C# 8 is designed to work the Microsoft .NET Core 3 runtime, and .NET Standard 2.1 .

A First C# Program
Here is a program that multiplies 12 by 30 and prints the result, 360, to the
screen. The double forward slash indicates that the remainder of a line is a
Comment:

```
using System;               // Importing namespace

class Test                  // Class declaration
{
  static void Main()          // Method declaration
  {
   int x = 12 * 30;          // Statement 1
   Console.WriteLine (x);    // Statement 2
  }                           // End of method
}                           // End of class
```

Statements in C# execute sequentially and are terminated by a semicolon

Main Method
C# recognizes a method called Main as signaling the default entry point of execution

Methods
Methods are one of several kinds of functions in C#. Another kind of function we used was the * operator, which performs multiplication.There are also constructors, Properties, events, indexers, and finalizers.

Console class

The Console class groups members that handle command-line input/output functionality, such as the WriteLine method.

Namespace
At the outermost level of a program, types are organized into namespaces.The using directive makes the System namespace available to our application to use the Console class. We could define all of our classes within the TestPrograms namespace, as follows:

```
using System;

namespace TestPrograms
{
  class Test  {...}
  class Test2 {...}
}
```

Compilation
The C# compiler compiles source code, specified as a set of files with the.cs extension, into an assembly, which is the unit of packaging and deployment in .NET. An assembly can be either an application or library, the difference being that an application has an entry point (“Main”method), whereas a library does not.

Syntax
C# syntax is inspired by C and C++ syntax. In this section, we describe C#’s elements of syntax, using the following program:

```
using System;

class Test
{
  static void Main()
```

```
 {
   int x = 12 * 30;
   Console.WriteLine (x);
 }
}
```

Identifiers and keywords
Identifiers are names that programmers choose for their classes, methods,variables, and so on. These are the identifiers in our example program :

System   Test   Main   x   Console   WriteLine

Keywords are names that mean something special to the compiler. These are the keywords in our example program:

using   class   static   void   int

Most keywords are reserved, which means that you can't use them as identifiers.

Avoiding conflicts
If you really want to use an identifier that clashes with a reserved keyword, you can do so by qualifying it with the @ prefix.

String Type

C#'s string type represents an

immutable (unmodifiable) sequence of Unicode characters. A string literal

:is specified within double quotes

string a = "Heat";

Escape sequences

The escape sequences that are valid for char literals also work within Strings

```
;"string a = "Here's a tab:\t
```

The cost of this is that whenever you need a literal backslash, you must write it twice

```
;"string a1 = "\\\\server\\fileshare\\helloworld.cs
```

Verbatim string

To avoid this problem, C# allows verbatim string literals. A verbatim

string literal is prefixed with @ and does not support escape sequences. The following verbatim string is identical to the preceding one

```
;"string a2 = @"\\server\fileshare\helloworld.cs
```

A verbatim string literal can also span multiple lines. you can include the

double-quote character in a verbatim literal by writing it twice.

String Concatenation

The + operator concatenates two strings , Using the + operator repeatedly to build up a string can be inefficient , a better solution is to use the System.Text.StringBuilder type—this

represents a mutable (editable) string, and has methods to efficiently

String Interpolation

A string preceded with the $ character is called an Interpolated string

:Interpolated strings can include expressions within braces

;int x = 4

Console.Write ($"A square has {x} sides");// Prints: A square has 4 sides

String Comparisons

string does not support <and> operators for comparisons. you must

instead use string’s *CompareTo* method, which returns a positive number

a negative number, or zero, depending on whether the first value comes

:after, before, or alongside the second value

```
Console.Write ("Boston".CompareTo ("Austin"));      // 1
Console.Write ("Boston".CompareTo ("Boston"));     // 0
Console.Write ("Boston".CompareTo ("Chicago"));  // -1
```

Manipulation String

Because string is immutable, all the methods that “manipulate” a string

return a new one, leaving the original untouched , Substring extracts a portion of a string.

Insert and Remove insert and remove characters at a specified Position

PadLeft and PadRight add whitespace.

TrimStart, TrimEnd, and Trim remove whitespace.

The string class also defines ToUpper and ToLower methods for changing

case, a Split method to split a string into substrings (based on supplied

delimiters), and a static Join method to join substrings back into a string.

Arrays

An array represents a fixed number of elements of a particular type. The
elements in an array are always stored in a contiguous block of memory
providing highly efficient access.

Char[ ] vowels = new char[ ] {'a','e','i','o','u'};
Or :
Char[ ] vowels = {'a','e','i','o','u'};

Arrays also implement IEnumerable<T>, so you can also enumerate members with the foreach statement.

All arrays inherit from the System.Array class,This includes properties
such as Length and Rank, and static methods to do the following:
Dynamically create an array (CreateInstance)

Get and set elements regardless of the array type
(GetValue/SetValue)
Search a sorted array (BinarySearch) or an unsorted array
(IndexOf,
LastIndexOf, Find, FindIndex, FindLastIndex)
Sort an array (Sort)
Copy an array (Copy)

Indices and Ranges

C# 8 introduces indices and ranges to simplify working with
elements or
portions of an array.

Indices
Indices let you refer to elements relative to the end of an array,
with the ^
operator. ^1 refers to the last element, ^2 refers to the second-to-
last
element, and so on:

```
Char[ ] vowels = new char[ ] {'a','e','i','o','u'};
char lastElement  = vowels[^1];   // 'u'
char secondToLast = vowels[^2];   // 'o'
```

Ranges
Ranges let you “slice” an array with the .. operator:

```
Char[ ] firstTwo =  vowels [..2];               // 'a', 'e'
Char[ ] lastThree = vowels [2..];      // 'i', 'o', 'u'
Char[ ] middleOne = vowels [2..3]     // 'i'
```

The second number in the range is exclusive, so ..2 returns the
elements
before vowels[2].

Multidimensional Arrays

Multidimensional arrays come in two varieties: rectangular and jagged.

Rectangular arrays
To declare rectangular arrays, use commas to separate each dimension.
The following declares a rectangular two-dimensional array, where the
dimensions are 3 × 3:

Int[,] matrix = new int [3, 3];

jagged arrays
To declare jagged arrays, use successive square-bracket pairs for each
dimension. Here is an example of declaring a jagged two-dimensional
array, for which the outermost dimension is 3:

Int[ ][ ] matrix = new int[3][ ] ;

Simplified Array Initialization Expressions

We've already seen how to simplify array initialization expressions by
omitting the new keyword and type declaration:

Char[ ] vowels = new char[ ] {'a','e','i','o','u'};
Char[ ] vowels = {'a','e','i','o','u'} ;

Another approach is to omit the type name after the new keyword, and
have the compiler infer the array type. This is a useful shortcut when
you're passing arrays as arguments. For example, consider the following
method:

void Foo (char[ ] data) { ... }

Variables and Parameters

A variable represents a storage location that has a modifiable value. A
variable can be a local variable, parameter (value, ref, out, or in), field
(instance or static), or array element.

The Stack and the Heap
The stack and the heap are the places where variables reside. Each has very different lifetime semantics.

Stack
The stack is a block of memory for storing local variables and parameters.The stack logically grows and shrinks as a method or function.

Heap
The heap is the memory in which objects reside. Whenever a new object is created, it is allocated on the heap, and a reference to that object is returned. During a program’s execution, the heap starts filling up as new objects are created. The runtime has a

garbage collector that periodically deallocates objects from the heap.

Definite Assignment
C# enforces a definite assignment policy. In practice, this means that outside of an unsafe context, it’s impossible to access uninitialized memory. Definite assignment has three implications:
<u>*Local variables*</u> must be assigned a value before they can be read.
<u>*Function arguments*</u> must be supplied when a method is called (unless marked optional).
<u>*All other variables*</u> (such as fields and array elements) are automatically initialized by the runtime.

Parameters
A method can have a sequence of parameters. Parameters define the set of arguments that must be provided for that method. In this example, the method Foo has a single parameter named p, of type Int:

```
static void Foo (int p)     // p is a parameter
{
        ...
}
static void Main()
{
        Foo (8);
} // 8 is an argument
```

You can control how parameters are passed with the **ref**, **out**, and **in** modifiers

Passing arguments by value
By default, arguments in C# are passed by value, which is by far the most common case. This means that a copy of the value is created when it is passed to the method:

```
static void Foo (int p)
{
        p = p + 1;                // Increment p by 1
        Console.WriteLine (p)    ;        // Write p to screen
}
        static void Main(){
         int x = 8; Foo (x);              // Make a copy of x
        Console.WriteLine (x);           // x will still be 8
}
```

Assigning p a new value does not change the contents of x, because p and x reside in different memory locations.
Passing a reference type argument by value copies the reference but not the object. In the following example, Foo sees the same StringBuilderobject that Main instantiated, but has an independent reference to it. In Other words, sb and fooSB are separate variables that reference the sameStringBuilder object:

```
static void Foo (StringBuilder fooSB)
{
        fooSB.Append ("test");
        fooSB = null;
}
static void Main()
{
        StringBuilder sb = new StringBuilder();
        Foo (sb);
        Console.WriteLine (sb.ToString()); // test
}
```

Because fooSB is a copy of a reference, setting it to null doesn’t make sb null. (If, however, fooSB was declared and called with the ref modifier,sb would become null.)

The ref modifier
To pass by reference, C# provides the ref parameter modifier. In the following example, p and x refer to the same memory locations:

```
static void Foo (ref int p)
{
        p = p + 1;
        Console.WriteLine (p);
}


static void Main()
{
         int x = 8;
        Foo (ref x);              // Pass x by reference
       Console.WriteLine (x);        // x is now 9
}
```

Now assigning p a new value changes the contents of x. Notice how the ref modifier is required both when writing and calling the method. This Makes it very clear what’s going on.NOTES parameter can be passed by reference or by value, regardless of whether the parameter type is a reference type or a value type.

The out modifier

An out argument is like a ref argument, except for the following:It need not be assigned before going into the function.It must be assigned before it comes out of the function.The out modifier is most commonly used to get multiple return values back from a method.