Data Structures

C#.NET has a lot of different **data structures**, for example, one of the most common ones is an Array. However C# comes with many more basic data structures. Choosing the correct data structure to use is part of writing a well structured and efficient program.

ArrayList

The C# data structure, `ArrayList`, is a dynamic array. What that means is an `ArrayList` can have any amount of objects and of any type. This data structure was designed to simplify the processes of adding new elements into an array. Under the hood, an ArrayList is an array whose size is *doubled* every time it runs out of space. Doubling the size of the internal array is a very effective strategy that reduces the amount of element-copying in the long run. We won't get into the proof of that here. The data structure is very simple to use:

```
ArrayList myArrayList = new ArrayList();
myArrayList.Add(56);
myArrayList.Add("String");
myArrayList.Add(new Form());
```

The downside to the `ArrayList` data structure is one must cast the retrieved values back into their original type:

```
int arrayListValue = (int)myArrayList[0];
```

List<>

The `List` C# data structure was introduced in the .NET Framework 2.0 as part of the new set of generic collections.`List` is the generic version of ArrayList, which means that it behaves exactly the same way, but within a specified *type*. So for example, a List of integers would work as follows:

```
List<int> intList = new List<int>();
intList.Add(45);
intList.Add(34);
```

Since the `List<>` object is tailored to a specific data type, there is no need to cast when retrieving values:

```
int listValue = intList[0];
```

This results in much cleaner, and in my times, faster code. This is especially true when working with primative types. Unless you are using the .NET Framework 1.1, you should always use `List` over ArrayList. (Note that `List<Object>` is perfectly legal, although it defeats the purpose of having a generic dynamic array collection).

Dictionary<>

The Dictionary **C# data structure** is extremely useful data structure since it allows the programmer to handle the index keys. What does that mean? Well an `ArrayList` automatically makes its "keys" integers that go up one by one, 1, 2, etc, so to access a value in an ArrayList one goes like: `myArrayList[2]`;

So what the C# Dictionary data structure does is let us specify the keys, which can be any type of object. For example:

```
Dictionary<string, int> myDictionary = new Dictionary<string, int>();
myDictionary.Add("one", 1);
myDictionary.Add("twenty", 20);
```

Retrieving a value is pretty straight forward:

```
int myInt = myDictionary["one"];
```

Notice how convenient the Dictionary data structure is, in that there is no need to cast between types. Also there is nothing stopping you from creating a Dictionary like so:

```
Dictionary<int, Dictionary<string, int>> nestedDictionary =
            new Dictionary<int, Dictionary<string, int>>();
```

That is a nested Dictionary C# data structure and it is fair game.

I understand that it can be confusing on how to go about getting all the values out of a Dictionary data structure since we have no way to knowing the pattern in the keys. Luckily we don't have to, here is the code to transverse a C#.Net Dictionary:

```
//List<[same type as index]>
List<string> keyList = new List<string>(myDictionary.Keys);
for (int i = 0; i < keyList.Count; i++)
{
    int myInt = myDictionary[keyList[i]];

}
```

You can read about C# Dictionary in a little more detail if you are interested.

Hashtable

The C# Hashtable data structure is very much like the Dictionary data structure. A Hashtable also takes in a key/value pair, but it does so as generic objects as opposed to typed data.

Values are then stored in order according to their key's `HashCode`. Meaning that the order in which items are added to a C# `Hashtable` is not preserved. On the other hand, the `Dictionary` data structure *does* keep items in the same order.

The reason is speed. A C# Hashtable stores items faster than a C# Dictionary, which sacrifices speed for the sake of order..

(For those Java programmers, a `Dictionary` is more or less a `TreeMap` and a `Hashtable` is a `HashMap`).

```
Hashtable myTable = new Hashtable();
```

Stack

The `Stack` class is one of the many C# data structures that resembles an `List`. Like an `List`, a stack has an add and get method, with a slight difference in behavior.

To add to a stack data structure, you need to use the `Push` call, which is the `Add` equivalent of an `List`. Retrieving a value is slightly different. The stack has a `Pop` call, which returns and **removes** the last object added. If you want to check the top value in a Stack, use the `Peek` call.

The resulting behavior is what is called LIFO, which stands for Last-In-First-Out. This particular data structure is helpful when you need to retrace your steps so to speak.

There are two formats to define a `Stack` in C#:

```
Stack stack = new Stack();
Stack<string> stack = new Stack<string>();
```

The different between the data structures being that the simple `Stack` structure will work with `Objects` while the`Stack<>` one will accept only a specified object.

Here is the C# code to add and traverse through a `Stack` data structure:

```
Stack<string> stack = new Stack<string>();
stack.Push("1");
stack.Push("2");
stack.Push("3");

while (stack.Count > 0)
{
    MessageBox.Show(stack.Pop());
}
```

If you run the C# code you see that the list is returned in the order: 3, 2, 1.

Queue

Another one of the many C# data structures is the Queue. A Queue is very similar to the Stack data structure with one major difference.

Rather than follow a LIFO behavior, a Queue data structure goes by FIFO, which stands for First-In-First-Out. Whenever you submit an article to be approved on a website for example, the site adds your submission to a queue. That way the first objects added are the first ones to be processed.

The Add call for a queue (or the Push version) is Enqueue:

```
queue.Enqueue("1");
```

The Remove call is Dequeue:

```
queue.Dequeue();
```

Similarly the `Peek` call allows you to view the top value without removing it. This specific data structure is very often used in conjuction with stack data structures.

Here is some simple C# code to add items to a queue and the transverse it:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("1");
queue.Enqueue("2");
queue.Enqueue("3");

while (queue.Count > 0)
{
   MessageBox.Show(queue.Dequeue());
}
```

Also keep in mind the queue data structure can be defined as a general Queue and as a type-specific `Queue<>`