Visual Programming
Lecture 1
Introduction

By Dr. Aml Abdelazim Mohamed

Department of Computer Science,
Faculty of Computer and Information,
Assiut University

Outline

- .NET Basics.
- What is C#.
- Types of application.
- Creating simple app using Visual Studio 2022
- Error Types.
- Common escape sequences.
- if – Statement
- Conditional operator (?:)

.NET Basics

- The .NET Framework is a framework for developing and implementing software for personal computer, web etc.
- It was designed and is maintained by Microsoft Corporation.
- It came out around the year 2000, even though Microsoft started its development in early 90s.
- .NET has a rich collection of class library (called the Base Class Library) to implement GUI, query, database, web services etc.

.NET Basics

- Programs developed with .NET needs a virtual machine to run on a host. This virtual machine is called Common Language Runtime (CLR).
- Since the compiler doesn’t produce native machine code, and its product is interpreted by the CLR, there’s much security.
- .NET allows using types defined by one .NET language to be used by another under the Common Language Infrastructure (CLI) specification, for the conforming languages.

.NET Basics

- Any language that conforms to the Common Language Infrastructure (CLI) specification of the .NET, can run in the .NET run-time.
- Followings are some .NET languages.
  - Visual Basic
  - C#
  - C++ (CLI version)
  - J# (CLI version of Java)
  - A# (CLI version of ADA)
  - L# (CLI version of LISP)
  - IronRuby (CLI version of RUBY)

.NET Basics (IDE)

- Microsoft provides a comprehensive Integrated
- Development Environment (IDE) for the
- development and testing of software with .NET.
- Some IDEs are as follows
  - Visual Studio
  - Visual Web Developer
  - Visual Basic
  - Visual C#
  - Visual Basic

What is C#

- C# is a general purpose object oriented programming language developed by Microsoft for program development in the .NET Framework.
- It’s supported by .NET’s huge class library that makes development of modern Graphical User Interface applications for personal computers very easy.
- It’s a C-like language and many features resemble those of C++ and Java. For instance, like Java, it too has automatic garbage collection.

What is C#

- It came out around the year 2000 for the .NET platform. Microsoft's Anders Hejlsberg is the principal designer of C#.
- The "#" comes from the musical notation meaning C# is higher than C.
- More information about C#, tutorial, references, support and documentation can be found in the Microsoft Developers Network website.

Types of Application

- The product of the C# compiler is called the “Assembly”. It’s either a “.dll” or a “.exe” file. Both run on the Common Language Runtime and are different from native code that may also end with a “.exe” extension.
- C# has two basic types of application.
  - Windows From Application

    This is GUI based, runs in a window of some sort
  - Console Application

    This application runs in the command prompt

Types

1. Value type
    1. Variable name contains the actual value
    2. int, double and other primitive types
    3. Structure, Enumeration, etc.
2. Reference Type
    1. Variable name contains the reference or pointer to the actual value in memory
    2. Array, derived from class Array
    3. Class, Interface, Delegate, String, etc.

Types

- The value types are derived from System.ValueType
- All types in C# are derived from System.Object which is also accessed by the alias keyword ‘object’
- This type hierarchy is called Common Type System (CTS)

Types

Nullable

- The value types can’t be assigned a null. To enable regular primitive value types to take a null value, C# uses nullable types using ‘?’ with type name. Following example shows how.

```
int? a; a = null; int b; b = a ?? -99;
// the ?? operator picks -99 if null
System.Console.WriteLine("this is null. {0}",b);
a = 23; // not null
System.Console.WriteLine("this is not null. {0}", a ?? -99);
```

Types

- Variables can be defined without an explicit name and to encapsulate a set of values. This is useful for C#’s Language Integrated Query (which will not be discussed in this presentation)

```
var a = 3; // the type is automatically inferred by compiler
var b = new { id = 21, name = "Tito" };
System.Console.WriteLine("a={0}, b.id={1}, b.name={2}", a,
        b.id, b.name);
```

Simple App: Displaying a Line of Text

```
// Text-displaying app.
using System;
class Welcome1
{
        // Main method begins execution of C# app
        static void Main()
        {
                Console.WriteLine("Welcome to C# Programming!");
        } // end Main
} // end class Welcome1
```

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

// is called a **single-line comment**, **/* */ is used for a delimited comment**

```
/* This is a delimited comment.
It can be split over many lines */
```

Simple App: Displaying a Line of Text

using Directive

Line 3

is a **using directive** that tells the compiler where to look for a class that's used in this app. A great strength of Visual C# is its rich set of predefined classes that you can *reuse* rather than "reinventing the wheel." These classes are organized under **namespaces**—named collections of related classes. Collectively, .NET's predefined namespaces are known as **.NET Framework Class Library**. Each using directive identifies a namespace containing classes that a C# app should be able to use. The using directive in line 3 indicates that this example intends to use classes from the System namespace, which contains the predefined Console class (discussed shortly) used in line 10, and many other useful classes.

Class Declaration

Class Declaration

Line 5

begins a **class declaration** for the class named Welcome1. Every app consists of at least one class declaration that's defined by you—the programmer. These are known as **user-defined classes**. The **class keyword** introduces a class declaration and is immediately followed by the **class name** (Welcome1). Keywords (also called **reserved words**) are reserved for use by C#.

Class Name Convention

By convention, all class names begin with a capital letter and capitalize the first letter of each word they include (e.g., SampleClassName). This naming convention is known as **camel case**, because the uppercase letters stand out like a camel's humps. When the first letter is capitalized, it's known as **Pascal Case**. A class name is an **identifier**—a series of characters consisting of letters, digits and underscores (_) that does not begin with a digit and does not contain spaces. Some valid identifiers are Welcome1, identifier, _value and m_inputField1. The name 7button is *not* a valid identifier because it begins with a digit, and the name input field is *not* a valid identifier because it contains a space. Normally, an identifier that does not begin with a capital letter is not the name of a class. C# is **case sensitive**—that is, uppercase and lowercase letters are distinct, so a1 and A1 are different (but both valid) identifiers. Keywords are always spelled with all lowercase letters.

Class Declaration's File Name

For our app, the file name is Welcome1.cs.

Body of a Class Declaration

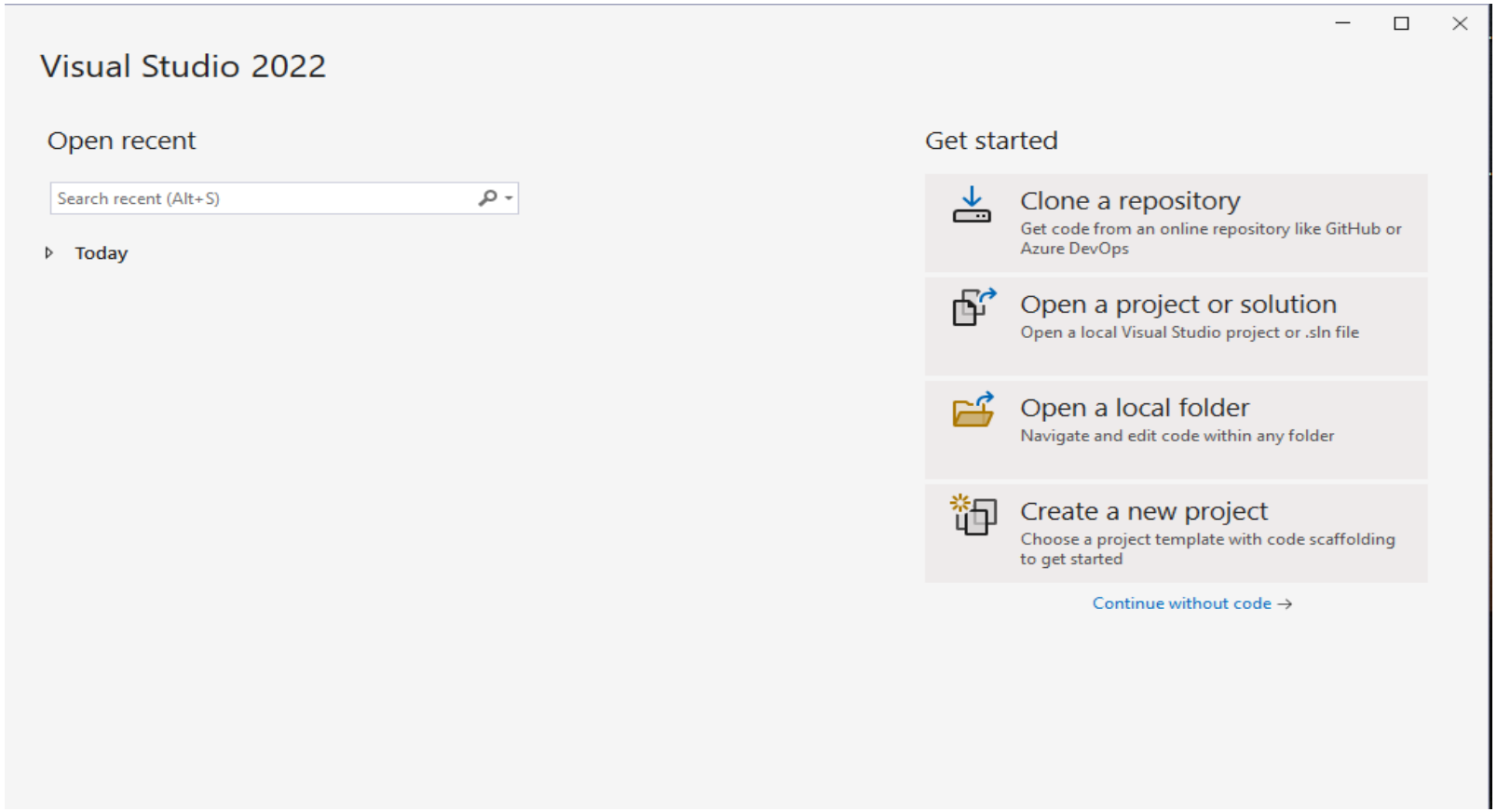
A **left brace**, { (in line 6 in Fig. 3.1), begins each class declaration's **body**. A corresponding **right brace**, } (in line 12), must end each class declaration. Lines 7–11 are indented. This indentation is a *spacing convention*.

Creating a Simple App in Visual Studio

Creating the Console App

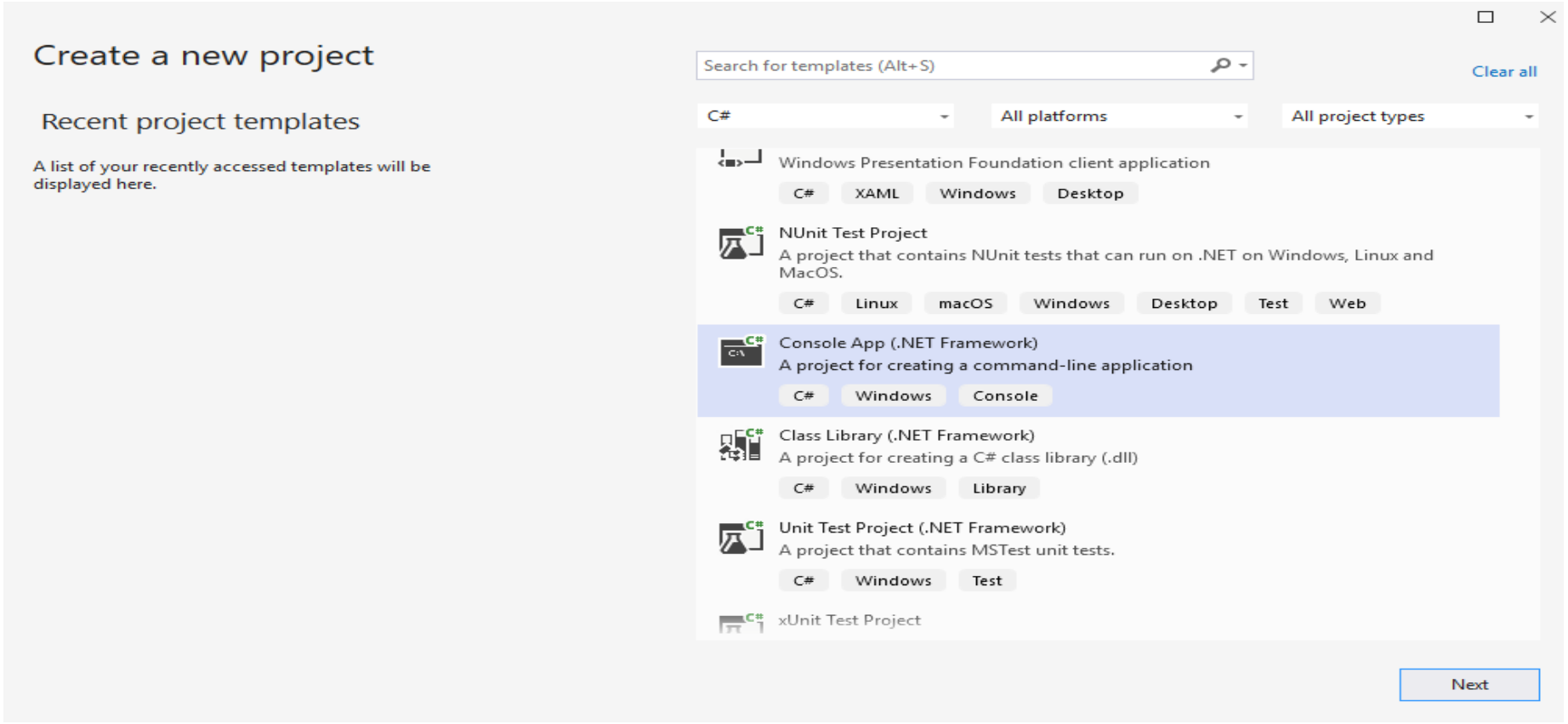
After opening Visual Studio, select **Create a new project**... to display the **Create a New Project** dialog. At the right side of the dialog, select the **Console Application** template. In the dialog's **Name** field, type Welcome1—your project's name—then click **Next** to create the project.

Creating a Simple App in Visual Studio



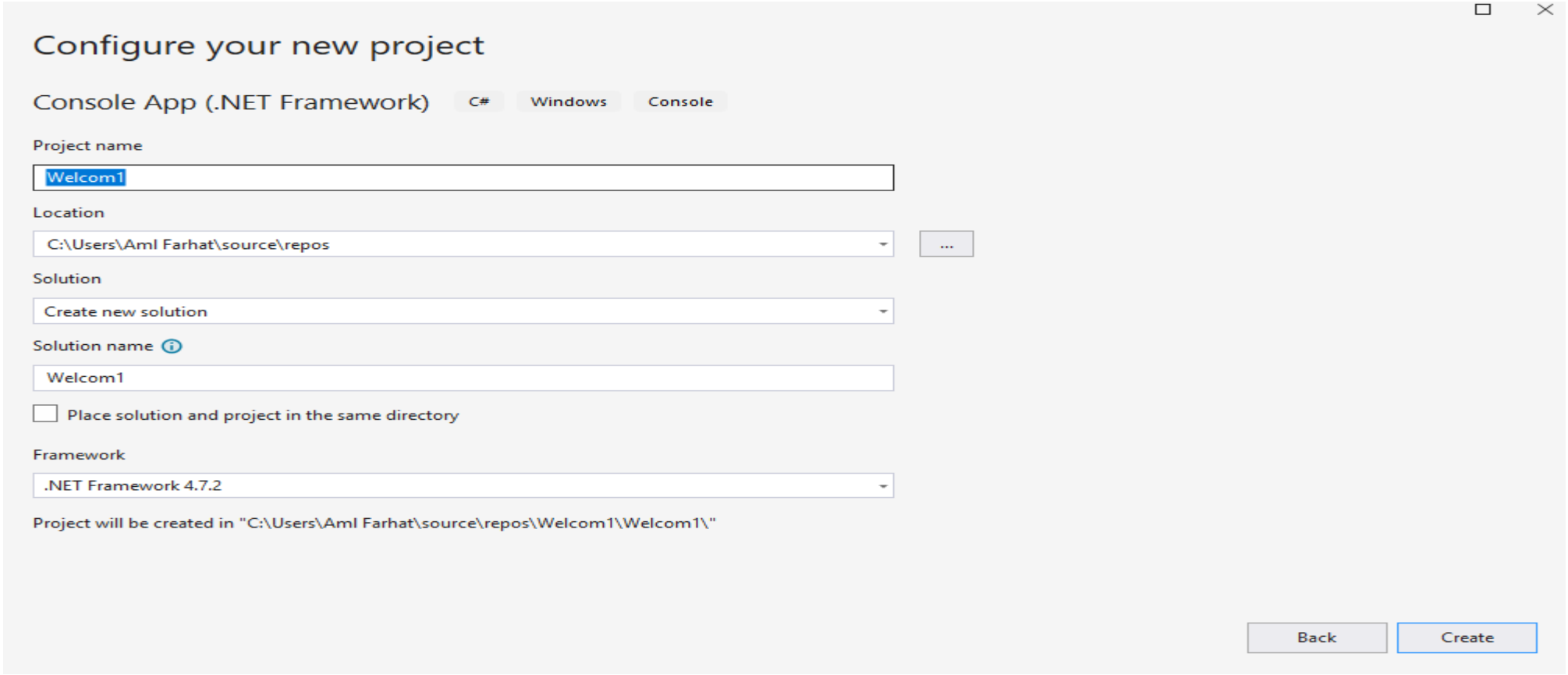
Selecting **Create a new project**.

Creating a Simple App in Visual Studio



Selecting **Console Application(.Net Framework)** in the **Create a new project** dialog.

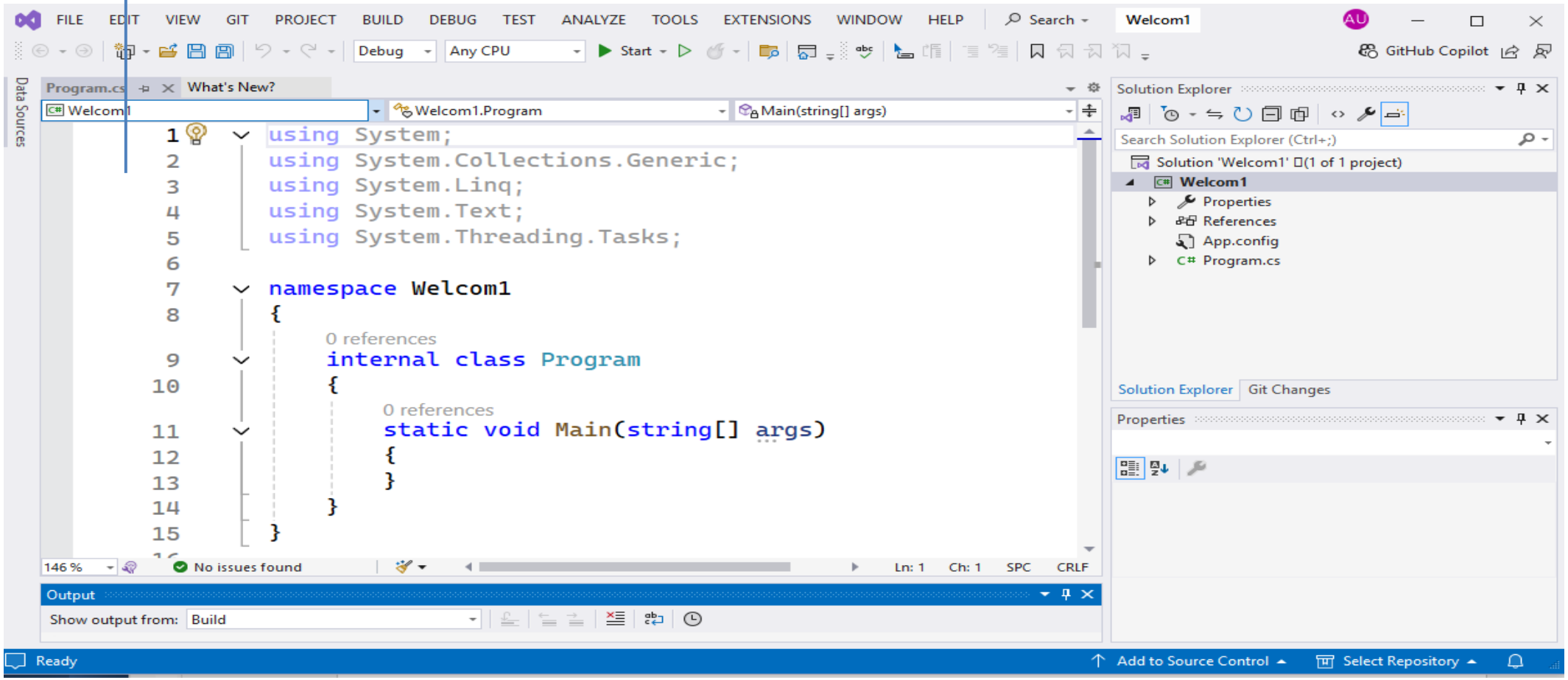
Creating a Simple App in Visual Studio



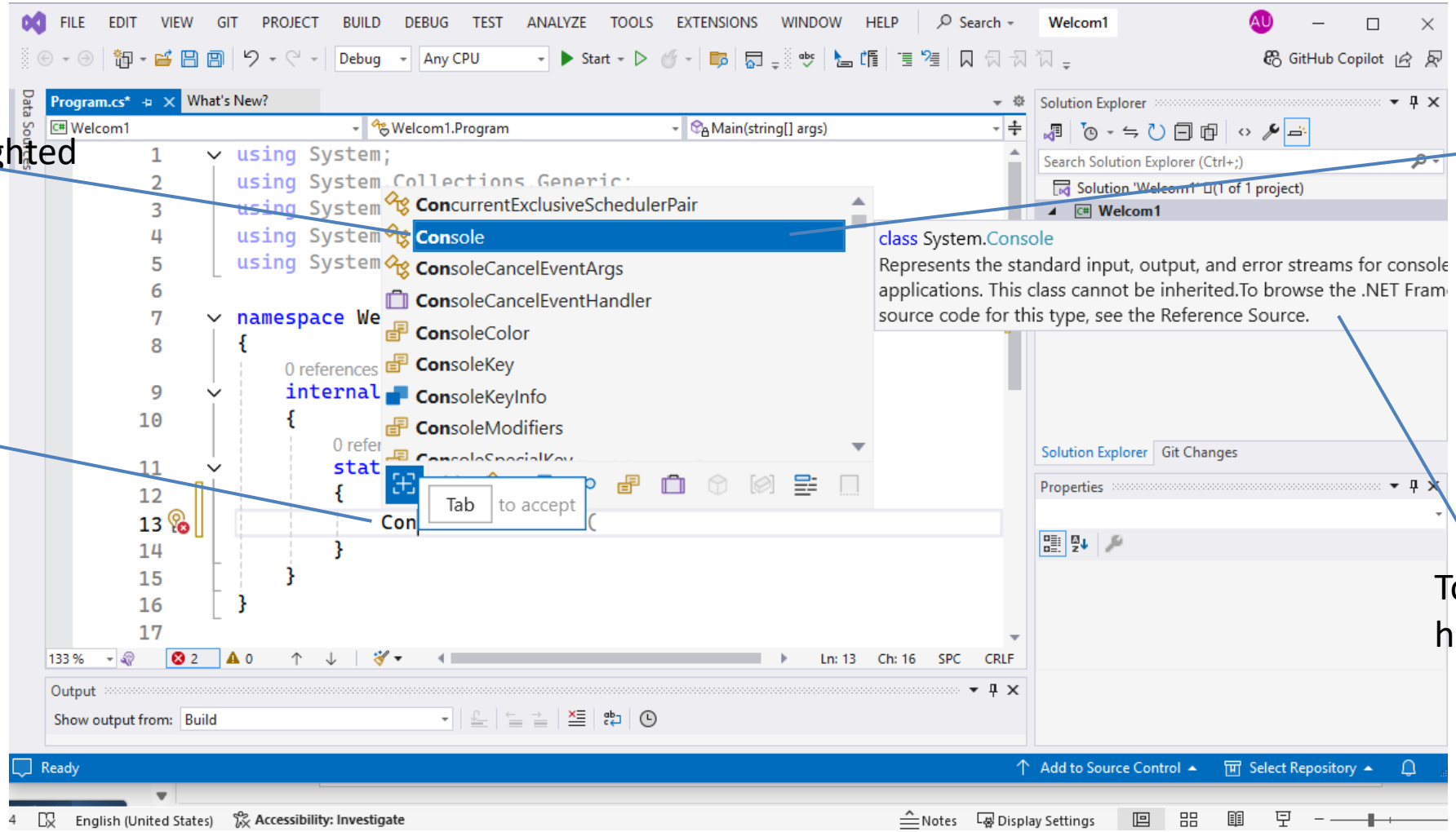
Write **App Name** in the **project Name** then press **Create**.

Creating a Simple App in Visual Studio

Editor window



Writing Code and Using IntelliSense



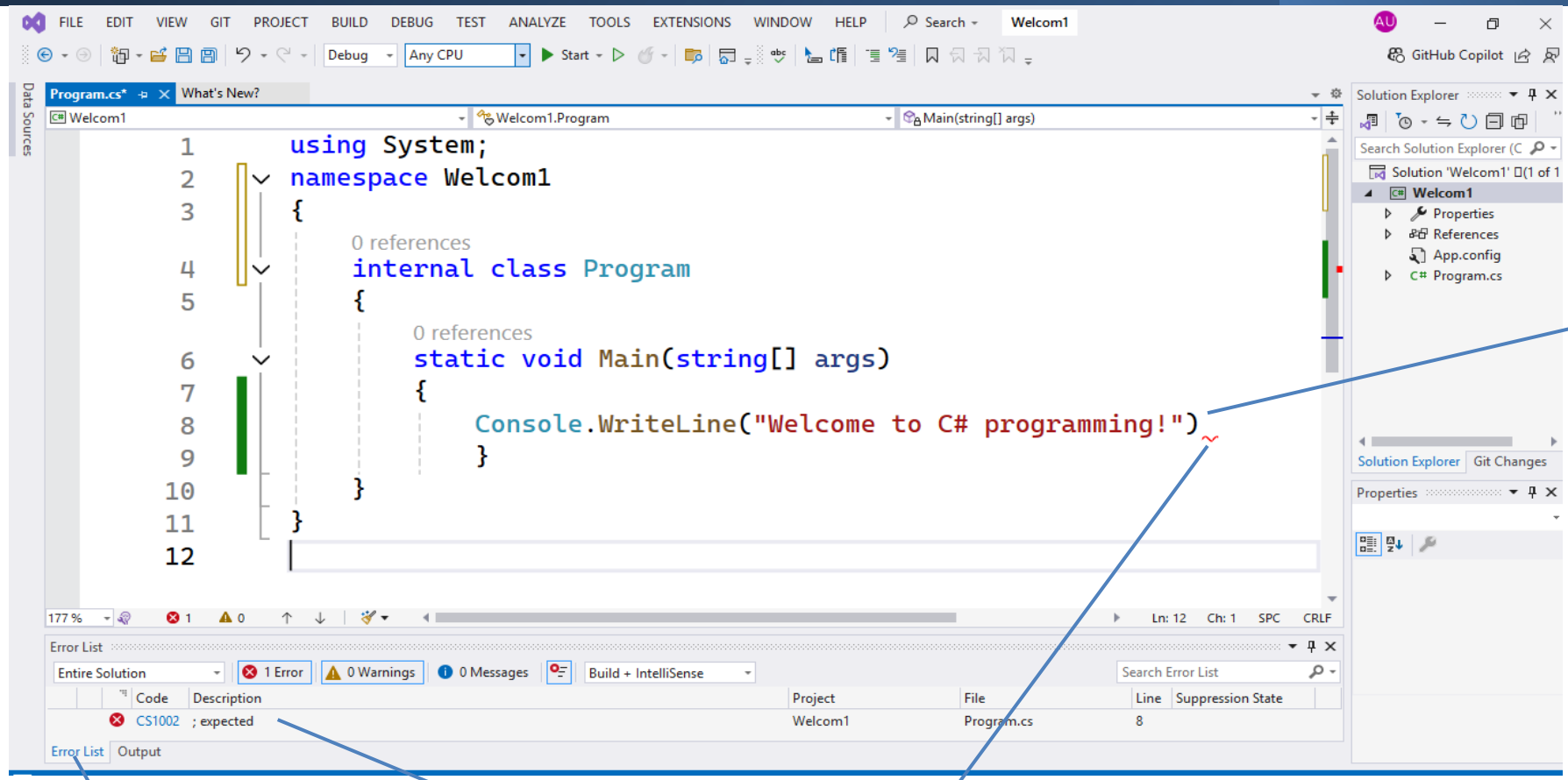
Compiling and Running the App

You're now ready to compile and execute your app. Depending on the project's type, the compiler may compile the code into files with the .exe (**executable**) **extension**, the **.dll**

(**dynamically linked library**) **extension** or one of several other extensions—you can find these files in project's subfolders on disk. Such files are called **assemblies** and are the packaging units for compiled C# code. These assemblies contain the Microsoft Intermediate Language (MSIL; Section 1.9.2) code for the app.

To compile the app, select **Build > Build Solution**. If the app contains no compile-time errors, this will compile your app and build it into an executable file (named Welcome1.exe, in one of the project's subdirectories). To execute it, press *Ctrl* + *F5*, which invokes the Main method

Errors, Error Messages and the Error List Window



Intentional omitted semicolon (syntax error)

Error List window

Error description(s)

Squiggly underline indicates a syntax error

Displaying a Single Line of Text with Multiple Statements

- **Displaying a Single Line of Text with Multiple Statements**

```
Console.Write("Welcome to ");
Console.WriteLine("C# Programming!");
```

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

- **Displaying Multiple Lines of Text with a Single Statement**
  - Console.WriteLine("Welcome\nto\nC#\nProgramming!");

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

- The **backslash** (\) is called an **escape character**.

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.

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.
- An interpolated string must begin with a $ (dollar sign). Then, you can insert **interpolation expressions** enclosed in braces, {}.

```
// Inserting content into a string with string interpolation.
using System;
class Welcome4
 {
// Main method begins execution of C# app
static void Main()
 {
        string person = "Ahmed"; // variable that stores the string "Paul"
        Console.WriteLine($"Welcome to C# Programming, {person}!");
} // end Main
} // end class Welcome4
```

Welcome to C# Programming, Ahmed!

int Interpolation

- **Declaring the int Variable number1**
  - **int** number1;
- **Reading a Value into Variable number1**
  - number1 = **int**.Parse(Console.ReadLine());
- **Displaying the sum with string Interpolation**
  - Console.WriteLine(**$"Sum is** {sum}**"**); // display sum
- **Performing Calculations in Output Statements**
  - Console.WriteLine(**$"Sum is** {number1 + number2}**"**);

Arithmetic Operator

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

Rules of Operator Precedence

- ***Evaluated first:*** ***, /, %**

If there are several operators of this type, they're evaluated from left to right.

- ***Evaluated next:*** **+, -**

If there are several operators of this type, they're evaluated from left to right.

- *Algebra:* $\frac{a+b+c+d+e}{5}$
- *C#:* m = (a + b + c + d + e) / 5;

if- Statement

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

Selection structures

C# has three types of selection structures

- **if statement**
- **if…else statement**
- The **switch** statement

if Single-Selection Statement

If student's grade is greater than or equal to 60 Display "Passed"

```
if (studentGrade >= 60)
{
Console.WriteLine("Passed");
}
```

if...else Double-Selection Statement

If student's grade is greater than or equal to 60
Display "Passed"
Else
Display "Failed"

```
if (grade >= 60)
{
        Console.WriteLine("Passed");
}
else
{
        Console.WriteLine("Failed");
}
```

Nested if…else Statements

```
If student's grade is greater than or equal to 90
        Display "A"
Else
        If student's grade is greater than or equal to 80
                Display "B"
        Else
                If student's grade is greater than or equal to 70
                        Display "C"
                Else
                        If student's grade is greater than or equal to 60
                                Display "D"
                        Else
                                Display "F"
```

Blocks

- The following example includes a block of *multiple* statements in the else part of an if…else statement:

```
if (studentGrade >= 60)
{
        Console.WriteLine("Passed");
}
else
{
        Console.WriteLine("Failed");
        Console.WriteLine("You must take this course again.");
}
```

```
Failed
You must take this course again.
```

- Without the braces surrounding the two statements in the else clause, the statement
- would be outside the body of the else part of the if…else statement and would execute
- *regardless* of whether the grade was less than 60.

Syntax and Logic Errors

- *Syntax errors* (such as when one brace in a block is left out of the program) are caught by the compiler.
- A **logic error** (such as the problem above or an incorrect calculation) has its effect at execution time.
  - A **fatal logic error** causes a program to fail and terminate prematurely.
  - A **nonfatal logic error** allows a program to continue executing but causes it to produce incorrect results.

Exercise_1

(Digits of an Integer) Write an app that inputs one number consisting of five digits from the user, separates the number into its individual digits and displays the digits separated from one another by three spaces each. For example, if the user types 42339, the app should display 4 2 3 3 9

Exercise_1 Solution

```
static void Main()
{
     Console.WriteLine ("Enter one number con
     int x=int.Parse(Console.ReadLine());
    int d=x/10000;
     x=x%10000;
     Console.Write($"{d}  ");
     d=x/1000;
     x=x%1000;
     Console.Write($"{d}  ");
     d=x/100;
     x=x%100;
     Console.Write($"{d}  ");
     d=x/10;
     x=x%10;
     Console.Write($"{d}  ");
     d=x;
     Console.Write($"{d}");
 }
```

```
0 references
static void Main(string[] args)
{
    Console.WriteLine("Enter one number consisting of five digits");
    int x = int.Parse(Console.ReadLine());
    int d = x / 10000;
    x = x % 10000;
    Console.Write($"{d}  ");
    d = x / 1000;
    x = x % 1000;
    Console.Write($"{d}  ");
    d = x / 100;
    x = x % 100;
    Console.Write($"{d}  ");
    d = x / 10;
    x = x % 10;
    Console.Write($"{d}  ");
    d = x;
    Console.Write($"{d}");

}
```

Exercise_2

Identify and correct the errors in each of the following statements:

a)

```
 if (c < 7);
{
Console.WriteLine("c is less than 7");
}
```

b)

```
if (c => 7)
{
Console.WriteLine("c is equal to or greater than 7");
}
```

Conditional Operator (?:)

- C# provides the conditional operator (?:) that can be used in place of an if...else statement.
- This can make your code shorter and clearer. The conditional operator is C#'s only ternary operator—it takes three operands. Together, the operands and the ?: symbols form a conditional expression:
- The first operand (to the left of the ?) is a bool expression that evaluates to true or false.
- The second operand (between the ? and :) is the value of the conditional expression if the bool expression is true.
- The third operand (to the right of the :) is the value of the conditional expression if the bool expression is false.

Conditional Operator (?:)

- For example, the statement:

```
Console.WriteLine(studentGrade >= 60 ? "Passed" : "Failed");
```

- The conditional expression in the preceding statement evaluates to the string "Passed" if the condition

```
studentGrade >= 60
```

- is true and to the string "Failed" if it’s false. Thus, this statement with the conditional operator performs essentially the same task as the first if...else statement
- Converting Between Simple Types Explicitly and Implicitly
- **double** average = (**double**) total / gradeCounter;

Compound Assignment Operators

- +=, -=, *=, /=, %=

Assignment operator	Sample expression	Explanation	Assigns
Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;			
+=	c += 7	c = c + 7	10 to c
-=	d -= 4	d = d - 4	1 to d
*=	e *= 5	e =e* 5	20 to e
/=	f /= 3	f = f / 3	2 to f
%=	g %= 9	g = g % 9	3 to g

Increment and Decrement Operators

Operator	Sample expression	Explanation
++ (prefix increment)	++a	Increments a by 1 and uses the new value of a in the expression in which a resides.
++ (postfix increment)	a++	Increments a by 1, but uses the original value of a in the expression in which a resides.
-- (prefix decrement)	--b	Decrements b by 1 and uses the new value of b in the expression in which b resides.
-- (postfix decrement)	b--	Decrements b by 1 but uses the original value of b in the expression in which b resides.

Prefix Increment vs. Postfix Increment

- The following code demonstrate the difference between the prefix-increment and postfix-increment versions of the ++ increment operator. The decrement operator (--) works similarly.

```
int c = 5; // assign 5 to c
Console.WriteLine($"c before postincrement: {c}"); // displays 5
Console.WriteLine($" postincrementing c: {c++}"); // displays 5
Console.WriteLine($" c after postincrement: {c}"); // displays 6
Console.WriteLine(); // skip a line
// demonstrate prefix increment operator
c = 5; // assign 5 to c
Console.WriteLine($" c before preincrement: {c}"); // displays 5
Console.WriteLine($" preincrementing c: {++c}"); // displays 6
Console.WriteLine($" c after preincrement: {c}"); // displays 6
```

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
    preincrementing c: 6
  c after preincrement: 6
```

Simplifying Increment Statements

- The arithmetic compound assignment operators and the increment and decrement operators can be used to simplify statements.

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

- can be written more concisely with compound assignment operators as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

- And even more concisely with prefix-increment operators as or with postfix-increment operators as

```
++passes ;
++failures;
++studentCounter ;
```

```
passes++ ;
failures++;
studentCounter++ ;
```

Visual Programming
Lecture 2
Control statements

Outline

- while – loop
- for – loop
- do- while loop
- switch statement

while Iteration Statement

- An iteration statement allows you to specify that a program should repeat an action while some condition remains true. The pseudocode statement

```
While there are more items on my shopping list
        Purchase next item and cross it off my list
```

```
int product = 3;
while (product <= 100)
{
        product = 3 * product;
}
```

while Iteration

```
class ClassAverage
 {
 static void Main()
 {
 // initialization phase
 int total = 0; // initialize sum of grades entered by the user
int gradeCounter = 1; // initialize grade # to be entered next
// processing phase uses counter-controlled iteration
 while (gradeCounter <= 10) // loop 10 times
 {
 Console.Write("Enter grade: "); // prompt
 int grade = int.Parse(Console.ReadLine()); // input grade
 total = total + grade; // add the grade to total
gradeCounter = gradeCounter + 1; // increment the counter by 1
 }
```

```
 // termination phase
int average = total / 10; // integer division
yields integer result it must be double and
division by 10.0 // display total and average
of grades
 Console.WriteLine($"\nTotal of all 10 grades
is {total}");
 Console.WriteLine($"Class average is
{average}");
 }
}
```

```
Enter grade: 88
Enter grade: 79
Enter grade: 95
Enter grade: 100
Enter grade: 48
Enter grade: 88
Enter grade: 92
Enter grade: 83
Enter grade: 90
Enter grade: 85

Total of all 10 grades is 848
Class average is 84
```

Braces in a while Statement

- Without the braces, the loop would consider its body to be only the first statement, which adds the grade to the total.

```
while (grade != -1)
total = total + grade; // add grade to total
gradeCounter = gradeCounter + 1; // increment counter
// prompt for input and read grade from user
Console.Write("Enter grade or -1 to quit: ");
grade = int.Parse(Console.ReadLine());
```

Essentials of Counter-Controlled Iteration

1. a **control variable** (or loop counter),
2. the control variable's **initial value**,
3. the control variable's **increment** that's applied during each iteration of the loop,
4. the **loop-continuation condition** that determines if looping should continue.

```
static void Main()
 {
        int counter = 1; // declare and initialize control variable
        while (counter <= 10 ) // loop-continuation condition
         {
                Console.Write($"{counter} ");
                ++counter; // increment control variable
         }
        Console.WriteLine();
}
```

for Iteration Statement

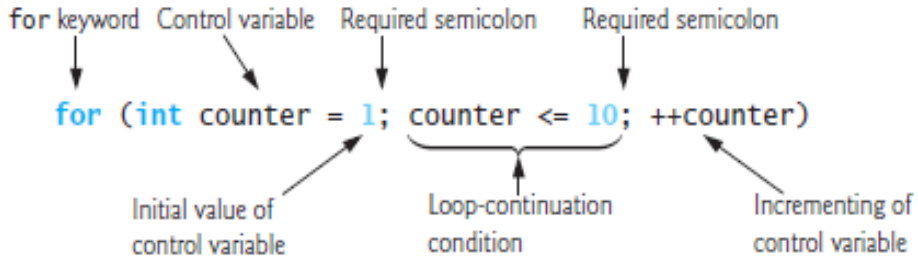
- **for iteration** statement, which specifies the elements of counter-controlled iteration in a single line of code. Typically, **for** statements are used for **counter-controlled iteration**, and **while** statements for **sentinel-controlled iteration**. However, **while** and **for** can each be used for either iteration type.

for Iteration Statement

```
static void Main()
 {
      // for statement header includes initialization,
      // loop-continuation condition and increment
      for (int counter = 1; counter <= 10; ++counter)
      {
      Console.Write($"{counter} ");
      }
      Console.WriteLine();
}
```

General Format of a for Statement

```
for (initialization; loopContinuationCondition; increment)
{
    statement
}
```



Examples Using the for Statement

- a) Vary the control variable from 1 to 100 in increments of 1.

```
for (int i = 1; i <= 100; ++i)
```

- b) Vary the control variable from 100 to 1 in decrements of 1.

```
for (int i = 100; i >= 1; --i)
```

- c) Vary the control variable from 7 to 77 in increments of 7.

```
for (int i = 7; i <= 77; i += 7)
```

- d) Vary the control variable from 20 to 2 in decrements of 2.

```
for (int i = 20; i >= 2; i -= 2)
```

- e) Vary the control variable over the sequence 2, 5, 8, 11, 14, 17, 20.

```
for (int i = 2; i <= 20; i += 3)
```

- f) Vary the control variable over the sequence 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for (int i = 99; i >= 0; i -= 11)
```

Compound-Interest Calculations

- A person invests $1,000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p\,(1 + r)^n$$

where

- p is the original amount invested (i.e., the principal)
- r is the annual interest rate (e.g., use 0.05 for 5%)
- n is the number of years
- a is the amount on deposit at the end of the nth year.

Compound-Interest Calculations

```
static void Main()
{
      decimal principal = 1000; // initial amount before interest
      double rate = 0.05; // interest rate
       // display headers
       Console.WriteLine("Year Amount on deposit");
       // calculate amount on deposit for each of ten years
      // calculate new amount for specified year
      for (int year = 1; year <= 10; ++year)
      {
            decimal amount = principal * ((decimal) Math.Pow(1.0 + rate, year));
            Console.WriteLine($"{year,4}{amount,20:C}");
            // display the year and the amount
      }
}
```

```
Year   Amount on deposit
   1            $1,050.00
   2            $1,102.50
   3            $1,157.63
   4            $1,215.51
   5            $1,276.28
   6            $1,340.10
   7            $1,407.10
   8            $1,477.46
   9            $1,551.33
  10            $1,628.89
```

do…while Iteration Statement

- The **do…while** iteration statement is similar to the while statement. In the while, the app tests the loop-continuation condition at the beginning of the loop, before executing the loop's body. If the condition is false, the body never executes. The **do…while** statement tests the loop-continuation condition after executing the loop's body; therefore, the **body** always executes at **least once**. When a do…while statement terminates, execution continues with the next statement in sequence.

do...while Iteration Statement Example

```
static void Main()
{
      int counter = 1; // initialize counter
     do
     {
          Console.Write($"{counter} ");
          ++counter;
     } while (counter <= 10); // required semicolon

      Console.WriteLine();
 }
```

switch Multiple-Selection Statement

- C# provides the **switch multiple-selection** statement to perform different actions based on the possible values of an expression, known as the **switch expression**. Each action is associated with one or more of the switch expression's possible values. These are specified as **constant integral expressions** or a **constant string expressions**:
  - A constant integral expression is any expression involving character and integer constants that evaluates to an integer value—i.e., values of type sbyte, byte, short, ushort, int, uint, long, ulong and char, or a constant from an enum type (enum is discussed in Section 7.9).
  - A constant string expression is any expression composed of string literals or const string variables that always results in the same string.

Using a switch statement to count letter grades

```
static void Main()
{
        int total = 0; // sum of grades
        int gradeCounter = 0; // number of grades entered
        int aCount = 0; // count of A grades
        int bCount = 0; // count of B grades
        int cCount = 0; // count of C grades
        int dCount = 0; // count of D grades
        int fCount = 0; // count of F grades
        Console.WriteLine("Enter the integer grades in the range 0-100.");
        Console.WriteLine(  "Type <Ctrl> z and press Enter to terminate input:");
        string input = Console.ReadLine(); // read user input
                while (input != null)
                {
                        int grade = int.Parse(input); // read grade off user input
                         total += grade; // add grade to total
                        ++gradeCounter; // increment number of grades
                        input = Console.ReadLine(); // read user input
                 }
```

```
 Console.WriteLine("\nGrade Report:");
switch (grade / 10)
{
                case 9: // grade was in the 90s
                case 10: // grade was 100
                                ++aCount; // increment aCount
                                break; // necessary to exit switch
                case 8: // grade was between 80 and 89
                                ++bCount; // increment bCount
                                break; // exit switch
                case 7: // grade was between 70 and 79
                                ++cCount; // increment cCount
                                break; // exit switch
                case 6: // grade was between 60 and 69
                                ++dCount; // increment dCount
                                break; // exit switch
                default: // grade was less than 60
                                ++fCount; // increment fCount
                                break; // exit switch
}
```

Using a switch statement to count letter grades

```
if (gradeCounter != 0)
{
 // calculate average of all grades entered
 double average = (double) total / gradeCounter;
 // output summary of results
 Console.WriteLine(  $"Total of the {gradeCounter} grades entered is {total}");
Console.WriteLine($"Class average is {average:F}");
Console.WriteLine("Number of students who received each grade:");
 Console.WriteLine($"A: {aCount}"); // display number of A grades
Console.WriteLine($"B: {bCount}"); // display number of B grades
Console.WriteLine($"C: {cCount}"); // display number of C grades
Console.WriteLine($"D: {dCount}"); // display number of D grades
Console.WriteLine($"F: {fCount}"); // display number of F grades
 }
 else // no grades were entered, so output appropriate message
 {
 Console.WriteLine("No grades were entered");
}
```

```
Enter the integer grades in the range 0-100.
Type <Ctrl> z and press Enter to terminate input:
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

break and continue Statements

In addition to selection and iteration statements, C# provides statements **break** and **continue**

to alter the flow of control. break can be used to terminate a switch statement's execution. Also, we can use **break** to terminate any iteration statement.

- **break** Statement

The break statement, when executed in a while, for, do…while, switch, or foreach, causes immediate exit from the loop or switch. Execution continues with the first statement after the control statement.

break Statement example

```
static void Main()
        {
                int count; // control variable also used after loop terminates
                for (count = 1; count <= 10; ++count) // loop 10 times
                {
                        if (count == 5) // if count is 5,
                        {
                                break; // terminate loop
                        }
                        Console.Write($"{count} ");
                }

                Console.WriteLine($"\nBroke out of loop at count = {count}");
        }
```

```
1 2 3 4
Broke out of loop at count = 5
```

continue Statement

The **continue** statement, when executed in a while, for, do…while, or foreach, skips the remaining statements in the loop body and proceeds with the next iteration of the loop. In while and do…while statements, the app evaluates the loop-continuation test immediately after the continue statement executes. In a for statement, the increment expression normally executes next, then the app evaluates the loop-continuation test.

continue Statement example

```
static void Main()
{
        for (int count = 1; count <= 10; ++count) // loop 10 times
        {
                if (count == 5) // if count is 5,
                {
                        continue; // skip remaining code in loop
                }
                Console.Write($"{count} ");
        }
        Console.WriteLine("\nUsed continue to skip displaying 5");
 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip displaying 5
```

Logical Operators

- C# provides **logical operators** to enable you to form more complex conditions by combining simple conditions. The logical operators are
- && (conditional AND),
- || (conditional OR),
- & (boolean logical AND),
- | (boolean logical inclusive OR),
- ^ (boolean logical exclusive OR) and
- ! (logical negation).

Conditional AND (&&) Operator

Suppose that we wish to ensure at some point in an app that two conditions are both true before we choose a certain path of execution. In this case, we can use the && (conditional AND) operator, as follows:

```
if (gender == 'F' && age >= 65)
{
        ++seniorFemales;
}
```

Conditional OR (||) Operator

- Now suppose we wish to ensure that *either or both* of two conditions are true before we choose a certain path of execution. In this case, we use the **|| (conditional OR)** operator, as in the following app segment:

```
if ((semesterAverage >= 90) || (finalExam >= 90))
{
        Console.WriteLine ("Student grade is A");
}
```

Boolean Logical Exclusive OR (^)

A complex condition containing the **boolean logical exclusive OR** (**^**) operator (also called the **logical XOR operator**) is true *if and only if one of its operands is true and the other is false*. If both operands are true or both are false, the entire condition is false.

expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Logical Negation (!) Operator

- The ! (**logical negation** or **not**) operator enables you to "reverse" the meaning of a condition. Unlike the logical operators &&, ||, &, | and ^, which are *binary* operators that combine two conditions, the logical negation operator is a *unary* operator that has only a single condition as an operand. The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is false, as in the code segment

```
if (! (grade == sentinelValue))
{
        Console.WriteLine($"The next grade is {grade}");
}
```

Visual Programming
Lecture 3
Introduction to Classes, Objects, Methods and strings

Definition of Class

- A **class** is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events.
- It defines the data and behavior of a type.
- Supports the OOP principle "encapsulation”.
- Fields and methods can be:
    - Instance (or object members)
    - Static (or class members)

Class Definition and Members

Class definition consists of

- **Class declaration**
- **Data Fields**
- **Constructors**
- **Properties**
- **Methods**

```
class test{
    public int a = 5, b = 10;

    public void mul(){
        Console.Write("The Result of
  Multiply is = {0}", a * b);
        Console.ReadLine();
   }}
```

Classes

- Classes are declared by using the keyword class followed by the class name and a set of class members surrounded by curly braces { }.

- class House
- {
- Body
- …
- }

Basic syntax for class declaration

- Class comprises a class header and class body.

```
class class_name
{
        class body
}
```

- Note: No semicolon (;) to terminate class block.
- But statements must be terminated with a semicolon ;

Visibility/Access Modifiers

- In C#, individual classes and class members need to be prefixed with visibility modifiers.

- By default, if you declare a member variable (or anything else) in a class but do not specify its access level, the member is considered **`private`** and cannot be accessed from outside, i.e. by a non-member of that class.

- Therefore, to make a member accessible by other classes, you must declare it as **`public`**.

Introduction to Object

- Object is an abstraction of real-world entity.
- Syntax for declaring objects is as follows :

- **MyClass objMyClass = new MyClass();**

Characteristics of objects

- It can have its own copy of data members.
- It can be passed to a function like normal variables.
- The members of the class can be accessed by the object using the object to member access operator or dot operator(.).

Calling

- Create Object
- Call function and variables with the help of object

- **MyClass objMyClass = new MyClass();**
- objMycClass.function_name();
- objMycClass.variable;

Constructors

- public MyClass(*parameters*) {…}
- new operator – create an instance of the class
- MyClass objMyClass = new MyClass();

```
public Person()
    {
        name = null;
        age = 0;
    }
```

Static Methods

```
class test
{
   public static int a = 5, b = 10;
   public static void mul()
  {
      Console.Write("The Result of Multiply is = {0}", a * b);
      Console.ReadLine();
  }
}
```

Instance vs Static

- MyClass objMyClass1 = new MyClass();
- objMycClass1.function_name();

- MyClass objMyClass2 = new MyClass();
- objMycClass2.function_name();

- MyClass.function_name();

Account class

A simple Account class that contains a private instance variable name and public methods to Set and Get name's value.

```
class Account
{
        private string name; // instance variable
        // method that sets the account name in the object
        public void SetName(string accountName)
        {
                name = accountName; // store the account name
        }
        // method that retrieves the account name from the object
        public string GetName()
        {
                return name; // returns name's value to this method's caller
        }
}
```

Access Modifiers private and public

The keyword private is an access modifier.

```
private string name; // instance variable
```

Instance variable name is private to indicate that name is accessible only to class Account's methods (and other members, like properties, as you'll see in subsequent examples). This is known as information hiding—the instance variable name is hidden and can be used only in class Account's methods (SetName and GetName). Most instance variables are declared private.

Access Modifiers private and public

Methods (and other class members) that are declared public are "available to the public."

```
public void SetName(string accountName) // instance variable
```

```
public string GetName()
```

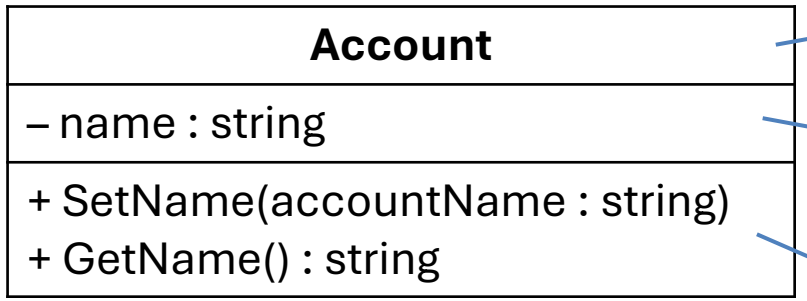
They can be used:

- by methods (and other members) of the class in which they're declared,
- by the class's clients—that is, methods (and other members) of any other classes

(in this app, class AccountTest's Main method is the client of class Account).

Account UML Class Diagram

UML class diagrams often used to summarize a class's *attributes* and *operations*



Top Compartment
contains the *class name*

Middle Compartment
contains the class's attribute names

Bottom Compartment
contains the class's **operations**

Creating, Compiling and Running a Visual C# Project with Two Classes

When you create the project for this app, you should rename Program.cs to Account-Test.cs and add the Account.cs file to the project. To set up a project with two classes:

1. Create a **Console Application** as you did in Chapter 3. We named this chapter's projects Account1, Account2, Account3 and Account4, respectively.
2. Rename the project's Program.cs file to AccountTest.cs. Replace the autogenerated code with class AccountTest's code.
3. Right click the project name in the **Solution Explorer** and select **Add > Class...** from the pop-up menu.
4. In the **Add New Item** dialog's **Name** field, enter the new file's name (Account.cs), then click **Add**. In the new Account.cs file, replace the auto-generated code with class Account's code

Account Class with a Property Rather Than Set and Get Methods

Our first **Account** class contained a **private** instance variable **name** and **public** methods **SetName** and **GetName** that enabled a client to assign to and retrieve from an Account's **name**, respectively. C# provides a more elegant solution—called **properties**—to accomplish the same tasks. A **property** encapsulates a **set accessor** for storing a value into a variable and a **get accessor** for getting the value of a variable.3 In this section, we'll revisit the AccountTest class to demonstrate how to interact with an Account object containing **a public Name property**, then we'll present the updated Account class and take a detailed look at properties.

Account Class with a Property Rather Than Set and Get Methods

```
using System;
class AccountTest
 {
      static void Main()
       {

      Account myAccount = new Account(); // create an Account object and assign it to myAccount
      Console.WriteLine($"Initial name is: {myAccount.Name }"); // display myAccount's initial name
      // prompt for and read the name, then put the name in the object
      Console.Write("Please enter the name: "); // prompt
      string theName = Console.ReadLine(); // read a line of text
      myAccount.Name = theName; // put theName in myAccount's Name
      // display the name stored in object myAccount
      Console.WriteLine($"myAccount's name is: {myAccount.Name}");
      }
}
```

```
Initial name is:
Please enter the name: Jane Green
myAccount's name is: Jane Green
```

Account Class with an Instance Variable and a Property

```
1  // Fig. 4.6: Account.cs
2  // Account class that replaces public methods SetName
3  // and GetName with a public Name property.
4
5  class Account
6  {
7     private string name; // instance variable
8
9     // property to get and set the name instance variable
10    public string Name
11    {
12       get // returns the corresponding instance variable's value
13       {
14          return name; // returns the value of name to the client code
15       }
16       set // assigns a new value to the corresponding instance variable
17       {
18          name = value; // value is implicitly declared and initialized
19       }
20    }
21 }
```

Fig. 4.6 | Account class that replaces public methods SetName and GetName with a public Name property.

Account class that replaces public methods SetName and GetName with a public Name property.

Auto-Implemented Properties

- We created an Account class with a private name instance variable and a public property Name to enable client code to access the name. With an auto-implemented property, the C# compiler automatically creates a *hidden* private instance variable, and the get and set accessors for *getting* and *setting* that hidden instance variable. This enables you to implement the property trivially, which is handy when you're first designing a class.

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

Initializing Objects with Constructors

```
1  // Fig. 4.8: Account.cs
2  // Account class with a constructor that initializes an Account's name.
3
4  class Account
5  {
6     public string Name { get; set; } // auto-implemented property
7
8     // constructor sets the Name property to parameter accountName's value
9     public Account(string accountName) // constructor name is class name
10    {
11       Name = accountName;
12    }
13 }
```

Fig. 4.8 | Account class with a constructor that initializes an Account's name.

```
Account account1 = new Account("Jane Green");
```

Class AccountTest: Initializing Account Objects When They're Created

```
1  // Fig. 4.9: AccountTest.cs
2  // Using the Account constructor to set an Account's name
3  // when an Account object is created.
4  using System;
5
6  class AccountTest
7  {
8     static void Main()
9     {
10       // create two Account objects
11       Account account1 = new Account("Jane Green");
12       Account account2 = new Account("John Blue");
13
14       // display initial value of name for each Account
15       Console.WriteLine($"account1 name is: {account1.Name}");
16       Console.WriteLine($"account2 name is: {account2.Name}");
17    }
18 }
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 4.9 | Using the `Account` constructor to set an `Account`'s name when an `Account` object is created. (Part 2 of 2.)

Default Constructor

```
Account myAccount = new Account();
```

Recall that line used new to create an Account object. The *empty* parentheses in the expression indicate a call to the class's **default constructor**—in any class that does *not* explicitly declare a constructor, the compiler provides a public default constructor (which always has no parameters). When a class has only the default constructor, the class's instance variables are initialized to their *default values*:

- 0 for numeric simple types,
- false for simple type bool and
- null for all other types.

There's No Default Constructor in a Class That Declares a Constructor

If you declare one or more constructors for a class, the compiler will *not* create a *default constructor* for that class. In that case, you will not be able to create an Account object with the expression **new Account()**

Adding the Constructor to Class Account's UML Class Diagram

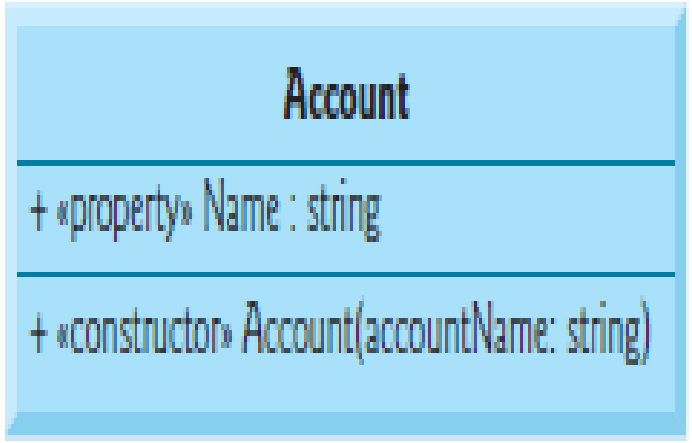


Fig. 4.10 | UML class diagram for Account class of Fig. 4.8.

Account Class with a Balance; Processing Monetary Amounts

we'll declare an Account class that maintains an Account's *balance* in addition to its name. Most account balances are not whole numbers (such as 0, –22 and 1024), rather they're numbers that include a decimal point, such as 99.99 or –20.15. For this reason, class Account represents the account balance using type **decimal**, which is designed to precisely represent numbers with decimal points, especially *monetary amounts*.

Account Class with a decimal balance Instance Variable

```
1   // Fig. 4.13: Account.cs
2   // Account class with a balance and a Deposit method.
3
4   class Account
5   {
6     public string Name { get; set; } // auto-implemented property
7     private decimal balance; // instance variable
8
9     // Account constructor that receives two parameters
10    public Account(string accountName, decimal initialBalance)
11    {
12       Name = accountName;
13       Balance = initialBalance; // Balance's set accessor validates
14    }
15
16    // Balance property with validation
17    public decimal Balance
18    {
19       get
20       {
21          return balance;
22       }
23       private set // can be used only within the class
24       {
25          // validate that the balance is greater than 0.0; if it's not,
26          // instance variable balance keeps its prior value
27          if (value > 0.0m) // m indicates that 0.0 is a decimal literal
28          {
29             balance = value;
30          }
31       }
32    }
33
```

Account Class with a decimal balance Instance Variable

```
34      // method that deposits (adds) only a valid amount to the balance
35      public void Deposit(decimal depositAmount)
36      {
37         if (depositAmount > 0.0m) // if the depositAmount is valid
38         {
39            Balance = Balance + depositAmount; // add it to the balance
40         }
41      }
42   }
```

Fig. 4.11 | Account class with a `decimal` instance variable `balance` and a `Balance` property and `Deposit` method that each perform validation. (Part 2 of 2.)

AccountTest Class That Uses Account Objects with Balances

```
1  // Fig. 4.12: AccountTest.cs
2  // Reading and writing monetary amounts with Account objects.
3  using System;
4
5  class AccountTest
6  {
7     static void Main()
8     {
9        Account account1 = new Account("Jane Green", 50.00m);
10       Account account2 = new Account("John Blue", -7.53m);
11
12       // display initial balance of each object
13       Console.WriteLine(
14          $"{account1.Name}'s balance: {account1.Balance:C}");
15       Console.WriteLine(
16          $"{account2.Name}'s balance: {account2.Balance:C}");
17
18       // prompt for then read input
19       Console.Write("\nEnter deposit amount for account1: ");
20       decimal depositAmount = decimal.Parse(Console.ReadLine());
21       Console.WriteLine(
22          $"adding {depositAmount:C} to account1 balance\n");
23       account1.Deposit(depositAmount); // add to account1's balance
24
```

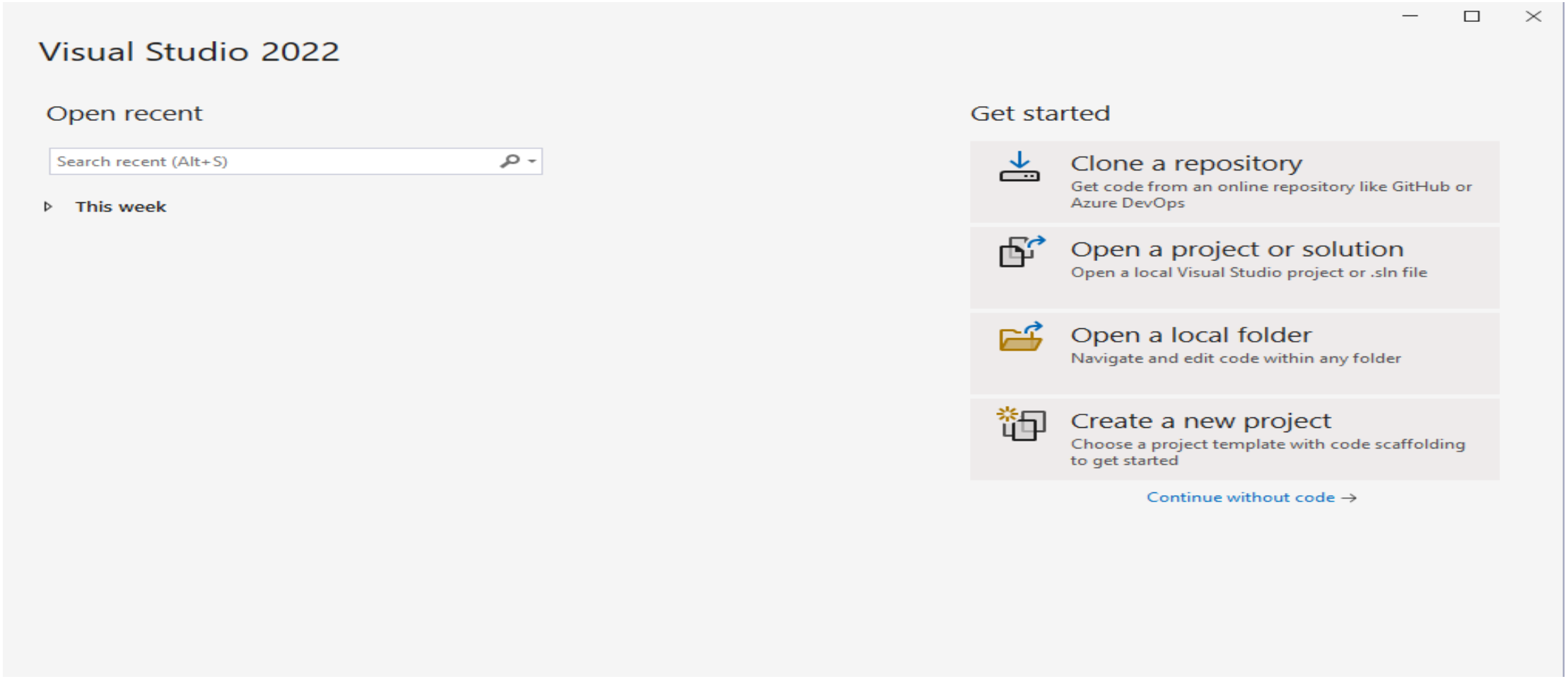
AccountTest Class That Uses Account Objects with Balances

```
25          // display balances
26          Console.WriteLine(
27             $"{account1.Name}'s balance: {account1.Balance:C}");
28          Console.WriteLine(
29             $"{account2.Name}'s balance: {account2.Balance:C}");
30
31          // prompt for then read input
32          Console.Write("\nEnter deposit amount for account2: ");
33          depositAmount = decimal.Parse(Console.ReadLine());
34          Console.WriteLine(
35             $"adding {depositAmount:C} to account2 balance\n");
36          account2.Deposit(depositAmount); // add to account2's balance
37
38          // display balances
39          Console.WriteLine(
40             $"{account1.Name}'s balance: {account1.Balance:C}");
41          Console.WriteLine(
42             $"{account2.Name}'s balance: {account2.Balance:C}");
43       }
44    }
```

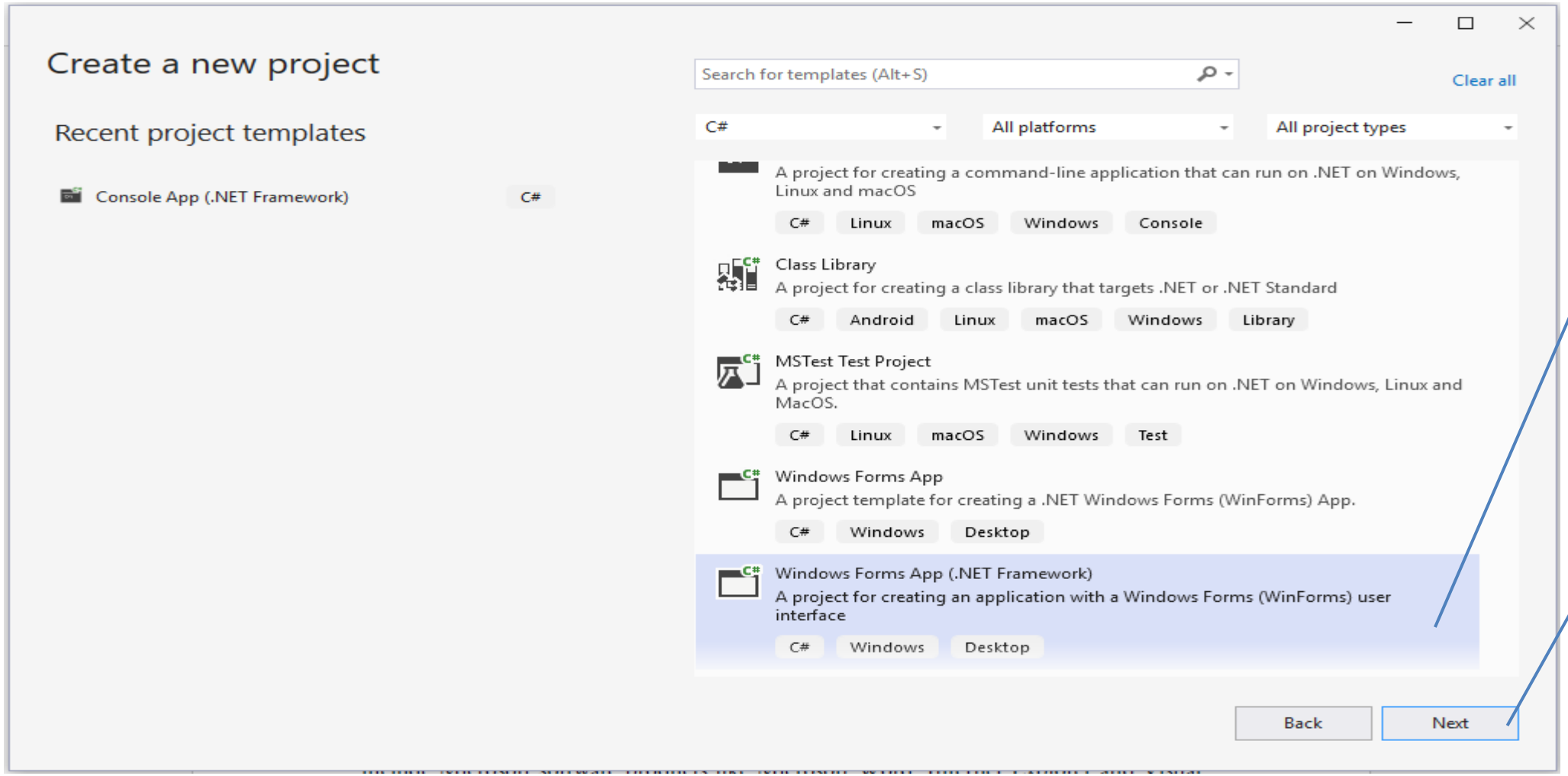
```
Jane Green's balance: $50.00
John Blue's balance: $0.00

Enter deposit amount for account1: 25.53
adding $25.53 to account1 balance

Jane Green's balance: $75.53
John Blue's balance: $0.00

Enter deposit amount for account2: 123.45
adding $123.45 to account2 balance

Jane Green's balance: $75.53
John Blue's balance: $123.45
```

Visual Programming
Lecture 4
Introduction to Visual Studio 2022 and Visual Programming

Links on the Start Page



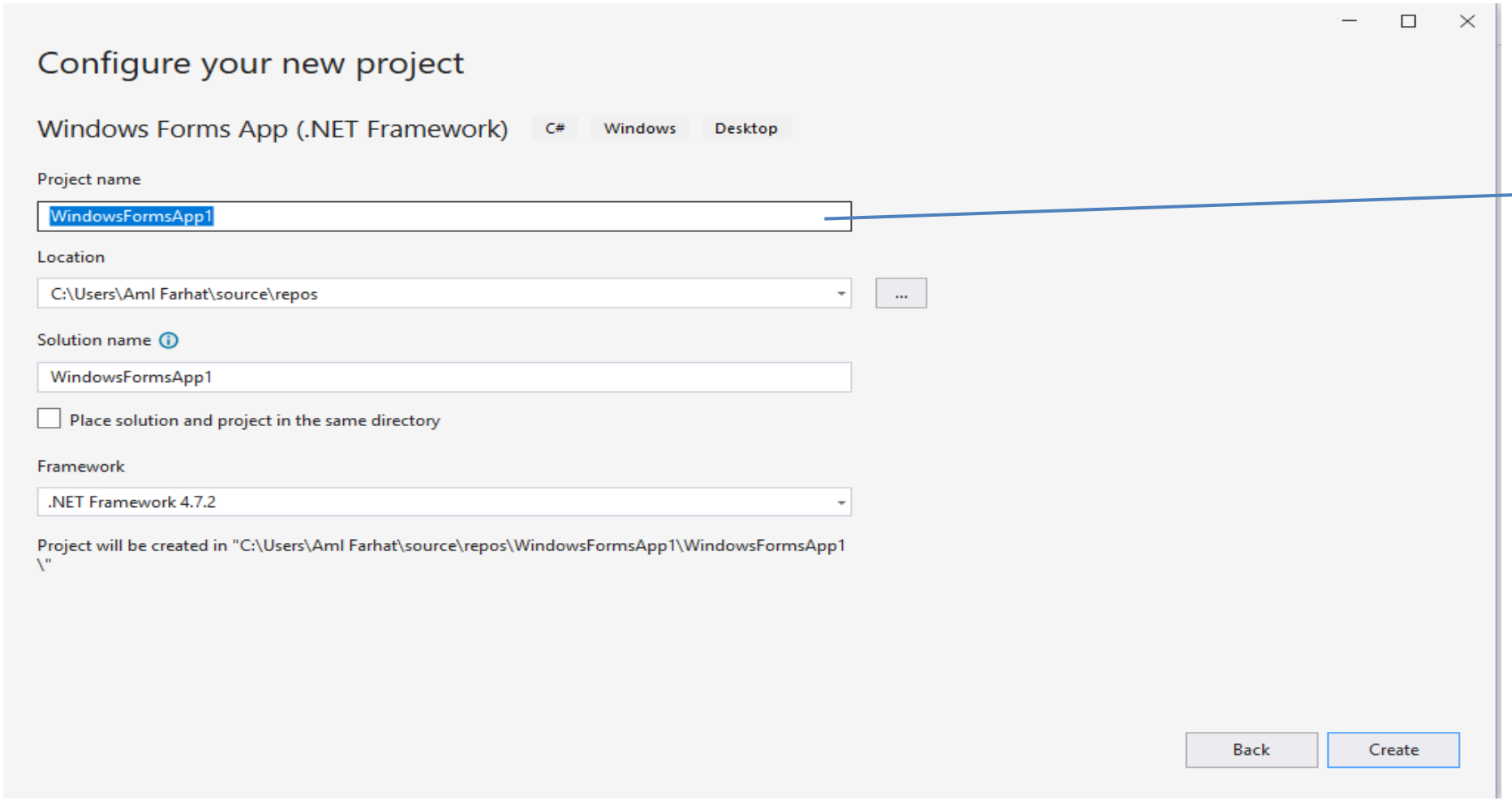
Create a new project Dialog and Project Templates



Visual C# **Windows Forms Application** (selected)

Press next

Configure your new project dialog.



Default project name (provided by Visual Studio)

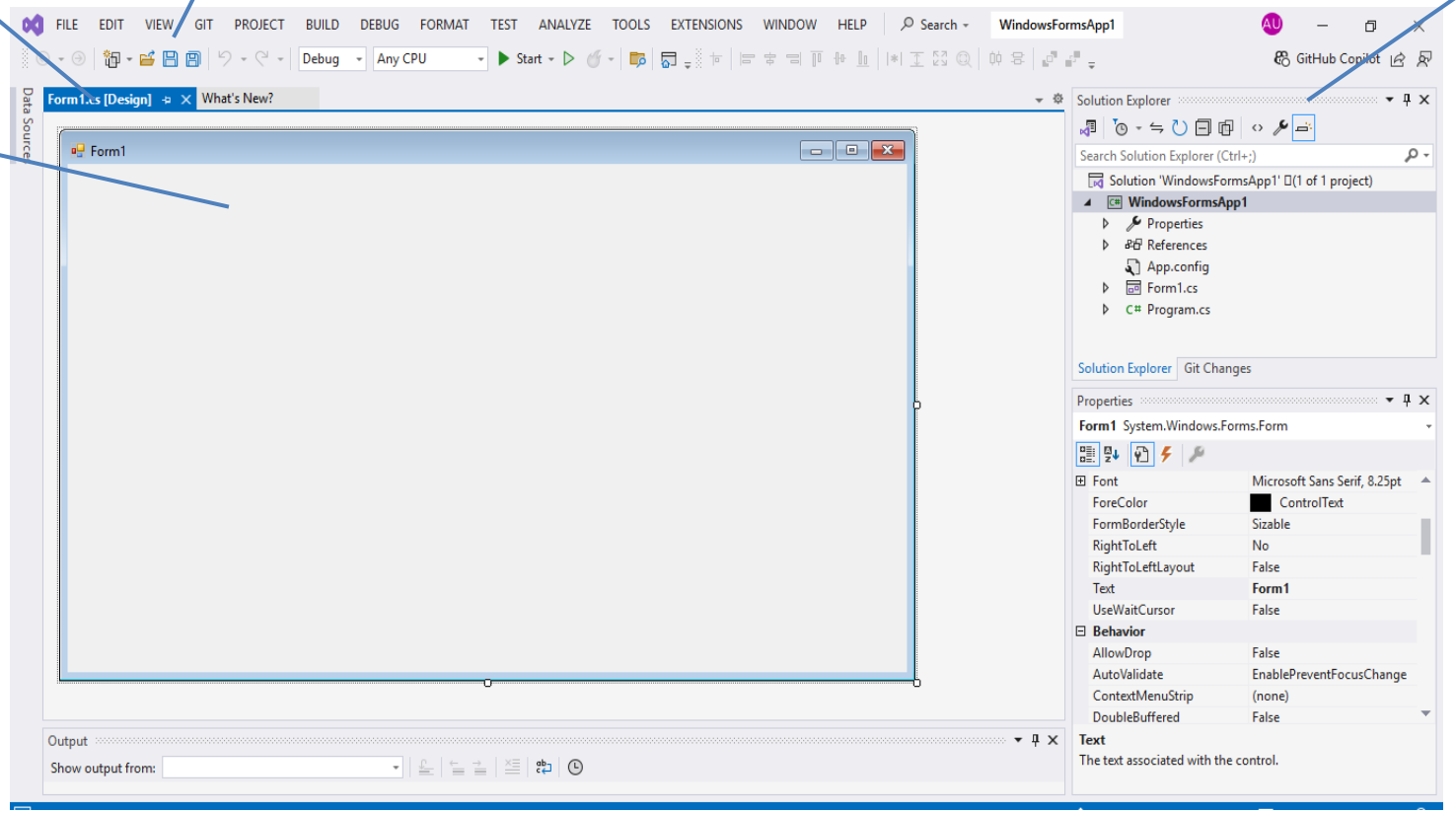
Design view of the IDE.

Active tab

Menu in the menu bar

Form

Solution Explorer window



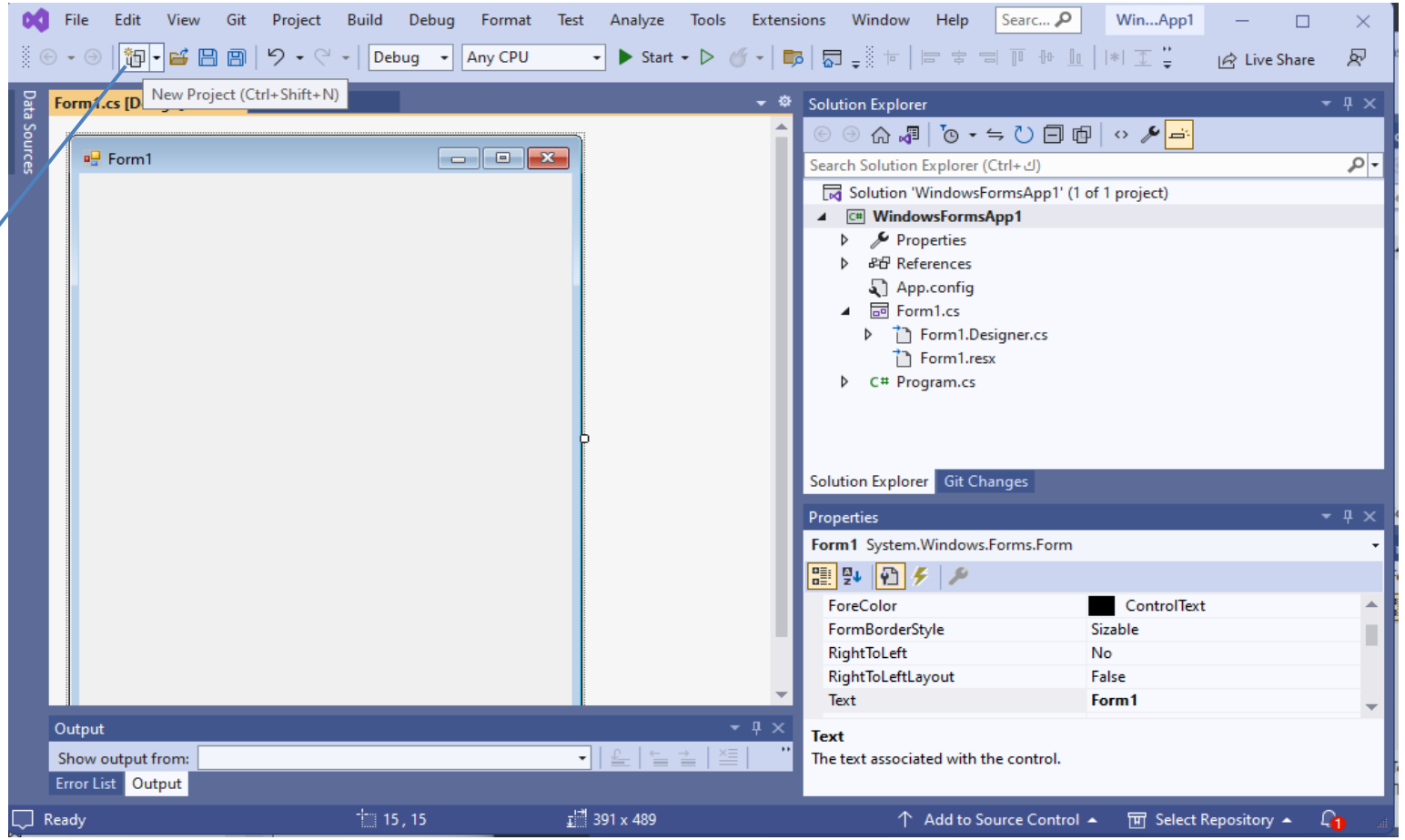
Forms and Controls

Menu	Contains commands for
File	Opening, closing, adding and saving projects, as well as printing project data and exiting Visual Studio.
Edit	Editing apps, such as cut, copy, paste, undo, redo, delete, find and select.
View	Displaying IDE windows (for example, **Solution Explorer**, **Toolbox**, **Properties** window) and for adding toolbars to the IDE.
Project	Managing projects and their files.
Build	Turning your app into an executable program.
Debug	Compiling, debugging (that is, identifying and correcting problems in apps) and running apps.
Team	Connecting to a Team Foundation Server—used by development teams that typically have multiple people working on the same app.
Format	Arranging and modifying a `Form`'s controls. The **Format** menu appears *only* when a GUI component is selected in **Design** view.
Tools	Accessing additional IDE tools and options for customizing the IDE.
Test	Performing various types of automated testing on your app.
Analyze	Locating and reporting violations of the .NET Framework Design Guidelines (`https://msdn.microsoft.com/library/ms229042`).

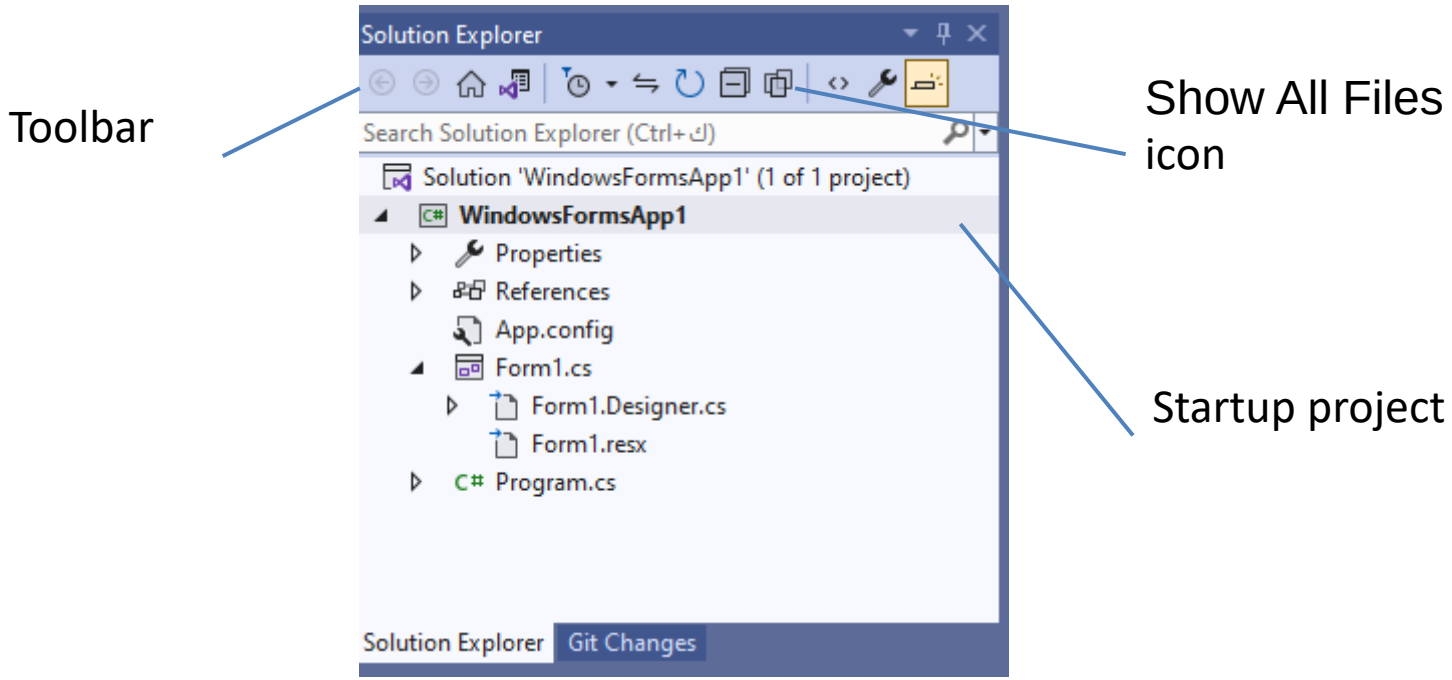
Fig. 2.6 | Summary of Visual Studio menus that are displayed when a `Form` is in **Design**

Tool tip for the New Project button

Tool tip appears when you place the mouse pointer on an icon



Solution Explorer

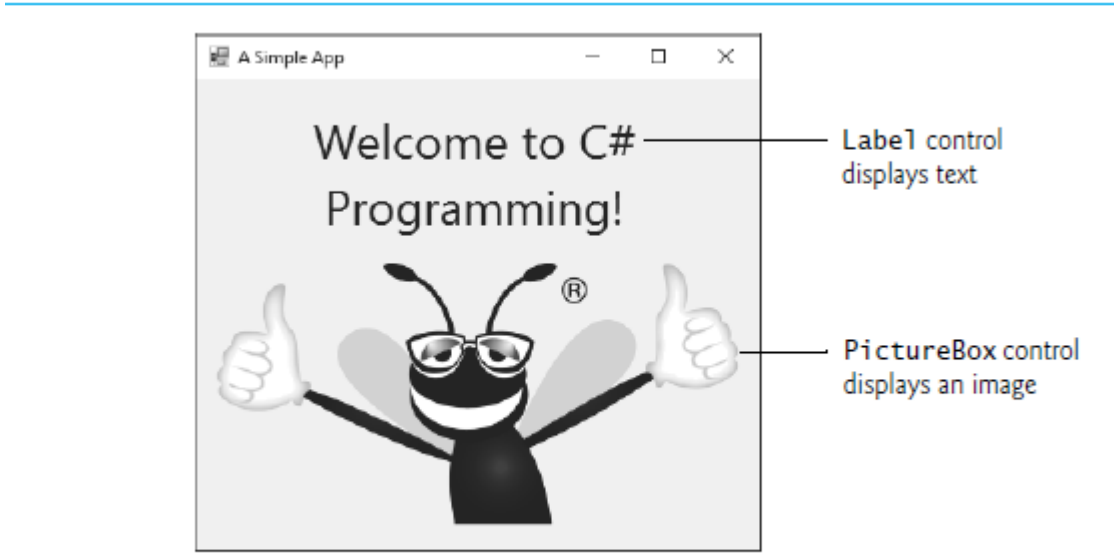


Properties window.

Fig. 2.16 | Properties window.

Creating a Simple App that Displays Text and an Image

In this example, you won't write any C# code. Instead, you'll use visual app-development techniques.



Simple app executing.

Creating a Simple App that Displays Text and an Image

Visual app development is useful for building GUI-intensive apps that require a significant amount of user interaction. To create, save, run and terminate this first app, perform the following steps.

Step 1: Closing the Open Project

If the project you were working with earlier in this chapter is still open, close it by selecting **File > Close Solution**.

Step 2: Creating the New Project

To create a new Windows Forms app:

1. Select **File > New > Project...** to display the Create a **New Project** dialog.
2. Select **Windows Forms Application**. Name the project **ASimpleApp**, specify the **Location**

where you want to save it and click **OK**. We stored the app in the IDE's default location—in your user account's Documents folder under the Visual Studio 2022\Projects.

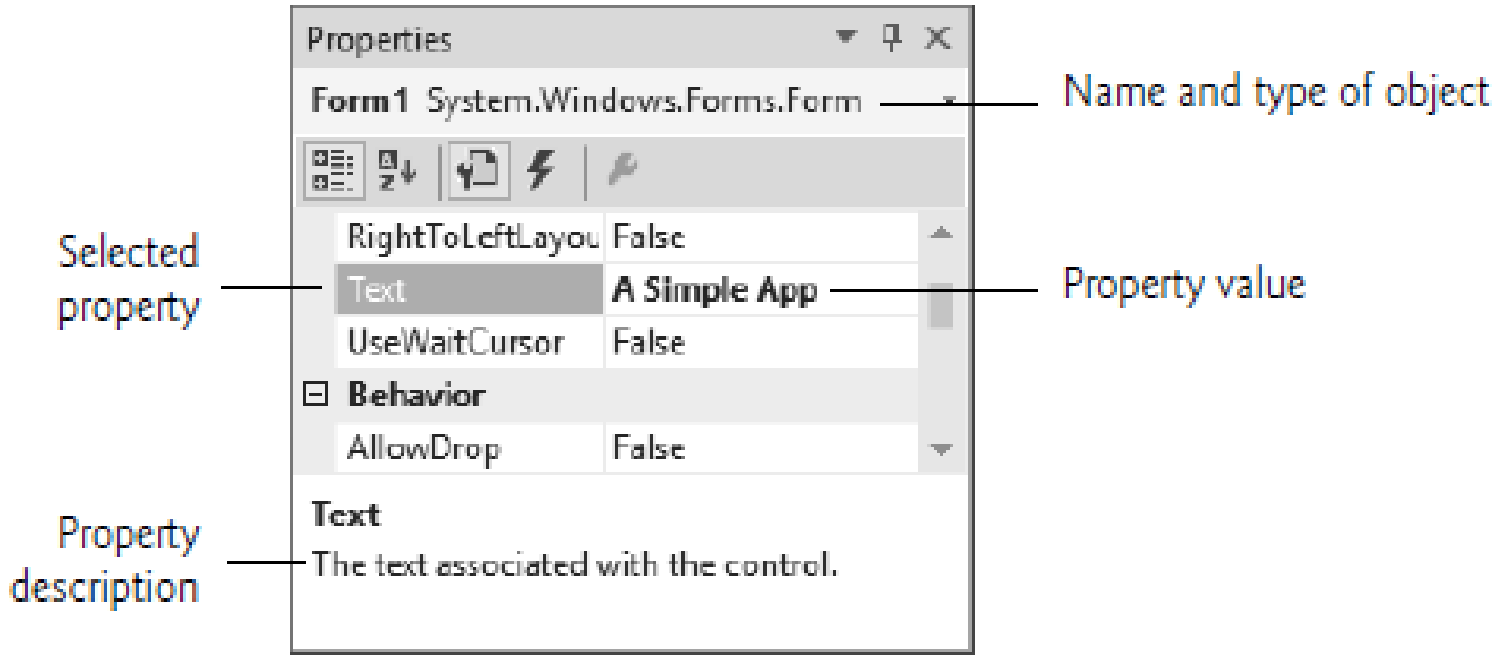
Creating a Simple App that Displays Text and an Image

The text **Form1.cs [Design]** in the tab containing the Form means that we're designing the Form ***visually*** rather than ***programmatically***. An asterisk (*****) at the end of the text in a tab indicates that you've changed the file and the changes have not yet been saved.

Step 3: Setting the Text in the Form's Title Bar

The text in the Form's title bar is determined by the Form's **Text property.**. If the **Properties** window is not open, select **View > Properties Window** and pin down the window so it doesn't auto hide. Click anywhere in the Form to display the Form's properties in the **Properties** window. In the textbox to the right of the Text property, type "A Simple App". Press the *Enter* key—the Form's title bar is updated immediately

Creating a Simple App that Displays Text and an Image

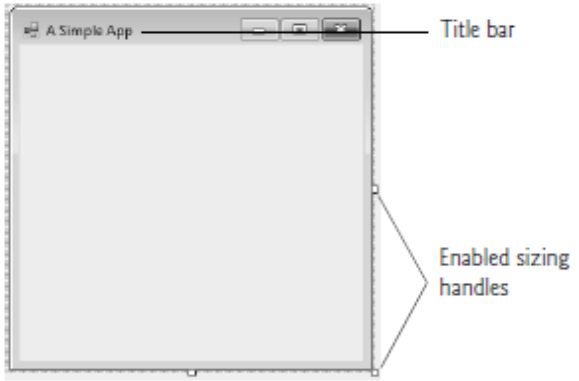


Setting the Form's Text property in the Properties window.

Creating a Simple App that Displays Text and an Image

Step 4: Resizing the Form

The Form's size is specified in pixels (that is, dots on the screen). By default, a Form is 300 pixels wide and 300 pixels tall. You can resize the Form by dragging one of its **sizing handles.** You also can do this via the Form's Size property in the **Properties** window.



Form with updated title-bar text and enabled sizing handles.

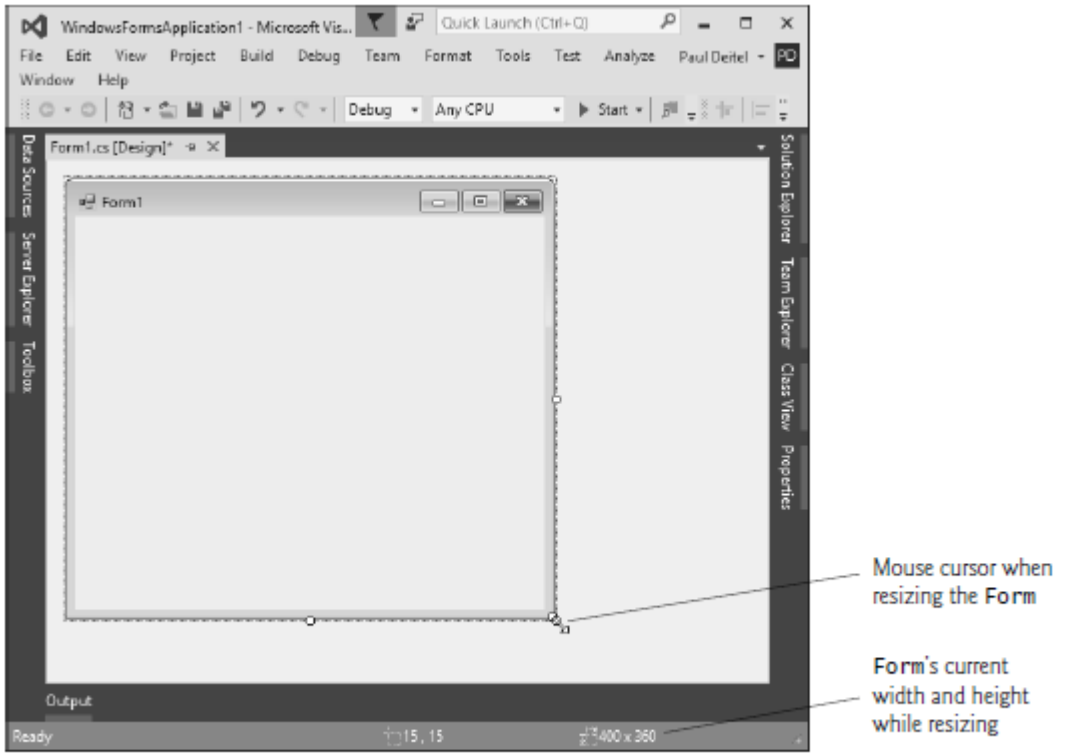


Fig. 2.21 | Resizing the Form.

Creating a Simple App that Displays Text and an Image

Changing the Form's Background Color

The **BackColor property** specifies a Form's or control's background color. Clicking Back-Color in the **Properties** window causes a down-arrow button to appear next to the value of the property

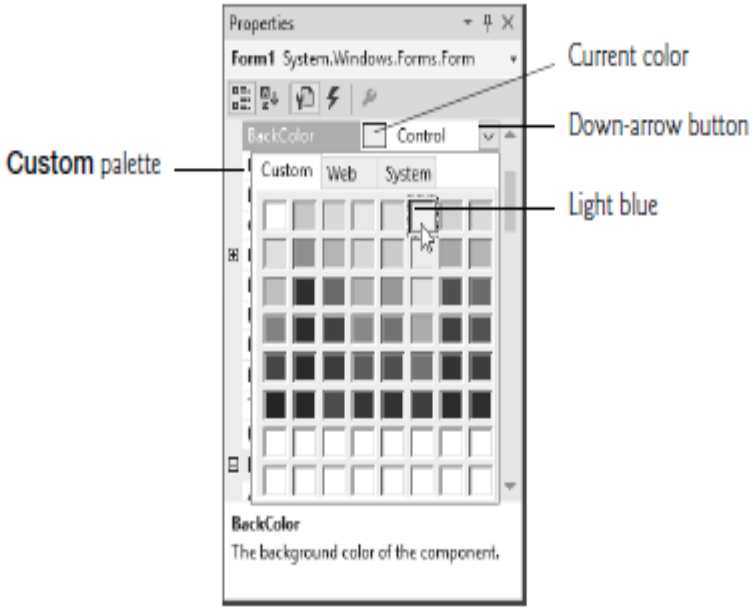


Fig. 2.22 | Changing the Form's BackColor property.

Creating a Simple App that Displays Text and an Image

Step 6: Adding a Label Control to the Form

For the app we're creating in this chapter, the typical controls we use are located in the **Toolbox**'s **Common Controls** group, and also can be found in the **All Windows Forms** group. If the **Toolbox** is *not* already open, select **View > Toolbox** to display the set of controls you'll use for creating your apps.

Fig. 2.24 | Adding a Label to the Form.

Creating a Simple App that Displays Text and an Image

Step 7: Customizing the Label's Appearance

Click the Label's text in the Form to select it and display its properties in the **Properties** window. The Label's Text property determines the text that the Label displays. The Form and Label each have their own Text property—Forms and controls can have the *same* property names (such as Text, BackColor etc.) without conflict. Each common properties purpose can vary by control.

Fig. 2.25 | GUI after the Form and Label have been customized.

Creating a Simple App that Displays Text and an Image

Step 8: Setting the Label's Font Size

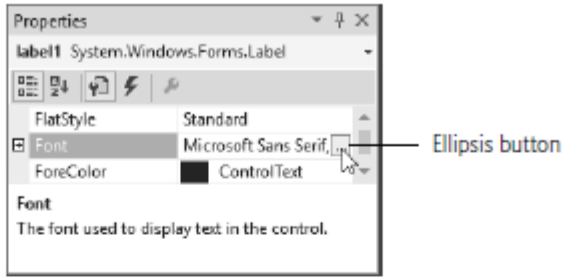


Fig. 2.26 | **Properties** window displaying the `Label`'s `Font` property.

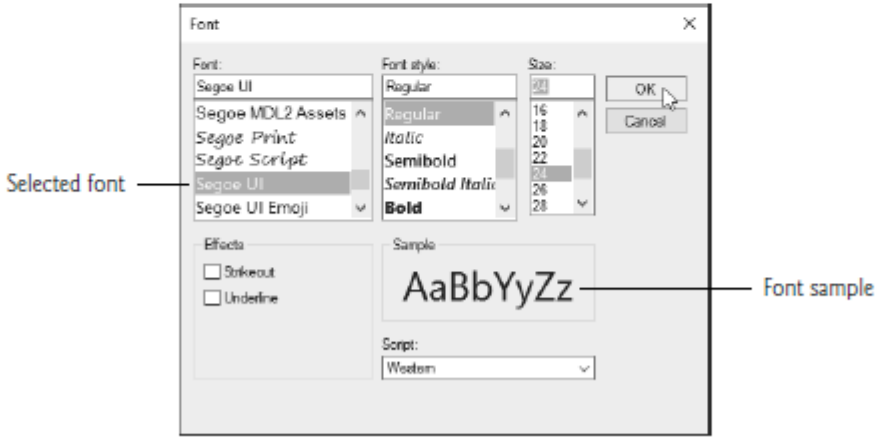


Fig. 2.27 | **Font** dialog for selecting fonts, styles and sizes.

Creating a Simple App that Displays Text and an Image

Step 9: Aligning the Label's Text

Select the Label's **TextAlign** property, which determines how the text is aligned within the Label. A three-by-three grid of buttons representing alignment choices is displayed (Fig. 2.28). The position of each button corresponds to where the text appears in the Label.

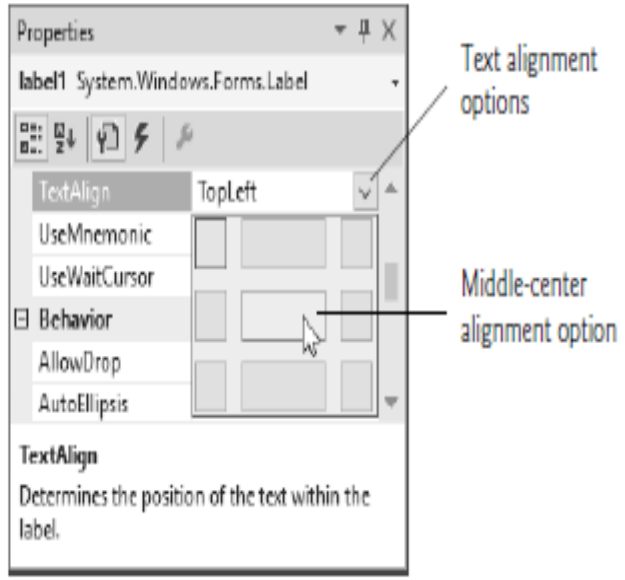


Fig. 2.28 | Centering the Label's text.

Creating a Simple App that Displays Text and an Image

Step 10: Adding a PictureBox to the Form

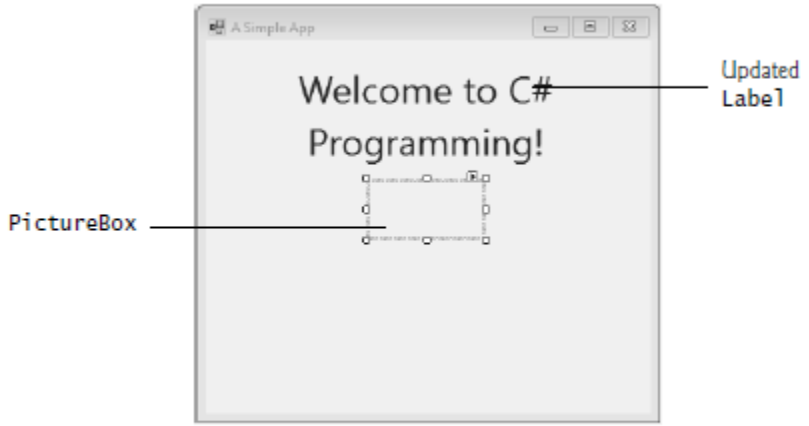


Fig. 2.29 | Inserting and aligning a `PictureBox`.

Creating a Simple App that Displays Text and an Image

Step 11: Inserting an Image

Click the PictureBox to display its properties in the **Properties** window (Fig. 2.30), then:

1. Locate and select the **Image property**, which displays a preview of the selected image or **(none)** if no image is selected.
2. Click the ellipsis button to display the **Select Resource dialog** (Fig. 2.31), which is used to import files, such as images, for use in an app.

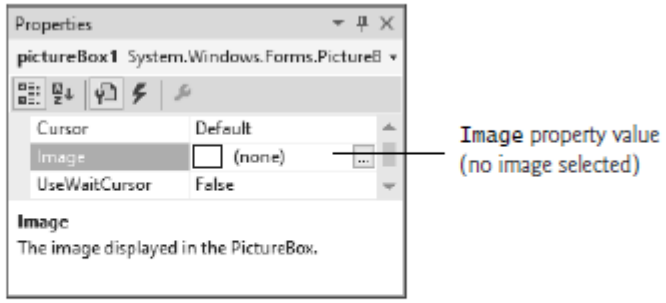


Fig. 2.30 | Image property of the PictureBox.

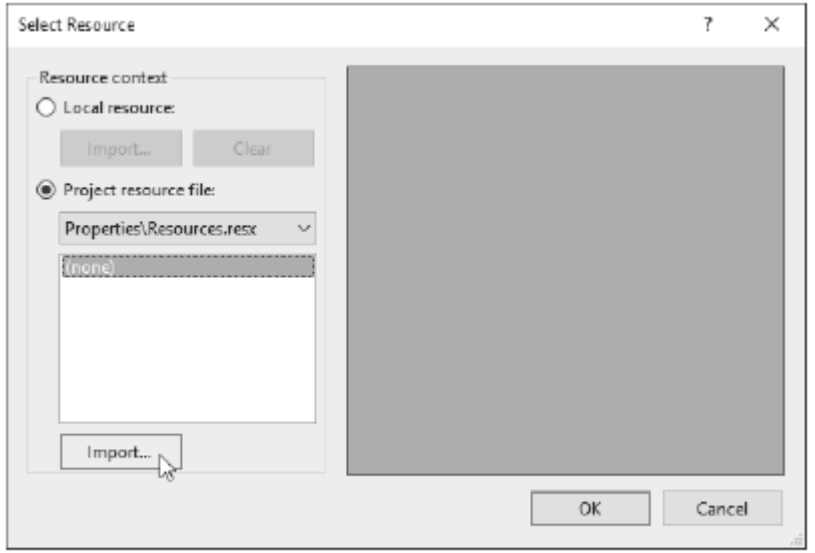


Fig. 2.31 | Select Resource dialog to select an image for the PictureBox.

Creating a Simple App that Displays Text and an Image

3. Click the **Import...** button to browse for an image, select the image file and click **OK** to add it to your project.

4. To scale the image to fit in the PictureBox, change the **SizeMode property** to **StretchImage** (Fig. 2.33). Resize the PictureBox, making it larger (Fig. 2.34), then re-center the PictureBox horizontally.

Creating a Simple App that Displays Text and an Image

Step 12: Saving the Project

Select **File > Save All** to save the entire solution. The solution file (which has the filename extension .sln) contains the name and location of its project, and the project file (which has the filename extension .csproj) contains the names and locations of all the files in the project. If you want to reopen your project at a later time, simply open its .sln file.

Creating a Simple App that Displays Text and an Image

Fig. 2.32 | **Select Resource** dialog displaying a preview of selected image.

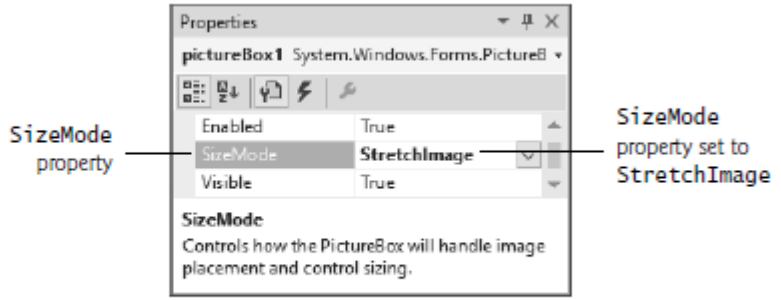


Fig. 2.33 | Scaling an image to the size of the `PictureBox`.

Fig. 2.34 | `PictureBox` displaying an image.

Creating a Simple App that Displays Text and an Image

Step 13: Running the Project

Select **Debug > Start Debugging** to execute the app (or press the *F5* key). The IDE enters run mode and displays "**(Running)**" next to the app's name in the IDE's title bar.

Figure 2.35 shows the running app, which appears in its own window outside the IDE.

Fig. 2.35 | IDE in run mode, with the running app in the foreground.

Creating a Simple App that Displays Text and an Image

Step 14: Terminating the App

You can terminate the app by clicking its close box in the top-right corner of the running app's window. You also can select **Debug > Stop Debugging** to terminate the app.

Fig. 2.35 | IDE in run mode, with the running app in the foreground.

GUI:

The .NET framework provides a class library of various graphical user interface tools such as frame, text box, buttons etc. that C# can use to implement a GUI very easily and fast.
The .NET framework also equips these classes with events and event handlers to perform action upon interacting with these visual items by the user.

GUI: Visual Items

The following are some of the visual items.

- Form: displays as a window.
- Button: displays a clickable button
- Label: displays a label
- TextBox: displays an area to edit
- RadioButton: displays a selectable button

GUI: Events

When anything of interest occurs, it's called an event such as a button click, mouse pointer movement, edit in a textbox etc.
An event is raised by a class. This is called publishing an event.
It can be arranged that when an event is raised, a class will be notified to handle this event, i.e. perform required tasks. This is called subscribing to the event. This is done via:

- Delegates or
- Anonymous function or
- Lambda expression.

These will be discussed shortly

GUI: Events

- The .NET Framework has many built-in events and delegates for easily subscribing to these events by various classes.
- For example, for Button class, the click event is called “Click” and the delegate for handler is called “System.EventHandler”. These are already defined in the .NET class library.
- To tie the event with the handler, the operator “+=” is used. (Will be discussed shortly)
- Then, when the Button object is clicked, that function will execute.

Visual Programming
Lecture 5
Arrays

Outline

- What is an array.
- Declaring and Creating Arrays in C#.
- Single dimension array.
- Multidimension array.

What is an Array?

- Array are the homogeneous(similar) collections of data types.
- Array Size remains same once created.
- Simple Declaration format for creating an array
  - type [] identifier= new type[integral value];

Arrays

- An array is a group of variables—called **elements**—containing values of the *same type*. Arrays are reference types—an array variable is actually a reference to an array object. An array's elements can be either value types or reference types, including other arrays—e.g., every element of an int array is an int *value*, and every element of a string array is a *reference* to a string object. Array names follow the same conventions as other variable names.
- Figure 8.1 shows a logical representation of an integer array called c that contains 12 elements.

- ***Indices Must Be Nonnegative Integer Values***
- Can we write this statement If we assume that variable a is 5 and variable b is 6 ? c[a + b] += 2;

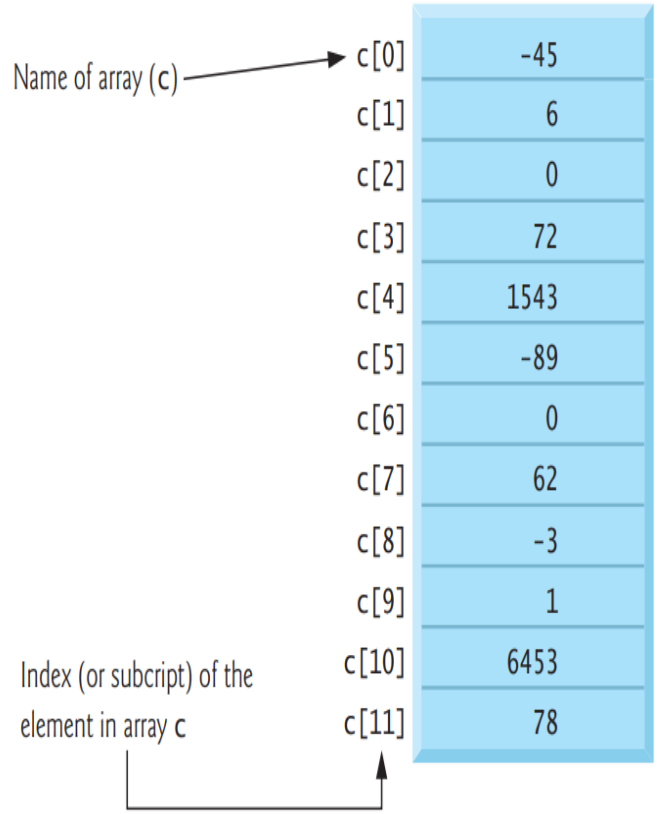


Fig. 8.1 | A 12-element array.

Declaring and Creating Arrays

- The following statement creates an array object containing 12 int elements—each initialized to 0 by default—and stores the array's reference in variable c:
- `int[] c = new int[12];`
- When you create an array with new, each element of the array receives the default value for the array's element type—0 for the numeric simple-type elements, false for bool elements and null for references.

```
int[] c; // declare the array variable
c = new int[12]; // create the array; assign to array variable
```

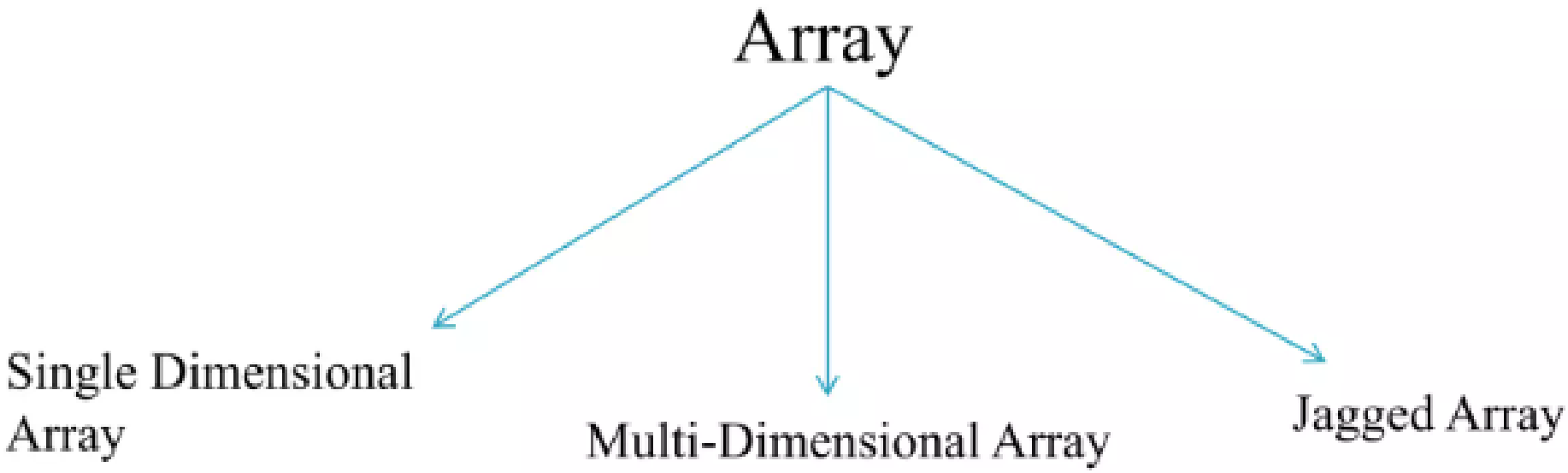
Resizing an Array

- Though arrays are fixed-length entities, you can use the static Array method Resize to create a new array with the specified length—class Array defines many methods for common array manipulations. This method takes as arguments
  - the array to be resized and
  - the new length

```
int[] newArray = new int[5];
Array.Resize(ref newArray, 10);
```

- If the new array is *smaller* than the old array, any content that cannot fit into the new array is *truncated without warning*.

Types of Array



Single-Dimensional Array

- The one-dimensional array or single dimensional array in C# is the simplest type of array that has only one row to store data.

- Declaration: double[] balance = new double[10];

Example: Creating and Initializing an Array

```
1  // Fig. 8.2: InitArray.cs
2  // Creating an array.
3  using System;
4
5  class InitArray
6  {
7     static void Main()
8     {
9        // create the space for array and initialize to default zeros
10       int[] array = new int[5]; // array contains 5 int elements
11
12       Console.WriteLine($"{"Index"}{"Value",8}"); // headings
13
14       // output each array element's value
15       for (int counter = 0; counter < array.Length; ++counter)
16       {
17          Console.WriteLine($"{counter,5}{array[counter],8}");
18       }
19    }
20 }
```

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
```

Example: Using an Array Initializer

```
1  // Fig. 8.3: InitArray.cs
2  // Initializing the elements of an array with an array initializer.
3  using System;
4
5  class InitArray
6  {
7     static void Main()
8     {
9        // initializer list specifies the value of each element
10       int[] array = {32, 27, 64, 18, 95};
11
12       Console.WriteLine($"{"Index"}{"Value",8}"); // headings
13
14       // output each array element's value
15       for (int counter = 0; counter < array.Length; ++counter)
16       {
17          Console.WriteLine($"{counter,5}{array[counter],8}");
18       }
19    }
20 }
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
```

Example: Calculating a Value to Store in Each Array Element

```
1  // Fig. 8.4: InitArray.cs
2  // Calculating values to be placed into the elements of an array.
3  using System;
4
5  class InitArray
6  {
7     static void Main()
8     {
9        const int ArrayLength = 5; // create a named constant
10       int[] array = new int[ArrayLength]; // create array
11
12       // calculate value for each array element
13       for (int counter = 0; counter < array.Length; ++counter)
14       {
15          array[counter] = 2 + 2 * counter;
16       }
17
18       Console.WriteLine($"{"Index"}{"Value",8}"); // headings
19
20       // output each array element's value
21       for (int counter = 0; counter < array.Length; ++counter)
22       {
23          Console.WriteLine($"{counter,5}{array[counter],8}");
24       }
25    }
26 }
```

```
Index    Value
    0        2
    1        4
    2        6
    3        8
    4       10
```

Example: Summing the Elements of an Array

```
1  // Fig. 8.5: SumArray.cs
2  // Computing the sum of the elements of an array.
3  using System;
4
5  class SumArray
6  {
7     static void Main()
8     {
9        int[] array = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
10       int total = 0;
11
12       // add each element's value to total
13       for (int counter = 0; counter < array.Length; ++counter)
14       {
15          total += array[counter]; // add element value to total
16       }
17
18       Console.WriteLine($"Total of array elements: {total}");
19    }
20 }
```

```
Total of array elements: 849
```

iterating Through Arrays with foreach

```
foreach (type identifier in arrayName)
{
    statement
}
```

Example: Summing the Elements of an Array using *foreach*

```
1  // Fig. 8.6: ForEachTest.cs
2  // Using the foreach statement to total integers in an array.
3  using System;
4
5  class ForEachTest
6  {
7     static void Main()
8     {
9        int[] array = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
10       int total = 0;
11
12       // add each element's value to total
13       foreach (int number in array)
14       {
15          total += number;
16       }
17
18       Console.WriteLine($"Total of array elements: {total}");
19    }
20 }
```

```
Total of array elements: 849
```

Exception Handling: Processing the Incorrect Response

- An exception indicates a problem that occurs while a program executes. Exception handling enables you to create fault-tolerant programs that can resolve (or handle) exceptions. In many cases, this allows a program to continue executing as if no problems were encountered. For example, the Student Poll app still displays results (Fig. 8.9), even though one of the responses was out of range. More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate. When the runtime or a method detects a problem, such as an invalid array index or an invalid method argument, it throws an exception—that is, an exception occurs. The exception here is thrown by the runtime.

The try Statement

To handle an exception, place any code that might throw an exception in a **try statement** (Fig. 8.9, lines 18–27). The **try block** (lines 18–21) contains the code that might *throw* an exception, and the **catch block** (lines 22–27) contains the code that *handles* the exception if one occurs. You can have many catch blocks to handle different types of exceptions that might be thrown in the corresponding try block. When line 20 correctly increments an element of the frequency array, lines 22–27 are ignored. The braces that delimit the bodies of the try and catch blocks are required.

Executing the catch Block

When the program encounters the value 14 in the responses array, it attempts to add 1 to frequency[14], which does *not* exist—the frequency array has only six elements. Because the runtime performs array bounds checking, it generates an exception—specifically line 20 throws an **IndexOutOfRangeException** to notify the program of this problem. At this point the try block *terminates* and the catch block begins executing—if you declared any variables in the try block, they no longer exist, so they're *not accessible* in the catch block.

Executing the catch Block

- The catch block declares an exception parameter's type (**IndexOutOfRangeException**) and name (ex). The catch block can handle exceptions of the specified type. Inside the catch block, you can use the parameter's identifier to interact with a caught exception object.

Using Arrays to Analyze Survey Results; Intro to Exception Handling

```
1  // Fig. 8.9: StudentPoll.cs
2  // Poll analysis app.
3  using System;
4
5  class StudentPoll
6  {
7     static void Main()
8     {
9        // student response array (more typically, input at runtime)
10       int[] responses = {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
11          2, 3, 3, 2, 14};
12       var frequency = new int[6]; // array of frequency counters
13
14       // for each answer, select responses element and use that value
15       // as frequency index to determine element to increment
16       for (var answer = 0; answer < responses.Length; ++answer)
17       {
18          try
19          {
20             ++frequency[responses[answer]];
21          }
22          catch (IndexOutOfRangeException ex)
23          {
24             Console.WriteLine(ex.Message);
25             Console.WriteLine(
26                $"    responses[{answer}] = {responses[answer]}\n");
27          }
28       }
```

Fig. 8.9 | Poll analysis app. (Part 1 of 2.)

Using Arrays to Analyze Survey Results; Intro to Exception Handling

```
29
30          Console.WriteLine($"{"Rating"}{"Frequency",10}");
31
32          // output each array element's value
33          for (var rating = 1; rating < frequency.Length; ++rating)
34          {
35             Console.WriteLine($"{rating,6}{frequency[rating],10}");
36          }
37       }
38    }
```

```
Index was outside the bounds of the array.
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

Fig. 8.9 | Poll analysis app. (Part 2 of 2.)

Message Property of the Exception Parameter

- When lines 22–27 *catch* the exception, the program displays a message indicating the problem that occurred. Line 24 uses the exception object's built-in **Message property** to get the error message and display it. Once the message is displayed, the exception is considered handled, and the program continues with the next statement after the catch block's closing brace. In this example, the end of the foreach statement is reached (line 28), so the program continues with the next iteration of the loop. We use exception handling again in Chapter 10, then Chapter 13 presents a deeper look at exception handling.

Multi-Dimensional Array

- Multi-dimensional arrays are also called rectangular array.
- Stored sequentially.
- It contains more than one rows.
- Declaration:

```
type[,] array = new type[9, 9];
Array[3,8]=100;
```

Multi-Dimensional Array

Multidimensional Arrays

Two-dimensional arrays are often used to represent **tables of values** consisting of information arranged in **rows** and **columns**. To identify a particular table element, we must specify *two* indices. By convention, the first identifies the element's row and the second its column. (**Multidimensional arrays** can have more than two dimensions, but such arrays are beyond the scope of this book.) C# supports two types of two-dimensional arrays— **rectangular arrays** and **jagged arrays**.

Multi-Dimensional Array

Rectangular Arrays

- Rectangular arrays are used to represent tables of information in the form of rows and columns, where each row has the same number of columns. Figure 8.17 illustrates a rectangular array named a containing three rows and four columns—a three-by-four array. In general, an array with *m* rows and *n* columns is called an ***m*-by-*n* array**.

Multi-Dimensional Array

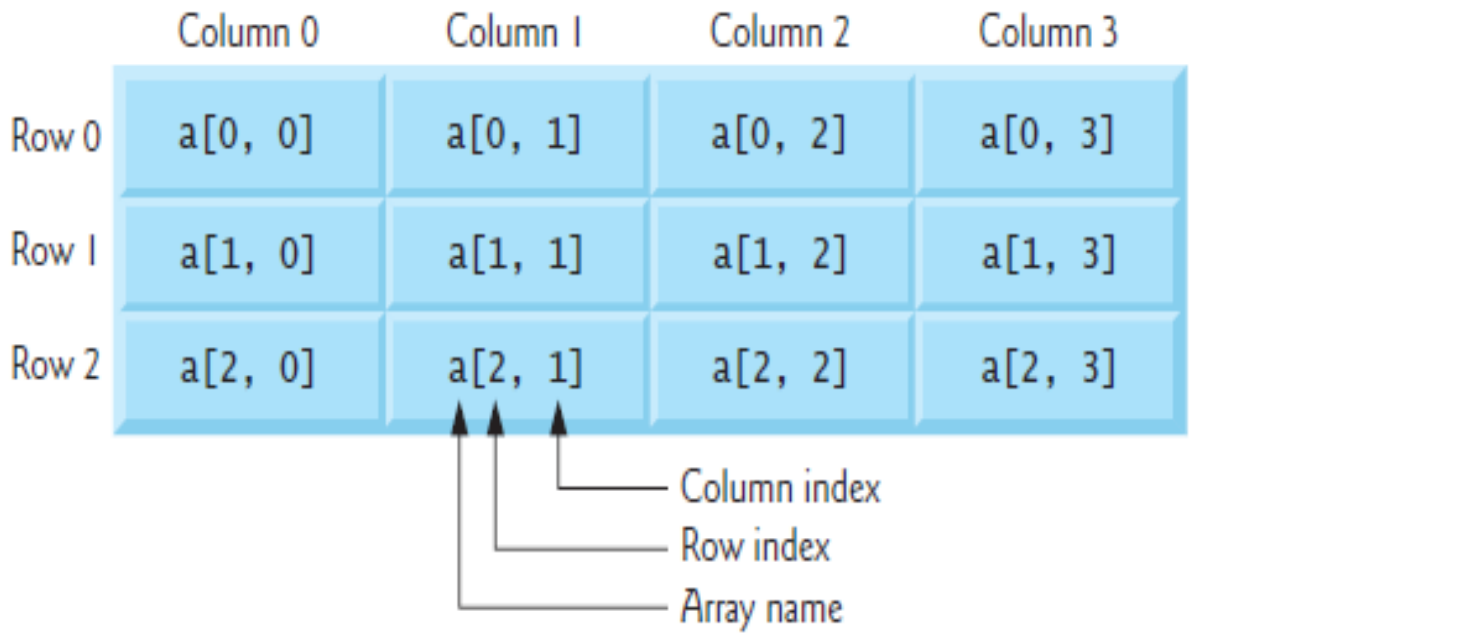


Fig. 8.17 | Rectangular array with three rows and four columns.

Multi-Dimensional Array

Array Initializer for a Two-Dimensional Rectangular Array

Like one-dimensional arrays, multidimensional arrays can be initialized with array initializers in declarations. A rectangular array b with two rows and two columns could be declared and initialized with **nested array initializers** as follows:

```
int[,] b = {{1, 2}, {3, 4}};
```

Jagged Arrays

A jagged array is maintained as a one-dimensional array in which each element refers to a one-dimensional array. The manner in which jagged arrays are represented makes them quite flexible, because the lengths of the rows in the array need *not* be the same. For example, jagged arrays could be used to store a single student's exam grades across multiple courses, where the number of exams may vary from course to course.

Jagged Arrays

Array Initializer for a Two-Dimensional Jagged Array

We can access the elements in a jagged array by an array-access expression of the form *arrayName*[*row*][*column*]—similar to the array-access expression for rectangular arrays, but with a separate set of square brackets for each dimension. A jagged array with three rows of different lengths could be declared and initialized as follows:

```
int[][] jagged = {new int[] {1, 2},
                  new int[] {3},
                  new int[] {4, 5, 6}};
```

Diagram of a Two-Dimensional Jagged Array in Memory

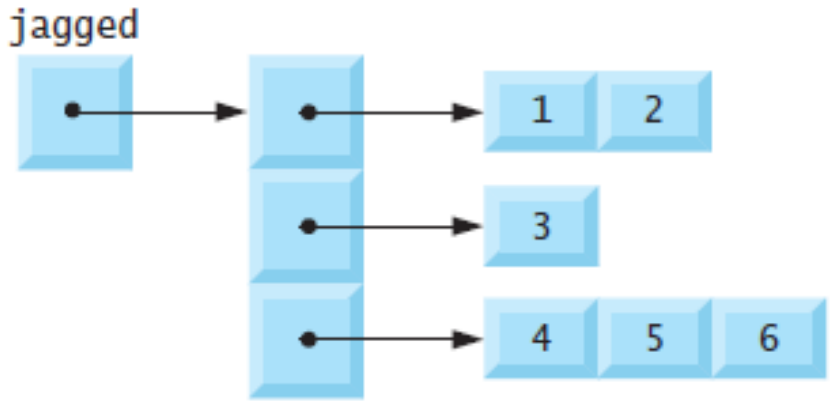


Fig. 8.18 | Jagged array with three rows of different lengths.

Creating Two-Dimensional Arrays with Array-Creation Expressions

- ***Creating Two-Dimensional Arrays with Array-Creation Expressions***
- A rectangular array can be created with an array-creation expression. For example, the following lines declare variable b and assign it a reference to a three-by-four rectangular array:

```
int[,] b;
b = new int[3, 4];
```

- In this case, we use the literal values 3 and 4 to specify the number of rows and number of columns, respectively, but this is *not* required—apps also can use variables and expressions to specify array dimensions. As with one-dimensional arrays, the elements of a rectangular array are initialized when the array object is created.

Creating Two-Dimensional Arrays with Array-Creation Expressions

A jagged array cannot be completely created with a single array-creation expression. The following statement is a syntax error:

```
int[][] c = new int[2][5]; // error
```

Instead, each one-dimensional array in the jagged array must be initialized separately. A jagged array can be created as follows:

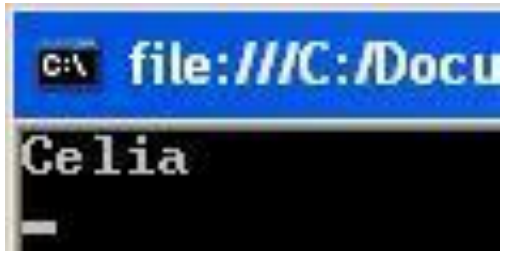
```
int[][] c;
c = new int[2][]; // create 2 rows
c[0] = new int[5]; // create 5 columns for row 0
c[1] = new int[3]; // create 3 columns for row 1
```

The preceding statements create a jagged array with two rows. Row 0 has five columns, and row 1 has three columns.

Properties

- Properties are members of a class that allows for easy and simplified getters and setters implementation of its private field variables.

```
class Client{
    private string name ;
    public string Name{
        get{
            return name;
        }
        set{
            name=value;
        }
    }
    static void Main(string[] args)
    {
        Client c = new Client();
        c.Name = "Celia";
        System.Console.WriteLine(c.Name);
        System.Console.ReadLine();
    }
}
```



Properties

Automatically Implemented

C# also has a feature to automatically implement the getters and setter for you.

Users have direct access to the data members of the class.

Following example does the same thing as the previous example, but using automatically implemented properties

```
class Client2{
    public string Name { get; set; }
    static void Main(string[] args){
        Client2 c = new Client2();
        c.Name = "Cruz";
        System.Console.WriteLine(c.Name);
        System.Console.ReadLine();
    }
}
```

Indexers

- Indexers allow a class to be used as an array. For instance the “[]” operator can be used and the ‘foreach’, ‘in’ keywords can also be used on a class that has indexers.
- The internal representation of the items in that class are managed by the developer.
- Indexers are defined by the following expression.

```
public int this[int idx]{
        get{/* your code*/}; set{/*code here*/};
}
```

Nested Classes

- C# supports nested class which defaults to private.

```
class Program
{
    public class InsiderClass
    {
        private int a;
    }
    static void Main(string[] args)
    {
    }
}
```

Inheritance and Interface

- A class can directly inherit from only one base class and can implement multiple interfaces.
- To override a method defined in the base class, the keyword ‘override’ is used.
- An abstract class can be declared with the keyword ‘abstract’.
- A static class is a class that is declared with the ‘static’ keyword. It can not be instantiated and all members must be static.

Inheritance and Interface

```
class BaseClass{
    public virtual void show(){
        System.Console.WriteLine("base class");}
}
interface Interface1{void showMe();}
interface Interface2{void showYou();}
class DerivedAndImplemented: BaseClass,Interface1,Interface2{
    public void showMe() { System.Console.WriteLine("Me!"); }
    public void showYou() { System.Console.WriteLine("You!"); }
    public override void show(){
        System.Console.WriteLine("I'm in derived Class");}
    static void Main(string[] args){
            DerivedAndImplemented de = new DerivedAndImplemented();
            de.show();
            System.Console.Read();}
}
```

Class Access and Partial

- The class access modifiers are public, private, protected and internal. ‘internal’ is an intermediate access level which only allows access to classes in the same assembly.
- The ‘partial’ keyword can be used to split up a class definition in to multiple location (file etc). Can be useful when multiple developers are working on different parts of the same class.

Delegates

- Delegates are types that describe a method signature. This is similar to function pointer in C.
- At runtime, different actual methods of same signature can be assigned to the delegate enabling encapsulation of implementation.
- These are extensively used to implement GUI and event handling in the .net framework (will be discussed later).

Delegates

```
class Program
{
    delegate int mydel(int aa);
    int myfunc(int a){ return a*a; }
    int myfunc2(int a) { return a + a; }
    static void Main(string[] args)
    {
        Program p=new Program();
        mydel d=p.myfunc; System.Console.WriteLine(d(5));
        d = p.myfunc2; System.Console.WriteLine(d(5));
        System.Console.Read();
    }
}
```

Delegates

- Lambda Expression
- In C#, implementors (functions) that are targeted by a delegate, can be created anonymously, inline and on the fly by using the lambda operator “=>”.

```
class Program {
    delegate int mydel(int aa, int bb);
    static void Main(string[] args) {
        mydel d = (a, b) => a + 2 * b;
        // in above line, read a,b go to a+b*2 to evaluate
        System.Console.WriteLine(d(2,3));
        System.Console.Read();
    }
}
```

Generics

- Generics are a powerful feature of C#. These enable defining classes and methods without specifying a type to use at coding time. A placeholder for type <T> is used and when these methods or classes are used, the client just simply has to plug in the appropriate type.
- Used commonly in lists, maps etc.

Delegates

```
class Genclass<T>{
    public void genfunc(int a, T b){
        for (int i = 0; i < a; i++){
            System.Console.WriteLine(b);
        }
    }
}
class Program{
    static void Main(string[] args){
        Genclass<float> p = new Genclass<float>();
        p.genfunc(3,(float)5.7);
        System.Console.Read();
    }
}
```

References

- Deitel, P., & Deitel, H. (2016). *Visual C# how to program*. Pearson.
- Sharp, J. (2013). *Microsoft Visual C# 2013 Step by Step*. Pearson Education.
- Watson, K., Hammer, J. V., Reid, J. D., Skinner, M., Kemper, D., & Nagel, C. (2012). *Beginning visual C# 2012 programming*. John Wiley & Sons.