## **Plugging in Equality and Order: Custom Comparers**

While a type's default equality or comparison implementation often aligns with its "natural" behavior, there are many scenarios where you need alternative logic. For example:

* You might need a dictionary where string keys are treated case-insensitively (e.g., "apple" is the same as "Apple").
* You might want to sort a list of Customer objects by their postcode, rather than by their name.

For these situations, .NET provides a set of **"plug-in" protocols**. These protocols achieve two main goals:

1. **Alternative Behavior:** They allow you to define and switch in different equality or comparison behaviors as needed.
2. **External Comparability:** They enable you to use a key type in a dictionary or sorted collection even if that type does not intrinsically implement IEquatable<T>, IComparable<T>, or override Equals() and GetHashCode() in a meaningful way for your specific use case.

These plug-in protocols consist of two primary interface pairs:

* **IEqualityComparer<T> and IEqualityComparer:** Used for plug-in equality comparison and hashing. They are recognized by hash-based collections like Dictionary<TKey, TValue> and Hashtable.
* **IComparer<T> and IComparer:** Used for plug-in order comparison. They are recognized by sorted collections (e.g., SortedDictionary<TKey, TValue>, SortedList<TKey, TValue>, SortedSet<T>) and also by static sorting methods like Array.Sort().

Both generic and non-generic forms of these interfaces are available. The IEqualityComparer interfaces also have a default implementation in a class called EqualityComparer<T>.

Additionally, there are interfaces called IStructuralEquatable and IStructuralComparable, which facilitate structural comparisons on composite types like arrays.

### **IEqualityComparer<T> and EqualityComparer<T>**

An **equality comparer** allows you to define and apply non-default equality and hashing behavior, primarily for use with Dictionary<TKey, TValue> and Hashtable. Recall that for a dictionary to work correctly, it needs to answer two fundamental questions about its keys:

1. Are two keys considered "the same"?
2. What is the integer hash code for a given key?

An equality comparer answers these questions by implementing the IEqualityComparer<T> interface (and optionally IEqualityComparer for non-generic compatibility):

| public interface IEqualityComparer<T> {  bool Equals(T x, T y); // Defines how two objects of type T are compared for equality  int GetHashCode(T obj); // Defines how to compute a hash code for an object of type T } |
| --- |

To create a custom comparer, you implement this interface. However, a more convenient approach is to subclass the abstract EqualityComparer<T> class:

| public abstract class EqualityComparer<T> : IEqualityComparer<T>, IEqualityComparer {  // ... default implementations for IEqualityComparer  public abstract bool Equals(T x, T y); // You override this  public abstract int GetHashCode(T obj); // You override this  public static EqualityComparer<T> Default { get; } // Provides a default comparer } |
| --- |

By subclassing EqualityComparer<T>, you only need to override the two abstract methods: Equals(T x, T y) and GetHashCode(T obj). The rules for implementing these methods are the same as for object.Equals() and object.GetHashCode(): if two objects are considered equal by Equals(), they *must* return the same hash code from GetHashCode().

Let's illustrate with a Customer class and a custom comparer that considers two customers equal if their FirstName and LastName match:

| public class Customer {  public string LastName;  public string FirstName;  public Customer(string last, string first) { LastName = last; FirstName = first; } }  public class LastFirstEqComparer : EqualityComparer<Customer> {  public override bool Equals(Customer x, Customer y)  => x.LastName == y.LastName && x.FirstName == y.FirstName;   public override int GetHashCode(Customer obj)  => (obj.LastName + ";" + obj.FirstName).GetHashCode(); // Simple, but effective hash combining } |
| --- |

Without this custom comparer, two distinct Customer objects with the same name would not be considered equal by default Dictionary semantics because Customer is a reference type, and object.Equals() performs reference equality:

| Customer c1 = new Customer("Bloggs", "Joe"); Customer c2 = new Customer("Bloggs", "Joe");  var d = new Dictionary<Customer, string>(); d[c1] = "Joe"; Console.WriteLine(d.ContainsKey(c2)); // Output: False (c1 and c2 are different references) |
| --- |

Now, by providing our custom equality comparer to the Dictionary:

| var eqComparer = new LastFirstEqComparer(); var d = new Dictionary<Customer, string>(eqComparer); // Pass the custom comparer d[c1] = "Joe"; Console.WriteLine(d.ContainsKey(c2)); // Output: True (comparer defines them as equal) |
| --- |

**Important Note:** When using mutable objects as keys in hash-based collections, you must be extremely careful not to change the properties that are used to compute the hash code (FirstName, LastName in our example) while the object is in the collection. Changing them would alter the hash code, making the object unfindable or incorrectly placed within the dictionary.

#### **EqualityComparer<T>.Default**

The static EqualityComparer<T>.Default property provides a general-purpose equality comparer. Its advantage over simply using object.Equals() is that it intelligently checks if T implements IEquatable<T>. If it does, EqualityComparer<T>.Default.Equals() will call that specialized, non-boxing implementation, improving performance for value types. This is particularly useful within generic methods where T could be either a reference or value type.

| static bool Foo<T>(T x, T y) {  bool same = EqualityComparer<T>.Default.Equals(x, y); // Uses optimal equality check for T  // ... } |
| --- |

#### **ReferenceEqualityComparer.Instance (.NET 5+)**

From .NET 5 onwards, ReferenceEqualityComparer.Instance provides an equality comparer that *always* applies referential equality, even for value types (for which its Equals method will always return false). This can be useful when you explicitly need to ensure objects are compared by reference, regardless of their Equals overrides.

### **IComparer<T> and Comparer<T>**

Comparers are used to introduce custom ordering logic for sorted dictionaries and collections. It's important to distinguish them from equality comparers:

* **IComparer<T> is for *ordering***, relevant for sorted collections and sorting algorithms.
* **IEqualityComparer<T> is for *equality and hashing***, relevant for hash-based collections.

The IComparer<T> interface defines a single method:

| public interface IComparer<T> {  int Compare(T x, T y); // Returns < 0 if x < y, 0 if x == y, > 0 if x > y } |
| --- |

Similar to equality comparers, there's an abstract Comparer<T> class that you can subclass for convenience:

| public abstract class Comparer<T> : IComparer<T>, IComparer {  public static Comparer<T> Default { get; }  public abstract int Compare(T x, T y); // You override this  // ... default implementation for IComparer } |
| --- |

You implement the Compare(T x, T y) method to define the sorting logic. A common pattern is to handle nulls and then delegate to an inner comparison or use built-in comparison methods (e.g., CompareTo()).

Let's define a Wish class and a comparer that sorts wishes by Priority:

| class Wish {  public string Name;  public int Priority;  public Wish(string name, int priority) { Name = name; Priority = priority; } }  class PriorityComparer : Comparer<Wish> {  public override int Compare(Wish x, Wish y)  {  if (object.Equals(x, y)) return 0; // Optimization: if same object, they are equal  if (x == null) return -1; // Null is "less than" non-null  if (y == null) return 1; // Non-null is "greater than" null  return x.Priority.CompareTo(y.Priority); // Compare by Priority  } } |
| --- |

Using object.Equals(x, y) is safer than x.Equals(y) here because it correctly handles cases where x might be null.

Here's how PriorityComparer can be used to sort a List<Wish>:

| var wishList = new List<Wish>(); wishList.Add(new Wish("Peace", 2)); wishList.Add(new Wish("Wealth", 3)); wishList.Add(new Wish("Love", 2)); wishList.Add(new Wish("3 more wishes", 1));  wishList.Sort(new PriorityComparer()); // Pass the custom comparer to Sort()  foreach (Wish w in wishList) Console.Write(w.Name + " | "); // Output: 3 more wishes | Love | Peace | Wealth | (sorted by priority) |
| --- |

Comparers are also essential for sorted collections like SortedDictionary<TKey, TValue>:

| class SurnameComparer : Comparer<string> // Compares strings {  string Normalize(string s)  {  s = s.Trim().ToUpper();  if (s.StartsWith("MC")) s = "MAC" + s.Substring(2); // Normalize "Mc" to "Mac"  return s;  }  public override int Compare(string x, string y)  => Normalize(x).CompareTo(Normalize(y)); // Compare normalized strings }  var dic = new SortedDictionary<string, string>(new SurnameComparer()); // Use custom comparer for keys dic.Add("MacPhail", "second!"); dic.Add("MacWilliam", "third!"); dic.Add("McDonald", "first!");  foreach (string s in dic.Values)  Console.Write(s + " "); // Output: first! second! third! (sorted by normalized key) |
| --- |

### 

### **StringComparer: Predefined String Comparers**

StringComparer is an incredibly useful predefined plug-in class specifically for equating and comparing strings. It implements both IEqualityComparer<string> and IComparer<string> (and their non-generic versions), making it suitable for any type of dictionary or sorted collection that uses strings as keys.

StringComparer allows you to specify various comparison rules, including language (culture) and case sensitivity. Since StringComparer is abstract, you obtain instances via its static properties:

* **StringComparer.CurrentCulture / CurrentCultureIgnoreCase:** Compares strings using the rules of the current culture, with or without ignoring case.
* **StringComparer.InvariantCulture / InvariantCultureIgnoreCase:** Compares strings using the culture-invariant rules (suitable for consistent comparisons across different locales), with or without ignoring case.
* **StringComparer.Ordinal / OrdinalIgnoreCase:** Compares strings based on the numerical value of characters (binary comparison), with or without ignoring case. This is the fastest and most reliable for consistent comparisons when linguistic rules are not needed.
* **StringComparer.Create(CultureInfo culture, bool ignoreCase):** Allows you to create a StringComparer instance with a specific CultureInfo and case sensitivity setting.

Example using StringComparer.OrdinalIgnoreCase with a Dictionary:

| var dict = new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase); dict["Joe"] = "User 1"; Console.WriteLine(dict.ContainsKey("JOE")); // Output: True (case-insensitive) |
| --- |

Example sorting an array of names using StringComparer.Create with a specific culture:

| string[] names = { "Tom", "HARRY", "sheila" }; CultureInfo ci = new CultureInfo("en-AU"); // Australian English culture Array.Sort(names, StringComparer.Create(ci, false)); // Case-sensitive, culture-aware sort |
| --- |

### 

### **IStructuralEquatable and IStructuralComparable**

While structs inherently provide structural comparison by default (two structs are equal if all their fields are equal), sometimes you need this capability as a plug-in option for other types, especially for composite types like arrays. The IStructuralEquatable and IStructuralComparable interfaces facilitate this:

| public interface IStructuralEquatable {  bool Equals(object other, IEqualityComparer comparer); // Compares structure using provided element comparer  int GetHashCode(IEqualityComparer comparer); // Computes hash using provided element comparer }  public interface IStructuralComparable {  int CompareTo(object other, IComparer comparer); // Compares structure using provided element comparer } |
| --- |

These interfaces are implemented by types like arrays (e.g., int[] implements them). When you call their Equals or CompareTo methods, you pass an IEqualityComparer or IComparer respectively. This comparer is then applied to *each individual element* within the composite object to determine its structural equality or order.

Example comparing two arrays for structural equality:

| int[] a1 = { 1, 2, 3 }; int[] a2 = { 1, 2, 3 };  IStructuralEquatable se1 = a1; // Cast the array to IStructuralEquatable  Console.WriteLine(a1.Equals(a2)); // Output: False (reference equality for arrays) Console.WriteLine(se1.Equals(a2, EqualityComparer<int>.Default)); // Output: True (structural equality using default int comparer) |
| --- |

Another example demonstrating case-insensitive structural equality for string arrays:

| string[] sArr1 = "the quick brown fox".Split(); string[] sArr2 = "THE QUICK BROWN FOX".Split();  IStructuralEquatable seArr1 = sArr1; bool isTrue = seArr1.Equals(sArr2, StringComparer.InvariantCultureIgnoreCase); // Structural equality, ignoring case Console.WriteLine(isTrue); // Output: True |
| --- |