Item 7: Prefer Immutable Atomic Value Types

Immutable types are simple: After they are created, they are constant. If you validate the parameters used to construct the object, you know that it is in a valid state from that point forward. You cannot change the object's internal state to make it invalid. You save yourself a lot of otherwise necessary error checking by disallowing any state changes after an object has been constructed. Immutable types are inherently thread safe: Multiple readers can access the same contents. If the internal state cannot change, there is no chance for different threads to see inconsistent views of the data. Immutable types can be exported from your objects safely. The caller cannot modify the internal state of your objects. Immutable types work better in hash-based collections. The value returned by `Object.GetHashCode()` must be an instance invariant (see Item 10); that's always true for immutable types.

Not every type can be immutable. If it were, you would need to clone objects to modify any program state. That's why this recommendation is for both atomic and immutable value types. Decompose your types to the structures that naturally form a single entity. An `Address` type does. An address is a single thing, composed of multiple related fields. A change in one field likely means changes to other fields. A customer type is not an atomic type. A customer type will likely contain many pieces of information: an address, a name, and one or more phone numbers. Any of these independent pieces of information might change. A customer might change phone numbers without moving. A customer might move, yet still keep the same phone number. A customer might change his or her name without moving or changing phone numbers. A customer object is not atomic; it is built from many different immutable types using composition: an address, a name, or a collection of phone number/type pairs. Atomic types are single entities: You would naturally replace the entire contents of an atomic type. The exception would be to change one of its component fields.

Here is a typical implementation of an address that is mutable:

```
// Mutable Address structure.
public struct Address
{
  private string  _line1;
  private string _line2;
```

```
private string  _city;
private string _state;
private int    _zipCode;

// Rely on the default system-generated
// constructor.

public string Line1
{
  get { return _line1; }
  set { _line1 = value; }
}
public string Line2
{
  get { return _line2; }
  set { _line2 = value; }
}
public string City
{
  get { return _city; }
  set { _city= value; }
}
public string State
{
  get { return _state; }
  set
  {
    ValidateState(value);
    _state = value;
  }
}
public int ZipCode
{
  get { return _zipCode; }
  set
  {
    ValidateZip( value );
    _zipCode = value;
  }
}
```

```
  // other details omitted.
}

// Example usage:
Address a1 = new Address( );
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111 ;
// Modify:
a1.City = "Ann Arbor"; // Zip, State invalid now.
a1.ZipCode = 48103; // State still invalid now.
a1.State = "MI"; // Now fine.
```

Internal state changes means that it's possible to violate object invariants, at least temporarily. After you have replaced the `City` field, you have placed `a1` in an invalid state. The city has changed and no longer matches the state or ZIP code fields. The code looks harmless enough, but suppose that this fragment is part of a multithreaded program. Any context switch after the city changes and before the state changes would leave the potential for another thread to see an inconsistent view of the data.

Okay, so you're not writing a multithreaded program. You can still get into trouble. Imagine that the ZIP code was invalid and the set threw an exception. You've made only some of the changes you intended, and you've left the system in an invalid state. To fix this problem, you would need to add considerable internal validation code to the address structure. That validation code would add considerable size and complexity. To fully implement exception safety, you would need to create defensive copies around any code block in which you change more than one field. Thread safety would require adding significant thread-synchronization checks on each property accessor, both sets and gets. All in all, it would be a significant undertaking—and one that would likely be extended over time as you add new features.

Instead, make the `Address` structure an immutable type. Start by changing all instance fields to read-only:

```
public struct Address
{
  private readonly string  _line1;
  private readonly string  _line2;
```

```
  private readonly string  _city;
  private readonly string  _state;
  private readonly int    _zipCode;

  // remaining details elided
}
```

You'll also want to remove all set accessors to each property:

```
public struct Address
{
  // ...
  public string Line1
  {
    get { return _line1; }
  }
  public string Line2
  {
    get { return _line2; }
  }
  public string City
  {
    get { return _city; }
  }
  public string State
  {
    get { return _state; }
  }
  public int ZipCode
  {
    get { return _zipCode; }
  }
}
```

Now you have an immutable type. To make it useful, you need to add all necessary constructors to initialize the `Address` structure completely. The `Address` structure needs only one additional constructor, specifying each field. A copy constructor is not needed because the assignment operator is just as efficient. Remember that the default constructor is still

accessible. There is a default address where all the strings are null, and the ZIP code is 0:

```
public struct Address
{
  private readonly string  _line1;
  private readonly string  _line2;
  private readonly string  _city;
  private readonly string  _state;
  private readonly int    _zipCode;

  public Address( string line1,
    string line2,
    string city,
    string state,
    int zipCode)
  {
    _line1 = line1;
    _line2 = line2;
    _city = city;
    _state = state;
    _zipCode = zipCode;
    ValidateState( state );
    ValidateZip( zipCode );
  }

  // etc.
}
```

Using the immutable type requires a slightly different calling sequence to modify its state. You create a new object rather than modify the existing instance:

```
// Create an address:
Address a1 = new Address( "111 S. Main",
  "", "Anytown", "IL", 61111 );

// To change, re-initialize:
a1 = new Address( a1.Line1,
  a1.Line2, "Ann Arbor", "MI", 48103 );
```

The value of `a1` is in one of two states: its original location in Anytown, or its updated location in Ann Arbor. You do not modify the existing address to create any of the invalid temporary states from the previous example. Those interim states exist only during the execution of the `Address` constructor and are not visible outside of that constructor. As soon as a new `Address` object is constructed, its value is fixed for all time. It's exception safe: `a1` has either its original value or its new value. If an exception is thrown during the construction of the new `Address` object, the original value of `a1` is unchanged.

To create an immutable type, you need to ensure that there are no holes that would allow clients to change your internal state. Value types do not support derived types, so you do not need to defend against derived types modifying fields. But you do need to watch for any fields in an immutable type that are mutable reference types. When you implement your constructors for these types, you need to make a defensive copy of that mutable type. All these examples assume that `Phone` is an immutable value type because we're concerned only with immutability in value types:

```
// Almost immutable: there are holes that would
// allow state changes.
public struct PhoneList
{
  private readonly Phone[] _phones;

  public PhoneList( Phone[] ph )
  {
    _phones = ph;
  }

  public IEnumerator Phones
  {
    get
    {
      return _phones.GetEnumerator();
    }
  }
}

Phone[] phones = new Phone[10];
```

```
// initialize phones
PhoneList pl = new PhoneList( phones );

// Modify the phone list:
// also modifies the internals of the (supposedly)
// immutable object.
phones[5] = Phone.GeneratePhoneNumber( );
```

The array class is a reference type. The array referenced inside the `PhoneList` structure refers to the same array storage (phones) allocated outside of the object. Developers can modify your immutable structure through another variable that refers to the same storage. To remove this possibility, you need to make a defensive copy of the array. The previous example shows the pitfalls of a mutable collection. Even more possibilities for mischief exist if the `Phone` type is a mutable reference type. Clients could modify the values in the collection, even if the collection is protected against any modification. This defensive copy should be made in all constructors whenever your immutable type contains a mutable reference type:

```
// Immutable: A copy is made at construction.
public struct PhoneList
{
  private readonly Phone[] _phones;

  public PhoneList( Phone[] ph )
  {
     _phones = new Phone[ ph.Length ];
     // Copies values because Phone is a value type.
     ph.CopyTo( _phones, 0 );
  }

  public IEnumerator Phones
  {
    get
    {
      return _phones.GetEnumerator();
    }
  }
}
```

```
Phone[] phones = new Phone[10];
// initialize phones
PhoneList pl = new PhoneList( phones );

// Modify the phone list:
// Does not modify the copy in pl.
phones[5] = Phone.GeneratePhoneNumber( );
```

You need to follow the same rules when you return a mutable reference type. If you add a property to retrieve the entire array from the `PhoneList` struct, that accessor would also need to create a defensive copy. See Item 23 for more details.

The complexity of a type dictates which of three strategies you will use to initialize your immutable type. The `Address` structure defined one constructor to allow clients to initialize an address. Defining the reasonable set of constructors is often the simplest approach.

You can also create factory methods to initialize the structure. Factories make it easier to create common values. The .NET Framework `Color` type follows this strategy to initialize system colors. The static methods `Color.FromKnownColor()` and `Color.FromName()` return a copy of a color value that represents the current value for a given system color.

Third, you can create a mutable companion class for those instances in which multistep operations are necessary to fully construct an immutable type. The .NET string class follows this strategy with the `System.Text.StringBuilder` class. You use the `StringBuilder` class to create a string using multiple operations. After performing all the operations necessary to build the string, you retrieve the immutable string from the `StringBuilder`.

Immutable types are simpler to code and easier to maintain. Don't blindly create `get` and `set` accessors for every property in your type. Your first choice for types that store data should be immutable, atomic value types. You easily can build more complicated structures from these entities.