### C# Concepts

In this chapter you examine some basic concepts in C#. The purpose of this analysis is to get you up to speed on the terminology and ideas that we will be expanding on later in the book. This chapter is worth a quick read even if you’re familiar with, say, C++ or Java.

##### Application Startup

Let’s begin by looking at how application startup works in C#.

An application starts to run when the execution environment calls a designated method, called the

*entry point*. This entry point is always called Main. The entry point can take on one of four signatures:

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

As you can see, it is possible for the entry point to return an int value that can be used during application termination.

It is possible for the entry point to have one and only one parameter. This parameter can be called anything, but it has to conform to the following rules:

* + The value of the parameter cannot be null.
  + If you call the parameter args and if the length of the array designated by args is greater than zero, the array members args[0] through args[args.Length-1], inclusive, will be strings called *application parameters*. These are supplied with implementation-defined values by the host environment prior to the application being started (think of command- line arguments).

There are also a few simple rules related to the Main method:

* A program can only contain one Main method entry point. Multiple definitions through over- loading are not allowed.
* The entry point cannot be a generic class declaration or a generic struct declaration.

##### Application Termination

You’ve looked at application startup; now you’ll look at application termination.

Application termination is where control is returned to the execution environment. If the return type of the application’s entry point method is set to int, the value returned will be the application’s termina- tion status code. This code allows the execution environment to determine whether the termination was successful or not.

If the return type of the entry point method is void, reaching the right closing brace (}), which ends the method, or executing a return statement that has no expression will both result in a termination status code of 0.

At the point just before an application termination, finalizers (see Chapter 3) for all of the objects used that have not yet been dealt with by garbage collection are called (unless this is suppressed).

##### C# Declarations

Declarations in C# are used to define separate aspects of a C# program. C# programs are built around a number of declarations:

* **Type declarations.** Used to define classes, delegates, enums, interfaces, and structs
* **Namespace declarations.** Contain type declarations and nested namespace declarations
* **Various other declarations.** For example, class declarations, which can contain declarations such as:
  + Constants
  + Events
  + Fields
  + Finalizers
  + Indexers
  + Instance constructors
  + Methods
  + Nested types
    - Operators
    - Properties
    - Static constructors

A declaration defines a name in the declaration space to which the declaration belongs. A compiler error will be generated if two or more declarations introduce members with the same name in a declaration space, unless:

* Two or more namespace declarations with the same name are allowable in the same declaration space. When this is the case, the individual namespace declarations are combined to form a sin- gle logical namespace with a single declaration space.
* A namespace declaration and one or more type declarations in the same declaration space can have the same name as long as the type declarations all have a minimum of one type parameter.
* Two or more methods with the same name but with different signatures are allowed in the same declaration.
* Two or more type declarations with the same name but different numbers of type parameters are allowed in the same declaration space.
* Two or more type declarations with the partial modifier in the same declaration space can have the same name, the same number of type parameters, and the same classification. These are combined into a single declaration space.

Declarations in separate programs but in the same namespace declaration space are allowed to have the same name.

*A type declaration space can never contain different kinds of members that have an identical name.*

There are a number of different kinds of namespace declarations:

* Within the source files of a program, namespace-member-declarations with no enclosing namespace-declaration are members of a single combined declaration space called the global declaration space.
* Within the source files of a program, namespace-member-declarations within namespace- declarations that have the same fully qualified namespace name are members of a single combined declaration space.
* Each compilation-unit and namespace-body has an alias declaration space. The extern- alias-directive and using-alias-directive of the compilation-unit or namespace- body contributes a member to the alias declaration space.
* Each nonpartial class, struct, or interface declaration creates a new declaration space. Each partial class, struct, or interface declaration contributes to a declaration space shared by all matching parts in the same program. All the names are introduced into this declaration space through the type-parameter-list and class-member-declarations, struct-member-declarations, or interface-member-declarations. With the exception of overloaded instance constructor declarations and static constructor declarations, a class or struct member declaration are not able to introduce a member by the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Also, a class or struct permits the declara- tion of overloaded instance constructors, operators, and types.
* Each enumeration declaration creates a new declaration space. The names are introduced into the declaration space through enum-member-declarations.
* Every block or switch block creates a declaration space for local variables and local constants called the local variable declaration space. Names are introduced into this declaration space through local-variable-declarations and local-constant-declarations.
* Every block or switch block creates a separate declaration space for labels called the label decla- ration space of the block. All names are introduced into this declaration space through labeled-statements, and the names are referenced through goto-statements.

The order in which the names are declared is usually of no significance. For example, the order is not significant for the declaration and use of:

* Constants
* Events
* Finalizers
* Indexers
* Instance constructors
* Methods
* Namespaces
* Operators
* Properties
* Static constructors
* Types

However, declaration order is significant in the following circumstances:

* Declaration order for field declarations and local variable declarations determines the order in which any initializers are executed.
* Local variables and local constants have to be defined before they are used.

Declaration order for enum member declarations is important when constant-expression values are not present.

##### Members

Namespaces and types all have members. Members of a type can either be declared in the type or inher- ited from the base class of the type.

When a type inherits from a base class, all members of the base class (except finalizers, instance con- structors, and static constructors) become members of the derived type.

The declared accessibility of a base class member does not control whether the member’s inherited- inheritance covers any member that isn’t an instance constructor, static constructor, or finalizer.

Namespace Members

Any namespaces and types that don’t have an enclosing namespace are members of the global namespace. Any namespaces and types declared within a namespace are members of that namespace.

Namespaces have no access restrictions and are always publicly accessible. You cannot declare private, protected, or internal namespaces.

Struct Members

The members of a struct are the members declared in the struct and the members inherited from the direct base class of the struct System.ValueType and the indirect base class object.

Enumeration Members

The members of any enumeration are the constants declared in the enumeration itself and the members inherited from the direct base class System.Enum of the enumeration, along with the indirect base classes System.ValueType and object.

Class Members

The members of a class are the members declared in the class along with the members inherited from the base class.

The members inherited from the base class include all of the following of the base class:

* + Constants
  + Events
  + Fields
  + Indexers
  + Methods
  + Operators
  + Properties
  + Types

The following are not included:

* + Finalizers
  + Instance constructors
  + Static constructors

*Base class members are inherited irrespective of their accessibility.*

A class declaration can contain the following declarations:

* Constants
* Events
* Fields
* Finalizers
* Indexers
* Instance constructors
* Methods
* Operators
* Properties
* Static constructors
* Types

Interface Members

Members of an interface are the members declared in the interface along with those declared in the base interfaces of the interface.

Array Members

All the members of an array are inherited from class System.Array, which is the abstract base type of all array types.

Delegate Members

All the members of a delegate are inherited from class System.Delegate. Delegates will be covered in greater detail in later chapters.

##### Member Access

Member declarations are allowed control over member access using declared accessibility (covered in the following section). When access is allowed, the member is accessible; otherwise, it is inaccessible.

Declared Accessibility

Declared accessibility of a member can be set to one of the following five categories:

* **Public.** In this case, access is not limited.
* **Protected.** Access is limited to the containing class or type derived from the containing class.
  + **Internal.** Access is limited to the program.
  + **Protected internal.** Access is limited to the program or types derived from the containing class.
  + **Private.** Access is limited to the containing type.

When a member declaration does not include any access modifiers, there is a default declared accessibility:

* + Namespaces implicitly have public declared accessibility (in fact, no access modifiers are allowed on namespace declarations).
  + Types declared in compilation units or namespaces default to internal declared accessibility.
  + Class members default to private declared accessibility.
  + Struct members default to private declared accessibility.
  + Interface members implicitly have public declared accessibility (no access modifiers are allowed).
  + Enumeration members implicitly have public declared accessibility (no access modifiers are allowed).

##### Signatures

In C#, all indexes, instance constructors, methods, and operators are characterized by their signature. The following sections provide a rundown of the signature of each of these.

Index Signatures

The signature of an indexer is made up of the type of each of its formal parameters. They are processed in left-to-right order.

The signature of an indexer does not include the element type or parameter names. Additionally, it does not include the params modifier that can be specified for the right-most parameter.

Instance Constructor Signatures

The signature of an instance constructor is made up of the type and style of the parameters (that is, whether it is value, reference, or output). They are processed in left-to-right order.

The signature of an instance constructor does not include the parameter names or the params modifier specified for the right-most parameter.

Method Signatures

The signature of a method is made up of the following:

* + The name of the method
  + The number of type parameters
  + The type and style of the parameters (that is, whether it is value, reference, or output)

They are processed in left-to-right order.

Note that the signature of a method does not include the following:

* Return type
* Parameter names
* Type parameter names
* The params modifier that can be specified for the right-most parameter

Operator Signatures

The signature of an operator is made up of the name of the operator and the type of each of the parame- ters. They are processed in left-to-right order.

The signature of an operator does not include the following:

* Result type
* Parameter names

Signatures and Overloading

Signatures are a mechanism that allows for the overloading of members in classes, interfaces, and structs.

Overloading Indexers

Overloading indexers allows a class, interface, or struct to declare multiple indexers as long as their sig- natures are unique within that class, interface, or struct.

Overloading Instance Constructors

Overloading instance constructors allows a class or struct to declare multiple instance constructors as long as their signatures are different within that class or struct.

Overloading Methods

Overloading a method allows a class, interface, or struct to declare multiple methods where each has the same name as long as their signatures are different within the class, interface, or struct.

Overloading Operators

Overloading operators allows a class or struct to declare multiple operators with the same name as long as their signatures are different within that class or struct.

##### Scope

Scope is a term used in programming to describe the region of code within a program where it is possi- ble to refer to an entity that’s been declared without having to qualify the name.

It is possible for various scopes to be nested, and an inner scope can declare again the meaning of a name from an outer scope. In this case, the name from the outer scope is hidden in the region of code covered by the inner scope. Furthermore, access to the outer name is possible only by qualifying the name.

Here are the rules governing scope:

* + The scope of a namespace member declared by a namespace-member-declaration that has no enclosing namespace-declaration is the entire program.
  + The scope of a namespace member declared by a namespace-member-declaration within a namespace-declaration that has the fully qualified name is N (a shorthand representation) is the namespace-body of every namespace-declaration that has the fully qualified name is N or starts with N and is followed by a period.
  + The scope of a namespace member declared by a namespace-member-declaration that has no enclosing namespace-declaration is the entire program.
  + The scope of a namespace member declared by a namespace-member-declaration within a namespace-declaration that has the fully qualified name is N is the namespace-body of every namespace-declaration that has the fully qualified name N or starts with N and is fol- lowed by a period.
  + The scope of a name defined by an extern-alias-directive covers the using-directives, global-attributes, and namespace-member-declarations of the compilation-unit or namespace-body where the extern-alias-directive is found.
  + The scope of a name defined by a using-directive covers the global-attributes and namespace-member-declarations of the compilation-unit or namespace-body in which the using-directive is found.
  + The scope of a member declared by a class-member-declaration is the class-body where the declaration is found. The scope of a class member also extends to the class-body of derived classes included in the accessibility domain of the member.
  + The scope of a member declared by a struct-member-declaration is the struct-body

where the declaration is found.

* + The scope of a member declared by an enum-member-declaration is the enum-body where the declaration is found.
  + The scope of a parameter declared in a method-declaration is the method-body of that

method-declaration.

* + The scope of a parameter declared in an indexer-declaration is the accessor-declarations

of that indexer-declaration.

* + The scope of a parameter declared in an operator-declaration is the block of that

operator-declaration.

* + The scope of a parameter declared in a constructor-declaration is the constructor- initializer and block of that constructor-declaration.
  + The scope of a label declared in a labeled-statement is the block in which the declaration occurs.
* The scope of a local variable declared in a local-variable-declaration is the block in which the declaration occurs.
* The scope of a local variable declared in a switch-block of a switch statement is the switch

block.

* The scope of a local variable declared in a for-initializer of a for statement is the

for-initializer, the for-condition, and the for-iterator, along with the contained statement of the for statement.

* The scope of a local constant declared in a local-constant-declaration is the block in which the declaration is found.

##### Namespace and Type Names

A number of contexts in a C# program require a namespace-name or a type-name to be specified. The following shows the syntax for namespaces and type names.

namespace-name:

namespace-or-type-name

type-name:

namespace-or-type-name

namespace-or-type-name:

identifier type-argument-listopt qualified-alias-member

namespace-or-type-name . identifier type-argument-listopt

The namespace-or-type-name of a namespace-name has to refer to a namespace. Type arguments cannot be in a namespace-name.

A type-name is a namespace-or-type-name that refers to a type. Following resolution as described in the following section, the namespace-or-type-name of a type-name has to refer to a type.

##### Memory Management in C#

C# has at its core a rigorous memory management scheme built into the .NET Framework. This means that programmers have to write less code. Automatic memory-management policies are carried out by the garbage collector, and these policies mean that the programmer doesn’t have to manually allocate and free memory used by objects.

Here is the general lifecycle of an object:

1. The object is created.
2. Memory is allocated for the object.
3. The constructor is run.
4. The object is now live.
5. If the object is no longer in use (other than running finalizers), it needs finalization.
6. Finalizers are run (unless overridden).
7. The object is now inaccessible and is available for the garbage collector to carry out clean-up.
8. The garbage collector frees up associated memory.

##### Summary

In this chapter you looked at a number of key concepts in C#.

* Application startup
* Application termination
* Declarations
* Members
* Member access
* Signatures
* Overloading
* Scope
* Namespaces and type names
* Memory management

In Chapter 6, you look at C# types.