Value Type Internals

Now that we have an idea how reference types are laid out in memory and what purpose the object header fields serve, it's time to discuss value types. Value types have a much simpler memory layout, but it introduces limitations and boxing, an expensive process that compensates for the incompatibility of using a value type where a reference is required. The primary reason for using value types, as we have seen, is their excellent memory density and lack of overhead; every bit of performance matters when you develop your own value types.

For the purpose of our discussion, let's commit to a simple value type we discussed at the onset of this chapter, `Point2D`, representing a point in two-dimensional space:

```
public struct Point2D
{
  public int X;
  public int Y;
}
```

The memory layout of a `Point2D` instance initialized with `X=5`, `Y=7` is simply the following, with no additional "overhead" fields clutter:

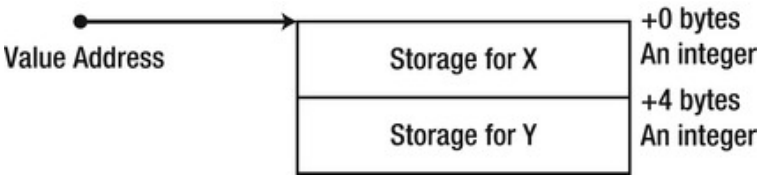

***Figure 3-7**. Memory layout for a Point2D value type instance*

In some rare cases, it may be desired to customize value type layout – one example is for interoperability purposes, when your value type instance is passed unchanged to unmanaged code. This customization is possible through the use of two attributes, `StructLayout` and `FieldOffset`. The `StructLayout` attribute can be used to specify that the object's fields are to be laid out sequentially according to the type's definition (this is the default), or explicitly according to the instructions provided by the `FieldOffset` attribute. This enables the creation of C-style unions, where fields may overlap. A crude example of this is the following value type, which can "convert" a floating-point number to the four bytes used by its representation:

```
[StructLayout(LayoutKind.Explicit)]
public struct FloatingPointExplorer
{
  [FieldOffset(0)] public float F;
  [FieldOffset(0)] public byte B1;
  [FieldOffset(1)] public byte B2;
  [FieldOffset(2)] public byte B3;
  [FieldOffset(3)] public byte B4;
}
```

When you assign a floating-point value to the object's `F` field, it concurrently modifies the values of `B1-B4`, and vice versa. Effectively, the `F` field and the `B1-B4` fields overlap in memory, as demonstrated by Figure 3-8:

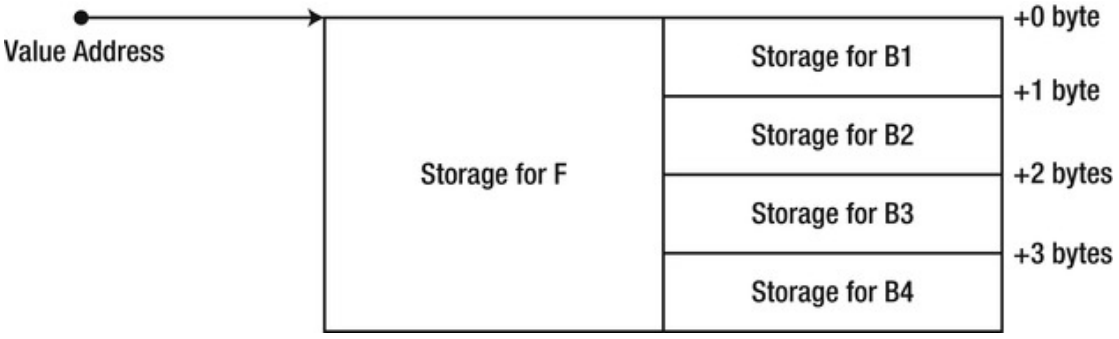

***Figure 3-8**. Memory layout for a FloatingPointExplorer instance. Blocks aligned horizontally overlap in memory*

Because value type instances do not have an object header word and a method table pointer, they cannot allow the same richness of semantics offered by reference types. We will now look at the limitations their simpler layout introduces, and what happens when developers attempt to use value types in settings intended for reference types.

Value Type Limitations

First, consider the object header word. If a program attempts to use a value type instance for synchronization, it is most often a bug in the program (as we will see shortly), but should the runtime make it illegal and throw an exception? In the following code example, what should happen when the `Increment` method of the same `Counter` class instance is executed by two different threads?

```
class Counter
{
  private int _i;
  public int Increment()
  {
```

```
    lock (_i)
    {
    return ++_i;
    }
   }
}
```

As we attempt to verify what happens, we stumble upon an unexpected obstacle: the C# compiler would not allow using value types with the `lock` keyword. However, we are now seasoned in the inner workings of what the `lock` keyword does, and can attempt to code a workaround:

```
class Counter
{
  private int _i;
  public int Increment()
  {
   bool acquired=false;
   try
   {
     Monitor.Enter(_i, ref acquired);
     return ++_i;
   }
   finally
   {
     if (acquired) Monitor.Exit(_i);
   }
 }
}
```

By doing so, we introduced a bug into our program – it turns out that multiple threads will be able to enter the lock and modify `_i` at once, and furthermore the `Monitor.Exit` call will throw an exception (to understand the proper ways of synchronizing access to an integer variable, refer to Chapter 6). The problem is that the `Monitor.Enter` method accepts a `System.Object` parameter, which is a reference, and we are passing to it a value type – by value. Even if it were possible to pass the value unmodified where a reference is expected, the value passed to the `Monitor.Enter` method does not have the same *identity* as the value passed to the `Monitor.Exit` method; similarly, the value passed to the `Monitor.Enter` method on one thread does not have the same *identity* as the value passed to the `Monitor.Enter` method on another thread. If we pass around values (by value!) where references are expected, there is no way to obtain the correct locking semantics.

Another example of why value type semantics are not a good fit for object references arises when returning a value type from a method. Consider the following code:

```
object GetInt()
{
  int i = 42;
  return i;
}
object obj = GetInt();
```

The `GetInt` method returns a value type – which would typically be returned by value. However, the caller expects, from the method, to return an object reference. The method could return a direct pointer to the stack location where `i` is stored during the method's execution. Unfortunately, it would be a reference to an invalid memory location, because the stack frame for the method is cleaned up before it returns. This demonstrates that the copy-by-value semantics value types have by default are not a good fit for when an object reference (into the managed heap) is expected.

Virtual Methods on Value Types

We haven't considered the method table pointer yet, and already we have insurmountable problems when attempting to treat value types as first-class citizens. Now we turn to virtual methods and interface implementations. The CLR forbids inheritance relationships between value types, making it impossible to define new virtual methods on value types. This is fortunate, because if it were possible to define virtual methods on value types, invoking these methods would require a method table pointer which is not part of a value type instance. This is not a substantial limitation, because the copy-by-value semantics of reference types make them ill suited for polymorphism, which requires object references.

However, value types come equipped with virtual methods inherited from `System.Object`. There are several of them: `Equals`, `GetHashCode`, `ToString`, and `Finalize`. We will discuss only the first two here, but much of the discussion applies to the other virtual methods as well. Let's start by inspecting their signatures:

```
public class Object
{
  public virtual bool Equals(object obj) ...
  public virtual int GetHashCode() ...
}
```

These virtual methods are implemented by every .NET type, including value types. It means that given an instance of a value type, we should be able to dispatch the virtual method successfully, even though it doesn't have a method table pointer! This third example of how the value type memory layout affects our ability to do even simple operations with value type instances demands a mechanism that can "turn" value type instances into something that can more plausibly stand for an "authentic" object.

Boxing

Whenever the language compiler detects a situation that requires treating a value type instance as a reference type, it emits the `box` IL instruction. The JIT compiler, in turn, interprets this instruction and emits a call to a method that allocates heap storage, copies the content of the value type instance to the heap, and wraps the value type contents with an object header – object header word and method table pointer. It is this "box" that is used whenever an object reference is required. Note that the box is detached from the original value type instance – changes made to one do not affect the other.

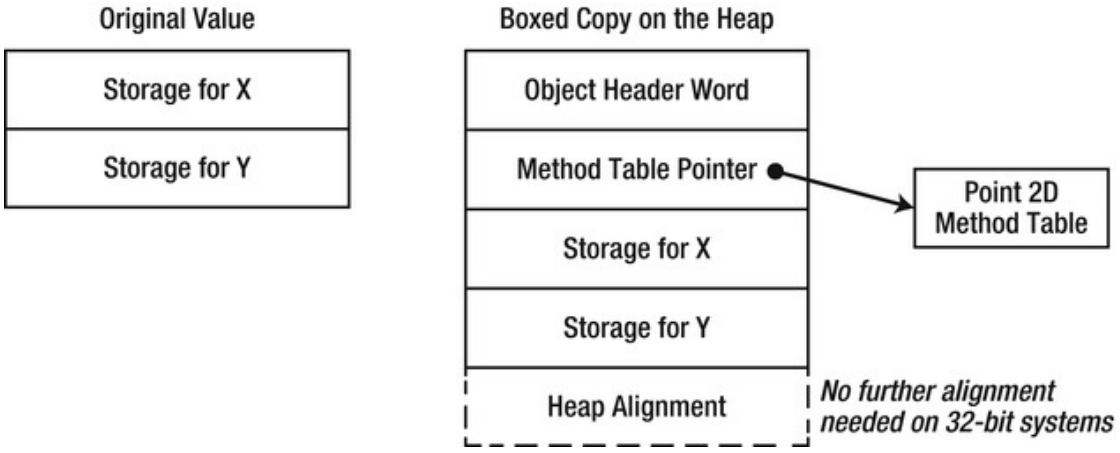

***Figure 3-9**. Original value and boxed copy on the heap. The boxed copy has the standard reference type "overhead" (object header word and method table pointer), and may require further heap alignment*

```
.method private hidebysig static object GetInt() cil managed
{
   .maxstack 8
   L_0000: ldc.i4.s 0x2a
   L_0002: box int32
   L_0007: ret
}
```

Boxing is an expensive operation – it involves a memory allocation, a memory copy, and subsequently creates pressure on the garbage collector when it struggles to reclaim the temporary boxes. With the introduction of generics in CLR 2.0, there is hardly any need for boxing outside Reflection and other obscure scenarios. Nonetheless, boxing remains a significant performance problem in many applications; as we will see, "getting value types right" to prevent all kinds of boxing is not trivial without some further understanding of how method dispatch on value types operates.

Setting the performance problems aside, boxing provides a remedy to some of the problems we encountered earlier. For example, the `GetInt` method returns a reference to box on the heap which contains the value 42. This box will survive as long as there are references to it, and is not affected by the lifetime of local variables on the method's stack. Similarly, when the `Monitor.Enter` method expects an object reference, it receives at runtime a reference to a box on the heap and uses that box for synchronization. Unfortunately, boxes created from the same value type instance at different points in code are not considered identical, so the box passed to `Monitor.Exit` is not the same box passed to `Monitor.Enter`, and the box passed to `Monitor.Enter` on one thread is not the same box passed to `Monitor.Enter` on another thread. This means that any use of value types for monitor-based synchronization is inherently wrong, regardless of the partial cure afforded by boxing.

The crux of the matter remains the virtual methods inherited from `System.Object`. As it turns out, value types do not derive from `System.Object` directly; instead, they derive from an intermediate type called `System.ValueType`.

Note Confusingly, `System.ValueType` is a reference type – the CLR tells value types and reference types apart based on the following criterion: value types are types derived from `System.ValueType`. According to this criterion, `System.ValueType` is a reference type.

`System.ValueType` overrides the `Equals` and `GetHashCode` virtual methods inherited from `System.Object`, and does it for a good reason: value types have different default equality semantics from reference types and these defaults must be implemented somewhere. For example, the overridden `Equals` method in `System.ValueType` ensures that value types are compared based on their content, whereas the original `Equals` method in `System.Object` compares only the object references (identity).

Regardless of how `System.ValueType` implements these overridden virtual methods, consider the following scenario. You embed ten million `Point2D` objects in a `List<Point2D>`, and proceed to look up a single `Point2D` object in the list using the `Contains` method. In turn, `Contains` has no better alternative than to perform a linear search through ten million objects and compare them individually to the one you provided.

```
List<Point2D> polygon = new List<Point2D>();
//insert ten million points into the list
Point2D point = new Point2D { X = 5, Y = 7 };
bool contains = polygon.Contains(point);
```

Traversing a list of ten million points and comparing them one-by-one to another point takes a while, but it's a relatively quick operation. The number of bytes accessed is approximately 80,000,000 (eight bytes for each `Point2D` object), and the comparison operations are very quick. Regrettably, comparing two `Point2D` objects requires calling the `Equals` virtual method:

```
Point2D a = ..., b = ...;
a.Equals(b);
```

There are two issues at stake here. First, `Equals` – even when overridden by `System.ValueType` – accepts a `System.Object` reference as its parameter. Treating a `Point2D` object as an object reference requires boxing, as we have already seen, so `b` would have to be boxed. Moreover, dispatching the `Equals` method call requires boxing `a` to obtain the method table pointer!

Note The JIT compiler has a short-circuit behavior that could permit a direct method call to `Equals`, because value types are sealed and the virtual dispatch target is determined at compile-time by whether `Point2D` overrides `Equals` or not (this is enabled by the `constrained` IL prefix). Nevertheless, because `System.ValueType` is a reference type, the `Equals` method is free to treat its `this` implicit parameter as a reference type as well, whereas we are using a value type instance (`Point2D a`) to call `Equals` – and this requires boxing.

To summarize, we have two boxing operations for each `Equals` call on `Point2D` instances. For the 10,000,000 `Equals` calls performed by the code above, we have 20,000,000 boxing operations, each allocating (on a 32-bit system) 16 bytes, for a total of 320,000,000 bytes of allocations and 160,000,000 bytes of memory copied into the heap. The cost of these allocations surpasses by far the time required to actually compare points in two-dimensional space.

Avoiding Boxing on Value Types with the Equals Method

What can we do to get rid of these boxing operations entirely? One idea is to override the `Equals` method and provide an implementation suitable for our value type:

```
public struct Point2D
{
  public int X;
```

```
  public int Y;
  public override bool Equals(object obj)
  {
   if (!(obj is Point2D)) return false;
   Point2D other = (Point2D)obj;
   return X == other.X && Y == other.Y;
  }
}
```

Using the JIT-compiler's short-circuit behavior discussed previously, `a.Equals(b)` still requires boxing for `b`, because the method accepts an object reference, but no longer requires boxing for `a`. To get rid of the second boxing operation, we need to think outside the box (pun intended) and provide an *overload* of the `Equals` method:

```
public struct Point2D
{
  public int X;
  public int Y;
  public override bool Equals(object obj) ... //as before
  public bool Equals(Point2D other)
  {
   return X == other.X && Y == other.Y;
   }
}
```

Whenever the compiler encounters `a.Equals(b)`, it will definitely prefer the second overload to the first, because its parameter type matches more closely the argument type provided. While we're at it, there are some more methods to overload – often enough, we compare objects using the `==` and `!=` operators:

```
public struct Point2D
{
  public int X;
  public int Y;
  public override bool Equals(object obj) ... // as before
  public bool Equals(Point2D other) ... //as before
  public static bool operator==(Point2D a, Point2D b)
  {
   return a.Equals(b);
  }
  public static bool operator!= (Point2D a, Point2D b)
  {
   return !(a == b);
  }
}
```

This is almost enough. There is an edge case that has to do with the way the CLR implements generics, which still causes boxing when `List<Point2D>` calls `Equals` on two `Point2D` instances, with `Point2D` as a realization of its generic type parameter (`T`). We will discuss the exact details in Chapter 5; for now it suffices to say that `Point2D` needs to implement `IEquatable<Point2D>`, which allows clever behavior in `List<T>` and `EqualityComparer<T>` to dispatch the method call to the *overloaded* `Equals` method through the interface (at the cost of a virtual method call to the `EqualityComparer<T>.Equals` abstract method). The result is a 10-fold improvement in execution time and complete elimination of all memory allocations (introduced by boxing) when searching a list of 10,000,000 `Point2D` instances for a specific one!

```
public struct Point2D : IEquatable<Point2D>
{
  public int X;
  public int Y;
  public bool Equals(Point2D other) ... //as before
}
```

This is a good time to reflect upon the subject of interface implementations on value types. As we have seen, a typical interface method dispatch requires the object's method table pointer, which would solicit boxing where value types are concerned. Indeed, a conversion from a value type instance to an interface type variable requires boxing, because interface references can be treated as object references for all intents and purposes:

```
Point2D point = ...;
IEquatable<Point2D> equatable = point; //boxing occurs here
```

However, when making an interface call through a statically typed value type variable, no boxing will occur (this is the same short-circuiting enabled by the `constrained` IL prefix, discussed above):

```
Point2D point = ..., anotherPoint = ...;
point.Equals(anotherPoint); //no boxing occurs here, Point2D.Equals(Point2D) is invoked
```

Using value types through interfaces raises a potential concern if the value types are mutable, such as the `Point2D` we are churning through in this chapter. As always, modifying the boxed copy will not affect the original, which can lead to unexpected behavior:

```
Point2D point = new Point2D { X = 5, Y = 7 };
Point2D anotherPoint = new Point2D { X = 6, Y = 7 };
IEquatable<Point2D> equatable = point; //boxing occurs here
equatable.Equals(anotherPoint); //returns false
point.X = 6;
```

```
point.Equals(anotherPoint); //returns true
equatable.Equals(anotherPoint); //returns false, the box was not modified!
```

This is one ground for the common recommendation to make value types immutable, and allow modification only by making more copies. (Consider the `System.DateTime` API for an example of a well-designed immutable value type.)

The final nail in the coffin of `ValueType.Equals` is its actual implementation. Comparing two arbitrary value type instances according to their content is not trivial. Disassembling the method offers the following picture (slightly edited for brevity):

```
public override bool Equals(object obj)
{
  if (obj == null) return false;
  RuntimeType type = (RuntimeType) base.GetType();
  RuntimeType type2 = (RuntimeType) obj.GetType();
  if (type2 != type) return false;
  object a = this;
  if (CanCompareBits(this))
  {
   return FastEqualsCheck(a, obj);
  }
  FieldInfo[] fields = type.GetFields(BindingFlags.NonPublic | ← BindingFlags.Public | BindingFlags.Instance)
  for (int i = 0; i < fields.Length; i++)
  {
   object obj3 = ((RtFieldInfo) fields[i]).InternalGetValue(a, false);
   object obj4 = ((RtFieldInfo) fields[i]).InternalGetValue(obj, false);
   if (obj3 == null && obj4 != null)
   return false;
   else if (!obj3.Equals(obj4))
   return false;
  }
  return true;
}
```

In brief, if `CanCompareBits` returns true, the `FastEqualsCheck` is responsible for checking equality; otherwise, the method enters a Reflection-based loop where fields are fetched using the `FieldInfo` class and compared recursively by invoking `Equals`. Needless to say, the Reflection-based loop is where the performance battle is conceded completely; Reflection is an extremely expensive mechanism, and everything else pales compared to it. The definition of `CanCompareBits` and `FastEqualsCheck` is deferred to the CLR – they are "internal calls", not implemented in IL – so we can't disassemble them easily. However, from experimentation we discovered that `CanCompareBits` returns true if either of the following conditions hold:

1. The value type contains only primitive types and does not override `Equals`
2. The value type contains only value types for which (1) holds and does not override `Equals`
3. The value type contains only value types for which (2) holds and does not override `Equals`

The `FastEqualsCheck` method is similarly a mystery, but effectively it executes a `memcmp` operation – comparing the memory of both value type instances (byte-by-byte). Unfortunately, both of these methods remain an internal implementation detail, and relying on them as a high-performance way to compare instances of your value types is an extremely bad idea.

The GetHashCode Method

The final method that remains and is important to override is `GetHashCode`. Before we can show a suitable implementation, let's brush up our knowledge on what it is used for. Hash codes are used most often in conjunction with hash tables, a data structure that (under certain conditions) allows constant-time (O(1)) insertion, lookup, and deletion operations on arbitrary data. The common hash table classes in the .NET Framework include `Dictionary<TKey,TValue>`, `Hashtable`, and `HashSet<T>`. A typical hash table implementation consists of a dynamic-length array of buckets, each bucket containing a linked list of items. To place an item in the hash table, it first calculates a numeric value (by using the `GetHashCode` method), and then applies to it a hash function that specifies to which bucket the item is mapped. The item is inserted into its bucket's linked list.

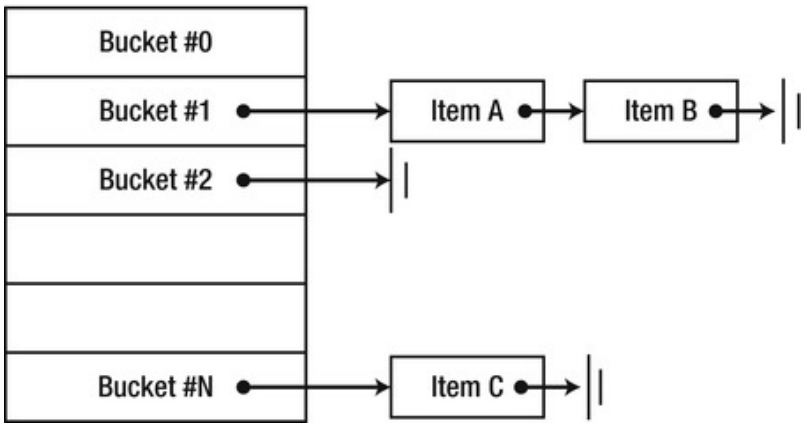

***Figure 3-10**. A hash table, consisting of an array of linked lists (buckets) in which items are stored. Some buckets may be empty; other buckets may contain a considerable number of items*

The performance guarantees of hash tables rely strongly upon the hash function used by the hash table implementation, but require also several properties from the `GetHashCode` method:

1. If two objects are equal, their hash codes are equal.

2. If two objects are not equal, it should be unlikely that their hash codes are equal.
3. `GetHashCode` should be fast (although often it is linear in the size of the object).
4. An object's hash code should not change.

Caution Property (2) cannot state "if two objects are not equal, their hash codes are not equal" because of the pigeonhole principle: there can be types of which there are many more objects than there are integers, so unavoidably there will be many objects with the same hash code. Consider `long`s, for example; there are 2^{64} different `long`s, but only 2^{32} different integers, so there will be at least one integer value which is the hash code of 2^{32} different `long`s!

Formally, property (2) can be stated as follows to require a uniform distribution of hash codes: set an object *A*, and let *S(A)* be the set of all the objects *B* such that:

- *B* is not equal to *A;*
- The hash code of *B* is equal to the hash code of *A*.

Property (2) requires that the size of *S(A)* is roughly the same for every object *A*. (This assumes that the probability to see every object is the same – which is not necessarily true for actual types.)

Properties (1) and (2) emphasize the relationship between object equality and hash code equality. If we went to the trouble of overriding and overloading the virtual Equals method, there is no choice but to make sure that the `GetHashCode` implementation is aligned with it as well. It would seem that a typical implementation of `GetHashCode` would rely somehow on the object's fields. For example, a good implementation of `GetHashCode` for an `int` is simply to return the integer's value. For a `Point2D` object, we might consider some linear combination of the two coordinates, or combine some of the bits of the first coordinate with some other bits of the second coordinate. Designing a good hash code in general is a very difficult undertaking, which is beyond the scope of this book.

Lastly, consider property (4). The reasoning behind it is as follows: suppose you have the point (5, 5) and embed it in a hash table, and further suppose that its hash code was 10. If you modify the point to (6, 6) – and its hash code is also modified to 12 – then you will not be able to find in the hash table the point you inserted to it. But this is not a concern with value types, because you *cannot* modify the object you inserted into the hash table – the hash table stores a *copy* of it, which is inaccessible to your code.

What of reference types? With reference types, content-based equality becomes a concern. Suppose that we had the following implementation of `Employee.GetHashCode`:

```
public class Employee
{
  public string Name { get; set; }
  public override int GetHashCode()
  {
   return Name.GetHashCode();
  }
}
```

This seems like a good idea; the hash code is based on the object's contents, and we are utilizing `String.GetHashCode` so that we don't have to implement a good hash code function for strings. However, consider what happens when we use an `Employee` object and change its name after it was inserted into the hash table:

```
HashSet<Employee> employees = new HashSet<Employee>();
Employee kate = new Employee { Name = "Kate Jones" };
employees.Add(kate);
kate.Name = "Kate Jones-Smith";
employees.Contains(kate); //returns false!
```

The object's hash code has changed because its contents have changed, and we can no longer find the object in the hash table. This is somewhat expected, perhaps, but the problem is that now we can't *remove* Kate from the hash table at all, even though we have access to the original object!

The CLR provides a default `GetHashCode` implementation for reference types that rely on the object's identity as their equality criterion. If two object references are equal if and only if they reference the same object, it would make sense to store the hash code somewhere in the object itself, such that it is never modified and is easily accessible. Indeed, when a reference type instance is created, the CLR can embed its hash code in the object heard word (as an optimization, this is done only the first time the hash code is accessed; after all, many objects are never used as hash table keys). To calculate the hash code, there's no need to rely on a random number generation or consider the object's contents; a simple counter would do.

Note How can the hash code coexist in the object header word alongside the sync block index? As you recall, most objects never use their object header word to store a sync block index, because they are not used for synchronization. In the rare case that an object is linked to a sync block through the object header word storing its index, the hash code is copied to the sync block and stored there until the sync block is detached from the object. To determine whether the hash code or the sync block index is currently stored in the object header word, one of the field's bits is used as a marker.

Reference types using the default `Equals` and `GetHashCode` implementation need not concern themselves with any of the four properties stressed above – they get them for free. However, if your reference type should choose to override the default equality behavior (this is what `System.String` does, for example), then you should consider making your reference type immutable if you use it as a key in a hash table.