Maharaja Education Trust (R), Mysuru

Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

Approved by AICTE, New Delhi,
Affiliated to VTU, Belagavi & Recognized by Government of Karnataka

Lecture Notes on

DOT NET FRAMEWORK FOR APPLICATION DEVELOPMENT (17CS564)

Prepared by

Prof. Ramya S & Prof.SmithaShree K P

Department of Information Science and Engineering

Maharaja Education Trust (R), Mysuru

Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

Vision

"To be recognized as a premier technical and management institution promoting extensive education fostering research, innovation and entrepreneurial attitude"

Mission

- To empower students with indispensable knowledge through dedicated teaching and collaborative learning.
- To advance extensive research in science, engineering and management disciplines.
- To facilitate entrepreneurial skills through effective institute - industry collaboration and interaction with alumni.
- To instill the need to uphold ethics in every aspect.
- To mould holistic individuals capable of contributing to the advancement of the society.

VISION OF THE DEPARTMENT

To be recognized as the best centre for technical education and research in the field of information science and engineering.

MISSION OF THE DEPARTMENT

- To facilitate adequate transformation in students through a proficient teaching learning process with the guidance of mentors and all-inclusive professional activities.
- To infuse students with professional, ethical and leadership attributes through industry collaboration and alumni affiliation.
- To enhance research and entrepreneurship in associated domains and to facilitate real time problem solving.
-

PROGRAM EDUCATIONAL OBJECTIVES:

- Proficiency in being an IT professional, capable of providing genuine solutions to information science problems.
- Capable of using basic concepts and skills of science and IT disciplines to pursue greater competencies through higher education.
- Exhibit relevant professional skills and learned involvement to match the requirements of technological trends.

PROGRAM SPECIFIC OUTCOME:

Student will be able to

- **PSO1:** Apply the principles of theoretical foundations, data Organizations, networking concepts and data analytical methods in the evolving technologies.
- **PSO2:**Analyse proficient algorithms to develop software and hardware competence in both professional and industrial areas

Program Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Overview

SUBJECT: DOT NET FRAMEWORK FOR APPLICATION DEVELOPMENT

SUBJECT CODE: 17CS564

C# is a modern, object-oriented programming language intended to create simple yet robust programs. Designed specifically to take advantage of CLI features, C# is the core language of the Microsoft .NET framework. C# was developed by Microsoft within its .NET framework initiative and later approved as a standard by ECMA C# programming language is a general-purpose, OOPS based programming language. C# development by "Anders Hejlsberg" in 2002.

In this course student will learn Microsoft Visual Studio framework along with powerful features such as object-oriented concepts and Extensible Types**.** C# syntax simplifies many of the complexities of C++ and provides powerful features such as value types,refernce type, enumerations, delegates, lambda expressions and direct memory access. C# supports students to learn generic methods which provide increased type safety and performance, and iterators, which enable implementers of collection classes to define custom iteration behaviors that are simple to use by client code. Language-Integrated Query (LINQ) expressions make the strongly-typed query a first-class language construct.

Students able to gain the skills to exploit the capabilities of C# and of the .NET Framework to develop programs useful for a broad range of desktop and Web applications. They can use C# to create Windows client applications, XML Web services, distributed components, client-server applications, database applications, and much more. Visual C# provides an advanced code editor, convenient user interface designers, integrated debugger, and many other tools to make it easier to develop applications based on the C# language.

Course Objectives

1. Inspect Visual Studio programming environment and toolset designed to build applications for Microsoft Windows
2. Understand Object Oriented Programming concepts in C# programming language.
3. Interpret Interfaces and define custom interfaces for application.
4. Build custom collections and generics in C#
5. Construct events and query data using query expression

Course Outcomes

CO's	DESCRIPTION OF THE OUTCOMES
17CS564.1	Apply the syntax and semantics of C# on Visual Studio .NET FRAMEWORK to build applications.
17CS564.2	Apply the object oriented programming concepts in c# programming language.
17CS564.3	Analyze value type and reference type in c# programming language.
17CS564.4	Analyze extensible types in c# programming languages
17CS564.5	Develop console applications using c# programming language to resolve a given problems.

Maharaja Institute of Technology Mysore

Department of Information Science and Engineering

Syllabus

SUBJECT: DOT NET FRAMEWORK FOR APPLICATION DEVELOPMENT
SUBJECT CODE: 17CS564

Topics Covered as per Syllabus	Teaching Hours
MODULE-1:	
Introducing Microsoft Visual C# and Microsoft Visual Studio 2015 Welcome to C#, Working with variables, operators and expressions, Writing methods and applying scope, Using decision statements, Using compound assignment and iteration statements, Managing errors and exceptions	**8 Hours**
MODULE-2:	
Understanding the C# object model: Creating and Managing classes and objects, Understanding values and references, Creating value types with enumerations and structures, Using arrays.	**8 Hours**
MODULE -3:	
Understanding parameter arrays, Working with inheritance, Creating interfaces and defining abstract classes, Using garbage collection and resource management	**8 Hours**
MODULE-4:	
Defining Extensible Types with C#: Implementing properties to access fields, Using indexers, Introducing generics, Using collections.	**8 Hours**
MODULE-5:	
Enumerating Collections, Decoupling application logic and handling events, Querying in-memory data by using query expressions, Operator overloading	**8 Hours**
List of Text Books	

1. T1. John Sharp, Microsoft Visual C# Step by Step, 8th Edition, PHI Learning Pvt. Ltd. 2016

List of Reference Books

T1.Christian Nagel, “C# 6 and .NET Core 1.0”, 1st Edition, Wiley India Pvt Ltd, 2016. Andrew Stellman and Jennifer Greene, “Head First C#”, 3rd Edition, O’Reilly Publications, 2013.

T2. Mark Michaelis, “Essential C# 6.0”, 5th Edition, Pearson Education India, 2016.

T3. Andrew Troelsen, “Prof C# 5.0 and the .NET 4.5 Framework”, 6th Edition, Apress and Dreamtech Press, 2012.

List of URLs, Text Books, Notes, Multimedia Content, etc

1. https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework
2. https://dotnet.microsoft.com/download/dotnet-framework
3. https://dotnet.microsoft.com/learn

Maharaja Institute of Technology Mysore

Department of Information Science and Engineering

Index

SUBJECT: DOT NET FRAMEWORK FOR APPLICATION DEVELOPMENT

SUBJECT CODE: 17CS564

MODULE 1 [CHAPTER 1]

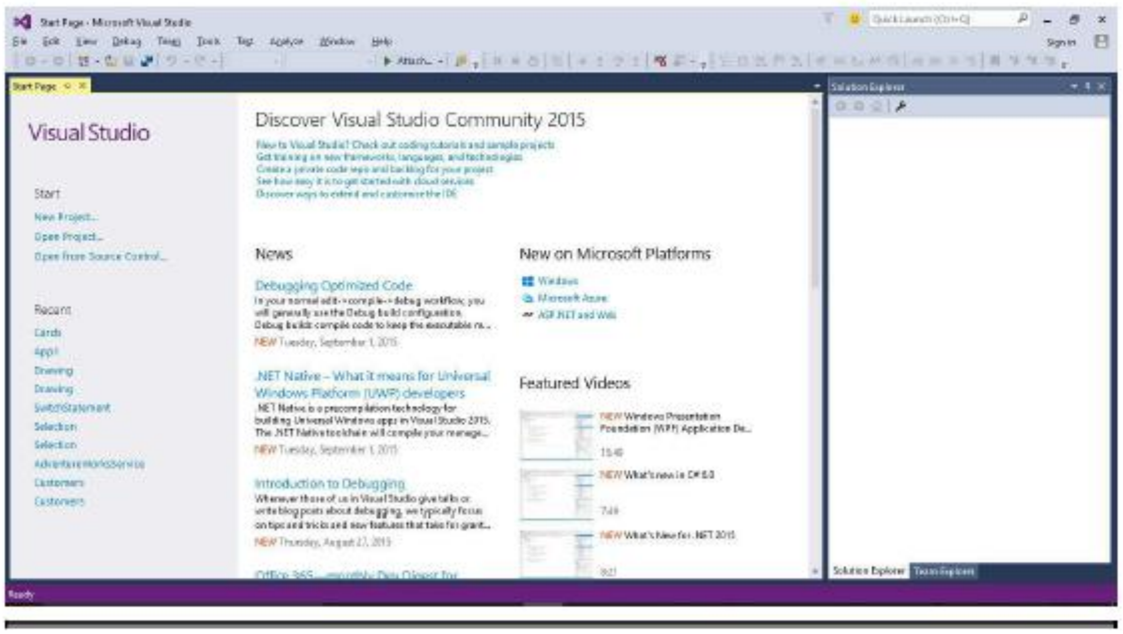
WELCOME TO C#

Beginning programming with the Visual Studio environment

Visual Studio 2015 is a tool-rich programming environment containing the functionality that you need to create large or small C# projects running on Windows. You can even construct projects that seamlessly combine modules written in different programming languages, such as C++, Visual Basic, and F#.

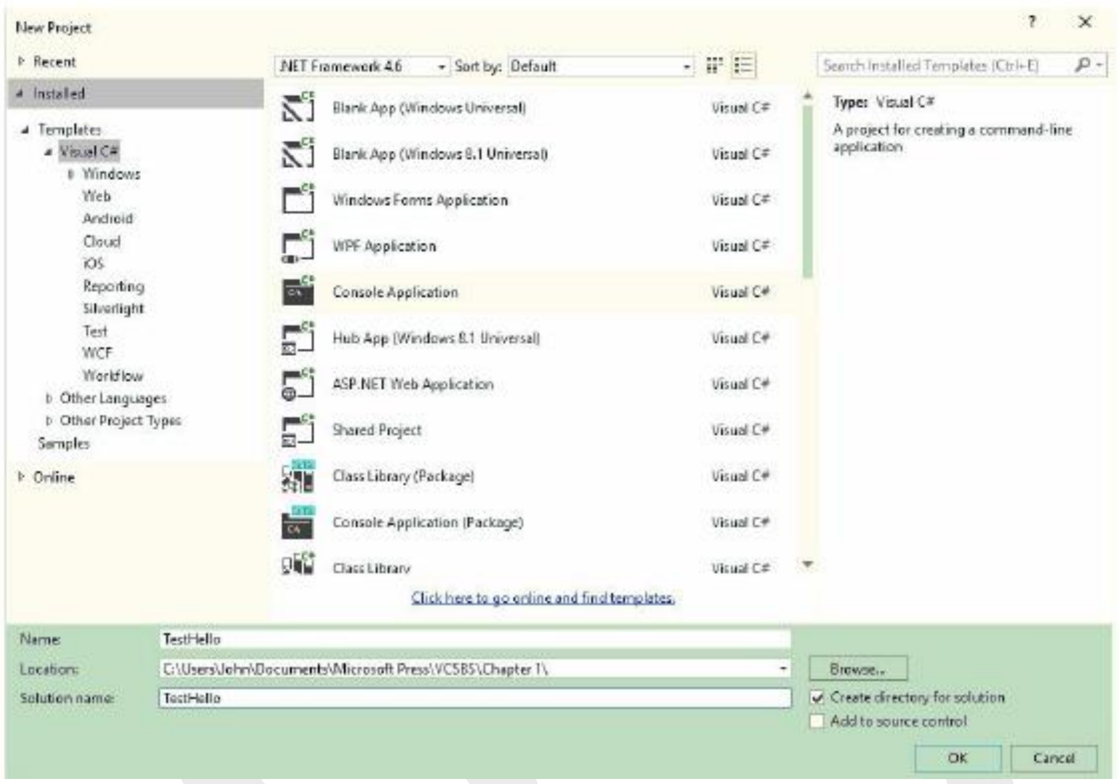
Create a console application in Visual Studio 2015

1. On the Windows taskbar, click Start, type Visual Studio 2015, and then press Enter. Visual Studio 2015 starts and displays the Start page, similar to the following.


Start Page - Microsoft Visual Studio
Visual Studio
Start
New Project...
Open Project...
Open from Source Control...
Recent
Discover Visual Studio Community 2015
News
Debugging Optimized Code
.NET Native – What it means for Universal Windows Platform (UWP) developers
Introduction to Debugging
New on Microsoft Platforms
Windows
Microsoft Azure
ASP.NET and Web
Featured Videos
Solution Explorer
Team Explorer
Ready

2. On the File menu, point to New, and then click Project. The New Project dialog box opens. This dialog box lists the templates that you can use as a starting point for building an application.

3. In the left pane, expand the Installed node (if it is not already expanded), expand Templates, and then click Visual C#.

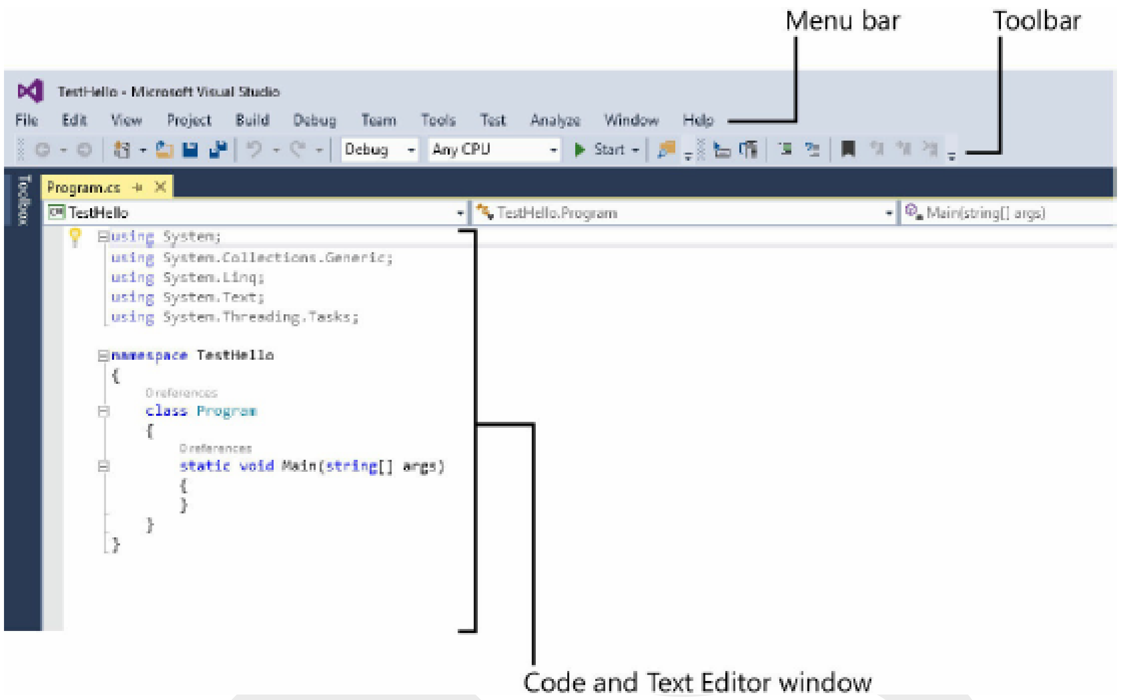

New Project
Recent
Installed
Templates
Visual C#
Windows
Web
Android
Cloud
iOS
Reporting
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Samples
Online
.NET Framework 4.6
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank App (Windows Universal) Visual C#
Blank App (Windows 8.1 Universal) Visual C#
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Hub App (Windows 8.1 Universal) Visual C#
ASP.NET Web Application Visual C#
Shared Project Visual C#
Class Library (Package) Visual C#
Console Application (Package) Visual C#
Class Library Visual C#
Click here to go online and find templates.
Type: Visual C#
A project for creating a command-line application
Browse...
OK
Cancel

Name: TestHello
Location: C:\Users\John\Documents\Microsoft Press\VCSBS\Chapter 1\
Solution name: TestHello
☑ Create directory for solution
☐ Add to source control

4. In the Location box, type **C:\Users\*YourName*\Documents\Microsoft Press\VCSBS\Chapter 1**. Replace the text *YourName* in this path with your Windows user name.

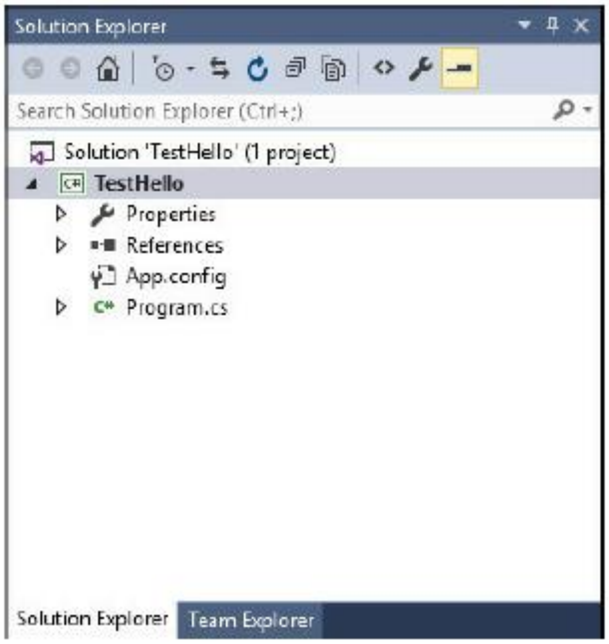
5. In the Name box, type **TestHello.**

6. Ensure that the Create Directory For Solution check box is selected and that the Add To Source Control check box is clear, and then click OK.


Menu bar
Toolbar
TestHello - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug
Any CPU
Start
Toolbox
Program.cs
TestHello
TestHello.Program
Main(string[] args)
Code and Text Editor window

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestHello
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
        }
    }
}
```

The Solution Explorer pane appears on the right side of the IDE, adjacent to the Code and Text Editor window:



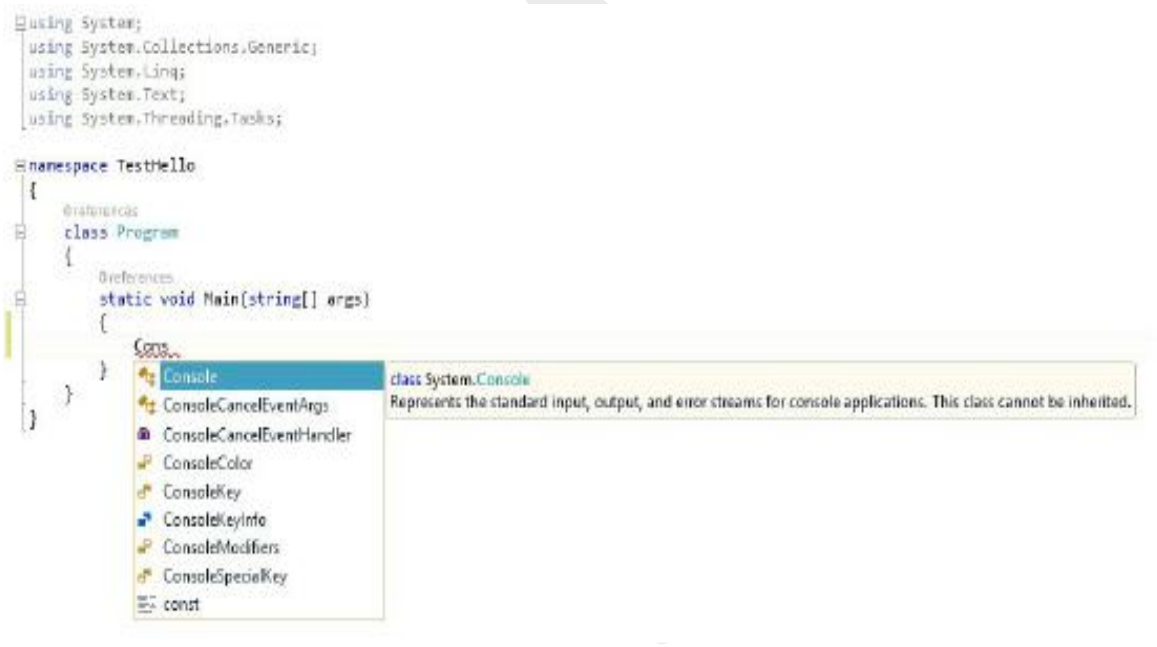
Writing your first program

The Program.cs file defines a class called *Program* that contains a method called *Main*. In C#, all executable code must be defined within a method, and all methods must belong to a class or a struct.

Write the code by using Microsoft IntelliSense

1. In the Code and Text Editor window displaying the *Program.cs* file, place the cursor in the *Main* method, immediately after the opening curly brace ({), and then press Enter to create a new line.

2. On the new line, type the word **Console**; this is the name of another class provided by the assemblies referenced by your application. It provides methods for displaying messages in the console window and reading input from the keyboard.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestHello
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Cons
        }
    }
}
```

class System.Console
Represents the standard input, output, and error streams for console applications. This class cannot be inherited.

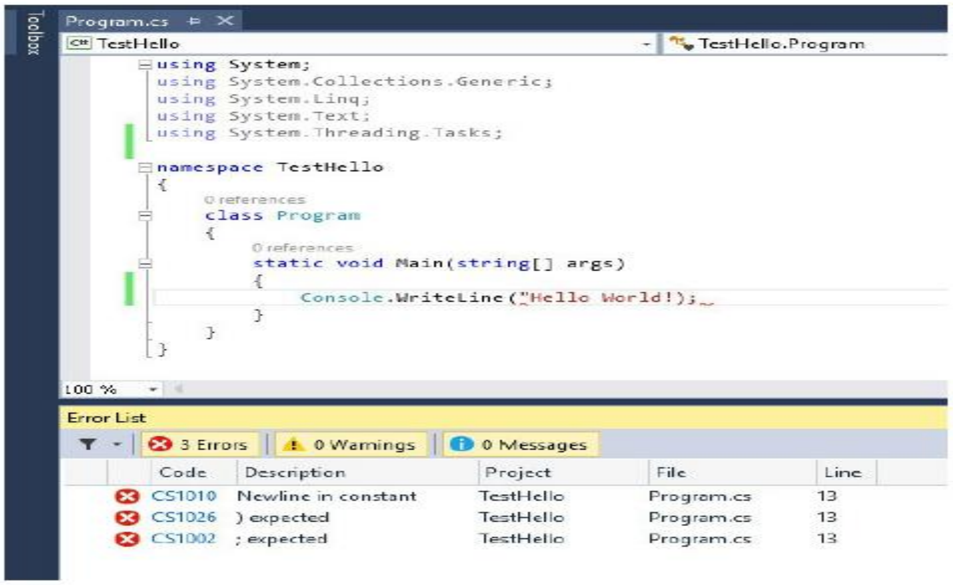
IntelliSense icons

When you type a period after the name of a class, IntelliSense displays the name of every member of that class. To the left of each member name is an icon that depicts the type of member.

Icon	Meaning
	Method (discussed in Chapter 3)
🔧	Property (discussed in Chapter 15, "Implementing properties to access fields")
	Class (discussed in Chapter 7)
	Struct (discussed in Chapter 9)
	Enum (discussed in Chapter 9)
	Extension method (discussed in Chapter 12, "Working with Inheritance")
	Interface (discussed in Chapter 13, "Creating interfaces and defining abstract classes")
	Delegate (discussed in Chapter 17, "Introducing generics")
⚡	Event (discussed in Chapter 17)
{ }	Namespace (discussed in the next section of this chapter)

Build and run the console application

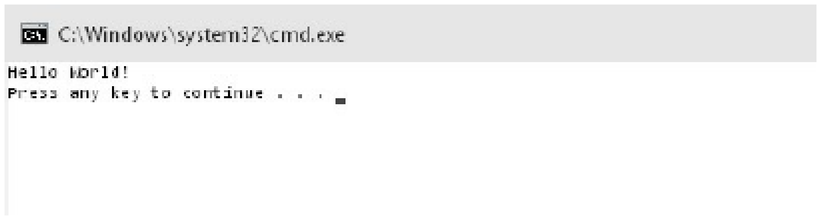
1. On the Build menu, click Build Solution. This action compiles the C# code, resulting in a program that you can run. The Output window appears below the Code and Text Editor window.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TestHello
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!);
        }
    }
}
```

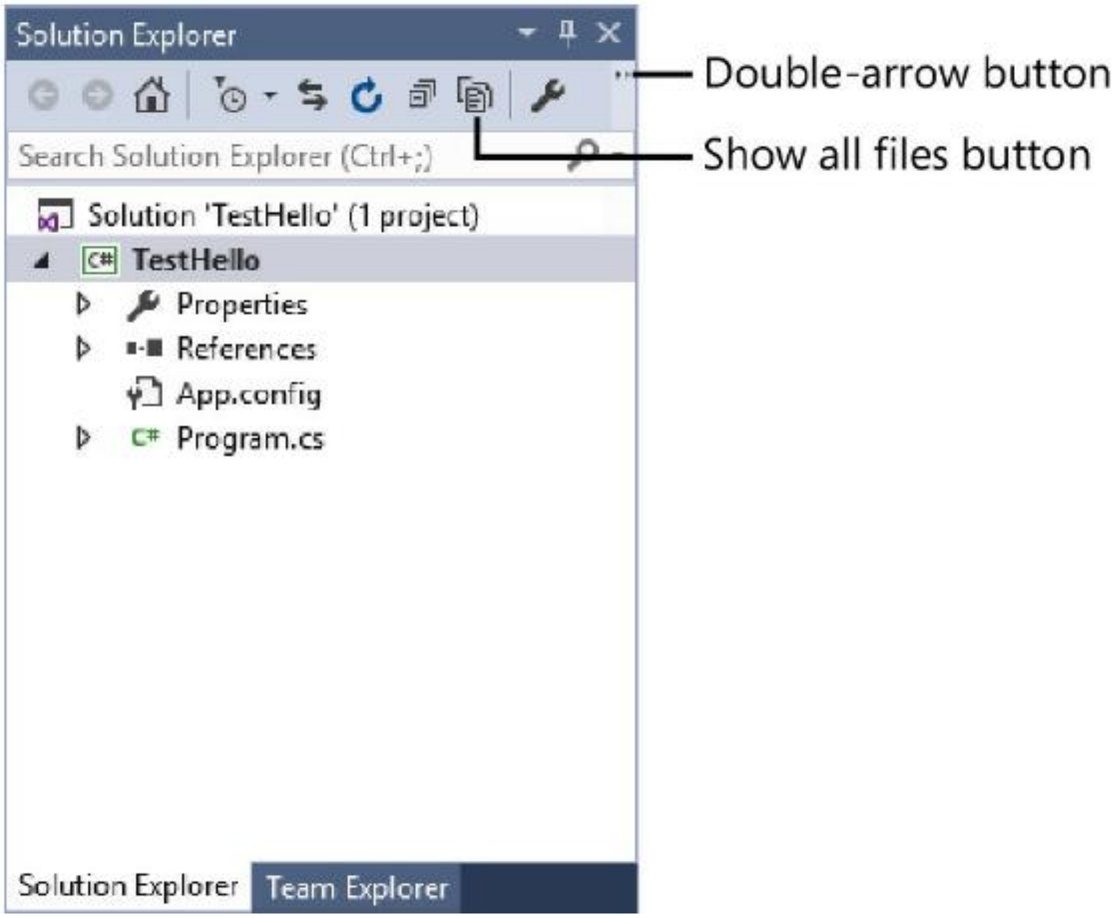
Code	Description	Project	File	Line
CS1010	Newline in constant	TestHello	Program.cs	13
CS1026	) expected	TestHello	Program.cs	13
CS1002	; expected	TestHello	Program.cs	13

2. On the Debug menu, click Start Without Debugging. A command window opens and the program runs. The message “Hello World!” appears.



```
C:\Windows\system32\cmd.exe
Hello World!
Press any key to continue . . .
```

3. Ensure that the command window displaying the program’s output has the focus (meaning that it’s the window that’s currently active), and then press Enter.
4. In Solution Explorer, click the *TestHello* project (not the solution), and then, on the Solution Explorer toolbar, click the Show All Files button.

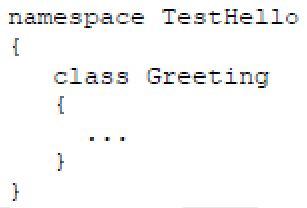


5. In Solution Explorer, expand the *bin* entry.
6. In Solution Explorer, expand the Debug folder. Several more items appear, including a file named *TestHello.exe*.

Using namespaces

First, it is harder to understand and maintain big programs than it is to understand and maintain smaller ones. Second, more code usually means more classes, with more methods, requiring you to keep track of more names. As the number of names increases, so does the likelihood of the project build failing because two or more names clash; for example, you might try to create two classes with the same name. The situation becomes more complicated when a program references assemblies written by other developers who have also used a variety of names**.**

Namespaces help solve this problem by creating a container for items such as classes. Two classes with the same name will not be confused with each other if they live in different namespaces. You can create a class named *Greeting* inside the namespace named *TestHello* by using the *namespace* keyword like this:



```
namespace TestHello
{
   class Greeting
   {
      ...
   }
}
```

You can then refer to the *Greeting* class as *TestHello.Greeting* in your programs. If another developer also creates a *Greeting* class in a different namespace, such as *NewNamespace*, and you install the assembly that contains this class on your computer, your programs will still work as expected because they are using your *TestHello.Greeting* class.

MODULE 1[CHAPTER 2]

Working with variables, operators and expressions

Understanding statements

- A ***statement*** is a command that performs an action, such as calculating a value and storing the result, or displaying a message to a user.
- We combine statements to create methods.
- Statements in C# follow a well-defined set of rules describing their format and construction. These rules are collectively known as ***syntax*.** (In contrast, the specification of what statements *do* is collectively known as ***semantics***.) One of the simplest and most important C# syntax rules states that we must terminate all statements with a semicolon. For example, Without the terminating semicolon, the following statement won't compile:
  Console.WriteLine("Hello, World!");

NOTE: C# is a "free format" language, which means that white space, such as a space character or a new line, is not significant except as a separator. In other words, we are free to lay out our statements in any style we choose. However, we should adopt a simple, consistent lawet style to make our programs easier to read and understand.

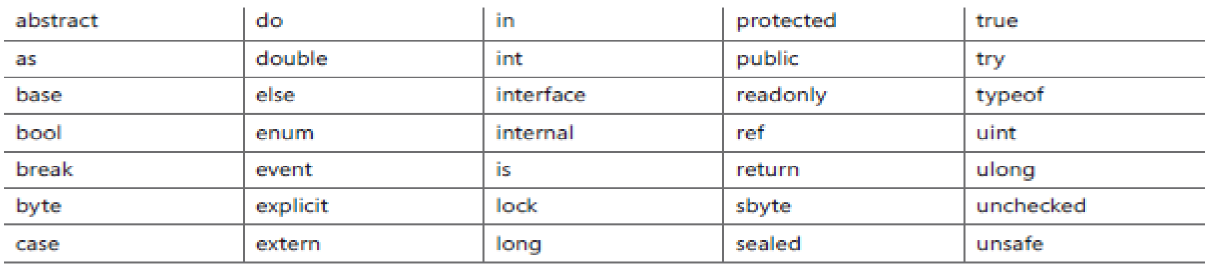
Using identifiers

- ***Identifiers*** are the names that we use to identify the elements in our programs, such as namespaces, classes, methods, and variables. In C#, we must adhere to the following **syntax rules** when choosing identifiers:
  1. We can use only letters (uppercase and loourcase), digits, and underscore characters.
  2. An identifier must start with a letter or an underscore.
- For example, *result, _score, footballTeam,* and *plan9* are all valid identifiers, whereas *result%, footballTeam$,* and *9plan* are not.

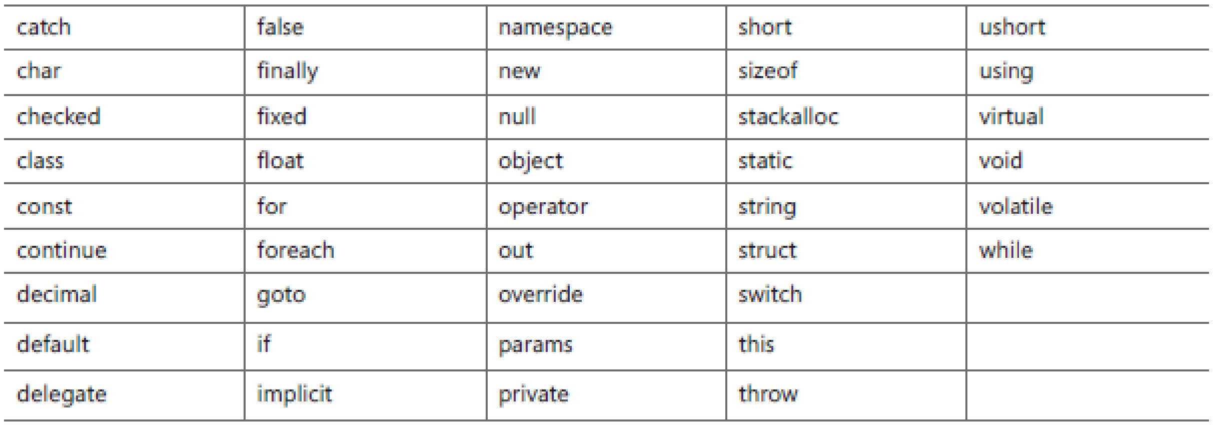
NOTE: C# is a case-sensitive language: *footballTeam* and *FootballTeam* are two different identifiers.

Identifying keywords

- The C# language reserves 77 identifiers for its own use, and we cannot reuse these identifiers for our own purposes. These identifiers are called *keywords*, and each has a particular meaning. Examples of keywords are *class, namespace,* and *using*. The following is the list of keywords:

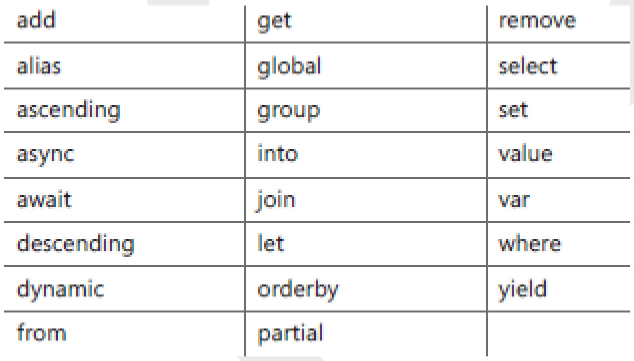


abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe



catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

- C# also uses the identifiers that follow. These identifiers are not reserved by C#, which means that we can use these names as identifiers for our own methods, variables, and classes, but we should avoid doing so if at all possible.



add	get	remove
alias	global	select
ascending	group	set
async	into	value
await	join	var
descending	let	where
dynamic	orderby	yield
from	partial	

Using variables

- A *variable* is a storage location that holds a value. we can think of a variable as a box in the computer's memory that holds temporary information. We must give each variable in a program an unambiguous name that uniquely identifies it in the context in which it is used.
- We use a variable's name to refer to the value it holds. For example, if we want to store the value of the cost of an item in a store, we might create a variable simply called *cost* and store the item's cost in this variable. Later on, if we refer to the *cost* variable, the value retrieved will be the item's cost that we stored there earlier.

Naming variables

- we should adopt a naming convention for variables that helps us to avoid confusion concerning the variables we have defined. The following list contains some general recommendations
- Don't start an identifier with an underscore. Although this is legal in C#, it can limit the interoperability of our code with applications built by using other languages, such as Microsoft Visual Basic.
- Don't create identifiers that differ only by case. For example, do not create one variable named *myVariable* and another named *MyVariable* for use at the same time, because it is too easy to get them confused. Also, defining identifiers that differ only by case can limit the ability to reuse classes in applications developed by using other languages that are not case sensitive, such as Visual Basic.
- Start the name with a loourcase letter.
- In a multiword identifier, start the second and each subsequent word with an uppercase letter. (This is called *camelCase notation*.)
- Don't use Hungarian notation.
- For example, *score*, *footballTeam*, *_score*, and *FootballTeam* are all valid variable names, but only the first two are recommended.

Declaring variables

- Variables hold values. C# has many different types of values that it can store and process—integers, floating-point numbers, and strings of characters, to name three. When we declare a variable, we must specify the type of data it will hold.
- we declare the type and name of a variable in a declaration statement. For example, the statement that follows declares that the variable named *age* holds *int* (integer) values. As always, we must terminate the statement with a semicolon.

```
int age;
```

The variable type *int* is the name of one of the *primitive* C# types, *integer*, which is a whole number.

- After we've declared our variable, we can assign it a value. The statement that follows assigns *age* the value 42. Again, note that the semicolon is required.

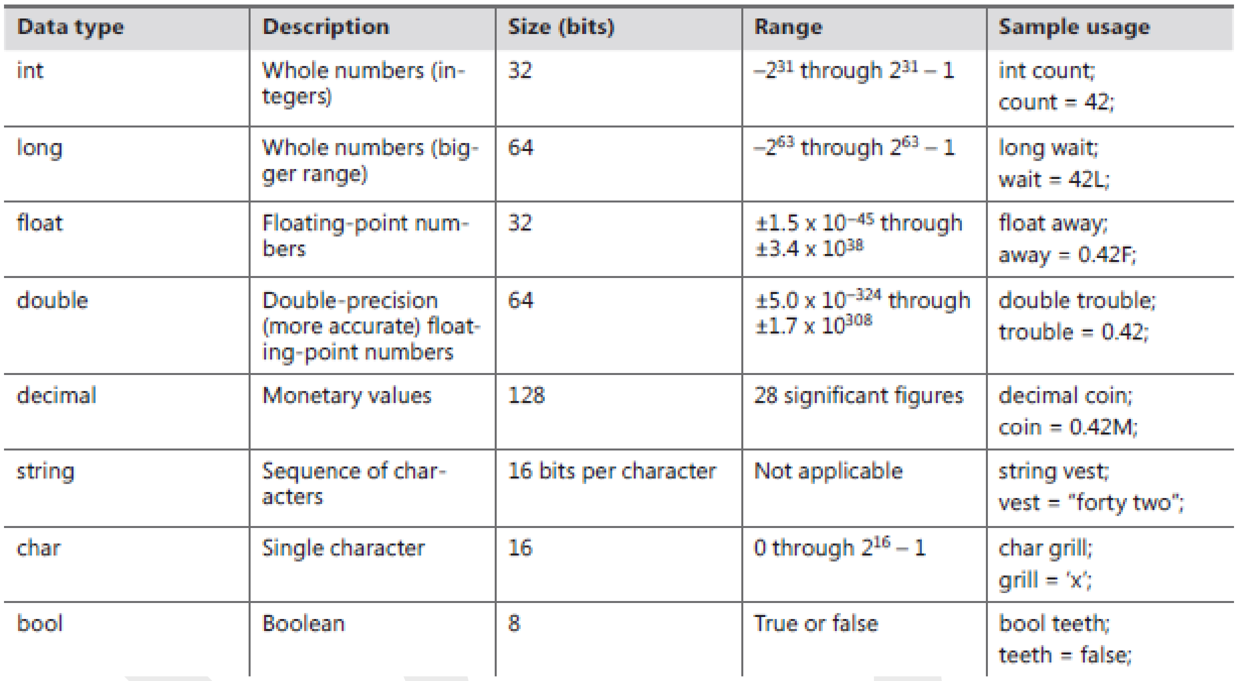
```
age = 42;
```

- The equal sign (=) is the *assignment* operator, which assigns the value on its right to the variable on its left. After this assignment, we can use the *age* variable in our code to refer to the value it holds. The next statement writes the value of the *age* variable (42) to the console:

```
Console.WriteLine(age);
```

Working with primitive data types

- C# has a number of built-in types called *primitive data types*. The following table lists the most commonly used primitive data types in C# and the range of values that we can store in each.



Data type	Description	Size (bits)	Range	Sample usage

int	Whole numbers (in-tegers)	32	-2^{31} through $2^{31} - 1$	int count;
				count = 42;

long	Whole numbers (big-ger range)	64	-2^{63} through $2^{63} - 1$	long wait;
				wait = 42L;

float	Floating-point num-bers	32	$\pm1.5 \times 10^{-45}$ through $\pm3.4 \times 10^{38}$	float away;
				away = 0.42F;

double	Double-precision (more accurate) float-ing-point numbers	64	$\pm5.0 \times 10^{-324}$ through $\pm1.7 \times 10^{308}$	double trouble;
				trouble = 0.42;

decimal	Monetary values	128	28 significant figures	decimal coin;
				coin = 0.42M;

string	Sequence of char-acters	16 bits per character	Not applicable	string vest;
				vest = "forty two";

char	Single character	16	0 through $2^{16} - 1$	char grill;
				grill = 'x';

bool	Boolean	8	True or false	bool teeth;
				teeth = false;

Unassigned local variables

- When we declare a variable, it contains a random value until we assign a value to it. This behavior was a rich source of bugs in C and C++ programs that created a variable and accidentally used it as a source of information before giving it a value.
- C# does not allow us to use an unassigned variable. we must assign a value to a variable before we can use it; otherwise, our program will not compile. This requirement is called the ***definite assignment rule*.**
- For example, the following statements generate the compile-time error message "Use of unassigned local variable 'age'" because the *Console.WriteLine* statement attempts to display the value of an uninitialized variable:

```
 int age;
Console.WriteLine(age); // compile-time error
```

Using arithmetic operators

- C# supports the regular arithmetic operations we learned in our childhood: the plus sign (+) for addition, the minus sign (–) for subtraction, the asterisk (*) for multiplication, and the forward slash (/) for division. The symbols +, –, *, and / are called *operators* because they "operate" on values to create new values.
- In the following example, the variable *moneyPaidToConsultant* ends up holding the product of 750 (the daily rate) and 20 (the number of days the consultant was employed):

```
long moneyPaidToConsultant;

moneyPaidToConsultant = 750 * 20;
```

Note: The values on which an operator performs its function are called *operands*. In the expression 750 * 20, the * is the operator, and 750 and 20 are the operands.

Operators and types

- Not all operators are applicable to all data types. The operators that we can use on a value depend on the value's type. For example, we can use all the arithmetic operators on values of type *char, int, long, float, double*, or *decimal*.
- However, with the exception of the plus operator, +, we can't use the arithmetic operators on values of type *string*, and we cannot use any of them with values of type *bool*. So, the following statement is not allowed, because the *string* type does not support the minus operator (subtracting one string from another is meaningless):

```
// compile-time error Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

- However, we can use the + operator to concatenate string values. We need to be careful because this can have unexpected results. For example, the following statement writes "431" (not "44") to the console:

```
Console.WriteLine("43" + "1");
```

Note: The .NET Framework provides a method called *Int32.Parse* that we can use to convert a string value to an integer if we need to perform arithmetic computations on values held as strings.

- We should also be aware that the type of the result of an arithmetic operation depends on the type of the operands used. For example, the value of the expression 5.0/2.0 is 2.5; the type of both operands is *double*, so the type of the result is also *double*.
- However, the value of the expression 5/2 is 2. In this case, the type of both operands is *int*, so the type of the result is also *int*. C# always rounds toward zero in circumstances like this.
- The situation gets a little more complicated if we mix the types of the operands. For example, the expression 5/2.0 consists of an *int* and a *double*.
- The C# compiler detects the mismatch and generates code that converts the *int* into a *double* before performing the operation. The result of the operation is therefore a *double* (2.5). However, although this works, it is considered poor practice to mix types in this way.
- C# also supports one less-familiar arithmetic operator: the *remainder*, or *modulus*, operator, which is represented by the percent sign (%). The result of *x % y* is the remainder after dividing the value *x* by the value *y*. So, for example, 9 % 2 is 1 because 9 divided by 2 is 4, remainder 1.

Note: if we divide zero by anything, the result is zero, but if we divide anything by zero the result is infinity. The expression 0.0/0.0 results in a paradox—the value must be zero and infinity at the same time. C# has another special value for this situation called *NaN*, which stands for "not a number." So if we evaluate 0.0/0.0, the result is *NaN*.

Controlling precedence

- ***Precedence*** governs the order in which an expression's operators are evaluated. Consider the following expression, which uses the + and * operators:

  2 + 3 * 4

- This expression is potentially ambiguous: do we perform the addition first or the multiplication? The order of the operations matters because it changes the result:
- If we perform the addition first, followed by the multiplication, the result of the addition (2 + 3) forms the left operand of the * operator, and the result of the whole expression is 5 * 4, which is 20.
- If we perform the multiplication first, followed by the addition, the result of the multiplication (3 * 4) forms the right operand of the + operator, and the result of the whole expression is 2 + 12, which is 14.
- In C#, the multiplicative operators (*, /, and %) have precedence over the additive operators (+ and –), so in expressions such as 2 + 3 * 4, the multiplication is performed first, followed by the addition. The ansour to 2 + 3 * 4 is therefore 14.
- We can use parentheses to override precedence and force operands to bind to operators in a different way. For example, in the following expression, the parentheses force the 2 and the 3 to bind to the + operator (making 5), and the result of this addition forms the left operand of the * operator to produce the value 20:

  (2 + 3) * 4

Using associativity to evaluate expressions

- What happens when an expression contains different operators that have the same precedence? This is where *associativity* becomes important.
- **Associativity** is the direction (left or right) in which the operands of an operator are evaluated. Consider the following expression that uses the / and * operators:

  4 / 2 * 6

- At first glance, this expression is potentially ambiguous. Do we perform the division first or the multiplication? The precedence of both operators is the same (they are both multiplicative), but the order in which the operators in the expression are applied is important because we can get two different results:
- If we perform the division first, the result of the division (4/2) forms the left operand of the * operator, and the result of the whole expression is (4/2) * 6, or 12.
- If we perform the multiplication first, the result of the multiplication (2 * 6) forms the right operand of the / operator, and the result of the whole expression is 4/(2 * 6), or 4/12.

- In this case, the associativity of the operators determines how the expression is evaluated. The * and / operators are both left-associative, which means that the operands are evaluated from left to right. In this case, 4/2 will be evaluated before multiplying by 6, giving the result 12.

Associativity and the assignment operator

- In C#, the equal sign (=) is an operator. All operators return a value based on their operands. The assignment operator = is no different. It takes two operands: the operand on the right side is evaluated and then stored in the operand on the left side.
- The value of the assignment operator is the value that was assigned to the left operand. For example, in the following assignment statement, the value returned by the assignment operator is 10, which is also the value assigned to the variable *myInt*:

```
int myInt;

myInt = 10; // value of assignment expression is 10
```

- Well, because the assignment operator returns a value, we can use this same value with another occurrence of the assignment statement, like this:

```
int myInt;int myInt2;myInt2 = myInt = 10;
```

- The value assigned to the variable *myInt2* is the value that was assigned to *myInt*. The assignment statement assigns the same value to both variables. This technique is useful if we want to initialize several variables to the same value. It makes it very clear to anyone reading our code that all the variables must have the same value:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

Incrementing and decrementing variables

- If we want to add 1 to a variable, we can use the + operator, as demonstrated here:

```
count = count + 1;
```

- However, adding 1 to a variable is so common that C# provides its own operator just for this purpose: the ++ operator. To increment the variable *count* by 1, we can write the following statement:

```
count++;
```

- Similarly, C# provides the -- operator that we can use to subtract 1 from a variable, like this:

```
count--;
```

- The ++ and -- operators are *unary* operators, meaning that they take only a single operand. They share the same precedence and are both left-associative.

Prefix and postfix

- The increment (++) and decrement (--) operators are unusual in that we can place them either before or after the variable.
- Placing the operator symbol before the variable is called the *prefix form* of the operator, and using the operator symbol after the variable is called the *postfix form*. Here are examples:

```
count++; // postfix increment
++count; // prefix increment
count--; // postfix decrement
--count; // prefix decrement
```

- The value returned by *count++* is the value of *count* before the increment takes place, whereas the value returned by *++count* is the value of *count* after the increment takes place. Here is an example:

```
 int x;
 x = 42;
Console.WriteLine(x++); // x is now 43, 42 written out
 x = 42;Console.WriteLine(++x); // x is now 43, 43 written out
```

Declaring implicitly typed local variables

- We can also ask the C# compiler to infer the type of a variable from an expression and use this type when declaring the variable by using the *var* keyword in place of the type, as demonstrated here:

```
var myVariable = 99;

var myOtherVariable = "Hello";
```

- The variables *myVariable* and *myOtherVariable* are referred to as *implicitly typed* variables. The *var* keyword causes the compiler to deduce the type of the variables from the types of the expressions used to initialize them. In these examples, *myVariable* is an *int*, and *myOtherVariable* is a *string*.
- However, it is important for we to understand that this is a convenience for declaring variables only, and that after a variable has been declared we can assign only values of the inferred type to it—we cannot assign *float*, *double*, or *string* values to *myVariable* at a later point in our program, for example.
- We should also understand that we can use the *var* keyword only when we supply an expression to initialize a variable. The following declaration is illegal and causes a compilation error:

```
var yetAnotherVariable; // Error - compiler cannot infer type.
```

MODULE 1[CHAPTER 3]
Writing methods and applying scope

Creating methods

- A ***method*** is a named sequence of statements. A method has a name and a body. The method name should be a meaningful identifier that indicates the overall purpose of the method. The method body contains the actual statements to be run when the method is called.
- Additionally, methods can be given some data for processing and can return information, which is usually the result of the processing. Methods are a fundamental and poourful mechanism.

Declaring a method

The syntax for declaring a C# method is as follows:

```
returnType methodName ( parameterList )
{
// method body statements go here
 }
```

The following is a description of the elements that make up a declaration:

- The *returnType* is the name of a type and specifies the kind of information the method returns as a result of its processing. This can be any type, such as *int* or *string*. If we're writing a method that does not return a value, we must use the keyword *void* in place of the return type.
- The *methodName* is the name used to call the method. Method names follow the same identifier rules as variable names. For example, *addValues* is a valid method name, whereas *add$Values* is not. For now, we should follow the camelCase convention for method names; for example, *displayCustomer*.
- The *parameterList* is optional and describes the types and names of the information that we can pass into the method for it to process. We write the parameters between opening and closing parentheses, ( ), as though we're declaring variables, with the name of the type followed by the name of the parameter. If the method we're writing has two or more parameters, we must separate them with commas.
- The method body statements are the lines of code that are run when the method is called. They are enclosed between opening and closing braces, { }.

NOTE:we should note that C# does not support global methods. We must write all our methods inside a class; otherwise, our code will not compile

- Here's the definition of a method called *addValues* that returns an *int* result and has two *int* parameters, *leftHandSide* and *rightHandSide*:

```
int addValues(int leftHandSide, int rightHandSide)
 {
     // ...
```

```
                    // method body statements go here
                   // ...
                 }
```

NOTE: We must explicitly specify the types of any parameters and the return type of a method. We cannot use the *var* keyword.

- Here's the definition of a method called *showResult* that does not return a value and has a single *int* parameter, called *ansour*:

```
void showResult(int ansour)
{
  // ...
}
```

Notice the use of the keyword *void* to indicate that the method does not return anything.

Returning data from a method

- If we want a method to return information (that is, its return type is not *void*), we must include a *return* statement at the end of the processing in the method body.
- A *return* statement consists of the keyword *return* followed by an expression that specifies the returned value, and a semicolon. The type of the expression must be the same as the type specified by the method declaration.
- For example, if a method returns an *int*, the *return* statement must return an *int*; otherwise, our program will not compile. Here is an example of a method with a *return* statement:

```
int addValues(int leftHandSide, int rightHandSide)
{
// ...
return leftHandSide + rightHandSide;
}
```

- The *return* statement is usually positioned at the end of the method because it causes the method to finish, and control returns to the statement that called the method, as described later in this chapter. Any statements that occur after the *return* statement are not executed (although the compiler warns we about this problem if we place statements after the *return* statement).
- If we don't want our method to return information (that is, its return type is *void*), we can use a variation of the *return* statement to cause an immediate exit from the method. We write the keyword *return* and follow it immediately by a semicolon. For example:

```
void showResult(int ansour)
      {
        // display the ansour
            ...
```

```
    return;
  }
```

Calling methods

- We call a method by name to ask it to perform its task. If the method requires information (as specified by its parameters), we must supply the information requested. If the method returns information (as specified by its return type), we should arrange to capture this information somehow.

Specifying the method call syntax

The syntax of a C# method call is as follows:

```
result = methodName ( argumentList )
```

The following is a description of the elements that make up a method call:

- The ***methodName*** must exactly match the name of the method we're calling. Remember, C# is a case-sensitive language.
- The ***result*** **=** clause is optional. If specified, the variable identified by *result* contains the value returned by the method. If the method is *void* (that is, it does not return a value), we must omit the *result* = clause of the statement. If we don't specify the *result* = clause and the method does return a value, the method runs but the return value is discarded.
- The ***argumentList*** supplies the information that the method accepts. We must supply an argument for each parameter, and the value of each argument must be compatible with the type of its corresponding parameter. If the method we're calling has two or more parameters, we must separate the arguments with commas.
- To clarify these points, take a look at the *addValues* method again:

```
int addValues(int leftHandSide, int rightHandSide)
  {
      // ...
  }
```

- The *addValues* method has two *int* parameters, so we must call it with two comma-separated *int* arguments, such as this:

```
addValues(39, 3); // okay
```

- We can also replace the literal values 39 and 3 with the names of *int* variables. The values in those variables are then passed to the method as its arguments, like this:

```
int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2);
```

- If we try to call *addValues* in some other way, we will probably not succeed for the reasons described in the following examples:

```
addValues; // compile-time error, no parentheses
```

```
addValues(); // compile-time error, not enough arguments
addValues(39); // compile-time error, not enough arguments
addValues("39", "3"); // compile-time error, wrong types for arguments
```

- The *addValues* method returns an *int* value. We can use this *int* value wherever an *int* value can be used. Consider these examples:

```
int result = addValues(39, 3); // on right-hand side of an assignment
showResult(addValues(39, 3)); // as argument to another method
```

Applying scope

- We create variables to hold values. We can create variables at various points in our applications. For example, the *calculateClick* method in the Methods project creates an *int* variable called *calculatedValue* and assigns it an initial value of zero, like this:

```
private void calculateClick(object sender, RoutedEventArgs e)
   {
    int calculatedValue = 0;
    ...
   }
```

- When a variable can be accessed at a particular location in a program, the variable is said to be in *scope* at that location. The *calculatedValue* variable has method scope; it can be accessed throughout the *calculateClick* method but not outside of that method. We can also define variables with different scope; for example, we can define a variable outside of a method but within a class, and this variable can be accessed by any method within that class. Such a variable is said to have *class scope*.

Defining local scope

- The opening and closing braces that form the body of a method define the scope of the method. Any variables we declare inside the body of a method are scoped to that method; they disappear when the method ends and can be accessed only by code running in that method. These variables are called *local variables* because they are local to the method in which they are declared; they are not in scope in any other method.

- The scope of local variables means that we cannot use them to share information between methods. Consider this example:

```
class Example
{
        void firstMethod()
        {
        int myVar;
        ...
        }
        void anotherMethod()
        { myVar = 42;             // error - variable not in scope ... }
}
```

- This code fails to compile because *anotherMethod* is trying to use the variable *myVar*, which is not in scope. The variable *myVar* is available only to statements in *firstMethod* that occur after the line of code that declares *myVar*.

Defining class scope

- The opening and closing braces that form the body of a class define the scope of that class. Any variables we declare within the body of a class (but not within a method) are scoped to that class. The proper C# term for a variable defined by a class is *field*.
  Here is an example:

```
class Example
{
        void firstMethod()
        {
        myField = 42; // ok ...
        }
        void anotherMethod()
        {
        myField++; // ok ...
        }
 int myField = 0;
}
```

- The variable *myField* is defined in the class but outside the methods *firstMethod* and *anotherMethod*. Therefore, *myField* has class scope and is available for use by all methods in that class.

Overloading methods

- If two identifiers have the same name and are declared in the same scope, they are said to be *overloaded*. However, there is a way that we can overload an identifier for a method, and that way is both useful and important.

- Consider the *WriteLine* method of the *Console* class, where each version of the *WriteLine* method takes a different set of parameters; one version takes no parameters and simply outputs a blank line, another version takes a *bool* parameter, outputs it as a string, and so on. At compile time, the compiler looks at the types of the arguments we are passing in and then arranges for our application to call the version of the method that has a matching set of parameters. Here is an example:

```
static void Main()
{
 Console.WriteLine("The ansour is ");
 Console.WriteLine(42);
}
```

- Overloading is primarily useful when we need to perform the same operation on different data types or varying groups of information. We can overload a method when they have the same name but a different number of parameters, or when the types of the parameters differ. When we call a method, we supply a comma-separated list of arguments and type of the arguments is used by the compiler to select one of the overloaded methods. We can't declare two methods with the same name that differ only in their return type.

Using optional parameters and named arguments

- Optional parameters are also useful in other situations. They provide a compact and simple solution when it is not possible to use overloading because the types of the parameters do not vary sufficiently to enable the compiler to distinguish between implementations.
  For example, consider the following method:
```
public void DoWorkWithData(int iData, float fData, int moreData)
        {
         ...
        }
public void DoWorkWithData(int intData, float floatData)
         {
          ...
         }
```
- If we write a statement that calls the *DoWorkWithData* method, we can provide either two or three parameters of the appropriate types, and the compiler uses the type information to determine which overload to call:
```
         int arg1 = 99; float arg2 = 100.0F; int arg3 = 101;
        DoWorkWithData(arg1, arg2, arg3); // Call overload with three parameters
        DoWorkWithData(arg1, arg2);         // Call overload with two parameters
```
- Consider the following constructors
```
public void DoWorkWithData(int iData)
{
   ...
 }
public void DoWorkWithData(int moreData)
  {
     ...
   }
```
- The issue here is that to the compiler, these two overloads appear identical. Our code will fail to compile and will instead generate the error "Type '*typename*' already defines a member called 'DoWorkWithData' with the same parameter types." To understand why this is so, if this code oure legal, consider the following statements:
```
int arg1 = 99; int arg3 = 101;
DoWorkWithData(arg1);
 DoWorkWithData(arg3);
```
- Which overload or overloads would the calls to *DoWorkWithData* invoke? Using optional parameters and named arguments can help to solve this problem.

Defining optional parameters

- We specify that a parameter is optional when we define a method by providing a default value for the parameter. We indicate a default value by using the assignment operator. In the *optMethod* method shown next, the *first* parameter is mandatory because it does not specify a default value, but the *second* and *third* parameters are optional:

```
void optMethod(int first, double second = 0.0, string third = “Hello”)
{
 ...
 }
```

```
optMethod(99, 123.45, “World”); // Arguments provided for all three parameters
optMethod(100, 54.321); // Arguments provided for first two parameters only
```

- We must specify all mandatory parameters before any optional parameters. The first call to the *optMethod* method provides values for all three parameters. The second call specifies only two arguments, and these values are applied to the *first* and *second* parameters. The *third* parameter receives the default value of “Hello” when the method runs.

Passing named arguments

- The C# allow us to specify parameters by name. This feature lets we pass the arguments in a different sequence. To pass an argument as a named parameter, we specify the name of the parameter, followed by a colon and the value to use. The following examples perform the same function as those shown in the previous section, except that the parameters are specified by name:

```
optMethod(first : 99, second : 123.45, third : “World”);
optMethod(first : 100, second : 54.321);
```

- Named arguments give we the ability to pass arguments in any order. We can rewrite the code that calls the *optMethod* method, such as shown here:

```
optMethod(third : “World”, second : 123.45, first : 99);
optMethod(second : 54.321, first : 100);
optMethod(first : 99, third : “World”); // omit arguments.
```

- Additionally, we can mix positional and named arguments. However, if we use this technique, we must specify all the positional arguments before the first named argument.

```
optMethod(99, third : “World”); // First argument is positional
```

Resolving ambiguities with optional parameters and named arguments

- Using optional parameters and named arguments can result in some possible ambiguities in our code. We need to understand how the compiler resolves these ambiguities; otherwise, we might find our applications behaving in unexpected ways. Suppose that we define the *optMethod* method as an overloaded method, as shown in the following example:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
 ...
 }
void optMethod(int first, double second = 1.0, string third = "Goodbye", int
fourth = 100 )
{
 ...
 }
```

- The following example, arise problem if we attempt to call the *optMethod* method and omit some of the arguments corresponding to one or more of the optional parameters:

```
optMethod(1, 2.5, "World");
optMethod(1, fourth : 101);
```

- In this code, the call to *optMethod* omits arguments for the *second* and *third* parameters, but it specifies the *fourth* parameter by name. Only one version of *optMethod* matches this call, so this is not a problem. This next one will get we thinking, though:

```
optMethod(1, 2.5);
optMethod(1, third : "World");
optMethod(1);optMethod(second : 2.5, first : 1);
```

- This is an unresolvable ambiguity, and the compiler does not let we compile the application.

MODULE 1[CHAPTER 4]

Using decision statements

Using Boolean operators

- A Boolean operator is an operator that performs a calculation whose result is either true or false. C# has several very useful Boolean operators, the simplest of which is the *NOT* operator, represented by the exclamation point (!). The *!* operator negates a Boolean value, yielding the opposite of that value. In the preceding example, if the value of the variable *areOuready* is true, the value of the expression *!areOuready* is false.

Understanding equality and relational operators

- Two Boolean operators that we will frequently use are equality (==) and inequality (*!*=). These are binary operators with which we can determine whether one value is the same as another value of the same type, yielding a Boolean result. The following table summarizes how these operators work, using an *int* variable called *age* as an example.

Operator	Meaning	Example	Outcome if age is 42
==	Equal to	age == 100	false
!=	Not equal to	age != 0	true

- Closely related to == and *!*= are the *relational* operators. We use these operators to find out whether a value is less than or greater than another value of the same type. The following table shows how to use these operators.

Operator	Meaning	Example	Outcome if age is 42
<	Less than	age < 21	false
<=	Less than or equal to	age <= 18	false
>	Greater than	age > 16	true
>=	Greater than or equal to	age >= 30	true

Understanding conditional logical operators

- C# also provides two other binary Boolean operators: the logical AND operator, which is represented by the *&&* symbol, and the logical OR operator, which is represented by the || symbol. Collectively, these are known as the conditional logical operators.
- Their purpose is to combine two Boolean expressions or values into a single Boolean result. These operators are similar to the equality and relational operators in that the value of the expressions in which they appear is either true or false, but

they differ in that the values on which they operate must also be either true or false.

- The outcome of the *&&* operator is *true* if and only if both of the Boolean expressions it's evaluating are *true*. For example, the following statement assigns the value *true* to *validPercentage* if and only if the value of *percent* is greater than or equal to 0 and the value of *percent* is less than or equal to 100:

  bool validPercentage;validPercentage = (percent >= 0) && (percent <= 100);

- The outcome of the || operator is *true* if either of the Boolean expressions it evaluates is *true*. We use the || operator to determine whether any one of a combination of Boolean expressions is *true*. For example, the following statement assigns the value *true* to *invalidPercentage* if the value of *percent* is less than 0 or the value of *percent* is greater than 100:

  bool invalidPercentage;invalidPercentage = (percent < 0) || (percent > 100);

Short-circuiting

- The *&&* and || operators both exhibit a feature called *short-circuiting*. Sometimes, it is not necessary to evaluate both operands when ascertaining the result of a conditional logical expression.
- For example, if the left operand of the *&&* operator evaluates to *false*, the result of the entire expression must be *false*, regardless of the value of the right operand.
- Similarly, if the value of the left operand of the || operator evaluates to *true*, the result of the entire expression must be *true*, irrespective of the value of the right operand. In these cases, the *&&* and || operators bypass the evaluation of the right operand. Here are some examples:

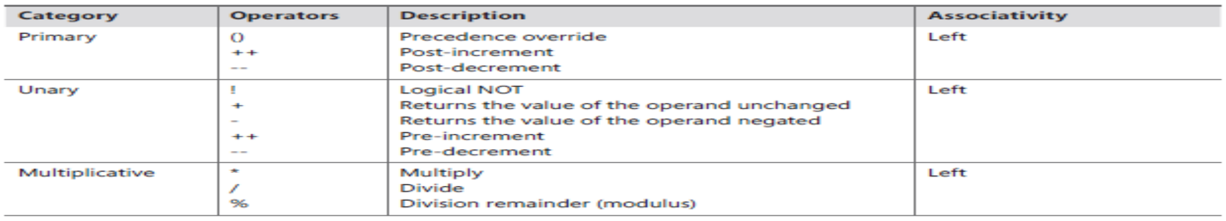
  (percent >= 0) && (percent <= 100)

- In this expression, if the value of *percent* is less than 0, the Boolean expression on the left side of *&&* evaluates to *false*. This value means that the result of the entire expression must be *false*, and the Boolean expression to the right of the *&&* operator is not evaluated.

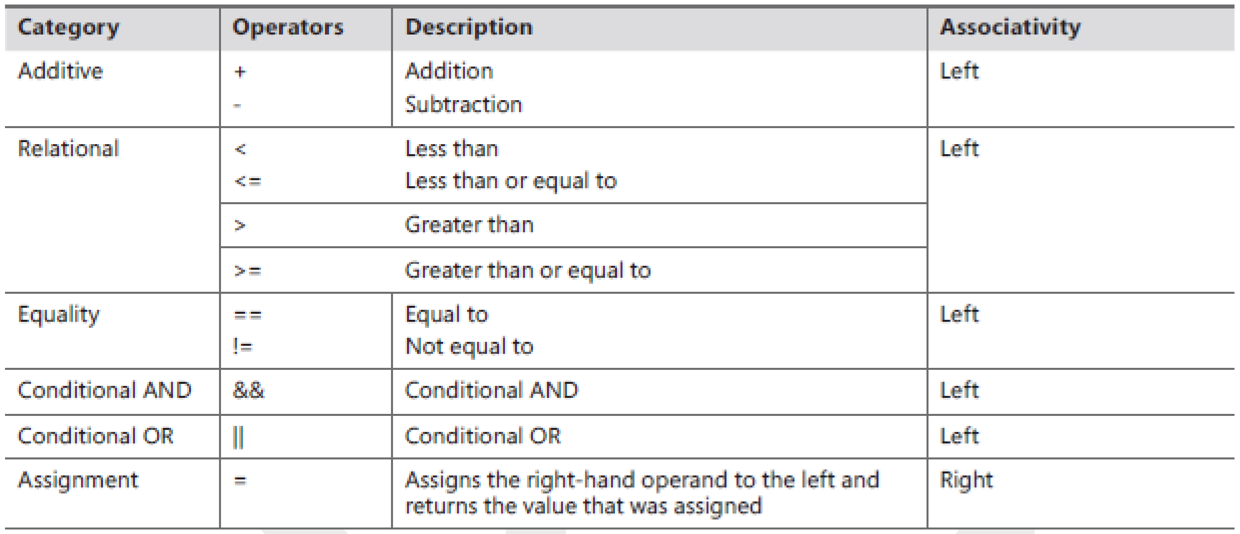
  (percent < 0) || (percent > 100)

- In this expression, if the value of *percent* is less than 0, the Boolean expression on the left side of || evaluates to *true*. This value means that the result of the entire expression must be *true* and the Boolean expression to the right of the || operator is not evaluated.

Operator precedence and associativity table



Category	Operators	Description	Associativity
Primary	() ++ --	Precedence override Post-increment Post-decrement	Left
Unary	! + - ++ --	Logical NOT Returns the value of the operand unchanged Returns the value of the operand negated Pre-increment Pre-decrement	Left
Multiplicative	* / %	Multiply Divide Division remainder (modulus)	Left



Category	Operators	Description	Associativity
Additive	+ -	Addition Subtraction	Left
Relational	< <=	Less than Less than or equal to	Left
	>	Greater than	
	>=	Greater than or equal to	
Equality	== !=	Equal to Not equal to	Left
Conditional AND	&&	Conditional AND	Left
Conditional OR	||	Conditional OR	Left
Assignment	=	Assigns the right-hand operand to the left and returns the value that was assigned	Right

Using *if* statements to make decisions

Understanding *if* statement syntax

The syntax of an *if* statement is as follows (*if* and *else* are C# keywords):

```
if ( booleanExpression )
statement-1;
else
statement-2;
```

- If *booleanExpression* evaluates to *true*, *statement-1* runs; otherwise, *statement-2* runs. The *else* keyword and the subsequent *statement-2* are optional. If there is no *else* clause and the *booleanExpression* is *false*, execution continues with whatever code follows the *if* statement. Also, notice that the Boolean expression must be enclosed in parentheses; otherwise, the code will not compile.

For example, here's an *if* statement that increments a variable representing the second hand of a stopwatch.

```
int a=15,b=15;
...
if (a ==b)
console.writeln(“equal”);
else
 console.writeln(“not equal”);
```

NOTE: Boolean expressions only: The expression in an *if* statement must be enclosed in parentheses. Additionally, the expression must be a Boolean expression. Sometimes, we'll want to perform more than one statement when a Boolean expression is true. We could group the statements inside a new method and then call the new method, but a simpler solution is to group the statements inside a *block*. A block is simply a sequence of statements grouped between an opening brace and a closing brace.

- In the following example, two statements that reset the *seconds* variable to 0 and increment the *minutes* variable are grouped inside a block, and the entire block executes if the value of *seconds* is equal to 59:

```
int seconds = 0;
int minutes = 0;...
if (seconds == 59)
   {
      seconds = 0;
       minutes++;
  }
Else
  {
      seconds++;
   }
```

- A block also starts a new scope. We can define variables inside a block, but they will disappear at the end of the block. The following code fragment illustrates this point:

```
      if (...)
             {
                int myVar = 0;
                 // myVar can be used here
                    ...
            } // myVar disappears here
     else
            {
                  // myVar cannot be used here ...
             }
                // myVar cannot be used here
```

Cascading *if* statements

- We can nest *if* statements inside other *if* statements. In this way, we can chain together a sequence of Boolean expressions, which are tested one after the other until one of them evaluates to *true*.
- In the following example, if the value of *day* is 0, the first test evaluates to *true* and *dayName* is assigned the string *"Sunday"*. If the value of *day* is not 0, the first test fails and control passes to the *else* clause, which runs the second *if* statement and compares the value of *day* with 1.

- The second *if* statement executes only if the first test is *false*. Similarly, the third *if* statement executes only if the first and second tests are *false*.

```
if (day == 0)
{
dayName = "Sunday";
}
else if (day == 1)
{
dayName = "Monday";
}
…..
else if (day == 6)
{
DayName = "Saturday";
}
else
{
dayName = "unknown";
}
```

Using *switch* statements

- Sometimes, when we write a cascading *if* statement, each of the *if* statements look similar because they all evaluate an identical expression. The only difference is that each *if* compares the result of the expression with a different value. For example, consider the following block of code that uses an *if* statement to examine the value in the *day* variable and work out which day of the week it is:

```
 if (day == 0)
 {
   dayName = "Sunday";
 }
    else if (day == 1)
{
   dayName = "Monday";
 }
        ….
 else
  {
dayName = "Unknown";
   }
```

In these situations, often we can rewrite the cascading *if* statement as a *switch* statement to make our program more efficient and more readable.

Understanding *switch* statement syntax

The syntax of a *switch* statement is as follows (*switch*, *case*, and *default* are keywords):

```
switch ( controllingExpression )
 {
    case constantExpression :
    statements
    break;
    case constantExpression :
    statements
     break;
      ...
     default :
     statements
      break;
    }
```

- The *controllingExpression*, which must be enclosed in parentheses, is evaluated once. Control then jumps to the block of code identified by the *constantExpression*, whose value is equal to the result of the *controllingExpression*. (The *constantExpression* identifier is also called a *case label*.)
- Execution runs as far as the *break* statement, at which point the *switch* statement finishes and the program continues at the first statement that follows the closing brace of the *switch* statement. If none of the *constantExpression* values is equal to the value of the *controllingExpression*, the statements below the optional *default* label run.

- **Following the *switch* statement rules**

1. The *switch* statement is very useful, but unfortunately, we can't always use it when we might like to. Any *switch* statement we write must adhere to the following rules:
2. We can use *switch* only on certain data types, such as *int*, *char*, or *string*. With any other types (including *float* and *double*), we must use an *if* statement.
3. The *case* labels must be constant expressions, such as 42 if the *switch* data type is an *int*, '4' if the *switch* data type is a *char*, or "42" if the *switch* data type is a *string*. If we need to calculate our *case* label values at run time, we must use an *if* statement.
4. The *case* labels must be unique expressions. In other words, two *case* labels cannot have the same value.
5. We can specify that we want to run the same statements for more than one value by providing a list of *case* labels and no intervening statements, in which case the code for the final label in the list is executed for all cases in that list.

- So, we can rewrite the previous cascading *if* statement as the following *switch* statement:

```
switch (day)
{
case 0 : dayName = "Sunday";
break;
case 1 : dayName = "Monday";
break;
case 2 : dayName = "Tuesday";
break;
...
default : dayName = "Unknown";
break;
}
```

MODULE 1[CHAPTER 5]
Using compound assignment and iteration statements

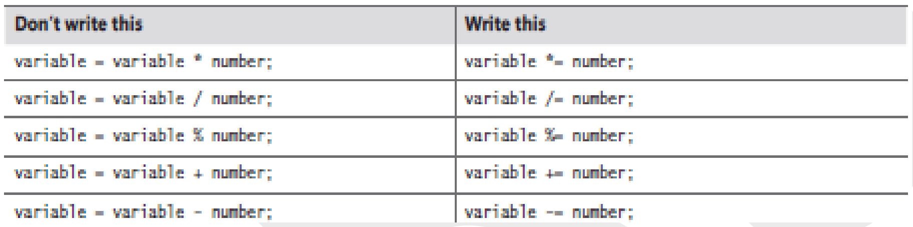
Using compound assignment operators

- When we write iteration statements, we usually need to control the number of iterations that we perform. We can achieve this by using a variable, updating its value as each iteration is performed, and stopping the process when the variable reaches a particular value.
- To help simplify this process, we'll start by learning about the special assignment operators that we should use to update the value of a variable in these circumstances.

For example, the following statement adds 42 to *ansour*. After this statement runs, the value of *ansour* is 42 more than it was before:

```
ansour = ansour + 42;
```

Following operators are collectively known as the *compound assignment operators*



Don't write this	Write this
variable = variable * number;	variable *= number;
variable = variable / number;	variable /= number;
variable = variable % number;	variable %= number;
variable = variable + number;	variable += number;
variable = variable - number;	variable -= number;

- The += operator also works on strings; it appends one string to the end of another. For example, the following code displays "Hello John" on the console:

```
string str1= "John";
string str2 = "Hello ";
str1 += str2;
Console.WriteLine(str1);
```

NOTE:We cannot use any of the other compound assignment operators on strings.

Writing *while* statements

- We use a *while* statement to run a statement repeatedly for as long as some condition is true. The syntax of a *while* statement is as follows:

```
Initialization
while (Boolean expression)
{
 Statements
 update control variable
}
```

- The Boolean expression is evaluated, and if it is true, the statement runs and then the Boolean expression is evaluated again. If the expression is still true, the statement is repeated until the Boolean expression evaluates to false, at which point the *while* statement exits. Execution then continues with the first statement that follows the *while* statement.
  1. The expression must be a Boolean expression.
  2. The Boolean expression must be written within parentheses.
  3. If the Boolean expression evaluates to false when first evaluated, the statement does not run.
  4. If we want to perform two or more statements under the control of a *while* statement, we must use braces to group those statements in a block.

- Example: *while statement that writes the values 0 through 9 to the console.*

```
int i = 0;
while (i < 10)
{
Console.WriteLine(i);
i++;
}
```

Writing *for* Statements

- The *for* statement in C# provides a more formal version of this kind of construct by combining the initialization, Boolean expression, and code that updates the control variable. Here is the syntax of a *for* statement:

```
for (initialization; Boolean expression; update control variable)
 statement
```

- The statement that forms the body of the *for* construct can be a single line of code or a code block enclosed in braces.
- Example : Displays the integers from 0 through 9 as the following *for* loop:

```
for (int i = 0; i < 10; i++)
{
 Console.WriteLine(i);
}
```

- Notice that the initialization occurs only once, that the statement in the body of the loop always executes before the update occurs, and that the update occurs before the Boolean expression reevaluates.
- We can omit any of the three parts of a for statements.

```
for (int i = 0; ;i++)
{
Console.WriteLine("somebody stop me!");
}
```

- If we omit the initialization and update parts, we have a strangely spelled *while* loop:

```
int i = 0;
for (; i < 10; )
{
 Console.WriteLine(i);
 i++;
}
```

- We can also provide multiple initializations and multiple updates in a *forloop*

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
 ...
}
```

Understanding *for* statement scope

- We can declare a variable in the initialization part of a *for* statement. That variable is scoped to the body of the *for* statement and disappears when the *for* statement finishes. This rule has two important consequences. First, we cannot use that variable after the *for* statement has ended because it's no longer in scope. Here's an example:

```
for (int i = 0; i < 10; i++)
{
...
}
Console.WriteLine(i); // compile-time error
```

- Second, we can write two or more *for* statements that reuse the same variable name because each variable is in a different scope, as shown in the following code:

```
for (int i = 0; i < 10; i++)
{ ...}
for (int i = 0; i < 20; i += 2)        // okay
{ ...}
```

Writing *do* statements

- The *while* and *for* statements both test their Boolean expression at the beginning of the loop. This means that if the expression evaluates to *false* on the first test, the body of the loop does not run—not even once. The *do* statement is different: its Boolean expression is evaluated after each iteration, so the body always executes at least once.
- The syntax of the *do* statement is as follows (don't forget the final semicolon):

```
Do
 Statement
while (booleanExpression);
```

- We must use a *statement* block if the body of the loop comprises more than one statement (the compiler will report a syntax error if we don't). Here's a version of the example that writes the values 0 through 9 to the console, this time constructed by using a *do* statement:

```
int i = 0;
do
{
Console.WriteLine(i);
 i++;
}while (i < 10);
```

MODULE 1[CHAPTER 6]
Managing errors and exceptions

Coping with errors

- Errors can occur at almost any stage when a program runs, and many of them might not actually be the fault of our own application, so how do we detect them and attempt to recover?

- A typical approach adopted by older systems such as UNIX involved arranging for the operating system to set a special global variable whenever a method failed. Then, after each call to a method, we checked the global variable to see whether the method succeeded. C# and most other modern object-oriented languages don't handle errors in this manner;. Instead, they use *exceptions*.

Trying code and catching exceptions

- C# makes it easy to separate the error-handling code from the code that implements the primary logic of a program by using exceptions and exception handlers. To write exception-aware programs, we need to do two things:
- Write our code within a *try* block (*try* is a C# keyword). When the code runs, it attempts to execute all the statements in the *try* block, and if none of the statements generates an exception, they all run, one after the other, to completion.
- However, if an error condition occurs, execution jumps out of the *try* block and into another piece of code designed to catch and handle the exception—a *catch* handler.
- Write one or more *catch* handlers (*catch* is another C# keyword) immediately after the *try* block to handle any possible error conditions. A *catch* handler is intended to capture and handle a specific type of exception, and we can have multiple *catch* handlers after a *try* block, each one designed to trap and process a specific exception;
- we can provide different handlers for the different errors that could arise in the *try* block. If any one of the statements within the *try* block causes an error, the runtime throws an exception. The runtime then examines the *catch* handlers after the *try* block and transfers control directly to the first matching handler.

- Here's an example of code in a *try* block that attempts to convert strings that a user has typed in some text boxes on a form to integer values, call a method to calculate a value, and write the result to another text box.

- Converting a string to an integer requires that the string contain a valid set of digits and not some arbitrary sequence of characters. If the string contains invalid characters, the *int.Parse* method throws a *FormatException*, and execution transfers to the corresponding *catch* handler. When the *catch* handler finishes, the program continues with the first statement that follows the handler. Note that if there is no handler that corresponds to the exception, the exception is said to be unhandled

```
try
  {
 int leftHandSide = int.Parse(lhsOperand.Text);
 int rightHandSide = int.Parse(rhsOperand.Text);
 int ansour = doCalculation(leftHandSide, rightHandSide);
 result.Text = ansour.ToString();
  }
 catch (FormatException fEx)
  {
 // Handle the exception
 ...
    }
```

- A *catch* handler employs syntax similar to that used by a method parameter to specify the exception to be caught. In the preceding example, when a *FormatException* is thrown, the *fEx* variable is populated with an object containing the details of the exception.
- The *FormatException* type has a number of properties that we can examine to determine the exact cause of the exception. Many of these properties are common to all exceptions..

Unhandled Exceptions

- What happens if a *try* block throws an exception and there is no corresponding *catch* handler? [refer previous example] it is possible that the *lhsOperand* text box contains the string representation of a valid integer but the integer it represents is outside the range of valid integers supported by C# (for example, "2147483648").
- In this case, the *int.Parse* statement throws an *OverflowException*, which will not be caught by the *FormatException catch* handler. If this occurs and the *try* block is part of a method, the method immediately exits and execution returns to the calling method.
- If the calling method uses a *try* block, the runtime attempts to locate a matching *catch* handler for this *try* block and execute it. If the calling method does not use a *try* block or there is no matching *catch* handler, the calling method immediately exits and execution returns to its caller, where the process is repeated.
- If a matching *catch* handler is eventually found, the handler runs and execution continues with the first statement that follows the *catch* handler in the catching method.

Using multiple catch handlers

To represent different kinds of failures. we can supply multiple *catch* handlers, one after the other, such as in the following:

```
Try
{
int leftHandSide = int.Parse(lhsOperand.Text);
 int rightHandSide = int.Parse(rhsOperand.Text);
 int ansour = doCalculation(leftHandSide, rightHandSide);
 result.Text = ansour.ToString();
}
catch (FormatException fEx)
{ //...}
catch (OverflowException oEx)
{ //...}
```

- If the code in the *try* block throws a *FormatException* exception, the statements in the *catch* block for the *FormatException* exception run. If the code throws an *OverflowException* exception, the *catch* block for the *OverflowException* exception runs.

Catching multiple exceptions

The exception-catching mechanism provided by C# and the Microsoft .NET Framework is quite comprehensive. The .NET Framework defines many types of exceptions, and any programs we write can throw most of them. It is highly unlikely that we will want to write *catch* handlers for every possible exception that our code can throw—remember that our application must be able to handle exceptions that we might not have even considered when we wrote it! So, how do we ensure that our programs catch and handle all possible exceptions?

The ansour to this question lies in the way the different exceptions are related to one another. Exceptions are organized into families called *inheritance hierarchies*. (We will learn about inheritance in Chapter 12, "Working with inheritance.") *FormatException* and *OverflowException* both belong to a family called *SystemException*, as do a number of other exceptions. *SystemException* is itself a member of a wider family simply called *Exception*, and this is the great-granddaddy of all exceptions. If we catch *Exception*, the handler traps every possible exception that can occur.

Note The *Exception* family includes a wide variety of exceptions, many of which are intend-ed for use by various parts of the .NET Framework. Some of these exceptions are somewhat esoteric, but it is still useful to understand how to catch them.

The next example shows how to catch all possible exceptions:

```
try{ int leftHandSide = int.Parse(lhsOperand.Text); int rightHandSide = int.Parse(rhsOperand.Text); int ansour = doCalculation(leftHandSide, rightHandSide); result.Text = ansour.ToString();}catch (Exception ex) // this is a general catch handler{ //...}
```

T**ip** If we want to catch *Exception*, we can actually omit its name from the *catch* handler because it is the default exception:catch { // ... }However, this is not recommended. The exception object passed in to the *catch* handler can contain useful information concerning the exception, which is not easily accessible when using this version of the *catch* construct.

There is one final question we should be asking at this point: What happens if the same exception matches multiple *catch* handlers at the end of a *try* block? If we catch *FormatException* and *Exception* in two different handlers, which one will run? (Or, will both execute?)

When an exception occurs, the runtime uses the first handler it finds that matches the exception, and the others are ignored. This means that if we place a handler for *Exception* before a handler for *FormatException*, the *FormatException* handler will never run. Therefore, we should place more specific *catch* handlers above a general *catch* handler after a *try* block. If none of the specific *catch* handlers matches the exception, the general *catch* handler will.

In the following exercises, we will see what happens when an application throws an unhandled exception, and then we will write a *try* block and catch and handle an exception.

Using checked and unchecked integer arithmetic

Chapter 2 discusses how to use binary arithmetic operators such as + and * on primitive data types such as *int* and *double*. It also instructs that the primitive data types have a fixed size. For example, a C# *int* is 32 bits. Because *int* has a fixed size, we know exactly the range of value that it can hold: it is –2147483648 to 2147483647.

T**ip** If we want to refer to the minimum or maximum value of *int* in code, we can use the *int.MinValue* or *int.MaxValue* property.

The fixed size of the *int* type creates a problem. For example, what happens if we add 1 to an *int* whose value is currently 2147483647? The ansour is that it depends on how the application is compiled. By default, the C# compiler generates code that allows the calculation to overflow silently and we get the wrong ansour. (In fact, the calculation wraps around to the largest negative integer value, and the result generated is –2147483648.) The reason for this behavior is performance: integer

arithmetic is a common operation in almost every program, and adding the overhead of overflow checking to each integer expression could lead to very poor performance. In many cases, the risk is acceptable because we know (or hope!) that our *int* values won't reach their limits. If we don't like this approach, we can turn on overflow checking.

T**ip** We can turn on and off overflow checking in Visual Studio 2013 by setting the project properties. In Solution Explorer, click *OurProject* (where *OurProject* is the actual name of the project). On the Project menu, click *OurProject* Properties. In the project properties dialog box, click the Build tab. Click the Advanced button in the loour-right corner of the page. In the Advanced Build Settings dialog box, select or clear the Check for Arithmetic Overflow/Underflow check box.

Regardless of how we compile an application, we can use the *checked* and *unchecked* keywords to turn on and off integer arithmetic overflow checking selectively in parts of an application that we think need it. These keywords override the compiler option specified for the project.

Writing checked statements

A checked statement is a block preceded by the *checked* keyword. All integer arithmetic in a checked statement always throws an *OverflowException* if an integer calculation in the block overflows, as shown in this example:

int number = int.MaxValue;checked{ int willThrow = number++; Console.WriteLine("this won't be reached");}

Important Only integer arithmetic directly inside the *checked* block is subject to overflow checking. For example, if one of the checked statements is a method call, checking does not apply to code that runs in the method that is called.

We can also use the *unchecked* keyword to create an *unchecked* block statement. All integer arithmetic in an *unchecked* block is not checked and never throws an *OverflowException*. For example:

Throwing exceptions

Suppose that we are implementing a method called *monthName* that accepts a single *int* argument and returns the name of the corresponding month. For example, *monthName(1)* returns "January", *monthName(2)* returns "February", and so on. The question is, what should the method return if the integer argument is less than 1 or greater than 12? The best ansour is that the method shouldn't return anything at all—it should throw an exception. The .NET Framework class libraries contain lots of exception classes specifically designed for situations such as this. Most of the time, we will find that one of these classes describes our exceptional condition. (If not, we can easily create our own exception class, but we need to know a bit more about the C# language before we can do that.) In this case, the existing .NET Framework *ArgumentOutOfRangeException* class is just right. We can throw an exception by using the *throw* statement, as shown in the following example:

public static string monthName(int month){ switch (month) { case 1 : return "January"; case 2 : return "February"; ... case 12 : return "December"; default : throw new ArgumentOutOfRangeException("Bad month"); }}

The *throw* statement needs an exception object to throw. This object contains the details of the exception, including any error messages. This example uses an expression that creates a new *ArgumentOutOfRangeException* object. The object is initialized with a string that populates its *Message* property by using a constructor. Constructors are covered in detail in Chapter 7, "Creating and managing classes and objects."

In the following exercises, we will modify the MathsOperators project to throw an exception if the user attempts to perform a calculation without specifying an operator.

Note This exercise is a little contrived, as any good application design would provide a de-fault operator, but this application is intended to illustrate a point.

Using a *finally* block

It is important to remember that when an exception is thrown, it changes the flow of execution through the program. This means that we can't guarantee a statement will always run when the previous statement finishes because the previous statement might throw an exception. Remember that in this case, after the *catch* handler has run, the flow of control resumes at the next statement in the block holding this handler and not at the statement immediately following the code that raised the exception.

Look at the example that follows, which is adapted from the code in Chapter 5, "Using compound assignment and iteration statements." It's very easy to assume that the call to *reader.Dispose* will always occur when the *while* loop completes (if we are using Windows 7 or Windows 8, we can replace *reader.Dispose* with *reader.Close* in this example). After all, it's right there in the code.

TextReader reader = ...; ... string line = reader.ReadLine(); while (line != null) { ... line = reader.ReadLine(); } reader.Dispose();

Sometimes it's not an issue if one particular statement does not run, but on many occasions it can be a big problem. If the statement releases a resource that was acquired in a previous statement, failing to execute this statement results in the resource being retained. This example is just such a case: when we open a file for reading, this operation acquires a resource (a file handle), and we must ensure that we call *reader.Dispose* to release the resource (*reader.Close* actually calls *reader.Dispose* in Windows 7 and Windows 8 to do this). If we don't, sooner or later we'll run out of file handles and be unable to open more files. If we find file handles are too trivial, think of database connections, instead.

The way to ensure that a statement is always run, whether or not an exception has been thrown, is to write that statement inside a *finally* block. A *finally* block occurs immediately after a *try* block or immediately after the last *catch* handler after a *try* block. As long as the program enters the *try* block associated with a *finally* block, the *finally* block will always be run, even if an exception occurs. If an exception is thrown and caught locally, the exception handler executes first, followed by the *finally* block. If the exception is not caught locally (that is, the runtime has to search through the list of calling methods to find a handler), the *finally* block runs first. In any case, the *finally* block always executes.

The solution to the *reader.Close* problem is as follows:

TextReader reader = ...;...try{ string line = reader.ReadLine(); while (line != null) { ... line = reader.ReadLine(); }}finally{ if (reader != null) { reader.Dispose(); }}

Even if an exception occurs while reading the file, the *finally* block ensures that the *reader.Dispose* statement always executes. We'll see another way to handle this situation in Chapter 14, "Using garbage collection and resource management."

MODULE-2 [CHAPTER-1]

CREATING AND MANAGING CLASSES AND OBJECTS

Understanding classification

Class is the root word of the term classification. When we design a class, we systematically arrange information and behaviour into a meaningful entity. For example, all cars share common behaviours (they can be steered, stopped, accelerated, and so on) and common attributes (they have a steering wheel, an engine, and so on). People use the word car to mean an object that shares these common behaviours and attributes. Without classification, it's hard to imagine how people could think or communicate at all.

The purpose of encapsulation

Encapsulation is an important principle when defining classes. Encapsulation actually has two purposes:
■ To combine methods and data within a class; in other words, to support classification
■ To control the accessibility of the methods and data; in other words, to control the use of the class

Defining and using a class

In C#, we use the *class* keyword to define a new class. The data and methods of the class occur in the body of the class between a pair of braces. Following is a C# class called *Circle* that contains one method (to calculate the circle's area) and one piece of data (the circle's radius):

```
class Circle
{
        int radius;
        double Area()
        {
        return Math.PI * radius * radius;
        }
}
```

- We create a variable specifying *Circle* as its type, and then we initialize the variable with some valid data. Here is an example:
  ```
  Circle c; // Create a Circle variable
   c = new Circle(); // Initialize it
  ```
- We cannot write a statement such as this because it gives an error:
  ```
  Circle c;
  c = 42;
  ```
- We can directly assign an instance of a class to another variable of the same type, like this:
  Circle c;
  ```
  c = new Circle();
  ```

```
Circle d;
d = c;
```

Controlling accessibility

■ A method or field is private if it is accessible only from within the class. To declare that a method or field is private, we write the keyword *private* before its declaration. As intimated previously, this is actually the default, but it is good practice to state explicitly that fields and methods are private to avoid any confusion.

■ A method or field is public if it is accessible from both within and outside of the class. To declare that a method or field is public, we write the keyword *public* before its declaration.

Area is declared as a public method and *radius* is declared as a private field:

```
class Circle
    {
        private int radius;
        public double Area()
        {
        return Math.PI * radius * radius;
        }
    }
```

- *radius* is declared as a private field and is not accessible from outside the class, *radius* is accessible from within the *Circle* class. The Area method is inside the *Circle* class, so the body of Area has access to *radius*.

Working with constructors

When we use the *new* keyword to create an object, the runtime needs to construct that object by using the definition of the class. The runtime must grab a piece of memory from the operating system, fill it with the fields defined by the class, and then invoke a constructor to perform any initialization required.

- A constructor is a special method that runs automatically when we create an instance of a class. It has the same name as the class, and it can take parameters, but it cannot return a value (not even void). Every class must have a constructor. If we don't write one, the compiler automatically generates a default constructor for us. (However, the compiler-generated default constructor doesn't actually do anything.)
- We can write our own default constructor quite easily. Just add a public method that does not return a value and give it the same name as the class. The following example shows the *Circle* class with a default constructor that initializes the *radius* field to 0:

```
    class Circle
{
        private int radius;
        public Circle() // default constructor
        {
                radius = 0;    }
        public double Area()
        {
        return Math.PI * radius * radius;}
}
```

- We use dot notation to invoke the *Area* method on a *Circle* object:

```
Circle c;
c = new Circle();
double areaOfCircle = c.Area();
```

Overloading constructors

Overloading constructor is a constructor where parameters are passed to it is called overloaded constructor. A constructor is just a special kind of method and it—like all methods—can be overloaded. We can add another constructor to the *Circle* class, with a parameter that specifies the radius to use, like this:

```
 class Circle
{
        private int radius;
        public Circle() // default constructor
        {
                radius = 0;
        }
        public Circle(int initialRadius) // overloaded constructor
        {
                radius = initialRadius;
        }
        public double Area()
        {
                return Math.PI * radius * radius;
        }
}
```

- We can then use this constructor when creating a new *Circle* object, such as in the following:

```
Circle c;
c = new Circle(45);
```

- The compiler works out which constructor it should call based on the parameters that we specify to the *new* operator.

NOTE: If we write wer own constructor for a class, the compiler does not generate a default constructor.

Partial classes

- A class can contain a number of methods, fields, and constructors, as well as other items
- We can split the source code for a class into separate files so that we can organize the definition of a large class into smaller, easier to manage pieces.
- When we split a class across multiple files, we define the parts of the class by using the *partial* keyword in each file.

For example:

```
partial class Circle
        {
 public Circle() // default constructor
             {                    this.radius = 0;              }

public Circle(int initialRadius) // overloaded constructor
        {         this.radius = initialRadius;                      }
 }
```

- The contents look like this:

```
partial class Circle
{
private int radius;
public double Area()
{
 return Math.PI * this.radius * this.radius;
}
}
```

- When we compile a class that has been split into separate files, we must provide all the files to the compiler.

Understanding static methods and data

In C#, all methods must be declared within a class. However, if we declare a method or a field as static, we can call the method or access the field by using the name of the class. No instance is required. This is how the *Sqrt* method of the *Math* class is declared:

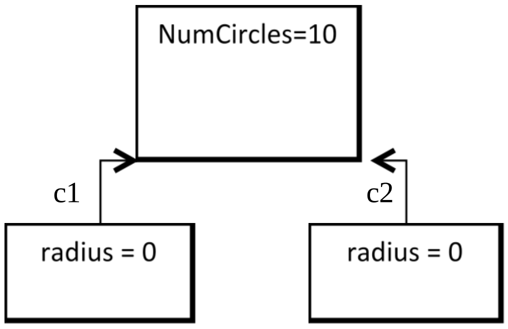
```
class Math
{
public static double Sqrt(double d)
{    ...     }
...}
```

- A static method does not depend on an instance of the class, and it cannot access any instance fields or instance methods defined in the class; it can use only fields and other methods that are marked as *static*.

Creating a shared field

Defining a field as static makes it possible for we to create a single instance of a field that is shared among all objects created from a single class.

```
class Circle
{
        private int radius;
        public static int NumCircles =10;
        public Circle() // default constructor
        {
                radius = 0;
                NumCircles++;
        }
        public static main()
        {       Circle c1=new Circle();
                Circle c2=new Circle();
        }}
```



- We can access the *NumCircles* field from outside of the class by specifying the *Circle* class rather than a *Circle* object, such as in the following example:

```
Console.WriteLine("Number of Circle objects: {0}", Circle.NumCircles);
```

Creating a static field by using the *const* keyword

- By prefixing the field with the *const* keyword, we can declare that a field is static but that its value can never change.
- The keyword *const* is short for constant.
- We can declare a field as const only when the field is a numeric type.
- Once the value is declared as *const*, it cannot be changed throughout the execution of program. Whereas *static* value changes.

Understanding *static* classes

- A *static* class can contain only *static* members.
- The purpose of a *static* class is purely to act as a holder of utility methods and fields. A *static* class cannot contain any instance data or methods, and it does not make sense to try to create an object from a *static* class by using the *new* operator.
- If we are defining our own version of the Math class, one containing only static members, it could look like this:

For example,

```
● public static class Math
{       public static double Sin(double x) {...}
        public static double Cos(double x) {...}
        public static double Sqrt(double x) {...}
        ...
}
● public static class Circle
  {
        public static num=10;
        public static method()
        {
                ….
        }
  }
        Console.writeLine("value:{0}", Circle.num);
        Circle.method();
```

<u>Anonymous classes</u>

An *anonymous* class is a class that does not have a name.

- ❖ We create an anonymous class simply by using the *new* keyword and a pair of braces defining the fields and values that we want the class to contain, like this:
  myAnonymousObject = new { Name = "John", Age = 47 };
  This class contains two public fields called Name (initialized to the string "John") and Age (initialized to the integer 47).

- ❖ When we define an anonymous class, the compiler generates its own name for the class, but it won't tell we what it is.

- ❖ We declare *myAnonymousObject* as an implicitly typed variable by using the *var* keyword, like this:
  var myAnonymousObject = new { Name = "John", Age = 47 };
  Remember that the *var* keyword causes the compiler to create a variable of the same type as the expression used to initialize it.

- ❖ We can access the fields in the object by using the familiar dot notation, as is demonstrated here:
  Console.WriteLine("Name: {0} Age: {1}", myAnonymousObject.Name, myAnonymousObject.Age};

- ❖ We can even create other instances of the same anonymous class but with different values, such as in the following:
  var anotherAnonymousObject = new { Name = "Diana", Age = 46 };

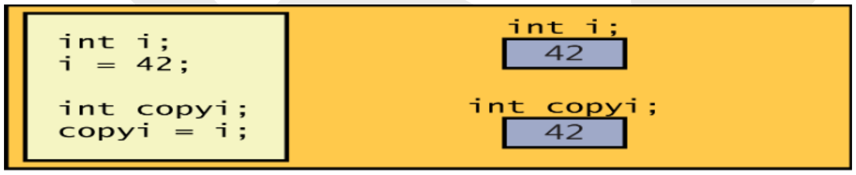
MODULE-2 [CHAPTER-2]

UNDERSTANDING VALUES AND REFERENCES

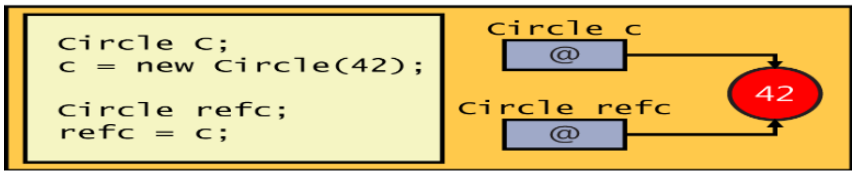
Copying value type variables and classes

- VALUE TYPE: Most of the primitive types built into C#, such as *int*, *float*, *double*, and *char* are collectively called *value types*. These types have a fixed size, and when we declare a variable as a value type, the compiler generates code that allocates a block of memory big enough to hold a corresponding value in *stack*.
- REFERENCE TYPE: Class types such as *Circle* are handled differently. When we declare a *Circle* variable, the compiler *does not* generate code that allocates a block of memory big enough to hold a Circle; all it does is allot a small piece of memory that can potentially hold the address of (or a reference to) another block of memory containing a Circle. A class is an example of a reference type. Reference types hold references to blocks of memory.

Eq: int i = 42; // declare and initialize i
int copyi = i; /* copyi contains a copy of the data in i:i and copyi both contain the value 42 */
i++; /* incrementing i has no effect on copyi; i now contains 43, but copyi still contains 42 */
Circle c = new Circle(42);
Circle refc = c;
refc.Circle(53); // Effects to both c and refc



```
int i;
i = 42;

int copyi;
copyi = i;
```



```
Circle c;
c = new Circle(42);

Circle refc;
refc = c;
```

Understanding null values and nullable types

➢ When we declare a variable, it is always a good idea to initialize it. With *value* types,
int i = 0;
double d = 0.0;

➢ Let's initialize *reference* type
Circle c = new Circle(42);
Circle copy = new Circle(99); // Some random value, for initializing copy
...
copy = c; // copy and c refer to the same object.

After assigning c to copy, what happens to the original Circle object with a radius of 99 that we used to initialize copy? Nothing refers to it anymore. In this situation, the runtime error occurs.

In C#, we can assign the null value to any reference variable. The null value simply means that the variable does not refer to an object in memory. We can use it like this:

```
Circle c = new Circle(42);
Circle copy = null; // Initialized
...
if (copy == null)
{
copy = c; // copy and c refer to the same object
...
}
```

Using nullable types

- ❖ The *null* value is useful for initializing reference types. Sometimes, we need an equivalent value for value types, but *null* is itself a reference, and so we cannot assign it to a value type. The following statement is therefore illegal in C#:
  int i = null; // illegal
- ❖ A nullable value type behaves in a similar manner to the original value type, but we can assign the *null* value to it. We use the question mark (?) to indicate that a value type is nullable, like this:
  int? i = null; // legal
- ❖ We can ascertain whether a nullable variable contains *null* by testing it in the same way as a reference type.
  if (i == null)
  ...
- ❖ We can assign an expression of the appropriate value type directly to a nullable variable. The following examples are all legal:
  int? i = null;
  int j = 99;
  i = 100; // Copy a value type constant to a nullable type
  i = j; // Copy a value type variable to a nullable type
- ❖ j = i; // Illegal
  This makes sense if we consider that the variable *i* might contain *null*, and *j* is a value-type that cannot contain *null*. This also means that we cannot use a nullable variable as a parameter to a method that expects an ordinary value type.

  int? i = 99;

Pass.Value(i); // Compiler error: **Because we cannot pass nullable to function call**

Understanding the properties of nullable types

A nullable type has a pair of properties:

- The HasValue property indicates whether a nullable type contains a value or is null.
- We can retrieve the value of a non-null nullable type by reading the Value property, like this:

```
int? i = null;
if (!i.HasValue)
{
i = 99;          // If i is null, then assign it the value 99
}
else
 {
Console.WriteLine(i.Value);  // If i is not null, then display its value
  }
```

This example tests the nullable variable i, and if it does not have a value (it is null), it assigns it the value 99; otherwise, it displays the value of the variable.

Using ref and out parameters

When we pass an argument to a method, the corresponding parameter is initialized with a copy of the argument. It's impossible for any change to the parameter to affect the value of the argument passed in. For example, in the following code, the value output to the console is 42 and not 43. The doIncrement method increments a copy of the argument (arg) and not the original argument,

```
static void doIncrement(int param)
{           param++;          }
static void Main()
{
int arg = 42;
doIncrement(arg);
Console.WriteLine(arg); // writes 42, not 43
}
```

- The key point is this: Although the data that was referenced changed, the argument passed in as the parameter did not—it still references the same object.
- We might want to write a method that actually needs to modify an argument. C# provides the *ref* and *out* keywords so that we can do this.

Creating ref parameters

- If we *prefix* a parameter with the *ref* keyword, the C# compiler generates code that passes a reference to the actual argument rather than a copy of the argument.

- While using a *ref* parameter, anything we do to the parameter, we also do to the original argument because the parameter and the argument both reference the same data.
- When we pass an argument as a ref parameter, we must also prefix the argument with the *ref* keyword.

```
static void doIncrement(ref int param) // using ref
{                    param++;          }
static void Main()
{
int arg = 42;
doIncrement(ref arg); // using ref
Console.WriteLine(arg); // writes 43
}
```

- This time, the doIncrement method receives a reference to the original argument rather than a copy, so any changes the method makes by using this reference actually change the original value.
- For example, this operation is allowed only if arg has a defined value:

```
static void doIncrement(ref int param)
{
param++;
}
static void Main()
{
int arg; // not initialized
doIncrement(ref arg);
Console.WriteLine(arg);//compile time error
}
```

NOTE: Remember that C# enforces the rule that we cannot pass an uninitialized value as an argument to a method even if an argument is defined as a ref argument.

Creating out parameters

- The times when we want the method itself to initialize the parameter. We can do this with the *out* keyword.
- We can *prefix* a parameter with the *out* keyword so that the parameter becomes an alias for the argument.
- When we pass an *out* parameter to a method, the method must assign a value to it before it finishes or retur
- An out parameter must be assigned a value by the method; we're allowed to call the method without initializing its argument. For example,

```
static void doInitialize(out int param)
{
param = 42;
}
static void Main()
```

```
{
int arg; // not initialized
doInitialize(out arg); // legal
Console.WriteLine(arg); // writes 42
}
```

How computer memory is organized

Computers use memory to hold programs that are being executed and the data that these programs use. To understand the differences between *value* and *reference* types, it is helpful to understand how data is organized in memory.

The two areas of memory are traditionally called the *stack* and the *heap*.

■ When we call a method, the memory required for its parameters and its local variables is always acquired from the *stack*.

■ When we create an object (an instance of a class) by using the *new* keyword, the memory required to build the object is always acquired from the *heap*.

The names *stack* and *heap* come from the way in which the runtime manages the memory:

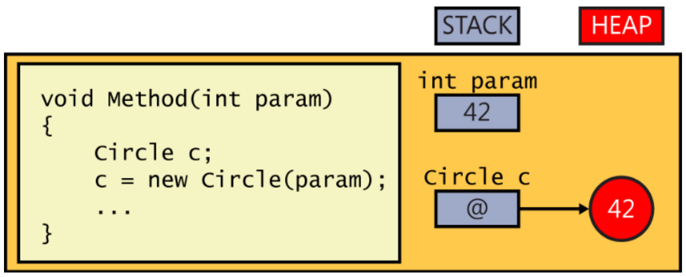
- ➢ *Stack memory* is organized like a stack of boxes piled on top of one another. When a method is called, each parameter is put in a box that is placed on top of the stack.
- ➢ *Heap memory* is like a large pile of boxes around a room rather than stacked neatly on top of each other. Each box has a label indicating whether it is in use. When a new object is created, the runtime searches for an empty box and allocate it to the object. The reference to the object is stored in a local variable on the *stack*. When the last reference disappears, the runtime marks the box as not in use.

Using the stack and the heap

Let's examine what happens when the following method Method is called:

```
void Method(int param)
{
        Circle c;
        c = new Circle(param);
        ...
}
```

Suppose the argument passed into *param* is the value 42. When the method is called, a block of memory is allocated from the *stack* and initialized with the value 42. *Circle* object is allocated from the *heap*.



```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    ...
}
```

NOTE: We should note two things:

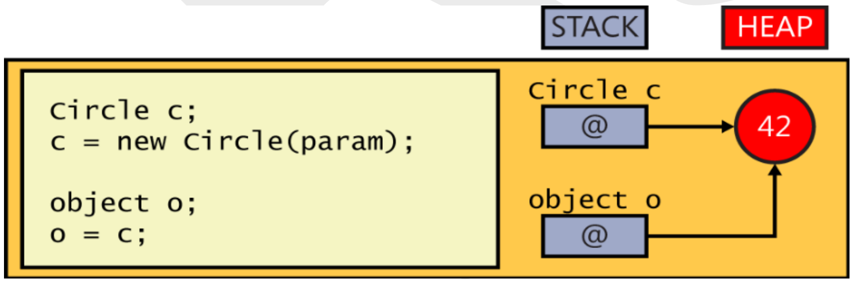
- ➢ Although the object is stored on the heap, the reference to the object (the variable c) is stored on the stack.
- ➢ Heap memory is not infinite. If heap memory is exhausted, the new operator will throw an OutOfMemoryException exception and the object will not be created.

The *System.Object* class

- We can use *System.Object* to create a variable that can refer to any reference type.
- *System.Object* is an important class that C# provides the *object* keyword as an alias for *System.Object*.

In the following example, the variables c and o both refer to the same Circle object.

Circle c;
c = new Circle(42);
object o;
o = c;



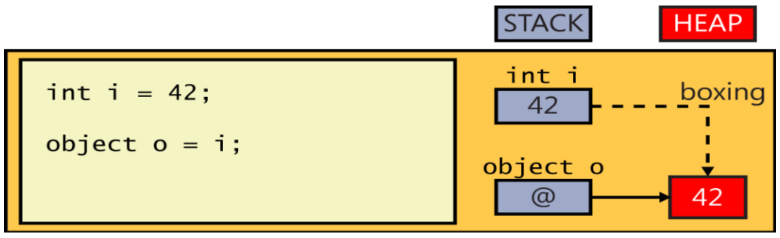
```
Circle c;
c = new Circle(param);

object o;
o = c;
```

Boxing

Variables of type object can refer to any item of any reference type. However, variables of type object can also refer to a value type. For example, the following two statements initialize the variable i to 42 and then initialize the variable o to i:

int i = 42;
object o = i;

Remember that ‘i’ is a value type and that it lives on the *stack*. If the reference inside o referred directly to i, the reference would refer to the *stack*. However, all references must refer to objects on the heap; the runtime allocates a piece of memory from the heap, copies the value of integer i to this piece of memory, and then refers the object o to this copy. This automatic copying of an item from the stack to the heap is called *boxing*.



Unboxing

We might expect to be able to access the boxed *int* value that a variable o refers to by using a simple assignment statement

int i = o;//**compile time error**

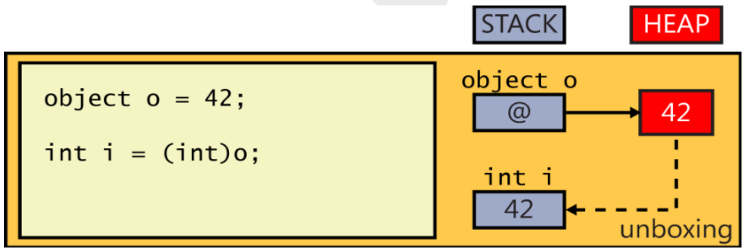
o could be referencing absolutely anything and not just an int. Consider the following code:

```
Circle c = new Circle();
int i = 42;
object o;
o = c; // o refers to a circle
i = o; // what is stored in i?
```

To obtain the value of the boxed copy, we must use a *cast*. This is an operation that checks whether it is safe to convert an item of one type to another before it actually makes the copy.

```
int i = 42;
object o = i; // boxes
i = (int)o; // compiles okay
```

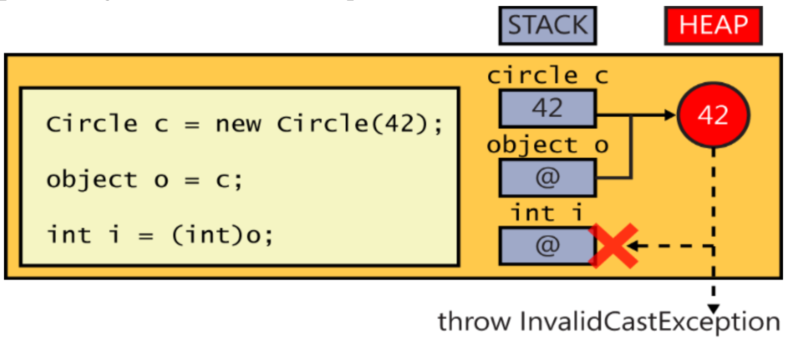
The effect of this *cast* is okay. If o really does refer to a boxed int and everything matches, the cast succeeds and the compiler-generated code extracts the value from the boxed int and copies it to i. This is called *unboxing*.



If o does not refer to a boxed int, there is a type mismatch, causing the cast to fail. The compiler-generated code throws an InvalidCastException exception at run time. An example of unboxing cast that fails:

```
Circle c = new Circle(42);
object o = c; // doesn't box because Circle is a reference variable
```

int i = (int)o; // compiles okay but throws an exception at run time



Casting data safely

If the type of object in memory does not match the cast, the runtime will throw an InvalidCastException, We should be prepared to catch this exception and handle it appropriately if it occurs.

➢ C# provides two more very useful operators that can help we perform casting in a much more elegant manner: the *is* and *as* operators.

The *is* operator

The *is* operator takes two operands: a reference to an object on the left ,and the name of a type on the right. If the type of the object referenced on the heap has the specified type, is evaluates to *true*; otherwise, is evaluates to *false*.
The preceding code attempts to cast the reference to the object variable o only if it knows that the cast will succeed.

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
if (o is WrappedInt)
{
        WrappedInt temp = (WrappedInt)o; // This is safe; o is a WrappedInt
        ...
}
```

The *as* operator

The *as* operator fulfills a similar role to is but in a slightly truncated manner.

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
```

```
WrappedInt temp = o as WrappedInt;
if (temp != null)
{
	... // Cast was successful
}
```

The *as* operator takes an object and a type as its operands. The runtime attempts to cast the object to the specified type. If the cast is successful, the result(object) is returned and, in this example, it is assigned to the WrappedInt variable temp. If the cast is unsuccessful, the *as* operator evaluates to the *null* value and assigns that to temp instead.

MODULE-2 [CHAPTER-3]

CREATING VALUE TYPES WITH ENUMERATIONS AND STRUCTURES

Working with enumerations

Suppose that we want to represent the seasons of the year in a program. We could use the integers 0, 1, 2, and 3 to represent spring, summer, fall, and winter, respectively. This system would work, but it’s not very intuitive. C# offers a better solution. We can create an enumeration (sometimes called an *enum* type), whose values are limited to a set of symbolic names.

Declaring an enumeration

We define an enumeration by using the *enum* keyword, followed by a set of symbols identifying the legal values that the type can have, enclosed between braces. Here’s how to declare an enumeration named Season whose literal values are limited to the symbolic names Spring, Summer, Fall, and Winter:

```
enum Season { Spring, Summer, Fall, Winter }
```

Using an enumeration

If the name of our enumeration is Season, we can create variables of type Season, fields of type Season and parameters of type Season, as in this example:

```
enum Season { Spring, Summer, Fall, Winter }
class Example
{
        public void Method(Season parameter) // method parameter example
        {
        Season localVariable; // local variable example
        ...
        }
        private Season currentSeason; // field example
}
```

❖ We can assign a value that is defined by the enumeration only to an enumeration variable, as is illustrated here:

```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // writes out 'Fall'
```

This is useful because it makes it possible for different enumerations to coincidentally contain literals with the same name.

❖ We can explicitly convert an enumeration variable to a string that represents its current value by using the built-in ToString method that all enumerations automatically contain.

```
string name = colorful.ToString();
Console.WriteLine(name); // also writes out 'Fall'
```

Choosing enumeration literal values

An enumeration type associates an integer value with each element of the enumeration. By default, the numbering starts at 0 for the first element and goes up in steps of 1. It's possible to retrieve the underlying integer value of an enumeration variable. To do this, we must cast it to its underlying type.

```
enum Season { Spring, Summer, Fall, Winter }
...
Season colorful = Season.Fall;
Console.WriteLine((int)colorful); // writes out '2'
```

- ❖ We can associate a specific integer constant (such as 1) with an enumeration literal (such as Spring).
  enum Season { Spring = 1, Summer, Fall, Winter }

In the above example, the underlying values of Spring, Summer, Fall, and Winter are now 1, 2, 3, and 4.

- ❖ We are allowed to give more than one enumeration literal the same underlying value. For example, in the United Kingdom, Fall is referred to as Autumn. For example:
  enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }

Choosing an enumeration's underlying type

- ❖ When we declare an enumeration, the enumeration literals are given values of type *int*. We can also choose different underlying integer type.

For example, to declare that Season's underlying type is a short rather than an int, we can write this:

```
enum Season : short { Spring, Summer, Fall, Winter }
```

- ❖ The main reason for doing this is to save memory; an *int* occupies more memory than a *short*.
- ❖ We can base an enumeration on any of the eight integer types: *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long*, or *ulong*.

Working with structures

- ❖ A structure is a *value* type. Because structures are stored on the *stack*, as long as the structure is reasonably small, the memory management overhead is often reduced.
- ❖ Like a class, a structure can have its own fields, methods, and constructors.

Declaring a structure

To declare a structure type, we use the *struct* keyword followed by the name of the type, followed by the body of the structure, between opening and closing braces. The process is similar to declaring a *class*.

```
struct Time
```

```
{
public int hours, minutes, seconds;
}
```

As like classes, making the fields of a structure public is not advisable in most cases; there is no way to control the values held in *public* fields. A better idea is to make the fields *private* and provide wer structure with constructors and methods to initialize and manipulate these fields, as in this example:

```
struct Time
{
private int hours, minutes, seconds;
...
public Time(int hh, int mm, int ss)
{
this.hours = hh % 24;
this.minutes = mm % 60;
this.seconds = ss % 60;
}
public int Hours()
{
return this.hours;
}
}
```

- ❖ When we copy a *value* type variable, we get two copies of the value. In contrast, when we copy a reference type variable, we get two references to the same object.

NOTE: use structures for small data values for which it's efficient to copy the value as it would be to copy an address. Use classes for more complex data that is too big to copy efficiently.

<u>Understanding structure and class differences</u>

A structure and a class are syntactically similar, but there are a few important differences.

- ❖ We can't declare a default constructor (a constructor with no parameters) for a structure. For example:

```
struct Time
{
public Time() { ... } // compile-time error
...
}
```

- ❖ The reason we can't declare our own default constructor for a structure is that the compiler always generates one.
- ❖ In a class, the compiler generates the default constructor only if we don't write a constructor our self. The compiler-generated default constructor for a structure always sets the fields to 0, false, or null—just as for a class.
- ❖ Therefore, we should ensure that a structure value created by the default constructor behaves logically and makes sense with these default values.

- We can initialize fields to different values by providing a nondefault constructor. Our nondefault constructor must explicitly initialize all fields in our structure; the default initialization no longer occurs.

```
struct Time
{
private int hours, minutes, seconds;
...
public Time(int hh, int mm)
{
this.hours = hh;
this.minutes = mm;
} // compile-time error: seconds not initialized
}
```

In a class, we can initialize instance fields at their point of declaration. But in a structure, we cannot. The following example would compile if Time was a class, but because Time is a structure, it causes a compile-time error:

```
struct Time
{
private int hours = 0; // compile-time error
private int minutes;
private int seconds;
...
}
```

The following table summarizes the main **differences between a structure and a class.**

Class	**Structure**	**Question**
A class is a reference type.	A structure is a value type.	Is this a value type or a reference type?
Class instances are called objects and live on the heap.	Structure instances are called values and live on the stack.	Do instances live on the stack or the heap?
Yes.	No.	Can we declare a default constructor?
No.	Yes.	If we declare wer own constructor, will the compiler still generate the default constructor?
Yes.	No.	If we don't initialize a field in wer own constructor, will the compiler automatically initialize it for we?
Yes.	No.	Are we allowed to initialize instance fields at their point of declaration?

Declaring structure variables

After defining a structure type, we can use it in the same way as any other type. For example, if we have defined the Time structure, we can create variables, fields, and parameters of type Time, as in this example:

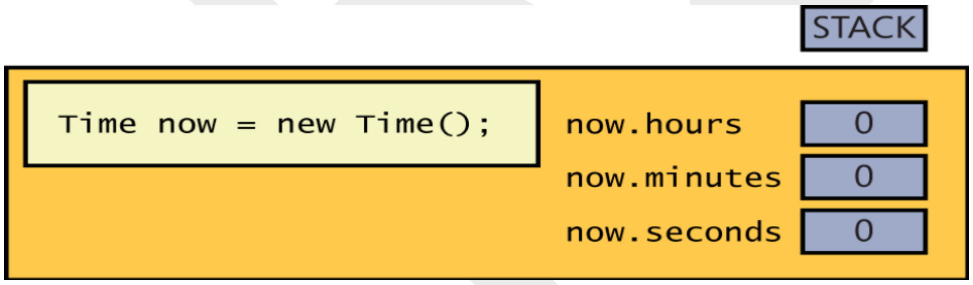
```
struct Time
{
private int hours, minutes, seconds;
...
}
class Example
{
private Time currentTime;
public void Method(Time parameter)
{
Time localVariable;
...
}
}
```

Understanding structure initialization

If we call a constructor, all the fields in the structure will be initialized:

Time now = new Time();

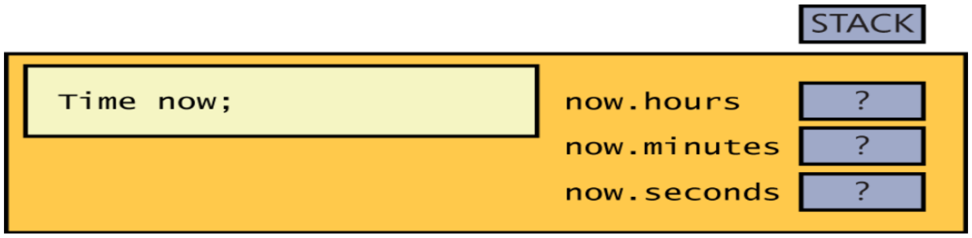
The following graphic depicts the state of the fields in this structure:



NOTE: Because structures are *value* types, we can also create structure variables without calling a constructor, like this:

Time now;

Here, the variable is created but its fields are left in their uninitialized state. The following graphic depicts the state of the fields in the now variable. Any attempt to access the values in these fields will result in a compiler error:



Note that in both cases, the Time variable is created on the *stack*.

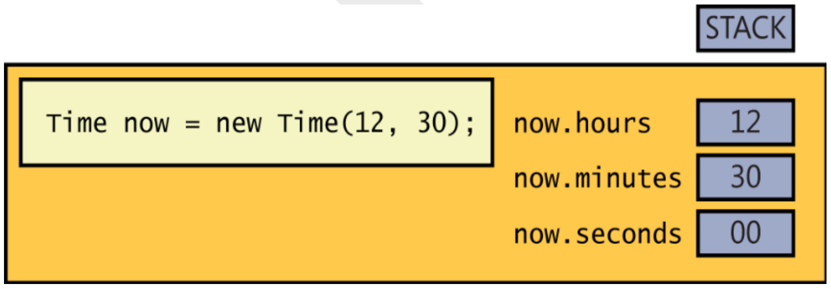
- ❖ If we’ve written our own structure constructor, we can also use that to initialize a structure variable. A structure constructor must always explicitly initialize all its fields. For example:

```
struct Time
{
private int hours, minutes, seconds;
...
public Time(int hh, int mm)
{
hours = hh;
minutes = mm;
seconds = 0;
}
}
```

This example initializes now by calling a user-defined constructor:

```
Time now = new Time(12, 30);
```

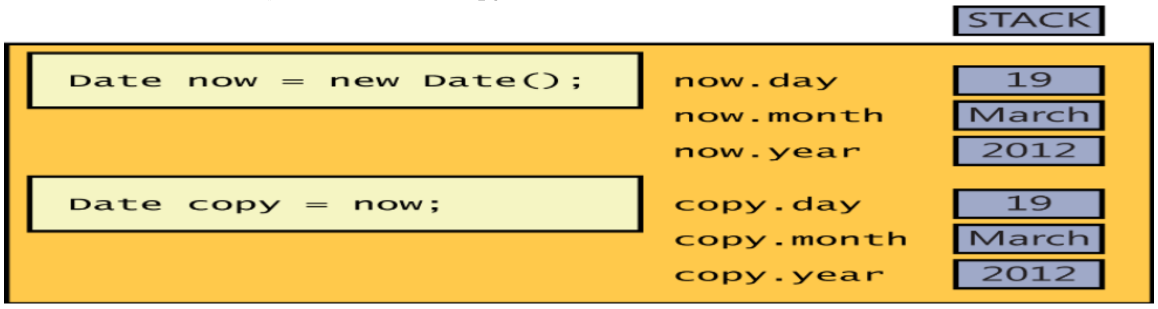
The following graphic shows the effect of this example:



Copying structure variables

We’re allowed to initialize or assign one structure variable to another structure variable, but only if the structure variable on the right side is completely initialized. The following example compiles because now is fully initialized.

Date now = new Date(); Date copy = now;



The following example fails to compile because now is not initialized:

```
Date now;
Date copy = now; // compile-time error: now has not been assigned
```

When we copy a structure variable, each field on the left side is set directly from the corresponding field on the right side.

MODULE-2 [CHAPTER-4]

USING ARRAYS

Declaring and creating an array

- ❖ An array is an unordered sequence of items.
- ❖ All the items in an array have the same type, unlike the fields in a structure or class, which can have different types.
- ❖ The items in an array live in a contiguous block of memory and are accessed by using an index, unlike fields in a structure or class, which are accessed by name.

Declaring array variables

We declare an array variable by specifying the name of the element type, followed by a pair of square brackets, followed by the variable name.

- ❖ The square brackets signify that the variable is an array.
  For example, to declare an array of int variables named pins (for holding a set of personal identification numbers) we can write the following:

```
int[] pins; // Personal Identification Numbers
```

- ❖ We are not restricted to primitive types as array elements.
- ❖ We can also create arrays of structures, enumerations, and classes. For example, we can create an array of Date structures like this:

```
Date[] dates;
```

Creating an array instance

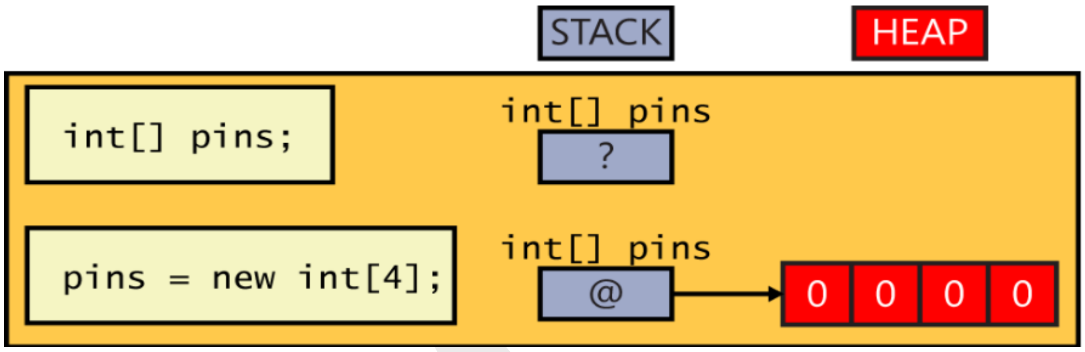
- ❖ Arrays are *reference* types, regardless of the type of their elements.
- ❖ This means that an array variable refers to a contiguous block of memory holding the array elements on the heap, just as a class variable refers to an object on the heap.
- ❖ This rule applies regardless of the type of the data items in the array. Even if the array contains a value type such as *int*, the memory will still be allocated on the *heap*; this is the one case where value types are not allocated memory on the *stack*.

Arrays follow the pattern: when we declare an array variable, we do not declare its size and no memory is allocated.

- ❖ The array is allocated memory only when the instance is created, and this is also the point at which we specify the size of the array.
- ❖ To create an array instance, we use the *new* keyword followed by the element type, followed by the size of the array we're creating between square brackets.
- ❖ Creating an array also initializes its elements by using the default values (0, null, or false, depending on whether the type is numeric, a reference, or a Boolean, respectively). For example, to create and initialize a new array of four integers for the pins variable declared earlier, we write this:

pins = new int[4];

The following graphic illustrates what happens when we declare an array, and later when we create an instance of the array:



NOTE: Because the memory for the array instance is allocated dynamically, the size of the array does not have to be a constant; it can be calculated at run time, as shown in this example:

int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];

❖ We can also create an array whose size is 0. It's useful for situations in which the size of the array is determined dynamically and could even be 0. An array of size 0 is not a *null* array; it is an array containing zero elements.

Populating and using an array

When we create an array instance, all the elements of the array are initialized to a default value depending on their type.

For example,

- all numeric values default to 0,
- objects are initialized to null,
- DateTime values are set to the date and time "01/01/0001 00:00:00", and
- strings are initialized to null.

1. We can modify this behaviour and initialize the elements of an array to specific values if we prefer. We achieve this by providing a comma-separated list of values between a pair of braces. For example, to initialize pins to an array of four int variables whose values are 9, 3, 7, and 2, we write:
   int[] pins = new int[4]{ 9, 3, 7, 2 };
2. The number of values between the braces must exactly match the size of the array instance being created:
   int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile-time error

int[] pins = new int[4]{ 9, 3, 7 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7, 2 }; // OK

3. When we're initializing an array variable in this way, we can actually omit the new expression and the size of the array. In this case, the compiler calculates the size from the number of initializers and generates code to create the array, such as in the example:
   int[] pins = { 9, 3, 7, 2 };

4. If we create an array of structures or objects, we can initialize each structure in the array by calling the structure or class constructor, as in this example:
   Time[] schedule = { new Time(12,30), new Time(5,30) };

Creating an implicitly typed array

❖ The element type when we declare an array must match the type of elements that we attempt to store in the array.
  For example, if we declare pins to be an array of *int*, we cannot store a *double*, *string*, *struct*, or anything that is not an int in this array.
❖ If we specify a list of initializers when declaring an array, we can let the C# compiler infer the actual type of the elements in the array for us, like this:
  var names = new[]{"John", "Diana", "James", "Francesca"};

Here, the C# compiler determines that the names variable is an array of strings.

❖ If we use this syntax, we must ensure that all the initializers have the same type.
  var bad = new[]{"John", "Diana", 99, 100};//compile time error

❖ In some cases, the compiler will convert elements to a different type. In the following code, the numbers array is an array of double because the constants 3.5 and 99.999 are both double, and the C# compiler can convert the integer values 1 and 2 to double values:
  var numbers = new[]{1, 2, 3.5, 99.999};

Generally, it is best to avoid mixing types and hoping that the compiler will convert them for us.

Implicitly typed arrays are most useful when we are working with anonymous types. The following code creates an array of anonymous objects, each containing two fields specifying the name and age of the members of my family:

```
var names = new[] { new { Name = "John", Age = 47 },
new { Name = "Diana", Age = 46 },
new { Name = "James", Age = 20 },
new { Name = "Francesca", Age = 18 } };
```

The fields in the anonymous types must be the same for each element of the array.

Accessing an individual array element

❖ To access an individual array element, we must provide an index indicating which element we require.

- ❖ Array indexes are zero-based; thus, the initial element of an array lives at index 0 and not index 1.
- ❖ An index value of 1 accesses the second element.
  For example, we can read the contents of element 2 of the pins array into an int variable by using the following code:
  ```
  int myPin;
  myPin = pins[2];
  ```
- ❖ Similarly, we can change the contents of an array by assigning a value to an indexed element:
  ```
  myPin = 1645;
  pins[2] = myPin;
  ```
- ❖ All array element access is bounds-checked. If we specify an index that is less than 0 or greater than or equal to the length of the array, the compiler throws an IndexOutOfRangeException exception, as in example:
  ```
  try
  {
      int[] pins = { 9, 3, 7, 2 };
      Console.WriteLine(pins[4]); // error, the 4th and last element is at index 3
  }
  catch (IndexOutOfRangeException ex)
  {
      ...
  }
  ```

Iterating through an array

All arrays are actually instances of the System.Array class, this class defines a number of useful properties and methods. For example, we can query the *Length* property to discover how many elements an array contains and *iterate* through all the elements of an array by using a *for* statement. The following sample code writes the array element values of the pins array to the console:

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
int pin = pins[index];
Console.WriteLine(pin);
}
```

- ❖ It is common for new programmers to forget that arrays start at element 0 and that the last element is numbered Length – 1.
- ❖ C# provides the *foreach* statement with which we can iterate through the elements of an array without worrying about these issues.
  For example, here's an equivalent foreach statement:
  ```
  int[] pins = { 9, 3, 7, 2 };
  foreach (int pin in pins)
  {
  Console.WriteLine(pin);
  ```

```
}
```

- ❖ The foreach statement declares an iteration variable (in this example, int pin) that automatically acquires the value of each element in the array.
- ❖ The type of this variable must match the type of the elements in the array.
- ❖ The foreach statement is the preferred way to iterate through an array; it expresses the intention of the code directly, and all of the for loop scaffolding drops away.

- ❖ However, in a few cases, we'll find that we have to revert to a for statement:

  1.A foreach statement always iterates through the entire array.
  2.A foreach statement always iterates from index 0 through index Length – 1.
  3.If the body of the loop needs to know the index of the element rather than just the value of the element, we'll have to use a for statement.
  4.If we need to modify the elements of the array, we'll have to use a for statement.

```
var names = new[] { new { Name = "John", Age = 47 },
 new { Name = "Diana", Age = 46 },
 new { Name = "James", Age = 20 },
 new { Name = "Francesca", Age = 18 } };

foreach (var familyMember in names)
{
Console.WriteLine("Name: {0}, Age: {1}", familyMember.Name,
familyMember.Age);
}
```

<u>Passing arrays as parameters and return values for a method</u>

- ❖ We can define methods that take arrays as parameters or pass them back as return values.
- ❖ The syntax for passing an array as a parameter is much the same as declaring an array.
  For example, the code sample that follows defines a method called *ProcessData* that takes an array of integers as a parameter. The body of the method iterates through the array and performs some unspecified processing on each element:

```
public void ProcessData(int[] data)
{
foreach (int i in data)
{
...
}
}
```

- ❖ It is important to remember that arrays are *reference* objects, so if we modify the contents of an array passed as a parameter inside a method such as *ProcessData*, the modification is visible through all references to the array, including the original argument passed as the parameter.

To *return* an array from a method, we specify the type of the array as the return type. The array created by the method is passed back as the return value:

```
public int[] ReadData()
{
Console.WriteLine("How many elements?");
string reply = Console.ReadLine();
int numElements = int.Parse(reply);
int[] data = new int[numElements];
for (int i = 0; i < numElements; i++)
{
Console.WriteLine("Enter data for element {0}", i);
reply = Console.ReadLine();
int elementData = int.Parse(reply);
data[i] = elementData;
}
return data;
}
```

- ❖ We can call the ReadData method like this:

```
int[] data = ReadData();
```

<u>Copying arrays</u>

- ❖ Arrays are reference types.
- ❖ An array variable contains a reference to an array instance.
- ❖ This means that when we copy an array variable, we actually end up with two references to the same array instance, as in the following example:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias and pins refer to the same array instance
```

If we modify the value at pins[1], the change will also be visible by reading alias[1].

- ❖ If we want to make a copy of the array instance (the data on the heap) that an array variable refers to, we have to do two things.
- ❖ First, we create a new array instance of the same type and the same length as the array we are copying.
- ❖ Second, we copy the data element by element from the original array to the new array, as in this example:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
copy[i] = pins[i];
}
```

Note that this code uses the Length property of the original array to specify the size of the new array.

Copying an array is actually a common requirement of many applications—so much so that the *System.Array* class provides some useful methods that we can employ to copy an array rather than writing wer own code.

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

Another way to copy the values is to use the *System.Array* static method named Copy. As with CopyTo, we must initialize the target array before calling Copy:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

Another alternative is to use the *System.Array* instance method named *Clone*. We can call this method to create an entire array and copy it in one action:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```

Using multidimensional arrays

We can create arrays with more than one dimension is called multidimensional array.

For example, to create a two-dimensional array, We specify an array that requires two integer indexes. The first dimension specifying a number of *rows*, and the second specifying a number of *columns*.

```
int[,] items = new int[4, 6];
```

To access an element in the two-dimensional array, we provide two index values to specify the "*cell*" holding the element. (A cell is the intersection of a row and a column).

```
items[2, 3] = 99; // set the element at cell(2,3) to 99
items[2, 4] = items [2,3]; // copy the element in cell(2, 3) to cell(2, 4)
items[2, 4]++; // increment the integer value at cell(2, 4)
```

There is no limit on the number of dimensions that we can specify for an array. The next example creates and uses an array called cube that contains three dimensions. Notice that we must specify *three* indexes to access each element in the array.

```
int[, ,] cube = new int[5, 5, 5];
cube[1, 2, 1] = 101;
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

Caution: About creating arrays with more than three dimensions.

- ❖ Arrays can consume a lot of memory.
- ❖ The cube array contains 125 elements (5 * 5 * 5).
- ❖ A four-dimensional array for which each dimension has a size of 5 contains 625 elements.

- ❖ If we start to create arrays with three or more dimensions, we can soon run out of memory. Therefore, we should always be prepared to catch and handle OutOfMemoryException exceptions when we use multidimensional arrays.

Creating jagged arrays

In C#, multidimensional arrays are sometimes referred to as rectangular arrays. Each dimension has a regular shape.

For example, in the following tabular two-dimensional items array, every row has a column containing 40 elements, and there are 160 elements in total:

```
int[,] items = new int[4, 40];
```

As mentioned above, multidimensional arrays can consume a lot of memory.If the application uses only some of the data in each column, allocating memory for unused elements is a waste. In this scenario, we can use a *jagged* array, for which each column has a different length, like this:

```
int[][] items = new int[4][];
int[] columnForRow0 = new int[3];
int[] columnForRow1 = new int[10];
int[] columnForRow2 = new int[40];
int[] columnForRow3 = new int[25];
items[0] = columnForRow0;
items[1] = columnForRow1;
items[2] = columnForRow2;
items[3] = columnForRow3;
...
```

In this example, the application requires only 3 elements in the first column, 10 elements in the second column,40 elements in the third column, and 25 elements in the final column. This code illustrates an array of arrays—rather than items being a two-dimensional array, it has only a single dimension, but the elements in that dimension are themselves arrays. Furthermore, the total size of the items array is 78 elements rather than 160; no space is allocated for elements that the application is not going to use.

- ❖ The following declaration specifies that items is an array of arrays of int.
  int[][] items;

- ❖ The following statement initializes items to hold four elements, each of which is an array of indeterminate length:
  items = new int[4][];

- ❖ The arrays columnForRow0 to columnForRow3 are all single-dimensional *int* arrays, initialized to hold the required amount of data for each column.
  items[0] = columnForRow0;

- We can populate data in this column either by assigning a value to an indexed element in columnForRow0 or by referencing it through the items array. The following statements are equivalent:

```
columnForRow0[1] = 99;

items[0][1] = 99;
```

MODULE 3 [CHAPTER 1]

Understanding parameter arrays

Overloading

Overloading is the technical term for declaring two or more methods with the same name in the same scope. It is very useful for cases in which we want to perform the same action on arguments of different types. The classic example of overloading is the Console.WriteLine method. This method is overloaded numerous times so that we can pass any primitive type argument. For example:

```
class Console

{       public static void WriteLine(Int value)
        public static void WriteLine(Double value)
        public static void WriteLine(Decimal value)
        public static void WriteLine(Boolean value)
        public static void WriteLine(String value)
        ...
}
```

Overloading doesn’t easily handle a situation in which the type of parameters doesn’t vary but the number of parameters does. Fortunately, there is a way to write a method that takes a variable number of arguments (a variadic method): we can use a parameter array (a parameter declared by using the params keyword).

Let’s first understand the uses and shortcomings of ordinary arrays.

Using array arguments

Suppose that we want to write a method to determine the minimum value in a set of values passed as parameters. One way is to use an array. For example, to find the smallest of several int values, we could write a static method named Min with a single parameter representing an array of int values:

```
class Util

  {     public static int Min(int[] paramList)

        {       // Verify that the caller has provided at least one parameter.
                // If not, throw an ArgumentException exception – it is not possible
                // to find the smallest value in an empty list.
                if (paramList == null || paramList.Length == 0)
                { throw new ArgumentException("Util.Min: not enough arguments");
                }

        // Set the current minimum value found in the list of parameters to the first item
                int currentMin = paramList[0];
                // Iterate through the list of parameters, searching to see whether any of them
                // are smaller than the value held in currentMin
                    foreach (int i in paramList)
                    {   // If the loop finds an item that is smaller than the value held in
                        // currentMin, then set currentMin to this value
                        if (i < currentMin)
                        {   currentMin = i;
                        }
                    }
                    // At the end of the loop, currentMin holds the value of the smallest
                    // item in the list of parameters, so return this value.
                    return

currentMin;

        }

}
```

To use the Min method to find the minimum of two int variables named first and second, we can write this:

```
int[] array = new int[2];
array[0] = first;
array[1] = second;

 int min = Util.Min(array);
```

We can see that this avoids the need for a large number of overloads. We can also use an anonymous array if we prefer, like this:

```
 int min = Util.Min(new int[] {first, second});
```

However, the point is that we still need to create and populate an array, and the syntax can get a little confusing. The solution is to get the compiler to write some of this code by using a params array as the parameter to the Min method.

Declaring a params Array

Using a params array, we can pass a variable number of arguments to a method. We indicate a params array by using the *params* keyword as an array parameter modifier when you define the method parameters. For example,

```
  class Util
{         public static int Min(params int[] paramList)
          {
                    // code exactly as before
          }
}
```

The effect of the *params* keyword on the Min method is that it makes it possible for us to call it by using any number of integer arguments without worrying about creating an array. For example, to find the minimum of two integer values, we can simply write this:

```
   int min = Util.Min(first, second);
```

The compiler translates this call into code similar to this:

```
int[] array = new int[2];

array[0] = first;

   array[1] = second;
int min = Util.Min(array);
```

To find the minimum of three integer values, which is also converted by the compiler to the corresponding code that uses an array:

```
    int min = Util.Min(first, second, third);
```

Both calls to Min (one call with two arguments and another with three arguments).

Now to the same Min method with the *params* keyword, we can call the method with any number of int arguments. The compiler just counts the number of int arguments, creates an int array of that size, fills the array with the arguments, and then calls the method by passing the single array parameter.

There are several points about params arrays:

- We can't use the *params* keyword with multidimensional arrays. The code in the following example will not compile:
- // compile-time error
  ```
  public static int Min(params int[,] table)
  ...
  ```
- We can't overload a method based on the params keyword. The params keyword does not form part of a method's signature, as shown in this example:

- // compile-time error: duplicate declaration
  public static int Min(int[] paramList)
  ...
  public static int Min(params int[] paramList)
  ...
- We're not allowed to specify the ref or out modifier with params arrays, as shown in this example:
  // compile-time errors
  public static int Min(ref params int[] paramList)
  ...
  public static int Min(out params int[] paramList)
  ...
- A params array must be the last parameter. (This means that you can have only one params array per method.) Consider this example:
- // compile-time error
  public static int Min(params int[] paramList, int i)
  ...
- A non-params method always takes priority over a params method. This means that if we want to, we can still create an overloaded version of a method for the common cases, For example: public static int Min(int leftHandSide, int rightHandSide)
  …
  public static int Min(params int[] paramList)
  ...

The first version of the Min method is used when called using two int arguments. The second version is used if any other number of int arguments is supplied.

Using params object[ ]

A parameter array of type int is very useful. With it, we can pass any number of int arguments in a method call. Now suppose if not only the number of arguments varies but also the argument type varies means then we have a way to solve this problem. Then we can use a parameters array of type *object* to declare a method that accepts any number of object arguments, allowing the arguments passed in to be of any type. For example:

```
  class Black
{         public static void Hole(params object [] paramList)


...

}
```

The method is called *Black.Hole*, because no argument can escape from it.

- We can pass the method no arguments at all, in which case the compiler will pass an object array whose length is 0:
- Black.Hole();
- // converted to Black.Hole(new object[0]);
- We can call the Black.Hole method by passing null as the argument. As array is a reference type, so we're allowed to initialize an array with null:
- Black.Hole(null);
- We can pass the Black.Hole method an actual array.
  ```
  object[] array = new object[2];
  array[0] = "forty two";
  array[1] = 42;
  Black.Hole(array);
  ```
- We can pass the Black.Hole method arguments of different types, and these arguments will automatically be wrapped inside an object array:
  ```
  Black.Hole("forty two", 42);
  //converted to Black.Hole(new object[]{"forty two", 42});
  ```

Comparing parameter arrays and optional parameters

There are some fundamental differences between them:

- A method that takes *optional parameters* still has a fixed parameter list, and we cannot pass an arbitrary list of arguments. The compiler generates code that inserts the default values onto the stack for any missing arguments before the method runs, and the method is not aware of which of the arguments are caller provided and which are compiler-generated defaults.
- A method that uses a *parameter array* effectively has a completely arbitrary list of parameters, and none of them has a default value. Furthermore, the method can determine exactly how many arguments the caller provided.

Generally, we use *parameter arrays* for methods that can take any number of parameters (including none), whereas we use *optional parameters* only where it is not convenient to force a caller to provide an argument for every parameter.

MODULE 3[CHAPTER 2]

Working with inheritance

What is inheritance?

Inheritance in programming is all about classification—it's a relationship between classes. For example, when we were at school, we probably learned about mammals, and we learned that horses and whales are examples of mammals. Each has every attribute that a mammal does (it breathes air, it suckles its young, it is warm-blooded and so on), but each also has its own special features (a horse has hooves, but a whale has flippers and a fluke).

In our program, we can create two distinct classes named Horse and Whale. Each class can implement the behaviours that are unique to that type of mammal, such as Trot (for a horse) or Swim (for a whale), in its own way. But to handle common behaviours of both the classes we can use class inheritance to address it, so we can create a class named Mammal that provides the common functionality exhibited by these types. You can then declare that the Horse, Whale, Human classes all inherited from Mammal class. These classes then automatically include the functionality of the Mammal class (Breathe, SuckleYoung and so on), but we can also add each class with the functionality unique to a particular type.

Using inheritance

We declare that a class inherits from another class by using the following syntax:

```
class DerivedClass : BaseClass

{
        ...

  }
```

The derived class inherits from the base class, and the methods in the base class become part of to derive from two or more classes. However, unless DerivedClass is declared as *sealed*, we can create further derived classes that inherit from DerivedClass using the same syntax.

```
 class DerivedSubClass : DerivedClass
{
        ...
}
```

We can declare the Mammal class as in the example below. The methods Breathe and SuckleYoung are common to all mammals.

```
  class Mammal
{       public void Breathe()
```

```
        {
                ...
        }
        public void SuckleYoung()
        {
                ...


}
        ...

}
```

We can then define classes for each different type of mammal, adding more methods as necessary, such as in the following example:

```
class Horse : Mammal
{
        ...
        public void Trot()
        {
                ...
        }
}
class Whale : Mammal
{
        ...
        public void Swim()
        {
                ...
        }

}
```

If we create a Horse object in our application, we can call the Trot, Breathe, and SuckleYoung methods:

```
Horse myHorse = new Horse();
myHorse.Trot();
myHorse.Breathe();
myHorse.SuckeYoung();
```

Similarly, we can create a Whale object, but this time we can call the Swim, Breathe, and SuckleYoung methods; Trot is not available because it is only defined in the Horse class.

The *System.Object* class revisited

The *System.Object* class is the root class of all classes. All classes implicitly derive from *System.Object*.

```
class Mammal : System.Object
{
        ...
}
```

Any methods in the *System.Object* class are automatically passed down the chain of inheritance to classes that derive from Mammal, such as Horse and Whale. This means that all classes that you define automatically inherit all the features of the System.Object class. This includes methods such as ToString, which is used to convert an object to a string, typically for display purposes.

Calling base class constructors

In addition to the methods that it inherits, a derived class automatically contains all the fields from the base class. These fields usually require initialization when an object is created. We typically perform this kind of initialization in a constructor. Remember that all classes have at least one constructor. (If we don't provide one, the compiler generates a default constructor for us).

It is good practice for a constructor in a derived class to call the constructor for its base class as part of the initialization, which enables the base-class constructor to perform any additional initialization that it requires. We can specify the *base* keyword to call a base-class constructor when we define a constructor for an inheriting class, as in this example:

```
 class Mammal // base class
{
        public Mammal(string name) // constructor for base class
        {
                ...
        }
        ...
}
class Horse : Mammal // derived class
{
        public Horse(string name)
        : base(name) // calls Mammal(name)
        {
                ...
        }
```

```
    ...
}
```

If we don't explicitly call a base class constructor in a derived-class constructor, the compiler attempts to silently insert a call to the base class's default constructor before executing the code in the derived-class constructor. The compiler rewrites it:

```
class Horse : Mammal
{
    public Horse(string name)
    {
        ...
    }
    ...
}


class Horse : Mammal
{
    public Horse(string name)
    : base()
    {
        ...
    }
    ...
}
```

This works only if Mammal has a public default constructor.

- **Assigning classes**

  There are examples of how the type-checking rules prevent us from assigning an object of one type to a variable declared as a different type. For example, given the definitions of the Mammal, Horse, and Whale classes, the code that follows these definitions is illegal:

```
class Mammal
{
    ...
}
class Horse : Mammal
{
_   ...
}
class Whale : Mammal
{
    ...
}
...
Horse myHorse = new Horse(...);
Whale myWhale = myHorse; // error - different types
```

- ❖ It is possible to refer to an object from a variable of a different type as long as the type used is a class that is higher up the inheritance hierarchy. So the following statements are legal:

  ```
  Horse myHorse = new Horse(...);
  Mammal myMammal = myHorse; // legal, Mammal is the base class of Horse
  ```

- ❖ There is one significant limitation, when referring to a Horse or Whale object by using a Mammal variable; you can access only methods and fields that are defined by the Mammal class. Any additional methods defined by the Horse or Whale class are not visible through the Mammal class.

  ```
  Horse myHorse = new Horse(...);
  Mammal myMammal = myHorse;
  myMammal.Breathe(); // OK - Breathe is part of the Mammal class
  myMammal.Trot(); // error - Trot is not part of the Mammal class
  ```

- ❖ Be warned that the converse situation is not true. We cannot assign a Mammal object to a Horse variable: Mammal myMammal = newMammal(...);

  ```
  Horse myHorse = myMammal; // error
  ```

These looks strange, but remember that not all Mammal objects are Horses—some might be Whales. We can assign a Mammal object to a Horse variable as long as we need to check that the Mammal is really a Horse first, by using the as or is operator, or by using a cast.

The example that follows uses the as operator to check that myMammal refers to a Horse.

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse; // myMammal refers to a Horse
...

Horse myHorseAgain = myMammal as Horse; // OK - myMammal was a Horse ...
Whale myWhale = new Whale(...);
myMammal = myWhale;
...
myHorseAgain = myMammal as Horse; // returns null - myMammal was a Whale
```

Declaring new methods

If a base class and a derived class happen to declare two methods that have the same signature, you will receive a warning when you compile the application.

Note: The method signature refers to the name of the method and the number and types of its parameters, but not its return type. Two methods that have the same name and that take the same list of parameters have the same signature, even if they return different types.

A method in a derived class masks (or hides) a method in a base class that has the same signature. For example, the compiler generates a warning message in this code informing us that *Horse*.*Talk* hides the inherited method *Mammal*.*Talk*:

```
class Mammal
{
        ...
        public void Talk() // assume that all mammals can talk
        {
                ...
        }
}
class Horse : Mammal
{
        ...
        public void Talk() // horses talk in a different way from other mammals!
        {
                ...
        }
}
```

However, the Talk method in the Horse class hides the Talk method in the Mammal class, and the *Horse.Talk* method will be called, instead. Most of the time, such a coincidence is at best a source of confusion, and we should consider renaming methods to avoid clashes. However, if we're sure that we want the two methods to have the same signature, thus hiding the Mammal.Talk method, we can silence the warning by using the *new* keyword, as follows:

```
  class Mammal
{
        ...
        public void Talk()
        {
                ...
        }

 }
class Horse : Mammal
{
        ...
        new public void Talk()
        {
                ...
        }

 }
```

Using the *new* keyword like this does not change the fact that the two methods are completely unrelated and that hiding still occurs. It just turns the warning off. In effect, the new keyword says, **"I know what I'm doing, so stop showing me these warnings."**

Declaring virtual methods

- Sometimes, we do want to hide the way in which a method is implemented in a base class.
- As an example, consider the ToString method in System.Object. The purpose of ToString is to convert an object to its string representation.
- Because this method is very useful, it is a member of the System.Object class, thereby automatically providing all classes with a ToString method.
- A derived class might contain any number of fields with interesting values that should be part of the string.
- All it can do is convert an object to a string that contains the name of its type, such as "Mammal" or "Horse".
- Obviously, ToString is a fine idea in concept, and all classes should provide a method that can be used to convert objects to strings for display or debugging purposes.

A method that is intended to be overridden is called a *virtual* method.

- We should be clear on the difference between overriding a method and hiding a method.
- *Overriding* a method is a mechanism for providing different implementations of the same method—the methods are all related because they are intended to perform the same task, but in a class-specific manner.
- *Hiding* a method is a means of replacing one method with another—the methods are usually unrelated and might perform totally different tasks.
- Overriding a method is a useful programming concept;
- Hiding a method is often an error.

We can mark a method as a virtual method by using the *virtual* keyword. For example, the ToString method in the *System.Object* class is defined like this:

```
namespace System
```

```
{
        class Object
        {
                public virtual string ToString()
                {
                        ...
                }
                ...
        }
        ...
}
```

Declaring override methods

If a base class declares that a method is virtual, a derived class can use the *override* keyword to declare another implementation of that method, as demonstrated here:

```
class Horse : Mammal
{
        ...
        public override string ToString()
        {
                ...
        }
}
```

The new implementation of the method in the derived class can call the original implementation of the method in the base class by using the *base* keyword, like this:

```
public override string ToString()
{
         base.ToString();
        ...
}
```

There are some important rules we must follow when declaring polymorphic methods (as in the sidebar "Virtual methods and polymorphism") by using the virtual and override keywords:

- ❖ A virtual method cannot be private; it is intended to be exposed to other classes through inheritance. Similarly, override methods cannot be private because a class cannot change the protection level of a method that it inherits. However, override methods can have a special form of privacy known as protected access.

- ❖ The signatures of the virtual and override methods must be identical; they must have the same name, number, and types of parameters. In addition, both methods must return the same type.
- ❖ You can only override a virtual method. If the base class method is not virtual and we try to override it, we'll get a compile-time error. This is sensible; it should be up to the designer of the base class to decide whether its methods can be overridden.
- ❖ If the derived class does not declare the method by using the override keyword, it does not override the base class method; it hides the method. In other words, it becomes an implementation of a completely different method that happens to have the same name. As before, this will cause a compile-time hiding warning, which we can silence by using the new keyword.
- ❖ An override method is implicitly virtual and can itself be overridden in a further derived class. However, we are not allowed to explicitly declare that an override method is virtual by using the virtual keyword.

Understanding protected access

- ❖ The public and private access keywords create two extremes of accessibility.
- ❖ These two extremes are sufficient when considering classes in isolation. However, the object-oriented programmers know, isolated classes cannot solve complex problems.
- ❖ Inheritance is a powerful way of connecting classes, and there is clearly close relationship between a derived class and its base class.
- ❖ It is useful for a base class to allow derived classes to access some of its members while hiding these same members from classes that are not part of the inheritance hierarchy. In this situation, we can mark members with the *protected* keyword. It works like this:
- ✓ If a class A is derived from another class B, it can access the protected class members of class B.
- ✓ If a class A is not derived from another class B, it cannot access any protected members of class B.
- ● Public fields violate encapsulation because all users of the class have direct, unrestricted access to the fields.
- ● However, protected fields still allow encapsulation to be violated by other classes that inherit from the base class.

Understanding extension methods

- ❖ Sometimes using inheritance is not the most appropriate mechanism for adding new behaviours, especially if we need to quickly extend a type without affecting existing code. For example, suppose we want to add a new feature to the int type, such as a method named Negate that returns the negative equivalent value that an integer currently contains.

```
class NegInt32 : System.Int32 // don't try this!
{       public int Negate()
        {
                ...
        }
}
```

NegInt32 will inherit all the functionality associated with the System.Int32 type in addition to the Negate method. There are two reasons why we might not want to follow this approach:

- This method applies only to the NegInt32 type, and if we want to use it with existing int variables in our code, we have to change the definition of every int variable to the NegInt32 type.
- The System.Int32 type is actually a structure, not a class, and we cannot use inheritance with structures.

This is where extension methods become very useful.

Using an extension method, we can extend an existing type (a class or a structure) with additional static methods.

We define an extension method in a static class and specify the type to which the method applies as the first parameter to the method, along with the this keyword. For example:

```
 static class Util
{
        public static int Negate(this int i)
        {
                return -i;
        }
}
```

The syntax looks a little odd, but it is the this keyword prefixing the parameter to Negate that identifies it as an extension method, and the fact that the parameter that this prefixes is an int means that we are extending the int type.

We can simply use dot notation (.) to reference the method, like this:

```
int x = 591;
Console.WriteLine("x.Negate {0}", x.Negate());
```

- Notice that we do not need to reference the Util class anywhere in the statement that calls the Negate method.
- The C# compiler automatically detects all extension methods for a given type from all the static classes that are in scope.
- We can also invoke the Util.Negate method passing an int as the parameter, using the regular syntax that we have seen before, although this use obviates the purpose of defining the method as an extension method:

```
int x = 591;
Console.WriteLine("x.Negate {0}", Util.Negate(x));
```

MODULE 3 [CHAPTER 3]

Creating interfaces and defining abstract classes

Understanding interfaces

- If we want to define a new class in which we can store collections of objects, like an array.
- The collection should enable the application to retrieve objects in numerical order.
- When we define the collection class, we do not want to restrict the types of objects that it can hold (the objects can even be class or structure types), and consequently we don't know how to order these objects.
- The question therefore is how do we provide a method in the collection class that sorts objects whose types we do not know when we actually write the collection class?
- There is no inheritance relationship between the collection class and the objects that it holds, so a virtual method would not be of much use.
- The solution, therefore, is to require that all the objects provide a method, such as the CompareTo method shown in the following example that the RetrieveInOrder method of the collection can call, making it possible for the collection to compare these objects with one another:

-
  ```
  int CompareTo(object obj)
  {
          // return 0 if this instance is equal to obj
          // return < 0 if this instance is less than obj
          // return > 0 if this instance is greater than obj
          ...
  }
  ```

We can define an interface for collectable objects that includes the CompareTo method.

- An interface is similar to a contract.
- If a class implements an interface, the interface guarantees that the class contains all the methods specified in the interface.
- This mechanism ensures that we are able to call the CompareTo method on all objects in the collection and sort them.

❖ Using interfaces, we can truly separate the "what" from the "how."
❖ The interface gives us only the name, return type, and parameters of the method.
❖ Exactly how the method is implemented is not a concern of the interface.
❖ The interface describes the functionality that a class should provide but not how this functionality is implemented.

Defining an interface

- Defining an interface is syntactically similar to defining a class, except that we use the *interface* keyword instead of the *class* keyword.
- Within the interface, we declare methods exactly as in a class or a structure, except that we never specify an access modifier (public, private, or protected).
- Additionally, the methods in an interface have no implementation; they are simply declarations, and all types that implement the interface must provide their own implementations.
- Consequently, we replace the method body with a semicolon (only prototypes). For example:

```
interface IComparable
{
        int CompareTo(object obj);
}
```

An interface cannot contain any data; we cannot add fields (not even private ones) to an interface.

Implementing an interface

- To implement an interface, we declare a class or structure that inherits from the interface and that implements all the methods specified by the interface.
- This is not really inheritance as such; although the syntax is the same and some of the semantics.
- We should note that unlike class inheritance, a struct can implement an interface.

For example, suppose that we are defining the Mammal hierarchy as land-bound mammals and provide a method named NumberOfLegs that returns as an int the number of legs that a mammal has. We can define the ILandBound interface that contains this method, as follows:

```
 interface ILandBound
{
        int NumberOfLegs();
}
```

We can then implement this interface in the Horse class and provide an implementation of every method defined by the interface.

```
 class Horse : ILandBound
{
        ...
        public int NumberOfLegs()
        {
                return 4;
        }
}
```

When we implement an interface, we must ensure that each method matches its corresponding interface method exactly, according to the following rules:

- The method names and return types match exactly.
- Any parameters (including ref and out keyword modifiers) match exactly.
- All methods implementing an interface must be publicly accessible. However, if we are using an explicit interface implementation, the method should not have an access qualifier.

NOTE: If there is any difference between the interface definition and its declared implementation, the class will not compile.

A class can inherit from another class and implement an interface at the same time. In this case, C# uses a positional notation. The base class is always named first, followed by a comma, followed by the interface. The following example defines Horse as a class that is a Mammal but that additionally implements the ILandBound interface:

```
  interface ILandBound
{
        ...
}
class Mammal
{
        ...
}
class Horse : Mammal , ILandBound
{
        ...
}
```

<u>Referencing a class through its interface</u>

We can reference an object by using a variable defined as a class that is higher up the hierarchy, we can reference an object by using a variable defined as an interface that its class implements. For example: we can reference a Horse object by using an ILandBound variable, as follows:

```
  Horse myHorse = new Horse(...);
ILandBound iMyHorse = myHorse; // legal
```

- This works because all horses are land-bound mammals, although the converse is not true

- We cannot assign an ILandBound object to a Horse variable without casting it first.

This technique is useful because we can use it to define methods that can take different types as parameters, as long as the types implement a specified interface. For example,

```
  int FindLandSpeed(ILandBound landBoundMammal)
{
        ...
}
```

We use the *is* operator to determine whether an object has a specified type, and it works with interfaces as well as classes and structs. For example,

```
  if (myHorse is ILandBound)
{
        ILandBound iLandBoundAnimal = myHorse;
}
```

NOTE: When referencing an object through an interface, we can invoke only methods that are visible through the interface.

Working with multiple interfaces

- A class can have at most one base class.
- But it is allowed to implement an unlimited number of interfaces. A class must implement all the methods declared by these interfaces.

If a structure or class implements more than one interface, we specify the interfaces as a comma-separated list. If a class also has a base class, the interfaces are listed after the base class, like this:

```
 class Horse : Mammal, ILandBound, IGrazable
{
        ...
}
```

Explicitly implementing an interface

The examples so far have shown classes that implicitly implement an interface. If we revisit the ILandBound interface and the Horse class, although the Horse class implements from the ILandBound interface, there is nothing in the implementation of the NumberOfLegs method in the Horse class that says it is part of the ILandBound interface:

```
   interface ILandBound
{
    int NumberOfLegs();
}
class Horse : ILandBound
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

There is nothing to prevent multiple interfaces from specifying a method with the same name, although they might have different semantics. For example,

```
interface IJourney
{


int NumberOfLegs();

 }
```

Now, if we implement this interface in the Horse class, we have an interesting problem:

```
class Horse : ILandBound, IJourney
{
    ...

    public int NumberOfLegs()
    {
        return 4;


}

}
```

This is legal code. By default, C# does not distinguish which interface the method is implementing, so the same method actually implements both interfaces.

To solve this problem and disambiguate which method is part of which interface implementation, we can implement interfaces explicitly. To do this, we specify which interface a method belongs to when we implement it, like this:

```
class Horse : ILandBound, IJourney
{
    ...
    int ILandBound.NumberOfLegs()
    {
        return 4;
    }
    int IJourney.NumberOfLegs()
    {
        return 3;
    }
}
```

Now, we can see that the horse has four legs and has pulled the coach for three legs of the journey.

- ❖ Apart from prefixing the name of the method with the interface name, there is one other subtle difference in this syntax: The methods are not marked as public. We cannot specify the protection for methods that are part of an explicit interface implementation.
- ❖ This leads to another interesting phenomenon. If we create a Horse variable in code, we cannot actually invoke either of the NumberOfLegs methods, because they are not visible. As far as the Horse class is concerned, they are both private.

```
Horse horse = new Horse();
...
int legs = horse.NumberOfLegs();
```

To access the methods we reference the Horse object through the appropriate interface, like this:

```
Horse horse = new Horse();
...
IJourney journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs();
ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs();
```

Interface restrictions

The essential idea to remember is that an interface never contains any implementation. The following restrictions are natural consequences of this:

- ❖ We're not allowed to define any fields in an interface, not even static fields.
- ❖ We're not allowed to define any constructors in an interface.
- ❖ We're not allowed to define a destructor in an interface. A destructor contains the statements used to destroy an object instance.

- ❖ We cannot specify an access modifier for any method. All methods in an interface are implicitly public.
- ❖ We cannot nest any types (such as enumerations, structures, classes, or interfaces) inside an interface.
- ❖ An interface is not allowed to inherit from a structure or a class, although an interface can inherit from another interface.

Abstract classes

We can implement the ILandBound and IGrazable interfaces in many different classes, depending on how many different types of mammals we want to model in our application. In these situations, it’s common for parts of the derived classes to share common implementations. For example, the duplication in the following two classes is obvious:

```
class Horse : Mammal, ILandBound, IGrazable
{
        ...
        void IGrazable.ChewGrass()
        {
                Console.WriteLine("Chewing grass");
                // code for chewing grass


};

}
class Sheep : Mammal, ILandBound, IGrazable
{
        ...
        void IGrazable.ChewGrass()
        {
                Console.WriteLine("Chewing grass");
                // same code as horse for chewing grass
        };
}
```

- Duplication in code is a warning sign.
- If possible, we should refactor the code to avoid this duplication and reduce any associated maintenance costs.
- One way to achieve this refactoring is to put the common implementation into a new class created specifically for this purpose. We can insert a new class into the class hierarchy, For example:

```
class GrazingMammal : Mammal, IGrazable
{
        ...
        void IGrazable.ChewGrass()
        {
                // common code for chewing grass
                Console.WriteLine("Chewing grass");
        }
}

   class Horse : GrazingMammal, ILandBound
{
        ...
}
class Sheep : GrazingMammal, ILandBound
{
        ...
}
```

To declare that creating instances of a class is not allowed, we can declare that the class is abstract by using the *abstract* keyword, For example:

```
 abstract class GrazingMammal : Mammal, IGrazable
{
        ...
}
```

If we now try to instantiate a GrazingMammal object, the code will not compile:
GrazingMammal myGrazingMammal = new GrazingMammal(...); // illegal

Abstract methods

- An abstract class can contain abstract methods. An abstract method is similar in principle to a virtual method, except that it does not contain a method body.
- A derived class must override this method.
- The following example defines the DigestGrass method in the GrazingMammal class as an abstract method; grazing mammals might use the same code for chewing grass, but they must provide their own implementation of the DigestGrass method.
- An abstract method is useful if it does not make sense to provide a default implementation in the abstract class but you want to ensure that an inheriting class provides its own implementation of that method.

```
abstract class GrazingMammal : Mammal, IGrazable
{
        abstract void DigestGrass();
        ...
}
```

Sealed classes

We can use the *sealed* keyword to prevent a class from being used as a base class if we decide that it should not be. For example:

```
sealed class Horse : GrazingMammal, ILandBound
{
        ...
}
```

If any class attempts to use Horse as a base class, a compile-time error will be generated. Note that a sealed class cannot declare any virtual methods and that an abstract class cannot be sealed.

Sealed methods

- ❖ We can also use the *sealed* keyword to declare that an individual method in an unsealed class is sealed.
- ❖ This means that a derived class cannot override this method.
- ❖ We can seal only an override method, and we declare the method as sealed override.

We can think of the interface, virtual, override, and sealed keywords as follows:

- An interface introduces the name of a method.
- A virtual method is the first implementation of a method.
- An override method is another implementation of a method.
- A sealed method is the last implementation of a method.

MODULE 3[CHAPTER 4]

Using garbage collection and resource management

The life and times of an object

First, let's see what happens when we create an object. We create an object by using the *new* operator. The following example creates a new instance of the Square class.

```
Square mySquare = new Square(); // Square is a reference type
```

According to us, the new operation is a single operation, but underneath, object creation is really a two-phase process:

1. The new operation allocates a chunk of raw memory from the heap. We have no control over this phase of an object's creation.
2. The new operation converts the chunk of raw memory to an object; it has to initialize the object. We can control this phase by using a constructor.

After we have created an object, we can access its members by using the dot operator (.). For example, the Square class includes a method named Draw that we can call:
mySquare.Draw();

When the mySquare variable goes out of scope, the Square object is no longer being actively referenced, and the object can be destroyed and the memory that it is using can be reclaimed.

Like object creation, object destruction is a two-phase process. The two phases of destruction exactly mirror the two phases of creation:
1. The Common Language Runtime (CLR) must perform some tidying up. We can control this by writing a destructor.

2. The CLR must return the memory previously belonging to the object back to the heap; the memory that the object lived in must be deallocated. We have no control over this phase.

The process of destroying an object and returning memory back to the heap is known as garbage collection.

Writing destructors

We can use a destructor to perform any tidying up required when an object is garbage collected. The CLR will automatically clear up any managed resources that an object uses. A destructor is a special method, a little like a constructor, except that the CLR calls it after the reference to an object has disappeared. The syntax for writing a destructor is a tilde (~) followed by the name of the class. For example:

```
class FileProcessor
{
        FileStream file = null;
        public FileProcessor(string fileName)
        {
                this.file = File.OpenRead(fileName); // open file for reading
        }
        ~FileProcessor()
        {
                this.file.Close(); // close file
        }
}
```

There are some very important restrictions that apply to destructors:

- Destructors apply only to reference types; we cannot declare a destructor in a value type, such as a struct.

  ```
  struct MyStruct
  {
          ~ MyStruct() { ... } // compile-time error
  }
  ```
- We cannot specify an access modifier (such as public) for a destructor. We never call the destructor in our own code; part of the CLR called the garbage collector does this for us.
- public ~ FileProcessor() { ... } // compile-time error
- A destructor cannot take any parameters. Again, this is because we never call the destructor ourselves.

  ~ FileProcessor(int parameter) { ... } // compile-time error

Internally, the C# compiler automatically translates a destructor into an override of the Object.Finalize method. The compiler converts this destructor

```
class FileProcessor
{
        ~ FileProcessor() { // your code goes here }
}
```

into this:

```
class FileProcessor
{
        protected override void Finalize()
        {
                try { // your code goes here }
                finally { base.Finalize(); }
```

```
        }
}
```

It’s important to understand that only the compiler can make this translation. We can’t write our own method to override Finalize, and we can’t call Finalize ourselves.

Why use the garbage collector?

We can never destroy an object ourselves by using C# code. There just isn’t any syntax to do it. Instead, the CLR does it for us at a time of its own choosing. In addition, keep in mind that we can also make more than one reference variable refer to the same object.

```
 FileProcessor myFp = new FileProcessor();
FileProcessor referenceToMyFp = myFp;
```

An object can be destroyed and its memory made available for reuse only when all the references to it have disappeared.

If it was our responsibility to destroy objects, sooner or later one of the following situations would arise:

- We would forget to destroy the object.
- We would try to destroy an active object and risk the possibility of one or more variables holding a reference to a destroyed object, known as a dangling reference.
- We would try to destroy the same object more than once. This might or might not be disastrous, depending on the code in the destructor.

The garbage collector makes the following guarantees:

- Every object will be destroyed, and its destructor will be run. When a program ends, all outstanding objects will be destroyed.
- Every object will be destroyed exactly once.
- Every object will be destroyed only when it becomes unreachable—that is, when there are no references to the object in the process running our application.

When does garbage collection occur?

This garbage collection occurs when an object is no longer needed. Garbage collection can be an expensive process, so the CLR collects garbage only when it needs to and then it collects as much as it can. Performing a few large sweeps of memory is more efficient than performing lots of little dustings.

One feature of the garbage collector is that we don’t know, and should not rely upon, the order in which objects will be destroyed. The final point to understand is arguably the most important: Destructors do not run until objects are garbage collected.

How does the garbage collector work?

The garbage collector runs in its own thread and can execute only at certain times—typically, when our application reaches the end of a method. While it runs, other threads running in our application will temporarily halt. This is because the garbage collector might need to move objects around and update object references, and it cannot do this while objects are in use.

The steps that the garbage collector takes are as follows:

1. It builds a map of all reachable objects. It does this by repeatedly following reference fields inside objects. The garbage collector builds this map very carefully and ensures that circular references do not cause an infinite recursion. Any object not in this map is deemed to be unreachable.

2. It checks whether any of the unreachable objects has a destructor that needs to be run (a process called finalization). Any unreachable object that requires finalization is placed in a special queue called the freachable queue (pronounced "F-reachable").

3. It deallocates the remaining unreachable objects (those that don't require finalization) by moving the reachable objects down the heap, thus defragmenting the heap and freeing memory at its top. When the garbage collector moves a reachable object, it also updates any references to the object.

4. At this point, it allows other threads to resume.

5. It finalizes the unreachable objects that require finalization (now in the freachable queue) by running the Finalize methods on its own thread.

Recommendations

Writing classes that contain destructors adds complexity to our code and to the garbage collection process, and makes our program run more slowly. Therefore, try to avoid using destructors except when we really need them; only use them to reclaim unmanaged resources.

We need to be very careful when we write a destructor. In particular, be aware that, if our destructor calls other objects, those other objects might have already had their destructor called by the garbage collector. Remember that the order of finalization is not guaranteed. Therefore, ensure that destructors do not depend on one another or overlap one another—don't have two destructors that try to release the same resource.

Resource management

Sometimes, it's inadvisable to release a resource in a destructor; some resources are just too valuable to lie around waiting for an arbitrary length of time until the garbage collector actually releases them. Scarce resources such as memory, database connections, or file handles need to be released, and they need to be released as soon as possible. In these situations, our only option is to release the resource ourselves. We can achieve this by creating a disposal method.

- A disposal method is a method that explicitly disposes of a resource. If a class has a disposal method, you can call it and control when the resource is released.

Disposal methods

An example of a class that implements a disposal method is the TextReader class from the System.IO namespace. This class provides a mechanism to read characters from a sequential stream of input. Here's an example that reads lines of text from a file by using the StreamReader class and then displays them on the screen:

```
  TextReader reader = new StreamReader(filename);
string line;
while ((line = reader.ReadLine()) != null)

{
        Console.WriteLine(line);

}
reader.Close();
```

The ReadLine method reads the next line of text from the stream into a string. The ReadLine method returns null if there is nothing left in the stream. It's important to call Close when we have finished. However, there is a problem with this example: it's not exception-safe. If the call to ReadLine or WriteLine throws an exception, the call to Close will not happen; it will be bypassed. If this happens often enough, we will run out of file handles and be unable to open any more files.

Exception-safe disposal

One way to ensure that a disposal method (such as Close) is always called, regardless of whether there is an exception, is to call the disposal method within a finally block. For example:

```
  TextReader reader = new StreamReader(filename);
try
{
        string line;
        while ((line = reader.ReadLine()) != null)
        {
                Console.WriteLine(line);
}

  }
finally
{
        reader.Close();
}
```

Using a finally block like this works, but it has several drawbacks that make it a less-than-ideal solution:

- It quickly becomes unwieldy if we have to dispose of more than one resource.
- We might need to modify the code to make it fit this idiom. (For example, you might need to reorder the declaration of the resource reference).
- It fails to create an abstraction of the solution. This means that the solution is hard to understand and we must repeat the code everywhere we need this functionality.
- The reference to the resource remains in scope after the finally block. This means that we can accidentally try to use the resource after it has been released.

The *using* statement is designed to solve all these problems. The using statement provides a clean mechanism for controlling the lifetimes of resources. We can create an object, and this object will be destroyed when the using statement block finishes.
The syntax for a using statement is as follows:

```
using (type variable = initialization )
{
        StatementBlock
}
```

Here is the best way to ensure that your code always calls Close on a TextReader:

```
using (TextReader reader = new StreamReader(filename))
{
        string line;
        while ((line = reader.ReadLine()) != null)
        {
                Console.WriteLine(line);


}

 }
```

This using statement is precisely equivalent to the following transformation:

```
{
        TextReader reader = new StreamReader(filename);
        try
        {
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                        Console.WriteLine(line);
                }


}

        finally
        {
                if (reader != null)
                {
                        ((IDisposable)reader).Dispose();
                }

        }
}
```

The variable we declare in a using statement must be of a type that implements the IDisposable interface. The IDisposable interface lives in the System namespace and contains just one method, named Dispose:

```
namespace

System

 {
        interface

IDisposable

        {
                void

Dispose();
        }
}
```

The purpose of the Dispose method is to free any resources used by an object. We can employ a using statement as a clean, exception-safe, and robust way to ensure that a resource is always released. This approach solves all of the problems that existed in the manual try/finally solution. We now have a solution that

- Scales well if we need to dispose of multiple resources.
- Doesn't distort the logic of the program code.
- Abstracts away the problem and avoids repetition.
- Is robust. We can't accidentally reference the variable declared within the using statement, after the using statement has ended because it's not in scope anymore—we'll get a compile-time error.

Calling the Dispose method from a destructor

A call to a destructor will happen, but we just don't know when. On the other hand, we know exactly when a call to the Dispose method happens, but we just can't be sure that it will actually happen, because it relies on the programmer using our classes remembering to write a using statement. However, it is possible to ensure that the Dispose method always runs by calling it from the destructor. This acts as a useful backup. You might forget to call the Dispose method, but at least we can be sure that it will be called, even if it's only when the program shuts down.

```
class Example : IDisposable
{
        private Resource scarce; // scarce resource to manage and dispose
_       private bool disposed = false; // flag to indicate whether the resource
        // has already been disposed
        ...
        ~Example()
        {
                this.Dispose(false);
        }
        public virtual void Dispose()
        {
                this.Dispose(true);
                GC.SuppressFinalize(this);
        }
        protected virtual void Dispose(bool disposing)
        {
                if (!this.disposed)
        {
                if (disposing)
                {
                        // release large, managed resource here
                        ...
                }
                // release unmanaged resources here
                ...
                this.disposed = true;
        }
    }
    public void SomeBehavior() // example method
    {
        checkIfDisposed();
        ...
    }
    ...
    private void checkIfDisposed()
    {
        if (this.disposed)
        {
                throw new ObjectDisposedException("Example: object has been disposed
of");                                          }
    }
}
```

Notice the following features of the Example class:

- The class implements the IDisposable interface.
- The public Dispose method can be called at any time by our application code.
- The public Dispose method calls the protected and overloaded version of the Dispose method that takes a Boolean parameter, passing the value true as the argument. This method actually performs the resource disposal.
- The destructor calls the protected and overloaded version of the Dispose method that takes a Boolean parameter, passing the value false as the argument. The destructor is called only by the garbage collector, when your object is being finalized.
- We can call the protected Dispose method safely multiple times. The variable disposed indicates whether the method has already been run and is a safety feature to prevent the method from attempting to dispose the resources multiple times if it is called concurrently. The resources are released only the first time the method runs.

- The protected Dispose method supports disposal of managed resources (such as a large array) and unmanaged resources (such as a file handle). If the disposing parameter is true, this method must have been called from the public Dispose method. In this case, the managed resources and unmanaged resources are all released. If the disposing parameter is false, this method must have been called from the destructor, and the garbage collector is finalizing the object. In this case, it is not necessary (or exception-safe) to release the managed resources, because they will be, or might already have been, handled by the garbage collector, so only the unmanaged resources are released.
- The public Dispose method calls the static GC.SuppressFinalize method. This method stops the garbage collector from calling the destructor on this object, because the object has already been finalized.
- All the regular methods of the class (such as SomeBehavior) check to see whether the object has already been discarded. If it has, they throw an exception.

MODULE 4 [CHAPTER 1]

Implementing properties to access fields

Implementing encapsulation by using methods

First, will see the original motivation for using methods to hide fields.

- Consider the following structure that represents a position on a computer screen as a pair of coordinates, x and y. Assume that the range of valid values for the x-coordinate lies between 0 and 1280, and the range of valid values for the y-coordinate lies between 0 and 1024.

```
struct ScreenPosition
{
public int X; public int Y;
public ScreenPosition(int x, int y)
{
        this.X = rangeCheckedX(x);
        this.Y = rangeCheckedY(y);
}
private static int rangeCheckedX(int x)
{
        if (x < 0 || x > 1280)
        {
        throw new ArgumentOutOfRangeException("X");
        }
return x;
}
private static int rangeCheckedY(int y)
{
        if (y < 0 || y > 1024)
        {
        throw new ArgumentOutOfRangeException("Y");
        }
return y;
} }
```

- One problem with this structure is that it does not follow the golden rule of encapsulation—that is, it does not keep its data private. Public data is often a bad idea because the class cannot control the values that an application specifies.

- For example, the *ScreenPosition* constructor range checks its parameters to ensure that they are in a specified range, but no such check can be done on the "raw" access to the public fields.

```
ScreenPosition origin = new ScreenPosition(0, 0);
...
```

```
int xpos = origin.X;
origin.Y = -100; // oops
```

- The common way to solve this problem is to make the fields private and add an accessor method and a modifier method to respectively read and write the value of each private field.

```
struct ScreenPosition
{
...
public int GetX()
{
return this.x;
}
public void SetX(int newX)
{
 this.x = rangeCheckedX(newX);
}
...
private static int rangeCheckedX(int x) { ... }
private static int rangeCheckedY(int y) { ... }
private int x, y;
}
```

<u>What are properties?</u>

A *property* is a cross between a field and a method—it looks like a field but acts like a method. You access a property by using exactly the same syntax that you use to access a field.

The syntax for a property declaration looks like this:

```
AccessModifier Type PropertyName
{
Get
 {
 // read accessor code
}
set
{
 // write accessor code
 }
}
```

- A property can contain two blocks of code, starting with the *get* and *set* keywords. The *get* block contains statements that execute when the property is read, and the *set* block contains statements that run upon writing to the property. The type of the property specifies the type of data read and written by the *get* and *set* accessors.
- The next code example shows the *ScreenPosition* structure rewritten by using properties. When looking at this code, notice the following:

- Lowercase *_x* and *_y* are *private* fields.
- Uppercase *X* and *Y* are *public* properties.
- All *set* accessors are passed the data to be written by using a hidden, built-in parameter named *value*.

```
struct ScreenPosition
{
private int _x, _y;
public ScreenPosition(int X, int Y)
 {
     this._x = rangeCheckedX(X);
      this._y = rangeCheckedY(Y);
 }
 public int X
 {
     get { return this._x; }
     set { this._x = rangeCheckedX(value);
 }
 }
public int Y
 {
     get { return this._y; }
     set { this._y = rangeCheckedY(value);
 }
 }
private static int rangeCheckedX(int x) { ... }
private static int rangeCheckedY(int y) { ... }
}
```

- In this example, a private field directly implements each property, but this is only one way to implement a property. All that is required is that a *get* accessor returns a value of the specified type.
- When you use a property in an expression, you can use it in a read context (when you are retrieving its value) and in a write context (when you are modifying its value). The following example shows how to read values from the *X* and *Y* properties of the *ScreenPosition* structure:

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X; // calls origin.X.get
int ypos = origin.Y; // calls origin.Y.get
```

- Notice that you access properties and fields by using identical syntax. When you use a property in a read context, the compiler automatically translates your field-like code into a call to the *get* accessor of that property.
- Similarly, if you use a property in a write context, the compiler automatically translates your field-like code into a call to the *set* accessor of that property.

```
origin.X = 40; // calls origin.X.set, with value set to 40origin.
Y = 100;        // calls origin.Y.Set, with value set to 100
```

Read-only properties

- You can declare a property that contains only a *get* accessor. In this case, you can use the property only in a read context. For example, here's the *X* property of the *ScreenPosition* structure declared as a read-only property:

```
struct ScreenPosition
{
 private int _x;
 ...
public int X
{
get { return this._x; }
 }
}
```

- The *X* property does not contain a *set* accessor; therefore, any attempt to use *X* in a write context will fail, as demonstrated in the following example:

```
origin.X = 140; // compile-time error
```

Write-only properties

- Similarly, you can declare a property that contains only a *set* accessor. In this case, you can use the property only in a write context. For example, here's the *X* property of the *ScreenPosition* structure declared as a write-only property:

```
struct ScreenPosition
{
private int _x;
 ...
 public int X
 {
Set { this._x = rangeCheckedX(value); }
}}
```

The *X* property does not contain a *get* accessor; any attempt to use *X* in a read context will fail, as illustrated here:

```
Console.WriteLine(origin.X); // compile-time error
origin.X = 200; // compiles OK
origin.X += 10; // compile-time error
```

Property accessibility

- You can specify the accessibility of a property (*public*, *private*, or *protected*) when you declare it. However, it is possible within the property declaration to override the property accessibility for the *get* and *set* accessors.

- For example, the version of the *ScreenPosition* structure shown in the code that follows defines the *set* accessors of the *X* and *Y* properties as *private*. (The *get* accessors are *public*, because the properties are *public*.)

```
struct ScreenPosition
{
private int _x, _y;
 ...
public int X
 {
 get { return this._x; }
 private set { this._x = rangeCheckedX(value); }
}
public int Y
 {
get { return this._y; }
private set { this._y = rangeCheckedY(value); }
 }
...}
```

You must observe some rules when defining accessors with different accessibility from one another:

- You can change the accessibility of only one of the accessors when you define it. It wouldn't make much sense to define a property as *public* only to change the accessibility of both accessors to *private* anyway.
- The modifier must not specify an accessibility that is less restrictive than that of the property. For example, if the property is declared as *private*, you cannot specify the read accessor as *public*. (Instead, you would make the property *public* and make the write accessor *private*.)

Understanding the property restrictions

- You can assign a value through a property of a structure or class only after the structure or class has been initialized. The following code example is illegal because the location variable has not been initialized (by using *new*):
  ```
  ScreenPosition location;
  location.X = 40; // compile-time error, location not assigned
  ```

- You can't use a property as a *ref* or an *out* argument to a method (although you can use a writable field as a *ref* or an *out* argument). This makes sense because the property doesn't really point to a memory location; rather, it points to an accessor method, such as in the following example:
  ```
  MyMethod(ref location.X); // compile-time error
  ```

- A property can contain at most one *get* accessor and one *set* accessor. A property cannot contain other methods, fields, or properties.

- The *get* and *set* accessors cannot take any parameters. The data being assigned is passed to the *set* accessor automatically by using the *value* variable.
- You can't declare *const* properties, such as is demonstrated here:
  const int X { get { ... } set { ... } } // compile-time error

Declaring interface properties

- We specify the *get* or *set* keyword, or both, but replace the body of the *get* or *set* accessor with a semicolon, as shown here:

```
interface IScreenPosition
{
     int X { get; set; } int Y { get; set; }

}
```

- Any class or structure that implements this interface must implement the *X* and *Y* properties with *get* and *set* accessor methods.

```
struct ScreenPosition : IScreenPosition
{ ...
    public int X { get { ... } set { ... } }
    public int Y { get { ... } set { ... } }
 ...}
```

- If you implement the interface properties in a class, you can declare the property implementations as *virtual*, which enables derived classes to override the implementations.

```
class ScreenPosition : IScreenPosition
{ ...
    public virtual int X { get { ... } set { ... } }
     public virtual int Y { get { ... } set { ... } }
}
```

- You can also choose to implement a property by using the explicit interface implementation

```
struct ScreenPosition : IScreenPosition
{ ...
         int IScreenPosition.X { get { ... } set { ... } }
         int IScreenPosition.Y { get { ... } set { ... } }
... }
```

Generating automatic properties

- The principal purpose of properties is to hide the implementation of fields from the outside world. However, there are at least two good reasons why you should define properties rather than exposing data as public fields even in these situations:

- **Compatibility with applications** Fields and properties expose themselves by using different metadata in assemblies. If you develop a class and decide to use public fields, any applications that use this class will reference these items as fields. Although you use the same C# syntax for reading and writing a field that you use when reading and writing a property, the compiled code is actually quite different—the C# compiler just hides the differences from you. If you later decide that you really do need to change these fields to properties, existing applications will not be able to use the updated version of the class without being recompiled. This is awkward if you have deployed the application on a large number of users' desktops throughout an organization.

- **Compatibility with interfaces** If you are implementing an item as a property, you must write a property that matches the specification in the interface, even if the property just reads and writes data in a private field.

- The designers of the C# language recognized that programmers are busy people who should not have to waste their time writing more code than they need to. To this end, the C# compiler can generate the code for properties for you automatically, like this:

```
class Circle
{
public int Radius{ get; set; }
 ...}
```

- The C# compiler converts this definition to a private field and a default implementation that looks similar to this:

```
class Circle
{
private int _radius;
public int Radius
{
get { return this._radius; }
set { this._radius = value; }
}
 ...}
```

- We can implement a simple property by using automatically generated code, and if you need to include additional logic later, you can do so without breaking any existing applications.

Initializing objects by using properties

- Object initializing using contractor: An object can have multiple constructors, and you can define constructors with varying parameters to initialize different elements in an object. For example, you could define a class that models a triangle, like this:

```
public class Triangle
{
                private int side1Length; private int side2Length; private int side3Length;
        public Triangle()
                {
                 this.side1Length = this.side2Length = this.side3Length = 10;
                }
        public Triangle(int length1)
                {
                 this.side1Length = length1; this.side2Length = this.side3Length = 10;
                 }
        public Triangle(int length1, int length2)
                {
                this.side1Length = length1; this.side2Length = length2; this.side3Length = 10;
                }
        public Triangle(int length1, int length2, int length3)
                {
                this.side1Length = length1; this.side2Length = length2; this.side3Length =
                length3;
                }
}
```

- Problem with constructor :For example, in the preceding *Triangle* class, you could not easily add a constructor that initializes only the *side1Length* and *side3Length* fields because it would not have a unique signature;
- One possible solution is to define a constructor that takes optional parameters and specify values for the parameters as named arguments when you create a *Triangle* object.
- Another transparent solution is to initialize the private fields to a set of default values and expose them as properties, like this:

```
        public class Triangle
        {
        private int side1Length = 10;
         private int side2Length = 10;
        private int side3Length = 10;
         public int Side1Length { set { this.side1Length = value; } }
        public int Side2Length { set { this.side2Length = value; } }
        public int Side3Length { set { this.side3Length = value; } }
        }
```

- When we create an instance of a class, you can initialize it by specifying the names and values for any public properties that have *set* accessors.

```
Triangle tri1 = new Triangle { Side3Length = 15 };
Triangle tri2 = new Triangle { Side1Length = 15, Side3Length = 20 };
Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };
Triangle tri4 = new Triangle { Side1Length = 9, Side2Length = 12, Side3Length = 15 };
```

Using indexers[Chapter 2]

What is an indexer?

- We can think of an *indexer* as a smart array in much the same way that you can think of a property as a smart field. Whereas a property encapsulates a single value in a class, an indexer encapsulates a set of values. The syntax that you use for an indexer is exactly the same as the syntax that you use for an array.

The best way to understand indexers is to work through an example.

An example that doesn't use indexers

- C# provides a set of operators that you can use to access and manipulate the individual bits in an *int*. These operators are as follows:
- **The NOT (~) operator** This is a unary operator that performs a bitwise complement. For example, if you take the 8-bit value *11001100* (*204* decimal) and apply the ~ operator to it, you obtain the result *00110011* (*51* decimal)—all the 1s in the original value become 0s, and all the 0s become 1s.
- **The left-shift (<<) operator** This is a binary operator that performs a left shift. The expression *204 << 2* returns the value *48*. (In binary, *204* decimal is *11001100*, and left-shifting it by two places yields *00110000*, or *48* decimal.) The far-left bits are discarded, and zeros are introduced from the right. There is a corresponding right-shift operator, >>.
- **The OR (|) operator** This is a binary operator that performs a bitwise OR operation, returning a value containing a 1 in each position in which either of the operands has a 1. For example, the expression *204 | 24* has the value *220* (*204* is *11001100, 24* is *00011000*, and *220* is *11011100*).
- **The AND (&) operator** This operator performs a bitwise AND operation. AND is similar to the bitwise OR operator, except that it returns a value containing a 1 in each position where both of the operands have a 1. So, *204 & 24* is *8* (*204* is *11001100, 24* is *00011000*, and *8* is *00001000*).
- **The XOR (^) operator** This operator performs a bitwise exclusive OR operation, return-ing a 1 in each bit where there is a 1 in one operand or the other but not both. (Two 1s yield a 0—this is the "exclusive" part of the operator.) So *204 ^ 24* is *212* (*11001100 ^ 00011000* is *11010100*).
- (bits & (1 << 5)) != 0
  Suppose that the *bits* variable contains the decimal value 42. In binary, this is 00101010. The decimal value 1 is 00000001 in binary, and the expression *1 << 5* has the value 00100000; the sixth bit is 1. In binary, the expression *bits & (1 << 5)* is *00101010 & 00100000*, and the value of this expression is binary 00100000, which is nonzero. If the variable *bits* contains the value *65,*
- The trouble with these examples is that although they work, they are fiendishly difficult to understand. They're complicated, and the solution is a very low-level one: it fails to create an abstraction of the problem that it solves, and it is consequently very difficult to maintain code that performs these kinds of operations.

The same example using indexers

- The best way to solve above problem is to use an *int* as if it were an array of bits.

  Eq:

  bits[5]   // access the bit 6 places from the right in the *bits* variable

  bits[3] = true   // set the bit 4 places from the right to *true*

- We can't use the square bracket notation on an *int*; it works only on an array or on a type that behaves like an array. So, the solution to the problem is to create a new type that acts like, feels like, and is used like an array of *bool* variables but is implemented by using an *int*. You can achieve this feat by defining an indexer.

```
struct IntBits
{       private int bits;
        public IntBits(int initialBitValue)
        { bits = initialBitValue; }
}
```

- To define the indexer, you use a notation that is a cross between a property and an array. You introduce the indexer with the *this* keyword, specify the type of the value returned by the indexer, and also specify the type of the value to use as the index into the indexer between square brackets.

```
public bool this [ int index ]
{
  get
  {
          return (bits & (1 << index)) != 0;
  }
  set
  {
           if (value)        // turn the bit on if value is true; otherwise, turn it off
              bits |= (1 << index);
        else
              bits &= ~(1 << index);
  }
}
```

Notice the following points:

- An indexer is not a method; there are no parentheses containing a parameter, but there are square brackets that specify an index. This index is used to specify which element is being accessed.
- All indexers use the *this* keyword. A class or structure can define at most one indexer and it is always named *this*.
- Indexers contain *get* and *set* accessors just like properties. In this example, the *get* and *set* accessors contain the complicated bitwise expressions previously discussed.

- The index specified in the indexer declaration is populated with the index value specified when the indexer is called. The *get* and *set* accessor methods can read this argument to determine which element should be accessed.
- After you have declared the indexer, you can use a variable of type *IntBits* instead of an *int* and apply the square bracket notation, as shown in the next example:

```
int adapted = 126;                  // 126 has the binary representation 01111110
IntBits bits = new IntBits(adapted);
bool peek = bits[6];                // retrieve bool at index 6; should be true (1)
bits[0] = true;                     // set the bit at index 0 to true (1)
bits[3] = false;                     // set the bit at index 3 to false (0)
```

Understanding indexer accessors

When we read an indexer, the compiler automatically translates your array-like code into a call to the *get* accessor of that indexer. Consider the following example:

- bool peek = bits[6]; // call to the *get* accessor for *bits*, and the *index* argument is set to *6*.
- bits[3] = true;   // call to the *set* accessor of that indexer, setting the *index* value to true
- bits[6] ^= true;   //This code is automatically translated into the following:
  bits[6] = bits[6] ^ true; //calls the indexer declares both a *get* and a *set* accessor.

Comparing indexers and arrays

When we use an indexer, the syntax is deliberately array-like. However, there are some important differences between indexers and arrays:

1. Indexers can use non-numeric subscripts, such as a string as shown in the following example, whereas arrays can use only integer subscripts.
   ```
   public int this [ string name ] { ... } // OK
   ```
2. Indexers can be overloaded (just like methods), whereas arrays cannot.
   ```
   public Name this [ PhoneNumber number ]
   { ... }
   public PhoneNumber this [ Name name ]
    { ... }
   ```
3. Indexers cannot be used as *ref* or *out* parameters, whereas array elements can.
   ```
   IntBits bits; // bits contains an indexerMethod(ref bits[1]); // compile-time error
   ```

Indexers in interfaces

- We can declare indexers in an interface. To do this, specify the *get* keyword, the *set* keyword, or both, but replace the body of the *get* or *set* accessor with a semicolon. Any class or structure that implements the interface must implement the *indexer* accessors declared in the interface, as demonstrated here:

```
interface IRawInt
{
        bool this [ int index ] { get; set; }
}
```

```
struct RawInt : IRawInt
{ ...
      public bool this [ int index ]
      { get { ... } set { ... } }
 ...}
```

- Implement the interface indexer in a class, we can declare the indexer implementations as *virtual*.

```
class RawInt : IRawInt
{ ...
     public virtual bool this [ int index ]
     { get { ... } set { ... } }
 ...}
```

- We can also choose to implement an indexer by using the explicit interface implementation syntax.An explicit implementation of an indexer is nonpublic and nonvirtual (and so cannot be overridden), as shown in this example:

```
struct RawInt : IRawInt
{ ...
          bool IRawInt.this [ int index ]
          { get { ... } set { ... } }
 ...}
```

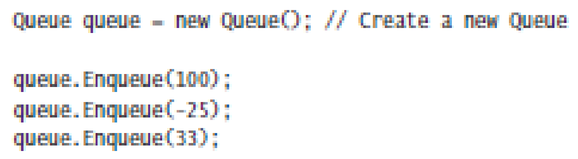
MODULE 4[CHAPTER 3]
Introducing generics

The problem with the *object* type

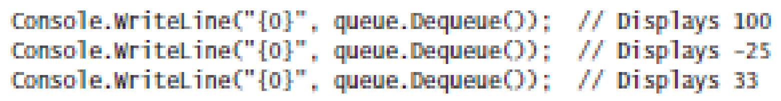
```
class Queue
{
    private const int DEFAULTQUEUESIZE = 100;
    private int[] data;
    private int head = 0, tail = 0;
    private int numElements = 0;

    public Queue()
    {
        this.data = new int[DEFAULTQUEUESIZE];
    }


      public Queue(int size)
      {
          if (size > 0)
          {
              this.data = new int[size];
          }
          else
          {
              throw new ArgumentOutOfRangeException("size", "Must be greater than zero");
          }
      }

      public void Enqueue(int item)
      {
          if (this.numElements == this.data.Length)
          {
              throw new Exception("Queue full");
          }

          this.data[this.head] = item;
          this.head++;
          this.head %= this.data.Length;
          this.numElements++;
      }

      public int Dequeue()
      {
          if (this.numElements == 0)
          {
              throw new Exception("Queue empty");
          }

          int queueItem = this.data[this.tail];
          this.tail++;
          this.tail %= this.data.Length;
          this.numElements--;
          return queueItem;
      }
  }
```

- To understand generics, it is worth looking in detail at the problem for which they are designed to solve. Suppose that we needed to model a first-in, first-out structure such as a queue. We could create a class such as the above it.

- This class uses an array to provide a circular buffer for holding the data. The size of this array is specified by the constructor. An application uses the *Enqueue* method to add an item to the queue and the *Dequeue* method to pull an item off the queue. The private *head* and *tail* fields keep track of where to insert an item into the array and where to retrieve an item from the array. The *numElements* field indicates how many items are in the array.



```
Queue queue = new Queue(); // Create a new Queue

queue.Enqueue(100);
queue.Enqueue(-25);
queue.Enqueue(33);
```



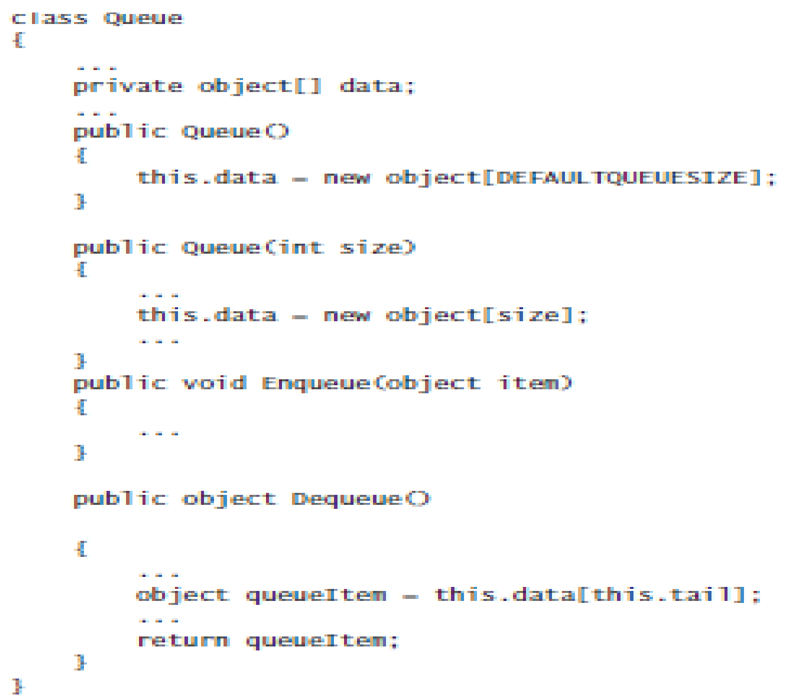
```
Console.WriteLine("{0}", queue.Dequeue());  // Displays 100
Console.WriteLine("{0}", queue.Dequeue());  // Displays -25
Console.WriteLine("{0}", queue.Dequeue());  // Displays 33
```

- The *Queue* class works well for queues of *int*s, but what if Wewant to create queues of strings, or floats, or even queues of more complex types such as *Circle*

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); //  error: Cannot convert from Horse to int
```

- One way around this restriction is to specify that the array in the *Queue* class contains items of type *object*, update the constructors, and modify the *Enqueue* and *Dequeue* methods to take an object parameter and return an object, such as in the following:

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Now legal – Horse is an object
```



```
class Queue
{
    ...
    private object[] data;
    ...
    public Queue()
    {
        this.data = new object[DEFAULTQUEUESIZE];
    }

    public Queue(int size)
    {
        ...
        this.data = new object[size];
        ...
    }
    public void Enqueue(object item)
    {
        ...
    }

    public object Dequeue()

    {
        ...
        object queueItem = this.data[this.tail];
        ...
        return queueItem;
    }
}
```

- Disadvantage of using the *object* approach to create generalized classes and methods is that it can consume additional memory and processor time if the runtime needs to convert an *object* to a value type and back again. Consider the following piece of code that manipulates a queue of *int* values:

```
Queue queue = new Queue();
int myInt = 99;queue.Enqueue(myInt); // box the int to an object
...
myInt = (int)queue.Dequeue(); // unbox the object to an int
```

- The *Queue* data type expects the items it holds to be objects, and *object* is a reference type. Enqueueing a value type, such as an *int*, requires it to be boxed to convert it to a reference type. Similarly, dequeueing into an *int* requires the item to be unboxed to convert it back to a value type.

The generics solution

- C# provides generics to remove the need for casting, improve type safety, reduce the amount of boxing required, and make it easier to create generalized classes and methods. Generic classes and methods accept *type parameters*, which specify the types of objects on which they operate.

```
class Queue<T>
{
...
}
```

The *T* in this example acts as a placeholder for a real type at compile time. When Wewrite code to instantiate a generic *Queue*, Weprovide the type that should be substituted for *T* (*Circle*, *Horse*, *int*, and so on).

```
class Queue<T>
{
    ...
    private T[] data; // array is of type 'T' where 'T' is the type parameter
    ...
    public Queue()
    {
        this.data = new T[DEFAULTQUEUESIZE]; // use 'T' as the data type
    }

    public Queue(int size)
    {
        ...
        this.data = new T[size];
        ...
    }
    public void Enqueue(T item)  // use 'T' as the type of the method parameter
    {
        ...
    }

    public T Dequeue() // use 'T' as the type of the return value
    {
        ...
        T queueItem = this.data[this.tail];  // the data in the array is of type 'T'
        ...
        return queueItem;
    }
}
```

- The following examples create a *Queue* of *int*s, and a *Queue* of *Horse*s:

    Queue<int> intQueue = new Queue<int>();
    Queue<Horse> horseQueue = new Queue<Horse>();

- The compiler now has enough information to perform strict type-checking when Webuild the application. Weno longer need to cast data when we call the *Dequeue* method, and the compiler can trap any type mismatch errors early:

```
intQueue.Enqueue(99);
int myInt = intQueue.Dequeue(); // no casting necessary
Horse myHorse = intQueue.Dequeue(); // compiler error:
                    // cannot implicitly convert type 'int' to 'Horse'
```

Generics vs. generalized classes

- It is important to be aware that a generic class that uses type parameters is different from a *generalized* class designed to take parameters that can be cast to different types. For example, the object-based version of the *Queue* class shown earlier is a generalized class.
- There is a *single* implementation of this class, and its methods take *object* parameters and return *object* types. Wecan use this class with *ints*, *string*s, and many other types, but in each case, Weare using instances of the same class and Wehave to cast the data Weare using to and from the *object* type.

- Compare this with the *Queue<T>* class. Each time Weuse this class with a type parameter (such as *Queue<int>* or *Queue<Horse>*), Wecause the compiler to generate an entirely new class that happens to have functionality defined by the generic class.

Generics and constraints

- By using a constraint, Wecan limit the type parameters of a generic class to those that implement a particular set of interfaces and therefore provide the methods defined by those interfaces. For example, if the *IPrintable* interface defined the *Print* method, Wecould create the *PrintableCollection* class like this:

```
public class PrintableCollection<T> where T : IPrintable
```

- When Webuild this class with a type parameter, the compiler checks to ensure that the type used for *T* actually implements the *IPrintable* interface; if it doesn't, it stops with a compilation error.

Creating a generic method

- Generic methods are frequently used in conjunction with generic classes; we need them for methods that take generic types as parameters or that have a return type that is a generic type.
- With a generic method, Wecan specify the types of the parameters and the return type by using a type parameter in a manner similar to that used when defining a generic class. We define generic methods by using the same type parameter syntax that Weuse when creating generic classes. For example, the generic *Swap<T>* method in the code that follows swaps the values in its parameters. Because this functionality is useful regardless of the type of data being swapped, it is helpful to define it as a generic method:

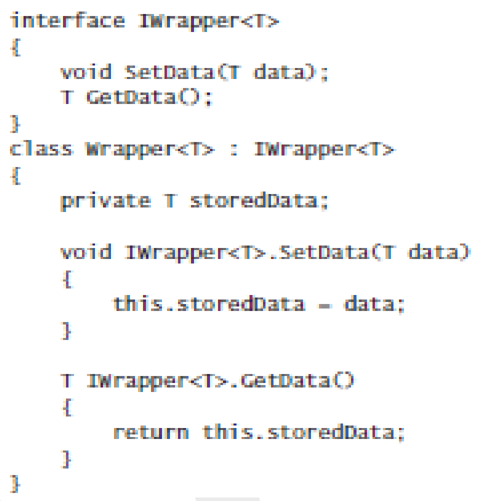
```
static void Swap<T>(ref T first, ref T second)
{
 T temp = first;
 first = second;
 second = temp;
}
```

- Invoke the method by specifying the appropriate type for its type parameter. The following examples show how to invoke the *Swap<T>* method to swap over two *int*s and two *string*s:

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
...string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2);
```

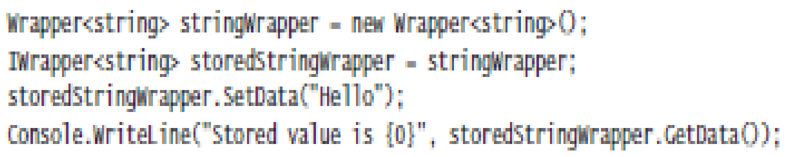
Variance and generic interfaces

- The *Wrapper<T>* class provides a simple wrapper around a specified type. The *IWrapper* interface defines the *SetData* method that the *Wrapper<T>* class implements to store the data and the *GetData* method that the *Wrapper<T>* class implements to retrieve the data.



```
interface IWrapper<T>
{
    void SetData(T data);
    T GetData();
}
class Wrapper<T> : IWrapper<T>
{
    private T storedData;

    void IWrapper<T>.SetData(T data)
    {
        this.storedData = data;
    }

    T IWrapper<T>.GetData()
    {
        return this.storedData;
    }
}
```

- we can create an instance of this class and use it to wrap a string like this:



```
Wrapper<string> stringWrapper = new Wrapper<string>();
IWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
Console.WriteLine("Stored value is {0}", storedStringWrapper.GetData());
```

Covariant interfaces

- Suppose that we defined the *IStoreWrapper<T>* and *IRetrieveWrapper<T>* interfaces, shown in the following example, in place of *IWrapper<T>* and implemented these interfaces in the *Wrapper<T>* class, like this:

```
interface IStoreWrapper<T>
{
    void SetData(T data);
}

interface IRetrieveWrapper<T>
{
    T GetData();
}

class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>
{
   private T storedData;

   void IStoreWrapper<T>.SetData(T data)
   {
       this.storedData = data;
   }

   T IRetrieveWrapper<T>.GetData()
   {
       return this.storedData;
   }
}
```

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IStoreWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;
Console.WriteLine("Stored value is {0}", retrievedStringWrapper.GetData());
```

IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper; //error

- The compiler does implicit conversions are legal and that it does not have to enforce strict type-safety. We do this by specifying the *out* keyword when Wedeclare the type parameter, like this:

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

- This feature is called *covariance*. Wecan assign an *IRetrieveWrapper<A>* object to an *IRetrieveWrapper<B>* reference as long as there is a valid conversion from type *A* to type *B*, or type *A* derives from type *B*. The following code now compiles and runs as expected:

```
 // string derives from object, so this is now legal
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

Contravariant interfaces

- Contravariance follows a similar principle to covariance except that it works in the opposite direction; it enables Weto use a generic interface to reference an object of type *B* through a reference to type *A* as long as type *B* derives type *A*. This sounds complicated, so it is worth looking at an example from the .NET Framework class library.
- The *System.Collections.Generic* namespace in the .NET Framework provides an interface called *IComparer*, which looks like this:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

- A class that implements this interface has to define the *Compare* method, which is used to compare two objects of the type specified by the *T* type parameter. The *Compare* method is expected to return an integer value: zero if the parameters *x* and *y* have the same value, negative if *x* is less than *y*, and positive if *x* is greater than *y*.

```
class ObjectComparer : IComparer<Object>
{
    int IComparer<Object>.Compare(Object x, Object y)
    {
        int xHash = x.GetHashCode();
        int yHash = y.GetHashCode();

        if (xHash == yHash)
            return 0;

        if (xHash < yHash)
            return -1;

        return 1;
    }
}
```

- We can create an *ObjectComparer* object and call the *Compare* method through the *IComparer<Object>* interface to compare two objects, like this:

```
Object x = ...;
Object y = ...;
ObjectComparer objectComparer = new ObjectComparer();
IComparer<Object> objectComparator = objectComparer;
int result = objectComparator.Compare(x, y);
```

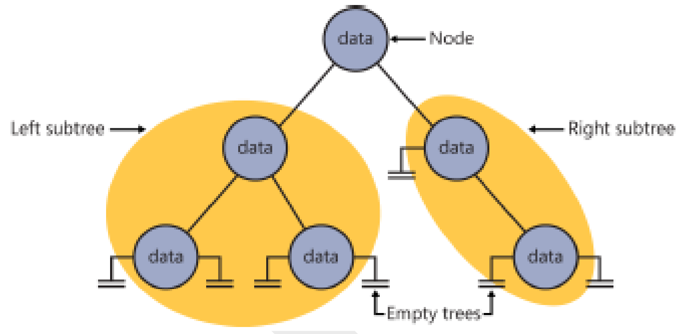
Covariance example If the methods in a generic interface can return strings, they can also return objects. (All strings are objects.)

Contravariance example If the methods in a generic interface can take object parameters, they can take string parameters.

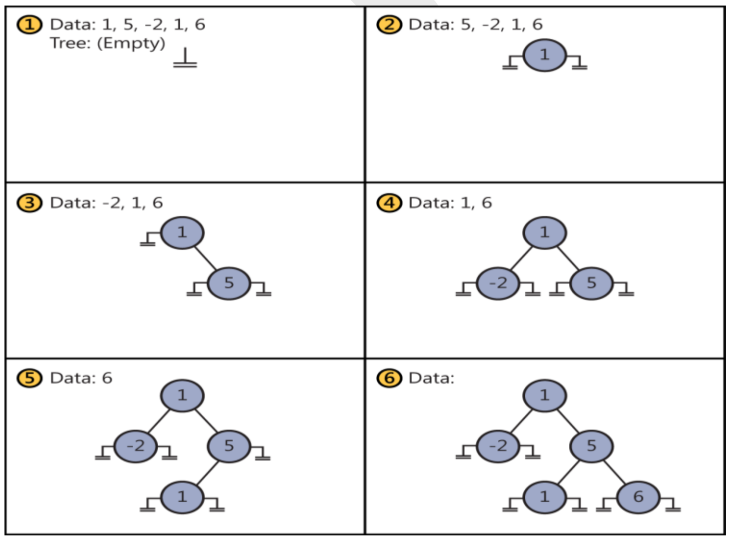
Creating a generic class

The theory of binary trees

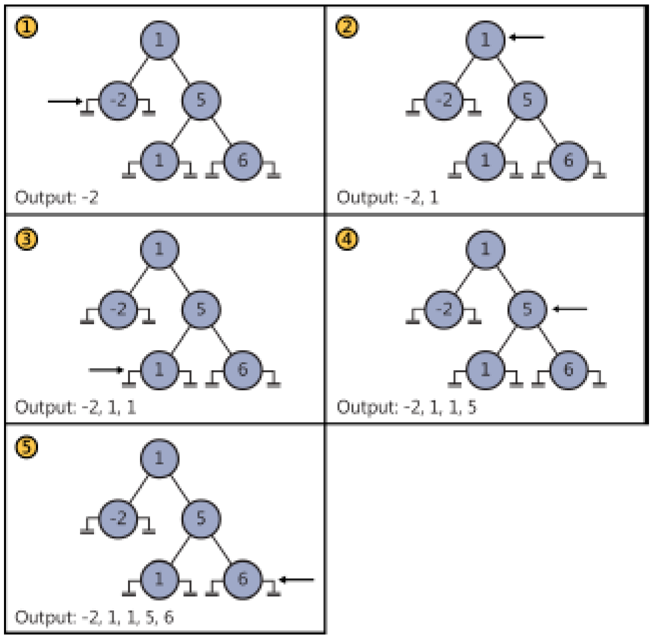
- A *binary tree* is a useful data structure that Wecan use for a variety of operations, including sorting and searching through data very quickly.
- A binary tree is a recursive (self-referencing) data structure that can either be empty or contain three elements: a datum, which is typically referred to as the *node*, and two subtrees, which are themselves binary trees. The two subtrees are conventionally called the *left subtree* and the *right subtree* because they are typically depicted to the left and right of the node, respectively.
- Each left subtree or right subtree is either empty or contains a node and other subtrees. In theory, the whole structure can continue ad infinitum. The following image shows the structure of a small binary tree.



Binary tree construction



Binary tree display



C# program for Btree using generic class and method:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
    public TItem NodeData { get; set; }
    public Tree<TItem> LeftTree { get; set; }
    public Tree<TItem> RightTree { get; set; }
}
```

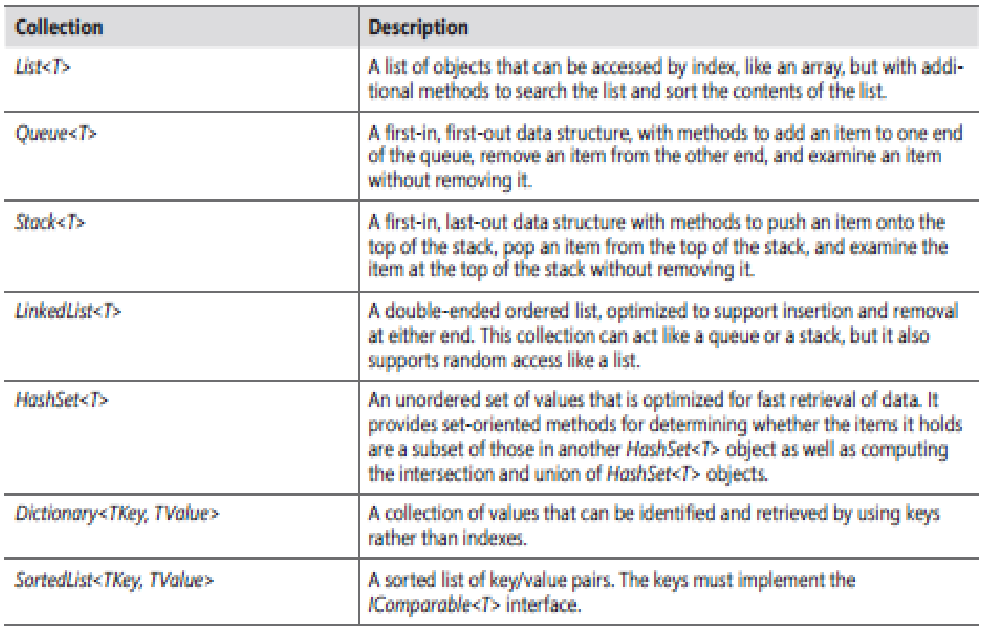
```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree,
params TItem[] data) where TItem : IComparable<TItem>
{
    foreach (TItem datum in data)
    {
        if (tree == null)
        {
            tree = new Tree<TItem>(datum);
        }
        else
        {
            tree.Insert(datum);
        }
    }
}
```

```
public string WalkTree()
{
    string result = "";

    if (this.LeftTree != null)
    {
        result = this.LeftTree.WalkTree();
    }

    result += String.Format(" {0} ", this.NodeData.ToString());

    if (this.RightTree != null)
    {
        result += this.RightTree.WalkTree();
    }

    return result;
}
```

```
static void Main(string[] args)
{
    Tree<int> tree1 = new Tree<int>(10);
    tree1.Insert(5);
    tree1.Insert(11);
    tree1.Insert(5);
    tree1.Insert(-12);
    tree1.Insert(15);
    tree1.Insert(0);
    tree1.Insert(14);
    tree1.Insert(-8);
    tree1.Insert(10);
    tree1.Insert(8);
    tree1.Insert(8);

    string sortedData = tree1.WalkTree();
    Console.WriteLine("Sorted data is: {0}", sortedData);
}
```

MODULE 4[CHAPTER 4]
Using collections

- The Microsoft .NET Framework provides several classes that collect elements together such that an application can access them in specialized ways. they live in the *System.Collections.Generic* namespace.
- As the namespace implies, these collections are generic types; they all expect We to provide a type parameter indicating the kind of data that your application will be storing in them. Each collection class is optimized for a particular form of data storage and access, and each provides specialized methods that support this functionality.
- For example, the *Stack<T>* class implements a last-in, first-out model, where We add an item to the top of the stack by using the *Push* method, and Wetake an item from the top of the stack by using the *Pop* method. The *Pop* method always retrieves the most recently pushed item and removes it from the stack. In contrast, the *Queue<T>* type provides the *Enqueue* and *Dequeue* methods The *Enqueue* method adds an item to the queue, whereas the *Dequeue* method retrieves items in the same order and removes them from the queue.



Collection	Description
List<T>	A list of objects that can be accessed by index, like an array, but with additional methods to search the list and sort the contents of the list.
Queue<T>	A first-in, first-out data structure, with methods to add an item to one end of the queue, remove an item from the other end, and examine an item without removing it.
Stack<T>	A first-in, last-out data structure with methods to push an item onto the top of the stack, pop an item from the top of the stack, and examine the item at the top of the stack without removing it.
LinkedList<T>	A double-ended ordered list, optimized to support insertion and removal at either end. This collection can act like a queue or a stack, but it also supports random access like a list.
HashSet<T>	An unordered set of values that is optimized for fast retrieval of data. It provides set-oriented methods for determining whether the items it holds are a subset of those in another *HashSet<T>* object as well as computing the intersection and union of *HashSet<T>* objects.
Dictionary<TKey, TValue>	A collection of values that can be identified and retrieved by using keys rather than indexes.
SortedList<TKey, TValue>	A sorted list of key/value pairs. The keys must implement the *IComparable<T>* interface.

The *List<T>* collection class

- The generic *List<T>* class is the simplest of the collection classes. We can use it much like an array—We can reference an existing element in a *List<T>* collection by using ordinary array notation, with square brackets and the index of the element, although We cannot use array notation to add new elements. However, in general, the *List<T>* class provides more flexibility than arrays and is designed to overcome the following restrictions exhibited by arrays:

1) If We want to resize an array, We have to create a new array, copy the elements (leaving out some if the new array is smaller), and then update any references to the original array so that they refer to the new array.
2) If We want to remove an element from an array, We have to move all the trailing elements up by one place. Even this doesn't quite work, because We end up with two copies of the last element.
3) If We want to insert an element into an array, We have to move elements down by one place to make a free slot. However, We lose the last element of the array!

- The *List<T>* collection class provides the following features that preclude these limitations:

  1) We don't need to specify the capacity of a *List<T>* collection when We create it; it can grow and shrink as We add elements. There is an overhead associated with this dynamic behavior, and if necessary We can specify an initial size. However, if We exceed this size, then the *List<T>* collection will simply grow as necessary.

  2)We can remove a specified element from a *List<T>* collection by using the *Remove* method. The *List<T>* collection automatically reorders its elements and closes the gap. We can also remove an item at a specified position in a *List<T>* collection by using the *RemoveAt* method.

  3) We can add an element to the end of a *List<T>* collection by using its *Add* method. We supply the element to be added. The *List<T>* collection resizes itself automatically.

  4) We can insert an element into the middle of a *List<T>* collection by using the *Insert* method. Again, the *List<T>* collection resizes itself.

  5)We can easily sort the data in a *List<T>* object by calling the *Sort* method.

```
using System;
using System.Collections.Generic;
...
List<int> numbers = new List<int>();

// Fill the List<int> by using the Add method
foreach (int number in new int[12]{10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1})
{
    numbers.Add(number);
}

// Insert an element in the penultimate position in the list, and move the last item up
// The first parameter is the position; the second parameter is the value being inserted
numbers.Insert(numbers.Count-1, 99);

// Remove first element whose value is 7 (the 4th element, index 3)
numbers.Remove(7);
// Remove the element that's now the 7th element, index 6 (10)
numbers.RemoveAt(6);

// Iterate remaining 11 elements using a for statement
Console.WriteLine("Iterating using a for statement:");
for (int i = 0; i < numbers.Count; i++)
{
    int number = numbers[i];  // Note the use of array syntax
    Console.WriteLine(number);
}

// Iterate the same 11 elements using a foreach statement
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Here is the output of this code:

```
Iterating using a for statement:
10
9
8
7
6
5
4
```

```
Iterating using a foreach statement:
10
9
8
7
6
5
4
```

The *LinkedList<T>* collection class

- The *LinkedList<T>* collection class implements a doubly linked list. Each item in the list holds the value for the item together with a reference to the next item in the list (the *Next* property) and the previous item (the *Previous* property). The item at the start of the list has the *Previous* property set to *null*, and the item at the end of the list has the *Next* property set to *null*.
- *LinkedList<T>* does not support array notation for inserting or examining elements. Instead, Wecan use the *AddFirst* method to insert an element at the start of the list,

moving the first item up and setting its *Previous* property to refer to the new item, or the *AddLast* method to insert an element at the end of the list, setting the *Next* property of the previously last item to refer to the new item. We can also use the *AddBefore* and *AddAfter* methods to insert an element before or after a specified item in the list.

- We can find the first item in a *LinkedList<T>* collection by querying the *First* property, whereas the *Last* property returns a reference to the final item in the list. To iterate through a linked list, We can start at one end and step through the *Next* or *Previous* references until We find an item with a *null* value for this property. Alternatively, We can use a *foreach* statement, which iterates forward through a *LinkedList<T>* object and stops automatically at the end.
- We delete an item from a *LinkedList<T>* collection by using the *Remove, RemoveFirst*, and *RemoveLast* methods.

```
using System;
using System.Collections.Generic;
...
LinkedList<int> numbers = new LinkedList<int>();

// Fill the List<int> by using the AddFirst method
foreach (int number in new int[] { 10, 8, 6, 4, 2 })
{
    numbers.AddFirst(number);
}

// Iterate using a for statement
Console.WriteLine("Iterating using a for statement:");
for (LinkedListNode<int> node = numbers.First; node != null; node = node.Next)
{
    int number = node.Value;
    Console.WriteLine(number);
}

// Iterate using a foreach statement
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// Iterate backwards
Console.WriteLine("\nIterating list in reverse order:");
for (LinkedListNode<int> node = numbers.Last; node != null; node = node.Previous)
{
    int number = node.Value;
    Console.WriteLine(number);
}
```

Here is the output generated by this code:

```
Iterating using a for statement:
2
4
6
8
10

Iterating using a foreach statement:
2
4
6
8
10
```

The *Queue<T>* collection class

- The *Queue<T>* class implements a first-in, first-out mechanism. An element is inserted into the queue at the back (the *Enqueue* operation) and is removed from the queue at the front (the *Dequeue* operation).

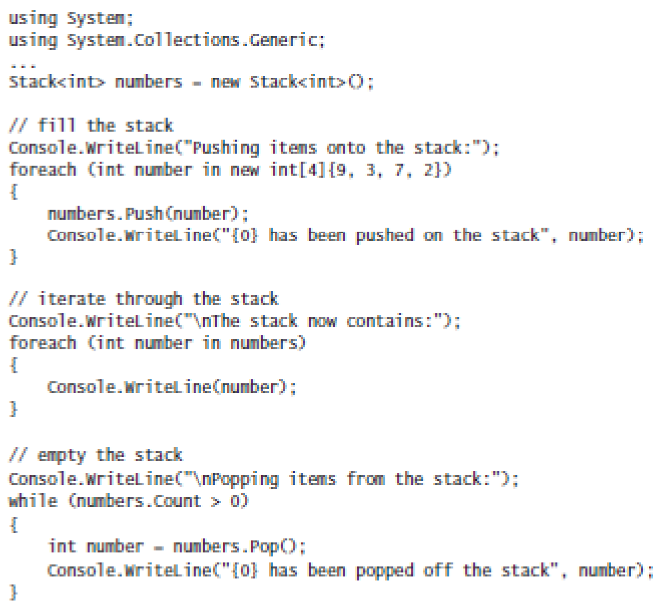
```
using System;
using System.Collections.Generic;
...
Queue<int> numbers = new Queue<int>();

// fill the queue
Console.WriteLine("Populating the queue:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Enqueue(number);
    Console.WriteLine("{0} has joined the queue", number);
}

// iterate through the queue
Console.WriteLine("\nThe queue contains the following items:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// empty the queue
Console.WriteLine("\nDraining the queue:");
while (numbers.Count > 0)
{
    int number = numbers.Dequeue();
    Console.WriteLine("{0} has left the queue", number);
}
```

Here is thee output from this code:

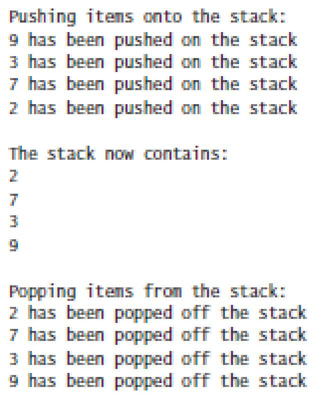
```
Populating the queue:
9 has joined the queue
3 has joined the queue
7 has joined the queue
2 has joined the queue


The queue contains the following items:
9
3
7
2

Draining the queue:
9 has left the queue
3 has left the queue
7 has left the queue
2 has left the queue
```

The *Stack<T>* collection class

- The *Stack<T>* class implements a last-in, first-out mechanism. An element joins the stack at the top (the push operation) and leaves the stack at the top (the pop operation). To visualize this, think of a stack of dishes: new dishes are added to the top and dishes are removed from the top, making the last dish to be placed on the stack the first one to be removed.



```
using System;
using System.Collections.Generic;
...
Stack<int> numbers = new Stack<int>();

// fill the stack
Console.WriteLine("Pushing items onto the stack:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Push(number);
    Console.WriteLine("{0} has been pushed on the stack", number);
}

// iterate through the stack
Console.WriteLine("\nThe stack now contains:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// empty the stack
Console.WriteLine("\nPopping items from the stack:");
while (numbers.Count > 0)
{
    int number = numbers.Pop();
    Console.WriteLine("{0} has been popped off the stack", number);
}
```



```
Pushing items onto the stack:
9 has been pushed on the stack
3 has been pushed on the stack
7 has been pushed on the stack
2 has been pushed on the stack

The stack now contains:
2
7
3
9

Popping items from the stack:
2 has been popped off the stack
7 has been popped off the stack
3 has been popped off the stack
9 has been popped off the stack
```

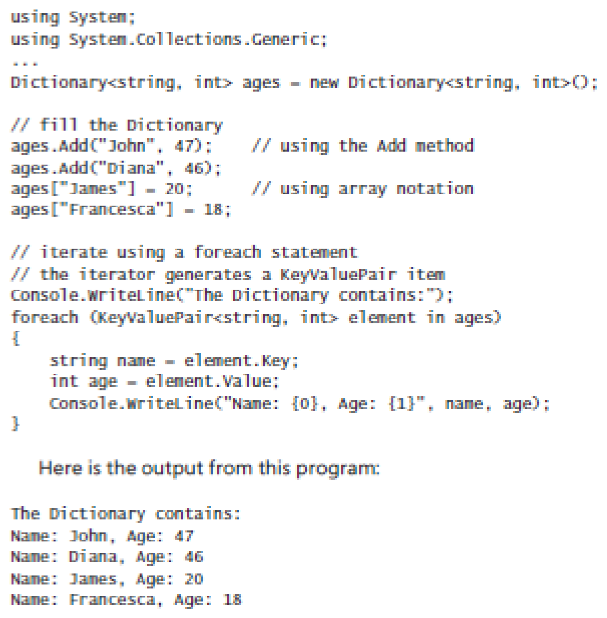
The *Dictionary<TKey, TValue>* collection class

- The array and *List<T>* types provide a way to map an integer index to an element, However, sometimes We might want to implement a mapping in which the type from which We map is not an *int* but rather some other type, such as *string, double*, or *Time*. In other languages, this is often called an *associative array*.
- The *Dictionary<TKey, TValue>* class implements this functionality by internally maintaining two arrays, one for the *keys* from which you're mapping and one for the *values* to which you're mapping. When We insert a key/value pair into a *Dictionary<TKey, TValue>* collection, it automatically tracks which key belongs to which value and makes it possible for Weto retrieve the value that is associated with a specified key quickly and easily.
- The design of the *Dictionary<TKey, TValue>* class has some important consequences:

  1) A *Dictionary<TKey, TValue>* collection cannot contain duplicate keys.

2) Internally, a *Dictionary<TKey, TValue>* collection is a sparse data structure that operates most efficiently when it has plenty of memory with which to work.

- When We use a *foreach* statement to iterate through a *Dictionary<TKey, TValue>* collection, We get back a *KeyValuePair<TKey, TValue>* item. This is a structure that contains a copy of the key and value elements of an item in the *Dictionary<TKey, TValue>* collection, and We can access each element through the *Key* property and the *Value* properties.



```
using System;
using System.Collections.Generic;
...
Dictionary<string, int> ages = new Dictionary<string, int>();

// fill the Dictionary
ages.Add("John", 47);    // using the Add method
ages.Add("Diana", 46);
ages["James"] = 20;      // using array notation
ages["Francesca"] = 18;

// iterate using a foreach statement
// the iterator generates a KeyValuePair item
Console.WriteLine("The Dictionary contains:");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

Here is the output from this program:

```
The Dictionary contains:
Name: John, Age: 47
Name: Diana, Age: 46
Name: James, Age: 20
Name: Francesca, Age: 18
```

The *SortedList<TKey, TValue>* collection class

- The *SortedList<TKey, TValue>* class is very similar to the *Dictionary<TKey, TValue>* class in that you can use it to associate keys with values. The main difference is that the keys array is always sorted, data retrieval is often quicker (or at least as quick), and *SortedList<TKey, TValue>* class uses less memory.

- When you insert a key/value pair into a *SortedList<TKey, TValue>* collection, the key is inserted into the keys array at the correct index to keep the keys array sorted. The value is then inserted into the values array at the same index. The *SortedList<TKey, TValue>* class automatically ensures that keys and values maintain synchronization, even when you add and remove elements. This means that you can insert key/value pairs into a *SortedList<TKey, TValue>* in any sequence; they are always sorted based on the value of the keys.
- Like the *Dictionary<TKey, TValue>* class, a *SortedList<TKey, TValue>* collection cannot contain duplicate keys. When you use a *foreach* statement to iterate through a *SortedList<TKey, TValue>*, you receive back a *KeyValuePair<TKey, TValue>* item.

However, the *KeyValuePair<TKey, TValue>* items will be returned sorted by the *Key* property.

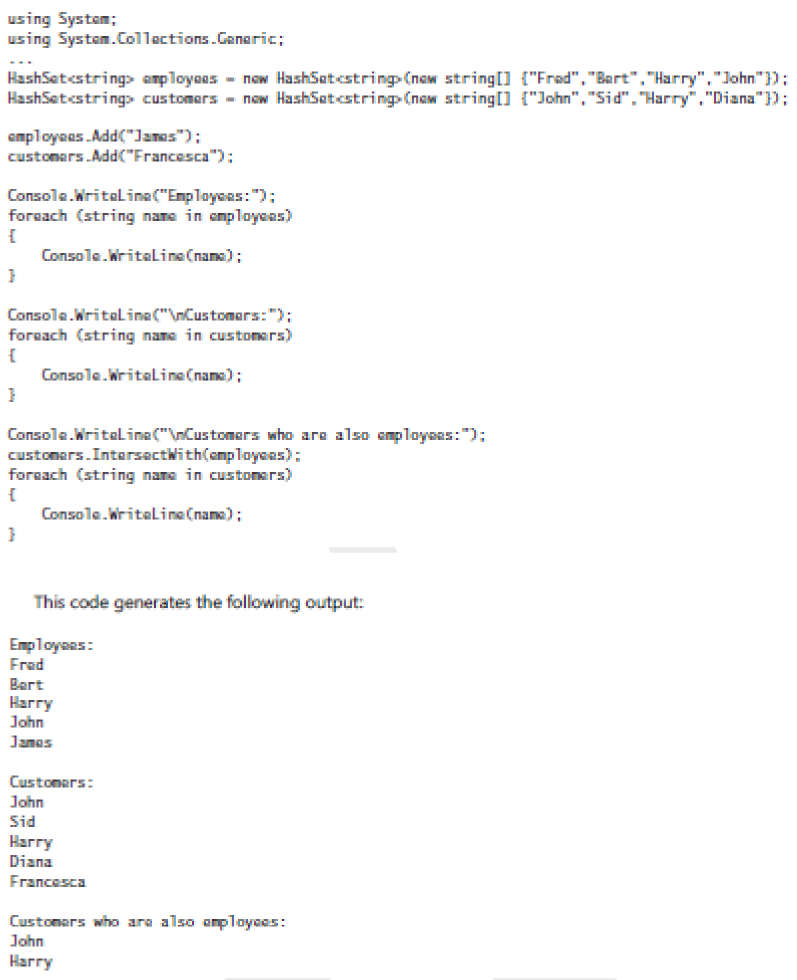
```
using System;
using System.Collections.Generic;
...
SortedList<string, int> ages = new SortedList<string, int>();

// fill the SortedList
ages.Add("John", 47);    // using the Add method
ages.Add("Diana", 46);
ages["James"] = 20;      // using array notation
ages["Francesca"] = 18;

// iterate using a foreach statement
// the iterator generates a KeyValuePair item
Console.WriteLine("The SortedList contains:");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine("Name: {0}, Age: {1}", name, age);
}
```

The output from this program is sorted alphabetically by the names of my family members:

```
The SortedList contains:
Name: Diana, Age: 46
Name: Francesca, Age: 18
Name: James, Age: 20
Name: John, Age: 47
```

The *HashSet<T>* collection class

- The *HashSet<T>* class is optimized for performing set operations such as determining set membership and generating the union and intersect of sets.
- We insert items into a *HashSet<T>* collection by using the *Add* method, and you delete items by using the *Remove* method. However, the real power of the *HashSet<T>* class is provided by the *IntersectWith*, *UnionWith*, and *ExceptWith* methods. These methods modify a *HashSet<T>* collection to generate a new set that either intersects with, has a union with, or does not contain the items in a specified *HashSet<T>* collection.
- We can also determine whether the data in one *HashSet<T>* collection is a superset or subset of another by using the *IsSubsetOf*, *IsSupersetOf*, *IsProperSubsetOf*, and *IsProperSupersetOf* methods. These methods return a Boolean value and are nondestructive.
- Internally, a *HashSet<T>* collection is held as a hash table, enabling fast lookup of items. However, a large *HashSet<T>* collection can require a significant amount of memory to operate quickly.



```
using System;
using System.Collections.Generic;
...
HashSet<string> employees = new HashSet<string>(new string[] {"Fred","Bert","Harry","John"});
HashSet<string> customers = new HashSet<string>(new string[] {"John","Sid","Harry","Diana"});

employees.Add("James");
customers.Add("Francesca");

Console.WriteLine("Employees:");
foreach (string name in employees)
{
    Console.WriteLine(name);
}

Console.WriteLine("\nCustomers:");
foreach (string name in customers)
{
    Console.WriteLine(name);
}

Console.WriteLine("\nCustomers who are also employees:");
customers.IntersectWith(employees);
foreach (string name in customers)
{
    Console.WriteLine(name);
}
```

This code generates the following output:

```
Employees:
Fred
Bert
Harry
John
James

Customers:
John
Sid
Harry
Diana
Francesca

Customers who are also employees:
John
Harry
```

Using collection initializers

- The examples in the preceding subsections have shown you how to add individual elements to a collection by using the method most appropriate to that collection (*Add* for a *List<T>* collection, *Enqueue* for a *Queue<T>* collection, *Push* for a *Stack<T>* collection, and so on). You can also initialize *some* collection types when you declare them, using a syntax similar to that supported by arrays.

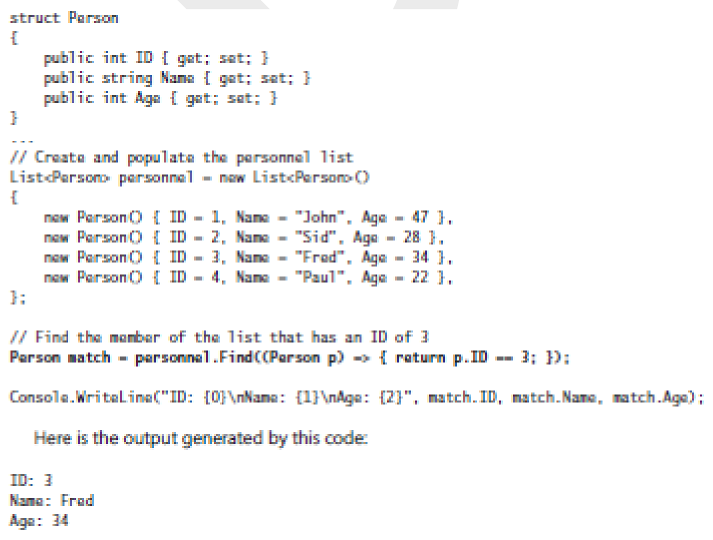
  List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};

- For more complex collections that take key/value pairs, such as the *Dictionary<TKey, TValue>* class, you can specify each key/value pair as an anonymous type in the initializer list, like this:

  Dictionary<string, int> ages = new Dictionary<string, int>(){{"John", 44}, {"Diana", 45}, {"James", 17}, {"Francesca", 15}};

The *Find* methods, predicates, and lambda expressions

- Using the dictionary-oriented collections (*Dictionary<TKey, TValue>*, *SortedDictionary<TKey*, *TValue>*, and *SortedList<TKey*, *TValue>*), you can quickly find a value by specifying the key to search for, and you can use array notation to access the value, as you have seen in earlier examples. Other collections that support nonkeyed random access, such as the *List<T>* and *LinkedList<T>* classes, do not support array notation but instead provide the *Find* method to locate an item.
- In the case of the *Find* method, as soon as the first match is found, the corresponding item is returned. Note that the *List<T>* and *LinkedList<T>* classes also support other methods such as *FindLast*, which returns the last matching object, and the *List<T>* class additionally provides the *FindAll* method, which returns a *List<T>* collection of all matching objects.
- The easiest way to specify the predicate is to use a *lambda expression*. A lambda expression is an expression that returns a method.
- A lambda expression contains two of these elements: a list of parameters and a method body. Lambda expressions do not define a method name, and the return type (if any) is inferred from the context in which the lambda expression is used.



```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
// Create and populate the personnel list
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 47 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};

// Find the member of the list that has an ID of 3
Person match = personnel.Find((Person p) => { return p.ID == 3; });

Console.WriteLine("ID: {0}\nName: {1}\nAge: {2}", match.ID, match.Name, match.Age);
```

Here is the output generated by this code:

```
ID: 3
Name: Fred
Age: 34
```

- In the call to the *Find* method, the argument *(Person p) => { return p.ID == 3; }* is a lambda expression that actually does the work. It has the following syntactic items:

  1) A list of parameters enclosed in parentheses. As with a regular method, if the method you are defining (as in the preceding example) takes no parameters, you must still provide the parentheses.

  2) The => operator, which indicates to the C# compiler that this is a lambda expression.

3) The body of the method. The example shown here is very simple, containing a single statement that returns a Boolean value indicating whether the item specified in the parameter matches the search criteria.

- A simplified form of the *Find* statement shown previously looks like this

```
Person match = personnel.Find(p => p.ID == 3);
```

Comparing arrays and collections

Here's a summary of the important differences between arrays and collections:

- An array instance has a fixed size and cannot grow or shrink. A collection can dynamically resize itself as required.
- An array can have more than one dimension. A collection is linear. However, the items in a collection can be collections themselves, so you can imitate a multidimensional array as a collection of collections.
- We store and retrieve an item in an array by using an index. Not all collections support this notion. For example, to store an item in a *List<T>* collection, you use the *Add* or *Insert* methods, and to retrieve an item, you use the *Find* method.
- Many of the collection classes provide a *ToArray* method that creates and populates an array containing the items in the collection. The items are copied to the array and are not removed from the collection. Additionally, these collections provide constructors that can populate a collection directly from an array.

MODULE 5[CHAPTER 1]

Enumerating collections

Let's consider foreach statement before using collections, an example of using the *foreach* statement to list the items in a simple array. The code looks like this:

```
int[] pins = { 9, 3, 7, 2 };
 foreach (int pin in pins)
{
Console.WriteLine(pin);
}
```

- The *foreach* construct provides an elegant mechanism that greatly simplifies the code we need to write, but it can be exercised only under certain circumstances—we can use *foreach* only to step through an *enumerable* collection.

So, **what is an enumerable collection**?

- The answer is that it is a collection that implements the *System.Collections.IEnumerable* interface. The *IEnumerable* interface contains a single method called *GetEnumerator*:

```
IEnumerator GetEnumerator();
```

- The *GetEnumerator* method return an enumerator object that implements the *System.Collections.IEnumerator* interface. The enumerator object is used for stepping through (enumerating) the elements of the collection. The *IEnumerator* interface specifies the following property and methods:

```
object Current { get; }
bool MoveNext();
void Reset();
```

- consider an enumerator as a pointer indicating elements in a list. Initially, the pointer points *before* the first element. We call the *MoveNext* method to move the pointer down to the next (first) item in the list; the *MoveNext* method should return *true* if there actually is another item and *false* if there isn't.
- we use the *Current* property to access the item currently pointed to, and we use the *Reset* method to return the pointer back to *before* the first item in the list. By creating an enumerator by using the *GetEnumerator* method of a collection and repeatedly calling the *MoveNext* method and retrieving the value of the *Current* property by using the enumerator, we can move forward through the elements of a collection one item at a time.

- This is exactly what the *foreach* statement does. So, if we want to create our own enumerable collection class, we must implement the *IEnumerable* interface in our collection class and also provide an implementation of the *IEnumerator* interface to be returned by the *GetEnumerator* method of the collection class

Implementing an enumerator by using an iterator

The process of making a collection enumerable can become complex and potentially error-prone. To make life easier, C# provides iterators that can automate much of this process.

- An ***iterator* is** a block of code that yields an ordered sequence of values. An iterator is not actually a member of an enumerable class; rather, it specifies the sequence that an enumerator should use for returning its values. In other words, an iterator is just a description of the enumeration sequence that the C# compiler can use for creating its own enumerator

A simple iterator

- The following *BasicCollection<T>* class illustrates the principles of implementing an iterator. The class uses a *List<T>* object for holding data and provides the *FillList* method for populating this list. Notice also that the *BasicCollection<T>* class implements the *IEnumerable<T>* interface. The *GetEnumerator* method is implemented by using an iterator:

```
using System;
using System.Collections.Generic;
 using System.Collections;
class BasicCollection<T> : IEnumerable<T>
{
private List<T> data = new List<T>();
 public void FillList(params T [] items)
{
foreach (var datum in items)
{
data.Add(datum);
} }
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
foreach (var datum in data)
{
yield return datum; } }
 IEnumerator IEnumerable.GetEnumerator()
{ // Not implemented in this example throw new NotImplementedException(); } }
```

- The *GetEnumerator* method it loops through the items in the *data* array, returning each item in turn. The key point is the use of the *yield* keyword. The *yield* keyword indicates the value that should be returned by each iteration. we can think of the *yield* statement as calling a temporary halt to the method, passing back a value to the caller.
- When the caller needs the next value, the *GetEnumerator* method continues at the point at which it left off, looping around and then yielding the next value. Eventually, the data is exhausted, the loop finishes, and the *GetEnumerator* method terminates. At this point, the iteration is complete.
- The code in the *GetEnumerator* method defines an *iterator*. The compiler uses this code to generate an implementation of the *IEnumerator<T>* class containing a *Current* method and a *MoveNext* method.
- we can invoke the enumerator generated by the iterator in the usual manner, as shown in the following block of code, which displays the words in the first line of the poem “Jabberwocky” by Lewis Carroll:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc)
{ Console.WriteLine(word); }
```

This code simply outputs the contents of the *bc* object in this order:

Twas, brillig, and, the, slithy, toves

- If we want to provide alternative iteration mechanisms presenting the data in a different sequence, we can implement additional properties that implement the *IEnumerable* interface and that use an iterator for returning data. For example, the *Reverse* property of the *BasicCollection<T>* class, shown here, emits the data in the list in reverse order:

```
class BasicCollection<T> : IEnumerable<T>
{
...
 public IEnumerable<T> Reverse
{
get {
for (int i = data.Count - 1; i >= 0; i--)
{
yield  return data[i]; } }
} }
```

You can invoke this property as follows:

```
BasicCollection<string> bc = new BasicCollection<string>(); bc.FillList("Twas", "brillig",
"and", "the", "slithy", "toves");

foreach (string word in bc.Reverse)

{ Console.WriteLine(word); }
```

This code outputs the contents of the *bc* object in reverse order:

```
toves, slithy, the, and, brillig, Twas
```

MODULE 5[CHAPTER 2]

Querying in-memory data by using query expressions

C# provides for querying data we have seen that we can define structures and classes for modeling data and that you can use collections and arrays for temporarily storing data in memory. However, to perform common tasks such as searching for items in a collection that match a specific set of criteria For example, if we have a collection of *Customer* objects, how do we find all customers that are located in London, or how can we find out which town has the most customers that have procured your services?

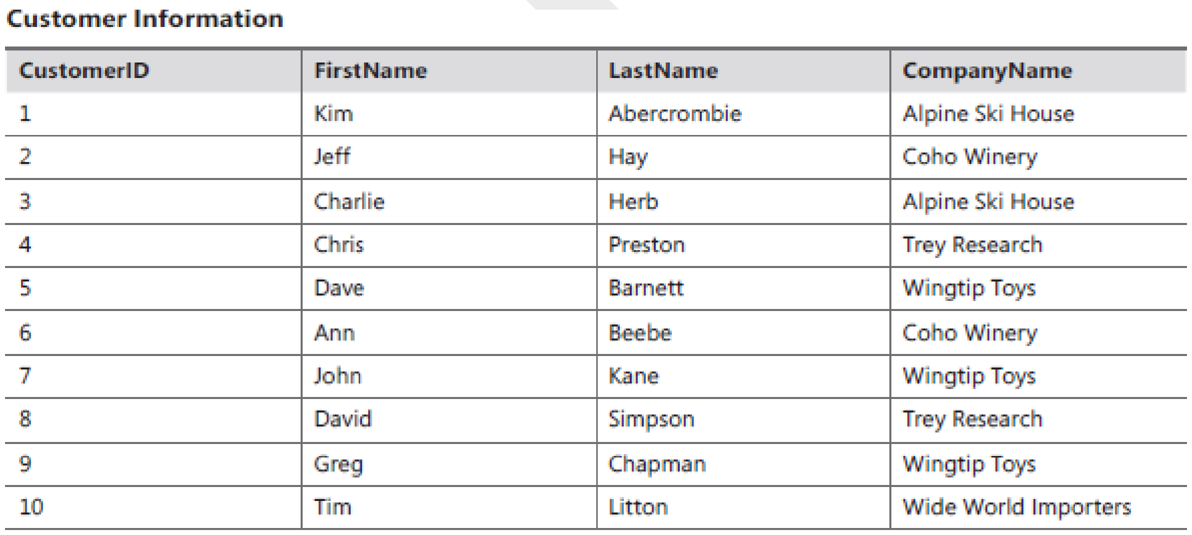
Solution :we can write our own code to iterate through a collection and examine the fields in each object, but these types of tasks occur so often that the designers of C# decided to include features in the language to minimize the amount of code we need to write. so will learn how to use these advanced C# language features to query and manipulate data.

What is Language-Integrated Query?

The features that abstract the mechanism that an application uses to query data from application code itself. These features are called Language-Integrated Query, or LINQ. LINQ provides syntax and semantics very reminiscent of SQL, and with many of the same advantages. We can change the underlying structure of the data being queried without needing to change the code that actually performs the queries.

Using LINQ in a C# application

To use the C# features that support LINQ is to work through some simple examples based on the following sets of customer and address information:



Customer Information

CustomerID	FirstName	LastName	CompanyName
1	Kim	Abercrombie	Alpine Ski House
2	Jeff	Hay	Coho Winery
3	Charlie	Herb	Alpine Ski House
4	Chris	Preston	Trey Research
5	Dave	Barnett	Wingtip Toys
6	Ann	Beebe	Coho Winery
7	John	Kane	Wingtip Toys
8	David	Simpson	Trey Research
9	Greg	Chapman	Wingtip Toys
10	Tim	Litton	Wide World Importers

Address Information

CompanyName	City	Country
Alpine Ski House	Berne	Switzerland
Coho Winery	San Francisco	United States
Trey Research	New York	United States
Wingtip Toys	London	United Kingdom
Wide World Importers	Tetbury	United Kingdom

- LINQ requires the data to be stored in a data structure that implements the *IEnumerable* or *IEnumerable<T>* interface.It does not matter what structure we use (an array, a *HashSet<T>*, a *Queue<T>*, or any of the other collection types, or even one that you define yourself) as long as it is enumerable.
- However assume that the customer and address information is held in the *customers* and *addresses* arrays shown in the following code example

```
var customers = new[] {
    new { CustomerID = 1, FirstName = "Kim", LastName = "Abercrombie",
          CompanyName = "Alpine Ski House" },
    new { CustomerID = 2, FirstName = "Jeff", LastName = "Hay",
          CompanyName = "Coho Winery" },
    new { CustomerID = 3, FirstName = "Charlie", LastName = "Herb",
          CompanyName = "Alpine Ski House" },
    new { CustomerID = 4, FirstName = "Chris", LastName = "Preston",
          CompanyName = "Trey Research" },
    new { CustomerID = 5, FirstName = "Dave", LastName = "Barnett",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 6, FirstName = "Ann", LastName = "Beebe",
          CompanyName = "Coho Winery" },
    new { CustomerID = 7, FirstName = "John", LastName = "Kane",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 8, FirstName = "David", LastName = "Simpson",
          CompanyName = "Trey Research" },
    new { CustomerID = 9, FirstName = "Greg", LastName = "Chapman",
          CompanyName = "Wingtip Toys" },
    new { CustomerID = 10, FirstName = "Tim", LastName = "Litton",
          CompanyName = "Wide World Importers" }
};
```

```
var addresses = new[] {
    new { CompanyName = "Alpine Ski House", City = "Berne",
          Country = "Switzerland"},
    new { CompanyName = "Coho Winery", City = "San Francisco",
          Country = "United States"},
    new { CompanyName = "Trey Research", City = "New York",
          Country = "United States"},
    new { CompanyName = "Wingtip Toys", City = "London",
          Country = "United Kingdom"},
    new { CompanyName = "Wide World Importers", City = "Tetbury",
          Country = "United Kingdom"}
};
```

Selecting data

Suppose that we want to display a list consisting of the first name of each customer in the *customers* array. we can achieve this task with the following code:

```
IEnumerable<string> customerFirstNames = customers.Select(cust => cust.FirstName);

foreach (string name in customerFirstNames)

{ Console.WriteLine(name);}
```

Using the *Select* method, we can retrieve specific data from the array—in this case, just the value in the *FirstName* field of each item in the array. The parameter to the *Select* method is actually another method that takes a row from the *customers* array and returns the selected data from that row. There are three important things that you need to understand at this point:

- The variable *cust* is the parameter passed in to the method. we can think of *cust* as an alias for each row in the *customers* array. The compiler deduces this from the fact that you are calling the *Select* method on the *customers* array. We can use any legal C# identifier in place of *cust*.
- The *Select* method does not actually retrieve the data at this time; it simply returns an enumerable object that will fetch the data identified by the *Select* method when you iterate over it later.
- The *Select* method is not actually a method of the *Array* type. It is an extension method of the *Enumerable* class. The *Enumerable* class is located in the *System.Linq* namespace and provides a substantial set of static methods for querying objects that implement the generic *IEnumerable<T>* interface.

The preceding example uses the *Select* method of the *customers* array to generate an *IEnumerable<string>* object named *customerFirstNames*. The *foreach* statement iterates through this collection of strings, printing out the first name of each customer in the following sequence:

```
Kim
Jeff
Charlie
Chris
Dave
Ann
John
David
Greg
Tim
```

The important point to understand is that the *Select* method returns an enumerable collection based on a single type. If we want the enumerator to return multiple items of data, such as the first and last name of each customer, we have at least two options:

- You can concatenate the first and last names together into a single string in the *Select* method, like this:

```
IEnumerable<string> customerNames =
    customers.Select(cust => String.Format("{0} {1}", cust.FirstName, cust.LastName));
```

- You can define a new type that wraps the first and last names, and use the *Select* method to construct instances of this type, like this:

```
class FullName
{
    public string FirstName{ get; set; }
    public string LastName{ get; set; }
}
...
IEnumerable<FullName> customerNames =
    customers.Select(cust => new FullName
    {
        FirstName = cust.FirstName,
        LastName = cust.LastName
    } );
```

Filtering data

With the *Select* method, we can specify, or *project*, the fields that we want to include in the enumerable collection. However, we might also want to restrict the rows that the enumerable collection contains. For example, suppose we want to list the names of all companies in the *addresses* array that are located in the United States only. To do this, we can use the *Where* method, as follows:

```
IEnumerable<string> usCompanies =
    addresses.Where(addr => String.Equals(addr.Country, "United States"))
             .Select(usComp => usComp.CompanyName);

foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

The *Where* method is similar to *Select*. It expects a parameter that defines a method that filters the data according to whatever criteria we specify. The *foreach* statement that iterates through this collection displays the following companies:

Coho Winery
Trey Research

Ordering, grouping, and aggregating data

To retrieve data in a particular order, you can use the *OrderBy* method. Like the *Select* and *Where* methods, *OrderBy* expects a method as its argument. This method identifies the expressions that you want to use to sort the data. For example, you can display the name of each company in the *addresses* array in ascending order, like this:

```
IEnumerable<string> companyNames =
    addresses.OrderBy(addr => addr.CompanyName).Select(comp => comp.CompanyName);

foreach (string name in companyNames)
{
    Console.WriteLine(name);
}
```

This block of code displays the companies in the addresses table in alphabetical order.

```
Alpine Ski House
Coho Winery
Trey Research
Wide World Importers
Wingtip Toys
```

- If you want to enumerate the data in descending order, you can use the *OrderByDescending* method, instead. If you want to order by more than one key value, you can use the *ThenBy* or *ThenByDescending* method after *OrderBy* or *OrderByDescending*.

To group data according to common values in one or more fields, you can use the *GroupBy* method. The following example shows how to group the companies in the *addresses* array by country:

```
var companiesGroupedByCountry =
    addresses.GroupBy(addrs => addrs.Country);

foreach (var companiesPerCountry in companiesGroupedByCountry)
{
    Console.WriteLine("Country: {0}\t{1} companies",
            companiesPerCountry.Key, companiesPerCountry.Count());
    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine("\t{0}", companies.CompanyName);
    }
}
```

The *GroupBy* method expects a method that specifies the fields by which to group the data. The output generated by the example code looks like this:

```
Country: Switzerland 1 companies
Alpine Ski House
Country: United States 2 companies
Coho Winery
Trey Research
Country: United Kingdom  2 companies
Wingtip Toys
Wide World Importers
```

- You can use many of the summary methods such as *Count*, *Max*, and *Min* directly over the results of the *Select* method

```
int numberOfCompanies = addresses.Select(addr => addr.CompanyName).Count();
Console.WriteLine("Number of companies: {0}", numberOfCompanies);
```

- Notice that the result of these methods is a single scalar value rather than an enumerable collection. The output from the preceding block of code looks like this:

  Number of companies: 5

- In fact, there are only three different countries in the *addresses* array—it just so happens that United States and United Kingdom both occur twice. You can eliminate duplicates from the calculation by using the *Distinct* method, like this:

```
int numberOfCountries =
    addresses.Select(addr => addr.Country).Distinct().Count();
Console.WriteLine("Number of countries: {0}", numberOfCountries);
```

The *Console.WriteLine* statement now outputs the expected result:

Number of countries: 3

Joining data

Just like SQL, LINQ gives you the ability to join multiple sets of data together over one or more common key fields. The following example shows how to display the first and last names of each customer, together with the name of the country where the customer is located

```
var companiesAndCustomers = customers
  .Select(c => new { c.FirstName, c.LastName, c.CompanyName })
  .Join(addresses, custs => custs.CompanyName, addrs => addrs.CompanyName,
  (custs, addrs) => new {custs.FirstName, custs.LastName, addrs.Country });
```

```
foreach (var row in companiesAndCustomers)
{
    Console.WriteLine(row);
}
```

- The customers' first and last names are available in the *customers* array, but the country for each company that customers work for is stored in the *addresses* array. The common key between the *customers* array and the *addresses* array is the company name.
- The *Select* method specifies the fields of interest in the *customers* array (*FirstName* and *LastName*), together with the field containing the common key (*CompanyName*). You use the *Join* method to join the data identified by the *Select* method with another enumerable collection. **The parameters to the *Join* method are as follows:**

- The enumerable collection with which to join
- A method that identifies the common key fields from the data identified by the *Select* method

- A method that identifies the common key fields on which to join the selected data
- A method that specifies the columns you require in the enumerable result set returned by the *Join* method

The code that outputs the data from the *companiesAndCustomers* collection displays the following information:

```
{ FirstName = Kim, LastName = Abercrombie, Country = Switzerland }
{ FirstName = Jeff, LastName = Hay, Country = United States }
{ FirstName = Charlie, LastName = Herb, Country = Switzerland }
{ FirstName = Chris, LastName = Preston, Country = United States }
{ FirstName = Dave, LastName = Barnett, Country = United Kingdom }
{ FirstName = Ann, LastName = Beebe, Country = United States }
{ FirstName = John, LastName = Kane, Country = United Kingdom }
{ FirstName = David, LastName = Simpson, Country = United States }
{ FirstName = Greg, LastName = Chapman, Country = United Kingdom }
{ FirstName = Tim, LastName = Litton, Country = United Kingdom }
```

Using query operators

- The previous examples shown you many of the features available for querying in-memory data by using the extension methods for the *Enumerable* class defined in the *System.Linq* namespace. The syntax makes use of several advanced C# language features, and the resultant code can sometimes be quite hard to understand and maintain.
- To relieve you of some of this burden, the designers of C# added query operators to the language with which you can employ LINQ features by using a syntax more akin to SQL
- You can rephrase previous retrieve first name query can be replaced using the *from* and *select* query operators, like this:

```
var customerFirstNames = from cust in customers
                         select cust.FirstName;
```

- Continuing in the same vein, to retrieve the first and last names for each customer, you can use the following statement

```
var customerNames = from cust in customers
                    select new { cust.FirstName, cust.LastName };
```

- You use the *where* operator to filter data. The following example shows how to return the names of the companies based in the United States from the *addresses* array:

```
var usCompanies = from a in addresses
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

- To order data, use the *orderby* operator, like this:

```
var companyNames = from a in addresses
                   orderby a.CompanyName
                   select a.CompanyName;
```

- You can group data by using the *group* operator in the following manner

```
var companiesGroupedByCountry = from a in addresses
                                group a by a.Country;
```

- You can invoke the summary functions such as *Count* over the collection returned by an enumerable collection, like this:

```
int numberOfCompanies = (from a in addresses
                         select a.CompanyName).Count();
```

- If you want to ignore duplicate values, use the *Distinct* method:

```
int numberOfCountries = (from a in addresses
                         select a.Country).Distinct().Count();
```

- You can use the *join* operator to combine two collections across a common key. The following example shows the query returning customers and addresses over the *CompanyName* column in each collection, this time rephrased using the *join* operator

```
var countriesAndCustomers = from a in addresses
                            join c in customers
                            on a.CompanyName equals c.CompanyName
                            select new { c.FirstName, c.LastName, a.Country };
```

MODULE 5[CHAPTER 3]

Operator overloading

Understanding operators

- You use operators to combine operands together into expressions. Each operator has its own semantics, dependent on the type with which it works. For example, the + operator means "add" when you use it with numeric types, or it means "concatenate" when you use it with strings.
- Each operator has a *precedence*. For example, the * operator has a higher precedence than the + operator. This means that the expression *a + b * c* is the same as *a + (b * c)*.
- Each operator also has an *associativity* to define whether the operator evaluates from left to right or from right to left. For example, the = operator is right-associative (it evaluates from right to left), so *a = b = c* is the same as *a = (b = c)*.
- A *unary operator* is an operator that has just one operand. For example, the increment operator (++) is a unary operator.
- A *binary operator* is an operator that has two operands. For example, the multiplication operator (*) is a binary operator.

Operator constraints

- You cannot change the precedence and associativity of an operator. The precedence and associativity are based on the operator symbol (for example, +) and not on the type (for example, *int*) on which the operator symbol is being used. Hence, the expression *a + b * c* is always the same as *a + (b * c)*, regardless of the types of *a*, *b*, and *c*.
- You cannot change the multiplicity (the number of operands) of an operator. For example, * (the symbol for multiplication) is a binary operator. If you declare a * operator for your own type, it must be a binary operator.
- You cannot invent new operator symbols. For example, you can't create a new operator symbol, such as ** for raising one number to the power of another number. You'd have to create a method to do that.
- You can't change the meaning of operators when applied to built-in types. For example, the expression *1 + 2* has a predefined meaning, and you're not allowed to override this meaning. If you could do this, things would be too complicated.
- There are some operator symbols that you can't overload. For example, you can't overload the dot (.) operator, which indicates access to a class member. Again, if you could do this, it would lead to unnecessary complexity.

Overloaded operators

- To define your own operator behavior, you must overload a selected operator. You use method-like syntax with a return type and parameters, but the name of the method is the keyword operator together with the *operator* symbol you are declaring.
- For example, the following code shows a user-defined structure named *Hour* that defines a binary + operator to add together two instances of *Hour*:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }
    ...
    private int value;
}
```

Notice the following:

- The operator is *public*. All operators *must* be public.
- The operator is *static*. All operators *must* be static. Operators are never polymorphic and cannot use the *virtual*, *abstract*, *override*, or *sealed* modifiers.
- A binary operator (such as the + operator shown earlier) has two explicit arguments, and a unary operator has one explicit argument.

When you use the + operator on two expressions of type *Hour*, the C# compiler automatically converts your code to a call to your *operator* + method. The C# compiler transforms this code

```
Hour Example(Hour a, Hour b)
{
    return a + b;
}
```

to this:

```
Hour Example(Hour a, Hour b)
{
    return Hour.operator +(a,b); // pseudocode
}
```

Creating symmetric operators

- In the preceding section, you saw how to declare a binary + operator to add together two instances of type *Hour*. The *Hour* structure also has a constructor that creates an *Hour* from an *int*. This means that you can add together an *Hour* and an *int*; you just have to first use the *Hour* constructor to convert the *int* to an *Hour*, as in the following example:

```
Hour a = ...;
int b = ...;
Hour sum = a + new Hour(b);
```

- This is certainly valid code, but it is not as clear or concise as adding together an *Hour* and an *int* directly, like this:

```
Hour a = ...;
int b = ...;
Hour sum = a + b;
```

- To make the expression *(a + b)* valid, you must specify what it means to add together an *Hour* (*a*, on the left) and an *int* (*b*, on the right). In other words, you must declare a binary + operator whose first parameter is an *Hour* and whose second parameter is an *int*. The following code shows the recommended approach:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }

    public static Hour operator +(Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }
    ...
    private int value;
}
```

- This *operator+* declares how to add together an *Hour* as the left operand and an *int* as the right operand. It does not declare how to add together an *int* as the left operand and an *Hour* as the right operand:

```
int a = ...;
Hour b = ...;
Hour sum = a + b; // compile-time error
```

- This is counterintuitive. If you can write the expression *a + b*, you expect to also be able to write *b + a*. Therefore, you should provide another overload of *operator+*:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator +(Hour lhs, int rhs)
    {
        return lhs + new Hour(rhs);
    }

    public static Hour operator +(int lhs, Hour rhs)
    {
         return new Hour(lhs) + rhs;
    }
    ...
    private int value;
}
```

Understanding compound assignment evaluation

- A compound assignment operator (such as +=) is always evaluated in terms of its associated simple operator (such as +). In other words, the statement

  a += b;

  is automatically evaluated like this:

  a = a + b;

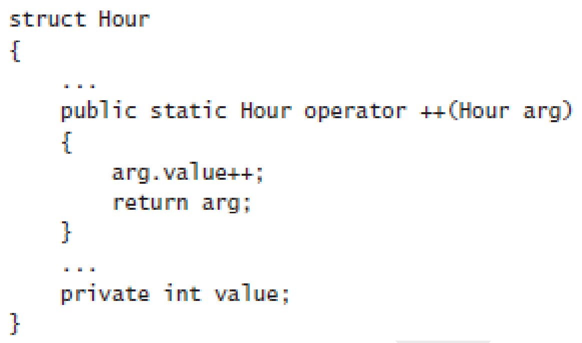
- In general, the expression *a @= b* (where @ represents any valid operator) is always evaluated as *a = a @ b*. If you have overloaded the appropriate simple operator, the overloaded version is automatically called when you use its associated compound assignment operator, as is shown in the following example:

```
Hour a = ...;
int b = ...;
a += a; // same as a = a + a
a += b; // same as a = a + b
```

- The first compound assignment expression *(a += a)* is valid because *a* is of type *Hour*, and the *Hour* type declares a binary *operator+* whose parameters are both *Hour*. Similarly, the second compound assignment expression *(a += b)* is also valid because *a* is of type *Hour* and *b* is of type *int*. The *Hour* type also declares a binary *operator+* whose first parameter is an *Hour* and whose second parameter is an *int*.
- Be aware, however, that you cannot write the expression *b* += *a* because that's the same as *b* = *b* + *a*. Although the addition is valid, the assignment is not, because there is no way to assign an *Hour* to the built-in *int* type

Declaring increment and decrement operators

- With C#, you can declare your own version of the increment (++) and decrement (– –) operators. The usual rules apply when declaring these operators: they must be public, they must be static, and they must be unary (they can take only a single parameter). Here is the increment operator for the *Hour* structure:



```
struct Hour
{
    ...
    public static Hour operator ++(Hour arg)
    {
        arg.value++;
        return arg;
    }
    ...
    private int value;
}
```

- The increment and decrement operators are unique in that they can be used in prefix and postfix forms. C# cleverly uses the same single operator for both the prefix and postfix versions. The result of a postfix expression is the value of the operand *before* the expression takes place. In other words, the compiler effectively converts the code

```
Hour now = new Hour(9);
Hour postfix = now++;
```

to this:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocode, not valid C#
```

- The result of a prefix expression is the return value of the operator, so the C# compiler effectively transforms the code

```
Hour now = new Hour(9);
Hour prefix = ++now;
```

to this:

```
Hour now = new Hour(9);
now = Hour.operator ++(now); // pseudocode, not valid C#
Hour prefix = now;
```

- This equivalence means that the return type of the increment and decrement operators must be the same as the parameter type.

Comparing operators in structures and classes

- Be aware that the implementation of the increment operator in the *Hour* structure works only because *Hour* is a structure. If you change *Hour* into a class but leave the implementation of its increment operator unchanged, you will find that the postfix translation won't give the correct answer.
- you can see in the following example why the operators for the *Hour* class no longer function as expected:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocode, not valid C#
```

- If *Hour* is a class, the assignment statement *postfix* = now makes the variable *postfix* refer to the same object as *now*. Updating *now* automatically updates *postfix*! If *Hour* is a structure, the assignment statement makes a copy of *now* in *postfix*, and any changes to *now* leave *postfix* unchanged.

The correct implementation of the increment operator when *Hour* is a class is as follows:

```
class Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static Hour operator ++(Hour arg)
    {
        return new Hour(arg.value + 1);
    }
    ...
    private int value;
}
```

- Notice that *operator* ++ now creates a new object based on the data in the original. The data in the new object is incremented, but the data in the original is left unchanged.

Defining operator pairs

- Some operators naturally come in pairs. For example, if you can compare two *Hour* values by using the *!*= operator, you would expect to be able to also compare two *Hour* values by using the == operator.

  Here are the == and *!*= operators for the *Hour* structure:

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }
    ...
    public static bool operator ==(Hour lhs, Hour rhs)
    {
        return lhs.value == rhs.value;
    }

    public static bool operator !=(Hour lhs, Hour rhs)
    {
        return lhs.value != rhs.value;
    }
    ...
    private int value;
}
```

- This neither-or-both rule also applies to the < and > operators and the <= and >= operators.

Understanding conversion operators

- Sometimes, you need to convert an expression of one type to another. For example, the following method is declared with a single *double* parameter:

```
class Example
{
    public static void MyDoubleMethod(double parameter)
    {
        ...
    }
}
```

- You might reasonably expect that only values of type *double* could be used as arguments when calling *MyDoubleMethod*, but this is not so. The C# compiler also allows *MyDoubleMethod* to be called with an argument of some other type, but only if the value of the argument can be converted to a *double*.

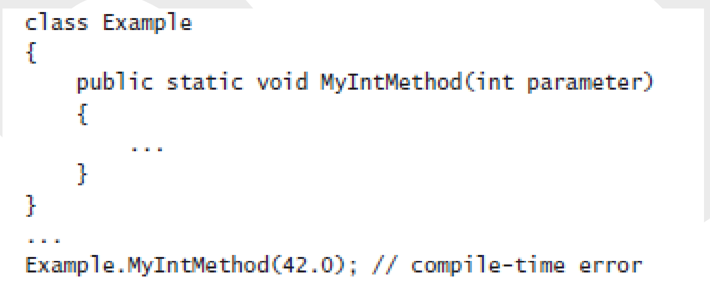
- For example, if you provide an *int* argument, the compiler generates code that converts the value of the argument to a *double* when the method is called.

Providing built-in conversions

- The built-in types have some built-in conversions. For example, as mentioned previously, an *int* can be implicitly converted to a *double*. An implicit cThe built-in types have some built-in conversions. For example, an *int* can be implicitly converted to a *double*. An implicit conversion requires no special syntax and never throws an exception.onversion requires no special syntax and never throws an exception.

  Example.MyDoubleMethod(42); // implicit int-to-double conversion

- An implicit conversion is sometimes called a ***widening conversion*** because the result is *wider* than the original value—it contains at least as much information as the original value, and nothing is lost. In the case of *int* and *double*, the range of *double* is greater than that of *int*, and all *int* values have an equivalent *double* value. However, the converse is not true, and a *double* value cannot be implicitly converted to an *int*:



```
class Example
{
    public static void MyIntMethod(int parameter)
    {
        ...
    }
}
...
Example.MyIntMethod(42.0); // compile-time error
```

- When you convert a *double* to an *int*, you run the risk of losing information, so the conversion will not be performed automatically. (Consider what would happen if the argument to *MyIntMethod* were 42.5: How should this be converted?) A *double* can be converted to an *int*, but the conversion requires an explicit notation (a cast):

  Example.MyIntMethod((int)42.0);

- An explicit conversion is sometimes called a ***narrowing conversion*** because the result is *narrower* than the original value (that is, it can contain less information) and can throw an *OverflowException* exception if the resulting value is out of the range of the target type.

Implementing user-defined conversion operators

- The syntax for declaring a user-defined conversion operator has some similarities to that for declaring an overloaded operator, but also some important differences. Here's a conversion operator that allows an *Hour* object to be implicitly converted to an *int*:

```
struct Hour
{
    ...
    public static implicit operator int (Hour from)
    {
        return from.value;
    }

    private int value;
}
```

- A conversion operator must be *public* and it must also be *static*. The type from which you are converting is declared as the parameter (in this case, *Hour*), and the type to which you are converting is declared as the type name after the keyword *operator* (in this case, *int*). There is no return type specified before the keyword *operator*
- When declaring your own conversion operators, you must specify whether they are implicit conversion operators or explicit conversion operators. You do this by using the *implicit* and *explicit* keywords. For example, the *Hour* to *int* conversion operator mentioned earlier is implicit, meaning that the C# compiler can use it without requiring a cast.

```
class Example
{
    public static void MyOtherMethod(int parameter) { ... }
    public static void Main()
    {
        Hour lunch = new Hour(12);
        Example.MyOtherMethod(lunch); // implicit Hour to int conversion
    }
}
```

- If the conversion operator had been declared *explicit*, the preceding example would not have compiled, because an explicit conversion operator requires a cast.

Example.MyOtherMethod((int)lunch); // explicit Hour to int conversion

MODULE 5[CHAPTER 4]

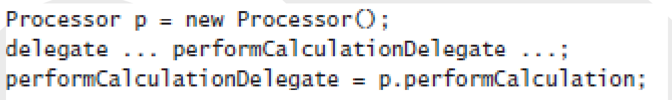
Decoupling application logic and handling events

Understanding delegates

A delegate is a reference to a method. It is a very simple concept with extraordinarily powerful implications.

Note : Delegates are so named because they “delegate” processing to the referenced method when they are invoked

- A *delegate* is an object that refers to a method. You can assign a reference to a method to a delegate in much the same way that you can assign an *int* value to an *int* variable. The example creates a delegate named *performCalculationDelegate* that references the *performCalculation* method of the *Processor* object.



```
Processor p = new Processor();
delegate ... performCalculationDelegate ...;
performCalculationDelegate = p.performCalculation;
```

- It is important to understand that the statement that assigns the method reference to the delegate does not run the method at this point; there are no parentheses after the method name, and you do not specify any parameters This is just an assignment statement.
- Having stored a reference to the *performCalculation* method of the *Processor* object in the delegate, the application can subsequently invoke the method through the delegate, like this:

performCalculationDelegate();

Examples of delegates in the .NET Framework class library

The Microsoft .NET Framework class library makes extensive use of delegates for many of its types, here we have some examples

The following code use the *Find* method:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 47 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};
...
// Find the member of the list that has an ID of 3
Person match = personnel.Find(p => p.ID == 3);
```

- Other examples of methods exposed by the *List<T>* class that use delegates to perform their operations include *Average*, *Max*, *Min*, *Count*, and *Sum*.
- In the following examples, the *Average* method is used to calculate the average age of items in the personnel collection (the *Func<T>* delegate simply returns the value in the *Age* field of each item in the collection), the *Max* method is used to determine the item with the highest ID, and the *Count* method calculates how many items have an *Age* between 30 and 39 inclusive.

```
double averageAge = personnel.Average(p => p.Age);
Console.WriteLine("Average age is {0}", averageAge);
...
int id = personnel.Max(p => p.ID);
Console.WriteLine("Person with highest ID is {0}", id);
...
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
Console.WriteLine("Number of personnel in their thirties is {0}", thirties);
```

This code generates the following output:

```
Average age is 32.75
Person with highest ID is 4
Number of personnel in their thirties is 1
```

Example: The automated factory scenario

Consider writing the control systems for an automated factory. The factory contains a large number of different machines, each performing distinct tasks in the production of the articles manufactured by the factory—shaping and folding metal sheets, welding sheets together, painting sheets, and so on.

Each machine has its own unique computer-controlled process (and functions) for shutting down safely, as summarized here:

```
StopFolding();    // Folding and shaping machine
FinishWelding();  // Welding machine
PaintOff();       // Painting machine
```

Implementing the factory control system without using delegates

A simple approach to implementing the shutdown functionality in the control program is as follows:

```
class Controller
{
    // Fields representing the different machines
    private FoldingMachine folder;
    private WeldingMachine welder;

            private PaintingMachine painter;
            ...
            public void ShutDown()
            {
                folder.StopFolding();
                welder.FinishWelding();
                painter.PaintOff();
            }
            ...
        }
```

- Although this approach works, it is not very extensible or flexible. If the factory buys a new machine, you must modify this code; the *Controller* class and code for managing the machines is tightly coupled.

Implementing the factory by using a delegate

- Although the names of each method are different, they all have the same "shape": they take no parameters, and they do not return a value. The general format of each method, therefore, is this:

  void *methodName();*

- This is where a delegate can be useful. You can use a delegate that matches this shape to refer to any of the machinery shutdown methods. You declare a delegate like this:

  delegate void stopMachineryDelegate();

Note the following points:

- You use the *delegate* keyword.
- You specify the return type (*void* in this example), a name for the delegate (*stopMachinery Delegate*), and any parameters (there are none in this case).

- After you have declared the delegate, you can create an instance and make it refer to a matching method by using the += compound assignment operator. We can do this in the constructor of the controller class like this:

```
class Controller
{
    delegate void stopMachineryDelegate();       // the delegate type
    private stopMachineryDelegate stopMachinery; // an instance of the delegate
    ...
    public Controller()
    {
        this.stopMachinery += folder.StopFolding;
    }
    ...
}
```

- It is safe to use the += operator on an uninitialized delegate. It will be initialized automatically. Alternatively, you can use the *new* keyword to initialize a delegate explicitly with a single specific method, like this:

    this.stopMachinery = new stopMachineryDelegate(folder.StopFolding);

  You can call the method by invoking the delegate, like this:

```
public void ShutDown()
{
    this.stopMachinery();
    ...
}
```

- An important advantage of using a delegate is that it can refer to more than one method at the same time. You simply use the += operator to add methods to the delegate, like this:

```
public Controller()
{
    this.stopMachinery += folder.StopFolding;
    this.stopMachinery += welder.FinishWelding;
    this.stopMachinery += painter.PaintOff;
}
```

Invoking *this.stopMachinery()* in the *Shutdown* method of the *Controller* class automatically calls each of the methods in turn. The *Shutdown* method does not need to know how many machines there are or what the method names are.

- You can remove a method from a delegate by using the –= compound assignment operator, as demonstrated here:

this.stopMachinery -= folder.StopFolding;

- The current scheme adds the machine methods to the delegate in the *Controller* constructor. To make the *Controller* class totally independent of the various machines, you need to make *stopMachineryDelegate* type public and supply a means of enabling classes outside *Controller* to add methods to the delegate. You have several options:

- Make the *stopMachinery* delegate variable, public:
  public stopMachineryDelegate stopMachinery ;

- Keep the *stopMachinery* delegate variable private, but create a read/write property to provide access to it:

```
private delegate void stopMachineryDelegate();
...
public stopMachineryDelegate StopMachinery
{
    get
    {
        return this.stopMachinery;
    }

    set
    {
        this.stopMachinery = value;
    }
}
```

- Provide complete encapsulation by implementing separate *Add* and *Remove* methods. The *Add* method takes a method as a parameter and adds it to the delegate, whereas the *Remove* method removes the specified method from the delegate.

```
public void Add(stopMachineryDelegate stopMethod)
{
    this.stopMachinery += stopMethod;
}

public void Remove(stopMachineryDelegate stopMethod)
{
    this.stopMachinery -= stopMethod;
}
```

Lambda expressions and delegates

- All the examples of adding a method to a delegate that you have seen so far use the method's name. For example, returning to the automated factory scenario described earlier, you add the *StopFolding* method of the *folder* object to the *stopMachinery* delegate, like this

  this.stopMachinery += folder.StopFolding;

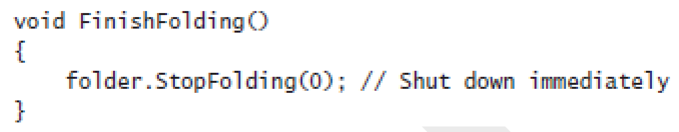
- This approach is very useful if there is a convenient method that matches the signature of the delegate, but what if this is not the case? Suppose that the *StopFolding* method actually had the following signature:

void StopFolding(int shutDownTime); // Shut down in the specified number of seconds

- This signature is now different from that of the *FinishWelding* and *PaintOff* methods, and therefore you cannot use the same delegate to handle all three methods so the solution:

Creating a method adapter

Solution is to create another method that calls *StopFolding* but that takes no parameters itself, like this:



```
void FinishFolding()
{
    folder.StopFolding(0); // Shut down immediately
}
```

- You can then add the *FinishFolding* method to the *stopMachinery* delegate in place of the *StopFolding* method, using the same syntax as before:

  this.stopMachinery += folder.FinishFolding;

- When the *stopMachinery* delegate is invoked, it calls *FinishFolding*, which in turn calls the *Stop Folding* method, passing in the parameter of 0

The forms of lambda expressions

- some examples showing the different forms of lambda expressions available in C#:

```
x => x * x // A simple expression that returns the square of its parameter
           // The type of parameter x is inferred from the context.

x => { return x * x ; } // Semantically the same as the preceding
                        // expression, but using a C# statement block as
                        // a body rather than a simple expression

(int x) => x / 2 // A simple expression that returns the value of the
                 // parameter divided by 2
                 // The type of parameter x is stated explicitly.

() => folder.StopFolding(0) // Calling a method
                            // The expression takes no parameters.
                            // The expression might or might not
                            // return a value.

(x, y) => { x++; return x / y; } // Multiple parameters; the compiler
                                 // infers the parameter types.
                                 // The parameter x is passed by value, so
                                 // the effect of the ++ operation is
                                 // local to the expression.

(ref int x, int y) => { x++; return x / y; } // Multiple parameters
                                             // with explicit types
                                             // Parameter x is passed by
                                             // reference, so the effect of
                                             // the ++ operation is permanent.
```

- some features of lambda expressions of which you should be aware:

1) If a lambda expression takes parameters, you specify them in the parentheses to the left of the => operator. You can omit the types of parameters, and the C# compiler will infer their types from the context of the lambda expression.
2) Lambda expressions can return values, but the return type must match that of the delegate to which they are being added.
3) The body of a lambda expression can be a simple expression or a block of C# code made up of multiple statements, method calls, variable definitions, and other code items.
4) Variables defined in a lambda expression method go out of scope when the method finishes.
5) A lambda expression can access and modify all variables outside the lambda expression that are in scope when the lambda expression is defined.

Declaring an event

- An *event source* is usually a class that monitors its environment and raises an event when something significant happens. In the automated factory, an event source could be a class that monitors the temperature of each machine. The temperature-monitoring class would raise a “machine overheating” event if it detects that a machine has exceeded its thermal radiation boundary (that is, it has become too hot).

- We declare an event similarly to how you declare a field. However, because events are intended to be used with delegates, the type of an event must be a delegate, and we must prefix the declaration with the *event* keyword. Use the following syntax to declare an event:

  event *delegateTypeName eventName*

- As an example, here’s the *StopMachineryDelegate* delegate from the automated factory. It has been relocated to a new class called *TemperatureMonitor*. We can define the *MachineOverheating* event, which will invoke the *stopMachineryDelegate*, like this:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
}
```

Subscribing to an event

Like delegates, events come ready-made with a += operator. We subscribe to an event by using this += operator.

```
    class TemperatureMonitor
    {
        public delegate void StopMachineryDelegate();
        public event StopMachineryDelegate MachineOverheating;
        ...
    }
...
TemperatureMonitor tempMonitor = new TemperatureMonitor();
...
tempMonitor.MachineOverheating += (() => { folder.StopFolding(0); });
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;
```

Unsubscribing from an event

Knowing that you use the += operator to attach a delegate to an event, we can probably guess that you use the –= operator to detach a delegate from an event.

Raising an event

We can raise an event, just like a delegate, by calling it like a method. When we raise an event, all the attached delegates are called in sequence. For example, here’s the *TemperatureMonitor* class with a private *Notify* method that raises the *MachineOverheating* event:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
    private void Notify()
    {
        if (this.MachineOverheating != null)
        {
            this.MachineOverheating();
        }
    }
    ...
}
```