and the Maximize  buttons are each controlled by a Boolean property. Therefore, to disable either one of the them, set its **MinimizeBox** or **MaximizeBox** property to **false**. If you set one of them to **false** while the other is

Coalesce

true, the one set to **false** would appear disabled. If you set both to **false**, both would be hidden and only the Close  button would appear.

To change a system button programmatically, call the desired button's property and assign it a true or false value. Here is an example that makes sure the user cannot maximize the form:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Drawing.Icon IC = new System.Drawing.Icon("C:\\Programs\\alarm1.ico");
    Icon = IC;

    MinimizeBox = true;
    MaximizeBox = false;
}
```

In the same way, to hide both the Maximize and the Minimize buttons programmatically, you change their values as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Drawing.Icon IC = new System.Drawing.Icon("C:\\Programs\\alarm1.ico");
    Icon = IC;

    MinimizeBox = false;
    MaximizeBox = false;
}
```

1. To make sure that the user cannot maximize the form, on the Properties window, click the **MaximizeBox** field to reveal its combo box and select **False**

Coalesce

2. Test the program



The Form Borders

A form can be made to look like a regular rectangular control host made of a system icon and the system buttons. Depending on your goals, you can also make a form appear as a dialog box or a dockable window. The borders of a form are controlled by the **FormBorderStyle** property.

If you set both the **MinimizeBox** and the **MaximizeBox** properties to **false**, we saw that the form would have only the system Close button, but the form can still be resized. If you want the form to behave like a dialog box that has only the system Close button and cannot be resized, set its **FormBorderStyle** property to **FixedDialog**. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Drawing.Icon IC = new System.Drawing.Icon("C:\\Programs\\alarm1.ico");
    Icon = IC;

    MinimizeBox = true;
    MaximizeBox = false;

    FormBorderStyle = FormBorderStyle.FixedDialog;
}
```

A tool window is a form equipped with a short title bar, no icon, and only a small system Close button. There are two types of tool windows. A tool window is referred to as fixed if the user cannot resize it.

FormBorderStyle = FormBorderStyle.FixedToolWindow;

A sizable tool window is a tool window that allows the user to resize it.

Coalesce

FormBorderStyle = FormBorderStyle::SizableToolWindow;

You can also create a form with no borders by assigning **None** to the **FormBorderStyle** property. If you do this, make sure you provide the user with a way to close the form; otherwise...



The Form's Body

The main area of the form can be referred to as its body but it is usually called the client area. It is made of a color a priori specified by the operating system. To change the color to anything you like, you can use the **BackColor** field of the Properties window. To programmatically change its color, assign a color from the **Color** structure to the **BackColor** property. Here is an example:

BackColor = Color.AliceBlue;

If you prefer to cover the client area with a picture, use the **BackgroundImage** property instead. If using the Properties window, you can easily locate and select a picture. To programmatically specify or change the picture used as background, declare and initialize a pointer to the **Bitmap** class. Then assign it to the **BackgroundImage** property. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Drawing.Icon IC = new System.Drawing.Icon("C:\\Programs\\alarm1.ico");
    Icon = IC;

    Bitmap Bg = new Bitmap("C:\\Programs\\E350.bmp");
    this.BackgroundImage = Bg;
}
```

Coalesce



The Window State of a Form

When a form appears as it was designed, it is said to be "normal". You can configure a form to be minimized or maximized when the application is launched. This ability is controlled by the **WindowState** property. The default value of this property is **Normal** which means the form would appear in a normal fashion. If you want the form to be minimized or maximized at startup, in the Properties window, select the desired value for the **WindowState** property.

To control the window's state programmatically, assign the **Maximized** or **Minimized** value, which are members of the **FormWindowState** enumerator, to the **WindowState** property. Here is an example:

```
Fm.WindowState = FormWindowState.Maximized;
```

If you want to check the state of a window before taking action, simply use a conditional statement to compare its **WindowState** property with the **Normal**, the **Maximized**, or the **Minimized** values.

The Form's Taskbar Presence

When an application displays on the screen along with other applications, its form can be positioned on top or behind forms of other applications. This is allowed by multitasking assignments. When a form is hidden, the taskbar allows you to access it because the form would be represented by a button. This aspect of forms is controlled by the **ShowInTaskbar** Boolean property. Its default value is **True**, which indicates that the form would be represented on the taskbar by a button.

If you create an application made of various forms, you may not want to show all of its forms on the taskbar. Usually the first or main form would be enough. To prevent a button for a form to display on the taskbar, set its **ShowInTaskbar** property to False.

Methods to Manage a Form

Form Creation

Coalesce

The form is implemented by the **Form** class from Forms namespace of the Forms namespace of the Windows namespace of the System namespace. The **Form** class is equipped with a constructor that allows you to dynamically create it.

Form Closure

When the user has finished using a form, he or she must be able to close it. Closing a form is made possible by a simple call to the `Close()` method. Its syntax is:

```
void Close();
```

Although this method can be used to close any form of an application, if it is called by the main form, it also closes the application.

Form Activation

When two or more forms are running on the computer, only one can receive input from the user; that is, only one form can actually be directly used at one particular time. Such a window has a title bar with the color identified in Control Panel as Active Window. The other window(s), if any, display(s) its/their title bar with a color called Inactive Window.

To manage this setting, the windows are organized in a 3-dimensional coordinate system and they are incrementally positioned on the Z coordinate, which defines the (0, 0, 0) origin on the screen (on the top-left corner of your monitor) with Z coordinate coming from the screen towards you.

In order to use a form other than the one that is active, it must be activated. To do this, the **Activated()** event must be fired.

Form Deactivation

If there is more than one form or application on the screen, only one can be in front of the others and be able to receive input from the others. If a form is not active and you want to bring it to the top, you must activate it, which sends the **Activated()** event. When a form is being activated, the one that was on top would become deactivated. The form that was on top, as it loses focus, would fire the **Deactivated()** event.

An Application of Multiple Forms

Using Multiple Forms

When you create a Windows Application, its starting form is made available to you. If one form is not enough for your application, you can add as many as necessary. To add (or to create) a (new) form, you should display the Add New Item dialog box. To do this,

- on the main menu, you can click File -> Add New Item...
- on the main menu, you can click Project -> Add New Item...

Coalesce


- In the Solution Explorer, you can right-click the name of the project and click Add New Item...

On the Add New Item dialog box, in the Templates section, click Window Form, provide a name in the Name edit box and click Open.

If your application is using various forms and you want to display a particular one at design time, on the main menu, you can click Window. On the list of files,

- to display the "physical" view of a form, click its name that has [Design]
- To display the source code of a form, click its name that has the .h

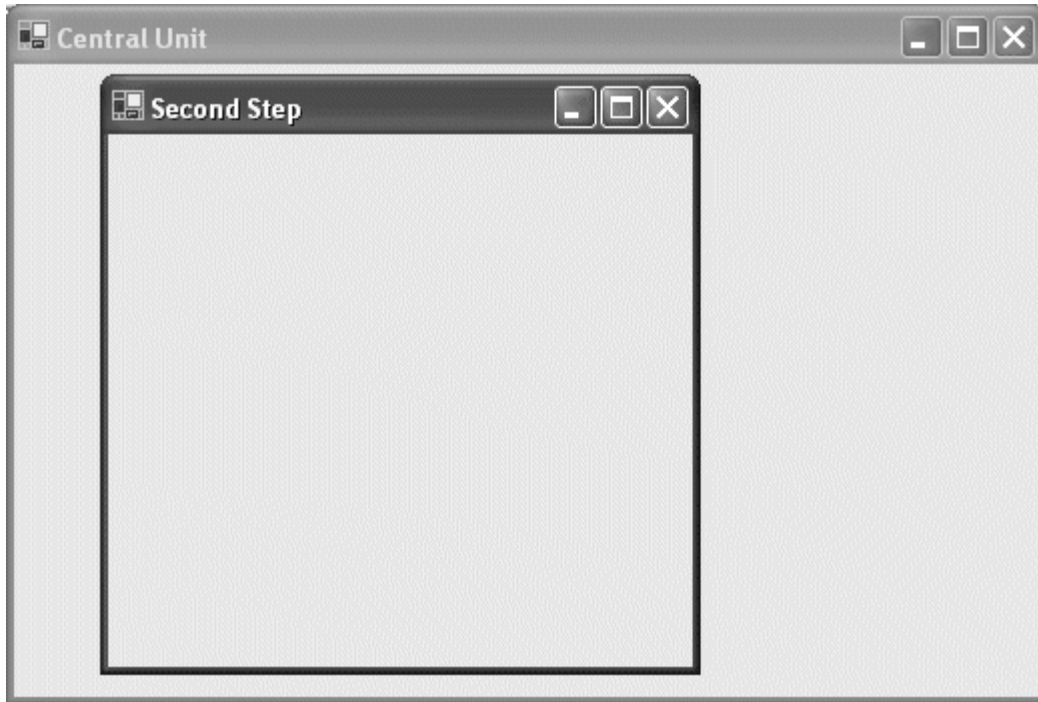
If you visually add two (or more) forms to your application, you may need to link them, allow one to call the other. Since each form is created in its own header file, to let one form know about the existence of another, you must include its header. A form or unit that wants to communicate with another is called a client of the form.

1. Create a new Windows Application and name it MultiForms1
2. While the new form is still displaying, in the Properties window, click Text and type **Central Unit**
3. Click Size, type **520, 350** and press Enter
4. To add another form, on the main menu, click File -> Add New Item...
5. In the Add New Item dialog box, in the Templates list, click Windows Form
6. Replace the contents of the Name edit box with Second and press Enter
7. While the second form is still displaying, in the Properties window, edit its **Text** property to display **Second Step**
8. Set its **ShowInTaskbar** property to **False**
9. To display the first form, in the code editor, click the Form1.cs [Design] tab
10. In the Properties window, click the Events button 
11. Double-click the **DoubleClick** field and implement its event as follows:

```
private void Form1_DoubleClick(object sender, System.EventArgs e)
{
    Second Dos = new Second();
    Dos.Show();
}
```

Coalesce

12. Execute the application



13. If you double-click the first form, the second would display. You may find out that there is a problem in the fact that every time you double-click the first form, a new instance of the second form displays, which may not be very professional
14. Close it and return to MSVC
15. Display the second form. Using the Events button in the Properties window, access its Closing event and implement it as follows:

```
private void Second_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    // When the user attempts to close the form, don't close it
    e.Cancel = true;
    // Only hide it
    this.Visible = false;
}
```

16. Display the source file of the first form (Form1.cs) and change it as follows:

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

namespace MultiForms1
{
```

Coalesce

```
call
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;
        private Second Dos = new Second();

        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            //
            // TODO: Add any constructor code after InitializeComponent
            //
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( 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()
        {
            //
            // Form1
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.ClientSize = new System.Drawing.Size(512, 316);
            this.Name = "Form1";
        }
    }
}
```


Coalesce

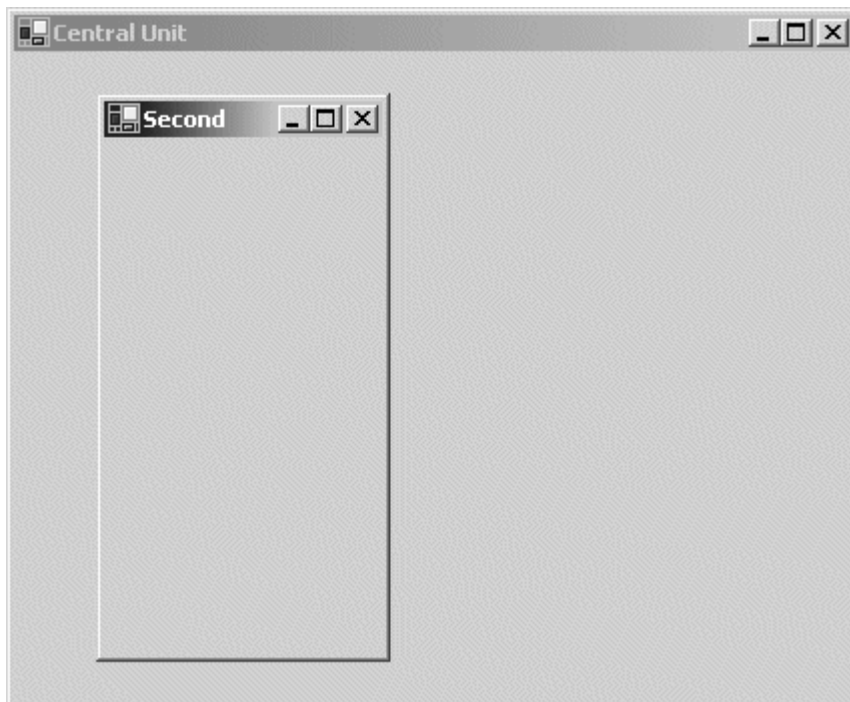
```
        this.Text = "Central Unit";
        this.DoubleClick += new
System.EventHandler(this.Form1_DoubleClick);

    }
    #endregion

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.Run(new Form1());
    }

    private void Form1_DoubleClick(object sender, System.EventArgs e)
    {
        Dos.Visible = true;
    }
}
```

17. Execute the application



18. Close it and return to Visual Studio

Coalesce

The Multiple Document Interface (MDI)

Introduction to MDI-Based Applications

A multiple document interface, abbreviated MDI, is an application whose main form directly "owns" other forms. The main form is also considered the parent. The forms that the parent form owns can display only within the rectangular borders of the parent form. The main idea behind an MDI is to allow the user to open different documents and work on them without having to launch another instance of the application.

As opposed to an MDI, a Single Document Interface (SDI) allow only one document at a time in the application.

WordPad is an example of an SDI. The user can open only one document at a time. If he wants another WordPad document, he must open an additional instance of WordPad.

Each form that is child of the main form in an MDI can be a fully functional form and most, if not all, child forms are of the same kind. There are two main ways a child document of an MDI displays. To provide information about its state, a child form is equipped with a title bar. The title bar of a child form displays an icon, the name of its document, and its own system buttons. Since it can be confined only within the parent form, it can be minimized, maximized, or restored within the parent form. When a child form is not maximized, it clearly displays within the parent form. If it gets closed and there is no other child form, the parent form would appear empty. If there are various child forms, they share the size of the client area of the parent form. If one of the child forms is maximized, it occupies the whole client area of the main form and it displays its name on the title bar of the main form.

MDI Creation

It is not particular difficult to create an MDI application. The challenge may come when you need to configure its functionality. To start, you must set a form to be the eventual parent of the application. This can be taken care of by setting its **IsMDIContainer** property to true. After doing this, the body of the form is painted in gray with a sunken client area that indicates that it can host other form.

After creating the parent form of the MDI, you must provide a way to display a child form when needed. This is usually done with a menu, which we haven't covered yet. When displaying a child form of the MDI, access its **MdiParent** property and assign it the name of the parent form. Then display the child.

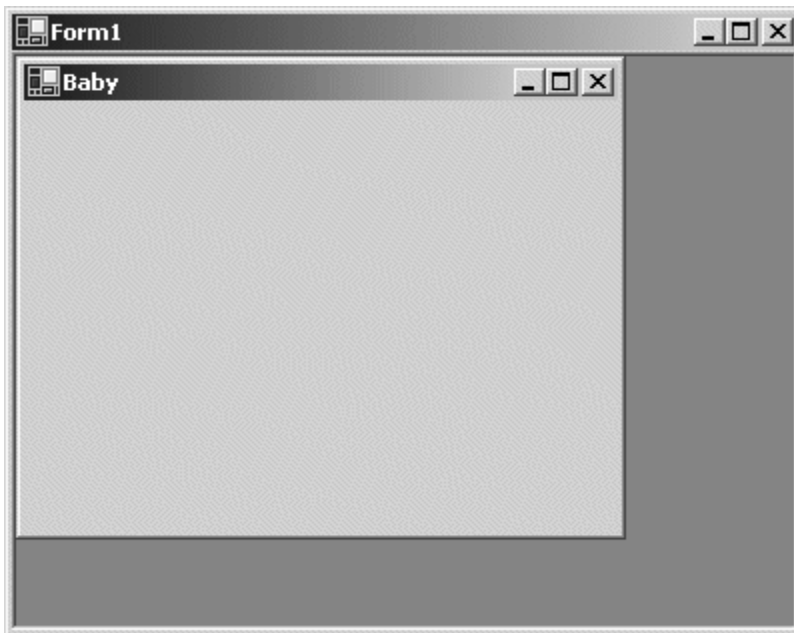
1. Start a new Windows Application and name it **MDI1**
2. While the main form is still displaying, in the Properties window, set its **IsMdiContainer** property to **True**
3. Click Size, type **620, 400** and press Enter
4. To add another form that will be used as a child, on the main menu, click Project -> Add New Item...
5. In the Add New Item dialog box, in the Templates list, click Windows Form (.NET)
6. Replace the contents of the Name edit box with **Baby** and press Enter

Coalesce

7. To display the first form, in the code editor, click the Form1.cs [Design] tab
8. Double-click in the middle of the first form to access its Load() event and implement it as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    Baby MyBaby = new Baby();
    MyBaby.MdiParent = this;
    MyBaby.Show();
}
```

9. Execute the application



10. Close it and return to Visual Studio

Chapter 18

Controls Design


When creating an application, your primary job will consist of adding objects called Windows controls, or simply controls, to the application. These controls are what the users of your application use to interact with the computer. Therefore, you will select the necessary controls and add them to your application before configuring their behavior.

To support the idea of rapid application development (RAD), the Visual Studio .Net IDE provides the necessary objects on the Toolbox located on the left side of the screen. To select an object to add

Coalesce

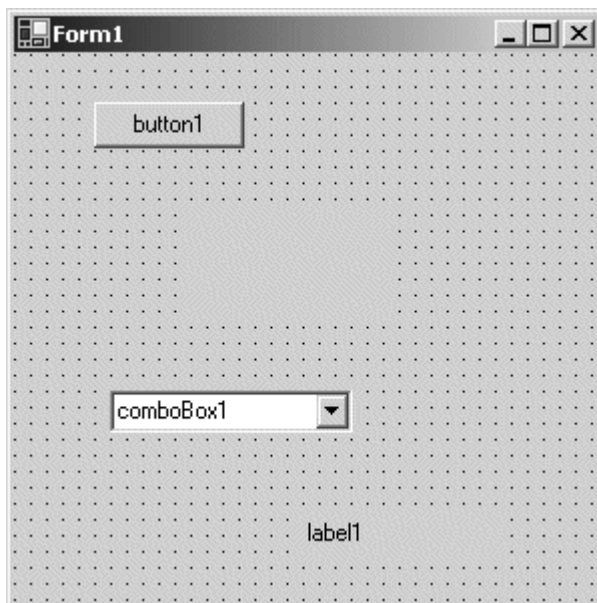
to your application, you can first display or expand the Toolbox. To do this, you can position the mouse on the Toolbox bar. This expands the Toolbox and displays its list of Windows controls.

As long as the mouse is positioned on it, the controls would be available. If you move the mouse away, the Toolbox would go back. This is convenient if you want to select one or only a few objects. If you are designing a controls-intensive application, you may want the Toolbox to stay available until you decide to dismiss it.

To keep the Toolbox or any other window fixed in its expanded mode, click its Auto Hide button  .

Adding Controls

The primary object you will use in your applications is the form. Its main job is to host other controls. Therefore, to use a control in your application, you can select it from the Toolbox and click the desired area on the form. Once added, a control is positioned where your mouse landed. In the same way, you can add other controls as you judge them necessary for your application. Here is an example of a few controls added to a form:



Alternatively, to add a control, you can also double-click it from the Toolbox and it would be added to the top-left section of the form.

If you want to add a certain control many many times, before selecting it on the Toolbox, press and hold Ctrl. Then click it in the Toolbox. This permanently selects the control. Every time you click the form, the control would be added. Once you have added the desired number of this control, on the Toolbox, click the Pointer button to dismiss the control.

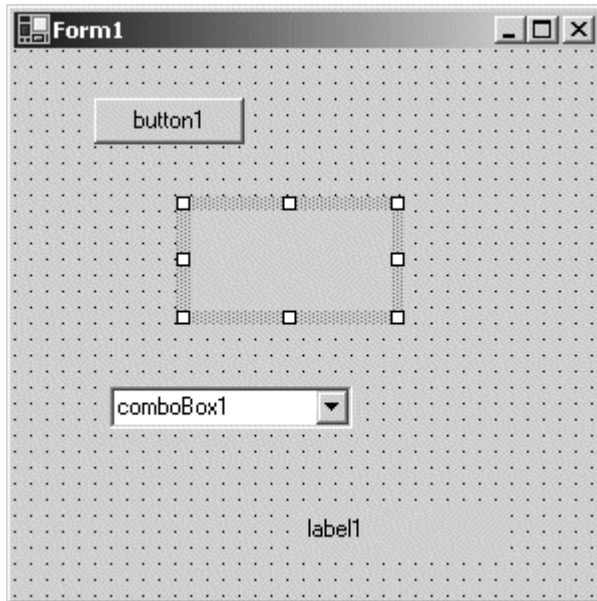
Selecting Controls

Before performing any type of configuration on a control, you will first need to select it on the form. Sometimes you will need to select a group of controls.

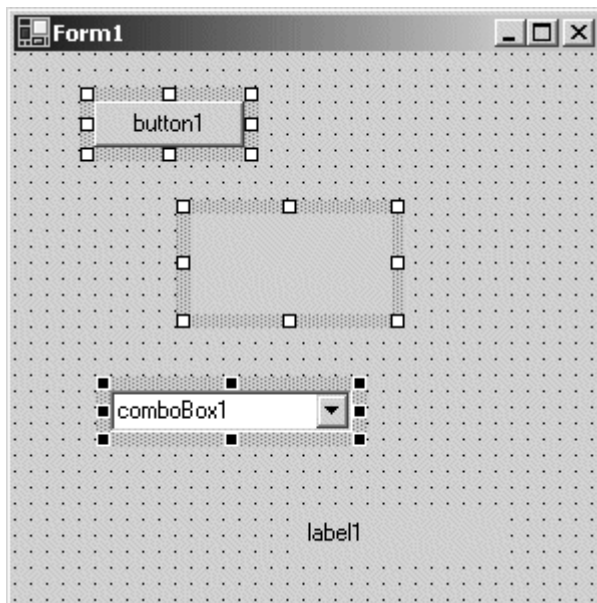
To select one control on the form, you can simply click it. A control that is selected indicates this by displaying 8 small squares, also called handles, around it. Between these handles, the control is

Coalesce

surrounded by a thick dotted rectangle. In the following picture, the selected rectangle displays 8 small squares around its shape:



To select more than one control on the form, click the first. Press and hold either Shift or Ctrl. Then click each of the desired controls. If you click a control that should not be selected, click it again. After selecting the group of control, release either Shift or Ctrl that you were holding. When a group of controls are being selected, the last selected control displays 8 handles too but its handles are dark. In the following picture, three controls are selected:



Another technique you can use to select various controls consists of clicking on an unoccupied area on the form, holding the mouse down, drawing a fake rectangle, and releasing the mouse. Every control touched by the fake rectangle or included in it would be selected.

Moving Controls

Coalesce

One of the operations you will perform during design consists of moving controls around to give them a better location and take advantage of the form's real estate. Various options are available to do this.

To move one control, click and hold the mouse on it, then drag in the desired direction, and release the mouse. Alternatively, click the control to select it. Then, to move it, press one of the arrow keys, either left to move the control left, up to move the control up, right to move the control in the right direction, or down to move the control down.

To move a group of control, select them first. Then click and drag any area in the selection to the desired location. Alternatively, once you have the controls, you can press one of the arrow keys to move the whole group.

When moving either a control or a group using either the mouse or the keyboard, the control or the group would follow the grids on the form and it can move only one grid mark at a time. This allows you to have a better alignment of controls. If you want to move the control or the group in smaller units than those of the grid, press and hold Ctrl. Then press one of the arrow keys. Once the control or the group is positioned to your liking, release the Ctrl key.

Chapter 19

Characteristics of Windows Controls

A Windows control, also called a control, is an object you provide to the user as part of your application. This object allows the user to interact with the computer. As various applications address different issues, there are various types of controls. There are some characteristics that all controls share. A characteristic of a control is also referred to as, or called, a property of the control.

There are some characteristics that are appropriate for a category of controls. And there are some characteristics so restrictively particular that only a few or just one control would use them. We will review properties shared by all or many controls. Characteristics that only one or a few controls use will be reviewed when studying those controlled.

Controls Names

Every object of your computer has a name. That is, except for the letters you see on a document, anything that has a "physical" presence on your application must have a name. The name allows you and the operating system to identify the object and refer to it appropriately when necessary.

The first object created on a Windows Application is the form. It is named Form1. When you add a control to a form, it automatically receives a name. For example, the first button would be called button1. Every control is named after its class.

Coalesce

The names automatically given to controls added to an application are incremental. That is, the first button is named button1. If you add another button, it would be named button2, etc. If an application is using various controls, and especially if it is using various copies of the same type of controls, such names become insignificant and can lead to some confusion. Therefore, you should make it a (good) habit to give appropriate names to your controls. For example, if a control deals with an issue related to the employees marital status, the name of the control should reflect it.

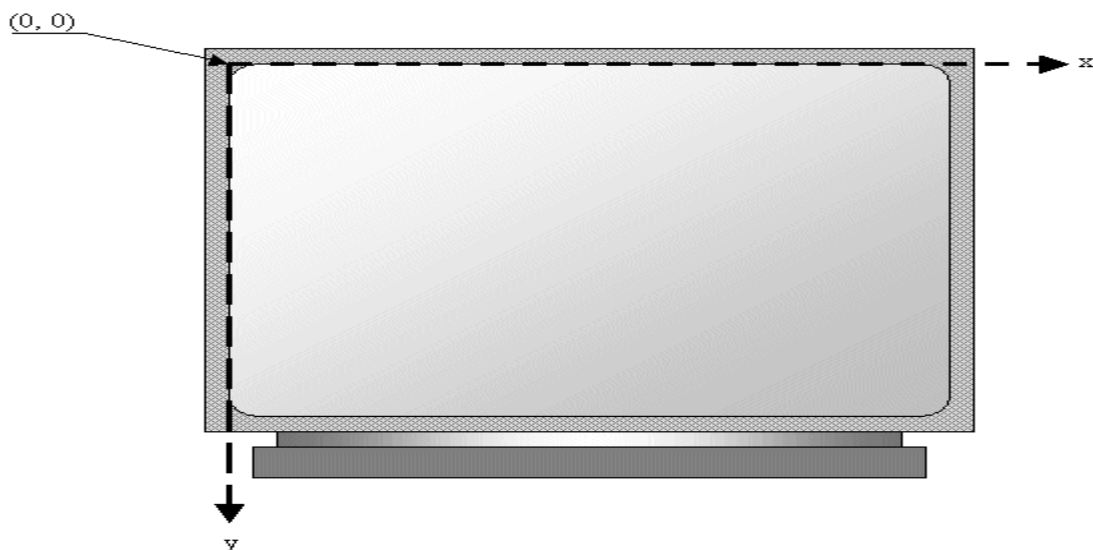
When naming controls, you must follow C#'s rules of naming variables. Besides these rules, you should also follow the .Net's suggestions that the name of a control starts in lowercase. Examples are address or age. If the name is a combination of words, the first part should be in lowercase. The first letter of the second part should start in uppercase. Examples are firstName, dateOfBirth.

Control Location

Once a control is part of your application, it must have "physical" presence. Every control has a parent. The parent gives presence to the control. The parent is also responsible for destroying the control when the parent is destroyed. When you create a Windows Application, it creates a default or first form. The parent of this main form is the Windows desktop. In fact any form that is not part of an MDI is located with regards to its parent the desktop.

The location of an object whose parent is the desktop is based on a coordinate system whose origin is located on the top-left corner of the desktop:

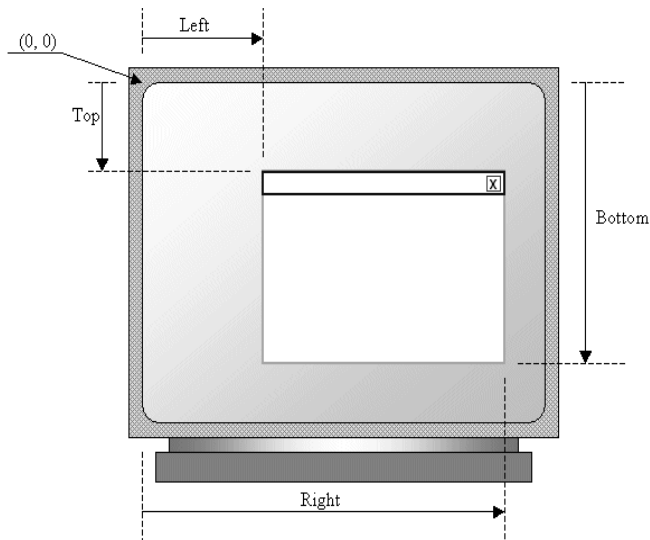
The axes of this coordinate system are oriented so the x axis moves from the origin to the right and the y axis moves from the origin down:



The distance from the left side of the desktop to the left border of the form is called the Left property. The distance from the top section of the desktop to the top border of the form is referred to as the Top property.

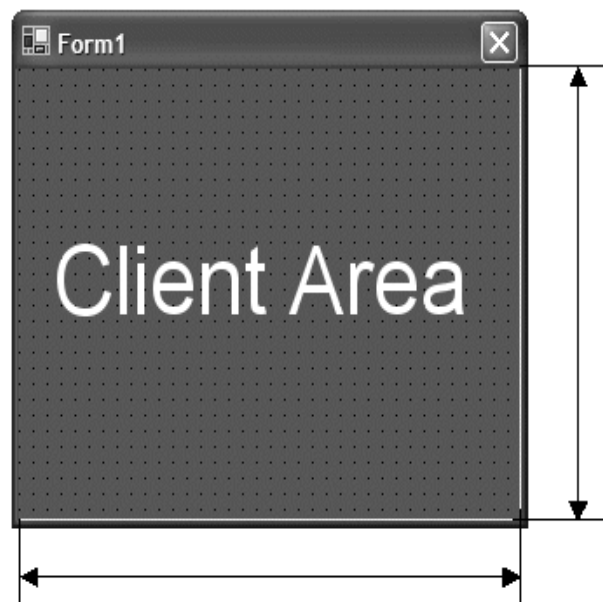
The distance from the left border of the desktop to the right border of the form is called the Right property. The distance from the top border of the desktop to the bottom border of the form is called the Bottom property. These characteristics can be illustrated as follows:

Coalesce



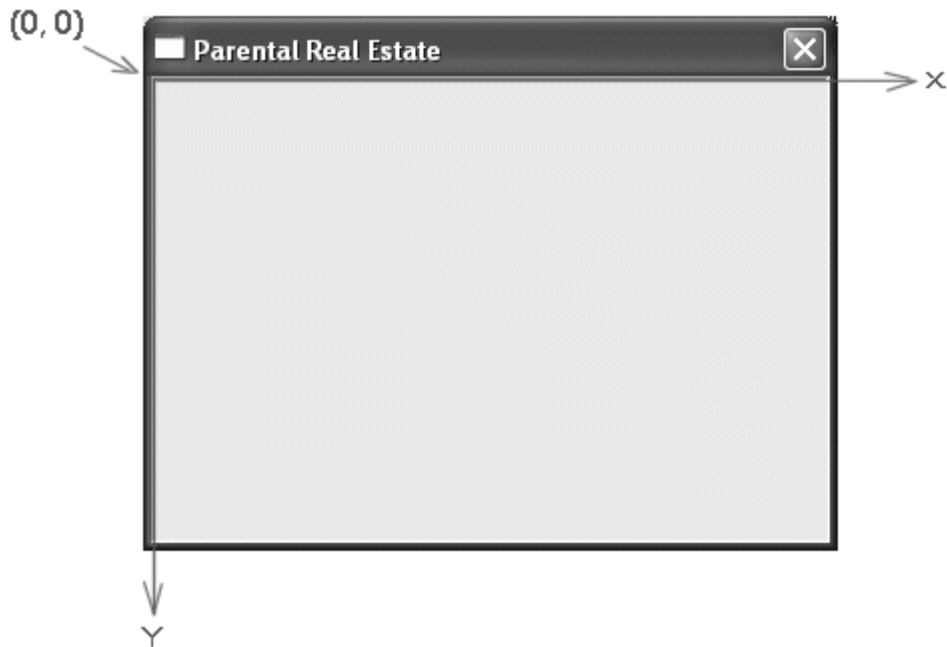
The rectangular area that the desktop makes available to all objects you can see on your screen is called the client area because this area is reserved to the children, called clients, of the desktop.

When a form has been created, we saw in the previous lesson that you can add Windows controls to it. These controls can be positioned only in a specific area, the body of the form. The body spans from the left border, excluding the border, of the form to the right border, excluding the border, of the form. It also spans from the top side, just under the title bar, to the bottom border, excluding the border, of the form. The area that the form makes available to the controls added to it is called the client area:

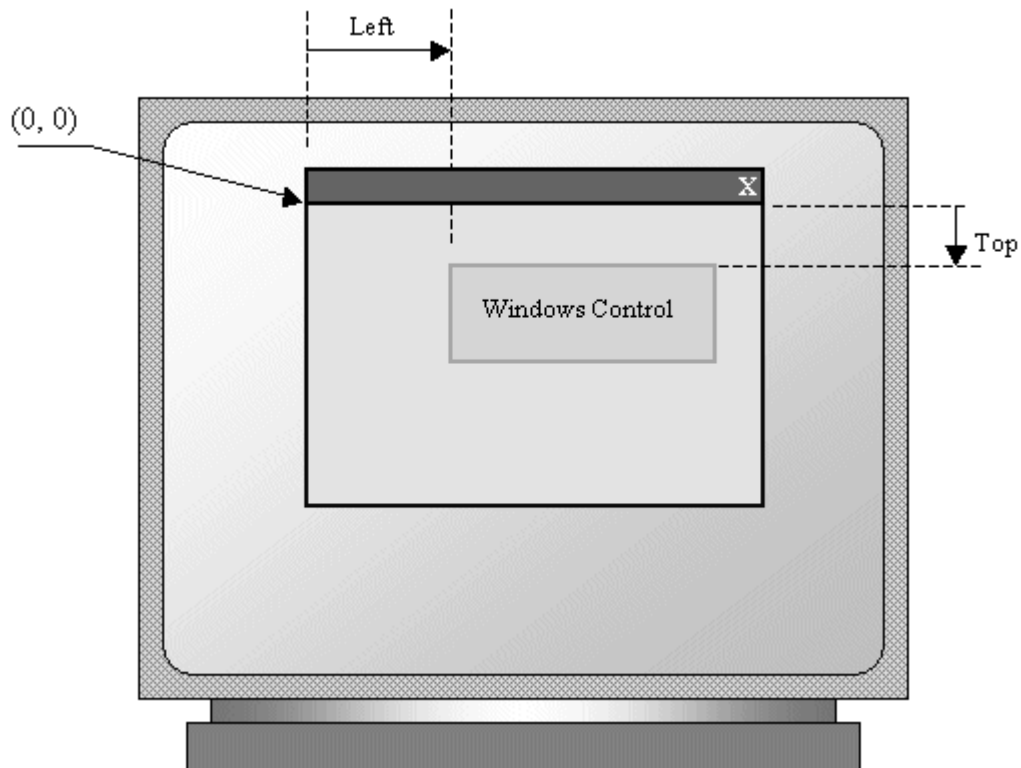


If a control is positioned on a form, its location uses a coordinate system whose origin is positioned on the top-left section of the client area (just under the title bar). The x axis moves from the origin to the left. The y axis moves from the origin down:

Coalesce



The distance from the left border of the client area to the left border of the control is the **Left** property. The distance from the top border of the client area to the top border of the control is the **Top** property. These can be illustrated as follows:



To change the location of a control, at design time, click and drag it in the desired direction. Once you get to the necessary location, release the mouse. Alternatively, select it and, in the Properties

Coalesce

window, click the + button of the Location field. Type the Left value on the X field and type the Top value in the Y field.

To programmatically change the location of a control, call the Point constructor from the Drawing namespace of the System with the desired value. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    Location = new System.Drawing.Point(200, 180);
}
```





To retrieve the location of a control, declare a Point variable and assign the Location property of the desired control to it. This would be done as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Drawing.Point Pt = Location;
}
```

Control Size

A control size is the amount of space it is occupying on the screen or on a form. It is expressed as its width and its height. The width of a control is the distance from its left to its right borders. The height is the distance from the top border of the control to its bottom border. When you create a form or add a control to a form, it assumes a default size, unless you draw the control while adding it.

To set the size of a control, at design time, select it. Then, position the mouse on one of its handles and drag in the desired direction. To assist you with this, the mouse cursor changes depending on the handle you grab. The cursors used are:

Cursor	Role
	Heightens or shortens the control
	Narrows or enlarges the control
	Resizes the control in the North-East <-> South-West direction
	Resizes the control in the South-East <-> North-West direction

Alternatively, to change the size of a control, select it and, in the Properties window, click the + button of the Size field. Then type the desired value for the **Width** and the desired value for the **Height**.

To programmatically change the size of a control, assign the desired values of either or both its Width and Height properties. Here is an example:

Coalesce

```
static void Main()
{
    Form1 Fm = new Form1();
    Fm.Size = new System.Drawing.Size(450, 320);
    Application.Run(Fm);
}
```

To find out the size of a control, declare a **Size** variable and assign it the **Size** property of the control.

The Bounding Rectangle of a Control

When a control is positioned on the screen, we saw that it can be located by its **Left** and its **Top** properties. A control also occupies an area represented by its **Width** and its **Height**. These four values can be grouped in an entity called the bounding rectangle and represented by the **Bounds** property.

Control Text or Caption

Some controls must display text to indicate what they are used for. Such a text can also be referred to as a caption. Some controls may not need this text and some others should or must display it. For a form, this text displays on the title bar. For most other controls, it displays in the middle of their body. If you create a form, it automatically gets text on its title bar. If you add a control to a form, it automatically gets a default text if that control is supposed to display it.

To change the text of a control, select it. Then, in the Properties window, click the **Text** field and type the desired value. At design time, you can only provide a static type of text to the control. If you want text that can be changed in response to another action, you must change it programmatically.

To programmatically change the text of a control, assign a string to its **Text** property. Here is an example:

```
static void Main()
{
    Form1 Fm = new Form1();
    Fm.Text = "Employment Application";
    Fm.Size = new System.Drawing.Size(450, 320);
    Application.Run(Fm);
}
```

The text can also be created from an expression or an operation.

Control Visibility

In order to use a control, the user must be able to see it. When a control is not visible, it is hidden. This aspect is controlled by the control's visibility.

Coalesce

The visibility of a control is controlled by the **Visible** property, which is a Boolean type. To make a control visible, at design time, set its **Visible** property to **True**. To hide it, set this property to **False**.

To programmatically display or hide a control, assign the **true** or **false** value to its **Visible** property. If you assign a **true** value to this property, if the control was hidden, it would become visible. If the control was already visible, nothing would happen. On the other hand, if you assign a **false** value to a visible control, it would become hidden. If it was already hidden, nothing would happen.

To find out whether a control is visible or hidden at one time, check the state of its **Visible** property, whether it is **true** or **false**.

Control Availability

When a control is visible, in order to use it, it must allow it to the user. A control is referred to as enabled if it can receive input from the user. For example, if it is a control that can be clicked, at a certain time, it must be "clickable". If it is not, clicking it would not make any difference. This aspect is controlled by the **Enabled** property.

To make a control available to the user, set its **Enabled** property to **True**. To prevent the user from interacting with a control, set its **Enabled** property to **False**.

To programmatically enable or disable a control, assign a value of true or false to its **Enabled** property. To find out whether a control is enabled or disabled, check its **Enabled** state by comparing to a **true** or **false** value.

A control that is enabled displays in its normal or regular appearance. A control that is disabled sometimes appears dimmed.

Tab Ordering

A user can navigate through controls using the Tab key. When that key has been pressed, the focus moves from one control to the next. By their designs, not all controls can receive focus and not all controls can participate in tab navigation. Even controls that can receive focus must be explicitly included in the tab sequence.

At design time, the participation to tab sequencing is controlled by the **TabStop** property on the Properties window. Fortunately, every control that can receive focus is also configured to have this Boolean property set to true. If you want to remove a control from this sequence, set its **TabStop** value to false.

If a control has the **TabOrder** property set to true, to arrange the navigation order of controls, at design time, change the value of its **TabIndex** field. The value must be a positive short integer.

Chapter 20

Methods of Managing Controls

Control Creation

In the previous lesson, we saw that the easiest way to create a control is by selecting it from the Toolbox and adding it to the form. If for any reason you cannot visually add a control, you can programmatically create it.

The classes used to manage controls of the .Net ensemble are created in the **System** namespace. Inside of the **System** namespace, there is a **Windows** namespace. Inside of the **Windows** namespace, there is the **Forms** namespace. All Windows control available in the .Net programming studio are created in the **System.Windows.Forms** namespace.

Every .Net control is based on a class. Every one of these control classes has at least a default constructor. You can use this constructor to dynamically create a control. The general syntax used is:

ClassName VariableName;

Here is an example:

```
System.Windows.Forms.Button btnSubmit;
```

You must use the **new** operator and call its default constructor to initialize the control. After calling its constructor, you can initialize any of its properties as necessary. Here is an example:

```
public class Form1 : System.Windows.Forms.Form
{
    System.Windows.Forms.Button btnSubmit;
```

Coalesce

```
/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1()
{
    btnSubmit = new System.Windows.Forms.Button();
    btnSubmit.Location = new System.Drawing.Point(88, 32);
    btnSubmit.TabIndex = 0;
    btnSubmit.Text = "Submit";
    Controls.Add(btnSubmit);
    ...
}
```

Focus

The focus is a visual aspect that indicates that a control is ready to receive input from the user. Various controls have different ways of expressing that they have received focus.

Buttons controls indicate that they have focus by drawing a dotted rectangle around their caption. A text-based control indicates that it has focus by displaying a blinking cursor. A list-based control indicates that it has focus when one of its items has a surrounding dotted rectangle.

To give focus to a control, the user can click it or press a key. To programmatically give focus to a control, call the **Focus()** method. Its syntax is:

```
bool Focus();
```

Chapter 21


Controls Events

Introduction

An application is made of various objects or controls. During the lifetime of the application, its controls regularly send messages to the operating system to do something on their behalf. These messages are similar to human messages and must be processed appropriately. Also, most of the time, more than one application is running on the computer. The controls of such an application also send messages to the operating system. As the operating system is constantly asked to perform these assignments, because there can be so many requests presented unpredictably, the operating system leaves it up to the controls to specify what they want, when they want it, and what behavior or result they expect.

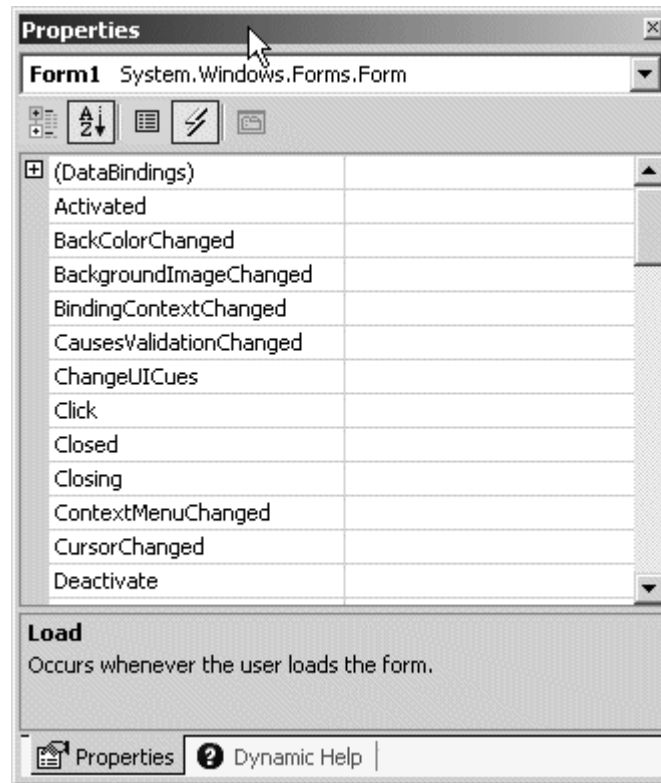
An event is the result of a message that the control needs to be dealt with. The message is what needs to be processed. The event is the action of processing the message. To process a message, it must provide at least two pieces of information: a value that carries the message and another value that indicates a method that would process the message. Both values are passed as the arguments to the event. The first must be a class derived from **EventArgs**. The second must be a delegate.

Event Implementation

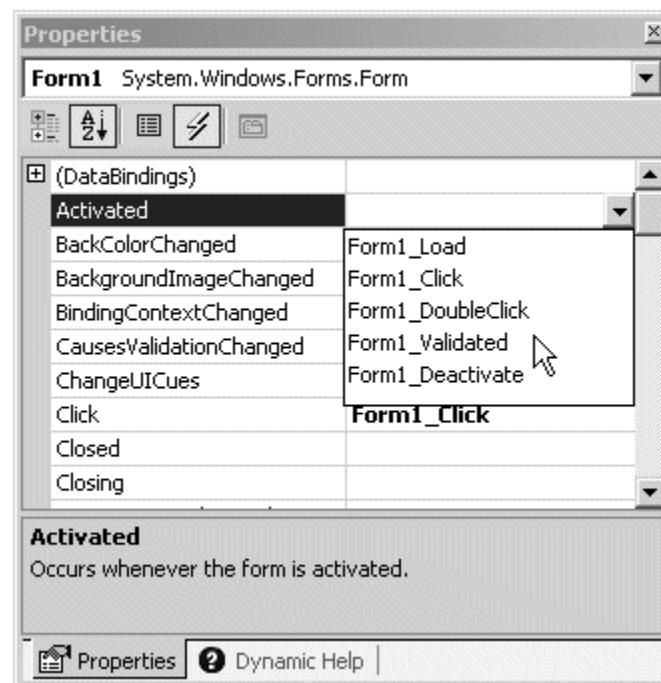
Although there are different ways you can implement an event, there are two main ways you can initiate its coding. If the control has a default event and if you double-click it, the compiler would initiate the default event. Another technique you can use is to click the Events button  in the

Coalesce

Properties window when the form itself or the control is selected on the form. This would display a list of the events associated with a control:



The list is divided in two columns. The name of each event is displayed on the left side. You can click the name of an event to reveal a combo box. If a similar event has already been written, you can click the arrow of the combo box and select it from the list:



Coalesce

Similar events are those that share a behavior. Otherwise, to initiate an event double-click either the name of the event or the right field to the desired event.

When an event has been initiated, you would be transported to the Code Editor and the cursor would be positioned in the body of the event, ready to receive your instructions. To customize an event, the compiler divides its structure in three sections.

Control Painting

While an application is opened on the screen or it needs to be shown, the operating system must display its controls. To do this, the controls colors and other visual aspects must be retrieved and restored. This is done by painting it (the control). If the form that hosts the controls was hidden somewhere such as behind another window or was minimized, when it comes up, the operating system needs to paint it.

When a control gets painted, it fires the **Paint()** event. The syntax of the **Paint()** event is:

```
void PaintEventHandler(object sender, PaintEventArgs e);
```

This event is a **PaintEventArgs** type. To process the painting message, a **PaintEventArgs** argument is passed to the **PaintEventHandler** delegate. The **PaintEventArgs** parameter provides information about the area to be painted and the graphics object to paint.

Control Resizing

When using an application, one of the actions a user can perform on a form or a control is to change its size, provided the object allows it. Also, some time to time, if possible, the user can minimize, maximize, or restore a window. Whenever any of these actions occur, the operating system must keep track of the location and size of a control. For example, if a previously minimized or maximized window is being restored, the operating system must remember where the object was previously positioned and what its dimensions were.

When the size of a control has been changed, it fires the **Resize()** event, which is a **EventArgs** type.

Keyboard Messages

A keyboard is a hardware object attached to the computer. By default, it is used to enter recognizable symbols, letters, and other characters on a control. Each key on the keyboard displays a symbol, a letter, or a combination of those, to give an indication of what the key could be used for.

The user typically presses a key, which sends a signal to a program. The signal is analyzed to find its meaning. If the program or control that has focus is equipped to deal with the signal, it may produce the expected result. If the program or control cannot figure out what to do, it ignores the action.

Each key has a code that the operating system can recognize.

The Key Down Message

When a keyboard key is pressed, a message called **KeyDown** is sent. **KeyDown** is a **EventArgs** type.

The Key Up Message

Coalesce

As opposed to the key down message that is sent when a key is down, the **KeyUp** message is sent when the user releases the key. Like **KeyDown**, **KeyUp** is a **KeyEventArgs** type.

The Key Press Message

When the user presses a key, the **KeyPress** message is sent. Unlike the other two keyboard messages, the key pressed for this event should (must) be a character key. **KeyPress** produces the **KeyPressEventArgs** type.

The Key argument must be a letter or a recognizable symbol. Lowercase alphabetic characters, digits, and the lower base characters such as ; , ' [] - = / are recognized as they are. For an uppercase letter or an upper base symbols, the user must press Shift + the key. The character would be identified as one entity. This means that the symbol % typed with Shift + 5 is considered as one character.

Mouse Messages

The mouse is another object that is attached to the computer allowing the user to interact with the machine. The mouse and the keyboard can each accomplish some tasks that are not normally available on the other and both can accomplish some tasks the same way.

The mouse is equipped with two, three, or more buttons. When a mouse has two buttons, one is usually located on the left and the other is located on the right. When a mouse has three buttons, one is in the middle of the other two. The mouse is used to select a point or position on the screen. Once the user has located an item, which could also be an empty space, a letter or a word, he or she would position the mouse pointer on it.

To actually use the mouse, the user would press either the left, the middle (if any), or the right button. If the user presses the left button once, this action is called Click. If the user presses the right mouse button, the action is referred to as Right-Click. If the user presses the left button twice and very fast, the action is called Double-Click.

The Mouse Down Message

Imagine the user has located a position or an item on a document and presses one of the mouse buttons. While the button is pressed and is down, a button-down message is sent. This event is called **MouseDown** and is of type **MouseEventArgs**. To implement this event, a **MouseEventArgs** argument is passed to the **MouseEventHandler** event implementer. **MouseEventArgs** provides the necessary information about the event such as what button was clicked, how many times the button was clicked, and the location of the mouse.

The Mouse Up Message

After pressing a mouse button, the user usually releases it. While the button is being released, a button-up message is sent and it depends on the button, left or right, that was down. The event produced is **MouseUp**.

Like the **MouseDown** message, the **MouseUp** event is of type **MouseEventArgs** which is passed to the **MouseEventHandler** for processing.

Coalesce

The Mouse Move Message

Whenever the mouse is positioned and being moved on top of a control, a mouse event is sent. This event is called **MouseMove** and is of type **MouseEventArgs**.

Custom Message Implementation

To process Windows messages, you can use **WndProc()** method. Its syntax is:

```
virtual void WndProc(ref Message m);
```

In order to use this method, you must override it in your own class.

Coalesce

Chapter 22

Controls Containers

The Form

The form, implying the dialog box, is the primary object used to host, hold, or carry other controls of the application. Since we have had a thorough review of forms and dialog boxes already, we will not spend any more time on them.

The Panel Control

Introduction

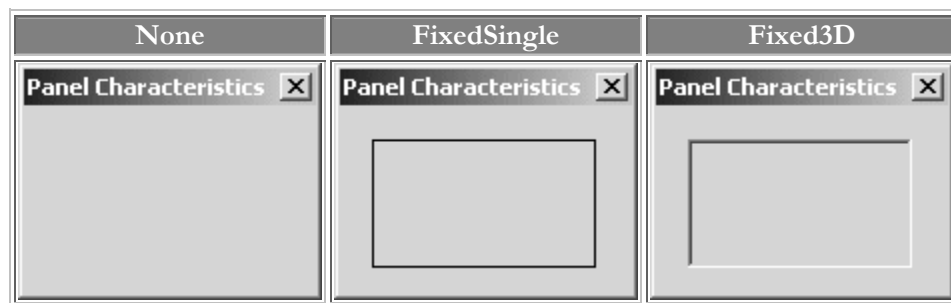
A panel is a visible rectangular object that can provide two valuable services for application design. A panel allows you to design good-looking forms by adjusting colors and other properties. A panel is also a regularly used control container because it holds and carries controls placed on it. When a panel moves, it does so with the controls placed on it. When a panel is visible, the controls placed on it can be visible too, unless they have their own visibility removed. When a panel is hidden, its child controls are hidden too, regardless of their own visibility status. This property also applies to the availability.

Panels are not transparent. Therefore, their color can be changed to control their background display. A panel is a complete control with properties, methods, and events.

Characteristics of a Panel

Unlike the form, during design, a panel must primarily be positioned on another container which would be a form or another panel. To add a panel to a container, you can click the Panel button from the Toolbox and click in the desired location on the container.

By default, a panel object is drawn without borders. If you want to add borders to it, use the **BorderStyle** property. It provides three values: None, FixedSingle, and Fixed3D and their effects are as follows:



A panel can be used as a button, in which case the user would click it to initiate an action.

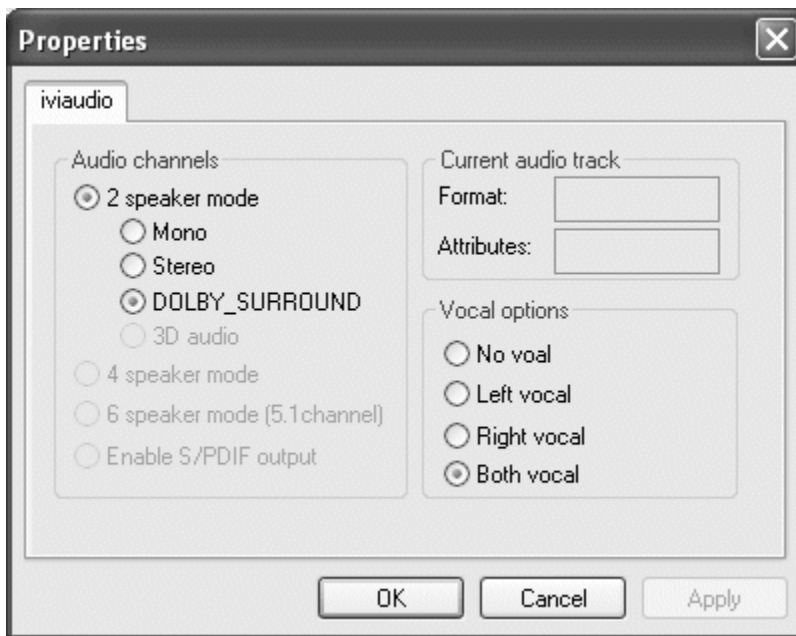
A property that is highly used on panels (and forms) is the Color. If you change the

Coalesce

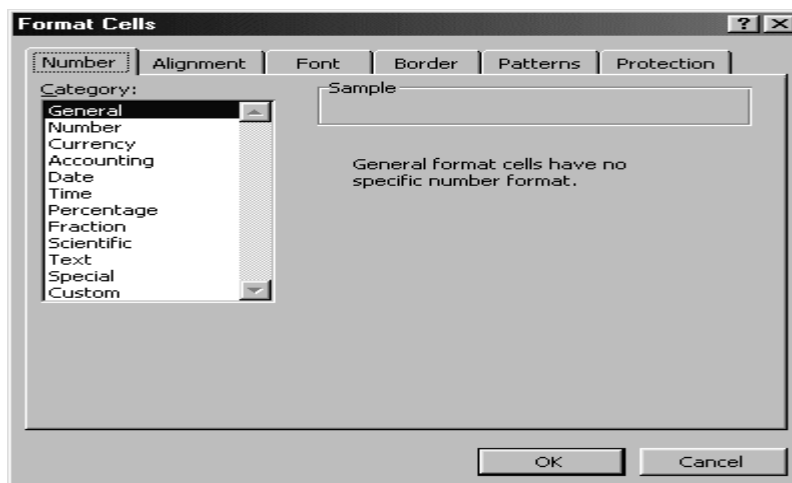
BackColor property, the new color would cover the face of the panel. Property Sheets and Property Pages

As your application becomes crowded with various controls, you may find yourself running out of space. To solve such a problem, you can create many controls on a form or container and display some of them only in response to some action from the user. The alternative is to group controls in various containers and specify when the controls hosted by a particular container must be displayed. This is the idea behind the concept of property pages.

A property page is a control container that appears as a form or a frame. A property page can appear by itself. Here is an example:

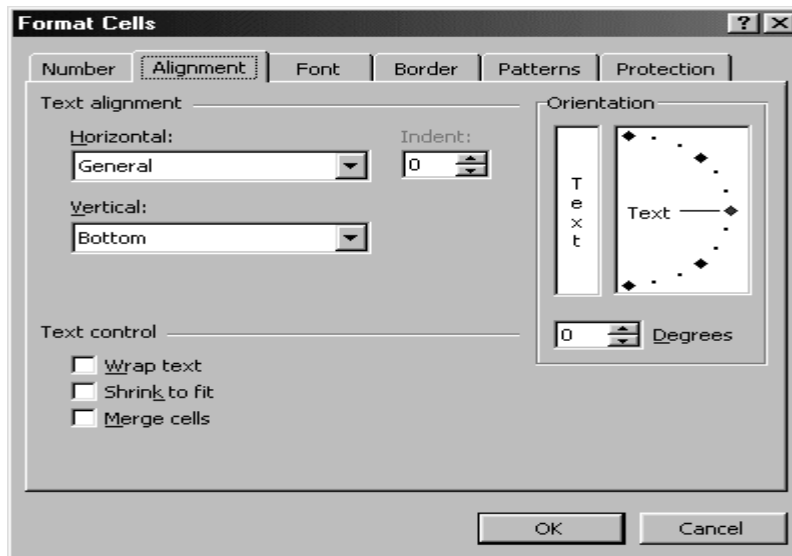


In most other cases, a property page appears in a group with other pages. It functions like a group of pieces of paper placed on top of each other. Each piece is represented by a tab that allows the user to identify them:



Coalesce

To use a property page, the user clicks the header, also called a tab, of the desired page. This brings that page up and sends the others in the background:



If the user needs access to another page, he or she can click the desired tab, which would bring that page in front and send the previously selected to the back.

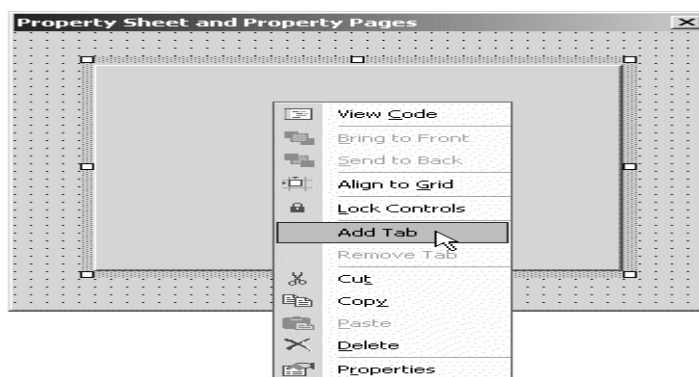
The pages are grouped and hosted by an object called a property sheet. Like a form, the pages of a property sheet are simply used to carry or hold other controls. The major idea of using a property sheet is its ability to have many controls available in a relatively smaller container.

Property Sheet Creation

Property pages of a property sheet are also referred to as tab controls. In a .Net application, a property sheet is created using the **TabControl** control or class which serves as the property sheet. To implement a property sheet in your application, on the Toolbox, you can click the **TabControl** button and click in the form that would be used as the property sheet's platform.

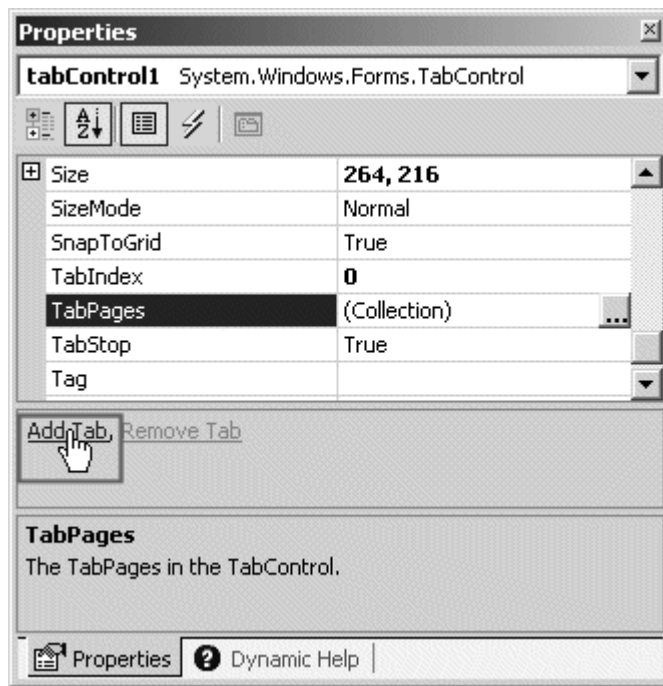
To create or add the actual property pages of the property sheet,


- You can right-click the **TabControl** control and click Add Tab:

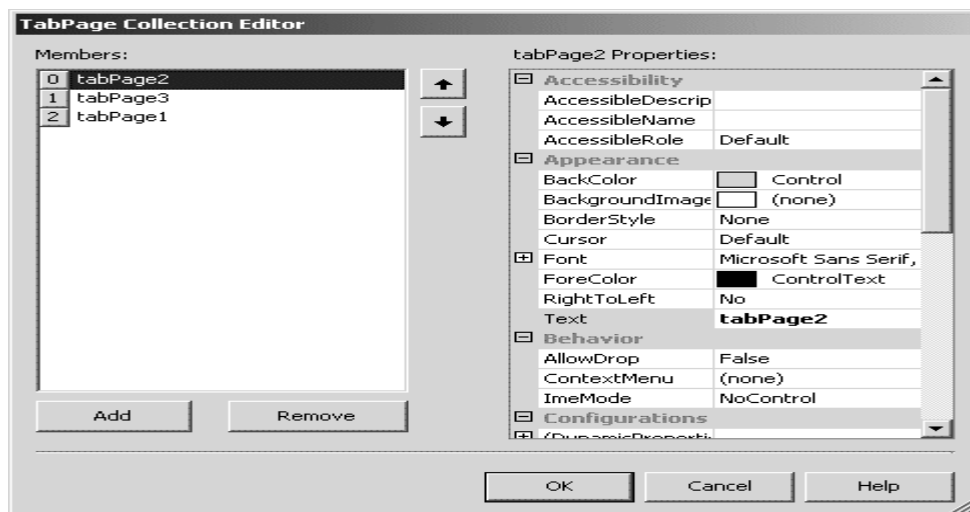


Coalesce

- While the **TabControl** control is selected on the form, in the lower section of the Properties window, you can click the Add Tab link:



- While the **TabControl** control is selected on the form, in the Properties window, click the **TabPage** field and click its ellipsis button  to display the **TabPage Collection Editor** dialog box that allows you create and configure each page:



If you have added a page by mistake or you don't want a particular page anymore, you can remove it. To remove a property page, first click its tab to select it. Then,

- You can right-click the **TabControl** control and click **Remove Tab**

Coalesce

- While the undesired tab page is selected on the tab control, press Delete
- While the undesired tab page is selected on the tab control, in the lower section of the Properties window, you can click the Remove Tab link
- You can right-click the body of the undesired page and click Delete

Many of the effects you will need on pages have to be set on the **TabControl** and not on individual pages. This means that, to manage the characteristics of pages, you will change the properties of the parent **TabControl** control. At any time, whether working on the **TabControl** control or one of its tab pages, you should first know what object you are working on by selecting it.

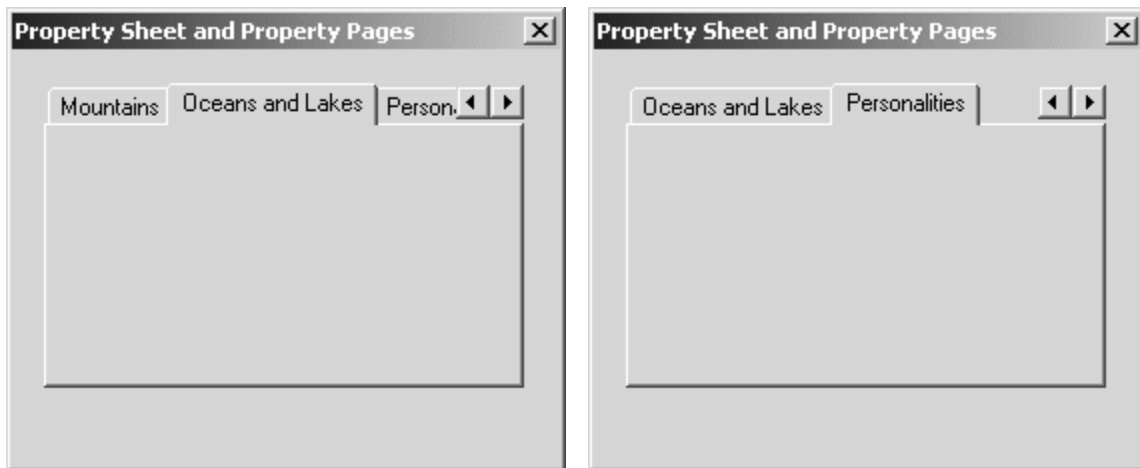
To select the **TabControl** control itself, you have two main options:

- Any time you want to select the **TabControl**, click an unoccupied area on the right side of the most right tab
- While one tab page is selected, click another tab

If you want the property sheet to occupy the whole form or to occupy a section of it, you can specify this using the **Dock** property.

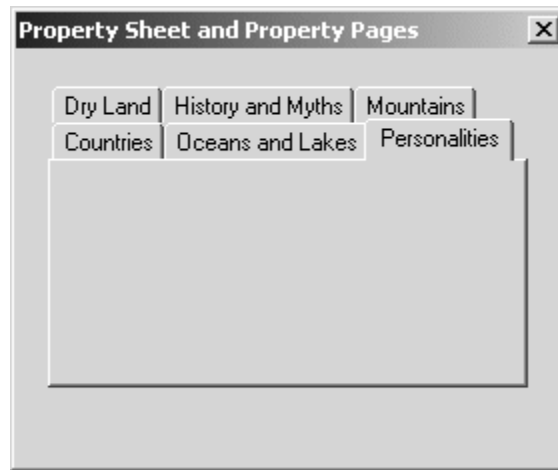
If you want the property pages to have bitmaps on their tabs, you should first add a series of images using an **ImageList** control and then assign that control to the Images property of the **TabControl** object.

If you have many property pages and the width of the PageControl cannot show all tabs at the same time, the control would add two navigation arrows to its top right corner to let the user know that there are more property pages:



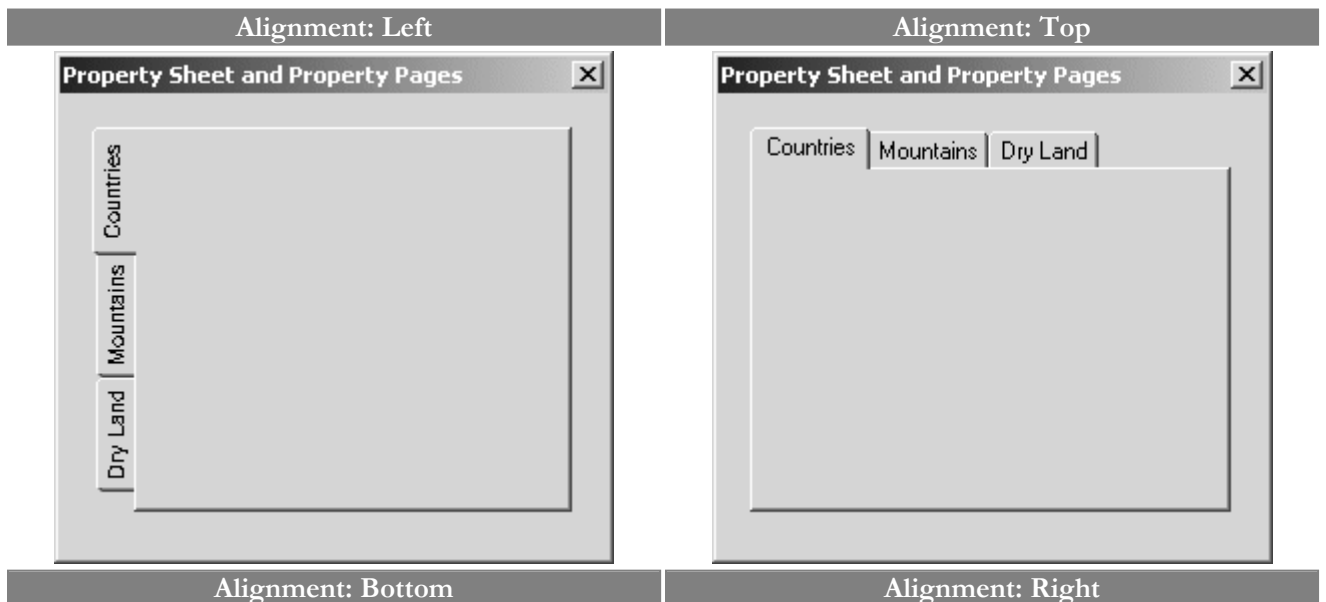
By default, the navigation buttons would come up because the control uses a property that controls their availability. If you do not want the navigation arrows, you can set the **MultiLine** property of the **TabControl** control to true. This would create cascading tabs and the user can simply select the desired property page from its tab:

Coalesce

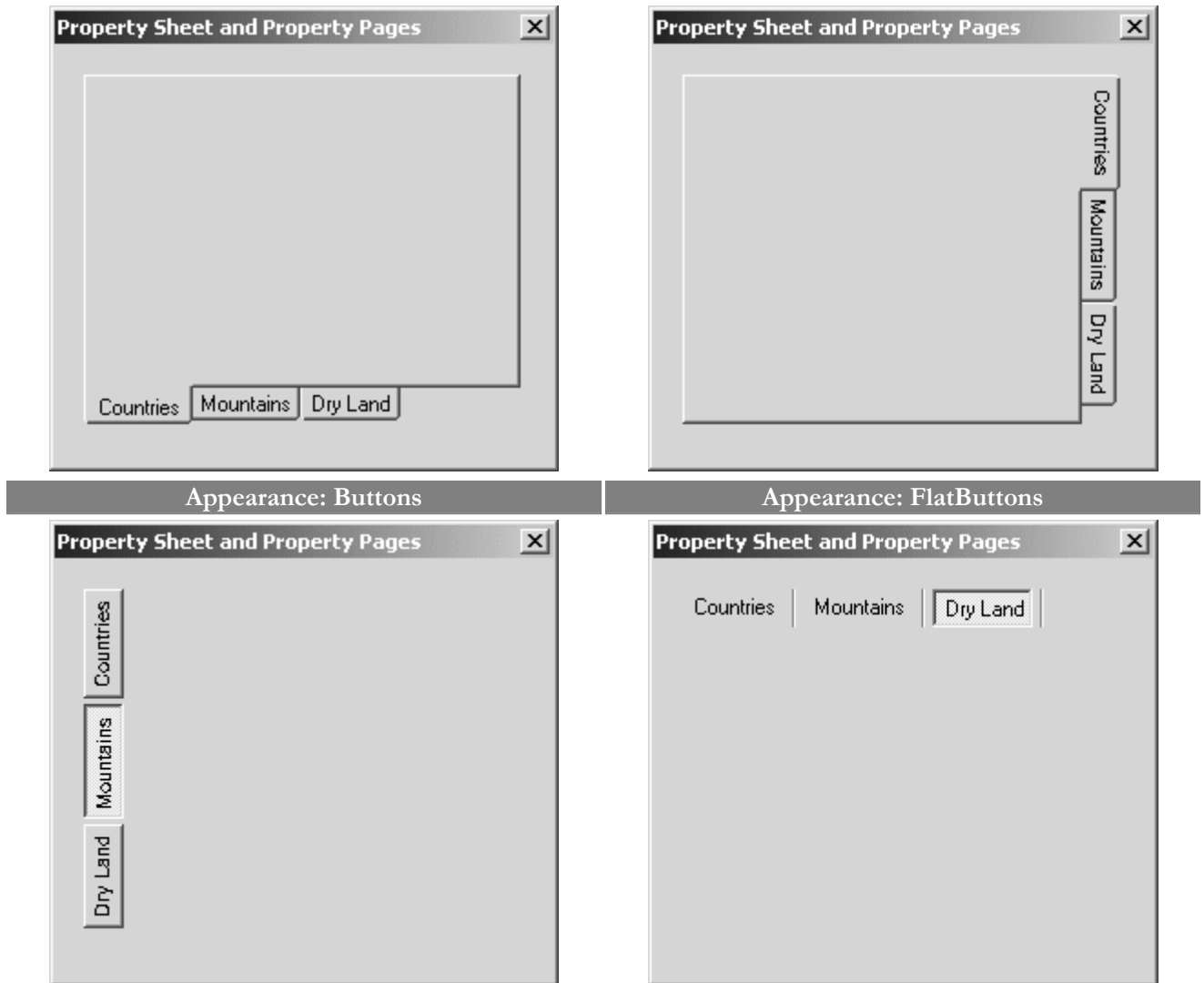


As you are adding pages to a **TabControl** control, the tabs of the pages are positioned on the top border of the **TabControl** area. You can reposition them to the left, the right, or the bottom borders of the control. The placement of the tab is set using the **Alignment** property of the **TabControl**. The possible values of this property are **Top**, **Left**, **Right**, and **Bottom**:

If you want to create a discrete property sheet and do not want to display the traditional tabs, you can replace the tabs with buttons. This is controlled by the **Appearance** property that presents three options: **Normal** (the default), **Buttons**, and **FlatButtons**. The **Normal** and the **Buttons** values can be used on all four views. The **FlatButtons** option is available only if the **Alignment** property is set to **Top**.



Coalesce



After adding the **TabControl** control to your form and after adding one or more tab pages, the property pages are created where the **TabPage** control is positioned and its dimensions are used by the eventual property pages. This means that, if you want a different position, a smaller or larger property sheet, you must modify the dimensions of the **TabControl** control and not those of the tab pages, even though each tab page has a **Location** property and dimensions (the **Size** property).

Like all other controls, the names of property pages are cumulative. As you add them, the first would be named `tabPage1`, the second would be `tabPage2`, etc. If you plan to programmatically refer to the tab pages, you should give them more explicit names. As done with any other control, to set the name of a property page, after selecting it, in the Properties window, change the value of the **Name** property.

Probably the first obvious characteristic of a property page is the word or string that identifies it to the user. That is, the string that displays on the tab of a property page. This is known as its title or caption. By default, the captions are set cumulatively as the pages are added. Usually you will not use these titles because they are meaningless. To display a custom title for a property page, first select it and, on the Properties window, change the value of the **Text** property. You can also change the title of a property page programmatically, for example, in response to an intermediary action. To change the title of a page, assign a string to its **Text** property. Here is an example:

Coalesce

```
private void Form1_Load(object sender, System.EventArgs e)
{
    Me.tabPage3.Text = "Work Experience";
}
```

If you had associated an **ImageList** control with the **TabControl** object and you want to use images on the title of a property page, specify the desired image using the **ImageIndex** property. You can use the images on some or all property pages.

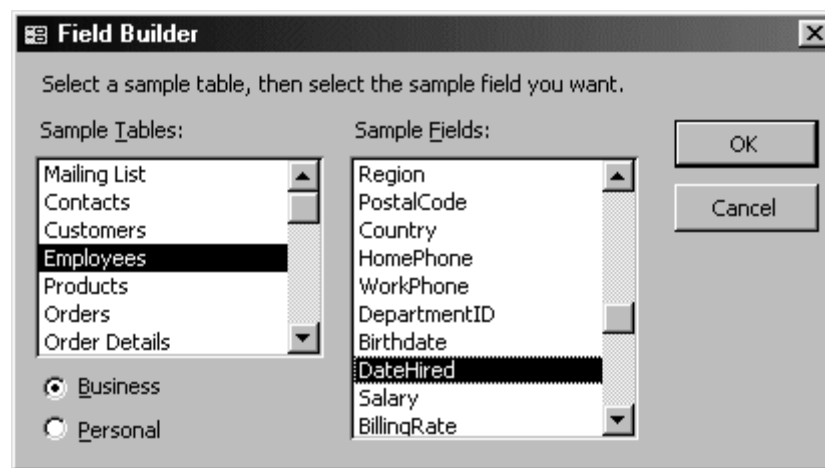
Coalesce

Chapter 23


Dialog Boxes

Regular Dialog Boxes

A dialog box is a form with particular properties. Like a form, a dialog box is referred to as a container. It is the primary interface of user interaction with the computer. By itself, a dialog box means nothing. The controls it hosts accomplish the role of dialog between the user and the machine. Here is an example of a dialog box:



A dialog box has the following characteristics:

- It is equipped with the system Close button . As the only system button, this button allows the user to dismiss the dialog and ignore whatever the user would have done on the dialog box.
- It cannot be minimized, maximized, or restored. A dialog box does not have any other system button but Close.
- It is usually modal, in which case the user is not allowed to continue any other operation on the same application until the dialog box is dismissed.
- It provides a way for the user to close or dismiss the dialog. Most dialog boxes have an OK and a Cancel buttons, although this depends on the application developer. When the dialog has the OK and the Cancel buttons, the OK button is configured to behave as if the user had pressed Enter. In that case, whatever the user had done would be acknowledged and transferred to the hosting dialog box, window, or application. Pressing Esc applies the same behavior as if the user had clicked Cancel.

Coalesce

Dialog Box Creation

There are two main actions you can perform on a form to qualify it as a dialog box (but normally, these are only suggestions, not rules). Based on the Microsoft Windows design and standards, to create a dialog box:

- You should set a form's **FormBorderStyle** property to **FixedDialog**. Setting this property to true prevents the dialog box from being resized at run time, which respects one of the rules of a dialog box.
- Set the **MaximizeBox** property to False to remove the Maximize buttons
- Set **MinimizeBox** property to False to remove the system Minimize button
When both the **MaximizeBox** and the **MinimizeBox** properties are set to False, the system Maximize and Minimize buttons are removed from the dialog box
- You should/must provide a way for the user to close the dialog box. A dialog box should have at least one button labeled OK. This button allows the user to acknowledge the message of the dialog box and close it by clicking the button. If the user press Enter, the dialog box should also be closed as if the OK button was clicked. To fulfill this requirement, from the Toolbox, you can click the Button control and click the dialog box.

Often the user will be presented with various options on a dialog box and may be asked to make a decision on the available controls. Most of the time, if you are creating such a dialog box, besides the OK button, it should also have a Cancel button. The OK button should be the default so that if the user presses Enter, the dialog box would be closed as if the user had clicked OK. Clicking OK or pressing Enter would indicate that, if the user had made changes on the controls of the dialog box, those changes would be acknowledged and kept when the dialog box is closed. The changed values of the control would be transferred to another dialog box or form.

To fulfill this rule for the OK button, after adding the button(s), click the form to select it. Then, click its **AcceptButton** field in the Properties window. This reveals its combo box. Click its arrow and select the name of the button whose action would be performed when the user presses Enter.

The Cancel button is used to allow the user to dismiss whatever changes would have been made on the controls of the dialog box. The dialog box should also be configured so that if the user presses Esc, the dialog box would be closed as if the user had clicked Cancel.

To fulfill this rule for the Cancel button, after adding the buttons, click the form to select it. Then, in the Properties window, click its **CancelButton** field to display its combo box. Click the arrow of the **CancelButton** combo box and select the name of the button whose action would be performed when the user presses Esc. Make sure the button selected for the **AcceptButton** property is not the same as the one selected for the **CancelButton** property.

Besides the OK and the Cancel buttons, a dialog box can be created with additional buttons such as Finish or Help, etc. It depends on its role and the decision is made by the application developer.

Modal and Modeless Dialog Boxes

Modal Dialog Boxes

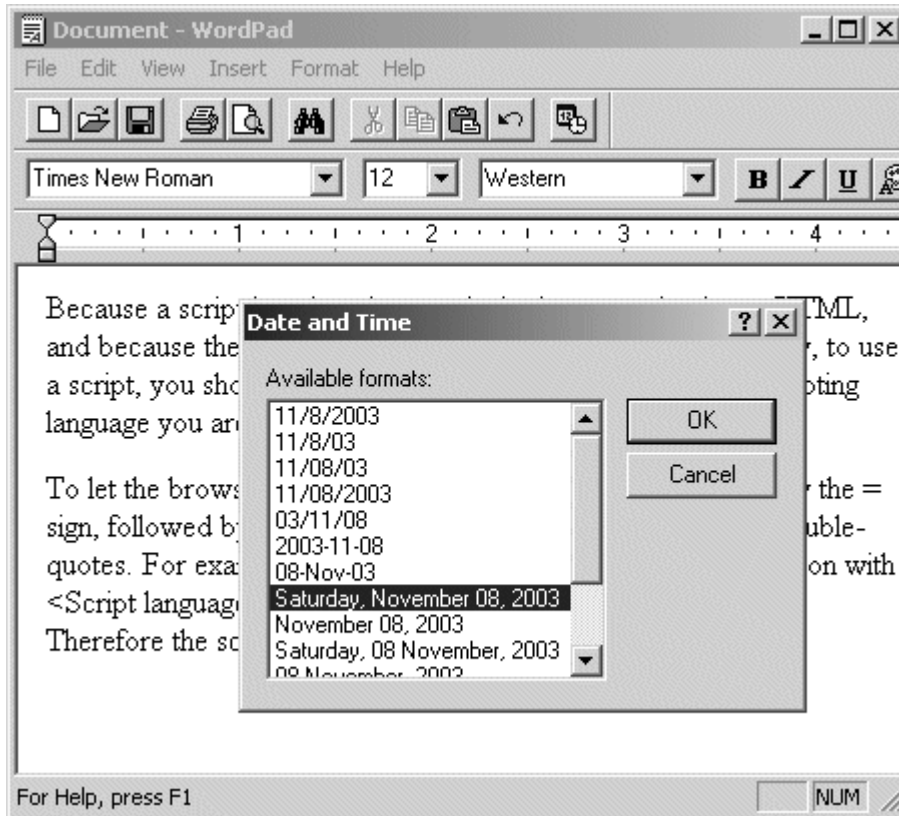
There are two types of dialog boxes: modal and modeless. A Modal dialog box is one that the user must first close in order to have access to any other form or dialog box of the same application.

Coalesce

One of the scenarios in which you use a dialog box is to create an application that is centered around one. In this case, if either there is no other form or dialog box in your application or all the other forms or dialog boxes depend on this central dialog box, it must be created as modal. Such an application is referred to as dialog-based.

Some applications require various dialog boxes to complete their functionality. When in case, you may need to call one dialog box from another and display it as modal.

The **Date and Time** dialog box of WordPad is an example of a modal dialog box: if opened, the user must close it in order to continue using WordPad.



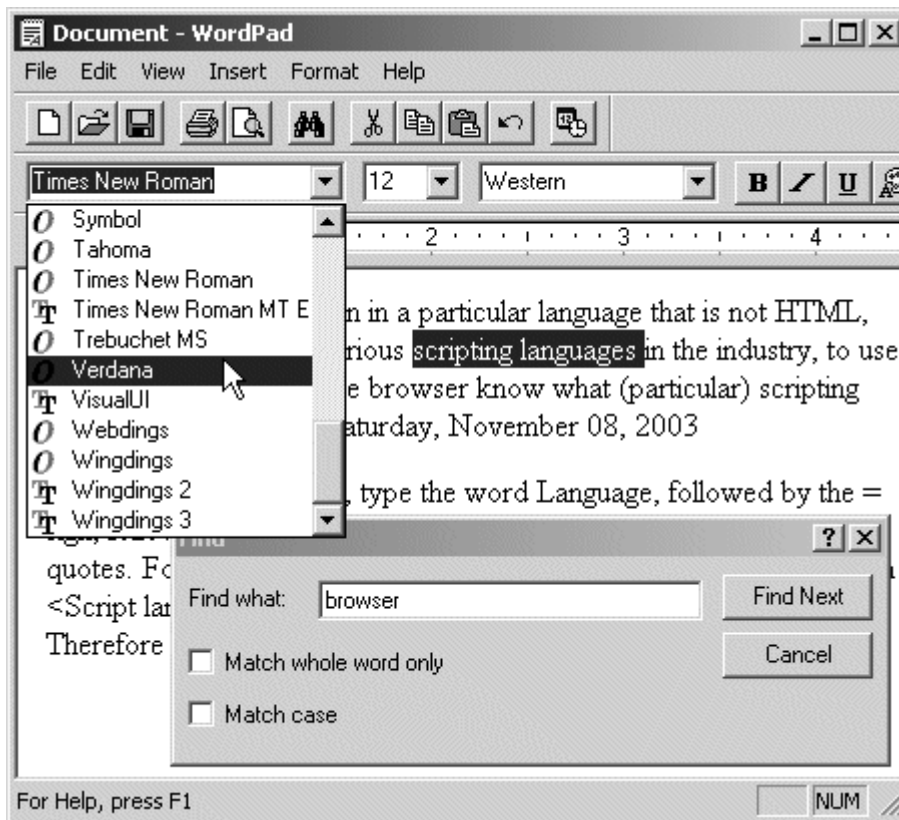
After creating a form and configuring it as a dialog box, to call it as modal, use the **ShowDialog()** method.

Modeless Dialog Boxes

A dialog box is referred to as modeless if the user does not have to close it in order to continue using the application that owns the dialog box:

The **Find** dialog box of WordPad (also the **Find** dialog box of most applications) is an example of a modeless dialog box. If it is opened, the user doesn't have to close in order to use the application or the document in the background.

Coalesce



A modeless dialog box should/must not display its button on the task bar. The user should still know that the dialog box is opened. To make the presence of a modeless dialog box obvious to the user, it typically displays on top of its host application until the user closes it.

A modeless dialog box is created from a form but it should look like a regular dialog box or a tool window. Therefore, to create a modeless dialog box, set the **FormBorderStyle** property to an appropriate value such as **FixedSingle**, **FixedToolWindow**, **Sizable** or **SizableToolWindow**. Also, set its **ShowIn Taskbar** property to **False**.

Chapter 24

Collections-Based Controls

The Main Menu

A menu of a computer program is a list of actions that can be performed on that application. To be aware of these actions, the list must be presented to the user upon request.

A menu is considered a main menu, when it carries most of the actions the user can perform on a particular application. Such a menu is positioned on the top section of the form in which it is used. By design, although a main menu is positioned on a form, it actually belongs to the application.

A main menu is divided in categories of items and each category is represented by a word. On the Visual Studio's IDE, the categories of menus are File, Edit, Project, Window, etc. To use a menu, the user first clicks one of the words that displays on top. Upon clicking, the menu expands and displays a list of items that belong to that category.

There is no strict rule on how a menu is organized. There are only suggestions. For example, actions that are related to file processing, such as creating a new file, opening an existing file, saving a file, printing the open file, or closing the file usually stay under a category called File. In the same way, actions related to viewing documents can be listed under a View menu.

Main Menu Creation

The main menu is implemented through the MainMenu class. To create a main menu, on the Toolbox, you can click the MainMenu button and click the form that will use the menu. After clicking the form, an empty menu is initiated, waiting for you to add the necessary menu item.

To create a menu category that starts a list of items that belong to the menu object, click the Type Here line and type the desired item then press Enter.

If you look at the main menu of Visual Studio, you would see that a letter on each menu is underlined. This letter allows the user to access the menu using a keyboard. For example, if the letter e is underline in a File menu as in File, the user can access the File by pressing the Alt then the E keys. To create this functionality, choose a letter on the menu item and precede it with the & character. For example, &File would produce File.

A shortcut is a key or a combination of keys that the user can press to perform an action that would also be performed using a menu item. When creating a menu, to specify a shortcut, use the **Shortcut** field in the Properties window.

If you have used applications like Microsoft Word or Adobe Photoshop, you may know that they don't show all of their shortcuts on the menu. If you want to hide a shortcut, after specifying it, in the Properties window, set the **ShowShortcut** property to **False**.

1. Start a new Windows Forms Application (.NET) named **Menus1**
2. Change the form's Text to **Employees Related Issues**
3. On the Toolbox, click the MainMenu button and click on the form
4. On the form, click Type Here, type **&Staff** and press Enter

Coalesce

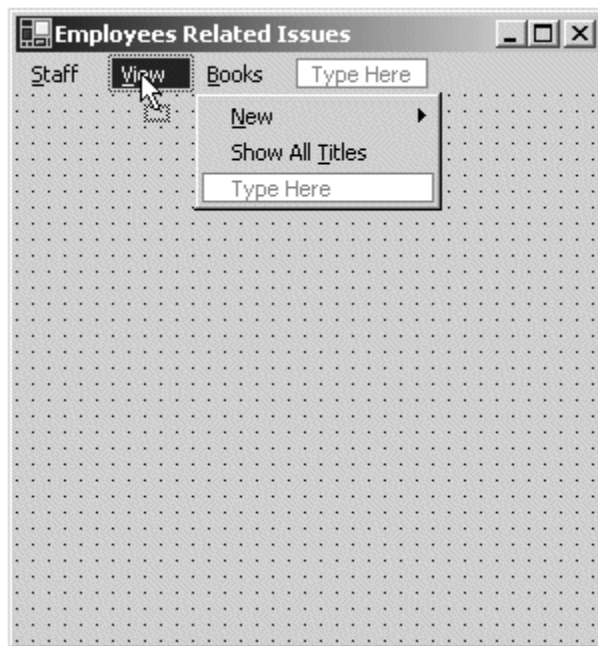
5. On the form, click Staff and click the Type Here box under it
6. Type **&New Hire**
7. In the Properties window, click **Shortcut**. Click the arrow of the Shortcut right field and select **CtrlN**
8. On the form, click the Type Here line under New Hire and type **&Records**
9. Using the Properties window, set the **Shortcut** to **CtrlR**
10. On the form, click the item under Records and type **Time &Sheet...**
11. Set its **Shortcut** to **CtrlM**
12. Set its **ShowShortcut** property to **False**
13. On the form, click Records and press Insert
14. Type **Searc&h...**
15. To add a separator, click the Type Here line under Time Sheet
16. Type - and press Enter
17. Click the empty item under the previously added separator line, type **E&xit** and press Enter
18. To move the Search item, on the form , click and drag Search down:



19. When it is positioned under Time Sheet item, release the mouse.
20. To start a new menu category, click the Type Here box on the right side of Staff
21. Type **&Books** and press Enter
22. Click Books and click the empty item under it
23. Type **&New** and press Enter
24. Type **Show All &Titles** and press Enter
25. To create a submenu, click New and click the Type Here box on its right

Coalesce

26. Type **&Title** and press Enter
27. Type **&Author** and press Enter
28. Type **&Category** and press Enter
29. Click the Type Here box on the right side of Books. Type **&View** and press Enter
30. Click View and click the empty box under it
31. Type **&Toolbar** and press Enter
32. Type **&Status Bar** and press Enter
33. Click Toolbar to select it
34. To put a check mark on a menu item, on the Properties window, set the **Checked** property to **True**
35. Also, for the Status Bar item, set the **Checked** property to **True**
36. To move the whole View menu category, click and drag View to the left:



37. When it is positioned between Staff and Book, release the mouse

Coding a Main Menu Item

If you create a menu as we have just done, to write code for one of the menu items, you can double-click the menu item. This would open the Click event of the menu item in the Code Editor and you can start writing the desired code for that item.

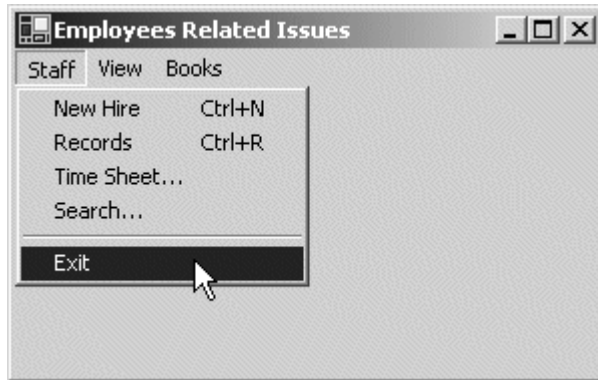
1. On the form, click Staff and double-click Exit
2. Implement the event as follows:

```
private void menuItem7_Click(object sender, System.EventArgs e)
{
```

Coalesce

```
        this.Close();  
    }  
}
```

3. Test the form



4. To close it, click Staff -> Exit

Popup or Context Menus

A menu is considered, or qualifies as, popup if, or because, it can appear anywhere on the form as the programmer wishes. Such a menu is also referred to as context-sensitive because its appearance and behavior depends on where it displays on the form or on a particular control.

The first difference between a main menu and a popup menu is that a popup menu appears as one category or one list of items and not like a group of categories of menus like a main menu. Secondly, while a main menu by default is positioned on the top section of a form, a popup menu doesn't have a specific location on the form.

To use a popup menu, usually the user can right-click the section of the form or the object on which the menu is configured to appear.



A popup menu is based on the **ContextMenu** class. To visually create a popup menu, on the Toolbox, click the **ContextMenu** button and click on the form. Once you have a **ContextMenu** object, you can create its menu items. To do this, click the ContextMenu box to display the first Type Here line and configure the menu item as you would proceed with a menu item on the main menu.

Unlike a main menu, a popup menu provides a single list of items. If you want different popup menus for your form, you have two options. You can create various popup menus or programmatically change your single popup menu in response to something or some action on your form.

There is nothing particularly specific with writing code for a popup menu item. You approach it exactly as if you were dealing with a menu item of a main menu. You can write code for an item of a popup menu independent of any other item of a main menu. If you want an item of a popup menu to respond to the same request as an item of a main menu, you can write code for one of the menu items (either the item on the main menu or the item on the popup menu) and simply call its Click event in the event of the other menu item

1. On the Toolbox, scroll down, click the **ContextMenu** button and click on the form
2. On the form, click the ContextMenu box

Coalesce

3. Click the Type Here box. Type **&Font...** and press Enter
4. Click the empty box under Font to select it. Type **&Options** and press Enter
5. Type **&Properties** and press Enter
6. Type - and press Enter
7. Type **Finis&h** and press Enter
8. Click Finish
9. On the Properties window, click the Events button 
10. Click the Click field to display its combo box. Click the arrow of the right field and select the only item in the list, which corresponds to the Exit menu item of the main menu
11. Click an empty area on the form to select the form
12. In the Properties window, click the Properties button  and click the ContextMenu field
13. Click the arrow of the ContextMenu property and select contextMenu1
14. Execute the application



15. Right-click in the middle of the form and click Finish.

Coalesce

Toolbars

A toolbar is a Windows control that allows the user to perform some actions on a form by clicking a button instead of using a menu. What a toolbar provides is a convenient group of buttons that simplifies the user's job by bringing the most accessible actions as buttons so that, instead of performing various steps to access a menu, a button on a toolbar can bring such common actions closer to the user.

Toolbars usually display under the main menu. They can be equipped with buttons but sometimes their buttons or some of their buttons have a caption. Toolbars can also be equipped with other types of controls.

To create a toolbar, on the Toolbox, click the **ToolStrip** button and the form. By default, a toolbar is positioned on the top section of the form.

Like a form, a toolbar is only a container and does not provide much role by itself. To make a toolbar efficient, you should equip it with the necessary controls. To add controls to a toolbar, on the Properties window, click the ellipsis button of the **Buttons** field. This would open the **ToolStripButton Collection Editor**.

To add a control to a toolbar, in the **ToolStripButton Collection Editor**, click the **Add** button. The most common control used on a toolbar is a button. The types of buttons used are set using the **Style** property. A regular button has the **PushButton** style. The new button would appear empty.

A button on a toolbar is a rectangular object with a picture on top. The picture on the button should suggest what the button is used for but this is not always possible. Providing a picture is more important. The easiest way to provide pictures for your toolbar is by creating an image list and associating it with the toolbar. If a toolbar is given an image list, whenever you add a button, use the **ImageIndex** property to specify the image that would display on the button. You can keep adding buttons in this fashion.

A separator is a line that separates buttons (or controls) as groups of objects on a toolbar. To create a separator, after clicking **Add**, select the item and set its **Style** to **Separator**.

As a toolbar can be equipped with various types of controls, there are various types of buttons you can place on a toolbar. The types of buttons can be configured using the **Style** property on the Object Inspector.

1. To create a new popup menu, on the Toolbox, click the **ContextMenu** button and click the form
2. On the form, click **ContextMenu** and click **Type Here**
3. Type **&Staff** and press **Enter**
4. Type **&Customer** and press **Enter**
5. Type **&Book** and press **Enter**
6. Type **&Author** and press **Enter**
7. Under the form, click **contextMenu2** to select it. In the Properties window, click **Name**, type **mnuNewContext** and press **Enter**
8. To create an image list, on the Toolbox, click the **ImageList** button and click the form

Coalesce

9. While the new `imageList1` control is still selected, in the Properties window, click the ellipsis button of the `Images` field
10. In the Image Collection Editor, click the Add button
11. Using the Open dialog box, locate the `Drive:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Graphics\Bitmaps\Tbr_W95` folder
12. Click BACK and click Open
13. From the same folder and in the same way, add the following bitmaps MCR, UNGROUP, and UP1LVL
14. Click OK to close the Image Collection Editor
15. To create a new toolbar, on the Toolbox, click the `ToolBar` button and click on the form
16. On the Properties window, click the `ImageList` field and, from its combo box, select `imageList1`
17. Set the **Appearance** to **Flat**
18. To add the controls to the toolbox, in the Properties window, click the ellipsis button of the `Buttons` field
19. In the `ToolBarButton` Collection Editor, click the Add button
20. In the Properties section, click the **ImageIndex** field and, from its combo box, select 0
21. Click the Add button again and set its **ImageIndex** to 1
22. Click the Add button again and, while the new item is still selected, click the arrow of the `Style` field and select Separator
23. Click Add again. While the new item is still selected, in the Properties section click the arrow of the `Style` field and select **DropDownButton**
24. Set its **ImageIndex** to 3
25. Click the arrow of the `DropDownMenu` and select `mnuNewContext`
26. Click Add again and set the **ImageIndex** of this last button to 2
27. Click OK to close the `ToolBarButton` Collection Editor
28. To create a new popup menu, on the Toolbox, double-click **ContextMenu**
29. Change its Name to **mnuViewContext**
30. On the form, click Context Menu and click Type Here
31. Type **&Toolbar** and press Enter
32. Type **&Status Bar** and press Enter
33. Set the **Checked** property of each to **True**
34. Click in the body of the form to select it
35. On the main menu of the form, click View and double-click `Toolbar`
36. Implement the event as follows:

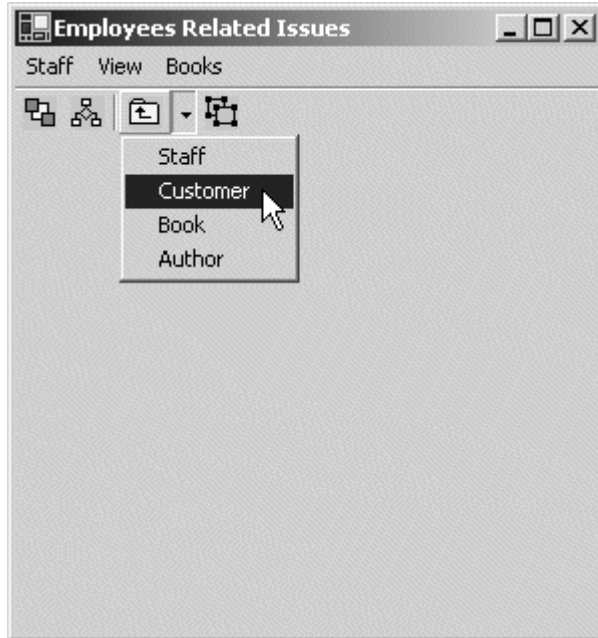
```
private void menuItem15_Click(object sender, System.EventArgs e)
{
```

Coalesce

```
this.toolBar1.Visible = !this.toolBar1.Visible;  
this.menuItem15.Checked = !this.menuItem15.Checked;
```

```
}
```

37. Test the application



38. Close it and return to MSVC

Toolbar Programming

The controls on a toolbar are controls by their owner, the toolbar. In order to write code for a control, you must first locate it. The buttons are located in a collection called Buttons. Each button can be located by calling the `IndexOf()` methods. Once this gives you access to a button, you can write its code as you see fit.

1. On the form, double-click the toolbar to access its Click event
2. Implement the event as follows:

```
private void toolBar1_ButtonClick(object sender, System.Windows.Forms.ToolBarButtonClickEventArgs e)  
{  
    int IndexOfButtonSelected = this.toolBar1.Buttons.IndexOf(e.Button);  
    if( IndexOfButtonSelected == 4 )  
        this.Close();  
}
```

3. Execute the application

Coalesce

Chapter 25

Text-Based Controls

Labels

A label is a control that displays text to the user. The user cannot directly change it but only allowed to read it. A label can be used by itself or placed next to another control because such a control cannot inherently indicated what it is used for.

To create a label, on the Toolbox, click the Label button and click the form.

To programmatically create a label, declare a Label variable. Use the **new** operator to initialize it using its default constructor. Once you have created the control, you can change its properties as you see fit. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    Label lblFirstName = new Label();
    lblFirstName.Text = "&First Name";
    lblFirstName.Location = new System.Drawing.Point(20, 80);
    this.Controls.Add(lblFirstName);
}
```

Text Boxes

A text box is a control that is used to display text to, or receive text from, the user. It is presented as a rectangular box that cannot indicate what it is used for. For this reason, a text box is usually accompanied by a label that indicates its purpose.

To create a text box, on the Toolbox, click the **TextBox** button and click the form.

The **TextBox** control is based on the **TextBox** class. This can be used to programmatically create the control. To do this, declare a pointer to **TextBox**. Use the **new** operator to initialize it with its default constructor.

Check Box Controls

A check box is a control that makes a statement true or false. To perform this validation, this control displays a small square box that the user can click. To start, the square box is empty ☐. If the user clicks it, a check mark appears in the square box ☒. To let the user know what the check box control represents, the control is accompanied by a label that displays the statement. When the square box is empty ☐, the statement is false. When the square box is filled with a check mark ☒, the statement is true.

Coalesce



Creating a Check Box

To create a check box control, you can use the `CheckBox` class.

1. Create a new Visual C# Windows Application named **Pizza1**
2. Change the Text of the form to **Pizza Application**
3. To add a check box, on the Toolbox, click `CheckBox` and click in the top-center section of the form

Check Box Properties

Regular Windows Controls

1. The most important property of any control is the name. This is changed in the Properties window.
While the `checkBox1` control is still selected, on the Properties window, click **Name**
2. Type **chExtraCheese** and press Enter
3. Click Location. Type **16, 16** and press Enter
4. Click Text. Type **&Extra Cheese**
5. y to specify the position of the control. Don't forget to specify the label of the check box, which is done using the Text property.
As an example, change the file as follows:

6. Test the application



Coalesce

Checked

By default, a check box appears empty, which makes its statement false. To make the statement true, the user clicks it. There are three main ways you can use this property of a check box. To select a check box, you can set its **Checked** property to **true**. You can also do it programmatically as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.checkBox1.Checked = true;
}
```

To find out if an item is selected, get the value of its **Checked** property. Another possibility consists of toggling the state of the check mark with regards to another action. For example, you can check or uncheck the check mark when the user clicks another button. To do this, you can simply negate the truthfulness of the control as follows:

```
checkBox1.Checked = !checkBox1.Checked;
```

The Check-Alignment

By default, the square box of a check box control is positioned to the left side of its accompanying label. In Microsoft .Net applications, you have many options. Besides the left position, the most common alignment consists of positioning the round box to the right side of its label. The position of the round box with regards to its label is controlled by the **CheckAlign** property. The possible values are: **TopLeft**, **TopCenter**, **TopRight**, **MiddleRight**, **BottomRight**, **BottomCenter**, and **BottomLeft**.

To programmatically set the check alignment of a check box, you can call the **ContentAlignment** enumerator and select the desired value. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.checkBox1.CheckAlign = ContentAlignment.MiddleRight;
}
```

The Checked State

Instead of being definitely checked, you can let the user know that the decision of making the statement true or false is not complete. To do this, a check box can display as "half-checked". In this case the check mark would appear as if it were disabled. This behavior is controlled through the **CheckState** property. To provide this functionality, assign the **Indeterminate** value to its **CheckState** property. You can do this programmatically as follows:

```
this.checkBox1.CheckState = CheckState.Indeterminate;
```

Coalesce



The Three-State

The **CheckState** property only allows setting the check box control as "undecided". If you actually want the user to control three states of the control as checked, half-checked, or unchecked, use the **ThreeState** property.

By setting the **CheckState** Boolean property to true, the user can click it two to three times to get the desired value. When this ability is given to the user, then you can use or check the value of the **Indeterminate** property to find out whether the control is checked, half-checked, or unchecked.

Here is an example of specifying the check box control as being able to display one of three states:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.checkBox1.Checked = true;
    this.checkBox1.ThreeState = true;
}
```

The Appearance of the Button

By default, a check box control appears as a square box that gets filled with a check mark when the user clicks it. Optionally, you can make a check box control appear as a toggle button. In that case, the button would appear as a regular button. When the user clicks it, it appears down. If the user clicks it again, it becomes up.

To change the appearance of a check box, assign the **Button** or **Normal** value to its **Appearance** property. The **Appearance** values are defined in the **Appearance** enumerator. You can also do this programmatically as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.checkBox1.Appearance = Appearance.Button;
}
```

Radio Button Controls

A radio button, sometimes called an option button, is a circular control that comes in a group with other controls of the same type. Each radio button is made of a small empty circle ☐. From the group, when the user clicks one of them, the radio button that was clicked becomes filled with a big dot, like this ☒. When one of the radio buttons in the group is selected and displays its dot, the others display empty circles. To guide the user as to what the radio buttons mean, each is accompanied by a label.

Coalesce



Creating Radio Buttons

To create a radio button, on the Toolbox, you can click the **RadioButton** control. Alternatively, you can declare a **RadioButton** class and use the form's constructor to initialize the radio buttons. Because radio buttons always come as a group, you should include them in another control that visibly shows that the radio buttons belong to the same group. The most common control used for this purpose is the group box created using the **GroupBox** control.

Radio Button Properties

Checked

While radio buttons come as a group, only one of them can be selected at a given time. The item that is selected has its **Checked** property set to true. There are two main ways you can use this property. To select a particular radio button, set its **Checked** property to true. To find out if an item is selected, get the value of its **Checked** property. You can also programmatically check a radio button. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.radioButton2.Checked = true;
}
```

The Check-Alignment

By default, the round box of a radio button control is positioned to the left side of its accompanying label. In Microsoft .Net applications, you have many options. Besides the left position, the most common alignment consists of positioning the round box to the right side of its label. The position of the round box with regards to its label is controlled by the **CheckAlign** property. The possible values are: **TopLeft**, **TopCenter**, **TopRight**, **MiddleRight**, **BottomRight**, **BottomCenter**, and **BottomLeft**. You can also do this programmatically as follows:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.radioButton1.CheckAlign = ContentAlignment.MiddleRight;
}
```

Coalesce

The Checked State

By default, radio buttons appear as rounded boxes that get filled with a big dot when the user selects one. Optionally, you can make a radio button appear as a toggle button. In that case, the buttons would appear as regular buttons. When the user clicks one, it appears down while the others are up. If the user clicks another button, the previous one becomes up while the new one would be down. To change the appearance of a radio button, assign the **Button** or **Normal** value to its **Appearance** property. The **Appearance** values are defined in the **Appearance** namespace. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.radioButton1.Appearance = Appearance.Button;
}
```

List-Based Controls: The Combo Box

A combo box is a control that maintains a list of objects but displays one at a time to the user. A regular combo box is equipped with a rectangular text area and a down-pointing arrow. To use the combo box, the user clicks the arrow, which displays its list of items. Then the user clicks one item from the list and the list retracts like a plastic. The item the user would have selected would display in the text box side of the control. To perform a different selection, the user uses the same process. Practically, there are different variations of the combo box control depending on the programmer's goal.

To create a combo box in MS .Net application, you can use the **ComboBox** control from the Toolbox. After adding the control to a form, you can create a list of the items it will contain. Here is an example:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    this.comboBox1.Items.Add("Small");
    this.comboBox1.Items.Add("Medium");
    this.comboBox1.Items.Add("Large");
    this.comboBox1.Items.Add("Jumbo");
}
```

Creating a combo box might not be the most difficult thing to do but when the user has performed a different selection, you will usually need to know what item the user selected. The item that is currently selected on the combo box is its **Text** property. You can retrieve it and do what you judge necessary. For example, you can display it on a label as follows:

```
private void comboBox1_SelectedIndexChanged(object sender, System.EventArgs e)
{
    this.label1.Text = this.comboBox1.Text;
}
```

A combo box maintains its list of items by their text or their indexes. You can use the same ability to retrieve either the text or the index of the item that was selected.