Chapter 9

The C# Type System

The C# Type System

Objectives

After completing this unit you will be able to:

- **Describe the difference between *reference* and *value* types in C#.**
- **Use *structs*, *class*es, and *enum*s in your C# programs.**
- **Specify what default values are assigned automatically by the compiler.**
- **Explain how *boxing* and *unboxing* work.**

Overview Of Types In C#

- **In C# there are three kinds of types:**
  - Value types
  - Reference types
  - Pointer types
- **Value types directly contain their data.**
  - Each variable of a value type has its own copy of the data.
  - Value types are typically allocated on the stack and are automatically destroyed when the variable goes out of scope.
  - Value types include the simple types discussed in Chapter 4, structures, and enumeration types.
- **Reference types do not contain data directly but only refer to data.**
  - Variables of reference types store references to data, called objects.
  - Two different variables can reference the same object.
  - Reference types are allocated on the **managed heap** and eventually get destroyed through a process known as **garbage collection**.
- **Reference types include *string*, *object*, class types, array types, interfaces, and delegates.**
- **Pointer types will be discussed in Chapter 20.**

Value Types

- **In this section we will survey all the value types, beginning with a review of the simple data types discussed in Chapter 4.**
  - We will see that there is a correspondence between these C# types and types in the Common Language Runtime, as expressed by classes in the **System** namespace.
  - We will also discuss structures and enumeration types.

Simple Types

- **The simple data types are general-purpose value data types, including numeric, character, and Boolean.**
  - The **sbyte** data type is an 8-bit signed integer.
  - The **byte** data type is an 8-bit unsigned integer.
  - The **short** data type is a 16-bit signed integer.
  - The **ushort** 16-bit unsigned integer.
  - The **int** data type is a 32-bit signed integer.
  - The **uint** 32-bit unsigned integer.
  - The **long** data type is a 64-bit signed integer.
  - The **ulong** 64-bit unsigned integer.
  - The **char** data type is a Unicode character (16 bits).
  - The **float** data type is a single-precision floating point.
  - The **double** data type is a double-precision floating point.
  - The **bool** data type is a Boolean (true or false).
  - The **decimal** data type is a decimal type with 28 significant digits (typically used for financial purposes).

Types in System Namespace

- **There is an exact correspondence between the simple C# types and types in the System namespace.**
  - C# reserved words are simply aliases for the corresponding type in the System namespace.
  - The table shows this correspondence.

C# Reserved Word	Type in System Namespace
sbyte	**System.SByte**
byte	**System.Byte**
short	**System.Int16**
ushort	**System.UInt16**
int	**System.Int32**
uint	**System.UInt32**
long	**System.Int64**
ulong	**System.UInt64**
char	**System.Char**
float	**System.Single**
double	**System.Double**
bool	**System.Boolean**
decimal	**System.Decimal**

Structures

- **A *struct* is a value type, which can group different types together.**
  - It can also have constructors and methods, although it is semantically different from a *class*, which is a reference type.
- **These basic features are illustrated in the program *HotelCreate*.**
  - There are fields **city**, **name**, **rooms**, and **cost**. There is a constructor, and there is a method **Show**.
  - A **struct** object is created using the **new** operator.

```
Hotel ritz = new Hotel("Boston", "Ritz", 100,
200.00m);
```

  - A **struct** object can also be created without **new**, but then the fields will be unassigned, and the object cannot be used until the fields have been initialized.

```
Hotel flop;
flop.city = "Podunk";
// Now it is OK to use the city field
flop.name = "Flop";
flop.rooms = 50;
flop.cost = 30.00m;
// Now it is OK to use the complete object
flop.Show();
```

Uninitialized Variables

- **The C# compiler will detect attempts to use uninitialized variables.**
  - A struct object cannot be used until its fields have been assigned.
  - A simple local variable must be initialized before it can be used.

```
int x;
Console.WriteLine("x = {0}", x);    // error
```

Test Program

- **See *HotelCreate\HotelCreate.cs* for a test program that exercises the *Hotel* structure.**

```
// HotelCreate.cs

using System;

public class HotelCreate
{
   public static void Main()
   {
      Hotel ritz = new Hotel("Boston", "Ritz", 100,
         200.00m);
      ritz.Show();
      Hotel flop;
      flop.city = "Podunk";
      // Now it is OK to use the city field
      flop.name = "Flop";
      flop.rooms = 50;
      flop.cost = 30.00m;
      // Now it is OK to use the complete object
      flop.Show();
      // Attempt to use an uninitialized variable
      int x;
      x = 5;      // NEED this initialization
      Console.WriteLine("x = {0}", x);
   }
}
```

- **Note that a constructor is invoked by *new*, but no constructor is involved when a struct object is created simply by declaring it.**

Copying a Structure

- **When you assign one *struct* variable to another, you will get two independent copies of the same data.**
  - If you subsequently change one copy, the two copies will be different.
  - This is in contrast to a class, where if you do an assignment, you will have two references to the same data.
  - The program **HotelCopy** illustrates a number of aspects of copying structures.
  - The class definition now has a special constructor for making a copy of a **Hotel** object. When do you think it will be invoked?
- **Two new objects, *flop* and *fancy*, are created by copying, shown by the comments #1 and #2 in the listing.**
  - In both cases an independent copy is made, and changing the value of one copy will not affect the value of the other copy.
  - Copy #1 does not involve any constructor; there is simply an implicit memberwise copy of all the elements of the **struc**t.
  - In the case of #2, we explicitly invoke the additional constructor by using the **new** operator, and any copying that gets done is performed within that constructor.
  - To demonstrate, we did not perform a perfect copy, but incremented the cost by $1 for the copy.

Classes and Structs

- **While in C++ the concept of *class* and *struct* is very close, there is more fundamental difference in C#.**
- **In C++ a class has default visibility of private and a struct has default visibility of public, and that is the only difference.**
- **In C# the key difference between a class and a struct is that a class is a *reference* type and a struct is a *value* type.**
  - A class must be instantiated explicitly using **new**.
  - The new instance is created on the heap, and memory is managed by the system through a garbage collection process.
  - A struct instance may simply be declared, or you may use **new**.
  - For a struct the new instance is created on the stack, and the instance will be deallocated when it goes out of scope.
- **There are different semantics for assignment, whether done explicitly or via call by value mechanism in a method call.**
  - For a class, you will get a second object reference, and both object references refer to the same data.
  - For a struct, you will get a completely independent copy of the data in the struct.

Enumeration Types

- **The final kind of value type is an enumeration type.**
  - An enumeration type is a distinct type with named constants.
- **Every enumeration type has an underlying type, which is one of the following.**
  - **byte**
  - **short**
  - **int**
  - **long**.

Enumeration Types Examples

- **An enumeration type is defined through an *enum* declaration.**

```
public enum BookingStatus : byte
{
    HotelNotFound,        // 0 implicitly
    RoomsNotAvailable,    // 1 implicitly
    Ok = 5           // explicit value
}
```

  - If the type is not specified, int is used.
  - By default, the first enum member is assigned the value 0, the second member 1, and so on.
  - Constant values can be explicitly assigned.
  - You can make use of an enumeration type by declaring a variable of the type indicated in the enum declaration (e.g., **BookingStatus**).
  - You can refer to the enumerated values by using the dot notation. Here is some illustrative code:

```
BookingStatus status;
status = hotel.ReserveRoom(name, date);
if (status == BookingStatus.HotelNotFound)
   Console.WriteLine("Hotel not found");
...
```

Reference Types

- **A variable of a reference type does not directly contain its data, but instead provides a reference to the data stored elsewhere (the heap). In C# there are the following kinds of reference types:**
  - Class
  - Array
  - Interface
  - Delegate
  - Reference types have a special value **null**, which indicates the absence of an instance.

Class Types

- **A class type defines a data structure that has fields, methods, constants, and other kinds of members.**
  - Class types support inheritance.
  - Through inheritance a derived class can extend or specialize a base class.
  - We introduced C# classes in the previous chapter, and we will discuss inheritance and other details about classes in later chapters.
- **There are two classes in the .NET Framework Class Library that are so important that they have C# reserved words as aliases for them: *object* and *string*.**

object

- **The *object* class type is the ultimate base type for all types in C#.**
- **Every C# type derives directly or indirectly from *object*.**
  - The object keyword in C# is an alias for the predefined **System.Object** class.
  - **System.Object** has methods such as **ToString**, **Equals**, and **Finalize**, which we will study later.

string

- **The *string* class encapsulates a Unicode character string.**
  - The **string** keyword is an alias for the predefined **System.String** class.
  - The string type is a **sealed** class. (A sealed class is one that cannot be used as the base class for any other classes.)
- **The *string* class inherits directly from the root *object* class.**
  - String literals are defined using double quotes.
  - There are useful built-in methods for **string**.
  - For now, note that the **Equals** method can be used to test for equality of strings.

```
string a = "hello";
if (a.Equals("hello"))
    Console.WriteLine("equal");
else
    Console.WriteLine("not equal");
```

  - There are also overloaded operators:

```
if (a == "hello")
    ...
```

  - We will study **string** in detail in Chapter 11.

Arrays

- **An array is a collection of elements that are all of the same type.**
  - Arrays are accessed using a square bracket and an index.
  - In C# array indices start at 0, as in other C family languages.
  - We previewed arrays in Chapter 6 and will study arrays in detail in Chapter 12.

Interfaces

- **The purpose of an interface is to specify a contract independently of implementation.**
- **Like a class, an interface has *methods*.**
  - But whereas a class provides an implementation of its methods, an interface only specifies them.
  - A class may implement one or more interfaces, specified by using a colon notation.

```
public class Acme : ICustomer, IHotelInfo,
IHotelReservation
{
...
```

  - We will study interfaces in detail in Chapter 17.

Delegates

- **The purpose of a delegate is to provide a "callback" behavior in an object-oriented, type-safe manner.**
  - In C/C++ you would use a function pointer, which is neither object-oriented nor type safe.
- **In C# you can encapsulate a reference to a method inside a delegate object.**
  - You can then pass this delegate object to other code, which can then call your method.
  - The code that calls your method does not have to know at compile time which method is being called.
  - We will study delegates in detail in Chapter 19.

Default Values

- **Several kinds of variables are automatically initialized to default values:**
  - Static variables
  - Instance variables of class and struct instances
  - Array elements
  - Local variables are not automatically initialized, as we saw earlier in the chapter.
  - The default value of a variable of reference type is **null**.

Default Values (Cont'd)

- **The default value of a variable of value type is the value assigned in the default constructor.**
  - For simple types this value corresponds to a bit pattern of all zeros:
  - For integer types, the default value is 0
  - For **char**, the default value is '\u0000'
  - For **float**, the default value is 0.0f
  - For **double**, the default value is 0.0d
  - For **decimal**, the default value is 0.0m
  - For **bool**, the default value is **false**.
  - For an **enum** type, the default value is 0.
  - For a **struct** type, the default value is obtained by setting all value type fields to their default values, as described above, and all reference type fields to **null**.

Boxing And Unboxing

- **One of the strong features of C# is that is has a unified type system.**
  - Every type, including the simple built-in types such as **int**, derive from **System.Object**.
  - In C# "everything is an object."
  - A language such as Smalltalk also has such a feature, but pays the price of inefficiency for simple types.
  - Languages such as C++ and Java treat simple built-in types differently than objects, thus obtaining efficiency but at loss of a unified type system.
- **C# enjoys the best of both worlds through a process known as *boxing*.**
- **Boxing converts a value type such as *int* or a *struct* to an object reference, and is done implicitly.**
  - **Unboxing** converts a boxed value type (stored on the heap) back to an unboxed simple value (stored on the stack).
  - Unboxing is done through a type cast.

```
int x = 5;
object o = x;         // boxing
x = (int) o;          // unboxing
```

Summary

- **In this chapter we discussed the overall type system of C#.**
- **There is a fundamental distinction between *value* types, whose storage is allocated immediately when the variable is declared, and *reference* types, where storage is allocated elsewhere and the variable is only a reference to the actual data.**
- **We looked at the various kinds of value types, including the simple types discussed in Chapter 4, *struct*, which is somewhat similar to *class* but different due to being a value type, and *enum*.**
- **We then surveyed several other important types, including string, array, interface, and delegate, which will be examined in detail later.**
- **The default value of value types is a bit pattern of all zeros, and the default value of reference types is *null*.**
- **We saw that all types in C# are rooted in a fundamental base class called *object*.**
- **In C# "everything is an object," and simple types are transparently converted to objects as needed through a process known as boxing. The inverse process, unboxing, returns an object to the simple value from which it came.**