Listkgljfkhgdfing 12.6:

Listing 12.8:

Listing 12.8B:

Listing 12.8C:

Listing 12.65:

Listing 12.23:

Listing 12.15:

Listing 12.2:

Listing 12.17:

Listing 12.19:

Listing 12.20:

Listing 12.24:

Listing 12.21:

Listing 12.7:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Beginner Topic  Categories of Types  All types Listing 12.2’s 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. |