Arrays and Collections

Arrays and other collection types are reference types that contain other objects. With arrays, every item must be the same data type but, unlike the simple arrays from C or C++, arrays in .NET benefit from inheritance. All arrays derive from `System.Array` , which gives them some useful features such as the `IClonable` , `IEnumerable,` `ICollection` , and `IList` interfaces. Possibly the most exciting feature that .NET brings to the table is array index validation, which ensures that we don't access memory that has not been allocated to the array.

Such buffer overflows are a common source for security issues and exploits, but with .NET, instead of a security vulnerability to exploit, you get an `IndexOutOfRange-Exception.` Behind the scenes, the framework is checking to confirm that the index is valid with every call. Some may worry about the performance overhead of these index validation tests, although they needn't. While it *is* possible to skip this validation by compiling with the "unsafe" option, this is rarely a good option, and the overhead of doing the index validations is minimal compared to the risks of having buffer overflows.

Before generics were introduced in.NET 2.0, `ArrayLists` were the most widely used collection type object, and they have a couple of differences from the humble array. Most notably, the `ArrayList` does not require that all elements have the same data type. This can lead to logic errors if you assume that every element will be of the same type. Instead, all we can assume is that every element will be of type `Object` , and since `Object` is the ultimate base type for *every* type, this could be anything, and so anything can go into an `ArrayList.` The problem is that this is not type safe. Another problem is that value types must be boxed to be loaded into the `ArrayList` , and unboxed to be used. Not to mention, even reference types must be explicitly type cast back to the expected data type.

The release of .NET 2.0 introduced the generically-typed `List<T>` , which eliminated both of these problems.

```
var list = new ArrayList
{
 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, "Something" , "strange", true

};

foreach (object number in list)
{
 Console.WriteLine(number);
}
```

Listing 4.52: Looping through an `ArrayList.`

The code in Listing 4.52 compiles and runs, even though it may not make sense and will probably not give the expected result.

`List<T>` is type-safe and more performant, and should replace `ArrayList` in most cases. The switch will generally be fairly straightforward, as in Listing 4.53.

```
var list = new List<int>
{
 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
};

foreach (int number in list)
{
 Console.WriteLine(number);
}
```

Listing 4.53: Looping through a strongly typed `List<int>`.

Of course, if you actually wanted to mix and match data types, you would be stuck with `ArrayList`, but that is rarely the case.

For many developers, the greatest appeal was how easily you could add items to the array list. A common pattern using the `List<T>` was as shown in Listing 4.54.

```
private List<string> Tokenize (string data)
{
  var returnValue = new List<string>();
  string[] tokens = data.Split(new char[] { ':' });
  foreach (string item in tokens)
  {
    returnValue.Add(item);
  }
  return returnValue;
}
```

Listing 4.54: A common pattern loading items into a generic `List<T>`.

This and similar patterns are very common and very intuitive, but they will also introduce memory problems. Behind the scenes there is still a simple array, and resizing these arrays can be an expensive operation. It is easy to forget about this overhead, but it is still there.

From Reflector, the implementation for the `Add` method is shown in Listing 4.55.

```
public void Add(T item)
{
  if (this._size == this._items.Length)
  {
    this.EnsureCapacity(this._size + 1);
  }
  this._items[this._size++] = item;
  this._version++;
}
```

Listing 4.55: Internal implementation of the `Add` method.

The dangerous part is the `EnsureCapacity` method shown in Listing 4.56.

```
private void EnsureCapacity(int min)
{
  if (this._items.Length < min)
  {
    int num = (this._items.Length == 0)? 4
    : (this._items.Length * 2);
    if (num < min)
    {
     num = min;
    }
    this.Capacity = num;
  }
 }
```

Listing 4.56: The `EnsureCapacity` method means that memory doubles when more is needed.

This method means that if there are no elements in the array, it will initialize it to 4 elements. If there *are* elements in the array but more are needed, we will double the size of the array. Of course, as a user of the `List<T>`, you are completely oblivious to this doubling, because the resizing logic happens in the `Capacity` property, which has two significant implications. Firstly, if we do not properly initialize our `List`, we will need to perform several resizing operations as the size grows. This means that, when you have a small list, you will spend more time resizing arrays than anything else. The second implication is that, with large arrays, we may actually have twice as much memory allocated as we would expect. As soon as you hit one of the power of 2 boundaries, the memory allocated will be doubled, which will cause your program to use more memory than you would expect.

It is important to not accept the default constructor for `List` and its variations. Instead, use the overloaded constructor where you can specify the initial capacity. This will reduce the number of initial resizing operations that are needed. If you pick the initial size well, you may be able to entirely eliminate the need for resizing operations. Just be careful, because if you guess too low, then as soon as you exceed your estimate the memory requirements of your code will double.

Using overloaded constructors, the `Tokenize` method that we saw earlier could be rewritten as in Listing 4.57.

```
private List<string> Tokenize(string data)
{
 string[] tokens = data.Split(new char[] { ‘:’ });
 var returnValue = new List<string>(tokens.Count());
 returnValue.AddRange(tokens);
 return returnValue;
}
```

Listing 4.57: A memory sensitive optimization of the original `Tokenize` method.

Or even as in Listing 4.58.

```
private List<string> Tokenize(string data)
{
 string[] tokens = data.Split(new char[] { ‘:’ });
 var returnValue = new List<string>(tokens );
 return returnValue;
}
```

Listing 4.58: Another memory sensitive optimization.

Either of these rewrites will result in the internal array being perfectly sized for the data in it.

Be suspicious of any code that does not specify an initial size for any `List<T>` or `ArrayList` objects. There is a very good chance that you will be using more memory than you anticipate, and more memory than you should.