### Types

Everything in C# is a type, so it’s important to get a handle on what these different types are and how they work within the confines of C#.

##### Three Types of Types

For the purposes of this chapter, there are three kinds of type in C#:

* Value types
* Reference types
* Type-parameter types (form part of generics and are discussed in Chapter 20)

There is also a fourth type, used only in unsafe code called pointers, which you will come across in Chapter 22.

The Difference Between Value and Reference Types

There is a fundamental difference between value and reference types that is quite easy to understand:

* **Value type variables:** These types directly contain data.
* **Reference type variables:** These types contain only a reference to data and are known as objects.

This fundamental difference leads to some very interesting possibilities. For example, with refer- ence types it’s possible for two or more variables to reference the same object, and if an operation is carried out on one variable, this affects the object referenced by all the other variables.

The situation is different with value types. With value types, the variables each have their own copy of data, and working on one copy does not affect any of the others. Thus:

* Reference types refer to a single source of data.
* Value types each have their own copy of data.

This fundamental difference has huge practical applications in programming but can also be the source of a lot of problems if you’re not aware of it.

ref and out Parameters

When a variable is either a ref or out parameter, it is important to note that the variable is in essence an alias for another variable rather than being a distinct variable itself. It doesn’t have its own storage but instead references the storage area of another variable.

##### The C# Type System

Every value of any type in C# is unified and can be treated as an object, and every type, either directly or indirectly, derives from the object class type. Also, object will be the base class for all types.

How the two types are treated as objects is also different:

* The values of reference types are handled as objects by simply viewing the value as object.
* The values of value types can only be treated as objects by carrying out boxing and unboxing operations (explained later in this chapter).

##### Value Types

Value types can be either:

* A struct type
* An enumeration type

C# offers a host of predefined struct types called simple types, and these are identified through reserved words, the syntax of which is listed as follows:

value-type:

struct-type enum-type

struct-type:

type-name simple-type nullable-type

simple-type:

numeric-type bool

numeric-type:

integral-type floating-point-type decimal

integral-type: sbyte byte short ushort int

uint long ulong char

floating-point-type: float

double

enum-type:

type-name

nullable-type:

non-nullable-value-type ?

non-nullable-value-type: enum-type

type-name simple-type

All value types will implicitly inherit from the class object, and it is not possible for types to derive a value type, which makes value types sealed.

One key aspect of a variable of the value types is that they will always, without exception, contain a value of that type. It is impossible for a value type to have a value that is null. Equally, the value of a value type cannot reference an object of a more derived type.

Assignment to any variable of a value type results in a copy of that value being assigned, keeping the original value safe from alteration. This is different from reference values, where the reference is copied but not the object itself.

System.ValueType

All value types inherit implicitly from the System.ValueType class. This class inherits from the object

class.

Bear in mind that the System.ValueType class is a class-type from which every value-type is derived rather than being a value-type itself.

Default Constructors

All value types implicitly declare a public parameterless instance constructor. This constructor is called a default constructor, and it returns a zero-initialized instance known as a default value for the type.

For all simple types, the default value will be produced by a bit pattern that corresponds to all zeros.

**Type Default value**

sbyte byte short ushort int uint long

ulong

0

Char \x0000

Float 0.0f

Double 0.0d

Decimal 0m

Bool false

For enum-types E (a shorthand notation), the default is 0.

For struct-type, the default value will be the value produced when setting all the value types to their default values and all reference fields to null.

Struct Types

A struct type is a value type that can declare any of the following:

* Constants
* Fields
* Indexers
* Instance constructors
* Methods
* Nested types
  + Operators
  + Properties
  + Static constructors

Simple Types

The predefined struct types in C# are called simple types. These are identified through the use of reserved words. These reserved words are aliases for predefined struct types contained in the System namespace.

Here is a list of reserved words, along with their aliased types:

**Reserved word Aliased type**

Ushort System.Uint16

Bool

System.Boolean

Byte

System.Byte

Char

System.Char

Decimal

System.Decimal

Double

System.Double

Float

System.Single

Int

System.Int32

Long

System.Int64

Sbyte

System.Sbyte

Short

System.Int16

Uint

System.Uint32

Ulong

System.Uint64

You can carry out more operations on simple types than is possible on other struct types:

* + Most simple types allow values to be created by writing literals.
  + When the operands of an expression are all value types (known as a constant expression), the compiler will evaluate the expression when it is compiled. This speeds program execution.
  + Constants of simple types can be declared using const declarations.

Integral Type

C# supports several different integral types, described in the following table:

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Value range** |
| Sbyte | Signed 8-bit integer | -128 |
|  |  | to |
|  |  | 127 |
| Byte | Unsigned 8-bit integer | 0 |
|  |  | to |
|  |  | 255 |
| Short | Signed 16-bit integer | -32768 |
|  |  | to |
|  |  | 32767 |
| Ushort | Unsigned 16-bit integer | 0 |
|  |  | to |
|  |  | 65535 |
| Int | Signed 32-bit integer | -2147483648 |
|  |  | to |
|  |  | 2147483647 |
| Uint | Unsigned 32-bit integer | 0 |
|  |  | to |
|  |  | 4294967295 |
| Long | Signed 64-bit integer | -9223372036854775808 |
|  |  | to |
|  |  | 9223372036854775807 |
| Ulong | Unsigned 64-bit integer | 0 |
|  |  | to |
|  |  | 18446744073709551615 |
| Char | Unsigned 16-bit integer | 0 |
|  | corresponding to the Unicode | to |
|  | character set | 65535 |

*Note that while* char *types are integral types, there are two differences:*

* *Implicit conversion to the* char *type from other types is not supported.*
* *Constants of the* char *type are written as* character-literals *or* integer-literals *and in combination with a cast to the* char *type.*

Types can also be signed (positive and negative) or unsigned:

**Type Signed?**

Sbyte

Byte Short Ushort Int Uint Long Ulong Char Float Double

Decimal

Bool

Yes

No Yes No Yes No Yes No N/A Yes Yes Yes No

Each type also occupies a specific number of bytes in memory.

|  |  |
| --- | --- |
| **Type** | **Bytes Occupied** |
| Sbyte | 1 |
| Byte | 1 |
| Short | 2 |
| Ushort | 2 |
| Int | 4 |
| Uint | 4 |
| Long | 8 |
| Ulong | 8 |
| Char | 2 |
| Float | 4 |
| Double | 8 |
| Decimal | 12 |
| Bool | 1/2 |

To reduce on the system requirements of code, use the most appropriate type for your data. For example, if a short integer will do instead of a long one, use it and save six bytes for each entry. Using decimal instead of short would mean that each variable would require 12 bytes instead of two.

The integral-type unary and binary operators always use the following levels of precision:

* signed 32-bit precision
* unsigned 32-bit precision
* signed 64-bit precision
* unsigned 64-bit precision

Using Types

Using types is easy. The type names prefix variable names. For example:

string str1 = “Hello, World!”;

string str2 = str1; //str1 equals str2 int x = 10;

int y = x; // y equals 10 y = 20; // y now equals 20

Floating-Point Types

C# supports two floating-point types:

* Float — Values ranging from approximately 1.5  1045 to 3.4  1038. Float has a precision accu- rate to 7 digits.
* Double — Values ranging from approximately 5.0  10324 to 1.7  10308. Double has a precision accurate to 15 or 16 digits.

float and double are represented using 32-bit single-precision and 64-bit double-precision formats. The following sets of values are allowed:

* **Positive and negative zero.** In most cases, these are identical to simple zero, but some opera- tions (division operations) distinguish between the two.
* **Positive and negative infinity.** Infinities are generated by dividing a nonzero number by zero.
* **Not-a-Number (NaN).** These are produced by invalid floating-point operations (carrying out a divide zero by zero, for example).

Floating-point operations do not produce exceptions. Instead, they produce one of the following in an exception situation:

* Zero
* Infinity
* NaN

Here are the rules by which these are generated:

* + The result of a floating-point operation can be rounded to the nearest value that can be repre- sented by the destination format, and this may cause a nonzero value to be rounded to zero.
  + If the magnitude of the result of a floating-point operation is too big for the destination format, the result of the operation is transformed into positive infinity or negative infinity.
  + If a floating-point operation is invalid, the result of the operation produces NaN.
  + If one or both operands of a floating-point operation are NaN, the result of the operation also becomes NaN.

Decimal Types

A decimal type is a 128-bit type. It has the range 1  10-28 to 1  1028 and has at least 28 significant digits. The decimal type is ideally suited for financial calculations.

If a decimal arithmetic operation produces a result where the magnitude is too large for the decimal for- mat, a System.OverflowException is thrown.

Again, be aware that rounding operations can cause a loss of precision or a rounding to zero.

bool Type

The bool type represents a Boolean logic quantity that can be either true or false. There is no stan- dard conversion between bool and other types, and it is distinct to integral types.

Enumeration Types

An enumeration type is a distinct type with named constants. Each enumeration type has an underlying type, which will be one of the following:

* + byte
  + sbyte
  + short
  + ushort
  + int
  + uint
  + long
  + ulong

Enumeration types are defined through enumeration declarations.

The direct base type of every enumeration type is the class System.Enum, while the direct base class of

System.Enum is System.ValueType.

##### Reference Types

A reference type is one of the following types:

* class type
* interface type
* array type
* delegate type

A reference type value is a reference to an instance of that type, known as an object. Null values are allowed for reference types and mean that there is no instance of the type.

reference-type: class-type interface-type array-type delegate-type

class-type:

type-name object string

interface-type: type-name

array-type:

non-array-type rank-specifiers

non-array-type: value-type class-type interface-type delegate-type type-parameter

rank-specifiers: rank-specifier

rank-specifiers rank-specifier

rank-specifier:

[ dim-separatorsopt ]

dim-separators:

,

dim-separators ,

delegate-type: type-name

Class Types

A class type is a data structure that contains the following:

* + **Data members.** These include constants and fields.
  + **Function members.** These include events, methods, properties, instance constructors, indexers, operators, finalizers, and static constructors.
  + Nested types.

*Note that class types do support inheritance.*

Object Type

The object class type is, ultimately, the base class of all other types and, every other type directly or indirectly derives from the object class type.

The object keyword is an alias for the System.Object class.

String Type

The string type is a sealed class that inherits directly from object. Instances of the string class repre- sent Unicode character strings and values of the string type can be written as string literals.

The string keyword is an alias for the System.String class.

Array Types

An array is a data structure. An array can contain zero or more variables that are accessed through indices. The variables contained in an array (also called the elements) must all be of the same type, called the element type of the array.

Delegate Types

A delegate is a data structure that refers to one or more methods. For instance, a delegate also refers to the corresponding object instances.

The null Type

The null literal evaluates to the null value, which is used to indicate a reference that doesn’t point to an object or array. It can also indicate the absence of a value.

The null type has a single value, which is the null value. This means that any expression that has a

null type can evaluate only to the null value.

Boxing and Unboxing

Boxing and unboxing are key components of C# types. They act as a pathway between value and refer- ence types by allowing value types to be converted to and from type object.

Boxing

A boxing conversion allows the programmer to implicitly convert any value type to object or System.ValueType or to any interface type implemented by the value type. There also exists an implicit boxing conversion from any enumeration type to System.Enum.

Boxing a value of a value type consists of allocating an object instance and copying the value type value into that instance.

Unboxing

An unboxing conversion allows the programmer to carry out an explicit conversion from object or System.ValueType to any value type, or from any interface type to any value type that implements the interface type. There is an explicit unboxing conversion from System.Enum to any enumeration type.

An unboxing operation consists of checking that the object instance is a boxed value of the given value type and then copying (not referring to) the value out of the instance.

Nullable Types

A nullable type is classed as a value type.

The type specified before the ? modifier in a nullable type is called the underlying type of the nullable type.

The underlying type of a nullable type can be any non-nullable value type or any type parameter limited to non-nullable value types.

The underlying type of a nullable type shall not be a nullable type or a reference type.

Members

An instance of a nullable type T? has two public properties that are read-only. These are:

* HasValue — The type of this property is bool.
* Value — The property is of type T.

For any instance where HasValue is true, it is said to be non-null. This instance will contain a value that will be returned by Value.

If HasValue is false, the instance is said to be null. Trying to read Value will cause a

System.InvalidOperationException to be thrown.

Every nullable type T? has a public constructor. This takes a single argument of type T. Given a value x of type T, the constructor invocation below creates a non-null instance of T? where the Value property is x.

new T? (x)

Implemented Interfaces

A type of the form T? implements the same interfaces as System.Nullable<T>.

This normally means that the interfaces implemented by T and T? are going to be different.

##### Summary

In this chapter you looked at a theme that is key to C# programming — types. This chapter has revolved around the fundamental difference between value types (where each variable has an independent copy of the data) and reference types (which refer to the same data).

In Chapter 7, you look at variables.