Contents

C# Quickstarts

5/4/2018 • 2 minutes to read • Edit Online

Welcome to the C# Quickstarts. These start with interactive lessons that you can run in your browser.

The first lessons explain C# concepts using small snippets of code. You'll learn the basics of C# syntax and how to work with data types like strings, numbers, and booleans. It's all interactive, and you'll be writing and running code within minutes. These first lessons assume no prior knowledge of programming or the C# language.

All the quickstarts following the Hello World lesson are available using the online browser experience or in your own local development environment. At the end of each quickstart, you decide if you want to continue with the next quickstart online or on your own machine. There are links to help you setup your environment and continue with the next quickstart on your machine.

Hello world

In the Hello world quickstart, you'll create the most basic C# program. You'll explore the `string` type and how to work with text.

Numbers in C#

In the Numbers in C# quickstart, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding, and how to perform mathematical calculations using C#. This quickstart is also available to run locally on your machine.

This quickstart assumes that you have finished the Hello world lesson.

Branches and loops

The Branches and loops quickstart teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions. This quickstart is also available to run locally on your machine.

This quickstart assumes that you have finished the Hello world and Numbers in C# lessons.

String interpolation

The String interpolation quickstart shows you how to insert values into a string. You'll learn how to create an interpolated string with embedded C# expressions and how to control the text appearance of the expression results in the result string. This quickstart is also available to run locally on your machine.

This quickstart assumes that you have finished the Hello world, Numbers in C#, and Branches and loops lessons.

List collection

The List collection lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists. This quickstart is also available to run locally on your machine.

This quickstart assumes that you have finished the lessons listed above.

Introduction to classes

This final quickstart is only available to run on your machine, using your own local development environment and .NET Core. You'll build a console application and see the basic object-oriented features that are part of the C# language.

This quickstart assumes you've finished the online quickstarts, and you've installed .NET Core SDK and Visual Studio Code.

Local environment

5/4/2018 • 2 minutes to read • Edit Online

The first step to trying a quickstart locally is to setup a development environment on your local machine. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux.

Alternatively, you can install the .NET Core SDK and Visual Studio Code.

Basic application development flow

You'll create applications using the `dotnet new` command. This command generates the files and assets necessary for your application. The quickstarts all use the `console` application type.

The other commands you'll use are `dotnet build` to build the executable, and `dotnet run` to run the executable.

Pick your quickstart

You can start with any of the following quickstarts:

Numbers in C#

In the Numbers in C# quickstart, you'll learn how computers store numbers and how to perform calculations with different numeric types. You'll learn the basics of rounding, and how to perform mathematical calculations using C#.

This quickstart assumes that you have finished the Hello world lesson.

Branches and loops

The Branches and loops quickstart teaches the basics of selecting different paths of code execution based on the values stored in variables. You'll learn the basics of control flow, which is the basis of how programs make decisions and choose different actions.

This quickstart assumes that you have finished the Hello world and Numbers in C# lessons.

String interpolation

The String interpolation quickstart shows you how to insert values into a string. You'll learn how to create an interpolated string with embedded C# expressions and how to control the text appearance of the expression results in the result string.

This quickstart assumes that you have finished the Hello world, Numbers in C#, and Branches and loops lessons.

List collection

The List collection lesson gives you a tour of the List collection type that stores sequences of data. You'll learn how to add and remove items, search for items, and sort the lists. You'll explore different kinds of lists.

This quickstart assumes that you have finished the lessons listed above.

Introduction to classes

This final quickstart is only available to run on your machine, using your own local development environment and .NET Core. You'll build a console application and see the basic object-oriented features that are part of the C# language.

Numbers in C# quickstart

5/4/2018 • 8 minutes to read • Edit Online

This quickstart teaches you about the number types in C# interactively. You'll write small amounts of code, then you'll compile and run that code. The quickstart contains a series of lessons that explore numbers and math operations in C#. These lessons teach you the fundamentals of the C# language.

This quickstart expects you to have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the introduction to the local quickstarts with links to more details.

Explore integer math

Create a directory named **numbers-quickstart**. Make that the current directory and run `dotnet new console -n NumbersInCSharp -o .`.

Open **Program.cs** in your favorite editor, and replace the line `Console.Writeline("Hello World!");` with the following:

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You've just seen one of the fundamental math operations with integers. The `int` type represents an **integer**, a positive or negative whole number. You use the `+` symbol for addition. Other common mathematical operations for integers include:

- `-` for subtraction
- `*` for multiplication
- `/` for division

Start by exploring those different operations. Add these lines after the line that writes the value of `c`:

```
c = a - b;
Console.WriteLine(c);
c = a * b;
Console.WriteLine(c);
c = a / b;
Console.WriteLine(c);
```

Run this code by typing `dotnet run` in your command window.

You can also experiment by performing multiple mathematics operations in the same line, if you'd like. Try `c = a + b - 12 * 17;` for example. Mixing variables and constant numbers is allowed.

TIP

As you explore C# (or any programming language), you'll make mistakes when you write code. The **compiler** will find those errors and report them to you. When the output contains error messages, look closely at the example code and the code in your window to see what to fix. That exercise will help you learn the structure of C# code.

You've finished the first step. Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Rename your `Main` method to `WorkingWithIntegers` and write a new `Main` method that calls `WorkingWithIntegers`. When you have finished, your code should look like this:

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;
            int c = a + b;
            Console.WriteLine(c);
            c = a - b;
            Console.WriteLine(c);
            c = a * b;
            Console.WriteLine(c);
            c = a / b;
            Console.WriteLine(c);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();
        }
    }
}
```

Explore order of operations

Comment out the call to `WorkingWithIntegers()`. It will make the output less cluttered as you work in this section:

```
//WorkingWithIntegers();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler does not generate any executable code from comments.

The C# language defines the precedence of different mathematics operations with rules consistent with the rules you learned in mathematics. Multiplication and division take precedence over addition and subtraction. Explore that by adding the following code to your `Main` method, and execuing `dotnet run`:

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

The output demonstrates that the multiplication is performed before the addition.

You can force a different order of operation by adding parentheses around the operation or operations you want performed first. Add the following lines and run again:

```
d = (a  + b) * c;
Console.WriteLine(d);
```

Explore more by combining many different operations. Add something like the following lines at the bottom of your `Main` method. Try `dotnet run` again.

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

You may have noticed an interesting behavior for integers. Integer division always produces an integer result, even when you'd expect the result to include a decimal or fractional portion.

If you haven't seen this behavior, try the following code at the end of your `Main` method:

```
int e = 7;
int f = 4;
int g = 3;
int h = (e  + f) / g;
Console.WriteLine(h);
```

Type `dotnet run` again to see the results.

Before moving on, let's take all the code you've written in this section and put it in a new method. Call that new method `OrderPrecedence`. You should end up with something like this:

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;
            int c = a + b;
            Console.WriteLine(c);
            c = a - b;
            Console.WriteLine(c);
            c = a * b;
            Console.WriteLine(c);
            c = a / b;
            Console.WriteLine(c);
        }

        static void OrderPrecedence()
        {
            int a = 5;
            int b = 4;
            int c = 2;
            int d = a + b * c;
            Console.WriteLine(d);

            d = (a  + b) * c;
            Console.WriteLine(d);

            d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
            Console.WriteLine(d);

            int e = 7;
            int f = 4;
            int g = 3;
            int h = (e  + f) / g;
            Console.WriteLine(h);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();

            OrderPrecedence();

        }
    }
}
```

Explore integer precision and limits

That last sample showed you that integer division truncates the result. You can get the **remainder** by using the **modulo** operator, the `%` character. Try the following code in your `Main` method:

```
int a = 7;
int b = 4;
int c = 3;
int d = (a  + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

The C# integer type differs from mathematical integers in one other way: the `int` type has minimum and maximum limits. Add this code to your `Main` method to see those limits:

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

If a calculation produces a value that exceeds those limits, you have an **underflow** or **overflow** condition. The answer appears to wrap from one limit to the other. Add these two lines to your `Main` method to see an example:

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Notice that the answer is very close to the minimum (negative) integer. It's the same as `min + 2`. The addition operation **overflowed** the allowed values for integers. The answer is a very large negative number because an overflow "wraps around" from the largest possible integer value to the smallest.

There are other numeric types with different limits and precision that you would use when the `int` type doesn't meet your needs. Let's explore those next.

Once again, let's move the code you wrote in this section into a separate method. Name it `TestLimits`.

Work with the double type

The `double` numeric type represents a double-precision floating point number. Those terms may be new to you. A **floating point** number is useful to represent non-integral numbers that may be very large or small in magnitude. **Double-precision** means that these numbers are stored using greater precision than **single-precision**. On modern computers, it is more common to use double precision than single precision numbers. Let's explore. Add the following code and see the result:

```
double a = 5;
double b = 4;
double c = 2;
double d = (a  + b) / c;
Console.WriteLine(d);
```

Notice that the answer includes the decimal portion of the quotient. Try a slightly more complicated expression with doubles:

```
double e = 19;
double f = 23;
double g = 8;
double h = (e  + f) / g;
Console.WriteLine(h);
```

The range of a double value is much greater than integer values. Try the following code below what you've written so far:

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

These values are printed out in scientific notation. The number to the left of the `E` is the significand. The number to the right is the exponent, as a power of 10.

Just like decimal numbers in math, doubles in C# can have rounding errors. Try this code:

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

You know that `0.3` repeating is not exactly the same as `1/3`.

Challenge

Try other calculations with large numbers, small numbers, multiplication and division using the `double` type. Try more complicated calculations.

After you've spent some time with the challenge, take the code you've written and place it in a new method. Name that new method `WorkWithDoubles`.

Work with fixed point types

You've seen the basic numeric types in C#: integers and doubles. There is one other type to learn: the `decimal` type. The `decimal` type has a smaller range but greater precision than `double`. The term **fixed point** means that the decimal point (or binary point) doesn't move. Let's take a look:

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Notice that the range is smaller than the `double` type. You can see the greater precision with the decimal type by trying the following code:

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

The `M` suffix on the numbers is how you indicate that a constant should use the `decimal` type.

Notice that the math using the decimal type has more digits to the right of the decimal point.

Challenge

Now that you've seen the different numeric types, write code that calculates the area of a circle whose radius is 2.50 centimeters. Remember that the area of a circle is the radius squared multiplied by PI. One hint: .NET contains a constant for PI, Math.PI that you can use for that value.

You should get an answer between 19 and 20. You can check your answer by looking at the finished sample code on GitHub

Try some other formulas if you'd like.

You've completed the "Numbers in C#" quickstart. You can continue with the Branches and loops quickstart in your own development environment.

You can learn more about numbers in C# in the following topics:

Integral Types Table
Floating-Point Types Table

Built-In Types Table

Implicit Numeric Conversions Table

Explicit Numeric Conversions Table

Branches and loops

5/30/2018 • 8 minutes to read • Edit Online

This quickstart teaches you how to write code that examines variables and changes the execution path based on those variables. You write C# code and see the results of compiling and running it. The quickstart contains a series of lessons that explore branching and looping constructs in C#. These lessons teach you the fundamentals of the C# language.

This quickstart expects you to have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the introduction to the local quickstarts with links to more details.

Make decisions using the `if` statement

Create a directory named **branches-quickstart**. Make that the current directory and run `dotnet new console -n BranchesAndLoops -o .`. This command creates a new .NET Core console application in the current directory.

Open **Program.cs** in your favorite editor, and replace the line `Console.Writeline("Hello World!");` with the following code:

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Try this code by typing `dotnet run` in your console window. You should see the message "The answer is greater than 10." printed to your console.

Modify the declaration of `b` so that the sum is less than 10:

```
int b = 3;
```

Type `dotnet run` again. Because the answer is less than 10, nothing is printed. The **condition** you're testing is false. You don't have any code to execute because you've only written one of the possible branches for an `if` statement: the true branch.

TIP

As you explore C# (or any programming language), you'll make mistakes when you write code. The compiler will find and report the errors. Look closely at the error output and the code that generated the error. The compiler error can usually help you find the problem.

This first sample shows the power of `if` and Boolean types. A *Boolean* is a variable that can have one of two values: `true` or `false`. C# defines a special type, `bool` for Boolean variables. The `if` statement checks the value of a `bool`. When the value is `true`, the statement following the `if` executes. Otherwise, it is skipped.

This process of checking conditions and executing statements based on those conditions is very powerful.

Make if and else work together

To execute different code in both the true and false branches, you create an `else` branch that executes when the condition is false. Try this. Add the last two lines in the code below to your `Main` method (you should already have the first four):

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

The statement following the `else` keyword executes only when the condition being tested is `false`. Combining `if` and `else` with Boolean conditions provides all the power you need to handle both a `true` and a `false` condition.

IMPORTANT

The indentation under the `if` and `else` statements is for human readers. The C# language doesn't treat indentation or white space as significant. The statement following the `if` or `else` keyword will be executed based on the condition. All the samples in this quickstart follow a common practice to indent lines based on the control flow of statements.

Because indentation is not significant, you need to use `{` and `}` to indicate when you want more than one statement to be part of the block that executes conditionally. C# programmers typically use those braces on all `if` and `else` clauses. The following example is the same as the one you just created. Modify your code above to match the following code:

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

TIP

Through the rest of this quickstart, the code samples all include the braces, following accepted practices.

You can test more complicated conditions. Add the following code in your `Main` method after the code you've written so far:

```
    int c = 4;
    if ((a + b + c > 10) && (a > b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is greater than the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not greater than the second");
}
```

The `&&` represents "and". It means both conditions must be true to execute the statement in the true branch. These examples also show that you can have multiple statements in each conditional branch, provided you enclose them in `{` and `}`.

You can also use `||` to represent "or". Add the following code after what you've written so far:

```
if ((a + b + c > 10) || (a > b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is greater than the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not greater than the second");
}
```

You've finished the first step. Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Rename your `Main` method to `ExploreIf` and write a new `Main` method that calls `ExploreIf`. When you have finished, your code should look like this:

```
using System;

namespace BranchesAndLoops
{
    class Program
    {
        static void ExploreIf()
        {
            int a = 5;
            int b = 3;
            if (a + b > 10)
            {
                Console.WriteLine("The answer is greater than 10");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
            }

            if ((a + b + c > 10) && (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("And the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("Or the first number is not greater than the second");
            }

            if ((a + b + c > 10) || (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("Or the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("And the first number is not greater than the second");
            }
        }

        static void Main(string[] args)
        {
            ExploreIf();
        }
    }
}
```

Comment out the call to `ExploreIf()`. It will make the output less cluttered as you work in this section:

```
//ExploreIf();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler does not generate any executable code from comments.

Use loops to repeat operations

In this section you use **loops** to repeat statements. Try this code in your `Main` method:

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

The `while` statement checks a condition and executes the statement or statement block following the `while`. It repeatedly checks the condition and executing those statements until the condition is false.

There's one other new operator in this example. The `++` after the `counter` variable is the **increment** operator. It adds 1 to the value of `counter` and stores that value in the `counter` variable.

IMPORTANT

Make sure that the `while` loop condition changes to false as you execute the code. Otherwise, you create an **infinite loop** where your program never ends. That is not demonstrated in this sample, because you have to force your program to quit using **CTRL-C** or other means.

The `while` loop tests the condition before executing the code following the `while`. The `do` ... `while` loop executes the code first, and then checks the condition. The do while loop is shown in the following code:

```
counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

This `do` loop and the earlier `while` loop produce the same output.

Work with the for loop

The **for** loop is commonly used in C#. Try this code in your Main() method:

```
for(int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

This does the same work as the `while` loop and the `do` loop you've already used. The `for` statement has three parts that control how it works.

The first part is the **for initializer**: `for index = 0;` declares that `index` is the loop variable, and sets its initial value to `0`.

The middle part is the **for condition**: `index < 10` declares that this `for` loop continues to execute as long as the value of counter is less than 10.

The final part is the **for iterator**: `index++` specifies how to modify the loop variable after executing the block following the `for` statement. Here, it specifies that `index` should be incremented by 1 each time the block executes.

Experiment with these yourself. Try each of the following:

- Change the initializer to start at a different value.

- Change the condition to stop at a different value.

When you're done, let's move on to write some code yourself to use what you've learned.

Combine branches and loops

Now that you've seen the `if` statement and the looping constructs in the C# language, see if you can write C# code to find the sum of all integers 1 through 20 that are divisible by 3. Here are a few hints:

- The `%` operator gives you the remainder of a division operation.
- The `if` statement gives you the condition to see if a number should be part of the sum.
- The `for` loop can help you repeat a series of steps for all the numbers 1 through 20.

Try it yourself. Then check how you did. You should get 63 for an answer. You can see one possible answer by viewing the completed code on GitHub.

You've completed the "branches and loops" quickstart.

You can continue with the String interpolation quickstart in your own development environment.

You can learn more about these concepts in these topics:

If and else statement
While statement
Do statement
For statement

String interpolation

7/23/2018 • 7 minutes to read • Edit Online

This quickstart teaches you how to use C# string interpolation to insert values into a single result string. You write C# code and see the results of compiling and running it. The quickstart contains a series of lessons that show you how to insert values into a string and format those values in different ways.

This quickstart expects that you have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the introduction to the local quickstarts with links to more details. You also can complete the interactive version of this quickstart in your browser.

Create an interpolated string

Create a directory named **interpolated**. Make it the current directory and run the following command from a console window:

```
dotnet new console
```

This command creates a new .NET Core console application in the current directory.

Open **Program.cs** in your favorite editor, and replace the line `Console.WriteLine("Hello World!");` with the following code, where you replace `<name>` with your name:

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Try this code by typing `dotnet run` in your console window. When you run the program, it displays a single string that includes your name in the greeting. The string included in the WriteLine method call is an *interpolated string*. It's a kind of template that lets you construct a single string (called the *result string*) from a string that includes embedded code. Interpolated strings are particularly useful for inserting values into a string or concatenating (joining together) strings.

This simple example contains the two elements that every interpolated string must have:

- A string literal that begins with the `$` character before its opening quotation mark character. There can't be any spaces between the `$` symbol and the quotation mark character. (If you'd like to see what happens if you include one, insert a space after the `$` character, save the file, and run the program again by typing `dotnet run` in the console window. The C# compiler displays an error message, "error CS1056: Unexpected character '$'".)
- One or more *interpolated expressions*. An interpolated expression is indicated by an opening and closing brace (`{` and `}`). You can put any C# expression that returns a value (including `null`) inside the braces.

Let's try a few more string interpolation examples with some other data types.

Include different data types

In the previous section, you used string interpolation to insert one string inside of another. The result of an interpolated expression can be of any data type, though. Let's include values of various data types in an interpolated string.

In the following example, first, we define a class data type `Vegetable` that has the `Name` property and the `ToString` method, which overrides the behavior of the Object.ToString() method. The `public` access modifier makes that method available to any client code to get the string representation of a `Vegetable` instance. In the example the `Vegetable.ToString` method returns the value of the `Name` property that is initialized at the `Vegetable` constructor:

```
public Vegetable(string name) => Name = name;
```

Then we create an instance of the `Vegetable` class by using `new` keyword and providing a name parameter for the constructor `Vegetable`:

```
var item = new Vegetable("eggplant");
```

Finally, we include the `item` variable into an interpolated string that also contains a DateTime value, a Decimal value, and a `Unit` enumeration value. Replace all of the C# code in your editor with the following code, and then use the `dotnet run` command to run it:

```
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };

   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per {unit}.");
   }
}
```

Note that the interpolated expression `item` in the interpolated string resolves to the text "eggplant" in the result string. That's because, when the type of the expression result is not a string, the result is resolved to a string in the following way:

- If the interpolated expression evaluates to `null`, an empty string ("", or String.Empty) is used.
- If the interpolated expression doesn't evaluate to `null`, typically the `ToString` method of the result type is called. You can test this by updating the implementation of the `Vegetable.ToString` method. You might not even need to implement the `ToString` method since every type has some implementation of this method. To test this, comment out the definition of the `Vegetable.ToString` method in the example (to do that, put a comment symbol, `//`, in front of it). In the output, the string "eggplant" is replaced by the fully qualified type name ("Vegetable" in this example), which is the default behavior of the Object.ToString() method. The default behavior of the `ToString` method for an enumeration value is to return the string representation of

the value.

In the output from this example, the date is too precise (the price of eggplant doesn't change every second), and the price value doesn't indicate a unit of currency. In the next section, you'll learn how to fix those issues by controlling the format of string representations of the expression results.

Control the formatting of interpolated expressions

In the previous section, two poorly formatted strings were inserted into the result string. One was a date and time value for which only the date was appropriate. The second was a price that didn't indicate its unit of currency. Both issues are easy to address. String interpolation lets you specify *format strings* that control the formatting of particular types. Modify the call to `Console.WriteLine` from the previous example to include the format strings for the date and price expressions as shown in the following line:

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

You specify a format string by following the interpolated expression with a colon (":") and the format string. "d" is a standard date and time format string that represents the short date format. "C2" is a standard numeric format string that represents a number as a currency value with two digits after the decimal point.

A number of types in the .NET libraries support a predefined set of format strings. These include all the numeric types and the date and time types. For a complete list of types that support format strings, see Format Strings and .NET Class Library Types in the Formatting Types in .NET article.

Try modifying the format strings in your text editor and, each time you make a change, rerun the program to see how the changes affect the formatting of the date and time and the numeric value. Change the "d" in `{date:d}` to "t" (to display the short time format), "y" (to display the year and month), and "yyyy" (to display the year as a four-digit number). Change the "C2" in `{price:C2}` to "e" (for exponential notation) and "F3" (for a numeric value with three digits after the decimal point).

In addition to controlling formatting, you can also control the field width and alignment of the formatted strings that are included in the result string. In the next section, you'll learn how to do this.

Control the field width and alignment of interpolated expressions

Ordinarily, when the result of an interpolated expression is formatted to string, that string is included in a result string without leading or trailing spaces. Particularly when you work with a set of data, being able to control a field width and text alignment helps to produce a more readable output. To see this, replace all the code in your text editor with the following code, then type `dotnet run` to execute the program:

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

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
   }
}
```

The names of authors are left-aligned, and the titles they wrote are right-aligned. You specify the alignment by adding a comma (",") after an interpolated expression and designating the *minimum* field width. If the specified value is a positive number, the field is right-aligned. If it is a negative number, the field is left-aligned.

Try removing the negative signs from the `{"Author",-25}` and `{title.Key,-25}` code and run the example again, as the following code does:

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

This time, the author information is right-aligned.

You can combine an alignment specifier and a format string for a single interpolated expression. To do that, specify the alignment first, followed by a colon and the format string. Replace all of the code inside the `Main` method with the following code, which displays three formatted strings with defined field widths. Then run the program by entering the `dotnet run` command.

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}] [{1063.342,15:N2}] feet");
```

The output looks something like the following:

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

You've completed the string interpolation quickstart.

You can continue with the List collection quickstart in your own development environment.

For more information, see the String interpolation topic and the String interpolation in C# tutorial.

C# Quickstart: Collections

5/4/2018 • 5 minutes to read • Edit Online

This quickstart provides an introduction to the C# language and the basics of the List<T> class.

This quickstart expects you to have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the introduction to the local quickstarts with links to more details.

A basic list example

Create a directory named **list-quickstart**. Make that the current directory and run `dotnet new console`.

NOTE

If you just completed Get started with .NET in 10 minutes, you can keep using the myApp application that you just created.

Open **Program.cs** in your favorite editor, and replace the existing code with the following:

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

namespace list_quickstart
{
    class Program
    {
        static void Main(string[] args)
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Replace `<name>` with your name. Save **Program.cs**. Type `dotnet run` in your console window to try it.

You've just created a list of strings, added three names to that list, and printed out the names in all CAPS. You're using concepts that you've learned in earlier quickstarts to loop through the list.

The code to display names makes use of the string interpolation feature. When you precede a `string` with the `$` character, you can embed C# code in the string declaration. The actual string replaces that C# code with the value it generates. In this example, it replaces the `{name.ToUpper()}` with each name, converted to capital letters, because you called the ToUpper method.

Let's keep exploring.

Modify list contents

The collection you created uses the List<T> type. This type stores sequences of elements. You specify the type of the elements between the angle brackets.

One important aspect of this List<T> type is that it can grow or shrink, enabling you to add or remove elements. Add this code before the closing `}` in the `Main` method:

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

You've added two more names to the end of the list. You've also removed one as well. Save the file, and type `dotnet run` to try it.

The List<T> enables you to reference individual items by **index** as well. You place the index between `[` and `]` tokens following the list name. C# uses 0 for the first index. Add this code directly below the code you just added and try it:

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

You cannot access an index beyond the end of the list. Remember that indices start at 0, so the largest valid index is one less than the number of items in the list. You can check how long the list is using the Count property. Add the following code at the end of the Main method:

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Save the file, and type `dotnet run` again to see the results.

Search and sort lists

Our samples use relatively small lists, but your applications may often create lists with many more elements, sometimes numbering in the thousands. To find elements in these larger collections, you need to search the list for different items. The IndexOf method searches for an item and returns the index of the item. Add this code to the bottom of your `Main` method:

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
} else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
} else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");

}
```

The items in your list can be sorted as well. The Sort method sorts all the items in the list in their normal order (alphabetically in the case of strings). Add this code to the bottom of our `Main` method:

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Save the file and type `dotnet run` to try this latest version.

Before you start the next section, let's move the current code into a separate method. That makes it easier to start working with a new example. Rename your `Main` method to `WorkingWithStrings` and write a new `Main` method that calls `WorkingWithStrings`. When you have finished, your code should look like this:

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

namespace list_quickstart
{
    class Program
    {
        static void Main(string[] args)
        {
            WorkingWithStrings();
        }

        public static void WorkingWithStrings()
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine();
            names.Add("Maria");
            names.Add("Bill");
            names.Remove("Ana");
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine($"My name is {names[0]}");
            Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

            Console.WriteLine($"The list has {names.Count} people in it");

            var index = names.IndexOf("Felipe");
            Console.WriteLine($"The name {names[index]} is at index {index}");

            var notFound = names.IndexOf("Not Found");
            Console.WriteLine($"When an item is not found, IndexOf returns {notFound}");

            names.Sort();
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Lists of other types

You've been using the `string` type in lists so far. Let's make a List<T> using a different type. Let's build a set of numbers.

Add the following to the bottom of your new `Main` method:

```
var fibonacciNumbers = new List<int> {1, 1};
```

That creates a list of integers, and sets the first two integers to the value 1. These are the first two values of a *Fibonacci Sequence*, a sequence of numbers. Each next Fibonacci number is found by taking the sum of the previous two numbers. Add this code:

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach(var item in fibonacciNumbers)
    Console.WriteLine(item);
```

Save the file and type `dotnet run` to see the results.

TIP

To concentrate on just this section, you can comment out the code that calls `WorkingWithStrings();`. Just put two `/` characters in front of the call like this: `// WorkingWithStrings();`.

Challenge

See if you can put together some of the concepts from this and earlier lessons. Expand on what you've built so far with Fibonacci Numbers. Try to write the code to generate the first 20 numbers in the sequence. (As a hint, the 20th Fibonacci number is 6765.)

Complete challenge

You can see an example solution by looking at the finished sample code on GitHub

With each iteration of the loop, you're taking the last two integers in the list, summing them, and adding that value to the list. The loop repeats until you've added 20 items to the list.

Congratulations, you've completed the list quickstart. You can continue with the Introduction to classes quickstart in your own development environment.

You can learn more about working with the `List` type in the .NET Guide topic on collections. You'll also learn about many other collection types.

Introduction to classes

5/22/2018 • 8 minutes to read • Edit Online

This quickstart expects that you have a machine you can use for development. The .NET topic Get Started in 10 minutes has instructions for setting up your local development environment on Mac, PC or Linux. A quick overview of the commands you'll use is in the introduction to the local quickstarts with links to more details.

Create your application

Using a terminal window, create a directory named **classes**. You'll build your application there. Change to that directory and type `dotnet new console` in the console window. This command creates your application. Open **Program.cs**. It should look like this:

```
using System;

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

In this quickstart, you're going to create new types that represent a bank account. Typically developers define each class in a different text file. That makes it easier to manage as a program grows in size. Create a new file named **BankAccount.cs** in the **classes** directory.

This file will contain the definition of a ***bank account***. Object Oriented programming organizes code by creating types in the form of ***classes***. These classes contain the code that represents a specific entity. The `BankAccount` class represents a bank account. The code implements specific operations through methods and properties. In this quickstart, the bank account supports this behavior:

1. It has a 10-digit number that uniquely identifies the bank account.
2. It has a string that stores the name or names of the owners.
3. The balance can be retrieved.
4. It accepts deposits.
5. It accepts withdrawals.
6. The initial balance must be positive.
7. Withdrawals cannot result in a negative balance.

Define the bank account type

You can start by creating the basics of a class that defines that behavior. It would look like this:

```
using System;

namespace classes
{
    public class BankAccount
    {
        public string Number { get; }
        public string Owner { get; set; }
        public decimal Balance { get; }

        public void MakeDeposit(decimal amount, DateTime date, string note)
        {
        }

        public void MakeWithdrawal(decimal amount, DateTime date, string note)
        {
        }
    }
}
```

Before going on, let's take a look at what you've built. The `namespace` declaration provides a way to logically organize your code. This quickstart is relatively small, so you'll put all the code in one namespace.

`public class BankAccount` defines the class, or type, you are creating. Everything inside the `{` and `}` that follows the class declaration defines the behavior of the class. There are five ***members*** of the `BankAccount` class. The first three are ***properties***. Properties are data elements and can have code that enforces validation or other rules. The last two are ***methods***. Methods are blocks of code that perform a single function. Reading the names of each of the members should provide enough information for you or another developer to understand what the class does.

Open a new account

The first feature to implement is to open a bank account. When a customer opens an account, they must supply an initial balance, and information about the owner or owners of that account.

Creating a new object of the `BankAccount` type means defining a ***constructor*** that assigns those values. A ***constructor*** is a member that has the same name as the class. It is used to initialize objects of that class type. Add the following constructor to the `BankAccount` type:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Constructors are called when you create an object using `new`. Replace the line `Console.WriteLine("Hello World!");` in ***program.cs*** with the following line (replace `<name>` with your name):

```
var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with {account.Balance} initial
balance.");
```

Type `dotnet run` to see what happens.

Did you notice that the account number is blank? It's time to fix that. The account number should be assigned when the object is constructed. But it shouldn't be the responsibility of the caller to create it. The `BankAccount` class code should know how to assign new account numbers. A simple way to do this is to start with a 10-digit number. Increment it when each new account is created. Finally, store the current account number when an object is constructed.

Add the following member declaration to the `BankAccount` class:

```
private static int accountNumberSeed = 1234567890;
```

This is a data member. It's `private`, which means it can only be accessed by code inside the `BankAccount` class. It's a way of separating the public responsibilities (like having an account number) from the private implementation (how account numbers are generated.) Add the following two lines to the constructor to assign the account number:

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Type `dotnet run` to see the results.

Create deposits and withdrawals

Your bank account class needs to accept deposits and withdrawals to work correctly. Let's implement deposits and withdrawals by creating a journal of every transaction for the account. That has a few advantages over simply updating the balance on each transaction. The history can be used to audit all transactions and manage daily balances. By computing the balance from the history of all transactions when needed, any errors in a single transaction that are fixed will be correctly reflected in the balance on the next computation.

Let's start by creating a new type to represent a transaction. This is a simple type that doesn't have any responsibilities. It needs a few properties. Create a new file named ***Transaction.cs***. Add the following code to it:

```
using System;

namespace classes
{
    public class Transaction
    {
        public decimal Amount { get; }
        public DateTime Date { get; }
        public string Notes { get; }

        public Transaction(decimal amount, DateTime date, string note)
        {
            this.Amount = amount;
            this.Date = date;
            this.Notes = note;
        }
    }
}
```

Now, let's add a List<T> of `Transaction` objects to the `BankAccount` class. Add the following declaration:

```
private List<Transaction> allTransactions = new List<Transaction>();
```

The List<T> class requires you to import a different namespace. Add the following at the beginning of **BankAccount.cs**:

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

Now, let's change how the `Balance` is reported. It can be found by summing the values of all transactions. Modify the declaration of `Balance` in the `BankAccount` class to the following:

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

This example shows an important aspect of ***properties***. You're now computing the balance when another programmer asks for the value. Your computation enumerates all transactions, and provides the sum as the current balance.

Next, implement the `MakeDeposit` and `MakeWithdrawal` methods. These methods will enforce the final two rules: that the initial balance must be positive, and that any withdrawal must not create a negative balance.

This introduces the concept of ***exceptions***. The standard way of indicating that a method cannot complete its work successfully is to throw an exception. The type of exception and the message associated with it describe the error. Here, the `MakeDeposit` method throws an exception if the amount of the deposit is negative. The `MakeWithdrawal` method throws an exception if the withdrawal amount is negative, or if applying the withdrawal results in a negative balance:

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

The `throw` statement **throws** an exception. Execution of the current method ends, and will resume when a matching `catch` block is found. You'll add a `catch` block to test this code a little later on.

The constructor should get one change so that it adds an initial transaction, rather than updating the balance directly. Since you already wrote the `MakeDeposit` method, call it from your constructor. The finished constructor should look like this:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now is a property that returns the current date and time. Test this by adding a few deposits and withdrawals in your `Main` method:

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "friend paid me back");
Console.WriteLine(account.Balance);
```

Next, test that you are catching error conditions by trying to create an account with a negative balance:

```
// Test that the initial balances must be positive:
try
{
    var invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
}
```

You use the `try` and `catch` statements to mark a block of code that may throw exceptions, and to catch those errors that you expect. You can use the same technique to test the code that throws for a negative balance:

```
// Test for a negative balance
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Save the file and type `dotnet run` to try it.

Challenge - log all transactions

To finish this quickstart, you can write the `GetAccountHistory` method that creates a `string` for the transaction history. add this method to the `BankAccount` type:

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    report.AppendLine("Date\t\tAmount\tNote");
    foreach (var item in allTransactions)
    {
        report.AppendLine($"{item.Date.ToShortDateString()}\t{item.Amount}\t{item.Notes}");
    }

    return report.ToString();
}
```

This uses the StringBuilder class to format a string that contains one line for each transaction. You've seen the string formatting code earlier in these quickstarts. One new character is `\t`. That inserts a tab to format the output.

Add this line to test it in **Program.cs**:

```
Console.WriteLine(account.GetAccountHistory());
```

Type `dotnet run` to see the results.

Next Steps

If you got stuck, you can see the source for this quickstart in our GitHub repo

Congratulations, you've finished all our Quickstarts. If you're eager to learn more, try our tutorials