Lesson 1

=======================================

History

FORTRAN and COBOL were among the first high-level languages, introduced in the late 1950s. Both are still used today, FORTRAN for scientific applications and COBOL for business. Smalltalk, released around 1980, is a fully object-oriented language that influenced its successors, including Java and C#.

Systems programmers who needed access to the machine hardware used assembly languages, which are very low-level and specific to the hardware. The C language, developed in the mid 1970s, is sometimes described as a portable assembly language. It is a high-level language, like FORTRAN and COBOL, but provides access to machine hardware. The UNIX operating system, developed for the then-new minicomputers, was mostly written in C, and both C and UNIX rapidly grew in popularity.

Although it is good for systems programming, C is a small language that does not facilitate the development of large software systems. Introduced in the later 1960s, object-oriented programming started to become popular in the mid 1980s. Languages that support object-oriented programming do facilitate the development of large software systems. C++ extends C to include constructs that support object-oriented programming, while still including those that access machine hardware. Consequently, C++ grew in popularity.

BASIC was developed in the 1960s as an easier way for students to learn to program. It used an interpreter so students could immediately see the results of execution. Originally, personal computers had very limited memory chips in which to hold programs, so BASIC, which did not require compilation to a larger low-level representation, became the main language used on early PCs. As memory became

cheaper and graphics capabilities grew, BASIC morphed to Visual Basic, an extremely popular language for the rapid development of user applications.

With the introduction of the .NET Framework, Visual Basic evolved to Visual Basic .NET, a cousin of C#. One way we might describe C# is as a language that tries to combine the rapid application development of Visual Basic with much of the power of C++.

With the rise of desktop computers and the rapid growth of the Internet in the mid 1990s came the need for a language to support programming to enable users on vastly different systems to interact in a secure way. Java, introduced in 1995, uses a Java Virtual Machine to provide security and enable programs developed on different systems to interact. A large library extends its capabilities for Internet programming. Because it suited the new demands on developers, Java has become very popular.

Lesson 1

==

The goals of C# are similar to those of Java. Those versed in one of these languages can rapidly convert to using the other. C# had the advantage of seeing the Java approach and how it might enhance it. C# adds features for the easy development of components to make it easier for developers to combine programs written on different systems. One can annotate a C# program with attributes that become part of the runtime used by the CLR. This metadata describes the program so that other programs can use it. C#, newly developed in the twenty-first century, promises to become very popular as the primary .NET programming language.

Introduction to the C# Language and the .NET Framework

C# is an elegant and type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework. You can use C# to create Windows client applications, XML Web services, distributed components, client-server applications, database applications, and much, much more. Visual C# provides an advanced code editor, convenient user interface designers, integrated debugger, and many other tools to make it easier to develop applications based on the C# language and the .NET Framework.

C# Language

C# is a modern, general-purpose, object-oriented, high-level prog-ramming language. Its syntax is similar to that of C and C++ but many features of those languages are not supported in C# in order to simplify the language, which makes programming easier.
The C# programs consist of **one or several files** with a **.cs** extension, which contain definitions of classes and other types. These files are compiled by the C# compiler (**csc**) to executable code and as a result assemblies are created, which are files with the same name but with a different extension (**.exe** or **.dll**). For example, if we compile **HelloCSharp.cs**, we will get a file with the name **HelloCSharp.exe** (some additional files will be created as well, but we will not discuss them at the moment).
We can run the compiled code like any other program on our computer (by double clicking it). If we try to execute the compiled C# code (for example **HelloCSharp.exe**) on a computer that does not have the .NET Framework, we will receive an error message.

Lesson 1

==

C# - As an object-oriented language

As an object-oriented language, C# supports the concepts of encapsulation, inheritance, and polymorphism. All variables and methods, including the `Main` method, the application's entry point, are encapsulated within class definitions. A class may inherit directly from one parent class, but it may implement any number of interfaces. Methods that override virtual methods in a parent class require the `override` keyword as a way to avoid accidental redefinition. In C#, a struct is like a lightweight class; it is a stack-allocated type that can implement interfaces but does not support inheritance.

In addition to these basic object-oriented principles, C# makes it easy to develop software components through several innovative language constructs, including the following:

- Encapsulated method signatures called *delegates*, which enable type-safe event notifications.
- Properties, which serve as accessors for private member variables.
- Attributes, which provide declarative metadata about types at run time.
- Inline XML documentation comments.
- Language-Integrated Query (LINQ) which provides built-in query capabilities across a variety of data sources.

THE .NET FRAMEWORK

Microsoft developed the C# language along with the .NET Framework, a new computing platform that simplifies application development in the distribution environment of the Internet. They designed the .NET Framework to fulfill the following objectives[1]:

■ To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.

■ To provide a code-execution environment that minimizes software deployment and versioning conflicts

■ To provide a code-execution environment that guarantees safe execution of code, including code created by an unknown or semi trusted third party

■ To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments

Lesson 1

- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code

The .NET Framework objective meets the needs of professional developers. This text presents only a small part of the .NET Framework, but we cannot climb the mountain until we take the first steps.

The two main parts of the .NET Framework are the Common Language Runtime and the .NET Framework class library.

The Common Language Runtime

The Common Language Runtime (CLR) manages the execution of code and provides services that make the execution of code easier. “Runtime” means that code is running, which is another way of saying it is being executed. “Common Language” means that this runtime manages the execution of code written in several languages that share the services provided.

Microsoft developed C# to take advantage of the CLR. Its features work especially well with the CLR. The popular Visual Basic language evolved to Visual Basic .NET, which is an object-oriented language that takes advantage of the CLR. Visual Basic programmers have to learn many new features to take advantage of the CLR using Visual Basic .Net. C++, like its C predecessor, has many capabilities that do not fit into the new approach. A version of C++, called managed C++, adapts C++ to work with the CLR, so C++ programmers can integrate code with other CLR users.

We call code that uses the CLR managed code. The Common Type System defines the types of data that managed code can use. A Common Language Specification (CLS) defines features that every language for developing managed code must provide. Programmers who use only features in the Common Language Specification can build an application combining programs in different languages. They will know that if they use a feature in one language, a program in another language will also be able to use that same construct.

To give a simplified example, consider numbers. Suppose we have space for only 10 numbers. If we use only nonnegative numbers, we can use 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. If we need to include both positive and negative values, we would choose -5, -4, -3, -2, – 1, 0, 1, 2, 3, and 4. The unsigned type allows us to use larger values, but the signed type allows us to use negative values. In an analogous situation, the Common Type System provides both unsigned and signed types, but only requires that languages provide signed types to satisfy the Common Language Specification.

A developer who uses only signed types can expect every language that follows the CLS to interoperate with the code produced. A developer who uses unsigned types may produce code that another language that follows the CLS is not required to support.

A big problem facing developers is the many different types of processors that run code. Windows, Macintosh, and Unix machines use a wide variety of hardware, as do personal digital assistants, cell phones, large computers, and other platforms. One way to make a program work on each of these devices is to translate the program to the native instruction set for that device. Say we have 10 programming languages and 10 devices. Using this approach, in the worst case2 we would need 100 compilers to translate programs in each of the 10 languages to native code for each of the 10 devices. Figure 1.4 diagrams this jumble of compilers.

Another approach, used by the CLR, is to provide an intermediate language that is much like the native languages of devices. This language is called MSIL, Microsoft Intermediate Language. We compile each language to MSIL. During runtime the CLR uses a JIT (Just In Time) compiler to compile the MSIL code to the native code for the device used.3 This requires one JIT compiler for each device to translate MSIL code to the native code for that device. This translation process is not as difficult as translating a high-level language to native code, because the MSIL code is similar to native code.

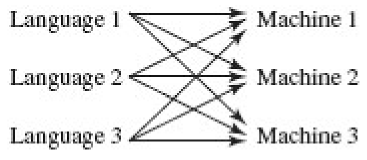

FIGURE 1.4 Compiling three languages to native code for three machines

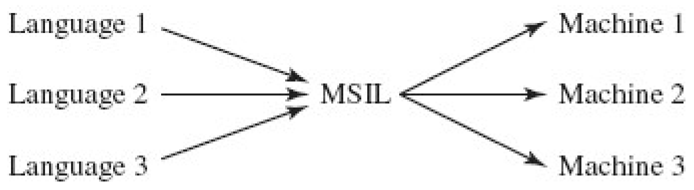

FIGURE 1.5 Compiling using an intermediate language

Moreover, using MSIL greatly reduces the number of compilers we need. For 10 languages and 10 devices, we need one compiler for each language to translate it to MSIL, and one JIT compiler for each device to translate MSIL code to native code for that device. Thus we need 20 compilers instead of 100. Figure 1.5 illustrates this approach.

The .NET Framework Class Library

The .NET Framework Class Library provides a large and very useful set of types that expedite the development process. The library groups the types in namespaces that combine related types. It contains about 100 namespaces of which we use only a small number in this text. Some of the namespaces from the library that we use are:

`System`
Contains fundamental types

`System.Collections`
Defines various collections of objects

`System.Data`
Manages data from multiple sources including databases

`System.Drawing`
Provides graphics

`System.IO`
Allows reading and writing

`System.Net`
Provides an interface computers use to communicate over networks

`System.Runtime.Remoting`
Supports distributed applications

`System.Text`
Handles character encoding

`System.Threading`
Enables multithreaded programming

`System.Web`
Enables browser–server communication

`System.Web.Services`
Enables the building and use of web services

`System.Windows.Forms`
For user interfaces in Windows-based applications

`System.Xml`
Provides support for processing XML

Lesson 1

==

How C# Works

C# uses a compiler, but does not immediately translate a high-level program to the machine instructions of each specific processor. The C# compiler takes as input the

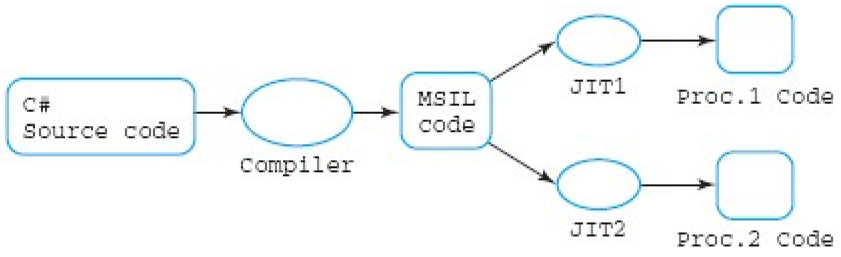

FIGURE 1.6 Compiling a C# program

high-level C# program and outputs an equivalent program written using the Microsoft Intermediate Language. During runtime the CLR uses a JIT compiler to translate the MSIL code to the instruction set of the processor. Figure 1.6 shows the compilation of a C# program to MSIL followed by the JIT compilation to native code on two different processors.

The advantage of producing MSIL code instead of code for each different processor is that the same intermediate code will run on any processor that has a CLR. We can download a C# program which has been compiled to intermediate code on one type of machine and run it on a much different type of machine. For networking, this feature of C# is invaluable.

In December 2001, ECMA (European Computer Manufacturers Association, a body that promulgates information technology standards) released the ECMA-334 standard for C#.

1.4 THE ELEMENTS OF A C# PROGRAM

We can describe natural language on many levels. For example, this text, written in the English language, uses the Latin character set. So, at the lowest level, we see letters ‘a', ‘b', ‘c',…. We also see punctuation, including commas and periods,

and blank spaces. Characters group into words, words form sentences, and sentences form paragraphs. On a higher level, we can discuss the parts of speech: nouns, verbs, adjectives, adverbs, and so on. Moreover, the text has content that it is trying to communicate.

Lesson 1

==

Programming languages have basic components that we form into larger units to build useful applications. Example 1.1 shows a simple C# program that will illustrate some of the elements of the C# programming language.

EXAMPLE 1.1 ■ Square.cs

```
/* Computes the square of a number.
 * Displays the result in a message.
 */

public class Square {
   // Execution starts here.

  static void Main() {
    int number = 345;
                                // '=' denotes assignment
    int squared = number * number;
                                // '*' denotes
                                // multiplication
    System.Console.WriteLine
          ("The square of {0} is {1}", number, squared);
  }
}
```

```
The square of 345 is 119025
```

Output

The square of 345 is 119025

Lexical Structure

The lexical structure identifies C# rules for dividing the program text into a sequence of input elements. Input elements include comments, whitespace, identifiers, keywords, literals, punctuators, and operators. Whitespace and comments help people to read programs but are omitted from the compiled program. We discuss each type of input element briefly.

Example 1.1 uses the familiar Latin characters, although C# can handle much larger character sets. C# is case sensitive, meaning that it distinguishes lowercase characters from uppercase. In the early days of computing character sets were small, and languages such as FORTRAN did not make that distinction.

Lesson 1

==

Whitespace includes spaces, tabs, and carriage returns that we use to format the text. Formatting programs nicely makes them much easier to read and modify, but C# does not require it. Example 1.1 would be correct if none of the lines were indented, but it would be very unpleasant for humans to read. Line structure is not important in C#. The line

```
int number = 345;
```

could have been replaced by

```
int number
          = 345;
```

without changing its meaning.

A **comment** provides information for the human reader, but is not part of the executable code. Example 1.1 uses two styles of comments. Two forward slashes, //, mark the remaining text on that line as a comment. For comments that span multiple lines, we use /*, with no space between the two characters, to start the comment, and */ to end it. Programmers use comments to clarify the program for human readers. A general-purpose programming language like C# must use very general constructs that programmers can apply to build diverse applications. Comments help to identify what the C# code is trying to accomplish.

Example 1.1 uses **punctuators** such as the left brace, {, the right brace,}, and the semicolon,; . The full list of punctuators is

```
( ) { } [ ] ; , . :
```

Operators symbolize operations such as addition and multiplication. C# includes many operators, some of which require two or three characters to represent. Example 1.1 uses the *, +, and = operators. The asterisk, *, represents the multiplication operator.

Literals represent specific values. In Example 1.1, the literal 345 represents an integer. The literal “The square of {0} is {1}” represents a character string.

Identifiers are like the words of the program. Some identifiers are names chosen by the programmer, and others are **keywords**, reserved for special uses. Keywords in Example 1.1 include public, static, class, int, and void. Identifiers include Square and number. Figure 1.7 shows the lexical analysis phase of compilation. The lexical analyzer takes each line of the C# program and divides it into **tokens**, which are identifiers, keywords, literals, punctuators, or operators. It discards comments and whitespace.

Lesson 1

==

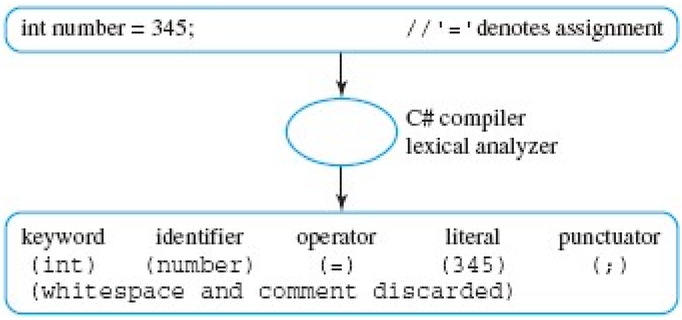

FIGURE 1.7 Lexical analysis of a single line of Example 1.1

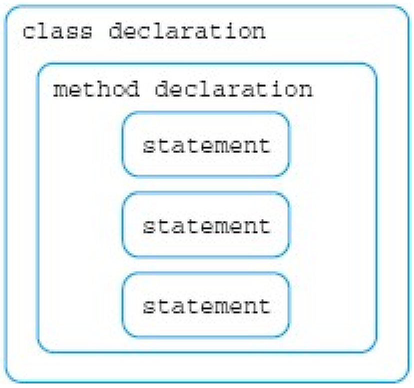

FIGURE 1.8 The overall syntactic structure of Example 1.1

Syntax

Syntax describes the grammatical rules for combining tokens into programs. In English, we form sentences from words. The sentence

```
I like to program
```

makes a declaration, and the sentence

```
Do you like to program?
```

asks a question.

C# syntactic structures include declarations, statements, and expressions. A **declaration** defines a part of a program.**Statements** control the sequence of execution of the program.**Expressions**compute values and have other effects. Figure 1.8 shows the overall syntactic structure of Example 1.1.

Example 1.1 declares a **class** named Square. Ignoring the outer comment, we see that, at the top level, the structure of Example 1.1 is this class declaration. Class is a key concept in C# and we shall have a lot to say about it. Programming languages use words in precise ways, in contrast to natural language, which can be vague. Nevertheless, C# uses the word class in ways that remind us of the English word class in the concept of classification. A class in that sense identifies a concept or type. We might classify animals into dogs, cats, horses, and so on.

A bacterium is a very simple type of animal compared to an elephant. We do not go to the zoo to see bacteria. Similarly, the Square class of Example 1.1 is a very limited example of a C# class. We shall later see C# classes that do define types that are more interesting. Roughly speaking, Example 1.1 does illustrate that a C# program often consists of a class declaration at its outer level.

A class contains member declarations. The Square class contains a **method** declaration for a method named Main. A method is a way of accomplishing something. We often name methods to signify what the method will accomplish. For example, Shuffle might name a method to shuffle a deck of cards. The name Main signifies the method that C# executes first.

The declaration of the Main method in Example 1.1 contains several parts. The Main method starts with "static void". The modifier static means that the Main method is part of the class definition. To understand the contrasting choice, consider the definition of an elephant, which describes a type of large animal. The Latin name for the African elephant, Loxodonta africana, is part of the definition of an elephant, but it is not part of an actual elephant. Later, when we declare more interesting classes, we shall see the analog of actual elephants, called instances or objects.

The modifier **void** indicates that the Main method does not return a value to its caller. Other methods may return to their callers a result of what they have accomplished.

The parentheses that immediately follow the method name, (), contain the names and types of any data that the method uses to accomplish its goal. The Main method in Example 1.1 does not use any additional data, so we leave this part of the declaration empty.

The remainder of Example 1.1 consists of the **body** of the Main method, which is a **block** of code. The body contains the code that enables the Main method to accomplish its purpose of computing and displaying the square of a number. A block contains a sequence of statements within braces, { }. The block in Example 1.1 contains three statements. We will discuss the details of the construction and meaning of these three statements later. At first glance, the first statement introduces a value of 345, the second statement squares that value, and the third statement displays the squared value. These statements contain expressions.

An **expression** specifies the computation of a value. For example, the expression

Lesson 1

==

```
7 * 7
```

specifies a computation which produces the result 49. Example 1.1 uses the multiplication operator.

Our First C# Program

Before we continue with an in depth description of the C# language and the .NET platform, let's take a look at a simple example, illustrating how a **program written in C#** looks like: `class`

```
HelloCSharp
{
static void Main(string[] args)
 {
       System.Console.WriteLine("Hello C#!");
  }
}
```

How Does Our First C# Program Work?

Our first program consists of three logical parts:
- Definition of a class **`HelloCSharp`**;
- Definition of a method **`Main()`**;
- Contents of the method **`Main()`**.

Defining a Class

On the first line of our program we define a class called **`HelloCSharp`**. The simplest definition of a class consists of the keyword **`class`**, followed by its name. In our case the name of the class is **`HelloCSharp`**. The content of the class is located in a block of program lines, surrounded by curly brackets: **`{}`**.

Lesson 1

==

Defining the Main() Method

On the third line we define a method with the name **`Main()`**, which is the starting point for our program. Every program written in C# starts from a **`Main()`** method with the following title (signature):

```
static void Main(string[] args)
```

C# Distinguishes between Uppercase and Lowercase!

The C# language distinguishes between **uppercase** and **lowercase** letters so we should use the correct casing when we write C# code. In the example above we used some keywords like **`class`**, **`static`**, **`void`** and the names of some of the system classes and objects, such as **`System.Console`**.

Be careful when writing! The same thing, written in upper-case, lower-case or a mix of both, means different things in C#. Writing `Class` is different from `class` and `System.Console` is different from `SYSTEM.CONSOLE`.

The Program Code Must Be Correctly Formatted

Formatting is adding characters such as spaces, tabs and new lines, which are insignificant to the compiler and they give the code a **logical structure** and make it **easier to read**.

Main Formatting Rules

If we want our code to be correctly formatted, we must follow several important **rules regarding indentation**:
- Methods are **indented** inside the definition of the class (move to the right by one or more **[Tab]** characters);

- Method contents are indented inside the definition of the method;

- The opening curly bracket **{** must be on its own line and placed exactly under the method or class it refers to;

- The closing curly bracket **}** must be on its own line, placed exactly vertically under the respective opening bracket (with the same indentation);

- All class names must start with a capital letter;

- Variable names must begin with a lower-case letter;

- Method names must start with a capital letter;

Code indentation follows a very simple rule: when some piece of code is logically inside another piece of code, it is indented (moved) on the right with a single [Tab]. For example if a method is defined inside a class, it is indented (moved to the right). In the same way if a method body is inside a method, it is indented. To simplify this, we can assume that when we have the character "**{**", all the code after it until its closing "**}**" should be indented on the right.

File Names Correspond to Class Names

Every C# program consists of **one or several class definitions**. It is accepted that each class is defined in a separate file with a name corresponding to the class name and a **.cs** extension. When these requirements are not met, the program will still work but navigating the code

will be difficult. In our example, the class is named **HelloCSharp**, and as a result we must save its source code in a file called **HelloCSharp.cs**.

Compilation and Execution of C# Programs

The time has come to **compile and execute** the simple example program written in C# we already discussed. To accomplish that, we need to do the following:
- Create a file named **HelloCSharp.cs**;

- Write the sample program in the file;

- Set the Path variable pointing to C:\Windows\Microsoft.NET\Framework\v4.0.30319

C:\> PATH = C:\Windows\Microsoft.NET\Framework\v4.0.30319

- Compile **HelloCSharp.cs** to an executable file **HelloCSharp.exe** using the console-based C# compiler (**csc.exe**);

- Execute the **HelloCSharp.exe** file.