Introduction to the C# Language:  
Language, Variables, Output, and Input

# C# and .NET

C# is a language that was created by Microsoft in 2000. Microsoft developed it in response to the growing popularity of Java. It is an object-oriented, type-safe, and managed programming language.

.NET (pronounced "dot net") is a software development platform that supports several different languages, editors, and libraries. It is designed to be cross-platform, meaning that you can write your website, server, or console app once in any of the supported languages and be able to run it on Windows, Mac, or Linux. Also, using Xamarin / Mono, you can use a .NET language to create a mobile application.

# Writing C# programs using Microsoft Visual Studio 2022

The easiest way to write a C# program is to use Microsoft Visual Studio 2022. The Community Edition is free to use for non-commercial use. [Download](https://visualstudio.microsoft.com/vs/community/) and install it, and then configure it to use “Visual C#” as the Development Settings.

Graphical user interface, application

Description automatically generated

# Create a new C# Project

Next, create a new C# project. Follow these screen shots to do that, clicking on "Create a new project".

Graphical user interface, application

Description automatically generated

Then click on Console App (C#) to create a new command-line application.

Graphical user interface, text, application

Description automatically generated

Then fill out the next dialog like the following screen shot. You can choose whatever name you'd like for the project, whatever directory you'd like for the location, and whatever name you'd like for the solution. Note that a "solution" in C# is just a container for one or more C# projects.

Graphical user interface, text, application

Description automatically generated

Next choose the Framework. You can think of a framework as a "generation" or "version" of classes. I recommend you choose .NET 6.0 or higher from the dropdown. Make sure you click the “Do not use top-level statements” checkbox so that Visual Studio doesn’t hide some important syntax pieces for us. (More on that point in the next section.)

A screenshot of a computer program

Description automatically generated

Click "Create". A code window will open like the one shown below.

A screen shot of a computer

Description automatically generated

The code you see will be part of a file called Program.cs. The ".cs" extension identifies this as a C-sharp program.

To run this code, click the green-outlined arrow, the one that has the pop-up hint that reads "Start Without Debugging (Ctrl + F5).

A picture containing diagram

Description automatically generated

The code will build – meaning it will be converted from the high-level language C# to an intermediate language called Common Language Runtime (CLR) – and then it will execute in a console window that pops up to show you the result.

Text

Description automatically generated

# Top-level statements and the code that actually comprises your application

What we just wrote is a C# program using the .NET 8 framework. Version 6 and later of the framework enables you to write only the statements that have to be executed. It achieves this through what are called *top-level statements*. Remember we checked the box to not use top-level statements? If we had used top-level statements, all we’d have to type is the one Console.WriteLine statement that prints the message.

But what we wrote is the whole shebang – the code that is actually part of your codebase. It shows your dependencies, the namespace you’re storing your code in, the name of the class, the Main function where your application starts, and then the instructions that are executed. All of these are important, which is why I’d like you always to check the box to not use top-level statements.

using System;

namespace edu.lewisu.cs.cpsc230

{

internal class MyFirstProgram

{

static void Main(string[] args)

{

Console.WriteLine("Hello, world!");

}

}

}

Here are some notes on what this code means.

*using System;*

System is a package or library that contains the definition for Console.

*Console*

Console is a class. A class is a type of thing in a programming language that both stores data and performs tasks. Console, for example, can WriteLine values to the console window so that they appear for the user to read.

Console.WriteLine("Hello, World!");

The WriteLine function of the Console class writes the text inside the parentheses to the terminal window followed by an end-of-line marker so that then next output appears on the next line. There is also a Console.Write("some text") function that writes the text but without following it with an end-of-line marker, so that the next text you write will start where your last text left off.

*namespace edu.lewisu.cs.cpsc230 { ... }*

A namespace is an organizational structure that enables you to differentiate your code from someone else's code. It is therefore advantageous to name namespaces in a way that makes them unique. A value that is guaranteed to be unique is your Internet domain name. The popular convention is to name a namespace starting with your Internet domain name backwards, followed by additional distinguishing labels separated by periods. Since our domain is cs.lewisu.edu, I started the namespace here with "edu.lewisu.cs" and followed it up with "cpsc230".

internal class MyFirstProgram { ... }

As an object-oriented program, a C# program consists of class definitions. You declare a class using the *class* keyword, followed by the name of the class. The modifier “internal” labels the class as being known only within this program. Other programs, in other words, won’t be able to call this function. It is a good idea to label your Main class either *public* or *internal*. The preference in the community is to use *internal* so that the code you write is specific to just this project. Use public if the code you’re writing has to be part of other projects too.

static void Main(string[] args) { ... }

The code that executes when a program launches is contained in the Main function. The Main function has a *void* return type. It must take in an argument (i.e. a value passed to it) that is a list of strings. A string is a sequence of characters, and a list of string values has the data type string[], where the [] distinguishes this as a list of string values rather than just a single one.

# Observations on C# statements

Every statement in C# ends in either a semicolon or an opening curly brace. If the statement introduces a block of related statements, such as a namespace, class, or function declaration, then the statement ends in an opening curly brace. Otherwise, the line is a self-contained one and ends in a semicolon.

Notice the capitalization conventions C# uses:

* Keywords like “static” and “void” are lowercase
* Function names like “Main” appear in TitleCase, meaning they start with a capital letter and each subsequent word in the name likewise starts with a capital letter.
* Class names like “MyFirstProgram” appear in TitleCase
* Simple data types like “string” and “int” appear in lower case
* Variable names appear in camelCase, meaning the first letter is lower case and subsequent words in the name start with a capital letter.

# Inputting Strings from the User

The Console class also has a ReadLine function. The ReadLine function inputs all characters entered by the user – spaces and all – through the invisible end-of-line character that is added to the end of their input when they press the Enter key.

Remember that a string is a sequence of characters. It is implemented in C# as a class, so it has both data (the characters that it stores) and functions (things it can do).

To use ReadLine, you must store what the user enters in a variable. A variable is a symbol that represents a location in memory in which you want to store a value. To declare a variable, you have to specify its data type and its name like so:

datatype variableName;

For example, we can declare a string variable named personName like so:

string personName;

We can set it equal to what the user types at the command line using this statement:

personName = Console.ReadLine();

We can then use personName in subsequent statements. For example, we can print it as part of a greeting. To do this, we would use Console.WriteLine like we did before, but we would concatenate the fixed part of the greeting with the value of personName using a + sign. When used with string values, the + sign merges the string values on its left and its right.

Here is code that demonstrates interacting with the user.

using System;

namespace edu.lewisu.cs.cpsc230

{

internal class MyFirstProgram

{

static void Main(string[] args)

{

Console.Write("Enter your name: ");

string personName;

personName = Console.ReadLine();

Console.WriteLine("It is good to meet you, " + personName + ".");

}

}

}

# Nullable data types

When you declare a data type, you can optionally indicate if the value is allowed to be null. To do that, you add a ? to the end of the data type name. For example, to declare a string that may be null, do this:

string? name;

Nullable types are useful because they allow you to use *null* to indicate that a value is unassigned. For example, we can define an int that is allowed to be nullable. An int that is nullable could store any integer value but also the value *null*. So, you can test if the value is null rather than using some bogus int value to convey that a value has not been assigned.

int? number = null;

# Testing for null values

To take full advantage of nullable values, it would be helpful to have a concise way to test whether they are equal to null. The ternary conditional operator in C# allows for this. For example, the following line will print “number not set” if number is null, or it will print the value of the number if it is not null.

Console.WriteLine(x == null ? “number not set” : x);

The syntax is the following:

condition\_you\_are\_testing ? what the value is if true : what the value is if false

As we’ll see in the next note set when we study conditional statements, you can use an if … else to do this. But the appeal of the ternary conditional operator is how compact it is.

# Inputting Other Types of Data from the User

By default, Console.ReadLine inputs a string value. However, it is possible to input other types of data by declaring variables of different types and using the services of the Convert class.

C# is a type-safe language, meaning that it rigorously checks that you are assigning values of the correct types to the variables you have declared. C# supports many of what are called *primitive* data types, or data types that are not classes, meaning they just store values. They are tabulated here:

Table

Description automatically generated

The Convert class has the ability to take a string and convert it to any of these data types. For example, Convert.ToInt32(someString) will convert the characters of someString to the integer they spell out, whereas Convert.ToDouble(someString) will convert the characters of someString to the double-precision floating point number they represent.

In the following program, we ask the user to enter their age in years and show them what their age in months is. We also ask them to enter their monthly earnings and show them how much they earn per year.

using System;

namespace edu.lewisu.cs.cpsc230

{

internal class MyFirstProgram

{

static void Main(string[] args)

{

Console.Write("Enter your name: ");

string personName;

personName = Console.ReadLine();

Console.WriteLine("It is good to meet you, " + personName + ".");

int age;

Console.Write("Enter your age in years: ");

age = Convert.ToInt32(Console.ReadLine());

int ageInMonths = age \* 12;

Console.WriteLine("I see you are " + age + " years old, or " +

ageInMonths + " months.");

double monthlyWage, yearlyWage;

Console.Write("Enter your monthly wage: ");

monthlyWage = Convert.ToDouble(Console.ReadLine());

yearlyWage = monthlyWage \* 12;

Console.WriteLine("You make $" + yearlyWage + " per year.");

}

}

}

# An alternative to Convert: the Parse function

Instead of using Convert.ToInt32 to convert a string value to an integer, you can use this:

int.Parse(someString);

That will return an int value.

Predictably, instead of using Convert.ToDouble to convert a string value to a double, you can use this:

double.Parse(someString);

This will return a double value.

# Converting data safely by catching exceptions

When you try to convert a string a user enters to a number, the program will crash if what the user enters can’t be converted. You can catch this situation gracefully and thus avoid a crash by using a try..catch block. Here’s how:

try {

string entered = Console.ReadLine();

int num = int.Parse(entered);

} catch (FormatException) {

Console.WriteLine(“The value you entered could not be converted to an int.”);

}

# Converting among numeric types

The Convert class is great for converting between strings and other data types. It is also possible to convert from one numeric type to another using typecasting. For example, suppose I have a double called yearlyPay:

double yearlyPay = 87032.42;

I can keep just the integer part of this by typecasting yearlyPay as shown here:

int trunctated = (int)yearlyPay;

In other words, you put the target data type in front of the value you want to convert to a different type.

# Formatted output

In addition to concatenating values to merge them with strings, you can embed values within a string. This will give you tighter control over how those values will appear when combined with the characters of the string.

To embed values with a string, embed {#} within the string.

For example, this code will print the person's name in *lastname, firstname* format.

using System;

namespace edu.lewisu.cs.cpsc230

{

internal class MyFirstProgram

{

static void Main(string[] args)

{

Console.WriteLine("Enter your first name: ");

string firstName = Console.ReadLine();

Console.WriteLine("Enter your last name: ");

string lastName = Console.ReadLine();

Console.WriteLine("Your name is {0}, {1}.", lastName, firstName);

}

}

}

This code will print the last name and first name right-justified in fields of width 20:

Console.WriteLine("Your name is {0,20}, {1,20}.", lastName, firstName);

This code will print the last name and first name left-justified in fields of width 20:

Console.WriteLine("Your name is {0,-20}, {1,-20}.", lastName, firstName);

If you have a floating point number – say, yearlyWage – and you want it to be printed with 2 digits after the decimal, you'd write this:

Console.WriteLine("Your annual wage is ${0:F2}.", yearlyWage);

The ":F2" part of the formatting sequence indicates that you want the yearlyWage value to be printed as a floating point value with two digits after the decimal point. The leading 0 means you don’t care how much space the number takes up.

Alternatively, by adding ":C" to the formatting sequence, you can print the value as currency, complete with commas between hundreds and thousands, an automatic currency symbol ($), and two digits after the decimal;

Console.WriteLine("Your annual wage is {0:C}.", yearlyWage);

We can also combine what we have learned about formatting to specify that the number should be printed with two digits after the decimal right-justified in a field of width 12. Note that in this example we re-introduce the $ symbol because we aren't using the automatic currency formatter.

Console.WriteLine("Your annual wage is ${0,12:F2}.", yearlyWage);

# An alternative way to format output

Let us now learn about a newer way to format output that is now supported by C#.

If you preface a quoted string with $, you can embed values to be printed – whether variables or literal values or calculations or function calls - within { } within the quoted string. Each value to be embedded would be formatted like this:

{ value, width:additionalformatting }

where "additionalformatting" could be F2, for example, for a floating-point number with two digits after the decimal, or C for currency. The width should be specified as positive if you want the value right-justified and negative if you want it left-justified.

Here's an example:

double x = 3 / 2.0;

Console.WriteLine($"{"Karen",-15}{x,-10:F2}Ray");

This prints "Karen" left-justified in a field of width 15, the value of x to two decimal places left-justified in a field of width 10, followed by "Ray".

You might prefer this way because it allows you to embed the actual values that are supposed to go in each position rather than require you to line up embedded { } with variables that follow the string to be printed.

# A shortcut involving Console

If you are getting tired of typing "Console" in front of Read, ReadLine, Write, and WriteLine, you could add this to the top of your program:

using static System.Console

You'll then be able to just type WriteLine instead of Console.WriteLine, for example.

# Practice problems

1. Write an application that processes payroll for a company. The application should ask for the employee’s name, payrate, and hours worked. It will then print a summary of the employee’s pay that shows their name, gross pay, taxes withheld and net pay. At the end, the program will print the total gross pay, taxes, and net pay that was disbursed nicely tabulated. The program must tolerate the user entering invalid data like character strings when numbers are expected.
2. You have been hired to write a program for a coffee shop to help generate receipts for their customers. The program should allow the user to input the details of an order, including the items purchased, their quantities, and prices. The program should then calculate the total cost, including a sales tax of **8%**, and display a formatted receipt. Here is an example of how the program should look:

Welcome to the Coffee Shop!

Enter the name of the first item: Latte

Enter the price of the Latte: 3.50

Enter the quantity of the Latte: 2

Enter the name of the second item: Muffin

Enter the price of the Muffin: 2.25

Enter the quantity of the Muffin: 3

Receipt:

----------------------------------------

Item Quantity Price Total

Latte 2 $3.50 $7.00

Muffin 3 $2.25 $6.75

----------------------------------------

Subtotal: $13.75

Sales Tax (8%): $ 1.10

Final Total: $14.85