ISAM 5430 Class 01

Syllabus (Lecture)
Chapter 1 (Lecture)
(*break*)
Chapter 3 (Lab)

We will be using two classrooms:
SSB 2311 (Lecture) and
SSB 2.201.03 (Lab)

Learning Outcomes: Part I
C# semantics/syntax, algorithms, debugging

- apply the C# language and the .NET framework to implement software solutions;
- apply appropriate C# data types, including int, bool, double/float, decimal, char, string, and arrays;
- practice various techniques and algorithms that involve tracking, mapping, permutation, and recursion;
- develop software codes that make use of variables and control statements with loops and if-statements;
- tabulate test cases, use cases, and runtime data;
- identify compilation errors, runtime errors, and logical errors in the codes;
- operate Visual Studio.NET as an integrated development environment to create and build C# solutions;
- experiment various Visual Studio features, such as intellisense, debugging, error logs, and variable watches.

Learning Outcomes (Part II and III)

- **Part II**: *Object-oriented programming*—
  - organize C# codes into classes, fields, properties, and methods;
  - employ object-oriented principles—encapsulation, abstraction, inheritance—into codes;
  - interpret written requirements into C# codes involving programming classes, operations, and attributes;
- **Part III**: *Using collections*—
  - employ data structures—lists, arrays, and hash tables/dictionaries—to solve new problems.

Course Items

- Assignments: Worth 15% of your overall grade—
  - There will be an assignment every week to hone your programming skills.
  - This ensures that you will practice writing codes at home.
  - Must be done through GITHUB assignments;
  - Codes must be pushed to github server before the class starts.
- Quizzes / Participations: Worth 20% of your overall grade—
  - Each week, you will have either a quiz, an activity, or simply an attendance.
  - This ensures that you will be able to practice writing codes in the class.
- Since this is a graduate-programming course, you are expected to spend at least:
  - 6 times the lecture hours (18h) outside weekly if you are not so well in coding;
  - or 4 times the lecture hours (12h) outside weekly if you are decent in coding.
  - The secondary objective is to build your thinking ability using application development: this ability is useful in performing well in jobs outside the programming.
  - Remember, after you finish this course, your skills should be on par to the market.

Coding Tests

- There is a total of four coding tests, in which you are required to take three coding tests.
- To reduce the amount of test materials, each coding test only targets a single objective/part. Hence, there is no need for a cumulative final exam in this course.
- Each coding test is designed so that each student does not need more than two hours to complete it, but to be fair, everyone will be given 3 hours of time to complete the test.
- Each coding test is closed-book and closed-notes. The Internet will be turned off too. That is, you can only use Visual Studio to develop and test your codes on PC.
- **Coding Test 1** (20%) – two tests on ***Saturdays*** on Part I of the course.
  - There are two parts for coding test 1: *Coding Test 1 (A)* and *Coding Test 1 (B)*.
  - Coding Test 1 score is the highest score from *Coding Test 1 (A)* and *Coding Test 2 (B),*
    - so if you receive a 60% in coding test 1 (A) and 80% in coding test 1 (B), your coding test 1 score is now worth 80%.
    - Thus, if you receive a 95% in Coding Test 1 (A), you can choose not to take Coding Test 1 (B): your coding test 1 score will simply be 95%.

Coding Tests 2 and 3

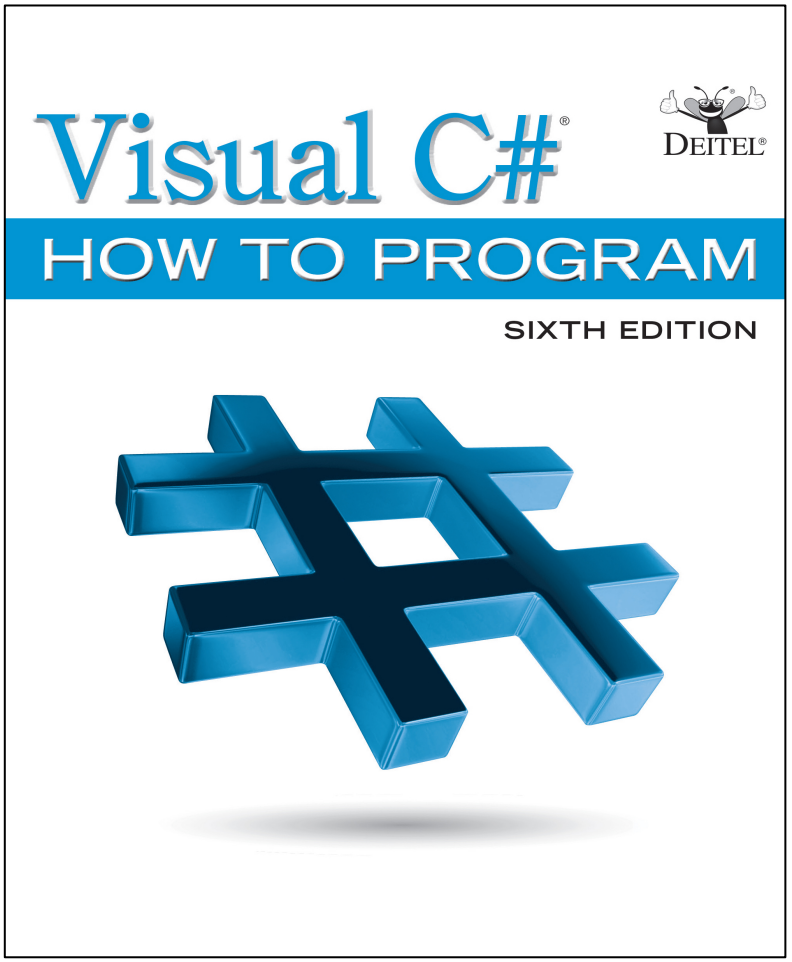
- **Coding Test 2:** one test on a ***Saturday***.
  - Part II of the course: Object-oriented programming;
  - Worth 25% of the grade.
  - Since coding test 2 is the only test that deals with OOP, it weighs slightly higher than coding tests 1 and 3.
- **Coding Test 3:** one test on ***Week 14*** (not during the final week)**.**
  - Part III of the course: collections;
  - Worth 20% of the grade;
  - Solve problems using collections whether arrays, lists, or dictionaries;
  - This test may contain similar problems as Coding Test 1 (B): solving problems with arrays.

GITHub

- Using **github** is required in this course.
- Create a **github.com** account with your **uhcl** email address.
- Go to https://classroom.github.com/classrooms/10246646-isam5430-sp20 (alias: http://git.mikeywu.com) for all the assignments.
- Visual Studio has also an excellent integration with GITHub / git.
- GIT is often used in software companies for version-tracking of codes for team collaborations.
- You can choose to your own GUI clients rather than the one in Visual Studio.
  - Windows has ***TortoiseGIT*** https://tortoisegit.org/
  - Both Mac and Windows have ***Fork git*** clients: https://git-fork.com/

Visual C# ® How to Program

Sixth Edition



Chapter 1

Introduction to Computers, the Internet and Visual C#

Pearson

Learning Objectives

- Learn basic computer hardware, software and data concepts.
- Be introduced to the different types of computer programming languages.
- Understand the history of the Visual C# programming language and the Windows operating system.
- Understand the parts that Windows, .NET, Visual Studio and C# plays in the C# ecosystem.

Outline (1 of 3)

Outline (2 of 3)

Outline (3 of 3)

1.1 Introduction

- There are billions of personal computers in use and an even larger number of mobile devices with computers at their core.
- Since it was released in 2001, C# has been used primarily to build applications for personal computers and systems that support them.
- The explosive growth of mobile phones, tablets and other devices also is creating significant opportunities for programming mobile apps.

1.3 Hardware and Software (1 of 2)

- Computers can perform calculations and make logical decisions phenomenally faster than human beings can.
- Today's personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime.
- **Supercomputers** are already performing **thousands of trillions (quadrillions)** of instructions per second!
- China's National University of Defense Technology's Tianhe-2 supercomputer can perform over 33 quadrillion calculations per second (33.86 **petaflops**)!
  - **In one second about 3 million calculations for every person on the planet!**

1.3 Hardware and Software (2 of 2)

- Computers (i.e., **hardware**) process **data** under the control of sequences of instructions called **computer programs**.
- These programs guide the computer through **actions** specified by people called **computer programmers**.
- The programs that run on a computer are referred to as **software**.

1.3.1 Moore's Law (1 of 2)

- Every year, you probably expect to pay at least a little more for most products and services.
- The opposite has been the case in the computer and communications fields, especially with regard to the hardware supporting these technologies. For many decades, hardware costs have fallen rapidly.
- Every year or two, the capacities of computers have approximately-**doubled** inexpensively.
- This remarkable trend often is called **Moore's Law**, named for the person who identified it in the 1960s, Gordon Moore, co-founder of Intel—a leading manufacturer of the processors in today's computers and embedded systems.

1.3.1 Moore's Law (2 of 2)

- Moore's Law and related observations apply especially to the amount of memory that computers have for programs, the amount of secondary storage (such as disk storage) they have to hold programs- and data over longer periods of time, and their processor speeds—the speeds at which they **execute** their programs (i.e., do their work).
- These increases make computers more capable, which puts greater demands on programming-language designers to innovate.
- Similar growth has occurred in the communications.

1.3.2 Computer Organization

- Regardless of differences in **physical** appearance, computers can be envisioned as divided into various **logical units** or sections (Figure 1.2).

Figure 1.2 Logical Units of a Computer (1 of 6)

Name	Description
Input unit	This “receiving” section obtains information (data and computer programs) from **input devices** and places it at the disposal of the other units for processing. Most user input is entered into computers through keyboards, touch screens and mouse devices. Other forms of input include receiving voice commands, scanning images and barcodes, reading from secondary storage devices (such as hard drives, DVD drives, Blu-ray Disc™ drives and USB flash drives—also called “thumb drives” or “memory sticks”), receiving video from a webcam and having your computer receive information from the Internet (such as when you stream videos from YouTube® or download e-books from Amazon). Newer forms of input include position data from a GPS device, motion and orientation information from an **accelerometer** (a device that responds to up/down, left/right and forward/backward acceleration) in a smartphone or game controller (such as Microsoft® Kinect® for Xbox®, Wii™ Remote and Sony® PlayStation® Move) and voice input from devices like Amazon Echo and the forthcoming Google Home.

Figure 1.2 Logical Units of a Computer (2 of 6)

Name	Description
Output unit	This “shipping” section takes information the computer has processed and places it on various **output devices** to make it available for use outside the computer. Most information that’s output from computers today is displayed on screens (including touch screens), printed on paper (“going green” discourages this), played as audio or video on PCs and media players (such as Apple’s iPods) and giant screens in sports stadiums, transmitted over the Internet or used to control other devices, such as robots and “intelligent” appliances. Information is also commonly output to secondary storage devices, such as solid-state drives (SS Ds), hard drives, DVD drives and USB flash drives. Popular recent forms of output are smartphone and game-controller vibration, virtual reality devices like Oculus Rift and Google Cardboard and mixed reality devices like Microsoft’s HoloLens.

Figure 1.2 Logical Units of a Computer (3 of 6)

Name	Description
Memory unit	This rapid-access, relatively low-capacity "warehouse" section retains information that has been entered through the input unit, making it immediately available for processing when needed. The memory unit also retains processed information until it can be placed on output devices by the output unit. Information in the memory unit is **volatile**—it's typically lost when the computer's power is turned off. The memory unit is often called either **memory**, **primary memory** or **RAM** (Random Access Memory). Main memories on desktop and notebook computers contain as much as 128 GB of RAM, though 2 to 16 GB is most common. GB stands for gigabytes; a gigabyte is approximately one billion bytes. A **byte** is eight bits. A bit is either a 0 or a 1.

Figure 1.2 Logical Units of a Computer (4 of 6)

Name	Description
Arithmetic and logic unit (ALU)	This “manufacturing” section performs **calculations**, such as addition, subtraction, multiplication and division. It also contains the **decision** mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they’re equal. In today’s systems, the AL U is implemented as part of the next logical unit, the CPU.

Figure 1.2 Logical Units of a Computer (5 of 6)

Name	Description
Central processing unit (CPU)	This "administrative" section coordinates and supervises the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the ALU when information from the memory unit should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices. Many of today's computers have multiple CPUs and, hence, can perform many operations simultaneously. A **multicore processor** implements multiple processors on a single integrated-circuit chip—a **dual-core processor** has two CP Us, a **quad-core processor** has four and an **octa-core processor** has eight. Today's desktop computers have processors that can execute billions of instructions per second. Chapter 23 explores how to write apps that can take full advantage of multicore architecture.

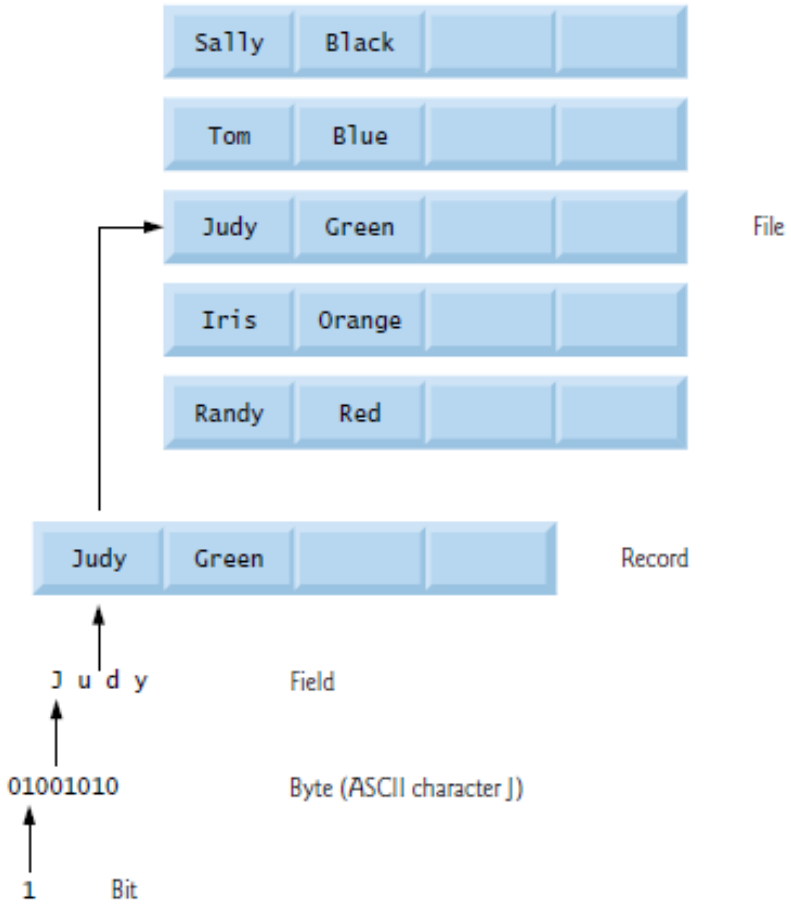
Figure 1.2 Logical Units of a Computer (6 of 6)

Name	Description
Secondary storage unit	This is the long-term, high-capacity "warehousing" section. Programs or data not actively being used by the other units normally are placed on secondary storage devices (example, your **hard drive**) until they're again needed, possibly hours, days, months or even years later. Information on secondary storage devices is **persistent**—it's preserved even when the computer's power is turned off. Secondary storage information takes much longer to access than information in primary memory, but its cost per unit is much less. Examples of secondary storage devices include solid-state drives (SSDs), hard drives, DVD drives and USB flash drives, some of which can hold over 2 TB (TB stands for terabytes; a terabyte is approximately one trillion bytes). Typical hard drives on desktop and notebook computers hold up to 2 TB, and some desktop hard drives can hold up to 6 TB.

1.4 Data Hierarchy (1 of 9)

- Data items processed by computers form a **data hierarchy** that becomes larger and more complex in structure as we progress from the simplest data items (called “bits”) to richer data items, such as characters, fields, and so on.

Figure 1.3 Data Hierarchy



1.4 Data Hierarchy (2 of 9)

Bits

- The smallest data item in a computer can assume the value 0 or the value 1.
- Such a data item is called a **bit** (short for "binary digit"—a digit that can assume either of two values).
- It's remarkable that the impressive functions performed by computers involve only the simplest manipulations of 0s and **1s—examining a bit's value, setting a bit's value** and **reversing a bit's value** (from 1 to 0 or from 0 to 1).

1.4 Data Hierarchy (3 of 9)

Characters

- We prefer to work with **decimal digits** (0–9), **uppercase letters** (A–Z), **lowercase letters** (a–z), and **special symbols** (example $, @, %, &, *, (,), –, +, ", :, ? and /).
- Digits, letters and special symbols are known as characters. The computer's character set is the set of all the **characters** used to write programs and represent data items on that device.
- Computers process only 1s and 0s, so every character is represented as a pattern of 1s and 0s.
- The **Unicode** character set contains characters for many of the world's languages.

1.4 Data Hierarchy (4 of 9)

- C# supports several character sets, including 16-bit Unicode® characters that are composed of two bytes—each byte is composed of eight bits.
- See Appendix B for more information on the **ASCII (American Standard Code for Information Interchange)** character set—the popular **subset** of Unicode that represents uppercase and lowercase letters in the English alphabet, digits and some common **special characters**.

1.4 Data Hierarchy (5 of 9)

Fields

- Just as characters are composed of bits, **fields** are composed of characters or bytes.
- A field is a group of characters or bytes that conveys meaning.
- For example, a field consisting of uppercase and lowercase letters could be used to represent a person's name, and a field consisting of decimal digits could represent a person's age.

1.4 Data Hierarchy (6 of 9)

Records

- Several related fields can be used to compose a **record**.
- In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses):
  - Employee identification number (a whole number)
  - Name (a string of characters)
  - Address (a string of characters)
  - Hourly pay rate (a number with a decimal point)
  - Year-to-date earnings (a number with a decimal point)
  - Amount of taxes withheld (a number with a decimal point)
- Thus, a record is a group of related fields.
- In the preceding example, all the fields belong to the same employee.

1.4 Data Hierarchy (7 of 9)

Files

- A **file** is a group of related records.
- More generally, a file contains arbitrary data in arbitrary formats.
- In some operating systems, a file is viewed simply as a **sequence of bytes**—any organization of the bytes in a file, such as organizing the data into records, is a view created by the programmer.

1.4 Data Hierarchy (8 of 9)

Database

- A **database** is a collection of data that's organized for easy access and manipulation.
- The most popular database model is the **relational database** in which data is stored in simple **tables**.
- A table includes **records** composed of **fields**.
  - For example, a table of students might include first name, last name, major, year, student ID number and grade point average fields.
  - The data for each student is a record, and the individual pieces of information in each record are the fields.
- You can **search, sort** and otherwise manipulate the data based on its relationship to multiple tables or databases.

Figure 1.4 Byte Measurements

Unit	Bytes	Which is approximately
1 kilobyte (KB)	1024 bytes	10^3 (1024) bytes exactly
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000) bytes
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000) bytes
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000) bytes
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000) bytes
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000) bytes
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000) bytes

1.5 Machine Languages, Assembly Languages and High-Level Languages (1 of 3)

- Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate **translation** steps.
- Any computer can directly understand only its own **machine language** (also called **machine code**), defined by its hard-ware architecture.
- Machine languages generally consist of numbers (ultimately reduced to 1s and 0s). Such languages are cum-bersome for humans.

1.5 Machine Languages, Assembly Languages and High-Level Languages (2 of 3)

- English-like **abbreviations** formed the basis of **assembly languages**.
- **Translator programs** called **assemblers** were developed to convert assembly-language programs to machine language.
- Although assembly-language code is clearer to humans, it's incomprehensible to computers until translated to machine language.

1.5 Machine Languages, Assembly Languages and High-Level Languages (3 of 3)

- **High-level languages** were developed in which single statements could be written to accomplish substantial tasks.
- High-level languages, such as C#, Visual Basic, C, C++, Java and Swift, allow you to write instructions that look more like everyday English and contain commonly used mathematical notations.
- Translator programs called **compilers** convert high-level language programs into machine language.
- **Interpreter** programs execute high-level language programs directly (without the need for compilation), although more slowly than compiled programs.
- **Scripting languages** such as the popular web languages JavaScript and PHP are processed by interpreters.

Performance Tip 1.1

Interpreters have an advantage over compilers in Internet scripting. An interpreted program can begin executing as soon as it's downloaded to the client's machine, without needing to be compiled before it can execute. On the downside, interpreted scripts generally run slower and consume more memory than compiled code. With a technique called JIT (just-in-time) compilation, interpreted languages can often run almost as fast as compiled ones.

1.8 C#

- In 2000, Microsoft announced the **C#** programming language.
- C# has roots in the C, C++ and Java programming languages.

1.8.1 Object-Oriented Programming

- C# is **object oriented**—we've discussed the basics of object technology and we present a rich treatment of object-oriented programming throughout the book.
- C# has access to the powerful **.NET Framework Class Library**—a vast collection of prebuilt classes that enable you to develop apps quickly (Figure 1.5).

Figure 1.5 Some Key Capabilities in the .Net Framework Class Library

Some key capabilities in the .NET Framework Class Library

Database	Debugging
Building web apps	Multithreading
Graphics	File processing
Input/output	Security
Computer networking	Web communication
Permissions	Graphical user interface
Mobile	Data structures
String processing	Universal Windows Platform GUI

1.8.5 An International Standard

- C# has been standardized through ECMA International:
  - http://www.ecma-international.org
- At the time of this writing, the C# standard document—ECMA-334—was still being updated for C# 6.
- For information on ECMA-334, visit
  - http://www.ecma-international.org/publications/standards/Ecma-334.htm
- Visit the Microsoft download center to find the latest version of Microsoft's C# 6 specification, other documentation and software downloads.

1.8.6 C# on Non-Windows Platforms

- Though C# was originally developed by Microsoft for the Windows platform, the language can be used on other platforms via the **Mono Project** and **.NET Core**—both are managed by the .NET Foundation
  - http://www.dotnetfoundation.org/

1.9 Microsoft's .NET

- In 2000, Microsoft announced its **.NET initiative** (www.microsoft.com/net), a broad vision for using the Internet and the web in the development, engineering, distribution and use of software.
- Rather than forcing you to use a single programming language, .NET permits you to create apps in **any** .NET-compatible language (such as C#, Visual Basic, Visual C++ and others).

1.9.1 .NET Framework

- The **.NET Framework Class Library** provides many capabilities that you'll use to build substantial C# apps quickly and easily.
- It contains **thousands** of valuable **prebuilt** classes that have been tested and tuned to maximize performance.
- You should **re-use** the .NET Framework classes whenever possible to speed up the software-development process, while enhancing the quality and performance of the software you develop.

1.9.2 Common Language Runtime (1 of 3)

- The **Common Language Runtime (CLR)** executes .NET programs and provides functionality to make them easier to develop and debug.
- The CLR is a **virtual machine** (**VM**)—software that manages the execution of programs and hides from them the underlying operating system and hardware.
- The source code for programs that are executed and managed by the CLR is called **managed code**.

1.9.2 Common Language Runtime (2 of 3)

- The CLR provides various services to managed code
  - integrating software components written in different .NET languages,
  - error handling between such components,
  - enhanced security,
  - automatic memory management and more.
- Unmanaged-code programs do not have access to the CLR's services, which makes unmanaged code more difficult to write.

1.9.2 Common Language Runtime (3 of 3)

- Managed code is compiled into machine-specific instructions in the following steps:
  - First, the code is compiled into **Microsoft Intermediate Language** (**MSIL**).
    - Code converted into MSIL from other languages and sources can be woven together by the CLR—this allows programmers to work in their preferred .NET programming language.
    - The MSIL for an app's components is placed into the app's **executable file**—the file that causes the computer to perform the app's tasks.
  - When the app executes, another compiler (known as the **just-in-time compiler** or **JIT compiler**) in the CLR translates the MSIL in the executable file into machine-language code (for a particular platform).
  - The machine-language code executes on that platform.

1.9.3 Platform Independence

- If the .NET Framework exists and is installed for a platform, that platform can run **any** .NET program.
- The ability of a program to run without modification across multiple platforms is known as **platform independence**.
  - Code written once can be used on another type of computer without modification, saving time and money.

1.9.4 Language Interoperability

- The .NET Framework provides a high level of **language interoperability**.
- Because software components written in different .NET languages (such as C# and Visual Basic) are all compiled into MSIL, the components can be combined to create a single unified program.
  - Thus, MSIL allows the .NET Framework to be **language independent**.

1.9.4 Language Interoperability

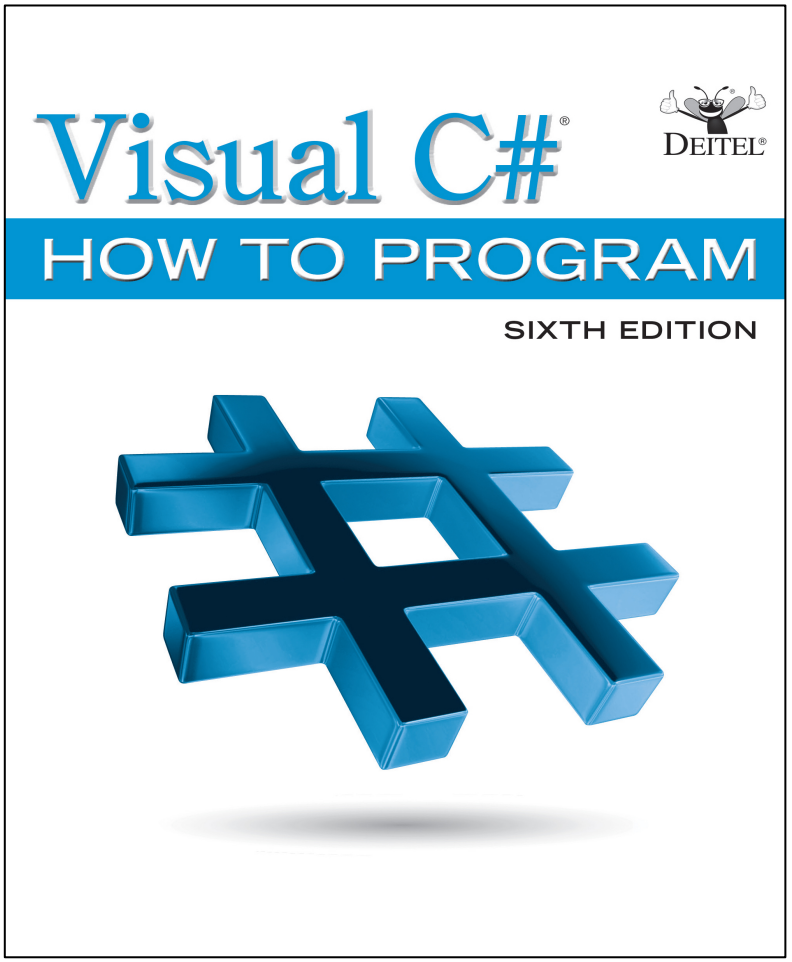
- The .NET Framework Class Library can be used by any .NET language. The latest release of .NET includes .NET 4.6 and .NET Core:
  - NET 4.6 introduces many improvements and new features, including ASP.NET 5 for web-based applications, improved support for today's high-resolution 4K screens and more.
  - .NET Core is the cross-platform subset of .NET for Windows, Linux, OS X and FreeBSD.

1.11 Visual Studio Integrated Development Environment

- C# programs can be created using Microsoft's Visual Studio—a collection of software tools called an **Integrated Development Environment (IDE)**.
- The **Visual Studio Community** edition IDE enables you to **write, run, test** and **debug** C# programs quickly and conveniently.
- It also supports Microsoft's Visual Basic, Visual C++ and F# programming languages and more.

Visual C#® How to Program

Sixth Edition



Chapter 3

Introduction to C# App Programming

Pearson

Learning Objectives

- Write simple C# apps using code rather than visual programming.
- Input data from the keyboard and output data to the screen.
- Use C# 6's `string` interpolation to create formatted `strings` by inserting values into `string` literals.
- Declare and use data of various types.
- Store data in memory and retrieve it.
- Use arithmetic operators.
- Determine the order in which operators are applied.
- Write decision-making statements with equality and relational operators.

Outline (1 of 4)

Outline (2 of 4)

Outline (3 of 4)

Outline (4 of 4)

3.1 Introduction

- **Console apps** input and output text in a **console window**, which in Windows is known as the **Command Prompt**.
- Figure 3.1 shows the app's source code and output.

3.2 Simple App: Displaying a Line of Text

- Figure 3.1 shows the app's source code and output.

Figure 3.1 Text-Displaying App

```
1  // Fig. 3.1: Welcome1.cs
2  // Text-displaying app.
3  using System;
4
5  class Welcome1
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       Console.WriteLine("Welcome to C# Programming!");
11    } // end Main
12 } // end class Welcome1
```

```
Welcome to C# Programming!
```

3.2.1 Comments (1 of 2)

- You'll insert **comments** to document your apps and improve their readability.
- The C# compiler ignores comments, so they do **not** cause the computer to perform any action when the app executes.
- A comment that begins with `//` is called a **single-line comment**, because it terminates at the end of the line on which it appears.

3.2.1 Comments (2 of 2)

- A // comment also can begin in the middle of a line and continue until the end of that line.
- **Delimited comments** begin with the delimiter /* and end with the delimiter */. All text between the delimiters is ignored by the compiler.

Common Programming Error 3.1

Forgetting one of the delimiters of a delimited comment is a syntax error. A programming language's **syntax** specifies the grammatical rules for writing code in that language. A **syntax error** occurs when the compiler encounters code that violates C#'s language rules. In this case, the compiler does not produce an executable file. Instead, it issues one or more error messages to help you identify and fix the incorrect code. Syntax errors are also called **compiler errors, compile-time errors** or **compilation errors**, because the compiler detects them during the compilation phase. You cannot execute your app until you correct all the compilation errors in it. We'll see that some compile-time errors are not syntax errors.

Error-Prevention Tip 3.1

When the compiler reports an error, the error may not be in the line indicated by the error message. First, check the line for which the error was reported. If that line does not contain syntax errors, check several preceding lines.

3.2.2 Using Directive

- A **`using` directive** tells the compiler where to look for a predefined class that's used in an app.
- Predefined classes are organized under **namespaces**—named collections of related classes. Collectively, .NET's predefined namespaces are known as the **.NET Framework Class Library**.
- The `System` namespace contains the predefined `Console` class and many other useful classes.

Error-Prevention Tip 3.2

Forgetting to include a `using` directive for a namespace that contains a class used in your app typically results in a compilation error, containing a message such as "`The name 'Console' does not exist in the current context`." When this occurs, check that you provided the proper `using` directives and that the names in them are spelled correctly, including proper use of uppercase and lowercase letters. In the editor, when you hover over an error's red squiggly line, Visual Studio displays a box containing the link "show potential fixes." If a `using` directive is missing, one potential fix shown is to add the `using` directive to your code—simply click that fix to have Visual Studio edit your code.

3.2.3 Blank Lines and Whitespace

- Blank lines and space characters make code easier to read and together with tab characters are known as **whitespace**.
- Whitespace is ignored by the compiler.

3.2.4 Class Declaration (1 of 3)

- **Keywords** (sometimes called **reserved words**) are reserved for use by C# and are always spelled with all lowercase letters.
- Every app consists of at least one **class declaration** that is defined by the programmer. These are known as **user-defined classes**.
- The **`class`** keyword introduces a class declaration and is immediately followed by the **class name**.

3.2.4 Class Declaration (2 of 3)

- A class name is an **identifier**:
  - Series of characters consisting of letters, digits and underscores.
  - Cannot begin with a digit and does not contain spaces.
- The complete list of C# keywords is shown in Figure 3.2.
- The contextual keywords in Figure 3.2 can be used as identifiers outside the contexts in which they are keywords, but for clarity this is not recommended.

Good Programming Practice 3.1

By convention, always begin a class name's identifier with a capital letter and start each subsequent word in the identifier with a capital letter.

Common Programming Error 3.2

C# is case sensitive. Not using the proper uppercase and lowercase letters for an identifier normally causes a compilation error.

Common Programming Error 3.3

Using a keyword as an identifier is a compilation error.

Figure 3.2 Keywords and Contextual Keywords (1 of 2)

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

Figure 3.2 Keywords and Contextual Keywords (2 of 2)

Keywords and contextual keywords

Contextual Keywords

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

3.2.4 Class Declaration (3 of 3)

- C# is **case sensitive**—that is, uppercase and lowercase letters are distinct, so `a1` and `A1` are different (but both valid) identifiers.

Good Programming Practice 3.2

By convention, a file that contains a single class should have a name that's identical to the class name (plus the `.cs` extension) in both spelling and capitalization. This makes it easy to identify which file contains the class's declaration.

3.2.4 Class Declaration

- A **left brace**, {, begins the **body** of every class declaration.
- A corresponding **right brace**, }, must end each class declaration.

Good Programming Practice 3.3

Indent the entire body of each class declaration one “level” of indentation between the left and right braces that delimit the body of the class. This format emphasizes the class declaration’s structure and makes it easier to read. You can let the IDE format your code by selecting **Edit → Advanced → Format Document**.

Good Programming Practice 3.4

Set a convention for the indent size you prefer, then uniformly apply that convention. The **Tab** key may be used to create indents, but tab stops vary among text editors. Microsoft recommends four-space indents, which is the default in Visual Studio. Due to the limited width of code lines in print books, we use three-space indents—this reduces the number of code lines that wrap to a new line, making the code a bit easier to read. We show how to set the tab size in the Before You Begin section that follows the Preface.

Error-Prevention Tip 3.3

Whenever you type an opening left brace, {, in your app, the IDE immediately inserts the closing right brace, }, then repositions the cursor between the braces so you can begin typing the body. This practice helps prevent errors due to missing braces.

Common Programming Error 3.4

It’s a syntax error if braces do not occur in matching pairs.

3.2.5 Main Method

- **Parentheses** after an identifier indicate that it is an app building block called a method. Class declarations normally contain one or more methods.
- Method names usually follow the same casing capitalization conventions used for class names.
- For each app, one of the methods in a class must be called `Main`; otherwise, the app will not execute.
- A method performs a task and returns information it completes its task.
- Keyword **`void`** indicates that `Main` will not return any information.
- The **body** of a method declaration begins with a left brace and ends with a corresponding right brace.

Good Programming Practice 3.5

Indent each method declaration's body statements one level of indentation between the left and right braces that define the body.

3.2.6 Displaying a Line of Text (1 of 2)

- Characters between double quotation marks are **strings**.
- Whitespace characters in strings are **not** ignored by the compiler.
- The **`Console.WriteLine` method** displays a line of text in the console window.
- The string in parentheses is the **argument** to the **`Console.WriteLine` method**.
- Method **`Console.WriteLine`** performs its task by displaying (also called outputting) its argument in the console window.

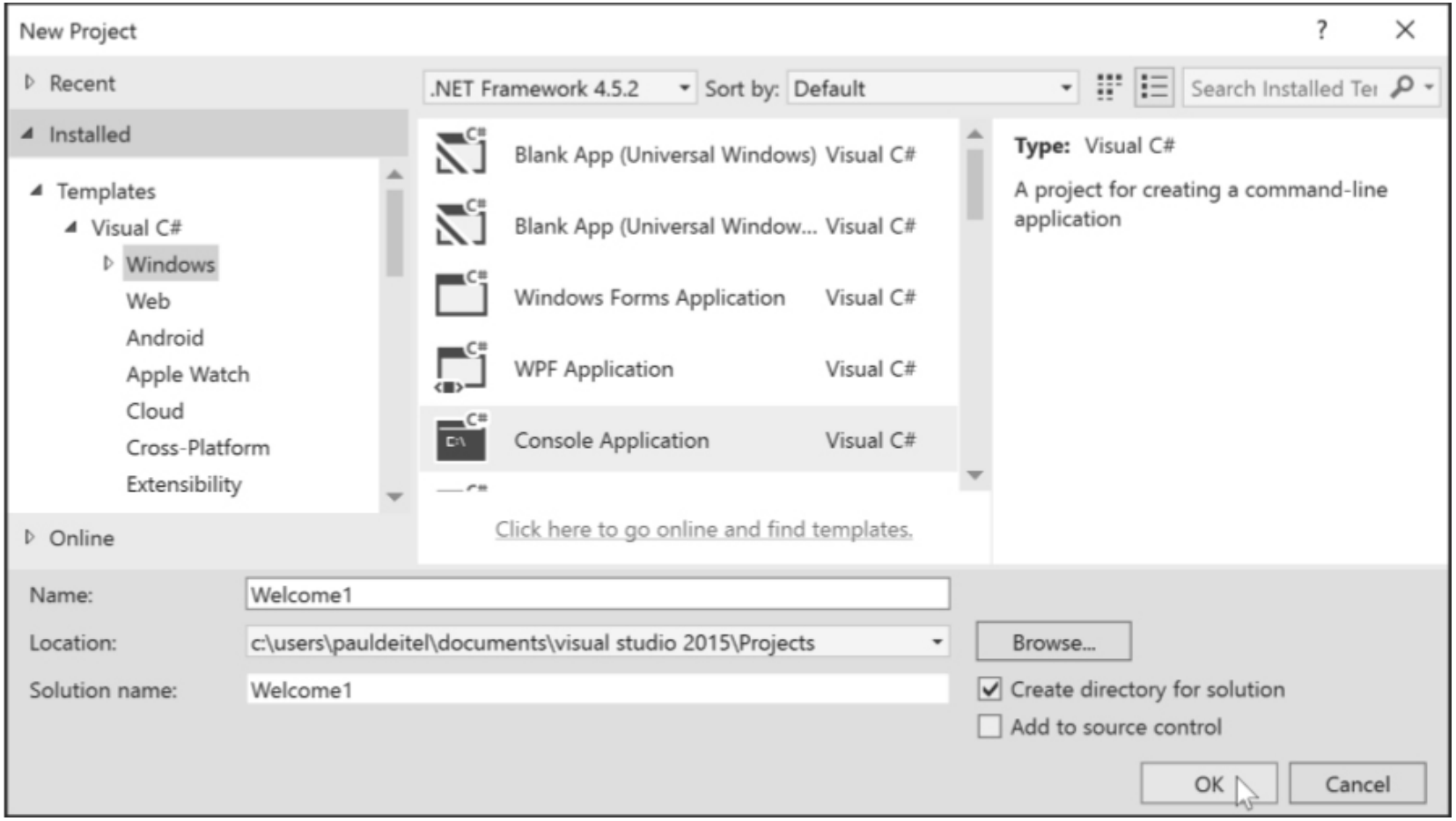
3.2.6 Displaying a Line of Text (2 of 2)

- A method is typically composed of one or more **statements** that perform the method's task.
- Most statements end with a semicolon.

3.3.1 Creating the Console App

- Select **File** → **New** → **Project…** to display the **New Project** dialog (Figure 3.3).
- At the left side of the dialog, under **Installed** → **Templates** → **Visual C#** select the **Windows** category, then in the middle of the dialog select the **Console Application** template.
- In the dialog's **Name** field, type `Welcome1`, then click **OK** to create the project.

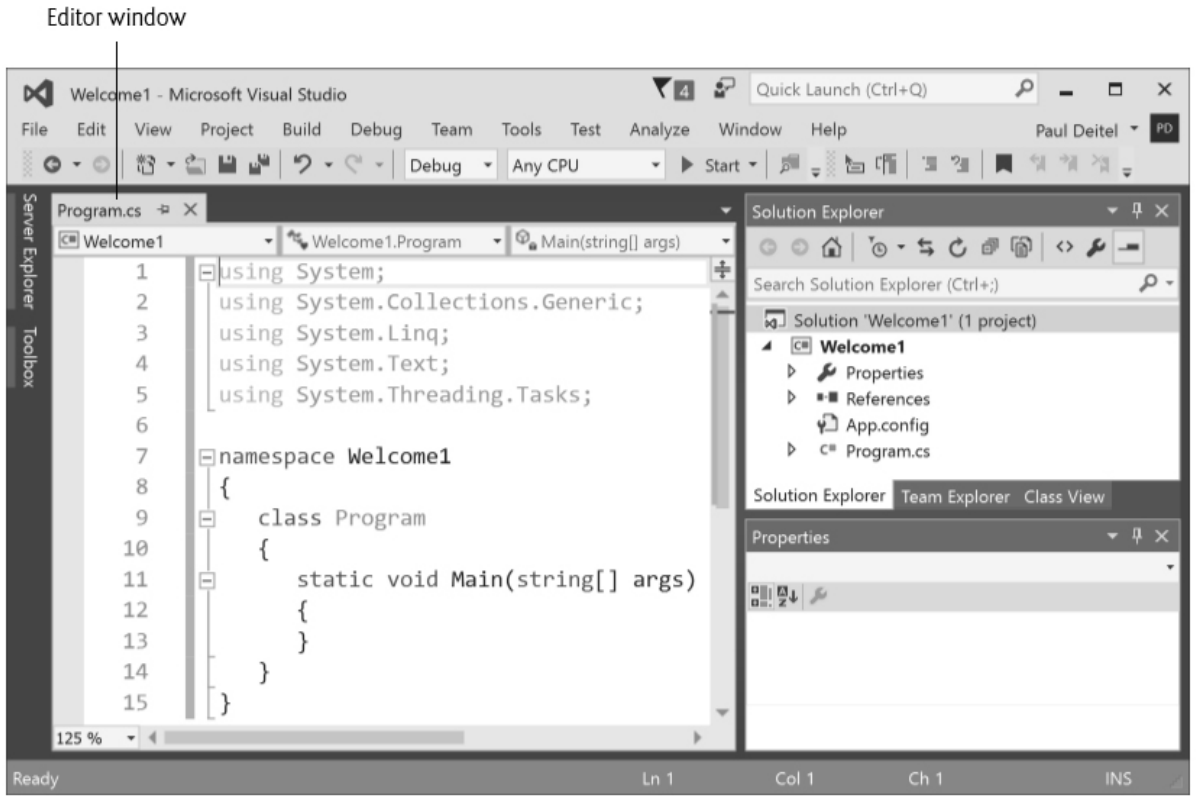
Figure 3.3 Selecting Console Application in the New Project Dialog



3.3 Creating a Simple App in Visual Studio

- The IDE now contains the open console app.
- The code coloring scheme used by the IDE is called **syntax-color highlighting** and helps you visually differentiate code elements.

Figure 3.4 IDE with an Open Console App's Code Displayed in the Editor and the Project's Contents Shown in the Solution Explorer at the IDE's Top-Right Side



3.3.2 Changing the Name of the App File

- To rename the app file, right click `Program.cs` in the **Solution Explorer** window and select **Rename** window
- Type `Welcome1` then press **Enter**.

Error-Prevention Tip 3.4

When changing a file name in a Visual Studio project, always do so in Visual Studio. Changing file names outside the IDE can break the project and prevent it from executing.

3.3.3 Writing Code and Using IntelliSense (1 of 2)

- **IntelliSense** lists a class's **members**, which include method names.
- As you type, **IntelliSense** lists various items that start with or contain the letters you've typed so far.
- **IntelliSense** also displays a tool tip containing a description of the first matching item.

3.3.3 Writing Code and Using IntelliSense (2 of 2)

- You can either type the complete member name, double click the member name in the member list or press the **Tab** key to complete the name.
- While the **IntelliSense** window is displayed pressing the **Ctrl** key makes the window transparent so you can see the code behind the window.

Figure 3.5 IntelliSense Window as You Type “Con”

a) *IntelliSense* window displayed as you type

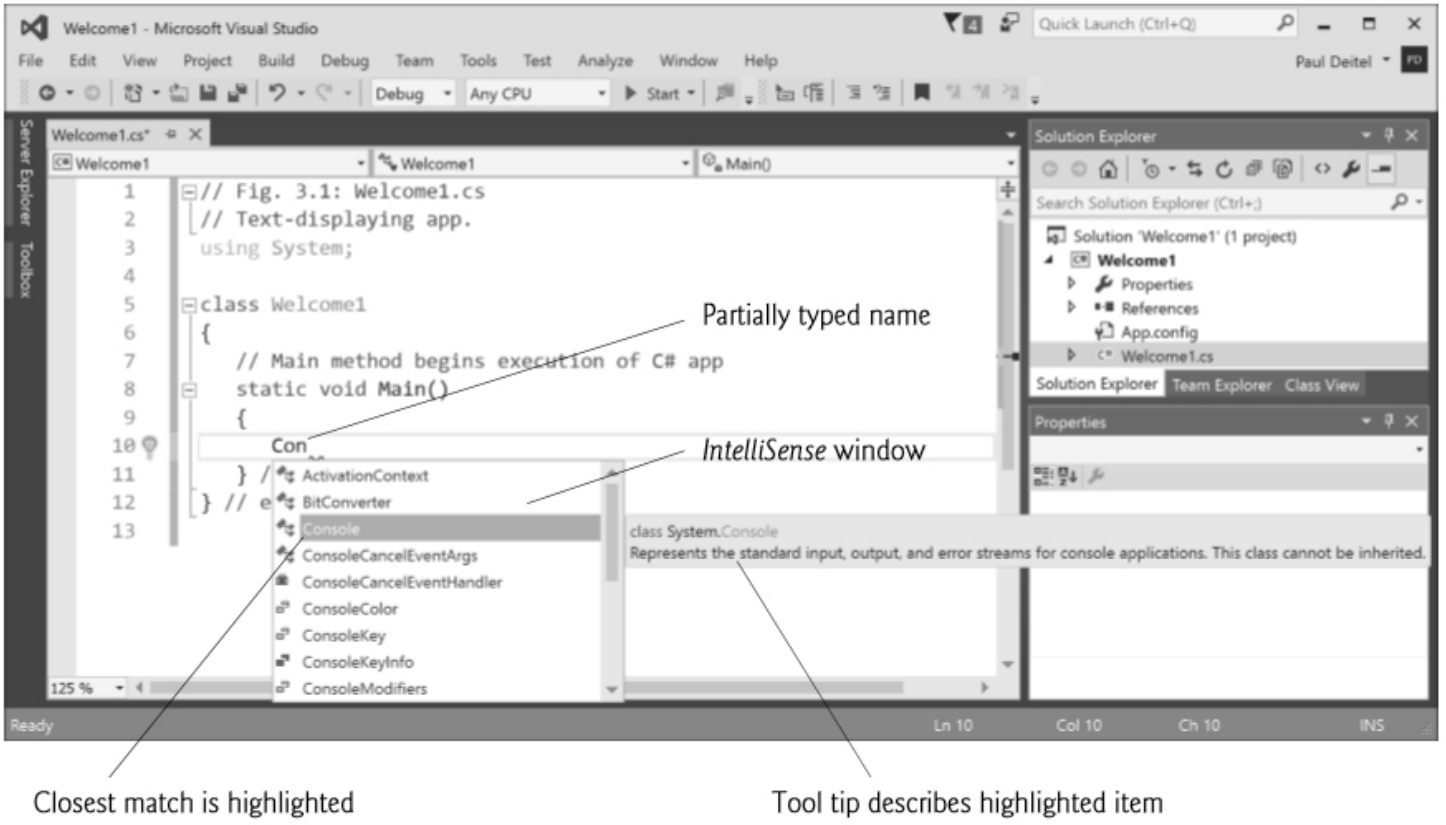
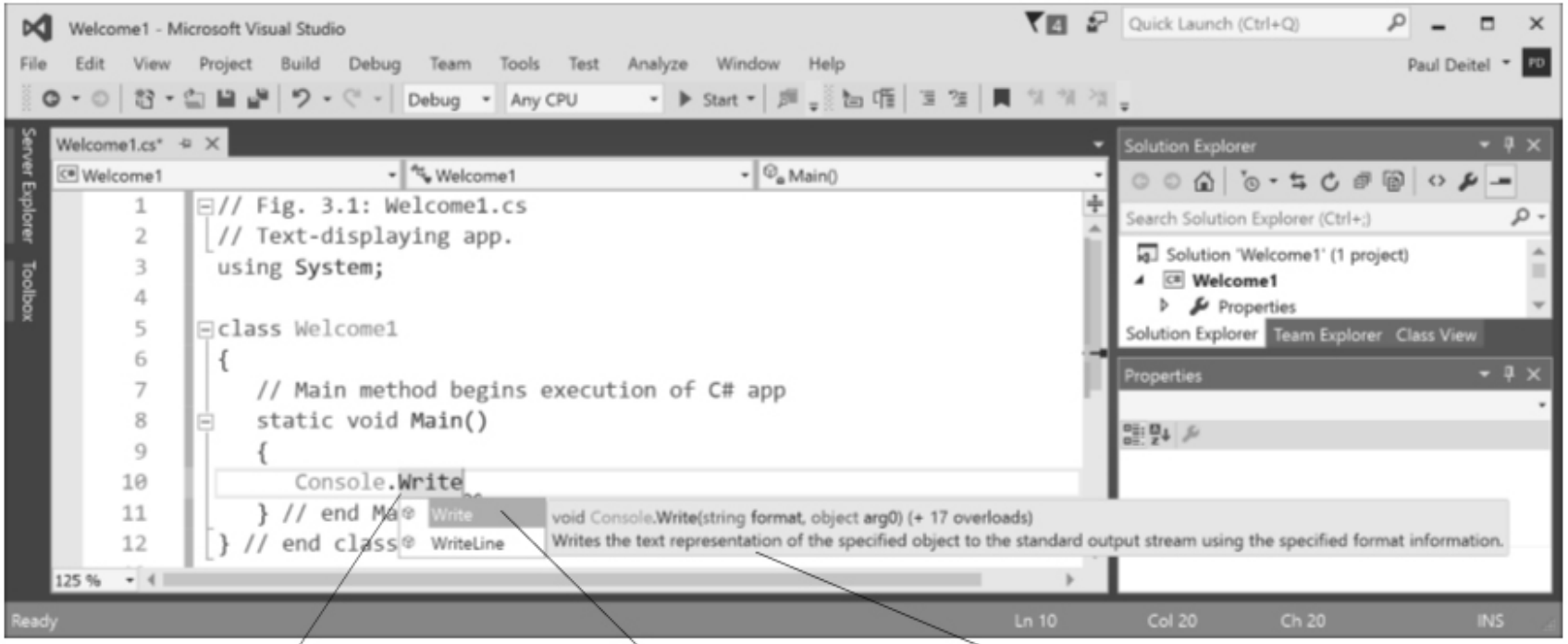


Figure 3.6 IntelliSense Window

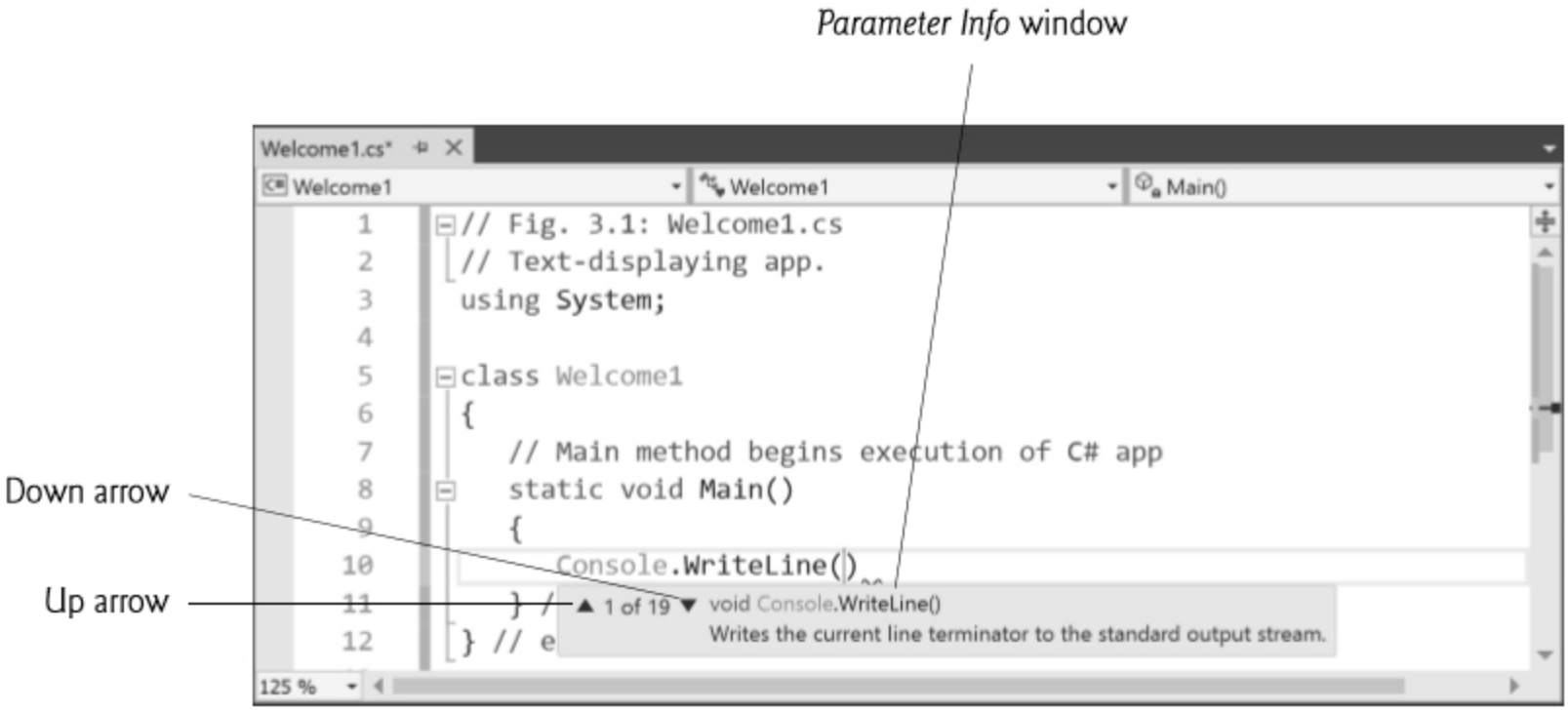
b) *IntelliSense* window showing method names that start with `Write`



3.3.3 Writing Code and Using IntelliSense

- When you type the open parenthesis character, (, after a method name, the **Parameter Info** window is displayed (Figure 3.7).
- This window contains information about the method's parameters.
- Up and down arrows allow you to scroll through overloaded versions of the method.

Figure 3.7 Parameter Info Window



3.3.4 Compiling and Running the App

- To compile the app, select **Build → Build Solution**.
  - This invokes the `Main` method.
- Figure 3.8 shows the results of executing this app, displayed in a console **(Command Prompt)** window.

Figure 3.8 Executing the App Shown in Figure 3.1

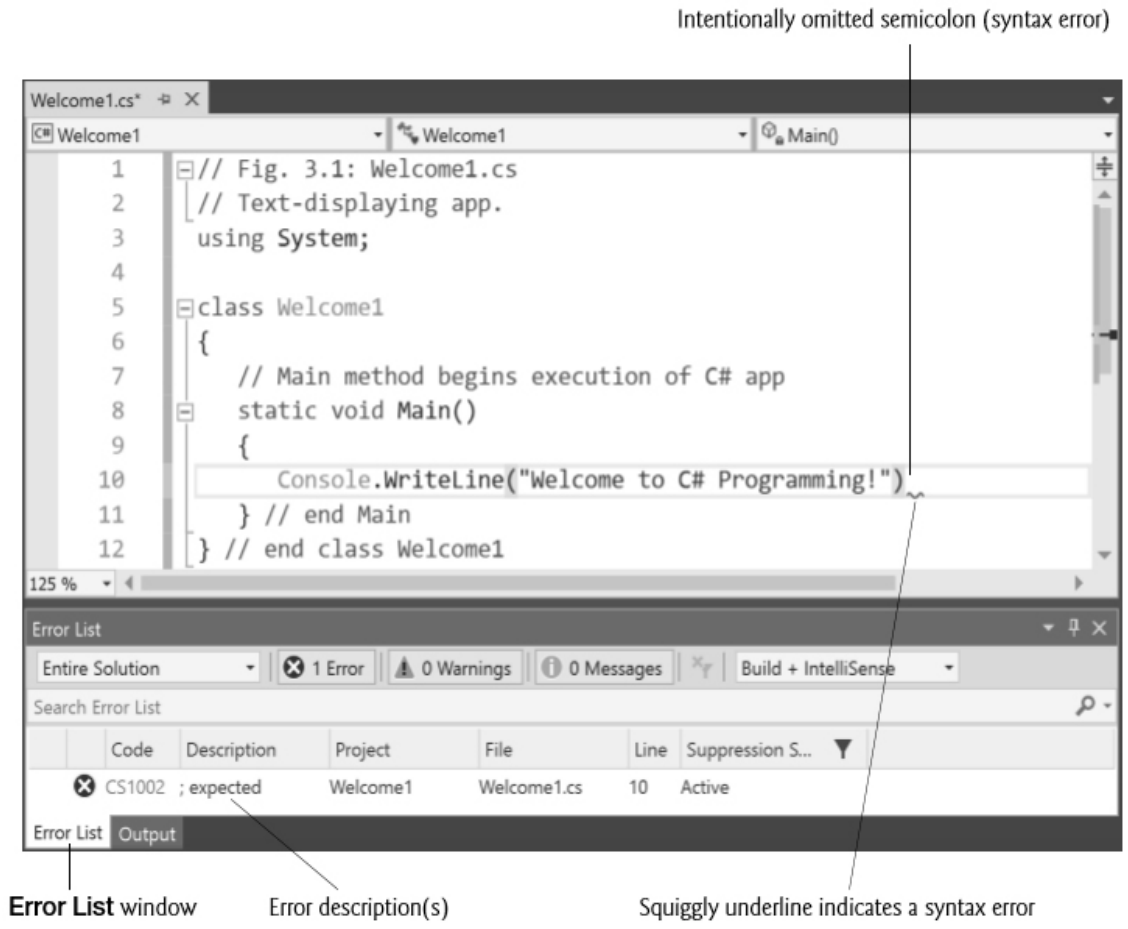
3.3.5 Errors, Error Messages and the Error List Window

- As you type code, the IDE responds either by applying syntax-color highlighting or by generating a **syntax error**.
- A syntax error indicates a violation of Visual C#'s rules for creating correct apps.
- When a syntax error occurs, the IDE underlines the location of the error with a red squiggly line and provides a description of it in the **Error List window** (Figure 3.9).

Error-Prevention Tip 3.5

One compile-time error can lead to multiple entries in the **Error List** window. Each error you correct could eliminate several subsequent error messages when you recompile your app. So when you see an error you know how to fix, correct it—the **IDE** will recompile your code in the background, so fixing an error may make several other errors disappear.

Figure 3.9 Syntax Error Indicated by the IDE



3.4.1 Displaying a Single Line of Text with Multiple Statements

- Class `Welcome2`, shown in Figure 3.10, uses two statements to produce the same output as that shown in the previous example.
- Unlike `WriteLine`, the `Console` class's `Write` method does not position the screen cursor at the beginning of the next line in the console window.

Figure 3.10 Displaying One Line of Text with Multiple Statements

```
1  // Fig. 3.10: Welcome2.cs
2  // Displaying one line of text with multiple statements.
3  using System;
4
5  class Welcome2
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       Console.Write("Welcome to ");
11       Console.WriteLine("C# Programming!");
12    } // end Main
13 } // end class Welcome2
```

```
Welcome to C# Programming!
```

3.4.2 Displaying Multiple Lines of Text with a Single Statement (1 of 2)

- A single statement can display multiple lines by using newline characters.
- Like space characters and tab characters, newline characters are whitespace characters.
- The app of Figure 3.11 outputs four lines of text, using newline characters to indicate when to begin each new line.

Figure 3.11 Displaying Multiple Lines with a Single Statement

```
1  // Fig. 3.11: Welcome3.cs
2  // Displaying multiple lines with a single statement.
3  using System;
4
5  class Welcome3
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       Console.WriteLine("Welcome\nto\nC#\nProgramming!");
11    } // end Main
12 } // end class Welcome3
```

```
Welcome
to
C#
Programming!
```

3.4.2 Displaying Multiple Lines of Text with a Single Statement (2 of 2)

- The **backslash** (\) is called an **escape character**, and is used as the first character in an **escape sequence**.
- The escape sequence `\n` represents the **newline character**.
- Figure 3.12 lists several common escape sequences and describes how they affect the display of characters in the console window.

Figure 3.12 Common Escape Sequences

Escape sequence	Description
`\n`	Newline. Positions the screen cursor at the beginning of the next line.
`\t`	Horizontal tab. Moves the screen cursor to the next tab stop.
`\"`	Double quote. Used to place a double-quote character (") in a string—e,g., `Console.Write("\"in quotes\"");` displays `"in quotes"`.
`\r`	Carriage return. Positions the screen cursor at the beginning of the current line—does not advance the cursor to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
`\\`	Backslash. Used to place a backslash character in a string.

3.5 String Interpolation

- Many programs format data into strings.
- C# 6 introduces a mechanism called **string interpolation** that enables you to insert values in string literals to create formatted strings.
- Figure 3.13 demonstrates this capability.

Figure 3.13 Inserting Content into a `string` with `string` Interpolation

```
1  // Fig. 3.13: Welcome4.cs
2  // Inserting content into a string with string interpolation.
3  using System;
4
5  class Welcome4
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       string person = "Paul"; // variable that stores the string "Paul"
11       Console.WriteLine($"Welcome to C# Programming, {person}!");
12    } // end Main
13 } // end class Welcome4
```

```
Welcome to C# Programming, Paul!
```

3.5 String Interpolation (1 of 2)

- A **variable declaration statement** (also called a **declaration**) specifies the name and type of a variable.
- A variable is a location in the computer's memory where a value can be stored for later use.
- Variables are declared with a name and a type before they're used:
  - A variable's name enables the app to access the corresponding value in
  - A variable's type specifies what kind of information is stored at that location in memory.
- Variables of type string store character-based information.
- Like other statements, declaration statements end with a semicolon (;).

3.5 String Interpolation (2 of 2)

- string interpolation inserts values into a `string`.
- An interpolated `string` must begin with a `$` (dollar sign). Then, you can insert interpolation expressions enclosed in braces, `{}`, anywhere between the quotes ("").
- When C# encounters an interpolated `string`, it replaces each braced interpolation expression with the corresponding value

3.6 Another C# App: Adding Integers

- Applications remember numbers and other data in the computer's memory and access that data through app elements called **variables**.
- A **variable** is a location in the computer's memory where a value can be stored for use later in an app.
- A **variable declaration statement** (also called a **declaration**) specifies the name and type of a variable.
  - A variable's name enables the app to access the value of the variable in memory—the name can be any valid identifier.
  - A variable's type specifies what kind of information is stored at that location in memory.

Figure 3.14 Displaying the Sum of Two Numbers Input from the Keyboard (1 of 2)

```
1  // Fig. 3.14: Addition.cs
2  // Displaying the sum of two numbers input from the keyboard.
3  using System;
4
5  class Addition
6  {
7     // Main method begins execution of C# app
8     static void Main()
9     {
10       int number1; // declare first number to add
11       int number2; // declare second number to add
12       int sum; // declare sum of number1 and number2
13
14       Console.Write("Enter first integer: "); // prompt user
15       // read first number from user
16       number1 = int.Parse(Console.ReadLine());
```

Figure 3.14 Displaying the Sum of Two Numbers Input from the Keyboard (2 of 2)

```
17
18          Console.Write("Enter second integer: "); // prompt user
19          // read second number from user
20          number2 = int.Parse(Console.ReadLine());
21
22          sum = number1 + number2; // add numbers
23
24          Console.WriteLine($"Sum is {sum}"); // display sum
25       } // end Main
26    } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

3.6.1 Declaring the `int` Variable `number1`

- Variables of type **`int`** store **integer** values (whole numbers such as `7`, `-11`, `0` and `31914`).
- Types **`float`**, **`double`** and **`decimal`** specify real numbers (numbers with decimal points).
- Type **`char`** represents a single character.
- These types are called **simple types**. Simple-type names are keywords and must appear in all lowercase letters.

Good Programming Practice 3.6

Declare each variable on a separate line. This format allows a comment to be easily inserted next to each declaration.

Good Programming Practice 3.7

Choosing meaningful variable names helps code to be **self-documenting** (i.e., one can understand the code simply by reading it rather than by reading documentation manuals or viewing an excessive number of comments).

Good Programming Practice 3.8

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter (e.g., **`firstNumber`**). This naming convention is known as camel case.

3.6.3 Prompting the User for Input

- A **prompt** because it directs the user to take a specific action

3.6.4 Reading a Value into Variable `number1` (1 of 2)

- `Console`'s **ReadLine** method waits for the user to type a string of characters at the keyboard and press the **Enter** key.
- `ReadLine` returns the text the user entered.
- The **`int.Parse`** method converts this sequence of characters into data of type `int`.

3.6.4 Reading a Value into Variable `number1` (2 of 2)

- A value can be stored in a variable using the **assignment operator**, **=**.
- Operator = is called a **binary operator**, because it works on two pieces of information, or **operands**.
- An **assignment statement** assigns a value to a variable.
- Everything to the right of the assignment operator, =, is always evaluated before the assignment is performed.

Good Programming Practice 3.9

Place spaces on either side of a binary operator to make the code more readable.

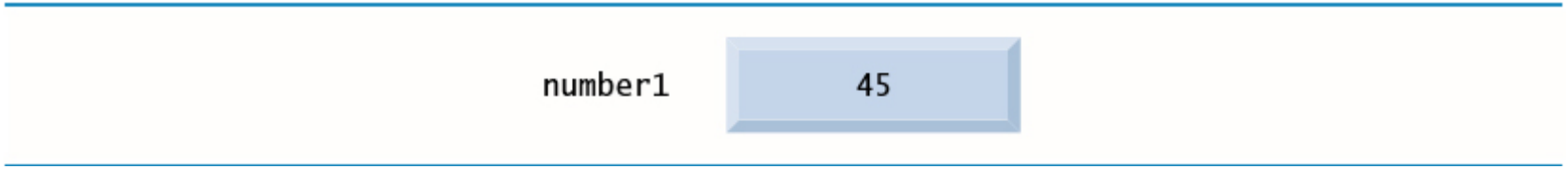
3.6.5 Summing `number1` and `number2`

- An **expression** is any portion of a statement that has a value associated with it.
  - The value of the expression `number1 + number2` is the sum of the numbers.
  - The value of the expression `Console.ReadLine()` is the string of characters typed by the user.
- Calculations can also be performed inside output statements.

3.7 Memory Concepts (1 of 4)

- Variable names actually correspond to **locations** in the computer's memory.
- Every variable has a **name**, a **type**, a **size** and a **value**.
- In Figure 3.15, the computer has placed the value `45` in the memory location corresponding to `number1`.

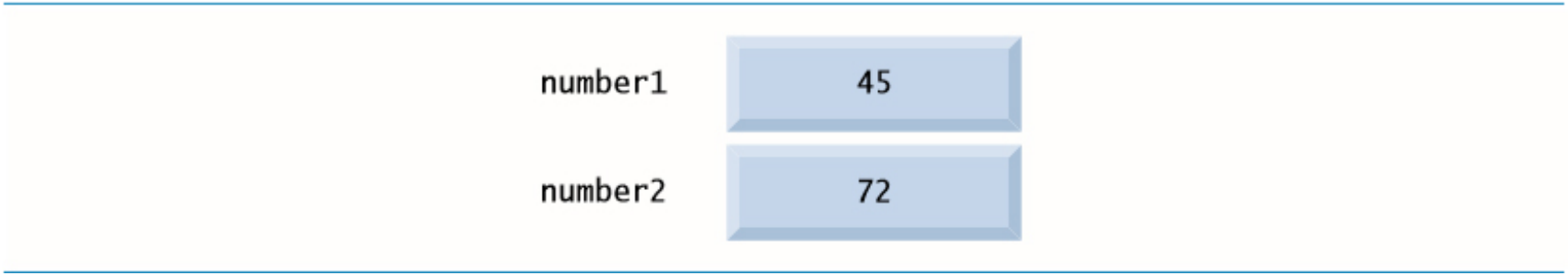
Figure 3.15 Memory Location Showing the Name and Value of Variable `number1`



3.7 Memory Concepts (2 of 4)

- In Figure 3.16, `72` has been placed in location `number2`.

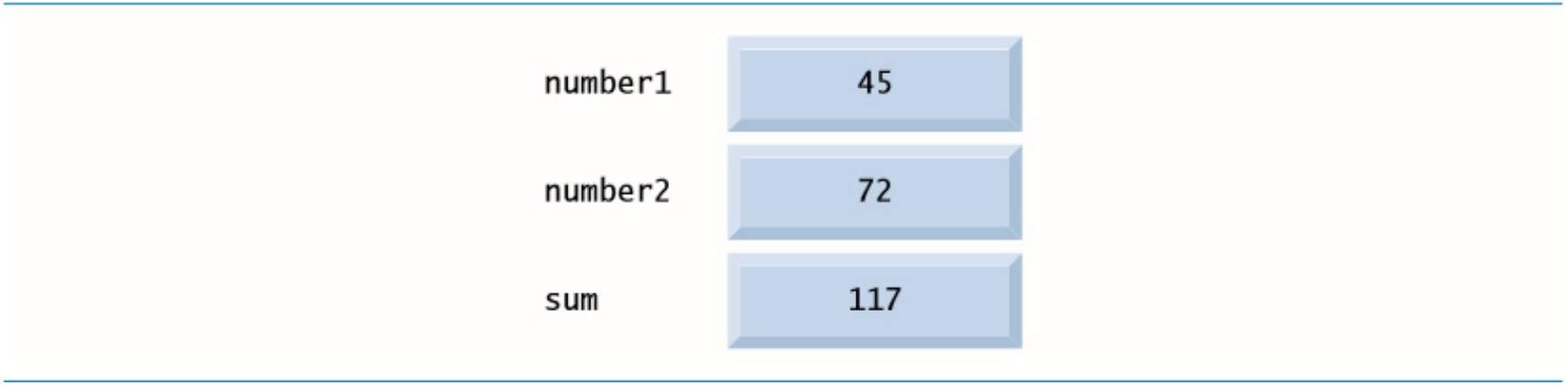
Figure 3.16 Memory Locations After Storing Values for `number1` and `number2`



3.7 Memory Concepts (3 of 4)

- After `sum` has been calculated, memory appears as shown in Figure 3.17.

Figure 3.17 Memory Locations After Calculating and Storing the Sum of `number1` and `number2`



3.7 Memory Concepts (4 of 4)

- Whenever a value is placed in a memory location, the value replaces the previous value in that location, and the previous value is lost.
- When a value is read from a memory location, the process is nondestructive.

3.8 Arithmetic (1 of 4)

- The **arithmetic operators** are summarized in Figure 3.18.
- The arithmetic operators in Figure 3.18 are binary operators.

Figure 3.18 Arithmetic Operators

C# operation	Arithmetic operator	Algebraic expression	C# expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	$b \cdot m$	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

3.8 Arithmetic (2 of 4)

- **Integer division** yields an integer quotient—any fractional part in integer division is simply **truncated** (discarded)—**no rounding** occurs.
- C# provides the remainder operator,%, which yields the remainder after division.
- The remainder operator is most commonly used with integer operands but can also be used with `floats`, `doubles`, and `decimals`.

3.8 Arithmetic (3 of 4)

- Arithmetic expressions must be written in **straight-line form** to facilitate entering an app's code into the computer.
  - Expressions such as "a divided by b" must be written as a / b in a straight line.
- Parentheses are used to group terms in C# expressions in the same manner as in algebraic expressions.
- If an expression contains **nested parentheses**, the expression in the innermost set of parentheses is evaluated first.

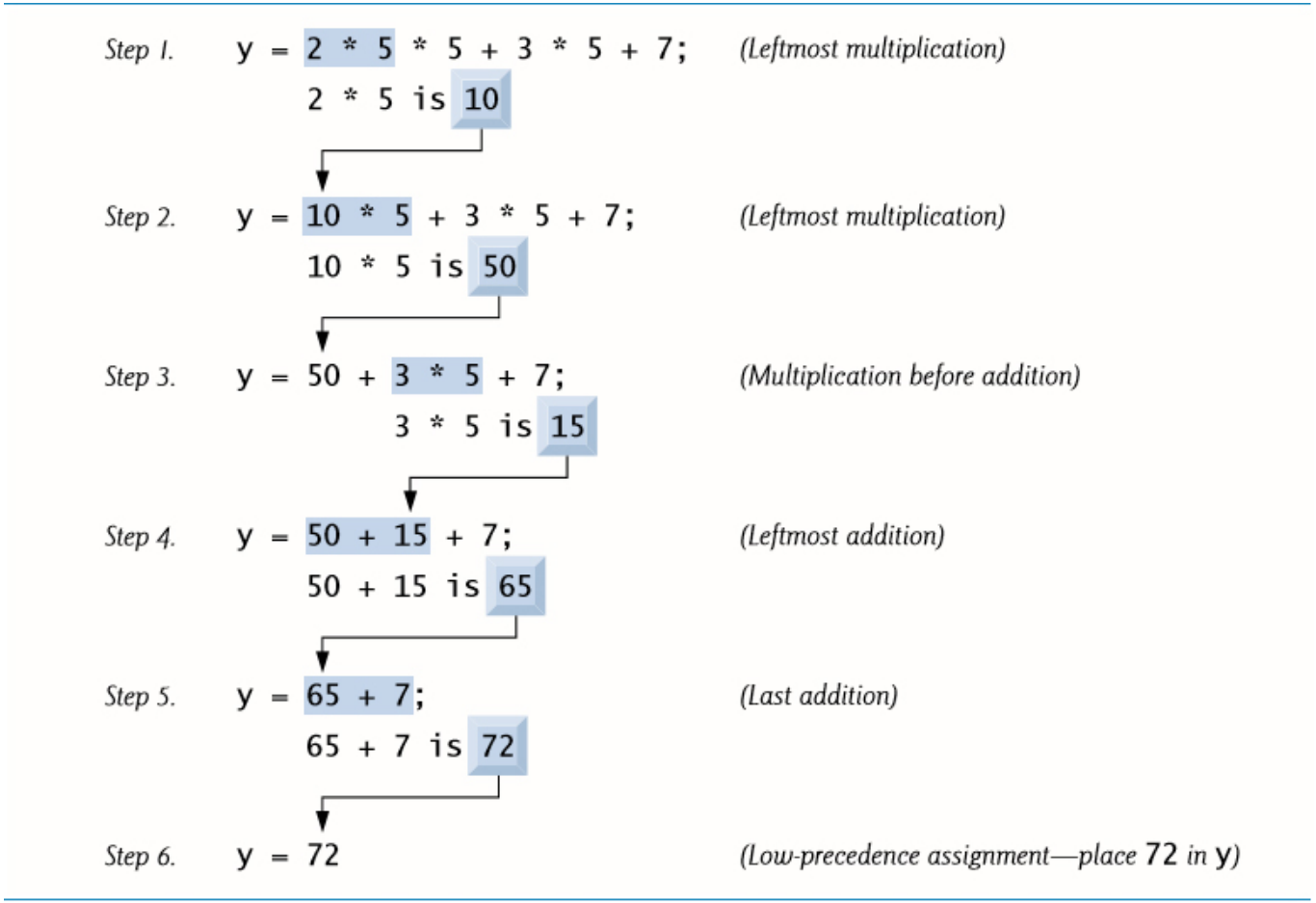
3.8 Arithmetic (4 of 4)

- Arithmetic operators are evaluated according to the **rules of operator precedence**, which are generally the same as those followed in algebra (Figure 3.19).
- As in algebra, it is acceptable to place unnecessary (**redundant**) parentheses in an expression to make the expression clearer.

Figure 3.19 Precedence of Arithmetic Operators

Operators	Operations	Order of evaluation (associativity)
Evaluated first		
*	Multiplication	If there are several operators of this type, they're evaluated from left to right.
/	Division	
%	Remainder	
Evaluated next		
+	Addition	If there are several operators of this type, they're evaluated from left to right.
-	Subtraction	

Figure 3.20 Order in Which a Second-Degree Polynomial is Evaluated



3.9 Decision Making: Equality and Relational Operators (1 of 4)

- A **condition** is an expression that can be either **true** or **false**.
- Conditions can be formed using the **equality operators** (== and !=) and **relational operators** (>, <, >= and <=) summarized in Figure 3.21.

Common Programming Error 3.5

Confusing the equality operator, == , with the assignment operator, =, can cause a logic error or a syntax error. The equality operator should be read as “is equal to,” and the assignment operator should be read as “gets” or “gets the value of.” To avoid confusion, some programmers read the equality operator as “double equals” or “equals equals.”

Figure 3.21 Relational and Equality Operators

Standard algebraic equality and relational operators	C# equality or relational operator	Sample C# condition	Meaning of C# condition
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

3.9 Decision Making: Equality and Relational Operators (2 of 4)

- Figure 3.22 uses six **`if statements`** to compare two integers entered by the user.

Figure 3.22 Comparing Integers Using `if` Statements, Equality Operators and Relational Operators (1 of 4)

```
1  // Fig. 3.22: Comparison.cs
2  // Comparing integers using if statements, equality operators
3  // and relational operators.
4  using System;
5
6  class Comparison
7  {
8     // Main method begins execution of C# app
9     static void Main()
10    {
11       // prompt user and read first number
12       Console.Write("Enter first integer: ");
13       int number1 = int.Parse(Console.ReadLine());
14
15       // prompt user and read second number
16       Console.Write("Enter second integer: ");
17       int number2 = int.Parse(Console.ReadLine());
18
19       if (number1 == number2)
20       {
21          Console.WriteLine($"{number1} == {number2}");
22       }
```

Figure 3.22 Comparing Integers Using `If` Statements, Equality Operators and Relational Operators (2 of 4)

```
23
24          if (number1 != number2)
25          {
26             Console.WriteLine($"{number1} != {number2}");
27          }
28
29          if (number1 < number2)
30          {
31             Console.WriteLine($"{number1} < {number2}");
32          }
33
34          if (number1 > number2)
35          {
36             Console.WriteLine($"{number1} > {number2}");
37          }
38
39          if (number1 <= number2)
40          {
41             Console.WriteLine($"{number1} <= {number2}");
42          }
43
```

Figure 3.22 Comparing Integers Using `If` Statements, Equality Operators and Relational Operators (3 of 4)

```
44          if (number1 >= number2)
45          {
46             Console.WriteLine($"{number1} >= {number2}");
47          }
48       } // end Main
49    } // end class Comparison
```

```
Enter first integer: 42
Enter second integer: 42
42 == 42
42 <= 42
42 >= 42
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

Figure 3.22 Comparing Integers Using `If` Statements, Equality Operators and Relational Operators (4 of 4)

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

3.9 Decision Making: Equality and Relational Operators (3 of 4)

- If the condition in an `if` statement is true, the statement associated with that `if` statement executes.
- An `if` statement always begins with keyword `if`, followed by a condition in parentheses.
- An `if` statement expects one statement in its body.

Good Programming Practice 3.10

Indent the statement(s) in the body of an `if` statement to enhance readability.

Error-Prevention Tip 3.6

You don't need to use braces, { }, around single-statement bodies, but you must include the braces around multiple-statement bodies. You'll see later that forgetting to enclose multiple-statement bodies in braces leads to errors. To avoid errors and make your code more readable, always enclose an `if` statement's body statement(s) in braces, even if it contains only a single statement.

Common Programming Error 3.6

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement now becomes a statement in sequence with the `if` statement and always executes, often causing the program to produce incorrect results.

Good Programming Practice 3.11

A lengthy statement can be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a commaseparated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all lines after the first until the end of the statement.

3.9 Decision Making: Equality and Relational Operators (4 of 4)

- Figure 3.23 shows the precedence of the operators introduced in this chapter from top to bottom in decreasing order of precedence.

Good Programming Practice 3.12

Refer to the operator precedence chart (the complete chart is in Appendix A) when writing expressions containing many operators. Confirm that the operations in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, use parentheses to force the order, as you would do in algebraic expressions. Some programmers also use parentheses to clarify the order. Observe that some operators, such as assignment, =, associate from right to left rather than left to right.

Figure 3.23 Precedence and Associativity of Operations Discussed So Far

Operators	Associativity	Type
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
=	right to left	assignment

Copyright