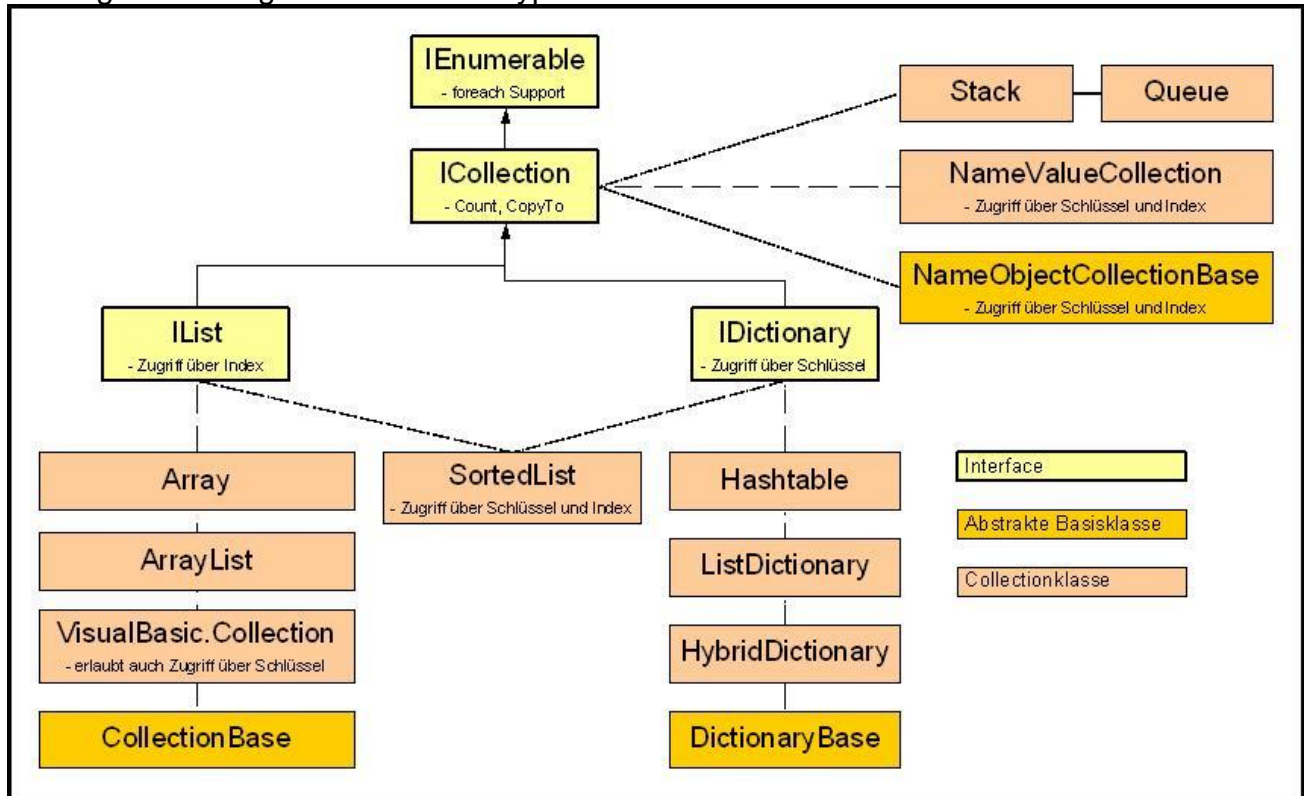


## Collections

Collection-Typen spielen in jeder Klassenbibliothek eine wichtige Rolle. Sie stellen Datenstrukturen zur Verfügung, mit denen man Sammlungen von Objekten verarbeiten kann, angefangen von Listen über Hashtabellen bis zu Stacks und Queues.

Auszug der wichtigsten Collection-Typen:



## Die Klasse Array

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable
```

### Eigenschaften und Methoden der Klasse Array (aus System)

<code>Array.BinarySearch(arr, obj)</code>	durchsucht das Feld <code>arr</code> nach dem Eintrag <code>obj</code> . Die Methode setzt voraus, dass das Feld sortiert ist, und liefert als Ergebnis die Indexnummer des gefundenen Eintrags. Optional kann eine eigene Vergleichsmethode angegeben werden.
<code>Array.BinarySearch(arr, o, ICompare)</code>	
<code>Array.Clear(arr, n, m)</code>	setzt <code>m</code> Elemente beginnend mit <code>arr[n]</code> auf den Standardwert des zugrunde liegenden Datentyps.
<code>arr2 = (Datentyp[])(arr1.Clone())</code>	weist <code>arr2</code> eine Kopie von <code>arr1</code> zu.
<code>Array.Copy(arr1, n1, arr2, n2, m)</code>	kopiert <code>m</code> Elemente vom Feld <code>arr1</code> in das Feld <code>arr2</code> , wobei <code>n1</code> der Startindex in <code>arr1</code> und <code>n2</code> der Startindex in <code>arr2</code> ist.
<code>arr = Array.CreateInstance (Typ, n [,m [,o]])</code>	erzeugt ein Feld der Größe <code>(n,m,o)</code> , wobei in den einzelnen Elementen Objekte des Typs <code>type</code> gespeichert werden können.
<code>arr.GetLength(dimension):</code>	ermittelt die Anzahl der Elemente der Dimension <code>dimension</code> des Arrays <code>arr</code> .
<code>arr.GetUpperBound(dimension):</code>	liefert die obere Grenze der Dimension <code>dimension</code> des Arrays <code>arr</code> .
<code>arr.GetLowerBound(dimension):</code>	liefert die untere Grenze der Dimension <code>dimension</code> des Arrays <code>arr</code> . Diese Funktion ist zwar selten nützlich, es kann jedoch vorkommen, dass ein Array nicht die Untergrenze 0 hat.
<code>Array.Reverse(arr)</code>	vertauscht die Reihenfolge der Elemente des Arrays <code>arr</code> .
<code>arr.SetValue(data, n [,m [,o]])</code>	speichert im Element <code>arr(n, m, o)</code> den Wert <code>data</code> .
<code>Array.Sort(arr [,ICompare] )</code>	sortiert <code>arr</code> (unter Anwendung der Vergleichsfunktion des <code>ICompare</code> -Objekts, falls angegeben).

## Beispiel: Array kopieren

```
class Program {
    static void Main(string[] args) {
        Random rnd = new Random();
        int[] x = new int[500];

        // Quell-Array mit Zahlen füllen
        for(int i = 0; i <= x.Length - 1; i++)
            x[i] = rnd.Next(0, 1000);

        Console.WriteLine("Wie viele Elemente sollen kopiert werden? ");
        int count= Convert.ToInt32(Console.ReadLine());

        // Teil des Quell-Arrays in das Ziel-Array kopieren
        int[] y = new int[count];
        Array.Copy(x, y, count);

        // Konsolenausgabe des Ziel-Arrays
        for(int i = 0; i <= count - 1; i++)
            Console.WriteLine("Element {0,3} = {1,4}", i + 1, y[i]);
        Console.ReadLine();
    }
}
```

## Beispiel: Array abwärts sortieren

```
int[] a = new int[5] { 3, 17, 5, 8, 6 };
int[] b = { 6, 2, 9, 2, 0 };
int[] c = (int[])a.Clone(); // kopiert ein Array

foreach (int i in a) Console.WriteLine(i);
Console.WriteLine();
foreach (int i in b) Console.WriteLine(i);
Console.WriteLine();
foreach (int i in c) Console.WriteLine(i);

Array.Sort(b); // sortiert ein Array
Console.WriteLine();
foreach (int i in b) Console.WriteLine(i);

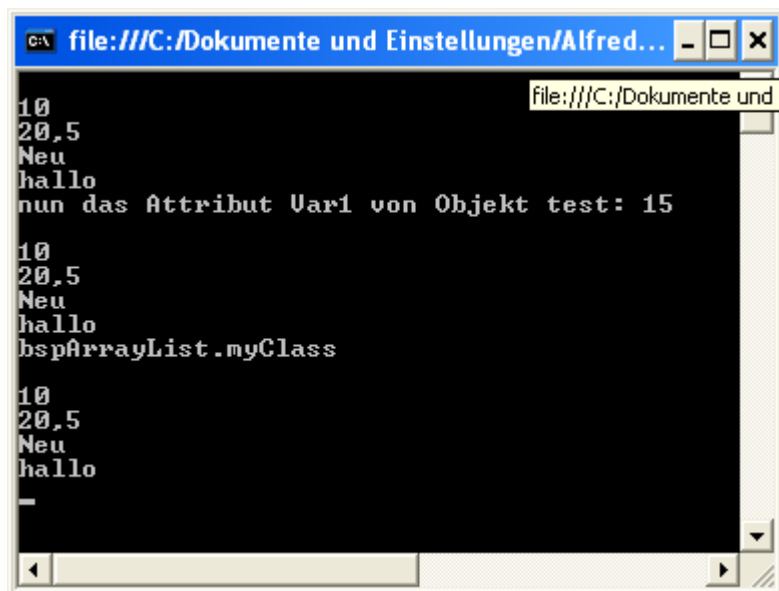
Array.Reverse(b); // Reihenfolge umdrehen
Console.WriteLine();
foreach (int i in b) Console.WriteLine(i);
```

## Die Klasse ArrayList

Die bisherigen Arrays konnten zwar während der Laufzeit erstellt werden, also mit dynamischer Speicherreservierung, aber eine Redimensionierung während der Laufzeit war nicht möglich. Solche flexiblen Arrays, die während der Laufzeit beliebig viele Elemente aufnehmen können, sind in den Klassen des Namensraumes `System.Collections` zu finden.

Die Klasse **ArrayList** ist eine einfache dynamische Listenklasse. Sie verfügt über Methoden zum Hinzufügen (*Add* und *Insert*) sowie zum Entfernen (*Remove*) von beliebigen Elementen. Die Klasse **ArrayList** benutzt die Klasse **object**.

Nach dem Starten des Programms (Beispiel nächste Seite) erscheint folgender Bildschirm:



```
file:///C:/Dokumente und Einstellungen/Alfred...
10
20,5
Neu
hallo
nun das Attribut Var1 von Objekt test: 15

10
20,5
Neu
hallo
bspArrayList.myClass

10
20,5
Neu
hallo
-
```

## Beispiel:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Collections;

namespace bspArrayList
{
    // Klasse zum Testen für das Aufnehmen in der Liste
    class myClass
    {
        private int var1;
        public int Var1
        {
            get { return var1; }
            set { var1 = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList AL = new ArrayList(); // Arrayliste instanzieren

            AL.Add(10);
            AL.Add(20.5);
            AL.Add("hallo");

            AL.Insert(2, "Neu");

            Console.WriteLine();
            foreach (object o in AL) Console.WriteLine(o);

            myClass test = new myClass();
            test.Var1 = 15;
            AL.Add(test); // auch Objekte können direkt in die Liste aufgenommen
                        //werden - können aber nicht einfach ausgegeben werden.

            myClass test1 = (myClass)AL[AL.Count-1]; // letztes Element
            // holen - hier das Objekt test
            Console.WriteLine("nun das Attribut Var1 von Objekt test:
                               "+test1.Var1);

            Console.WriteLine();
            foreach (object o in AL) Console.WriteLine(o); // Objekt kann
            // nicht ausgegeben werden.

            AL.Remove(test); // Objekt wieder löschen
            Console.WriteLine();
            foreach (object o in AL) Console.WriteLine(o); // Objekt kann
            // nicht ausgegeben werden.

            Console.Read(); // nur zum Anhalten für das DOS-Fenster
        }
    }
}
```

## Die Klasse Hashtable

**Hashtables** (Hash-Tabellen) sind **Datenstrukturen**, die den Arrays sehr ähnlich sind. Hashtables können in C# ähnlich wie eine ArrayList verwendet werden, der Zugriff erfolgt über einen **Index**.

Der Index ist ein spezieller **Key**, welcher eine Zahl (z.B. vom Typ *Float* oder *Integer*) genauso wie ein *String* sein darf:

```
1. // Beispiele, Syntax: hashtablename[key] = value;
2.
3. ht_pass[32.3] = "juhu";
4. ht_pass[5] = "tobias";
5. ht_pass["zoo"] = 45.3;
```

## Sinn der Hashtables

**Hashtables** sind sinnvoll, wenn sich Listen bzw. Arrays **verknüpfen bzw. verschachteln** lassen. So kann der Wert eines Listenelementes der Schlüssel zu einem Wert eines anderen Elementes sein. Zwar ließe sich das auch mit einem normalen Array (und somit Integer-Werten als Index) realisieren, dann wären jedoch einige zusätzliche Listen/Tabellen nötig, um von einer Zahl auf einen speziellen Schlüssel und von diesem Schlüssel wieder auf einen anderen Zahlenschlüssel zu verweisen.

Die Hashtables in C# sind sehr dynamisch, denn intern ist ein Hashtable auch nur so groß, wie es notwendig ist. Der Zugriff ist jedoch absolut schnell, **konstant schnell! Unabhängig von der Größe des Hashtable ist die Zugriffszeit konstant.**

Im unteren Beispiel wird ein Hashtable angelegt, welches Benutzernamen als Key mit dazugehörigem Benutzerpasswort als Value speichert:

```
1. // Deklaration
2. Hashtable ht_pass;
3.
4. // Instanziierung des Hashtable ht_pass
5. ht_pass = new Hashtable();
6.
7. // Anlegen von Schlüssel-Werte-Paare,
8. //ähnlich einem normalen Array
9. ht_pass["benny"] = "porsche";
10. ht_pass["david"] = "programmierer";
11. ht_pass["thomas"] = "kaufmann";
12. ht_pass["daniela"] = "hamburg";
13. ht_pass["sarah"] = "universität";
```

Über das Schlüsselwort kann auf das Array zu gegriffen werden und zum Beispiel das Passwort von Benny ausgegeben werden:

```
1. Console.Write("Name eingeben. ");
2. Name = Console.ReadLine();
3. Console.WriteLine("Ihr Passwort lautet: {0}", ht_pass[Name]);
```

Aber Vorsicht! Wird ein Schlüssel in den Hashtable eingesetzt, welcher nicht im Hashtable vorkommt, geschieht eine Exception! (Programmabsturz)

Deshalb sollte man vorher prüfen, ob der Hashtable nicht *Null* ist und der gewünschte Key im Hashtable vorkommt.

Der gesamte Inhalt kann mit der foreach-Schleife ausgegeben werden:

```
1. foreach (object key in ht_pass.Keys)
2. {
3.     Console.WriteLine("Das Passwort von {0} lautet: {1}", key, ht_pass[key]);
4. }
```

Ausgabe:

```
Das Passwort von benny lautet porsche
Das Passwort von david lautet programmierer
Das Passwort von thomas lautet kaufmann
Das Passwort von daniela lautet hamburg
Das Passwort von sarah lautet universität
```

## Die Generische Auflistungsklasse *List<T>*

List hat die gleichen Methoden und Eigenschaften wie ArrayList. Jedoch unterscheidet sie sich von ArrayList dadurch, dass Sie vorschreiben müssen, welcher Typ von der Collection verwaltet werden soll.

Eine ArrayList kann viele verschiedenen Datentypen aufnehmen, doch muss der Programmierer selbst dafür Sorge tragen, dass der Liste nur Elemente zugefügt werden, die auch fehlerlos verarbeitet werden können.

Eine Liste die von vornherein auf einem bestimmten Datentyp festgelegt ist, ist in den meisten Fällen sinnvoller. Dafür gibt es generische Listen wie „List<T>“.

siehe auch:

[http://openbook.galileocomputing.de/visual\\_csharp\\_2010/visual\\_csharp\\_2010\\_08\\_006.htm#mj18197a0d05ad69c2162f5f0fc8bcb796](http://openbook.galileocomputing.de/visual_csharp_2010/visual_csharp_2010_08_006.htm#mj18197a0d05ad69c2162f5f0fc8bcb796)

Zum Thema Generika oder generische Listen lesen Sie am besten:

<http://msdn.microsoft.com/de-de/library/512aeb7t%28v=vs.80%29.aspx>

oder

[http://openbook.galileocomputing.de/visual\\_csharp\\_2010/visual\\_csharp\\_2010\\_07\\_002.htm](http://openbook.galileocomputing.de/visual_csharp_2010/visual_csharp_2010_07_002.htm)

Die Definition einer generischen Liste ist denkbar einfach:

```
List<string> namen = new List<string>(); // erzeugt eine Liste aus Zeichenketten
```

```
List<int> zahlen = new List<int>(); // erzeugt eine Liste aus Ganzzahlen.
```

Diese Definition ist fest. Der Versuch artfremde Daten an die Liste anzuhängen wird vom Compiler verweigert. Dadurch minimiert sich auch die Gefahr, dass sich während der Nutzung des Programmes Daten in die Liste einschleichen können, die dort nicht hinein gehören.

Diese zwei Befehle funktionieren nicht!

```
namen.Add(1);
```

```
zahlen.Add("Hallo");
```

Eine generische Liste kann also aus beliebigen Daten gleichen Typs bestehen, und natürlich auch aus Daten benutzerdefinierten Typs. Auch aus einer Klasse..



Definition einer Liste mit Objekten:

```
List<Person> personen = new List<Person>();
```

Vorher muss eine Klasse definiert werden:

```
class Person
{
    public string name;
    public string vorname;

    public Person ()
    {}

    public Person (string vorname, string name)
    {
        this.name = name;
        this.vorname = vorname;
    }
}
```

Mit „Add“ können jetzt die Objekte der Liste zugefügt werden:

```
Person h = new Person("Hans", "Meier");
personen.Add(h);
```

oder:

```
personen.Add(new Person("Holger", "Schmitt"));
```

Mittels einer foreach-Schleife kann die komplette Liste auszugeben werden:

```
foreach (Person per in personen)
{
    Console.WriteLine("{0} {1}", per.vorname, per.name);
}
```

Weitere Informationen entnehmen Sie bitte aus:

<http://msdn.microsoft.com/de-de/library/vstudio/6sh2ey19.aspx>

oder

[http://openbook.galileocomputing.de/visual\\_csharp\\_2010/visual\\_csharp\\_2010\\_08\\_006.htm#mj7d3dbb77ecb4c840e172ac812852d9f2](http://openbook.galileocomputing.de/visual_csharp_2010/visual_csharp_2010_08_006.htm#mj7d3dbb77ecb4c840e172ac812852d9f2)

## Interfaces für Arrays und Listen (aus - <http://support.microsoft.com/kb/320727/de>)

Bei einfachen Datentypen (int, float, string) ist es möglich, die Listen mit den Befehl **sort** zu sortieren.

Das ist bei Objekten ohne weiteres nicht möglich. Hier kommen die Interfaces **IComparable** und **ICompare** ins Spiel. Sie zwingen den Programmierer für seine Klasse eine spezielle Methode (CompareTo - IComparable, Compare - IComparer) zu implementieren.

### IComparable:

Die Rolle von **IComparable** ist es, eine Methode zum Vergleich zweier Objekte eines bestimmten Typs bereitzustellen.

Dies ist erforderlich, wenn Sie zum Beispiel eine Sortierfunktion für Ihr Objekt bereitstellen möchten.

Wenn Sie die **IComparable** -Schnittstelle implementieren, müssen Sie die **CompareTo** - Methode wie folgt implementieren:

**int** CompareTo(**object** name)

Rückgabewert vom Typ **int**,

Übergabeparameter vom Typ **object** (irgendein Objekt ihrer Klasse)

### Beispiel:

// Implement IComparable CompareTo method - provide default sort order.

```
int IComparable.CompareTo(object obj)
{
    car c=(car)obj;
    return String.Compare(this.make,c.make);
}
```

Der Vergleich in der Methode unterscheidet sich je nach Datentyp des Werts, der verglichen wird. **String.Compare** wird in diesem Beispiel verwendet, da die Eigenschaft, die für den Vergleich ausgewählt ist, eine Zeichenfolge ist.

## IComparer

Die Rolle von **IComparer** ist es, zusätzliche Vergleichsmechanismen bereitzustellen. Beispielsweise sollten Sie Sortierfunktionen für Ihre Klassen zum Sortieren nach mehreren Feldern oder Eigenschaften in aufsteigender und absteigender Sortierung nach einem Feld oder beides bereitstellen.

Die Verwendung von **IComparer** ist ein zweistufiger Prozess. Zunächst deklarieren Sie eine Klasse, die **IComparer** implementiert, und dann implementieren Sie die Methode **Compare**:

```
private class sortYearAscendingHelper : IComparer
{
    int IComparer.Compare(object a, object b)
    {
        car c1=(car)a;
        car c2=(car)b;
        if (c1.year > c2.year)
            return 1;
        if (c1.year < c2.year)
            return -1;
        else
            return 0;
    }
}
```

Beachten Sie, dass die **IComparer.Compare** -Methode einen tertiären Vergleich benötigt. Je nachdem, ob ein Wert größer als, gleich oder kleiner als die andere ist 1, 0 oder -1 zurückgegeben. Die Sortierreihenfolge (aufsteigend oder absteigend) kann durch den Wechsel der logischen Operators in dieser Methode geändert werden.

Im zweite Schritt wird eine Methode deklariert, die eine Instanz des Objekts **IComparer** zurückgibt:

```
public static IComparer sortYearAscending()
{
    return (IComparer) new sortYearAscendingHelper();
}
```

In diesem Beispiel wird das Objekt als zweites Argument verwendet, wenn Sie die überladene **Array.Sort** -Methode aufrufen, die **IComparer** akzeptiert. Die Verwendung von **IComparer** ist nicht beschränkt auf Arrays. Es wird als Argument in eine Reihe von anderen Auflistung und Steuerelementklassen akzeptiert.

Lesen Sie auch den Artikel unter <http://support.microsoft.com/kb/320727/de>  
"Gewusst wie: Verwenden Sie die IComparable und IComparer-Schnittstellen in Visual C#"