9

Value Types

You have used value types throughout this book; for example, int is a value type. This chapter discusses not only using value types but also defining custom value types. There are two categories of custom value types: structs and enums. This chapter discusses how structs enable programmers to define new value types that behave very similarly to most of the predefined types discussed in Chapter 2. The key is that all newly defined value types have their own custom data and methods. The chapter also discusses how to use enums to define sets of constant values.

\*\*\*COMP: Insert 09mindmap pick up from previous edition 9781509303588 p. 379

|  |
| --- |
| Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_Mindmaps:09mindmap.pdf |

|  |  |
| --- | --- |
|  | |
|  | Beginner Topic  Categories of Types  All types discussed so far have fallen into one of two categories: reference types and value types. The differences between the types in each category stem from differences in copying strategies, which in turn result in each type being stored differently in memory. As a review, this Beginner Topic reintroduces the value type/reference type discussion for those readers who are unfamiliar with these issues.  Value Types  Variables of value types directly contain their values, as shown in Figure 9.1. The variable name is associated directly with the storage location in memory where the value is stored. Because of this, when a second variable is assigned the value of an original variable, a copy of the original variable’s value is made to the storage location associated with the second variable. Two variables never refer to the same storage location (unless one or both are out or ref parameters, which are, by definition, aliases for another variable). Changing the value of the original variable will not affect the value in the second variable, because each variable is associated with a different storage location. Consequently, changing the value of one value type variable cannot affect the value of any other value type variable.  \*\*\*COMP: pick up figure 9.1 from previous edition 9781509303588 p. 380   |  | | --- | | Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_NumberedFigures:09fig01.pdf | | Figure 9.1: Value Types Contain the Data Directly |   A value type variable is like a piece of paper that has a number written on it. If you want to change the number, you can erase it and replace it with a different number. If you have a second piece of paper, you can copy the number from the first piece of paper, but the two pieces of paper are then independent; erasing and replacing the number on one of them does not change the other.  Similarly, passing an instance of a value type to a method such as Console.WriteLine() will also result in a memory copy from the storage location associated with the argument to the storage location associated with the parameter, and any changes to the parameter variable inside the method will not affect the original value within the caller. Since value types require a memory copy, they generally should be defined to consume a small amount of memory (typically 16 bytes or less).   |  | | --- | | Guidelines  AVOID create value types that consume more than 16 bytes of memory. |   Values of value types are often short-lived; in many situations, a value is needed only for a portion of an expression or for the activation of a method. In these cases, variables and temporary values of value types can often be stored on the temporary storage pool, called the stack. (This term is actually a misnomer: There is no requirement that the temporary pool allocates its storage off the stack. In fact, as an implementation detail, it frequently chooses to allocate storage out of available registers instead.)  The temporary pool is less costly to clean up than the garbage-collected heap; however, value types tend to be copied more than reference types, and that copying can impose a performance cost of its own. Do not fall into the trap of believing that value types are faster because they can be allocated on the stack.  Reference Types  In contrast, the value of a reference type variable is a reference to an instance of an object (see Figure 9.2). Variables of reference type store the reference (typically implemented as the memory address) where the data for the object instance is located instead of storing the data directly, as a variable of value type does. Therefore, to access the data, the runtime reads the reference out of the variable and then dereferences it to reach the location in memory that actually contains the data for the instance.  \*\*\*COMP: pick up figure 9.3 from previous edition 9781509303588 p. 382   |  | | --- | | Macintosh HD:Users:annapopick:Desktop:Freelance:Pearson Freelance:Pearson_InProgress:9781509303588_Michaelis:Michaelis_Author:Michaelis_Word_AllEdits:Michaelis_Art:Michaelis_NumberedFigures:09fig02.pdf | | Figure 9.2: Reference Types Point to the Heap |   A reference type variable, therefore, has two storage locations associated with it: the storage location directly associated with the variable and the storage location referred to by the reference that is the value stored in the variable.  A reference type variable is, again, like a piece of paper that always has something written on it. Imagine a piece of paper that has a house address written on it—for example, “123 Sesame Street, New York City.” The piece of paper is a variable; the address is a reference to a building. Neither the paper nor the address written on it is the building, and the location of the paper need not have anything whatsoever to do with the location of the building to which its contents refer. If you make a copy of that reference on another piece of paper, the contents of both pieces of paper refer to the same building. If you then paint that building green, the building referred to by both pieces of paper can be observed to be green, because the references refer to the same thing.  The storage location directly associated with the variable (or temporary value) is treated no differently than the storage location associated with a value type variable: If the variable is known to be short-lived, it is allocated on the short-term storage pool. The value of a reference type variable is always either a reference to a storage location in the garbage-collected heap or null.  Compared to a variable of value type, which stores the data of the instance directly, accessing the data associated with a reference involves an extra hop: First the reference must be dereferenced to find the storage location of the actual data, and then the data can be read or written. Copying a reference type value copies only the reference, which is small. (A reference is guaranteed to be no larger than the bit size of the processor; a 32-bit machine has 4-byte references, a 64-bit machine has 8-byte references, and so on.) Copying the value of a value type copies all the data, which could be large. Therefore, in some circumstances, reference types are more efficient to copy. This is why the guideline for value types is to ensure that they are never more than 16 bytes or thereabouts; if a value type is more than four times as expensive to copy as a reference, it probably should simply be a reference type.  Since reference types copy only a reference to the data, two different variables can refer to the same data. In such a case, changing the data through one variable will be observed to change the data for the other variable as well. This happens both for assignments and for method calls.  To continue our previous analogy, if you pass the address of a building to a method, you make a copy of the paper containing the reference and hand the copy to the method. The method cannot change the contents of the original paper to refer to a different building. If the method paints the referred-to building, however, when the method returns, the caller can observe that the building to which the caller is still referring is now a different color. |
|  | |

Structs

All the C# built-in types, such as bool and int, are value types, except for string and object, which are reference types. Numerous additional value types are provided within the framework. It is also possible for developers to define their own value types.

To define a custom value type, you use a syntax similar to the syntax you would use to define class and interface types. The key difference in the syntax is that value types use the keyword struct, as shown in Listing 9.1. Here we have a value type that describes a high-precision angle in terms of its degrees, minutes, and seconds. (A minute is one-sixtieth of a degree, and a second is one-sixtieth of a minute. This system is used in navigation because it has the nice property that an arc of one minute over the surface of the ocean at the equator is exactly one nautical mile.)

\*\*\*COMP: Insert “Begin 6.0” tab

Listing 9.1: Declaring a struct

// Use keyword struct to declare a value type

struct Angle

{

public Angle(int degrees, int minutes, int seconds)

{

Degrees = degrees;

Minutes = minutes;

Seconds = seconds;

}

// Using C# 6.0 read-only, automatically implemented properties

public int Degrees { get; }

public int Minutes { get; }

public int Seconds { get; }

public Angle Move(int degrees, int minutes, int seconds)

{

return new Angle(

Degrees + degrees,

Minutes + minutes,

Seconds + seconds);

}

}

// Declaring a class as a reference type

// (declaring it as a struct would create a value type

// larger than 16 bytes)

class Coordinate

{

public Angle Longitude { get; set; }

public Angle Latitude { get; set; }

}

This listing defines Angle as a value type that stores the degrees, minutes, and seconds of an angle, either longitude or latitude. The resultant C# type is a struct.

Note that the Angle struct in Listing 9.1 is immutable because all properties are declared using C# 6.0’s read-only, automatically implemented property capability. To create a read-only property without C# 6.0, programmers will need to declare a property with only a getter that accesses its data from a readonly modified field (see Listing 9.3). C# 6.0 provides a noticeable code reduction when it comes to defining immutable types.

\*\*\*COMP: Insert “Begin 7.2” tab

Starting with C# 7.2, you can verify that, in fact, you have successfully defined a struct that is read only by declaring it as such:

readonly struct Angle {}

Now the compiler will verify that the entire struct is immutable, reporting an error if there is a field that is not read only or a property that has a setter.

\*\*\*COMP: Insert “Begin 8.0” tab

If, however, you need finer grained control than declaring the entire class as readonly, C# 8.0 allows you to define any struct member as readonly (including methods and even getters - which potentially modify an object's state even though they shouldn't). For example, in Listing 9.1, the Move() method can include a readonly modifier:

readonly public Angle Move(int degrees, int minutes, int seconds) { ... }

(Although, doing so is allowable but redundant when the class is readonly.) Regardless, readonly members that modify a struct’s data (properties or fields) or invoke a non-readonly member, will report a compile time error. By supporting the concept of readonly on members, developers declare the behavioral intent of whether a member can modify the object instance. Note that properties that are not automatically implemented can use the readonly modifier on either the getter or the setter (albeit the latter would be strange). To decorate both, the readonly modifier would be placed on the property itself, rather than the getter and setters individually.

\*\*\*COMP: Insert “End 8.0” tab

|  |
| --- |
| note  A good guideline is for value types to be immutable: Once you have instantiated a value type, you should not be able to modify the same instance. In scenarios where modification is desirable, you should create a new instance. Listing 9.1 supplies a Move() method that doesn’t modify the instance of Angle, but instead returns an entirely new instance.  There are two good reasons for this guideline. First, value types should represent values. One does not think of adding two integers together as mutating either of them; rather, the two addends are immutable and a third value is produced as the result.  Second, because value types are copied by value, not by reference, it is very easy to get confused and incorrectly believe that a mutation to one value type variable can be observed to cause a mutation in another, as it would with a reference type. |

|  |
| --- |
| Guidelines  DO make value types immutable. |

Initializing Structs

In addition to properties and fields, structs may contain methods and constructors. However, no default constructor may be defined. Instead, the C# compiler automatically generates a default constructor that initializes all fields to their default values. The default value is null for a field of reference type data, a zero value for a field of numeric type, false for a field of Boolean type, and so on.

To ensure that a local value type variable can be fully initialized by a constructor, every constructor in a struct must initialize all fields (and read-only, automatically implemented properties) within the struct. (In C# 6.0, initialization via a read-only, automatically implemented property is sufficient because the backing field is unknown and its initialization would not be possible.) Failure to initialize all data within the struct causes a compile-time error. To complicate matters slightly, C# disallows field initializers in a struct. Listing 9.2, for example, would not compile if the line \_Degrees = 42 was uncommented.

Listing 9.2: Initializing a struct Field within a Declaration, Resulting in an Error

struct Angle

{

// ...

// ERROR: Fields cannot be initialized at declaration time

// int \_Degrees = 42;

// ...

}

If not explicitly instantiated via the new operator’s call to the constructor, all data contained within the struct is implicitly initialized to that data’s default value. However, all data within a value type must be explicitly initialized to avoid a compiler error. This raises a question: When might a value type be implicitly initialized but not explicitly instantiated? This situation occurs when instantiating a reference type that contains an unassigned field of value type as well as when instantiating an array of value types without an array initializer.

To fulfill the initialization requirement on a struct, all explicitly declared fields must be initialized. Such initialization must be done directly. For example, in Listing 9.3, the constructor that initializes the property (if uncommented out) rather than the field produces a compile error.

Listing 9.3: Accessing Properties before Initializing All Fields

struct Angle

{

// ERROR: The "this" object cannot be used before

// all of its fields are assigned to

// public Angle(int degrees, int minutes, int seconds)

// {

// Degrees = degrees; // Shorthand for this.Degrees = ...;

// Minutes = minutes;

// Seconds = seconds;

// }

public Angle(int degrees, int minutes, int seconds)

{

\_Degrees = degrees;

\_Minutes = minutes;

\_Seconds = seconds;

}

public int Degrees { get { return \_Degrees; } }

readonly private int \_Degrees;

public int Minutes { get { return \_Minutes; } }

readonly private int \_Minutes;

public int Seconds { get { return \_Seconds; } }

readonly private int \_Seconds;

// ...

}

It is not legal to access this until the compiler knows that all fields have been initialized; such as accessing the Degrees property is implicitly this.Degrees. To resolve this issue, you need to initialize the fields directly, as demonstrated in the constructor of Listing 9.3 that is not commented out.

Because of the struct’s field initialization requirement, the succinctness of C# 6.0’s read-only, automatically implemented property support, and the guideline to avoid accessing fields from outside of their wrapping property, you should favor read-only, automatically implemented properties over fields within structs starting with C# 6.0.

|  |
| --- |
| Guidelines  DO ensure that the default value of a struct is valid; encapsulation cannot prevent obtaining the default “all zero” value of a struct. |

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Using new with Value Types  Invoking the new operator with a reference type causes the runtime to create a new instance of the object on the garbage-collected heap, initialize all of its fields to their default values, and call the constructor, passing a reference to the instance as this. The result is the reference to the instance, which can then be copied to its final destination. In contrast, invoking the new operator with a value type causes the runtime to create a new instance of the object on the temporary storage pool, initialize all of its fields to their default values, and call the constructor (passing the temporary storage location as a ref variable as this), resulting in the value being stored in the temporary storage location, which can then be copied to its final destination.  Unlike classes, structs do not support finalizers. Structs are copied by value; they do not have referential identity, as reference types do. Therefore, it is hard to know when it would be safe to execute the finalizer and free an unmanaged resource owned by the struct. The garbage collector knows when there are no “live” references to an instance of reference type and can choose to run the finalizer for an instance of reference type at any time after there are no more live references. Nevertheless, no part of the runtime tracks how many copies of a given value type exist at any moment.   |  | | --- | | Language Contrast: C++—struct Defines Type with Public Members  In C++, the difference between a type declared with struct and one declared with class is whether the default accessibility is public or private. The contrast is far greater in C#, where the difference is whether instances of the type are copied by value or by reference. | |
|  | |

\*\*\*COMP: End Advanced Topic after Language Contrast

Inheritance and Interfaces with Value Types

All value types are implicitly sealed. In addition, all non-enum value types derive from System.ValueType. As a consequence, the inheritance chain for structs is always from object to System.ValueType to the struct.

Value types can implement interfaces, too. Many of those built into the framework implement interfaces such as IComparable and IFormattable.

System.ValueType brings with it the behavior of value types, but it does not include any additional members. The System.ValueType customizations focus on overriding all of object’s virtual members. The rules for overriding these methods in a struct are almost the same as those for classes (see Chapter 10). However, one difference is that with value types, the default implementation for GetHashCode() is to forward the call to the first non-null field within the struct. Also, Equals() makes significant use of reflection. Therefore, if a value type is used frequently inside collections, especially dictionary-type collections that use hash codes, the value type should include overrides for both Equals() and GetHashCode() to ensure good performance. See Chapter 10 for more details.

|  |
| --- |
| Guidelines  DO override the equality operators (Equals(), ==, and !=) and GetHashCode() on value types if equality is meaningful. (Also consider implementing the IEquatable<T> interface.) |

Boxing

We know that variables of value type directly contain their data, whereas variables of reference type contain a reference to another storage location. But what happens when a value type is converted to one of its implemented interfaces or to its root base class, object? The result of the conversion has to be a reference to a storage location that contains something that looks like an instance of a reference type, but the variable contains a value of value type. Such a conversion, which is known as boxing, has special behavior. Converting a variable of value type that directly refers to its data to a reference type that refers to a location on the garbage-collected heap involves several steps.

1. Memory is allocated on the heap that will contain the value type’s data and the other overhead necessary to make the object look like every other instance of a managed object of reference type (namely, a SyncBlockIndex and method table pointer).

2. The value of the value type is copied from its current storage location into the newly allocated location on the heap.

3. The result of the conversion is a reference to the new storage location on the heap.

The reverse operation is unboxing. The unboxing conversion checks whether the type of the boxed value is compatible with the type to which the value is being unboxed, and then results in a copy of the value stored in the heap location.

Boxing and unboxing are important to consider because boxing has some performance and behavioral implications. Besides learning how to recognize these conversions within C# code, a developer can count the box/unbox instructions in a particular snippet of code by looking through the Common Intermediate Language (CIL). Each operation has specific instructions, as shown in Table 9.1.

|  |  |
| --- | --- |
| Table 9.1: Boxing Code in CIL | |
| C# Code | CIL Code |
| static void Main()  {  int number;  object thing;  number = 42;  // Boxing  thing = number;  // Unboxing  number = (int)thing;  return;  } | .method private hidebysig  static void Main() cil managed  {  .entrypoint  // Code size 21 (0x15)  .maxstack 1  .locals init ([0] int32 number,  [1] object thing)  IL\_0000: nop  IL\_0001: ldc.i4.s 42  IL\_0003: stloc.0  IL\_0004: ldloc.0  IL\_0005: box [mscorlib]System.Int32  IL\_000a: stloc.1  IL\_000b: ldloc.1  IL\_000c: unbox.any [mscorlib]System.Int32  IL\_0011: stloc.0  IL\_0012: br.s IL\_0014  IL\_0014: ret  } // end of method Program::Main |

When boxing and unboxing occur infrequently, their implications for performance are irrelevant. However, boxing can occur in some unexpected situations, and frequent occurrences can have a significant impact on performance. Consider Listing 9.4 and Output 9.1. The ArrayList type maintains a list of references to objects, so adding an integer or floating-point number to the list will box the value so that a reference can be obtained.

Listing 9.4: Subtle Box and Unbox Instructions

class DisplayFibonacci

{

static void Main()

{

int totalCount;

// Intentionally using ArrayList to demonstrate boxing

System.Collections.ArrayList list =

new System.Collections.ArrayList();

Console.Write("Enter a number between 2 and 1000:");

totalCount = int.Parse(Console.ReadLine());

if (totalCount == 7) // Magic number used for testing

{

// Triggers exception when retriving value as double.

// list.Add(0); // Cast to double or 'D' suffix required.

// Whether cast or using 'D' suffix,

// CIL is identical.

}

else

{

list.Add((double)0);

}

list.Add((double)0);

list.Add((double)1);

for (int count = 2; count < totalCount; count++)

{

list.Add(

(double)list[count - 1]! +

(double)list[count – 2]! );

}

// Using a foreach to clarify the box/unbox operations rather than

// Console.WriteLine(string.Join(", ", list.ToArray()));

foreach (double count in list)

{

Console.Write("{0}, ", count);

}

}

}

|  |
| --- |
| Output 5.2 |
| Enter a number between 2 and 1000: 42  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155, 165580141, |

The code shown in Listing 9.4, when compiled, produces five box and three unbox instructions in the resultant CIL.

1. The first two box instructions occur in the initial calls to list.Add(). The signature for the ArrayList[[1]](#footnote-2) method is int Add(object value). As such, any value type passed to this method is boxed.

2. Next are two unbox instructions in the call to Add() within the for loop. The return from an ArrayList’s index operator is always object because that is what ArrayList contains. To add the two values, you need to cast them back to doubles. This cast from a reference to an object to a value type is implemented as an unbox call.

3. Now you take the result of the addition and place it into the ArrayList instance, which again results in a box operation. Note that the first two unbox instructions and this box instruction occur within a loop.

4. In the foreach loop, you iterate through each item in ArrayList and assign the items to count. As you saw earlier, the items within ArrayList are references to objects, so assigning them to a double? is, in effect, unboxing each of them.

5. The signature for Console.WriteLine(), which is called within the foreach loop, is void Console.Write(string format, object arg). As a result, each call to it boxes the double to object.

Every boxing operation involves both an allocation and a copy; every unboxing operation involves a type check and a copy. Doing the equivalent work using the unboxed type would eliminate the allocation and type check. Obviously, you can easily improve this code’s performance by eliminating many of the boxing operations. Using an object rather than double in the last foreach loop is one such improvement. Another would be to change the ArrayList data type to a generic collection (see Chapter 12). The point being made here is that boxing can be rather subtle, so developers need to pay special attention and notice situations where it could potentially occur repeatedly and affect performance.

Another unfortunate boxing-related problem also occurs at runtime: When calling Add() without first casting to double, (or use a double literal), you could insert integers into the array list. Since ints will implicitly be converted to doubles, this would appear to be an innocuous modification. However, the casts to double when retrieving the value from within the foreach loop would fail. The problem is that immediately following the unbox operation is an attempt to perform a memory copy of the value of the boxed int into a double. You cannot do this without first casting to an int, because the code will throw an InvalidCastException at execution time. Listing 9.5 shows a similar error commented out and followed by the correct cast.

Listing 9.5: Unboxing Must Be to the Underlying Type

// ...

int number;

object thing;

double bigNumber;

number = 42;

thing = number;

// ERROR: InvalidCastException

// bigNumber = (double)thing;

bigNumber = (double)(int)thing;

// ...

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  Value Types in the lock Statement  C# supports a lock statement for synchronizing code. The statement compiles down to System.Threading.Monitor’s Enter() and Exit() methods. These two methods must be called in pairs. Enter() records the unique reference argument passed so that when Exit() is called with the same reference, the lock can be released. The trouble with using value types is the boxing. Therefore, each time Enter() or Exit() is called, a new value is created on the heap. Comparing the reference of one copy to the reference of a different copy will always return false, so you cannot hook up Enter() with the corresponding Exit(). Therefore, value types in the lock() statement are not allowed.  Listing 9.6 points out a few more runtime boxing idiosyncrasies, and Output 9.2 shows the results.  Listing 9.6: Subtle Boxing Idiosyncrasies  interface IAngle  {  void MoveTo(int degrees, int minutes, int seconds);  }  struct Angle : IAngle  {  // ...  // NOTE: This makes Angle mutable, against the general  // guideline  public void MoveTo(int degrees, int minutes, int seconds)  {  \_Degrees = degrees;  \_Minutes = minutes;  \_Seconds = seconds;  }  }  class Program  {  static void Main()  {  // ...  Angle angle = new Angle(25, 58, 23);  // Example 1: Simple box operation  object objectAngle = angle; // Box  Console.Write( ((Angle)objectAngle).Degrees);  // Example 2: Unbox, modify unboxed value, and discard value  ((Angle)objectAngle).MoveTo(26, 58, 23);  Console.Write(", " + ((Angle)objectAngle).Degrees);  // Example 3: Box, modify boxed value, and discard reference to box  ((IAngle)angle).MoveTo(26, 58, 23);  Console.Write(", " + ((Angle)angle).Degrees);  // Example 4: Modify boxed value directly  ((IAngle)objectAngle).MoveTo(26, 58, 23);  Console.WriteLine(", " + ((Angle)objectAngle).Degrees);  // ...  }  }   |  | | --- | | Output 9.2 | | 25, 25, 25, 26 |   Listing 9.6 uses the Angle struct and IAngle interface. Note also that the IAngle.MoveTo() interface changes Angle to be mutable. This change brings out some of the idiosyncrasies of mutable value types and, in so doing, demonstrates the importance of the guideline to make structs immutable.  In Example 1 of Listing 9.6, after you initialize angle, you then box it into a variable called objectAngle. Next, Example 2 calls MoveTo() to change \_Degrees to 26. However, as the output demonstrates, no change actually occurs the first time. The problem is that to call MoveTo(), the compiler unboxes objectAngle and (by definition) makes a copy of the value. Value types are copied by value—that is why they are called value types. Although the resultant value is successfully modified at execution time, this copy of the value is discarded and no change occurs on the heap location referenced by objectAngle.  Recall our analogy that suggested variables of value type are like pieces of paper with the value written on them. When you box a value, you make a photocopy of the paper and put the copy in a box. When you unbox the value, you make a photocopy of the paper in the box. Making an edit to this second copy does not change the copy that is in the box.  In Example 3, a similar problem occurs in reverse. Instead of calling MoveTo() directly, the value is cast to IAngle. The conversion to an interface type boxes the value, so the runtime copies the data in angle to the heap and provides a reference to that box. Next, the method call modifies the value in the referenced box. The value stored in variable angle remains unmodified.  In the last case, the cast to IAngle is a reference conversion, not a boxing conversion. The value has already been boxed by the conversion to object in this case, so no copy of the value occurs on this conversion. The call to MoveTo() updates the \_Degrees value stored in the box, and the code behaves as desired.  As you can see from this example, mutable value types are quite confusing because it is often unclear when you are mutating a copy of the value rather than the storage location you actually intend to change. By avoiding mutable value types in the first place, you can eliminate this sort of confusion.   |  | | --- | | Guidelines  AVOID mutable value types. | |
|  | |

\*\*\*COMP: End Advanced Topic after Guidelines

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  How Boxing Can Be Avoided during Method Calls  Anytime a method is called on a value type, the value type receiving the call (represented by this in the body of the method) must be a variable, not a value, because the method might be trying to mutate the receiver. Clearly, it must be mutating the receiver’s storage location, rather than mutating a copy of the receiver’s value and then discarding it. Examples 2 and 4 of Listing 9.6 illustrate how this fact affects the performance of a method invocation on a boxed value type.  In Example 2, the unboxing conversion logically produces the boxed value, not a reference to the storage location on the heap that contains the boxed copy. Which storage location, then, is passed as the this to the mutating method call? It cannot be the storage location from the box on the heap, because the unboxing conversion produces a copy of that value, not a reference to that storage location.  When this situation arises—a variable of value type is required but only a value is available—one of two things happens: Either the C# compiler generates code that makes a new, temporary storage location and copies the value from the box into the new location, resulting in the temporary storage location becoming the needed variable, or the compiler produces an error and disallows the operation. In this case, the former strategy is used. The new temporary storage location is then the receiver of the call; after it is mutated, the temporary storage location is discarded.  This process—performing a type check of the boxed value, unboxing to produce the storage location of the boxed value, allocating a temporary variable, copying the value from the box to the temporary variable, and then calling the method with the location of the temporary storage—happens every time you use the unbox-and-then-call pattern, regardless of whether the method actually mutates the variable. Clearly, if it does not mutate the variable, some of this work could be avoided. Because the C# compiler does not know whether any particular method you call will try to mutate the receiver, it must err on the side of caution.  These expenses are all eliminated when calling an interface method on a boxed value type. In such a case, the expectation is that the receiver will be the storage location in the box; if the interface method mutates the storage location, it is the boxed location that should be mutated. Therefore, the expense of performing a type check, allocating new temporary storage, and making a copy is avoided. Instead, the runtime simply uses the storage location in the box as the receiver of the call to the struct’s method.  In Listing 9.7, we call the two-argument version of ToString() that is found on the IFormattable interface, which is implemented by the int value type. In this example, the receiver of the call is a boxed value type, but it is not unboxed to make the call to the interface method.  Listing 9.7: Avoiding Unboxing and Copying  int number;  object thing;  number = 42;  // Boxing  thing = number;  // No unboxing conversion  string text = ((IFormattable)thing).ToString(  "X", null);  Console.WriteLine(text);  You might now wonder: Suppose that we had instead called the virtual ToString() method declared by object with an instance of a value type as the receiver. What happens then? Is the instance boxed, unboxed, or what? A number of different scenarios are possible depending on the details:  If the receiver is unboxed and the struct overrides ToString(), the overridden method is called directly. There is no need for a virtual call because the method cannot be overridden further by a more derived class; all value types are automatically sealed.  If the receiver is unboxed and the struct does not override ToString(), the base class implementation must be called, and it expects a reference to an object as its receiver. Therefore, the receiver is boxed.  If the receiver is boxed and the struct overrides ToString(), the storage location in the box is passed to the overriding method without unboxing it.  If the receiver is boxed and the struct does not override ToString(), the reference to the box is passed to the base class’s implementation of the method, which is expecting a reference. |
|  | |

Enums

Compare the two code snippets shown in Listing 9.8.

Listing 9.8: Comparing an Integer Switch to an Enum Switch

int connectionState;

// ...

switch (connectionState)

{

case 0:

// ...

break;

case 1:

// ...

break;

case 2:

// ...

break;

case 3:

// ...

break;

}

ConnectionState connectionState;

// ...

switch (connectionState)

{

case ConnectionState.Connected:

// ...

break;

case ConnectionState.Connecting:

// ...

break;

case ConnectionState.Disconnected:

// ...

break;

case ConnectionState.Disconnecting:

// ...

break;

}

Obviously, the difference in terms of readability is tremendous—in the second snippet, the cases are self-documenting. However, the performance at runtime is identical. To achieve this outcome, the second snippet uses enum values in each case.

An enum is a value type that the developer can declare. The key characteristic of an enum is that it declares at compile time a set of possible constant values that can be referred to by name, thereby making the code easier to read. The syntax for a typical enum declaration is shown in Listing 9.9.

Listing 9.9: Defining an Enum

enum ConnectionState

{

Disconnected,

Connecting,

Connected,

Disconnecting

}

|  |
| --- |
| note  An enum can be used as a more readable replacement for Boolean values as well. For example, a method call such as SetState(true) is less readable than SetState(DeviceState.On). |

You use an enum value by prefixing it with the enum's name. To use the Connected value, for example, you would use the syntax ConnectionState.Connected. Do not make the enum type name a part of the value’s name so as to avoid the redundancy of something such as ConnectionState.ConnectionStateConnected. By convention, the enum name itself should be singular (unless the enums are bit flags, as discussed shortly). That is, the nomenclature should be ConnectionState, not ConnectionStates.

Enum values are actually implemented as nothing more than integer constants. By default, the first enum value is given the value 0, and each subsequent entry increases by 1. However, you can assign explicit values to enums, as shown in Listing 9.10.

Listing 9.10: Defining an Enum Type

enum ConnectionState : short

{

Disconnected,

Connecting = 10,

Connected,

Joined = Connected,

Disconnecting

}

In this code, Disconnected has a default value of 0 and Connecting has been explicitly assigned 10; consequently, Connected will be assigned 11. Joined is assigned 11, the value assigned to Connected. (In this case, you do not need to prefix Connected with the enum name, since it appears within its scope.) Disconnecting is 12.

An enum always has an underlying type, which may be any integral type other than char. In fact, the enum type’s performance is identical to that of the underlying type. By default, the underlying value type is int, but you can specify a different type using inheritance type syntax. Instead of int, for example, Listing 9.10 uses a short. For consistency, the syntax for enums emulates the syntax of inheritance, but this doesn’t actually make an inheritance relationship. The base class for all enums is System.Enum, which in turn is derived from System.ValueType. Furthermore, these classes are sealed; you can’t derive from an existing enum type to add additional members.

|  |
| --- |
| Guidelines  CONSIDER using the default 32-bit integer type as the underlying type of an enum. Use a smaller type only if you must do so for interoperability; use a larger type only if you are creating a flags enum[[2]](#footnote-3) with more than 32 flags. |

An enum is really nothing more than a set of names thinly layered on top of the underlying type; there is no mechanism that restricts the value of a variable of enumerated type to just the values named in the declaration. For example, because it is possible to cast the integer 42 to short, it is also possible to cast the integer 42 to the ConnectionState type, even though there is no corresponding ConnectionState enum value. If the value can be converted to the underlying type, the conversion to the enum type will also be successful.

The advantage of this odd feature is that enums can have new values added in later API releases, without breaking earlier versions. Additionally, the enum values provide names for the known values while still allowing unknown values to be assigned at runtime. The burden is that developers must code defensively for the possibility of unnamed values. It would be unwise, for example, to replace case ConnectionState.Disconnecting with default and expect that the only possible value for the default case was ConnectionState.Disconnecting. Instead, you should handle the Disconnecting case explicitly, and the default case should report an error or behave innocuously. As indicated earlier, however, conversion between the enum and the underlying type, and vice versa, requires an explicit cast; it is not an implicit conversion. For example, code cannot call ReportState(10) if the method’s signature is void ReportState(ConnectionState state). The only exception occurs when passing 0, because there is an implicit conversion from 0 to any enum.

Although you can add more values to an enum in a later version of your code, you should do so with care. Inserting an enum value in the middle of an enum will bump the values of all later enum values (adding Flooded or Locked before Connected will change the Connected value, for example). This will affect the versions of all code that is recompiled against the new version. However, any code compiled against the old version will continue to use the old values, making the intended values entirely different. Besides inserting an enum value at the end of the list, one way to avoid changing enum values is to assign values explicitly.

Guidelines

|  |
| --- |
| Guidelines  CONSIDER adding new members to existing enums, but keep in mind the compatibility risk.  AVOID creating enums that represent an “incomplete” set of values, such as product version numbers.  AVOID creating “reserved for future use” values in an enum.  AVOID enums that contain a single value.  DO provide a value of 0 (none) for simple enums, knowing that 0 will be the default value when no explicit initialization is provided. |

Enums are slightly different from other value types because they derive from System.Enum before deriving from System.ValueType.

Type Compatibility between Enums

C# also does not support a direct cast between arrays of two different enums. However, the CLR does, provided that both enums share the same underlying type. To work around this restriction of C#, the trick is to cast first to System.Array, as shown at the end of Listing 9.11.

Listing 9.11: Casting between Arrays of Enums

enum ConnectionState1

{

Disconnected,

Connecting,

Connected,

Disconnecting

}

enum ConnectionState2

{

Disconnected,

Connecting,

Connected,

Disconnecting

}

class Program

{

static void Main()

{

ConnectionState1[] states =

(ConnectionState1[])(Array)new ConnectionState2[42];

}

}

This example exploits the fact that the CLR’s notion of assignment compatibility is more lenient than C#’s concept. (The same trick is possible for other illegal conversions, such as int[] to uint[].) However, use this approach cautiously because there is no C# specification requiring that this behavior work across different CLR implementations.

Converting between Enums and Strings

One of the conveniences associated with enums is that the ToString() method, which is called by methods such as System.Console.WriteLine(), writes out the enum value identifier:

System.Diagnostics.Trace.WriteLine(

$"The connection is currently { ConnectionState.Disconnecting }");

The preceding code will write the text in Output 9.3 to the trace buffer.

|  |
| --- |
| Output 9.3 |
| The connection is currently Disconnecting. |

Conversion from a string to an enum is a little more difficult to achieve, because it involves a static method on the System.Enum base class. Listing 9.12 provides an example of how to do it without generics (see Chapter 12), and Output 9.4 shows the results.

Listing 9.12: Converting a String to an Enum Using Enum.Parse()

ThreadPriorityLevel priority = (ThreadPriorityLevel)Enum.Parse(

typeof(ThreadPriorityLevel), "Idle");

Console.WriteLine(priority);

|  |
| --- |
| Output 9.4 |
| Idle |

In this code, the first parameter to Enum.Parse() is the type, which you specify using the keyword typeof(). This example depicts a compile-time way of identifying the type, like a literal for the type value (see Chapter 18).

Until Microsoft .NET Framework 4, there was no TryParse() method, so code written to target prior versions needs to include appropriate exception handling if there is a chance the string will not correspond to an enum value identifier. Microsoft .NET Framework 4’s TryParse<T>() method uses generics, but the type parameters can be inferred, resulting in the to-enum conversion behavior shown in Listing 9.13.

Listing 9.13: Converting a String to an Enum Using Enum.TryParse<T>()

System.Diagnostics.ThreadPriorityLevel priority;

if(Enum.TryParse("Idle", out priority))

{

Console.WriteLine(priority);

}

This technique eliminates the need to use exception handling if the string might not convert successfully. Instead, code can check the Boolean result returned from the call to TryParse<T>().

Regardless of whether the code uses the “Parse” or “TryParse” approach, the key caution about converting from a string to an enum is that such a cast is not localizable. Therefore, developers should use this type of cast only for messages that are not exposed to users (assuming localization is a requirement).

|  |
| --- |
| Guidelines  AVOID direct enum/string conversions where the string must be localized into the user’s language. |

Enums as Flags

Many times, developers not only want enum values to be unique but also want to be able to represent a combination of values. For example, consider System.IO.FileAttributes. This enum, shown in Listing 9.14, indicates various attributes on a file: read-only, hidden, archive, and so on. Unlike with the ConnectionState attribute, where each enum value was mutually exclusive, the FileAttributes enum values can and are intended for combination: A file can be both read-only and hidden. To support this behavior, each enum value is a unique bit.

Listing 9.14: Using Enums as Flags

[Flags] public enum FileAttributes

{

ReadOnly = 1<<0, // 000000000000000001

Hidden = 1<<1, // 000000000000000010

System = 1<<2, // 000000000000000100

Directory = 1<<4, // 000000000000010000

Archive = 1<<5, // 000000000000100000

Device = 1<<6, // 000000000001000000

Normal = 1<<7, // 000000000010000000

Temporary = 1<<8, // 000000000100000000

SparseFile = 1<<9, // 000000001000000000

ReparsePoint = 1<<10, // 000000010000000000

Compressed = 1<<11, // 000000100000000000

Offline = 1<<12, // 000001000000000000

NotContentIndexed = 1<<13, // 000010000000000000

Encrypted = 1<<14, // 000100000000000000

IntegrityStream = 1<<15, // 001000000000000000

NoScrubData = 1<<17, // 100000000000000000

}

|  |
| --- |
| note  Note that the name of a bit flags enum is usually pluralized, indicating that a value of the type represents a set of flags. |

To join enum values, you use a bitwise OR operator. To test for the existence of a particular flag, use the Enum.HasFlags() method (which was added with Microsoft .NET Framework 4.0) or use the bitwise AND operator. Both cases are illustrated in Listing 9.15.

Listing 9.15: Using Bitwise OR and AND with Flag Enums[[3]](#footnote-4)

using System;

using System.IO;

public class Program

{

public static void Main()

{

// ...

string fileName = @"enumtest.txt";

System.IO.FileInfo file =

new System.IO.FileInfo(fileName);

file.Attributes = FileAttributes.Hidden |

FileAttributes.ReadOnly;

Console.WriteLine($"{file.Attributes} = {(int)file.Attributes}");

// Added in C# 4.0/Microsoft .NET Framwork 4.0

if (!(

System.Runtime.InteropServices.RuntimeInformation.IsOSPlatform(

OSPlatform.Linux) ||

System.Runtime.InteropServices.RuntimeInformation.IsOSPlatform(

OSPlatform.OSX)))

{

if (!file.Attributes.HasFlag(FileAttributes.Hidden))

{

throw new Exception(“File is not hidden.”);

}

}

// Use bit operators prior to C# 4.0/.NET 4.0

if (( file.Attributes & FileAttributes.ReadOnly) !=

FileAttributes.ReadOnly)

{

throw new Exception("File is not read-only.");

}

// ...

}

The results of Listing 9.15 appear in Output 9.5.

|  |
| --- |
| Output 9.5 |
| Hidden | ReadOnly = 3 |

Using the bitwise OR operator allows you to set the file to both read-only and hidden.

Each value within the enum does not need to correspond to only one flag. It is perfectly reasonable to define additional flags that correspond to frequent combinations of values. Listing 9.16 shows an example.

Listing 9.16: Defining Enum Values for Frequent Combinations

[Flags] enum DistributedChannel

{

None = 0,

Transacted = 1,

Queued = 2,

Encrypted = 4,

Persisted = 16,

FaultTolerant =

Transacted | Queued | Persisted

}

It is a good practice to have a zero None member in a flags enum because the initial default value of a field of enum type or an element of an array of enum type is 0. Avoid enum values corresponding to items such as Maximum as the last enum, because Maximum could be interpreted as a valid enum value. To check whether a value is included within an enum, use the System.Enum.IsDefined() method.

Guidelines

|  |
| --- |
| Guidelines  DO use the FlagsAttribute to mark enums that contain flags.  DO provide a None value equal to 0 for all flag enums.  AVOID creating flag enums where the zero value has a meaning other than “no flags are set.”  CONSIDER providing special values for commonly used combinations of flags.  DO NOT include “sentinel” values (such as a value called Maximum); such values can be confusing to the user.  DO use powers of 2 to ensure that all flag combinations are represented uniquely. |

|  |  |
| --- | --- |
|  | |
|  | AdVanced Topic  FlagsAttribute  If you decide to use bit flag enums, the declaration of the enum should be marked with FlagsAttribute. In such a case, the attribute appears in square brackets (see Chapter 18) just prior to the enum declaration, as shown in Listing 9.17.  Listing 9.17: Using FlagsAttribute  // FileAttributes defined in System.IO  [Flags] // Decorating an enum with FlagsAttribute  public enum FileAttributes  {  ReadOnly = 1<<0, // 000000000000001  Hidden = 1<<1, // 000000000000010  // ...  }  using System;  using System.Diagnostics;  using System.IO;  class Program  {  public static void Main()  {  string fileName = @"enumtest.txt";  FileInfo file = new FileInfo(fileName);  file.Open(FileMode.Create).Dispose();  FileAttributes startingAttributes =  file.Attributes;  file.Attributes = FileAttributes.Hidden |  FileAttributes.ReadOnly;  Console.WriteLine("\"{0}\" outputs as \"{1}\"",  file.Attributes.ToString().Replace(",", " |"),  file.Attributes);  FileAttributes attributes =  (FileAttributes) Enum.Parse(typeof(FileAttributes),  file.Attributes.ToString());  Console.WriteLine(attributes);  File.SetAttributes(fileName,  startingAttributes);  file.Delete();  }  }  The results of Listing 9.17 appear in Output 9.6.   |  | | --- | | Output 9.6 | | "ReadOnly | Hidden" outputs as "ReadOnly, Hidden"  ReadOnly, Hidden |   The attribute documents that the enum values can be combined. Furthermore, it changes the behavior of the ToString() and Parse() methods. For example, calling ToString() on an enum that is decorated with FlagsAttribute writes out the strings for each enum flag that is set. In Listing 9.17, file.Attributes.ToString() returns ReadOnly, Hidden rather than the 3 it would have returned without the FileAttributes flag. If two enum values are the same, the ToString() call would return the first value. As mentioned earlier, however, you should use caution when relying on this behavior because it is not localizable.  Parsing a value from a string to the enum also works. Each enum value identifier is separated from the others by a comma.  Note that FlagsAttribute does not automatically assign unique flag values or check that they have unique values. Doing this wouldn’t make sense, given that duplicates and combinations are often desirable. Instead, you must assign the values of each enum item explicitly. |
|  | |

Summary

This chapter began with a discussion of how to define custom value types. Because it is easy to write confusing or buggy code when mutating value types, and because value types are typically used to model immutable values, it is a good idea to make value types immutable. We also described how value types are boxed when they must be treated polymorphically as reference types.

The idiosyncrasies introduced by boxing are subtle, and the vast majority of them lead to problematic issues at execution time rather than at compile time. Although it is important to know about these quirks so as to try to avoid them, in many ways paying too much attention to the potential pitfalls overshadows the usefulness and performance advantages of value types. Programmers should not be overly concerned about using value types. Value types permeate virtually every chapter of this book, yet the idiosyncrasies associated with them come into play infrequently. We have staged the code surrounding each issue to demonstrate the concern, but in reality these types of patterns rarely occur. The key to avoiding most of them is to follow the guideline of not creating mutable value types, and following this constraint explains why you don’t encounter them within the built-in value types.

Perhaps the only issue to occur with some frequency is repetitive boxing operations within loops. However, generics greatly reduce boxing, and even without them, performance is rarely affected enough to warrant their avoidance until a particular algorithm with boxing is identified as a bottleneck.

Furthermore, custom-built structs are relatively rare. They obviously play an important role within C# development, but the number of custom-built structs declared by typical developers is usually tiny compared to the number of custom-built classes. Heavy use of custom-built structs is most common in code targeted at interoperating with unmanaged code.

|  |
| --- |
| Guidelines  DO NOT define a struct unless it logically represents a single value, consumes 16 bytes or less of storage, is immutable, and is infrequently boxed. |

This chapter also introduced enums. Enumerated types are a standard construct available in many programming languages. They help improve both API usability and code readability.

The next chapter presents more guidelines for creating well-formed types—both value types and reference types. It begins by looking at overriding the virtual members of objects and defining operator-overloading methods. These two topics apply to both structs and classes, but they are somewhat more important when completing a struct definition and making it well formed.

[Structs 6](#_Toc35623515)

[Initializing Structs 8](#_Toc35623516)

[Inheritance and Interfaces with Value Types 11](#_Toc35623517)

[Boxing 12](#_Toc35623518)

[Enums 21](#_Toc35623519)

[Type Compatibility between Enums 25](#_Toc35623520)

[Converting between Enums and Strings 26](#_Toc35623521)

[Enums as Flags 27](#_Toc35623522)

1. Note that it is important that we use a collection of type object, not a strongly typed collection like a generic collection as discussed in Chapter 12. [↑](#footnote-ref-2)
2. See the discussion on **Enums as Flags** later in this chapter. [↑](#footnote-ref-3)
3. . Note that the FileAttributes.Hidden value does not work on Linux. [↑](#footnote-ref-4)