**Customizable Collections and Proxies**  
While the standard collection classes like List<T> are incredibly convenient for direct instantiation, they don't offer a mechanism to intercept or control operations such as adding or removing items. In application development, especially with strongly typed collections, this control becomes crucial for various scenarios:

* **Event Firing:** To trigger events when an item is added, removed, or changed.
* **Property Updates:** To automatically update related properties or states based on collection modifications.
* **Validation:** To enforce business rules, detect illegal operations, or throw exceptions if an action violates application logic.

The .NET BCL addresses these needs through specific collection classes found primarily in the System.Collections.ObjectModel namespace. These classes act as **proxies** or **wrappers**, implementing standard interfaces like IList<T> or IDictionary<TKey, TValue> by forwarding calls to an underlying collection. The key feature is that operations like Add, Remove, or Clear are routed through *virtual methods*, providing "gateways" for customization through overriding.

Customizable collection classes are frequently used for collections that are exposed publicly, such as a collection of UI controls on a System.Windows.Forms class, where you need to manage the interaction with elements added or removed.

### **Collection<T> and CollectionBase**

#### **Collection<T>: A Customizable Wrapper for List<T>**

The Collection<T> class serves as a customizable wrapper specifically for List<T>. It implements standard collection interfaces such as IList<T>, ICollection<T>, and IEnumerable<T>, along with their non-generic counterparts. Crucially, Collection<T> provides four additional protected virtual methods and one protected property designed for extensibility:

| public class Collection<T> : IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable {  // ... other members  protected virtual void ClearItems();  protected virtual void InsertItem(int index, T item);  protected virtual void RemoveItem(int index);  protected virtual void SetItem(int index, T item);  protected IList<T> Items { get; } // Access to the internal list } |
| --- |

These virtual methods (ClearItems, InsertItem, RemoveItem, SetItem) act as gateways. By overriding them in a subclass, you can intercept and modify the default behavior of Add, Remove, Clear, and element assignment operations. The protected Items property allows direct access to the underlying "inner list," enabling internal modifications without triggering the virtual gateway methods.

A typical use case involves creating a subclass of Collection<T> to add custom logic. Initially, a simple subclass functions identically to List<T}:

| public class Animal {  public string Name;  public int Popularity;  public Animal(string name, int popularity) { Name = name; Popularity = popularity; } }  public class AnimalCollection : Collection<Animal> {  // AnimalCollection is already a fully functioning list of animals.  // No extra code is required initially. }  public class Zoo {  public readonly AnimalCollection Animals = new AnimalCollection(); } |
| --- |

The power comes when you need to "hook in" to the collection's behavior. For instance, to automatically manage a Zoo property on Animal objects as they are added or removed from AnimalCollection:

| public class Animal {  public string Name;  public int Popularity;  public Zoo Zoo { get; internal set; } // Property to reference the containing Zoo  public Animal(string name, int popularity) { Name = name; Popularity = popularity; } }  public class AnimalCollection : Collection<Animal> {  private Zoo zoo; // Reference to the containing Zoo   public AnimalCollection(Zoo zoo) { this.zoo = zoo; }   protected override void InsertItem(int index, Animal item)  {  base.InsertItem(index, item); // Call base implementation first  item.Zoo = zoo; // Custom logic: set the Animal's Zoo property  }   protected override void SetItem(int index, Animal item)  {  base.SetItem(index, item);  item.Zoo = zoo; // Custom logic  }   protected override void RemoveItem(int index)  {  this[index].Zoo = null; // Custom logic: clear the Animal's Zoo property  base.RemoveItem(index);  }   protected override void ClearItems()  {  foreach (Animal a in this) a.Zoo = null; // Custom logic  base.ClearItems();  } }  public class Zoo {  public readonly AnimalCollection Animals;  public Zoo() { Animals = new AnimalCollection(this); } // Pass 'this' to the collection } |
| --- |

This example illustrates how overriding these virtual methods allows the AnimalCollection to maintain the Zoo property on Animal objects whenever they are added, set, or removed.

Collection<T> also has a constructor that accepts an existing IList<T>. Unlike other collections that copy elements, Collection<T> *proxies* the supplied list, meaning changes made through the Collection<T> wrapper will affect the underlying list, and vice versa (though changes made directly to the underlying list will bypass Collection<T>'s virtual methods).

**CollectionBase: The Non-Generic Counterpart**

CollectionBase is the non-generic version of Collection<T>. While it provides similar extensibility, it is generally considered more cumbersome to use. Instead of the simpler virtual methods of Collection<T>, CollectionBase uses a pair of "hook" methods for each operation (e.g., OnInsert, OnInsertComplete), effectively doubling the methods required for customization. Furthermore, being non-generic, you must implement typed methods (like a typed indexer and Add method) when subclassing it, leading to more verbose and less type-safe code compared to Collection<T>. CollectionBase is best considered for backward compatibility with older codebases.

### **KeyedCollection<TKey, TItem> and DictionaryBase**

#### **KeyedCollection<TKey, TItem>: List with Fast Keyed Lookup**

KeyedCollection<TKey, TItem> is a specialized collection that subclasses Collection<TItem>. It combines the features of a linear list (allowing access by index) with the functionality of a dictionary (allowing fast lookup by a unique key). The key difference from a standard dictionary is that KeyedCollection does *not* implement IDictionary; instead, the key for each TItem is extracted *from the item itself* via an abstract method you must implement.

You can think of KeyedCollection<TKey, TItem> as a Collection<TItem> with the added benefit of fast item retrieval by a specified key.

Key members of KeyedCollection<TKey, TItem>:

| public abstract class KeyedCollection<TKey, TItem> : Collection<TItem> {  // ... other members  protected abstract TKey GetKeyForItem(TItem item); // Abstract method to implement  protected void ChangeItemKey(TItem item, TKey newKey); // Call if an item's key changes  public TItem this[TKey key] { get; } // Indexer for fast lookup by key  protected IDictionary<TKey, TItem> Dictionary { get; } // Access to the internal dictionary } |
| --- |

* **GetKeyForItem(TItem item):** This is the abstract method that you *must* override in your subclass. It defines how the unique key (TKey) is extracted from each TItem.
* **ChangeItemKey(TItem item, TKey newKey):** If a property of an item that serves as its key changes after the item has been added to the collection, you must call this method to update the internal dictionary correctly.
* **this[TKey key]:** Provides the dictionary-like capability to retrieve an item directly using its key.
* **Dictionary property:** Exposes the internal dictionary used for keyed lookups. This dictionary is typically created on demand when the first item is added, or its creation can be delayed by specifying a creationThreshold in the constructor.

The most common application for KeyedCollection<TKey, TItem> is to provide a collection where items can be accessed both by their numerical index and by a unique identifier (like a name or ID):

| public class Animal {  private string name; // Backing field for the Name property   public string Name  {  get { return name; }  set  {  if (Zoo != null) Zoo.Animals.NotifyNameChange(this, value); // Notify collection of key change  name = value;  }  }  public int Popularity;  public Zoo Zoo { get; internal set; }   public Animal(string name, int popularity) { Name = name; Popularity = popularity; } }  public class AnimalCollection : KeyedCollection<string, Animal> {  private Zoo zoo;   public AnimalCollection(Zoo zoo) { this.zoo = zoo; }   internal void NotifyNameChange(Animal a, string newName) =>  this.ChangeItemKey(a, newName); // Call inherited method when Animal's name changes   protected override string GetKeyForItem(Animal item) => item.Name; // Implement key extraction   // Override Collection<T> methods (InsertItem, SetItem, RemoveItem, ClearItems) as before  // to manage the Animal.Zoo property.  protected override void InsertItem(int index, Animal item) { base.InsertItem(index, item); item.Zoo = zoo; }  protected override void SetItem(int index, Animal item) { base.SetItem(index, item); item.Zoo = zoo; }  protected override void RemoveItem(int index) { this[index].Zoo = null; base.RemoveItem(index); }  protected override void ClearItems() { foreach (Animal a in this) a.Zoo = null; base.ClearItems(); } }  public class Zoo {  public readonly AnimalCollection Animals;  public Zoo() { Animals = new AnimalCollection(this); } } |
| --- |

Demonstrating its use:

| Zoo zoo = new Zoo(); zoo.Animals.Add(new Animal("Kangaroo", 10)); zoo.Animals.Add(new Animal("Mr Sea Lion", 20));  Console.WriteLine(zoo.Animals[0].Popularity); // Access by index: 10 Console.WriteLine(zoo.Animals["Mr Sea Lion"].Popularity); // Access by key: 20  zoo.Animals["Kangaroo"].Name = "Mr Roo"; // Change the key via Animal property Console.WriteLine(zoo.Animals["Mr Roo"].Popularity); // Access by new key: 10 |
| --- |

#### **DictionaryBase: The Non-Generic Keyed Collection**

DictionaryBase is the non-generic, legacy counterpart to KeyedCollection<TKey, TItem>. It implements IDictionary and uses the same "hook" method pattern as CollectionBase (e.g., OnInsert, OnInsertComplete), making it similarly verbose and less type-safe. While it offers the advantage of directly implementing IDictionary (meaning you don't need to subclass just to get keys), this benefit is largely negated by the requirement to subclass it for custom behavior. DictionaryBase is primarily for backward compatibility.

### **ReadOnlyCollection<T>: Providing a Read-Only View**

ReadOnlyCollection<T> is a simple yet powerful wrapper class that provides a read-only view of any collection that implements IList<T>. Its primary purpose is to allow a class to expose a collection publicly in a way that prevents external modification, while the exposing class can still update the collection internally.

Key characteristics:

* **Wrapper:** It acts as a proxy, forwarding calls to an underlying IList<T> supplied in its constructor.
* **Live View:** It maintains a permanent reference to the input collection. It does *not* create a static copy. This means any subsequent internal changes made to the original collection will be reflected through the ReadOnlyCollection<T> wrapper.
* **Robustness:** Unlike simply exposing an IReadOnlyList<T> interface (which can sometimes be downcast at runtime to a mutable interface), ReadOnlyCollection<T> throws a NotSupportedException if an attempt is made to call a modifying method (like Add or Remove) through its interface or by casting it to IList<T>.

Consider a scenario where you want to expose a list of names publicly, but only allow internal modifications:

| public class Test {  private List<string> names = new List<string>();  public ReadOnlyCollection<string> Names { get; private set; } // Publicly expose as read-only   public Test()  {  Names = new ReadOnlyCollection<string>(names); // Wrap the internal list  }   public void AddInternally(string name)  {  names.Add(name); // Internal modification is allowed  } } |
| --- |

Demonstrating external vs. internal access:

| Test t = new Test(); Console.WriteLine(t.Names.Count); // 0  t.AddInternally("Alice"); // Internal addition Console.WriteLine(t.Names.Count); // 1  // t.Names.Add("Bob"); // Compiler error: 'Add' is not a member of ReadOnlyCollection<T> // ((IList<string>)t.Names).Add("Charlie"); // Runtime error: NotSupportedException |
| --- |

By using ReadOnlyCollection<T>, you provide a robust safeguard against unintended external modifications to your internal collection, enhancing the encapsulation and integrity of your class.