Collections

- Collections can hold more than one element – much like an array.
- Collections are *mutable* – they are not fixed in size like an array (which is *immutable*) – the number of elements a collection holds can shrink or grow.
- Collection *classes* allow for more flexibility in how data is handled than an array.
- An *untyped* collection can hold any type of data in the same collection object.
  - The only kind of collections provided by C# prior to version 2.0 of the .NET framework – in the *Systems.Collections* namespace.
  - Each data element is stored as a generic object type – when retrieving data from this type of collection, an explicit type cast must be used on each element.
  - Untyped collections are typically not used due to errors that can occur and the work that is required when casting data elements.
- A *typed* collection (also called a *generic* collection) can hold only one type of data in the same collection object.
  - Added to C# starting with version 2.0 of the .NET framework – in the *Systems.Collections.Generic* namespace.
  - The < > brackets indicate the use of a programming concept known as *Generics* which is covered later in this course. The **T** indicates the data **t**ype the collection will hold.
  - When a collection object is created, the programmer specifies the type of data to store.
  - Preferred collection classes to use since no casting is required and no casting errors will occur.

Collection Classes

Typed Collection Class	Untyped Collection Class
`List<T>`	`ArrayList`
`SortedList<TKey, TValue>`	`SortedList`
`Queue<T>`	`Queue`
`Stack<T>`	`Stack`
`Dictionary<TKey, TValue>`	`Hashtable`
`SortedDictionary<TKey, TValue>`	
`LinkedList<T>`	

The List<T> Collection Class

- The List<T> class has properties similar to an array. The key difference is that these classes automatically expand as the number of elements increases and shrink with calls to certain methods.
- Each element can be individually accessed by index, just like an array. You can set and get elements in the list collection using the index operator, where the index values start at zero and go to one less than the number of elements.

Creating a List<T> Object

General Form: `List<T> variableName = new List<T>();`

Examples:

```
List<int> nums = new List<int>();   // create a zero element List of ints (Capacity = 0, Count = 0)
List<double> values = new List<double>(3);   // create a 3 element List of doubles (Capacity = 3, Count = 0)
List<string> words = new List<string>();   // create a zero element List of strings (Capacity = 0, Count = 0)

// create and initialize a List of ints
List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};
```

Note: The `new List<int>()` is required – only array initialization can leave out this part.

Adding Elements to a List

- Method calls are required to add values to a list. Each value is added to the end of the list unless the Insert( ) method is used.
- Method calls are also required to remove values from a list.

```
List<int> nums = new List<int>();
nums.Add(1);   // element 1 – index 0
nums.Add(2);   // element 2 – index 1
nums.Add(4);   // element 3 – index 2
nums.Add(5);   // element 4 – index 3
nums.Add(6);   // element 5 – index 4
nums.Insert(2, 3);   // element 3 – index 2 – is now a new element, elements 4 to 6 have their index positions increased by 1
```

- To get or change an existing value, access the element using the appropriate index value using the index operator.

```
List<int> nums = new List<int>() {6, 2, 3, 4, 5, 6};
nums[0] = 1;   // replaces the value 6 with the value 1 for the first element – index 0
```

Common List<T> Properties & Methods

Name	Description	Example

`[index]`	Gets or sets the element at the specified index. The index for the first item in a list is 0.	`List<int> nums = new List<int>() {6, 2, 3, 4, 5, 6};`
		`nums[0] = 1; // first element – index 0 – now has a value of 1`
		`MessageBox.Show(nums[0] + "\n" + nums[2]); // 1 and 3 displayed`

`Capacity`	Gets or sets the number of elements the list can hold.	`List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};` `int numsCapacity = nums.Capacity; // numsCapacity = 8`
`Count`	Gets the number of actual elements in the list.	`List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};` `int numsCount = nums.Count; // numsCount = 6`
`Add``(value)`	Adds an element to the end of a list.	`List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};` `nums.Add(7); // added to end of list – element 7 added with a value of 7`

`Clear()`	Removes all elements from the list and sets its Count property to zero.	`List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};`
		`nums.Clear();`
		`int numsCapacity = nums.Capacity; // numsCapacity = 8 (no change)`
		`int numsCount = nums.Count; // numsCount = 0`

`Contains``(value)`	Returns a Boolean value that indicates if the list contains the specified value.	`List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};` `bool isPresent = nums.Contains(4); // isPresent = true`

`IndexOf``(value)`
`LastIndexOf``(value)`
	Retruns the index of the first/last occurrence of the specified value (-1 if value not found). *Overloaded*	`List<int> nums = new List<int>() {1, 2, 3, 3, 2, 1};`
		`int first = nums.IndexOf(2); // first = 2`
		`int last = nums.LastIndexOf(2); // last = 4`
		`int notThere = nums.LastIndexOf(6); // notThere = -1`

`Insert``(index, value)`	Inserts an element into a list at the specified index.	`List<int> nums = new List<int>() {1, 2, 3, 3, 2, 1};` `nums.Insert(3, 4); // nums List is now…{1, 2, 3, 4, 3, 2, 1}`

`Remove``(value)`	Removes the first occurrence of the specified value from the list.	`List<int> nums = new List<int>() {1, 2, 3, 4, 3, 2, 1};`
		`nums.Remove(4); // nums List is now…{1, 2, 3, 3, 2, 1}`
		`// a bool is returned to indicate if removal was successful`

`RemoveAt``(index)`	Removes the element at the specified index of a list.	`List<int> nums = new List<int>() {1, 2, 3, 4, 3, 2, 1};` `nums.RemoveAt(4); // nums List is now…{1, 2, 3, 4, 2, 1}`

`BinarySearch``(value)`	Searches a **sorted** list for a specified value and returns the index for that value.	`List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};`
		`int index = nums.BinarySearch(4); // index = 3`
		`index = nums.BinarySearch(12); // index = -7 (the negative Count)`

`Sort()`	Sorts the elements in a list into ascending order.	`List<int> nums = new List<int>() {1, 2, 3, 4, 3, 2, 1};` `nums.Sort(); // nums List is now…{1, 1, 2, 2, 3, 3, 4}`

Capacity vs. Count

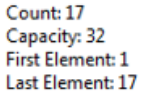
- The *Capacity* is the potential number of elements the List can store. The *Count* is the number of actual values stored in the List.
- If a List is not declared with an initial size (or is not initialized with values), the starting capacity is zero. When the first value is added to the List, the capacity becomes 4. When the fifth value is added, the capacity becomes 8. When the ninth value is added, the capacity becomes 16. The capacity keeps doubling – 4, 8, 16, 32, 64, 128, 256, 512, 1024…

```
List<int> nums = new List<int>() {1, 2};   // Count = 2, Capacity = 4
List<int> nums = new List<int>() {1, 2, 3, 4};   // Count = 4, Capacity = 4
List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6};   // Count = 6, Capacity = 8
List<int> nums = new List<int>() {1, 2, 3, 4, 5, 6, 7, 8, 9};   // Count = 9, Capacity = 16
```

Using a List<T>

Example: Using a for loop to initialize a List<> and a foreach loop to display the values of a List<>

```
List<int> nums = new List<int>();
// intialize list with numbers from 1 to 17
for (int count = 1; count <= 17; count++)
{
    nums.Add(count);
}
string display = "Count: " + nums.Count + "\nCapacity: " + nums.Capacity +
                 "\nFirst Element: " + nums[0] +
                 "\nLast Element: " + nums[nums.Count - 1];
```



```
// display each element
display = "";
foreach (int element in nums)
{
    display += element + "\n";
}
```

*OR…using a **for** loop*

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

```
// display each element
display = "";
for (int index = 0; index < nums.Count; index++)
{
    display += nums[index] + " ";
}
```

Example: Removing a Value Contained in a List<>

```
List<int> nums = new List<int>() { 1, 2, 3, 4, 3, 2, 1 };
if (nums.Contains(2))
    nums.Remove(2);
string display = "";
foreach (int value in nums)
    display += value + "  ";
```

List<T> Example

The following code example demonstrates several properties and methods of the List <T> generic class of type string.

```
// the default constructor is used to create a list of strings with the default capacity
List<string> dinosaurs = new List<string>();

// the Capacity property is displayed and then the Add( ) method is used to add several items.
string display = "Capacity: " + dinosaurs.Capacity + "\n";
dinosaurs.Add("Tyrannosaurus");
dinosaurs.Add("Amargasaurus");
dinosaurs.Add("Mamenchisaurus");
dinosaurs.Add("Deinonychus");
dinosaurs.Add("Compsognathus");

// the items are listed, the Capacity and Count property is displayed
// shows that the capacity has been increased as needed
foreach (string dino in dinosaurs)
{
    display += "\n" + dino;
}
display += "\n\nCapacity: " + dinosaurs.Capacity +
           "\nCount: " + dinosaurs.Count;

// the Contains( ) method is used to test for the presence of an item in the list
display += "\n\nContains(\"Deinonychus\"): " +
           dinosaurs.Contains("Deinonychus");

// the Insert( ) method is used to insert a new item in the middle of the list
// the list is displayed again
dinosaurs.Insert(2, "Compsognathus");
display += "\n\nInsert(2, \"Compsognathus\")\n";
foreach (string dino in dinosaurs)
{
    display += "\n" + dino;
}

// the indexer is used to retrieve an item
display += "\n\ndinosaurs[3]: " + dinosaurs[3];

// the Remove( ) method is used to remove the first instance of the duplicate item added earlier
// the contents are displayed again (the Remove( ) method always removes the first instance it encounters)
dinosaurs.Remove("Compsognathus");
display += "\n\nRemove(\"Compsognathus\")\n\n";
foreach (string dino in dinosaurs)
{
    display += dino + "\n";
}
```

```
Capacity: 0

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

Capacity: 8
Count: 5

Contains("Deinonychus"): True

Insert(2, "Compsognathus")

Tyrannosaurus
Amargasaurus
Compsognathus
Mamenchisaurus
Deinonychus
Compsognathus

dinosaurs[3]: Mamenchisaurus

Remove("Compsognathus")

Tyrannosaurus
Amargasaurus
Mamenchisaurus
Deinonychus
Compsognathus

TrimExcess()
Capacity: 5
Count: 5

Clear()
Capacity: 5
Count: 0
```

```
// the TrimExcess( ) method is used to reduce the capacity to match the count
// the Capacity and Count properties are displayed
// if the unused capacity had been less than 10 percent of total capacity, the list would not have been resized
dinosaurs.TrimExcess();
display += "\nTrimExcess()" +
           "\nCapacity: " + dinosaurs.Capacity +
           "\nCount: " + dinosaurs.Count;

// the Clear( ) method is used to remove all items from the list
// the Capacity and Count properties are displayed
dinosaurs.Clear();
display += "\n\nClear()" +
           "\nCapacity: " + dinosaurs.Capacity +
           "\nCount: " + dinosaurs.Count;
```

The Dictionary<TKey, TValue> Collection Class

- Represents a collection of keys and values
  - TKey: The data type of the keys in the dictionary
  - TValue: The type of the values in the dictionary
- Provides a way to retrieve data by using a key value instead of an index number
  - much like accessing a record in a database by first searching for its primary key
  - adds some complexity to the access of dictionary elements but lookups by key is more efficient than lookups by value
- Key values have to be unique or an exception is thrown

Creating a Dictionary<TKey, Tvalue> Object

General Form: `Dictionary<Tkey, Tvalue> variableName = new Dictionary<Tkey, Tvalue>();`

Examples:

```
// create a zero element Dictionary of integer keys and character values – Count = 0
Dictionary<int, char> letters = new Dictionary<int, char>();
// create a zero element Dictionary of string keys and string values
Dictionary<string, string> letters = new Dictionary<string, string>();

// create and initialize a 4 element Dictionary of int keys and char values
Dictionary<int, char> letters = new Dictionary<int, char>(){ {1, 'a'}, {2, 'b'},
                                                             {3, 'c'}, {4, 'd'}};
// create and initialize a 4 element Dictionary of int keys and char values
Dictionary<string, string> names = new Dictionary<string, string>() { {"1", "Ann"}, {"2", "Ben"},
                                                                      {"3", "Cal"}, {"4", "Dee"}};
```

Note: This is a lot like initializing a 2D rectangular array.

Adding Elements to a Dictionary Object

- Inserting elements into a dictionary can be done using the *Add( )* method…

```
Dictionary<string, string> names = new Dictionary<string, string>();
names.Add("1", "Ann");
names.Add("2", "Ben");
names.Add("3", "Cal");
names.Add("4", "Dee");
```

- Elements can also be inserted into a dictionary is to use the indexer – the data type is the type specified by *TKey* parameter. The key value used has to be unique (not already used) – instead of generating an exception like an array would, a new value is inserted into the dictionary collection. If the key value is already used, instead of adding a new element, the existing element corresponding to the listed key is updated.

```
Dictionary<string, string> names = new Dictionary<string, string>();
// note that the index value - in this case - is not an int but a string
names["1"] = "Al";   // a key value of "1" doesn't exist - hence, a new element is added
names["2"] = "Ben";
names["3"] = "Cal";
names["4"] = "Dee";
names["1"] = "Ann"; // the value associated with the key "1" is updated since the key already exists
```

Common Dictionary<TKey, TValue> Properties & Methods

Name	Description	Example

`[key]`	Gets, sets, or creates the element with the specified key. The key data type is specified by the TKey type.	`Dictionary<int, char> letters = new Dictionary<int, char>(){{1,'a'},{2,'b'},{3,'c'}};`
		`char initial = letters[3]; // get value: initial = 'c'`
		`letters[3] = 'd'; // set value: {3,'d'}`
		`letters[10] = 'j'; // create element: {10,'j'}`

`Count`	Gets the number (as an *int*) of key/value pairs stored.	`Dictionary<int, char> letters = new Dictionary<int, char>(){{1,'a'},{2,'b'},{3,'c'}};` `int numLetters = letters.Count;  // numLetters = 3`
`Add``(key, value)`	Adds the specified key and value to the dictionary.	`Dictionary<int, char> letters = new Dictionary<int, char>(){{1,'a'},{2,'b'},{3,'c'}};` `letters.Add(10,'j'); // create element: {10,'j'}`

`Clear()`	Removes all elements from the dictionary and sets its Count property to zero.	`Dictionary<int, char> letters = new Dictionary<int, char>(){{1,'a'},{2,'b'},{3,'c'}};`
		`letters.Clear();`
		`int numLetters = letters.Count;  // numLetters = 0`

`ContainsKey``(key)`
`ContainsValue``(value)`
	Returns a Boolean value that indicates if the dictionary contains the specified key/value.	`Dictionary<int, char> letters = new Dictionary<int, char>(){{1,'a'},{2,'b'},{3,'c'}};`
		`bool hasKey = letters.ContainsKey(2);  // true`
		`bool hasValue = letters.ContainsValue('d');  // false`

`Remove``(key)`	Removes the value with the specified key from the Dictionary.	`Dictionary<int, char> letters = new Dictionary<int, char>(){{1,'a'},{2,'b'},{3,'c'}};` `letters.Remove(2); // delete element: {2,'b'}`

`TryGetValue``(key, out value)`	Gets the value associated with the specified key and returns *true* (*false* if key doesn't exist).	`Dictionary<int, char> letters = new Dictionary<int, char>(){{1,'a'},{2,'b'},{3,'c'}};`
		`char initial;`
		`if (letters.TryGetValue(2, out initial))`
		`  MessageBox.Show("Value: " + initial);`
		`else`
		`  MessageBox.Show("Value not present.");`

Using a Dictionary<TKey, TValue>

Example: Using a foreach loop to display the values of a Dictionary<>

- Since the indexer is not usually an integer sequence, using any loop other than a *foreach* loop most likely will cause an exception.
- Elements in a Dictionary<> are arranged into a hash table using hash codes for rapid retrieval – looping through a Dictionary<> accesses values in no particular order.
- Since a key and a value are required to add an element to the Dictionary<>, the data type required for a *foreach* loop is a **KeyValuePair<TKey, TValue>** structure.

```
Dictionary<int, char> letters = new Dictionary<int, char>(){ {1, 'a'}, {2, 'b'}, {3, 'c'}, {4, 'd'},
                                                             {5, 'e'}, {6, 'f'}, {7, 'g'}, {8, 'h'}};
Dictionary<string, string> names = new Dictionary<string, string>(){ {"1", "Ann"}, {"2", "Ben"},
                                                                     {"3", "Cal"}, {"4", "Dee"},
                                                                     {"6", "Flo"}, {"9", "Ira"}};

// loop through letters dictionary getting the key-value pairs
foreach (KeyValuePair<int, char> initial in letters)
{
    displayLetters += initial + "\n";
}
```

```
[1, Ann]
[2, Ben]
[3, Cal]
[4, Dee]
[6, Flo]
[9, Ira]
```

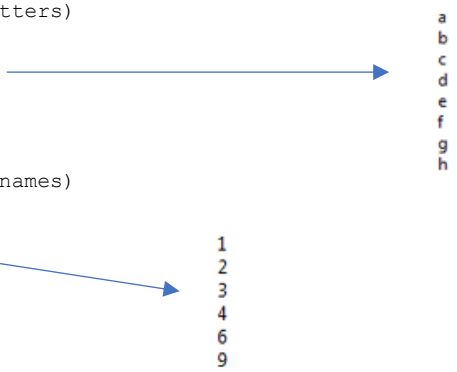
```
// loop through names dictionary getting the key-value pairs
foreach (KeyValuePair<string, string> name in names)
{
    displayNames += name + "\n";
}
```

```
[1, a]
[2, b]
[3, c]
[4, d]
[5, e]
[6, f]
[7, g]
[8, h]
```

- Accessing each element, as shown in the above code, results in a key-value pair.
- To access each key or value, use the properties associated with each Dictionary<> element
  - **Value**: returns the value of the Dictionary<> element
  - **Key**: returns the key of the Dictionary<> element

```
// loop through letters Dictionary<> getting the value
foreach (KeyValuePair<int, char> initial in letters)
{
    // use the Value property of each Dictionary<> element
    displayLetters += initial.Value + "\n";
}

// loop through names Dictionary<> getting the key
foreach (KeyValuePair<string, string> name in names)
{
    // use the Key property of each Dictionary<> element
    displayNames += name.Key + "\n";
}
```



Dictionary<TKey, TValue> Example

The following code example creates an empty Dictionary < > of strings with string keys and uses the Add method to add some elements. The example demonstrates that the Add method throws an ArgumentException when attempting to add a duplicate key.

The example uses the indexer to retrieve values, demonstrating that a KeyNotFoundException is thrown when a requested key is not present, and showing that the value associated with a key can be replaced.

The example shows how to use the TryGetValue method as a more efficient way to retrieve values if a program often must try key values that are not in the dictionary, and it shows how to use the ContainsKey method to test whether a key exists before calling the Add method.

The example shows how to enumerate the keys and values in the dictionary and how to enumerate the keys and values alone using the Keys property and the Values property.

Finally, the example demonstrates the Remove method.

```
Dictionary<string, string> programs = new Dictionary<string, string>();
string display = "";

// Add some elements to the dictionary. There are no duplicate keys, but some of the values are duplicates.
programs.Add("txt", "notepad.exe");
programs.Add("bmp", "paint.exe");
programs.Add("dib", "paint.exe");
programs.Add("rtf", "wordpad.exe");

// The Add method throws an exception if the new key is already in the dictionary.
try
{
    programs.Add("txt", "winword.exe");
}
catch (ArgumentException)
{
    display += "An element with Key = \"txt\" already exists.";
}

// The Item property is another name for the indexer, so you can omit its name when accessing elements.
display += "\nFor key = \"rtf\", value = " + programs["rtf"];

// The indexer can be used to change the value associated with a key.
programs["rtf"] = "winword.exe";
display += "\nFor key = \"rtf\", value = " + programs["rtf"];

// If a key does not exist, setting the indexer for that key adds a new key/value pair.
programs["doc"] = "winword.exe";
```

```
// The indexer throws an exception if the requested key is not in the dictionary.

try
{
    display += "\nFor key = \"tif\", value = " + programs["tif"];
}
catch (KeyNotFoundException)
{
    display += "\nKey = \"tif\" is not found.";
}

// When a program often has to try keys that turn out not to be in the dictionary,
// TryGetValue can be a more efficient way to retrieve values.

string value;
if (programs.TryGetValue("tif", out value))
{
    display += "\nFor key = \"tif\", value = " + value;
}
else
{
    display += "\nKey = \"tif\" is not found.";
}

// ContainsKey can be used to test keys before inserting them.

if (!programs.ContainsKey("ht"))
{
    programs.Add("ht", "hypertrm.exe");
    display += "\nValue added for key = \"ht\": " + programs["ht"];
}
// Output
```

```
An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe
For key = "rtf", value = winword.exe
Key = "tif" is not found.
Key = "tif" is not found.
Value added for key = "ht": hypertrm.exe
```

```
// When you use foreach to enumerate dictionary elements,
// the elements are retrieved as KeyValuePair objects.

display = "";
foreach (KeyValuePair<string, string> kvp in programs)
{
    display += "Key = " + kvp.Key + "   Value = " + kvp.Value + "\n";
}

// To get the values alone, use the Values property and assign to a ValueCollection class.
Dictionary<string, string>.ValueCollection valueColl = programs.Values;

// The elements of the ValueCollection are the type that was specified for dictionary values.
foreach (string s in valueColl)
{
    display += "\nValue = " + s;
}

// To get the keys alone, use the Keys property and assign to a KeyCollection class.
Dictionary<string, string>.KeyCollection keyColl = programs.Keys;

// The elements of the KeyCollection are the type that was specified for dictionary keys.
foreach (string s in keyColl)
{
    display += "\nKey = " + s;
}

// Use the Remove method to remove a key/value pair.

display += "\n\nRemove(\"doc\")";
programs.Remove("doc");

if (!programs.ContainsKey("doc"))
{
    display += "\nKey \"doc\" is not found.";
}
// Output
```

```
Key = txt   Value = notepad.exe
Key = bmp   Value = paint.exe
Key = dib   Value = paint.exe
Key = rtf   Value = winword.exe
Key = doc   Value = winword.exe
Key = ht   Value = hypertrm.exe

Value = notepad.exe
Value = paint.exe
Value = paint.exe
Value = winword.exe
Value = winword.exe
Value = hypertrm.exe
Key = txt
Key = bmp
Key = dib
Key = rtf
Key = doc
Key = ht

Remove("doc")
Key "doc" is not found.
```