Chapter 10

Yashica Mat 124G

Amsterdam Bridge

Coding For Collections

Learning Objectives

- *Enable user-defined data types to perform correctly in collections*
- *Define the term "natural ordering"*
- *State the difference between natural ordering and custom ordering*
- *Create classes and structures that can be used in equality comparisons*
- *Override System.Object.Equals() and System.Object.GetHashCode() methods*
- *Implement the IComparable and IComparable<T> interfaces to specify natural ordering*
- *Implement the IComparer and IComparer<T> interfaces to create a custom comparer*
- *Implement the IEquatable interface to allow objects to be used as keys*
- *Define the term "immutable" object*

Introduction

When creating user-defined data types you must stop for a moment to consider how they will be used in your program. If you intend to use them in collections then you must enable them to be used in equality and comparison operations. For example, if you intend to sort user-defined objects using the Array.Sort() method, then you must provide the ability for one object to be compared with another for the sort operation to work correctly. If you intend to use user-defined data types as keys in hashtables, dictionaries, or other keyed collections, then you'll need to know how to get your objects to behave correctly as keys. These topics are the focus of this chapter.

I'll start by showing you how to override the Object.Equals() and Object.GetHashCode() methods. I'll then explain why and how to overload the == and != operators.

Next I'll talk about comparison operations and show you how to specify *natural ordering* by implementing the IComparable and IComparable<T> interfaces. Following this I'll show you how to create individual comparer objects that are used to specify *custom ordering* by implementing the IComparer and IComparer<T> interfaces.

I wrap up the chapter by showing you how to create objects that can be used as keys in hashtables, dictionaries, and other keyed collections. This includes a discussion of *object immutability.*

Upon completing this chapter you'll have a thorough understanding of how to create user-defined types that behave well when used in collections. Now, let's get going!

Coding for Equality Operations

Objects of a particular type, when used in non-keyed collections like arrays and lists, must be able to be used in equality comparison operations. This section discusses the differences between *reference equality*, *value equality*, and *bitwise equality*, and shows you how to override the Object.Equals() and Object.GetHashCode() methods. Following this I'll show you how to overload the == and != operators.

Reference Equality vs. Value Equality

Normally, when you compare two reference objects for equality like this...

```
o1 == o2
```

...you are comparing their addresses. In other words, if o1 and o2 refer to the same location in memory then they must be equal because they refer to the same object. However, it's not always desirable to use an object's address as a basis for equality. Take strings for example. Two strings of equal value may be different objects as the following code snippet suggests:

```
String s1 = "Hello";
String s2 = "Hello";
```

The expression (`s1 == s2`) will yield true just as `s1.Equals(s2)` will yield true. This is because the Equals() method has been overridden and the == operator has been overloaded to perform a *value* or string content comparison, which is what you'd expect when comparing two strings.

For structures, the default behavior of the Object.Equals() method and the == operator is *bitwise equality*. For the most part, bitwise equality means the same thing as value equality, especially in the case of simple value types. (i.e., structures like Int32) If, however, the binary representation of the value type is complex, like the Decimal structure, then the Object.Equals() method is overridden and the == operator is overloaded to yield the expected value comparison behavior. For example, given two integer variables:

```
int i = 1;
int j = 2;
```

The expression (`i == j`) compares the value of i, which is 1, against the value of j, which is 2. In either case you can substitute the == operator with the Equals() method like so:

```
i.Equals(j);
```


Overriding Object.Equals() and Object.GetHashCode()

If the default behavior of the Object.Equals() method is insufficient for your user-defined data types, you'll need to override it and provide a custom implementation. Both the Object.Equals() and Object.GetHashCode() methods must be overridden together to ensure correct behavior. The following sections present the rules that should be followed when overriding these methods.

Rules For Overriding The Object.Equals() Method

When overriding the Object.Equals() method, you must ensure that it subscribes to the expected behavior as specified in the .NET Framework documentation. Table 10.1 lists the required behavior of an overridden Object.Equals() method. (**Note:** The overloaded == operator must work the same way!)

Should be...	Rule	Comment
Reflexive	x.Equals(x) returns true	Exception: floating-point types
Symmetric	x.Equals(y) returns the same as y.Equals(x)	
Transitive	(x.Equals(y) && y.Equals(z)) returns true if and only if x.Equals(z) returns true	
Consistent	Successive calls to x.Equals(y) return the same value as long as the objects referenced by x and y remain unchanged.	
	x.Equals(null) returns false	Or a null reference
	x.Equals(y) returns true if both x and y are NaN	NaN means Not a Number
	Calls to Object.Equals() must not throw exceptions.	No exceptions!
	Override the Object.GetHashCode() method.	If you override the Object.Equals() method.

Table 10-1: Rules for Overriding Object.Equals() method

Rules For Overriding The Object.GetHashCode() Method

When you override the Object.Equals() method you should also override the Object.GetHashCode() method to ensure proper object behavior. This section presents two approaches to implementing a suitable GetHashCode() method. Now, don't be alarmed when I reference two very good Java books. The techniques used to create a suitable hashcode algorithm apply equally to C# as well as Java.

The GetHashCode() method returns an integer which is referred to as the object's *hash value*. The default implementation of GetHashCode() found in the Object class will, in most cases, return a unique hash value for each distinct object even if they are logically equivalent. In most cases this default behavior is acceptable, however, if you intend to use a class of objects as keys to hashtables or other hash-based data structures, then you must override the GetHashCode() method and obey the general contract as specified in the .NET Framework API documentation. The general contract for the GetHashCode() is given in Table 10-2.

Check	Criterion
	The GetHashCode() method must consistently return the same integer when invoked on the same object more than once during an execution of a C# or .NET application, provided no information used in Equals() comparisons on the object is modified. This integer need not remain constant from one execution of an application to another execution of the same application.

Table 10-2: The GetHashCode() General Contract

Check	Criterion
	The GetHashCode() method must produce the same results when called on two objects if they are equal according to the Equals() method.
	The GetHashCode() method is not required to return distinct integer results for logically unequal objects, however, failure to do so may result in degraded hash table performance.

Table 10-2: The GetHashCode() General Contract

As you can see from Table 10-2 there is a close relationship between the Object.Equals() and Object.GetHashCode() methods. It is recommended that any fields used in the Equals() method comparison be used to calculate an object's hash code. Remember, the primary goal when implementing a GetHashCode() method is to have it return the same value consistently for logically equal objects. It would also be nice if the GetHashCode() method returned distinct hash code values for logically unequal objects, but according to the general contract this is not a strict requirement.

Before actually implementing a GetHashCode() method, I want to provide you with two hash code generation algorithms. These algorithms come from two excellent Java references. (Yes, I meant to say Java.) I have changed the text to reflect the .NET method names Object.Equals() and Object.GetHashCode() respectively, and have converted Java operations into compatible C# .NET operations.

Bloch's Hash Code Generation Algorithm

Joshua Bloch, in his book *Effective Java™ Programming Language Guide*, provides the following algorithm for calculating a hash code:

1. Start by storing a constant, nonzero value in an `int` variable called `result`. (Josh used the value 17)
2. For each significant field *f* in your object (each field involved in the Equals() comparison) do the following:
   a. Compute an `int` hash code `c` for the field:
      i. If the field is boolean (bool) compute: `(f?0:1)`
      ii. If the field is a byte, char, short, or int, compute: `(int)f`
      iii. If the field is a long compute: `(unsigned)(f^(f >> 32))`
      iv. If the field is a float compute: `Convert.ToInt32(f)`
      v. If the field is a double compute: `Convert.ToInt64(f)`, and then hash the resulting long according to step 2.a.iii.
      vi. If the field is an object reference and this class's Equals() method compares the field by recursively invoking Equals(), recursively invoke GetHashCode() on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke GetHashCode() on the canonical representation. If the value of the field is null, return 0.
      vii. If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values in step 2.b
   b. Combine the hash code `c` computed in step a into `result` as follows:
      `result = 37*result + c;`
3. Return `result`.
4. If equal object instances do not have equal hash codes fix the problem!

Ashmore's Hash Code Generation Algorithm

Derek Ashmore, in his book *The J2EE Architect's Handbook: How To Be A Successful Technical Architect For J2EE Applications*, recommends the following simplified hash code algorithm:

1. Concatenate the required fields (those involved in the Equals() comparison) into a string.
2. Call the GetHashCode() method on that string.
3. Return the resulting hash code value.

An Example: The Person Class

I'll use a class named Person to demonstrate how to override the Object.Equals() and Object.GetHashCode() methods. Example 10.1 lists the code for the Person class.

10.1 Person.cs (Overridden Equals() and GetHashCode() Methods)

```
using System;

public class Person {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;

  public Person(){}

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  public String FirstName {
    get { return _firstName; }
    set { _firstName = value; }
  }

  public String MiddleName {
    get { return _middleName; }
    set { _middleName = value; }
  }

  public String LastName {
    get { return _lastName; }
    set { _lastName = value; }
  }

  public Sex Gender {
    get { return _gender; }
    set { _gender = value; }
  }

  public DateTime Birthday {
    get { return _birthday; }
    set { _birthday = value; }
  }

  public Guid DNA {
    get { return _dna; }
```

```
75      set { _dna = value; }
76    }
77
78    public int Age {
79      get {
80        int years = DateTime.Now.Year - _birthday.Year;
81        int adjustment = 0;
82        if(DateTime.Now.Month < _birthday.Month){
83          adjustment = 1;
84      } else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
85               adjustment = 1;
86             }
87        return years - adjustment;
88      }
89    }
90
91    public String FullName {
92      get { return FirstName + " " + MiddleName + " " + LastName; }
93    }
94
95    public String FullNameAndAge {
96      get { return FullName + " " + Age; }
97    }
98
99    public override String ToString(){
100     return (FullName + "  " + Gender + "  " + Age + " " + DNA);
101   }
102
103   public override bool Equals(object o){
104     if(o == null) return false;
105     if(typeof(Person) != o.GetType()) return false;
106     return this.ToString().Equals(o.ToString());
107   }
108
109   public override int GetHashCode(){
110     return this.ToString().GetHashCode();
111   }
112
113 } // end Person class
```

Referring to example 10.1 — the Person class defines the usual fields you'd expect for a data type of this nature. I've also added a field called _dna of type Guid (Globally Unique Identifier). (I know, I'm being cheeky here calling the field _dna. In real life, the name of this field might be _id which would map to the primary key column of a relational database table where state values of person objects are persisted.) I've added the _dna field with its corresponding Guid type to make it easier to make Person objects unique.

The overridden Object.ToString() method is defined on line 99. It returns a concatenation of the FullName, Gender, Age, and DNA properties. (The Age property is an example of a calculated read-only property.) The overridden Object.Equals() method starts on line 103. It relies on the ToString() method to compare different person objects for value equality. The GetHashCode() method simply calls the GetHashCode() method on the string generated by the Person object's ToString() method.

Example 10.2 gives the code for a short application that creates a few Person objects and tests the Object.Equals() method, validating its conformance to the rules laid out in table 10-1.

10.2 MainApp.cs (Demonstrating Overridden Equals() & GetHashCode() Methods)

```
1   using System;
2
3   public class MainApp {
4     public static void Main(){
5       Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
6          new DateTime(1961, 2, 3), Guid.NewGuid());
7       Console.WriteLine("p1.Equals(p1) : {0}", p1.Equals(p1));
8       Console.WriteLine("p1.Equals(string) : {0}", p1.Equals("Hello!"));
9       Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
10         new DateTime(1972, 1, 1), Guid.NewGuid());
11      Console.WriteLine("p1.Equals(p2) : {0}", p1.Equals(p2));
12      Console.WriteLine("p2.Equals(p1) : {0}", p2.Equals(p1));
13      Console.WriteLine("p1.GetHashCode() = {0}", p1.GetHashCode());
14      Console.WriteLine("p2.GetHashCode() = {0}", p2.GetHashCode());
15    }
16  }
```

Referring to example 10.2 — On line 5 a Person reference named p1 is created and initialized. The Object.Equals() method is then called using the reference p1 as an argument. This of course should return true. Next, p1 is compared with a string object, which should return false. On line 9 a second Person reference named p2 is declared and initialized and it's compared with p1. Both tests should return false. Following this, the GetHashCode()

method is called on each reference. The values returned by these last two method calls will yield different values when you run this program on your computer. Figure 10-1 shows the results of running this program.

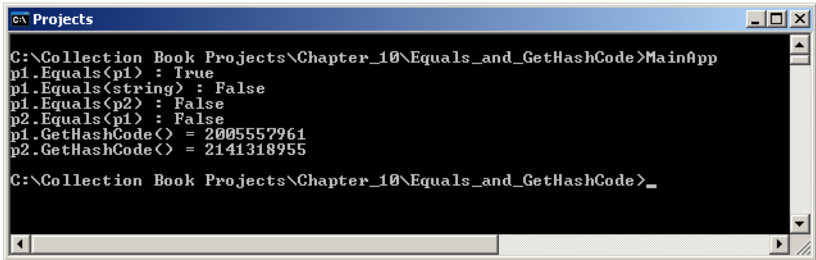

Figure 10-1: Results of Running Example 10.2

Overloading the == and != Operators

Although not strictly required to be overloaded for the purposes of collections, the == and != operators can be overloaded with little effort because they can simply use the overridden Object.Equals() method in their implementation. (Low hanging fruit!) Example 10.3 gives the modified Person class with the overloaded == and != operators.

10.3 Person.cs (Overloaded == and != Operators)

```
using System;

public class Person {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;



  public Person(){}

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday, Guid dna){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = dna;
  }

  public Person(String firstName, String middleName, String lastName,
                Sex gender, DateTime birthday){
     FirstName = firstName;
     MiddleName = middleName;
     LastName = lastName;
     Gender = gender;
     Birthday = birthday;
     DNA = Guid.NewGuid();
  }

  public Person(Person p){
     FirstName = p.FirstName;
     MiddleName = p.MiddleName;
     LastName = p.LastName;
     Gender = p.Gender;
     Birthday = p.Birthday;
     DNA = p.DNA;
  }

  // public properties
  public String FirstName {
```



```
51       get { return _firstName; }
52       set { _firstName = value; }
53     }
54
55     public String MiddleName {
56       get { return _middleName; }
57       set { _middleName = value; }
58     }
59
60     public String LastName {
61       get { return _lastName; }
62       set { _lastName = value; }
63     }
64
65     public Sex Gender {
66       get { return _gender; }
67       set { _gender = value; }
68     }
69
70     public DateTime Birthday {
71       get { return _birthday; }
72       set { _birthday = value; }
73     }
74
75     public Guid DNA {
76       get { return _dna; }
77       set { _dna = value; }
78     }
79
80     public int Age {
81       get {
82      int years = DateTime.Now.Year - _birthday.Year;
83         int adjustment = 0;
84      if(DateTime.Now.Month < _birthday.Month){
85            adjustment = 1;
86         } else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
87                  adjustment = 1;
88                }
89         return years - adjustment;
90       }
91     }
92
93     public String FullName {
94       get { return FirstName + " " + MiddleName + " " + LastName; }
95     }
96
97     public String FullNameAndAge {
98       get { return FullName + " " + Age; }
99     }
100
101    public override String ToString(){
102      return (FullName + "  " + Gender + "  " + Age + " " + DNA);
103    }
104
105    public override bool Equals(object o){
106      if(o == null) return false;
107      if(typeof(Person) != o.GetType()) return false;
108      return this.ToString().Equals(o.ToString());
109    }
110
111    public override int GetHashCode(){
112      return this.ToString().GetHashCode();
113    }
114
115    public static bool operator ==(Person lhs, Person rhs){
116      return lhs.Equals(rhs);
117    }
118
119    public static bool operator !=(Person lhs, Person rhs){
120      return !(lhs.Equals(rhs));
121    }
122
123 } // end Person class
```

Referring to example 10.3 — the == operator is overloaded on line 115. Note that it's a static method and that it defines two method parameters of type Person named lhs (left hand side) and rhs (right hand side). It simply calls the overridden Object.Equals() method to make the equality check. It can do this because the rules for overloading the == operator are the same as the rules for overriding the Object.Equals() method, so each must exhibit the same behavior.

The != operator is defined on line 119. It too relies on the overridden Object.Equals() method in its implementation. Note that it simply negates the result of comparing the lhs with the rhs with the Equals() method.

Example 10.4 demonstrates the use of the overloaded == and != operators.

10.4 MainApp.cs (Demonstrating Overloaded == and != Operators)

```
1    using System;
2
3    public class MainApp {
4      public static void Main(){
5        Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
6           new DateTime(1961, 2, 3), Guid.NewGuid());
7        Console.WriteLine("p1.Equals(p1) : {0}", p1.Equals(p1));
8        Console.WriteLine("p1.Equals(string) : {0}", p1.Equals("Hello!"));
9        Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
10          new DateTime(1972, 1, 1), Guid.NewGuid());
11       Console.WriteLine("p1.Equals(p2) : {0}", p1.Equals(p2));
12       Console.WriteLine("p2.Equals(p1) : {0}", p2.Equals(p1));
13       Console.WriteLine("p1.GetHashCode() = {0}", p1.GetHashCode());
14       Console.WriteLine("p2.GetHashCode() = {0}", p2.GetHashCode());
15       Console.WriteLine("p1 == p1 : {0}", p1 == p1);
16       Console.WriteLine("p1 == p2 : {0}", p1 == p2);
17       Console.WriteLine("p1 != p1 : {0}", p1 != p1);
18       Console.WriteLine("p1 != p2 : {0}", p1 != p2);
19     }
20   }
```

Referring to example 10.4 — the tests of the == and != operators have been added to the previous MainApp example. On line 15 the reference p1 is compared with itself using the == operator and again on line 17 using the != operator. These comparisons result in the compiler warnings shown in figure 10-2. You can safely ignore them here for the sake of testing. Figure 10-3 shows the results of running this program.

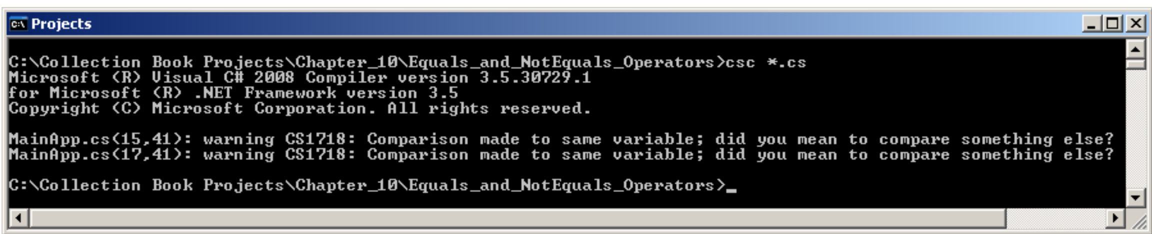

Figure 10-2: Compiler Warning Generated when Compiling Examples 10.3 and 10.4

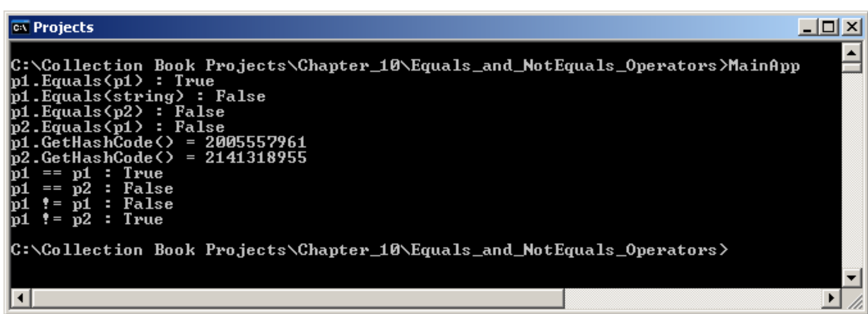

Figure 10-3: Results of Running Example 10.4

Quick Review

The first step in getting your user-defined types to behave well in collections is to override the Object.Equals() and Object.GetHashCode() methods. Make sure you adhere to the Object.Equals() method behavior rules. You can optionally overload the == and != methods as their behavior can be easily implemented in terms of the Object.Equals() method.

The overridden Object.GetHashCode() method can be easily implemented by calling the GetHashCode() method on the string returned by the object's overridden ToString() method.

Coding for Comparison Operations

If you intend to insert user-defined objects into a collection and sort them you'll need to define how, exactly, one object is to be compared with another in terms of being *less than*, *equal to*, or *greater than* another object. You do this by implementing either the *IComparable* or the *IComparable<T>* interfaces, or both if you plan to use user-defined objects in both non-generic and generic collections. In this section I explain the concept of natural ordering and show you how to implement each of these interfaces.

Natural Ordering

When you implement the IComparable and IComparable<T> interfaces in a class or structure you are specifying what is referred to as a *natural ordering* for that particular type. It's called natural ordering because you have instructed the type how to behave when compared with other objects of the same (or different) type.

Take integers for example. If you examine the .NET documentation for the Int32 structure you'll see that it implements both the IComparable and IComparable<T> (as IComparable<int>) interfaces. This allows integers to be compared with other integers when sorted with the Sort() method defined by the Array class and other collections that allow elements to be sorted.

IComparable and IComparable<T> Interfaces

The IComparable and IComparable<T> interfaces each declare one method named CompareTo(object other) that returns an integer, the value of which must reflect the results of the comparison as listed in the rules shown in table 10-3.

Return Value	Returned When...
Less than Zero (-1)	This object is less than the *other* parameter
Zero (0)	This object is equal to the *other* parameter
Greater than Zero (1)	This object is greater than the *other* parameter, or, the *other* parameter is null

Table 10-3: Rules For Implementing IComparable.CompareTo() Method

Referring to table 10-3 — as the rules state, if the object (represented by the `this` reference) is less than the `other` parameter, the CompareTo() method returns some value less than 0. (The value -1 is fine.) If both objects being compared are equal it returns 0, and if the `other` object is greater or *null* it returns a positive number. (1 is fine.) Example 10.5 shows how the IComparable and IComparable<T> interfaces can be implemented in the Person class.

10.5 Person.cs (Implementing IComparable and IComparable<T> Interfaces)

```
using System;

public class Person : IComparable, IComparable<Person> {

  //enumeration
  public enum Sex {MALE, FEMALE};

  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;



  public Person(){}

```



```
   public Person(String firstName, String middleName, String lastName,
                 Sex gender, DateTime birthday, Guid dna){
      FirstName = firstName;
      MiddleName = middleName;
      LastName = lastName;
      Gender = gender;
      Birthday = birthday;
      DNA = dna;
   }

   public Person(String firstName, String middleName, String lastName,
                 Sex gender, DateTime birthday){
      FirstName = firstName;
      MiddleName = middleName;
      LastName = lastName;
      Gender = gender;
      Birthday = birthday;
      DNA = Guid.NewGuid();
   }

   public Person(Person p){
      FirstName = p.FirstName;
      MiddleName = p.MiddleName;
      LastName = p.LastName;
      Gender = p.Gender;
      Birthday = p.Birthday;
      DNA = p.DNA;
   }

   // public properties
   public String FirstName {
     get { return _firstName; }
     set { _firstName = value; }
   }

   public String MiddleName {
     get { return _middleName; }
     set { _middleName = value; }
   }

   public String LastName {
     get { return _lastName; }
     set { _lastName = value; }
   }

   public Sex Gender {
     get { return _gender; }
     set { _gender = value; }
   }

   public DateTime Birthday {
     get { return _birthday; }
     set { _birthday = value; }
   }

   public Guid DNA {
     get { return _dna; }
     set { _dna = value; }
   }

   public int Age {
  get {
    int years = DateTime.Now.Year - _birthday.Year;
    int adjustment = 0;
    if(DateTime.Now.Month < _birthday.Month){
          adjustment = 1;
    }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
      adjustment = 1;
    }
    return years - adjustment;
  }
   }

   public String FullName {
     get { return FirstName + " " + MiddleName + " " + LastName; }
   }

   public String FullNameAndAge {
     get { return FullName + " " + Age; }
   }

```



```
101   protected String SortableName {
102     get { return LastName + FirstName + MiddleName; }
103   }
104
105   public override String ToString(){
106     return (FullName + "  " + Gender + "  " + Age + " " + DNA);
107   }
108
109   public override bool Equals(object o){
110     if(o == null) return false;
111     if(typeof(Person) != o.GetType()) return false;
112     return this.ToString().Equals(o.ToString());
113   }
114
115   public override int GetHashCode(){
116     return this.ToString().GetHashCode();
117   }
118
119   public static bool operator ==(Person lhs, Person rhs){
120     return lhs.Equals(rhs);
121   }
122
123   public static bool operator !=(Person lhs, Person rhs){
124     return !(lhs.Equals(rhs));
125   }
126
127   public int CompareTo(object obj){
128     if((obj == null) || (typeof(Person) != obj.GetType()))  {
129       throw new ArgumentException("Object is not a Person!");
130     }
131     return this.SortableName.CompareTo(((Person)obj).SortableName);
132   }
133
134   public int CompareTo(Person p){
135     if(p == null){
136       throw new ArgumentException("Cannot compare null objects!");
137     }
138     return this.SortableName.CompareTo(p.SortableName);
139   }
140
141 } // end Person class
```

Referring to example 10.5 — on line 3 the IComparable and IComparable<T> interfaces are listed as being implemented by the Person class. Note how the IComparable<T> interface actually reads IComparable<*Person*>. The non-generic CompareTo() method begins on line 127. This version of the method corresponds with the IComparable interface. It takes an object argument and must test it to see if it's the proper type. If it's not, or it's *null*, it throws an ArgumentException.

The CompareTo() method on line 134 corresponds to the IComparable<Person> interface. Note that since the type of parameter has been specified, it's no longer necessary to explicitly test the incoming object for type conformance, as this is handled by the compiler.

Also important to note here is how I've defined natural ordering for Person objects. I've chosen to order Person object's by last names, first names, and middle names. To help in this effort I have added another property to the Person class named SortableName which concatenates the name fields together for proper sorting.

Example 10.6 demonstrates how an array of Person objects can now be sorted by name.

10.6 MainApp.cs (Sorting and Array of Person Objects with Natural Ordering)

```
using System;

public class MainApp {
  public static void Main(){
    Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
       new DateTime(1961, 2, 3), Guid.NewGuid());
    Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
       new DateTime(1972, 1, 1), Guid.NewGuid());
    Person p3 = new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE,
       new DateTime(1959, 8, 8), Guid.NewGuid());
    Person p4 = new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE,
       new DateTime(1970, 5, 6), Guid.NewGuid());
    Person p5 = new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE,
       new DateTime(1983, 2, 3), Guid.NewGuid());
    Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE,
       new DateTime(1986, 10, 15), Guid.NewGuid());

    Person[] people_array = new Person[6];
    people_array[0] = p1;
    people_array[1] = p2;
```



```
21      people_array[2] = p3;
22      people_array[3] = p4;
23      people_array[4] = p5;
24      people_array[5] = p6;
25
26      Console.WriteLine("-------- Before Sorting ------------");
27
28      foreach(Person p in people_array){
29        Console.WriteLine(p.LastName + "," + p.FirstName);
30      }
31
32      Array.Sort(people_array);
33
34      Console.WriteLine("-------- After Sorting ------------");
35
36      foreach(Person p in people_array){
37        Console.WriteLine(p.LastName + "," + p.FirstName);
38      }
39    }
40  }
```

Referring to example 10.6 — the six Person objects created on lines 5 through 16 are used to initialize the six elements of the people_array on lines 19 through 24. The `foreach` statement on line 28 prints out the contents of the array to the console before sorting. The `foreach` statement on line 36 does the same after the array has been sorted. The Array.Sort() method called on line 32 expects the elements in the array passed to it as an argument to implement IComparable. If one or more elements in the array fail to implement IComparable, the Sort() method will throw an InvalidOperationException. Figure 10-4 shows the results of running this program.

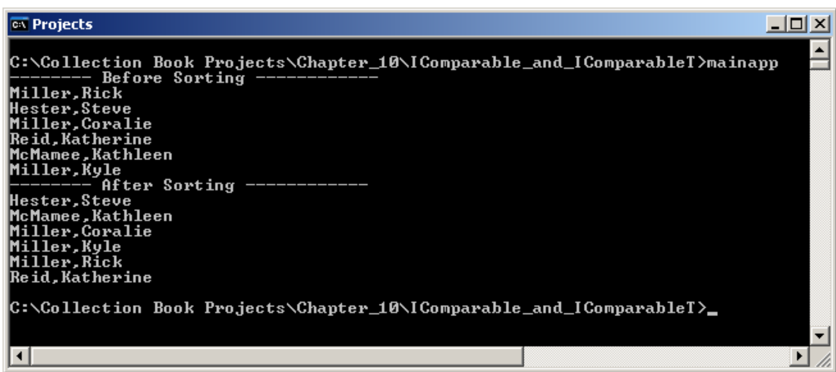

Figure 10-4: Results of Running Example 10.6

Custom Ordering: Creating Separate Comparer Objects

As you learned in the preceding section, to specify a natural ordering for your user-defined types you must implement the IComparable and IComparable<T> interfaces. If you want to order objects in a different way, you can create custom comparers by implementing the IComparer and IComparer<T> interfaces.

IComparer and IComparer<T> Interfaces

The IComparer and IComparer<T> interfaces both declare one method named Compare(). In the case of IComparer the method signature is *int Compare(object x, object y)* and for IComparer<T> it's *int Compare(T x, T y)*. The rules for implementing the Compare() methods are the same ones used to implement the CompareTo() methods discussed in the previous section.

These methods are easy to implement. In most cases, custom ordering boils down to one particular field within the user-defined type. For example, if you want to provide a custom ordering of Person objects by age, you would simply be comparing two integers: one person object's age against another's. And since all the built-in .NET types already implement the IComparable and IComparable<T> interfaces, you can implement the Compare() method in terms of each object's CompareTo() method.

An Example: PersonAgeComparer

Example 10.7 gives the code for a class named PersonAgeComparer. The PersonAgeComparer class implements both the IComparer and IComparer<T> interfaces.

10.7 PersonAgeComparer.cs

```
1    using System;
2    using System.Collections;
3    using System.Collections.Generic;
4
5    public class PersonAgeComparer : IComparer, IComparer<Person> {
6
7      public int Compare(object x, object y){
8        if((x == null) || (y == null) || (typeof(Person) != x.GetType())
9            || (typeof(Person) != y.GetType())){
10         throw new ArgumentException("Both objects must be of type Person!");
11       }
12
13       return ((Person)x).Age.CompareTo(((Person)y).Age);
14     }
15
16     public int Compare(Person x, Person y){
17      if((x == null) || (y == null)){
18         throw new ArgumentException("Both objects must be of type Person!");
19       }
20
21       return x.Age.CompareTo(y.Age);
22     }
23
24   }
```

Referring to example 10.7 — the non-generic Compare() method starts on line 7. The `if` statement on line 8 checks to ensure incoming arguments are valid Person objects. If the arguments fail this test the method throws an ArgumentException. Line 13 contains the meat of the method: It casts each parameter to type Person and calls the CompareTo() method via the x parameter passing the y parameter as an argument. Done!

The generic version of the Compare() method on line 16 safely skips the type testing part of the `if` statement since the method parameters already specify the type. If the arguments are *null* it throws an ArgumentException, otherwise, the comparison of the x parameter with the y parameter proceeds without the casting as was necessary in the non-generic version of the Compare() method.

Example 10.8 demonstrates the use of the PersonAgeComparer class.

10.8 MainApp.cs (Demonstrating Custom Ordering with PersonAgeComparer)

```
using System;


public class MainApp {
  public static void Main(){
    Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
       new DateTime(1961, 2, 3), Guid.NewGuid());
    Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
       new DateTime(1972, 1, 1), Guid.NewGuid());
    Person p3 = new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE,
       new DateTime(1974, 8, 8), Guid.NewGuid());
    Person p4 = new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE,
       new DateTime(1970, 5, 6), Guid.NewGuid());
    Person p5 = new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE,
       new DateTime(1983, 2, 3), Guid.NewGuid());
    Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE,
       new DateTime(1986, 10, 15), Guid.NewGuid());

    Person[] people_array = new Person[6];
    people_array[0] = p1;
    people_array[1] = p2;
    people_array[2] = p3;
    people_array[3] = p4;
    people_array[4] = p5;
    people_array[5] = p6;

    Console.WriteLine("-------- Before Sorting ------------");

    foreach(Person p in people_array){
      Console.WriteLine(p.FullNameAndAge);
    }

    Array.Sort(people_array, new PersonAgeComparer());

    Console.WriteLine("-------- After Sorting ------------");
```



```
36
37       foreach(Person p in people_array){
38         Console.WriteLine(p.FullNameAndAge);
39       }
40     }
41   }
```

Referring to example 10.8 — note on line 33 that a PersonAgeComparer object is passed as the second argument to the Array.Sort() method. If a custom comparer object is supplied to the Array.Sort() method, as is done here, it orders the elements in the array according to the custom comparer. The result in this case is that the elements are sorted by age vs. last, first, and middle names. Figure 10-5 shows the results of running this program.

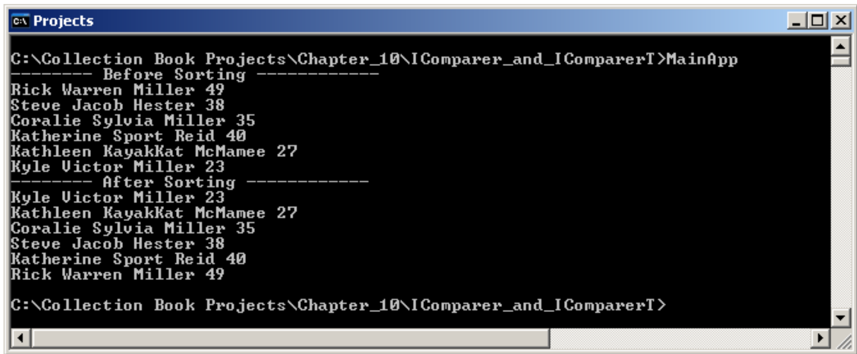

Figure 10-5: Results of Running Example 10.8

Quick Review

Implement both the IComparable and IComparable<T> interfaces to specify a natural ordering for user-defined types. Implement the IComparer and IComparer<T> interfaces to create a custom comparer. Custom comparers are used to specify a custom ordering. You can create as many custom comparers as required.

It's a good idea to always implement both the generic and non-generic versions of these interfaces. Doing so ensures your user-defined types will be sortable in generic and non-generic collections.

Using Objects as Keys

In keyed collections, objects are inserted into the collection in key/value pairs. Object's used as keys must obey certain rules. This section explains those rules and demonstrates how to create a type suitable for the creation of key objects.

Rules For Objects Used As Keys

Objects inserted into keyed collections are located within those collections via an operation performed upon their associated key. In chapter 9 you learned about the Hashtable and Dictionary<T Key, T Value> collections. In these collections, the value's location with the hash table is determined by applying a hash function to the key. Before an object can be used as a key it must adhere to a few rules as listed in table 10-4.

Rule	Comment
Key must be immutable	Objects used as keys must not change value while they are being used as keys. Object's whose state value cannot be changed after they are created are called immutable objects. Strings are immutable objects, which is why they can be safely used as keys.

Table 10-4: Rules For Creating Key Classes

Rule	Comment
Implement the IEquatable<T> interface	The IEquatable<T> interface is used by generic collections to test keys for equality. It defines one method named Equals().
Override the Object.Equals() method	Key objects need to be compared with each other for equality. If you implement IEquatable<T> you should also override the Object.Equals() method for consistency.
Override the Object.GetHashCode() method	Key objects, especially when used as keys in Hashtable and Dictionary<T Key, T Value> collections, must override the GetHashCode() method. You must also override this method if you override Object.Equals() to ensure consistent equality behavior.
Implement IComparable and IComparable<T> interfaces	If you're going to use the keys in sorted collections, the key objects must be sortable. If you don't implement these interfaces you can specify custom ordering by providing a custom comparer object.

Table 10-4: Rules For Creating Key Classes

Object Immutability

An immutable object is one whose state cannot be changed after it has been created. Strings are an example of immutable objects. One simple way to create an immutable type is to make the fields readonly and supply readonly properties. Object state values are set only through constructor methods. Care must also be taken not to return references to contained objects. Example 10.9 demonstrates this strategy.

10.9 MyImmutableType.cs

```
using System;

public class MyImmutableType {
  private readonly string _stringVal;
  private readonly int _intVal;

  public MyImmutableType(string s, int i){
    _stringVal = s;
    _intVal = i;
  }

  public string StringValue {
    get { return string.Copy(_stringVal); }
  }

  public int IntVal {
    get { return _intVal; }
  }

  public override string ToString(){
    return _stringVal + " " + _intVal;
  }
}
```

Referring to example 10.9 — the MyImmutableType class contains two readonly fields: one of type string named _stringVal and one of type int named _intVal. The constructor supplies the only way to set these field values. The StringValue and IntValue properties are readonly properties. (i.e., they only supply `get` operations) Note how the StringValue property returns a copy of the _stringVal field. Example 10.10 shows the MyImmutableType class in action, although there's not much going on!

10.10 MainApp.cs (Demonstrating MyImmutableType)

```
using System;

public class MainApp {
  public static void Main(){
    MyImmutableType mit = new MyImmutableType("An immutable type's state cannot be changed.", 49);
    Console.WriteLine(mit);
  }
} // end Main
```

Figure 10-6 shows the results of running this program.

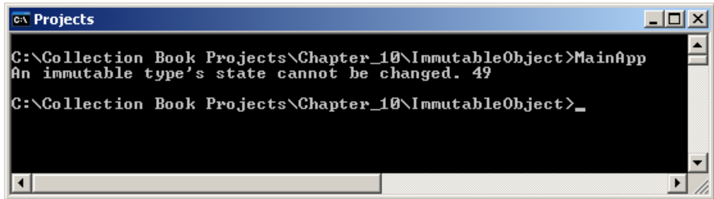

Figure 10-6: Results of Running Example 10.10

Example: PersonKey Class

Example 10.11 gives an extended key class example in the form of the PersonKey class. The PersonKey class implements most of the rules listed in table 10-4. Note that if I wanted to use this key class in sorting operations I would need to implement the IComparable and IComparable<T> interfaces.

10.11 PersonKey.cs

```
using System;

public class PersonKey : IEquatable<String> {

    private readonly string _keyString = String.Empty;

    public PersonKey(string s){
      _keyString = s;
    }

    public bool Equals(string other){
      return _keyString.Equals(other);
    }

    public override string ToString(){
      return String.Copy(_keyString);
    }

    public override bool Equals(object o){
      if(o == null) return false;
      if(typeof(string) != o.GetType()) return false;
      return this.ToString().Equals(o.ToString());
    }

    public override int GetHashCode(){
      return this.ToString().GetHashCode();
    }

}
```

Referring to example 10.11 — the PersonKey class implements the IEquatable<T> interface (as IEquatable<string>). It also overrides the Object.ToString(), Object.Equals() and Object.GetHashCode() methods. It's also immutable, as the only way to set the _keyString field value is via the constructor.

Example 10.12 gives a modified version of the Person class that contains a new Key property of type PersonKey.

10.12 Person.cs (With Key Property)

```
using System;

public class Person : IComparable, IComparable<Person> {

  //enumeration
  public enum Sex {MALE, FEMALE};


  // private instance fields
  private String   _firstName;
  private String   _middleName;
  private String   _lastName;
  private Sex      _gender;
  private DateTime _birthday;
  private Guid _dna;



  public Person(){}

  public Person(String firstName, String middleName, String lastName,
```



```
                 Sex gender, DateTime birthday, Guid dna){
      FirstName = firstName;
      MiddleName = middleName;
      LastName = lastName;
      Gender = gender;
      Birthday = birthday;
      DNA = dna;
   }

   public Person(String firstName, String middleName, String lastName,
                 Sex gender, DateTime birthday){
      FirstName = firstName;
      MiddleName = middleName;
      LastName = lastName;
      Gender = gender;
      Birthday = birthday;
      DNA = Guid.NewGuid();
   }

   public Person(Person p){
      FirstName = p.FirstName;
      MiddleName = p.MiddleName;
      LastName = p.LastName;
      Gender = p.Gender;
      Birthday = p.Birthday;
      DNA = p.DNA;
   }

   // public properties
   public String FirstName {
     get { return _firstName; }
     set { _firstName = value; }
   }

   public String MiddleName {
     get { return _middleName; }
     set { _middleName = value; }
   }

   public String LastName {
     get { return _lastName; }
     set { _lastName = value; }
   }

   public Sex Gender {
     get { return _gender; }
     set { _gender = value; }
   }

   public DateTime Birthday {
     get { return _birthday; }
     set { _birthday = value; }
   }

   public Guid DNA {
     get { return _dna; }
     set { _dna = value; }
   }

   public int Age {
     get {
  int years = DateTime.Now.Year - _birthday.Year;
  int adjustment = 0;
  if(DateTime.Now.Month < _birthday.Month){
     adjustment = 1;
  }else if((DateTime.Now.Month == _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
      adjustment = 1;
  }
  return years - adjustment;
  }
   }

   public String FullName {
     get { return FirstName + " " + MiddleName + " " + LastName; }
   }

   public String FullNameAndAge {
     get { return FullName + " " + Age; }
   }

   protected String SortableName {
```



```
103     get { return LastName + FirstName + MiddleName; }
104   }
105
106   public PersonKey Key {
107     get { return new PersonKey(this.ToString()); }
108   }
109
110   public override String ToString(){
111     return (FullName + "  " + Gender + "  " + Age + " " + DNA);
112   }
113
114   public override bool Equals(object o){
115     if(o == null) return false;
116     if(typeof(Person) != o.GetType()) return false;
117     return this.ToString().Equals(o.ToString());
118   }
119
120   public override int GetHashCode(){
121     return this.ToString().GetHashCode();
122   }
123
124   public static bool operator ==(Person lhs, Person rhs){
125     return lhs.Equals(rhs);
126   }
127
128   public static bool operator !=(Person lhs, Person rhs){
129     return !(lhs.Equals(rhs));
130   }
131
132   public int CompareTo(object obj){
133     if((obj == null) || (typeof(Person) != obj.GetType()))  {
134       throw new ArgumentException("Object is not a Person!");
135     }
136     return this.SortableName.CompareTo(((Person)obj).SortableName);
137   }
138
139   public int CompareTo(Person p){
140     if(p == null){
141       throw new ArgumentException("Cannot compare null objects!");
142     }
143     return this.SortableName.CompareTo(p.SortableName);
144   }
145 } // end Person class
```

Referring to example 10.12 — the Key property is defined on line 106. Note that a new instance of PersonKey is returned each time the Key property is accessed. Example 10.13 demonstrates how Person objects can be inserted into a Dictionary<T Key, T Value> collection with the help of the PersonKey key class.

10.13 MainApp.cs (Demonstrating the use of Person.Key Property with a Dictionary)

```
using System;
using System.Collections.Generic;


public class MainApp {
  public static void Main(){
    Person p1 = new Person("Rick", "Warren", "Miller", Person.Sex.MALE,
        new DateTime(1961, 2, 3), Guid.NewGuid());
    Person p2 = new Person("Steve", "Jacob", "Hester", Person.Sex.MALE,
        new DateTime(1972, 1, 1), Guid.NewGuid());
    Person p3 = new Person("Coralie", "Sylvia", "Miller", Person.Sex.FEMALE,
        new DateTime(1974, 8, 8), Guid.NewGuid());
    Person p4 = new Person("Katherine", "Sport", "Reid", Person.Sex.FEMALE,
        new DateTime(1970, 5, 6), Guid.NewGuid());
    Person p5 = new Person("Kathleen", "KayakKat", "McMamee", Person.Sex.FEMALE,
        new DateTime(1983, 2, 3), Guid.NewGuid());
    Person p6 = new Person("Kyle", "Victor", "Miller", Person.Sex.MALE,
        new DateTime(1986, 10, 15), Guid.NewGuid());

    Dictionary<PersonKey, Person> directory = new Dictionary<PersonKey, Person>();
    directory.Add(p1.Key, p1);
    directory.Add(p2.Key, p2);
    directory.Add(p3.Key, p3);
    directory.Add(p4.Key, p4);
    directory.Add(p5.Key, p5);
    directory.Add(p6.Key, p6);

    foreach(KeyValuePair<PersonKey, Person> kvp in directory){
      Console.WriteLine("Key: {0} Value: {1}", kvp.Key, kvp.Value.FullName);
    }
  }
}
```


Referring to example 10.13 — each of the six person objects created on lines 7 through 18 are inserted into the dictionary using their Key properties. The `foreach` statement on line 28 iterates over the dictionary collection and writes the value of each key and its associated value to the console.

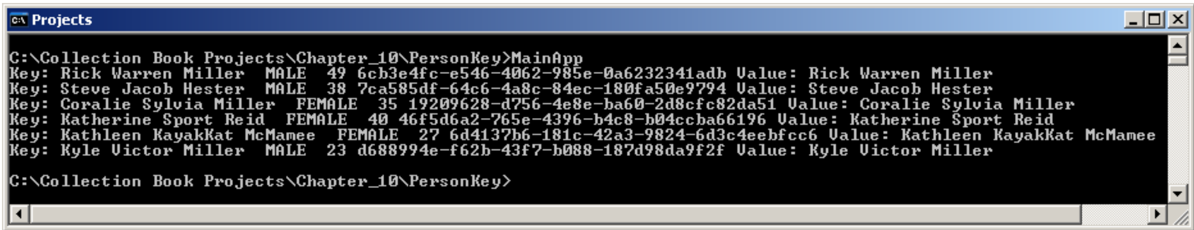

Figure 10-7: Results of Running Example 10.12

Quick Review

If an object is to be used as a key in a collection it must be immutable while it is being used as a key. Immutable object state value cannot be changed after the object is created. Key objects must also implement the IEquatable<T> interface and override the Object.Equals() and Object.GetHashCode() methods. Strings make ideal keys because they implement all the necessary interfaces and are immutable.

Summary

The first step in getting your user-defined types to behave well in collections is to override the Object.Equals() and Object.GetHashCode() methods. Make sure you adhere to the Object.Equals() method behavior rules. You can optionally overload the == and != methods as their behavior can be easily implemented in terms of the Object.Equals() method.

The overridden Object.GetHashCode() method be easily implemented by calling the GetHashCode() method on the string returned by the object's overridden ToString() method.

To specify a natural ordering for user-defined types, implement both the IComparable and IComparable<T> interfaces. To specify a custom ordering, create a custom comparer class by implementing the IComparer and IComparer<T> interfaces. It's a good idea to always implement both the generic and non-generic versions of these interfaces. Doing so ensures your user-defined types will be sortable in generic and non-generic collections.

If an object is to be used as a key in a collection it must be immutable while it is being used as a key. Immutable object state value cannot be changed after the object is created. Key objects must also implement the IEquatable<T> interface and override the Object.Equals() and Object.GetHashCode() methods. Strings make ideal keys because they implement all the necessary interfaces and are immutable.

References

Joshua Bloch. *Effective Java™ Programming Language Guide*. Addison-Wesley, Boston, MA. ISBN: 0-201-31005-8.

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Derek Ashmore. *The J2EE Architect's Handbook: How To Be A Successful Technical Architect For J2EE Applications. DVT Press, Lombard, IL. ISBN: 0972954899*

Notes