Item 10: Understand the Pitfalls of `GetHashCode()`

This is the only item in this book dedicated to one function that you should avoid writing. `GetHashCode()` is used in one place only: to define the hash value for keys in a hash-based collection, typically the `Hashtable` or `Dictionary` containers. That's good because there are a number of problems with the base class implementation of `GetHashCode()`. For reference types, it works but is inefficient. For value types, the base class version is often incorrect. But it gets worse. It's entirely possible that you cannot write `GetHashCode()` so that it is both efficient and correct. No single function generates more discussion and more confusion than `GetHashCode()`. Read on to remove all that confusion.

If you're defining a type that won't ever be used as the key in a container, this won't matter. Types that represent window controls, web page controls, or database connections are unlikely to be used as keys in a collection. In those cases, do nothing. All reference types will have a hash code that is correct, even if it is very inefficient. Value types should be immutable (see Item 7), in which case, the default implementation always works, although it is also inefficient. In most types that you create, the best approach is to avoid the existence of `GetHashCode()` entirely.

One day, you'll create a type that is meant to be used as a hashtable key, and you'll need to write your own implementation of `GetHashCode()`, so read on. Hash-based containers use hash codes to optimize searches. Every object generates an integer value called a hash code. Objects are stored in buckets based on the value of that hash code. To search for an object, you request its key and search just that one bucket. In .NET, every object has a hash code, determined by `System.Object.GetHashCode()`. Any overload of `GetHashCode()` must follow these three rules:

1. If two objects are equal (as defined by operator==), they must generate the same hash value. Otherwise, hash codes can't be used to find objects in containers.
2. For any object `A`, `A.GetHashCode()` must be an instance invariant. No matter what methods are called on `A`, `A.GetHashCode()` must always return the same value. That ensures that an object placed in a bucket is always in the right bucket.
3. The hash function should generate a random distribution among all integers for all inputs. That's how you get efficiency from a hash-based container.

Writing a correct and efficient hash function requires extensive knowledge of the type to ensure that rule 3 is followed. The versions defined in `System.Object` and `System.ValueType` do not have that advantage. These versions must provide the best default behavior with almost no knowledge of your particular type. `Object.GetHashCode()` uses an internal field in the `System.Object` class to generate the hash value. Each object created is assigned a unique object key, stored as an integer, when it is created. These keys start at `1` and increment every time a new object of any type gets created. The object identity field is set in the `System.Object` constructor and cannot be modified later. `Object.GetHashCode()` returns this value as the hash code for a given object.

Now examine `Object.GetHashCode()` in light of those three rules. If two objects are equal, `Object.GetHashCode()`returns the same hash value, unless you've overridden `operator==`. `System.Object`'s version of `operator==()` tests object identity. `GetHashCode()` returns the internal object identity field. It works. However, if you've supplied your own version of `operator==`, you must also supply your own version of `GetHashCode()` to ensure that the first rule is followed. See Item 9 for details on equality.

The second rule is followed: After an object is created, its hash code never changes.

The third rule, a random distribution among all integers for all inputs, does not hold. A numeric sequence is not a random distribution among all integers unless you create an enormous number of objects. The hash codes generated by `Object.GetHashCode()` are concentrated at the low end of the range of integers.

This means that `Object.GetHashCode()` is correct but not efficient. If you create a hashtable based on a reference type that you define, the default behavior from `System.Object` is a working, but slow, hashtable. When you create reference types that are meant to be hash keys, you should override `GetHashCode()`to get a better distribution of the hash values across all integers for your specific type.

Before covering how to write your own override of `GetHashCode`, this section examines `ValueType.GetHashCode()`with respect to those same three rules. `System.ValueType` overrides `GetHashCode()`, providing the

default behavior for all value types. Its version returns the hash code from the first field defined in the type. Consider this example:

```
public struct MyStruct
{
  private string   _msg;
  private int      _id;
  private DateTime _epoch;
}
```

The hash code returned from a `MyStruct` object is the hash code generated by the `_msg` field. The following code snippet always returns `true`:

```
MyStruct s = new MyStruct( );
return s.GetHashCode( ) == s._msg.GetHashCode( );
```

The first rule says that two objects that are equal (as defined by `operator==()`) must have the same hash code. This rule is followed for value types under most conditions, but you can break it, just as you could with for reference types. `ValueType.operator==()` compares the first field in the struct, along with every other field. That satisfies rule 1. As long as any override that you define for `operator==` uses the first field, it will work. Any struct whose first field does not participate in the equality of the type violates this rule, breaking `GetHashCode()`.

The second rule states that the hash code must be an instance invariant. That rule is followed only when the first field in the struct is an immutable field. If the value of the first field can change, so can the hash code. That breaks the rules. Yes, `GetHashCode()` is broken for any struct that you create when the first field can be modified during the lifetime of the object. It's yet another reason why immutable value types are your best bet (see Item 7).

The third rule depends on the type of the first field and how it is used. If the first field generates a random distribution across all integers, and the first field is distributed across all values of the struct, then the struct generates an even distribution as well. However, if the first field often has the same value, this rule is violated. Consider a small change to the earlier struct:

```
public struct MyStruct
{
  private DateTime _epoch;
```

```
  private string   _msg;
  private int      _id;
}
```

If the `_epoch` field is set to the current date (not including the time), all `MyStruct` objects created in a given date will have the same hash code. That prevents an even distribution among all hash code values.

Summarizing the default behavior, `Object.GetHashCode()` works correctly for reference types, although it does not necessarily generate an efficient distribution. (If you have overridden `Object.operator==()`, you can break `GetHashCode()`). `ValueType.GetHashCode()` works only if the first field in your struct is read-only. `ValueType.GetHashCode()` generates an efficient hash code only when the first field in your struct contains values across a meaningful subset of its inputs.

If you're going to build a better hash code, you need to place some constraints on your type. Examine the three rules again, this time in the context of building a working implementation of `GetHashCode()`.

First, if two objects are equal, as defined by `operator==()`, they must return the same hash value. Any property or data value used to generate the hash code must also participate in the equality test for the type. Obviously, this means that the same properties used for equality are used for hash code generation. It's possible to have properties participate in equality that are not used in the hash code computation. The default behavior for `System.ValueType` does just that, but it often means that rule 3 usually gets violated. The same data elements should participate in both computations.

The second rule is that the return value of `GetHashCode()` must be an instance invariant. Imagine that you defined a reference type, `Customer`:

```
public class Customer
{
  private string _name;
  private decimal _revenue;

  public Customer( string name )
  {
    _name = name;
  }
```

```
  public string Name
  {
    get { return _name; }
    set { _name = value; }
  }

  public override int GetHashCode()
  {
    return _name.GetHashCode();
  }
}
```

Suppose that you execute the following code snippet:

```
Customer c1 = new Customer( "Acme Products" );
myHashMap.Add( c1, orders );
// Oops, the name is wrong:
c1.Name = "Acme Software";
```

`c1` is lost somewhere in the hash map. When you placed `c1` in the map, the hash code was generated from the string `"Acme Products"`. After you change the name of the customer to `"Acme Software"`, the hash code value changed. It's now being generated from the new name: `"Acme Software"`. C1 is stored in the bucket defined by `"Acme Products"`, but it should be in the bucket defined for `"Acme Software"`. You've lost that customer in your own collection. It's lost because the hash code is not an object invariant. You've changed the correct bucket after storing the object.

The earlier situation can occur only if `Customer` is a reference type. Value types misbehave differently, but they still cause problems. If customer is a value type, a copy of `c1` gets stored in the hashmap. The last line changing the value of the name has no effect on the copy stored in the hashmap. Because boxing and unboxing make copies as well, it's very unlikely that you can change the members of a value type after that object has been added to a collection.

The only way to address rule 2 is to define the hash code function to return a value based on some invariant property or properties of the object. `System.Object` abides by this rule using the object identity, which does not change. `System.ValueType` hopes that the first field in

your type does not change. You can't do better without making your type immutable. When you define a value type that is intended for use as a key type in a hash container, it must be an immutable type. Violate this recommendation, and the users of your type will find a way to break hashtables that use your type as keys. Revisiting the `Customer` class, you can modify it so that the customer name is immutable:

```
public class Customer
{
  private readonly string _name;
  private decimal _revenue;

  public Customer( string name ) :
    this ( name, 0 )
  {
  }

  public Customer( string name, decimal revenue )
  {
    _name = name;
    _revenue = revenue;
  }

  public string Name
  {
    get { return _name; }
  }

  // Change the name, returning a new object:
  public Customer ChangeName( string newName )
  {
    return new Customer( newName, _revenue );
  }

  public override int GetHashCode()
  {
    return _name.GetHashCode();
  }
}
```

Making the name immutable changes how you must work with customer objects to modify the name:

```
Customer c1 = new Customer( "Acme Products" );
myHashMap.Add( c1,orders );
// Oops, the name is wrong:
Customer c2 = c1.ChangeName( "Acme Software" );
Order o = myHashMap[ c1 ] as Order;
myHashMap.Remove( c1 );
myHashMap.Add( c2, o );
```

You have to remove the original customer, change the name, and add the new customer object to the hashtable. It looks more cumbersome than the first version, but it works. The previous version allowed programmers to write incorrect code. By enforcing the immutability of the properties used to calculate the hash code, you enforce correct behavior. Users of your type can't go wrong. Yes, this version is more work. You're forcing developers to write more code, but only because it's the only way to write the correct code. Make certain that any data members used to calculate the hash value are immutable.

The third rule says that `GetHashCode()` should generate a random distribution among all integers for all inputs. Satisfying this requirement depends on the specifics of the types you create. If a magic formula existed, it would be implemented in `System.Object` and this item would not exist. A common and successful algorithm is to XOR all the return values from `GetHashCode()` on all fields in a type. If your type contains some mutable fields, exclude those fields from the calculations.

`GetHashCode()` has very specific requirements: Equal objects must produce equal hash codes, and hash codes must be object invariants and must produce an even distribution to be efficient. All three can be satisfied only for immutable types. For other types, rely on the default behavior, but understand the pitfalls.