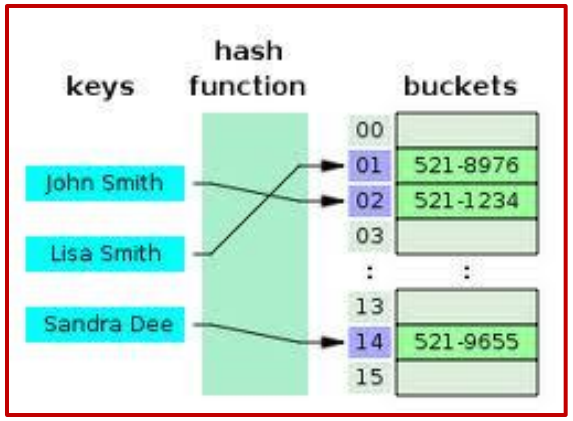
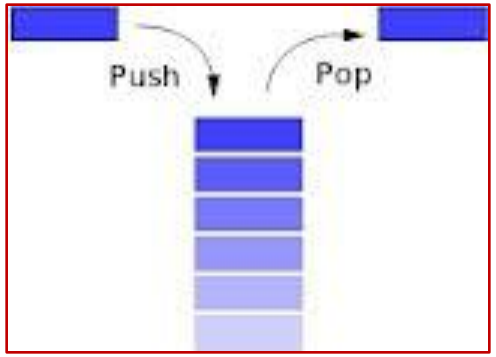
Collections and Generics

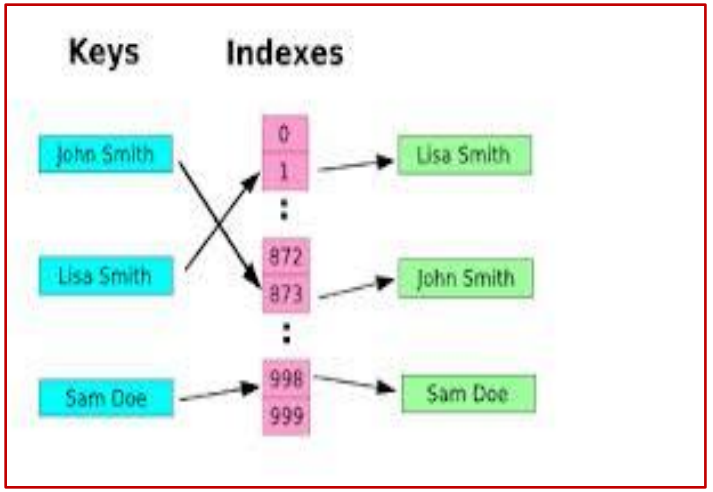
By Pakita Shamoi, Spring 2018

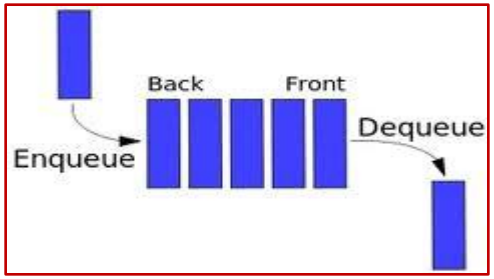
Intro

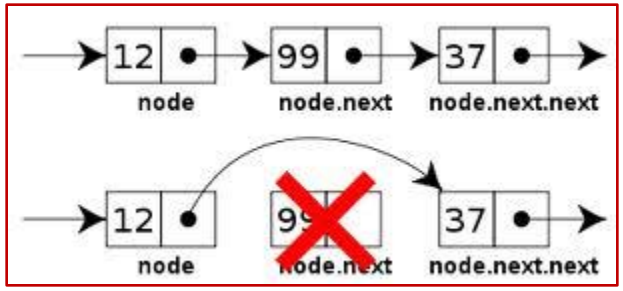
- Collections are classes that support the gathering of information in an orderly way.
- Your challenge will be to discern the right collection to use in a specific case.

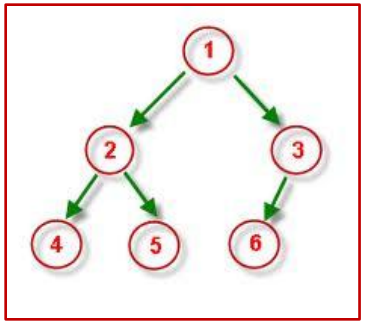












Name	Description
ArrayList	A simple resizeable, index-based collection of objects
SortedList	A sorted collection of name/value pairs of objects
Queue	A first-in, first-out collection of objects
Stack	A last-in, first-out collection of objects
Hashtable	A collection of name/value pairs of objects that allows retrieval by name or index
BitArray	A compact collection of *Boolean* values
StringCollection	A simple resizeable collection of strings
StringDictionary	A collection of name/values pairs of strings that allows retrieval by name or index
ListDictionary	An efficient collection to store small lists of objects
HybridDictionary	A collection that uses a *ListDictionary* for storage when the number of items in the collection is small, and then migrates the items to a *Hashtable* for large collections
NameValueCollection	A collection of name/values pairs of strings that allows retrieval by name or index

ArrayList – Add, AddRange

- Two methods for adding items to the collection: ***Add*** *and* ***AddRange.***
- The ***Add*** method allows you to add a single object to the collection.
- You can use the Add method to store any object in .NET.

```
ArrayList coll = new ArrayList();
coll.Add("hi");
coll.Add(50);//Boxing or Unboxing?
coll.Add(new object());

string[] anArray = new string[] { "more", "or", "less" };
coll.AddRange(anArray);
```

ArrayList – Insert, InsertRange, Indexers

- To insert the objects at specific positions, use **Insert** and **InsertRange** methods:
  - `coll.Insert(3, "Hey All");`
- You can also use indexers (it overwrites the old object):
  - `coll[3] = "Hey All"`

Removing elements

- 3 methods support removing items: ***Remove***, ***RemoveAt***, *and* ***RemoveRange***
  - coll.Remove("Hello"); // no exception if there is no “Hello”
  - coll.RemoveAt(0);
  - coll.RemoveRange(0, 4); // removes 4 elements

ArrayList -Additional methods

- **Clear** method is used to empty a collection
- **IndexOf** method is used to determine the index of a particular item in the collection.
- **Contains** method is used to test whether a particular object exists in the collection.

```
string myString = "My String";
if (coll.Contains(myString))
{
int index = coll.IndexOf(myString);
coll.RemoveAt(index);
}
else
{
coll.Clear();
}
```

Iterating over items

- Numeric indexer

```
for (int x = 0; x < coll.Count; ++x)
{
Console.WriteLine(coll[x]);}
```

  - Using **IEnumerator** interface

```
IEnumerator enumerator = coll.GetEnumerator();
while (enumerator.MoveNext())
{
Console.WriteLine(enumerator.Current);}
```

  - foreach // advantage?

```
foreach (object item in coll)
{
Console.WriteLine(item);
}
```

Interfaces in Collections

- ***ICollection*** - common interface for how a collection's application programming interface (API) should look.
- It derives from the ***IEnumerable*** interface.
- We also have ***IList*** interface , which is used to expose lists of items (derives from ***ICollection***)
  - We've already covered *ArrayList* class. It implements the *IList* interface

- *Why we need that?*
  - For simplification. It ensures that every collection supports a common way of getting the items in a collection. (Count, CopyTo, etc.).

Sorting Items

- coll.Sort();
- ***IComparer** interface. This* interface dictates that you implement a single method called ***Compare*** that takes two objects (for example, a and b) and returns an integer that represents the result of the comparison (0, >0, <0).
  - To perform case-insensitive sorting, use:
    - coll.Sort(new CaseInsensitiveComparer());

HW : *Write your own comparer that implements the IComparer . Your comparer must be used to perform sorting in descending order :*

- coll.Sort(new DescendingComparer());

Sequential lists

- The Queue and Stack classes are meant to be used to store data in a sequential basis.

- The Queue class is a collection for dealing with first-in, first-out (FIFO) handling of sequential objects.
- In contrast to the Queue class, the Stack class is a last-in, first-out (LIFO) collection

Queue

Name	Description
Dequeue	Retrieves an item from the front of the queue, removing it at the same time
Enqueue	Adds an item to the end of the queue
Peek	Retrieves the first item from the queue without actually removing it

```
Queue q = new Queue();
q.Enqueue("An item");
Console.WriteLine(q.Dequeue());
```

```
if (q.Peek() is String)
{
Console.WriteLine(q.Dequeue());
}
```

Stack

Name	Description
Pop	Retrieves an item from the top of the stack, removing it at the same time
Push	Adds an item to the top of the stack
Peek	Retrieves the top item from the stack without removing it

```
Stack s = new Stack();
s.Push("An item");
Console.WriteLine(s.Pop());
```

Dictionaries

- Dictionaries are collections that store lists of **key/value pairs** to allow lookup of values based on a key.
- In the most basic case, the **Hashtable** class is used to do this mapping of key/value pairs.
  - For example, assume that you need to map e-mail addresses to the full name of some user.

```
Hashtable emailLookup = new Hashtable();
emailLookup.Add("jonh@gmail.com", "John Scott");

// The indexer is functionally equivalent to Add
emailLookup["jonh@gmail.com"] = "John Scott";
Console.WriteLine(emailLookup["jonh@gmail.com"]);
```

Iterating over the dictionary

- A **DictionaryEntry** object is simply a container containing a *Key* and a *Value*.
- So getting the values out is as simple as doing the iteration, but you must retrieve the *Value* or *Key* as necessary for your needs.

```
foreach (DictionaryEntry entry in emailLookup)
{
Console.WriteLine(entry.Value);
}
```

```
foreach (object name in emailLookup.Values)
{
Console.WriteLine(name);
}
```

IDictionary interface

Name	Description
Keys	Gets an *ICollection* object containing a list of the keys in the collection
Values	Gets an *ICollection* object containing a list of the values in the collection

Name	Description
Add	Adds a key/value pair to the collection.
Clear	Removes all items in the collections.
Contains	Tests whether a specific key is contained in the collection.
GetEnumerator	Returns an *IDictionaryEnumerator* object for the collection. This method is different than the *IEnumerable* interface that returns an *IEnumerator* interface.
Remove	Removes the item in the collection that corresponds to a specific key.

Table 4-15 ***Hashtable*** **Methods**

Name	Description
ContainsKey	Determines whether the collection contains a specific key
ContainsValue	Determines whether the collection contains a specific value

Equality

- The Hashtable class uses an integer value (called a ***hash***) to aid in the storage of its keys.
- The ***hash*** speeds up the searching for a specific key in the collection.
- Every object in .NET derives from the **Object** class
- This class supports the GetHashCode method, which returns an integer (hash) that uniquely identifies the object.
- The Hashtable allows only unique hashes of keys, if you try to store the same key twice, the second call replaces the first call

This is how the Hashtable class tests for equality, by testing the hash code of the objects

```
public class Fish {
    string name;
    public Fish(string theName){
        name = theName;
    }
    public override bool Equals(object obj)
    {
        Fish otherFish = obj as Fish;
        if (otherFish == null) return false;
        return otherFish.name == name;
    }
    public override int GetHashCode()
    {
        return name.GetHashCode();
    }
}
class Program {
    static void Main(string[] args){
        Hashtable duplicates = new Hashtable();
        Fish key1 = new Fish("Herring");
        Fish key2 = new Fish("Herring");
        duplicates[key1] = "Hello";
        duplicates[key2] = "Hello";
        Console.WriteLine(duplicates.Count); // 2
        Console.ReadKey();
    }
```

- if we create two instance of the **Fish** with the same name, the **Hashtable** treats them as different objects
- This is because the Object class's implementation of **GetHashCode** creates a hash that is likely to be unique for each instance of a class
- Override the **GetHashCode** in the **Fish** class to try and let the **Hashtable** know they are equal
- **Object.Equals** will return false if the two objects are two different instances of the same class. So we need to also add an override of the **Equals** method

SortedList

- Dictionary class that shares some of its behavior with how simple lists work.
- This means that you can access items stored in the **SortedList** in order - *string v = mySortedList.Values[3];*

```
SortedList sort = new SortedList();
sort["First"] = "1st";
sort["Second"] = "2nd";
sort["Third"] = "3rd";
sort["Fourth"] = "4th";
foreach (DictionaryEntry entry in sort){
Console.WriteLine("{0} = {1}", entry.Key, entry.Value);
}

SortedList<string,string> openWith = new SortedList<string,string>();
openWith.Add("txt", "notepad.exe"); openWith.Add("bmp", "paint.exe");
foreach (KeyValuePair<string, string> kvp in mySortedList) {...}
```

Specialized Dictionaries

- Sometimes standard dictionaries (*SortedList* and *Hashtable*) have limitations, either functional limitations or performance-related ones.

- To bridge that gap, the .NET Framework supports three other dictionaries:
  - *ListDictionary*
  - *HybridDictionary*
  - *OrderedDictionary.*

ListDictionary

- Hashtable class requires a bit of overhead, and for small collections (fewer than ten elements) the overhead can impede performance.
- That is where the ListDictionary comes in.
- It implements the same interface as the Hashtable class, so it can be used as a drop-in replacement (none of the code is different except for the construction of the object)

HybridDictionary

- If you know your collection is small, use a ListDictionary
- If your collection is large, use a Hashtable.

But what if you just do not know how large your collection is?

- That is where the *HybridDictionary* comes in.
- It is implemented as a **ListDictionary** and only when the list becomes too large does it convert itself into a **Hashtable** internally.

OrderedDictionary

- Suitable for cases when you want the functionality of the Hashtable (fast dictionary) but you need to control the order of the elements in the collection.
  - So, it supports *access by index*

Name	Description
Insert	Inserts a key/value pair at a specific index in the collection
RemoveAt	Removes a key/value pair at a specific index in the collection

Summary

- The .NET Framework supports a variety of collection classes that can be used in different circumstances.
- The ArrayList is a simple collection of unordered items.
- The .NET Framework supports the Queue and Stack classes to provide collections that represent sequential lists of items.
- The IDictionary interface provides the basic calling convention for all Dictionary collections.
- The Hashtable class can be used to create lookup tables.
- You can use a DictionaryEntry object to get at the key and value of an object in a Dictionary collection.
- The SortedList can be used to create list of items that can be sorted by a key.

Specialized and Generic Collections

- We were introduced a series of collections that can be used to store any object in .NET.
- Although these are valuable tools, using them can often lead to you having to cast objects when you retrieve them from the collections.
- The .NET Framework supports a new namespace called *System.Collections.Specialized*
- That includes collections that are meant to work with specific types of data.

Working with bits

- Often you need to have a list of bits that can be either on or off.
- *BitArray and BitVector32* simplify working with collections of bits

- The *BitArray* class is a resizeable collection that can store Boolean values.
  - It also supports common bit-level operations such as and, not, or, and exclusive-or (Xor).

BitArray

- When you create a new instance of the *BitArray* class, you must specify the size of the collection.
  - Once the new instance has been created, you can change the size by changing the **Length** property.

- Unlike other collections, BitArray does not support Add or Remove.

```
BitArray bits = new BitArray(3);
bits[0] = false;
```

- You can perform Boolean operations on two BitArray objects (of the same size).

Example

```
BitArray bits = new BitArray(3);
bits[0] = false;
bits[1] = true;
bits[2] = false;
BitArray moreBits = new BitArray(3);
moreBits[0] = true;
moreBits[1] = true;
moreBits[2] = false;
BitArray xorBits = bits.Xor(moreBits);
foreach (bool bit in xorBits)
{
    Console.WriteLine(bit);
}
```

StringCollection

- Two specialized collections that are strongly typed to store strings: *StringCollection* and *StringDictionary.*
- StringCollection
  - dynamically sized collection
  - working with it is virtually identical to using an *ArrayList*

```
StringCollection coll = new StringCollection();
coll.Add("First");
coll.Add("Second");
string theString = coll[3];
// No longer need to
// string theString = (string) coll[3];
```

StringDictionary

- Use it just like a Hashtable, except that both the keys and values must be strings

```
StringDictionary dict = new StringDictionary();
dict["First"] = "1st";
dict["Second"] = "2nd";
dict["Third"] = "3rd";
dict["Fourth"] = "4th";
dict["fourth"] = "fourth";
string converted = dict["Second"];
// No casting needed
```

Case-Insensitive Collections

- Creating case-insensitive dictionary collections is a common need
- the .NET Framework has a CollectionUtil class that supports creating Hashtable and SortedList objects that are case insensitive.

```
Hashtable inTable =
CollectionsUtil.CreateCaseInsensitiveHashtable();
inTable["hello"] = "Hi";
inTable["HELLO"] = "Heya";
Console.WriteLine(inTable.Count); // 1
```

NameValueCollection

- You can think that *NVC* and a *StringDictionary* are similar because both allow you to add keys and values that are strings.

- However, there are some specific differences:
  - it allows multiple values per key
  - values can be retrieved by index as well as key.

- To retrieve all the values for some key, use *GetValues*

Example

```
NameValueCollection nv = new NameValueCollection();
nv.Add("Key", "Some Text");
nv.Add("Key", "More Text");
foreach (string s in nv.GetValues("Key"))
{
Console.WriteLine(s);
}
```

Interesting note

- if you add values with the indexer and with the Add method, you can see the difference in behavior
- It works for *NameValueCollection* and for other dictionaries as well

```
nv["First"] = "1st";
nv["First"] = "FIRST";
nv.Add("Second", "2nd");
nv.Add("Second", "SECOND");
Console.WriteLine(nv.GetValues("First").Length);
// 1
Console.WriteLine(nv.GetValues("Second").Length);
// 2
```

NameValueCollection

- Finally, the other difference between using the NameValueCollection and the StringDictionary is that you can retrieve items by key index.

```
NameValueCollection nv = new NameValueCollection();
nv.Add("First", "1st");
nv.Add("Second", "2nd");
nv.Add("Second", "Not First");
for (int x = 0; x < nv.Count; ++x)
{
Console.WriteLine(nv[x]);
}
// 1st
// 2nd,Not First
```

Generic collections

- In general, it is not type safe to use usual collections, like *ArrayList*.

- But it would be great to **just specify in the class what type you want to use**. Luckily, you can do it with **generic types**.

- When you use this generic class, you simply have to include the generic parameter (the name of a type) in the creation of the type. (e.g. **<string>)**

Equivalent generic classes

Type	Generic Type
ArrayList	*List<>*
Queue	*Queue<>*
Stack	*Stack<>*
Hashtable	*Dictionary<>*
SortedList	*SortedList<>*
ListDictionary	*Dictionary<>*
HybridDictionary	*Dictionary<>*
OrderedDictionary	*Dictionary<>*
SortedDictionary	*SortedDictionary<>*
NameValueCollection	*Dictionary<>*
DictionaryEntry	*NameValuePair<>*
StringCollection	*List<String>*
StringDictionary	*Dictionary<String>*
N/A	*LinkedList<>*

List

- The generic List class is used to create simple type-safe ordered lists of objects.
  - You can use indexers, foreach, add…

```
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);
```

Queue, Stack

- For both of these classes the code is the same as for non-generic ones, the only difference is in specifying type.
  - ```
    Queue<String> que = new Queue<String>();
    que.Enqueue("Hello");
    String queued = que.Dequeue();
    ```
- The same story for Stack
    ```

Dictionary

- The generic *Dictionary* class most closely resembles the Hashtable, ListDictionary, and HybridDictionary classes.
- The difference is in a need to specyfy the type of the key and the type of the values to store in the *Dictionary.*

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[3] = "Three";
dict[4] = "Four";
String str = dict[3];
foreach (KeyValuePair<int, string> i in dict)
{
Console.WriteLine("{0} = {1}", i.Key, i.Value);
}
```

SortedDictionary

- *SortedDictionary* class is like the generic *Dictionary* class, with the exception that it maintains its items sorted by the key of the collection.

```
SortedDictionary<string, int> sortedDict =
new SortedDictionary<string, int>();
sortedDict["One"] = 1;
sortedDict["Two"] = 2;
sortedDict["Three"] = 3;
foreach (KeyValuePair<string, int> i in sortedDict)
{
Console.WriteLine(i);
}
```