C#: Language Overview

C# is an object-oriented, component-oriented programming language. C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components. Since its origin, C# has added features to support new workloads and emerging software design practices. Several C# features aid in the construction of robust and durable applications.

- **Garbage collection** automatically reclaims memory occupied by unreachable unused objects.
- **Exception handling** provides a structured and extensible approach to error detection and recovery.
- **Lambda expressions** support functional programming techniques.
- **Query syntax** creates a common pattern for working with data from any source.
- Language support for **asynchronous operations** provides syntax for building distributed systems.
- **Pattern matching** provides syntax to easily separate data from algorithms in modern distributed systems.
- C# has a **unified type system**. All C# types, including primitive types such as *int* and *double*, inherit from a single root *object* type. All types share a set of common operations. Values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined reference types and value types. C# allows dynamic allocation of objects and in-line storage of lightweight structures.

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

The "Hello, World" program starts with a **using directive** that references the **System namespace**. **Namespaces** provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the System namespace contains a number of types, such as the Console class referenced in the program, and a number of other namespaces, such as IO and Collections. A using directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the using directive, the program can use Console.WriteLine as shorthand for System.Console.WriteLine.

The **Hello class** declared by the "Hello, World" program has a single member, the method named Main. The **Main method** is declared with the **static modifier**. While instance methods can reference a particular

enclosing object instance using the keyword this, static methods operate without reference to a particular object. By convention, a static method named Main serves as the entry point of a C# program.

The output of the program is produced by the **WriteLine method of the Console class** in the System namespace. This class is provided by the standard class libraries, which, by default, are automatically referenced by the compiler.

Program structure

The key organizational concepts in C# are **programs**, **namespaces**, **types**, **members**, and **assemblies**. Programs declare types, which contain members and can be organized into namespaces. Classes, structs, and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they're physically packaged into assemblies. Assemblies typically have the file extension .exe or .dll, depending on whether they implement applications or libraries, respectively.

C# programs can be stored in several source files. When a C# program is compiled, all of the source files are processed together, and the source files can freely reference each other. Conceptually, it's as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with few exceptions, declaration order is insignificant. C# doesn't limit a source file to declaring only one public type nor does it require the name of the source file to match a type declared in the source file.

C# programs are organized using **namespaces**. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system—a way of presenting program elements that are exposed to other programs. Using directives are provided to facilitate the use of namespaces. Namespaces are heavily used in C# programming in two ways. First, .NET uses namespaces to organize its many classes, as follows:

```
System.Console.WriteLine("Hello World!");
```

System is a namespace and Console is a class in that namespace. The using keyword can be used so that the complete name is not required, as in the following example:

```
using System;

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

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
```

```
    }
}
```

The name of the namespace must be a valid C# identifier name. An **identifier** is a variable name. An identifier is a sequence of unicode characters without any whitespace. An identifier may be a C# reserved word, if it is prefixed by @. That can be useful when interacting with other languages.

Assemblies form the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- For libraries that target the .NET Framework, you can share assemblies between applications by putting them in the global assembly cache (GAC). You must strong-name assemblies before you can include them in the GAC.
- Assemblies are only loaded into memory if they are required. If they aren't used, they aren't loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection.
- You can load an assembly just to inspect it by using the MetadataLoadContext class in .NET Core and the Assembly.ReflectionOnlyLoad or Assembly.ReflectionOnlyLoadFrom methods in .NET Core and .NET Framework.