Item 6: Distinguish Between Value Types and Reference Types

Value types or reference types? Structs or classes? When should you use each? This isn't C++, in which you define all types as value types and can create references to them. This isn't Java, in which everything is a reference type. You must decide how all instances of your type will behave when you create it. It's an important decision to get right the first time. You must live with the consequences of your decision because changing later can cause quite a bit of code to break in subtle ways. It's a simple matter of choosing the `struct` or `class` keyword when you create the type, but it's much more work to update all the clients using your type if you change it later.

It's not as simple as preferring one over the other. The right choice depends on how you expect to use the new type. Value types are not polymorphic. They are better suited to storing the data that your application manipulates. Reference types can be polymorphic and should be used to define the behavior of your application. Consider the expected responsibilities of your new type, and from those responsibilities, decide which type to create. Structs store data. Classes define behavior.

The distinction between value types and reference types was added to .NET and C# because of common problems that occurred in C++ and Java. In C++, all parameters and return values were passed by value. Passing by value is very efficient, but it suffers from one problem: partial copying (sometimes called slicing the object). If you use a derived object where a base object is expected, only the base portion of the object gets copied. You have effectively lost all knowledge that a derived object was ever there. Even calls to virtual functions are sent to the base class version.

The Java language responded by more or less removing value types from the language. All user-defined types are reference types. In the Java language, all parameters and return values are passed by reference. This strategy has the advantage of being consistent, but it's a drain on performance. Let's face it, some types are not polymorphic—they were not intended to be. Java programmers pay a heap allocation and an eventual garbage collection for every variable. They also pay an extra time cost to dereference every variable. All variables are references. In C#, you declare whether a new type should be a value type or a reference type using the `struct` or `class` keywords. Value types should be small, lightweight types. Reference types form your class hierarchy. This section examines

different uses for a type so that you understand all the distinctions between value types and reference types.

To start, this type is used as the return value from a method:

```
private MyData _myData;
public MyData Foo()
{
 return _myData;
}

// call it:
MyData v = Foo();
TotalSum += v.Value;
```

If `MyData` is a value type, the return value gets copied into the storage for `v`. Furthermore, `v` is on the stack. However, if `MyData` is a reference type, you've exported a reference to an internal variable. You've violated the principal of encapsulation (see Item 23).

Or, consider this variant:

```
private MyData _myData;
public MyData Foo()
{
 return _myData.Clone( ) as MyData;
}

// call it:
MyData v = Foo();
TotalSum += v.Value;
```

Now, `v` is a copy of the original `_myData`. As a reference type, two objects are created on the heap. You don't have the problem of exposing internal data. Instead, you've created an extra object on the heap. If `v` is a local variable, it quickly becomes garbage and `Clone` forces you to use runtime type checking. All in all, it's inefficient.

Types that are used to export data through public methods and properties should be value types. But that's not to say that every type returned from a public member should be a value type. There was an assumption in the earlier code snippet that `MyData` stores values. Its responsibility is to store those values.

But, consider this alternative code snippet:

```
private MyType _myType;
public IMyInterface Foo()
{
 return _myType as IMyInterface;
}

// call it:
IMyInterface iMe = Foo();
iMe.DoWork( );
```

The `_myType` variable is still returned from the `Foo` method. But this time, instead of accessing the data inside the returned value, the object is accessed to invoke a method through a defined interface. You're accessing the `MyType` object not for its data contents, but for its behavior. That behavior is expressed through the `IMyInterface`, which can be implemented by multiple different types. For this example, `MyType` should be a reference type, not a value type. `MyType`'s responsibilities revolve around its behavior, not its data members.

That simple code snippet starts to show you the distinction: Value types store values, and reference types define behavior. Now look a little deeper at how those types are stored in memory and the performance considerations related to the storage models. Consider this class:

```
public class C
{
  private MyType _a = new MyType( );
  private MyType _b = new MyType( );

  // Remaining implementation removed.
}

C var = new C();
```

How many objects are created? How big are they? It depends. If `MyType` is a value type, you've made one allocation. The size of that allocation is twice the size of `MyType`. However, if `MyType` is a reference type, you've made three allocations: one for the `C` object, which is 8 bytes (assuming 32-bit pointers), and two more for each of the `MyType` objects that are contained in a `C` object. The difference results because value types

are stored inline in an object, whereas reference types are not. Each variable of a reference type holds a reference, and the storage requires extra allocation.

To drive this point home, consider this allocation:

```
MyType [] var = new MyType[ 100 ];
```

If `MyType` is a value type, one allocation of 100 times the size of a `MyType` object occurs. However, if `MyType` is a reference type, one allocation just occurred. Every element of the array is null. When you initialize each element in the array, you will have performed 101 allocations—and 101 allocations take more time than 1 allocation. Allocating a large number of reference types fragments the heap and slows you down. If you are creating types that are meant to store data values, value types are the way to go.

The decision to make a value type or a reference type is an important one. It is a far-reaching change to turn a value type into a class type. Consider this type:

```
public struct Employee
{
  private string  _name;
  private int     _ID;
  private decimal _salary;

  // Properties elided

  public void Pay( BankAccount b )
  {
    b.Balance += _salary;
  }
}
```

This fairly simple type contains one method to let you pay your employees. Time passes, and the system runs fairly well. Then you decide that there are different classes of `Employees`: Salespeople get commissions, and managers get bonuses. You decide to change the `Employee` type into a class:

```
public class Employee
{
  private string  _name;
```

```
  private int     _ID;
  private decimal _salary;

  // Properties elided

  public virtual void Pay( BankAccount b )
  {
    b.Balance += _salary;
  }
}
```

That breaks much of the existing code that uses your customer struct. Return by value becomes return by reference. Parameters that were passed by value are now passed by reference. The behavior of this little snippet changed drastically:

```
Employee e1 = Employees.Find( "CEO" );
e1.Salary += Bonus; // Add one time bonus.
e1.Pay( CEOBankAccount );
```

What was a one-time bump in pay to add a bonus just became a permanent raise. Where a copy by value had been used, a reference is now in place. The compiler happily makes the changes for you. The CEO is probably happy, too. The CFO, on the other hand, will report the bug. You just can't change your mind about value and reference types after the fact: It changes behavior.

This problem occurred because the `Employee` type no longer follow the guidelines for a value type. In addition to storing the data elements that define an employee, you've added responsibilities—in this example, paying the employee. Responsibilities are the domain of class types. Classes can define polymorphic implementations of common responsibilities easily; structs cannot and should be limited to storing values.

The documentation for .NET recommends that you consider the size of a type as a determining factor between value types and reference types. In reality, a much better factor is the use of the type. Types that are simple structures or data carriers are excellent candidates for value types. It's true that value types are more efficient in terms of memory management: There is less heap fragmentation, less garbage, and less indirection. More important, value types are copied when they are returned from methods

or properties. There is no danger of exposing references to internal structures. But you pay in terms of features. Value types have very limited support for common object-oriented techniques. You cannot create object hierarchies of value types. You should consider all value types as though they were sealed. You can create value types that implement interfaces, but that requires boxing, which Item 17 shows causes performance degradation. Think of value types as storage containers, not objects in the OO sense.

You'll create more reference types than value types. If you answer yes to all these questions, you should create a value type. Compare these to the previous `Employee` example:

1. Is this type's principal responsibility data storage?
2. Is its public interface defined entirely by properties that access or modify its data members?
3. Am I confident that this type will never have subclasses?
4. Am I confident that this type will never be treated polymorphically?

Build low-level data storage types as value types. Build the behavior of your application using reference types. You get the safety of copying data that gets exported from your class objects. You get the memory usage benefits that come with stack-based and inline value storage, and you can utilize standard object-oriented techniques to create the logic of your application. When in doubt about the expected use, use a reference type.