Value Types

There are several value types defined in C#, including enumerators, structures, and primitives. Anytime you declare a variable of one of these types, you have allocated the number of bytes associated with that type on the stack and are working directly with that allocated array of bits. In addition, when you pass a variable that is a value type, you're passing that variable's value and not a reference to its underlying object.

Reference Types

reference types in C#: classes, arrays, delegates, and interfaces. Any time you declare a variable of one of these types, you allocate the number of bytes associated with that type on the heap, and you are working with a reference to that object instead of directly with the bits (as with value types).

**Boxing and Unboxing**

int foo = 42; // Value type.

object bar = foo; // foo is boxed to bar.

In the first line of this code, we're creating a variable (*foo*)of type *int*. As you know, the type *int* is a value type (because it's a primitive type). In the second line, the compiler sees that the variable *foo* is being copied to a reference type, which is represented by the variable *bar*. The compiler then spits out the MSIL code necessary to box this value.

Now, to convert *bar* back to a value type, you can perform an explicit cast:

int foo = 42; // Value type.

object bar = foo; // foo is boxed to bar.

int foo2 = (int)bar; // Unboxed back to int.

Notice that when boxing&%8212;that is, once again, when converting from a value type to a reference type—there is no explicit cast needed. However, when unboxing—converting from a reference type to a value type—the cast is needed. This is because in the case of unboxing, an object could be cast to any type. Therefore, the cast is necessary so that the compiler can verify that the cast is valid per the specified variable type.