.Net Containers

Container Types in .NET

IEnumerable

Contents

- What is a container?
- Arrays
- .NET Containers
- For each loops
- Sorting Containers

What is a Container?

- A container is a group of related objects
  - Containers are also known as Collections
- They make it easy to create and manage multiple objects of similar type at the same time
- Containers can dynamically grow and shrink in size to fit the requirements of the program

C# Containers

- C# has a limited set of containers by itself
- However, .NET has many
- All languages that are built on top of .NET have access to these various container types

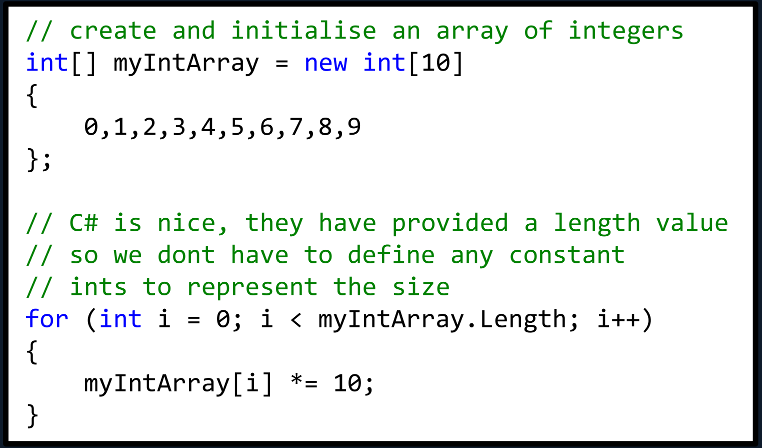
C# Arrays

- Arrays are part of C#
  - There are a few different types of arrays in C#
    - 1D Array
    - 2D / Multidimensional array
    - Jagged Array

- Once the size of an array has been specified, it cannot be resized

C# 1D Arrays

- Pretty straightforward, a fixed sized array of 10 items

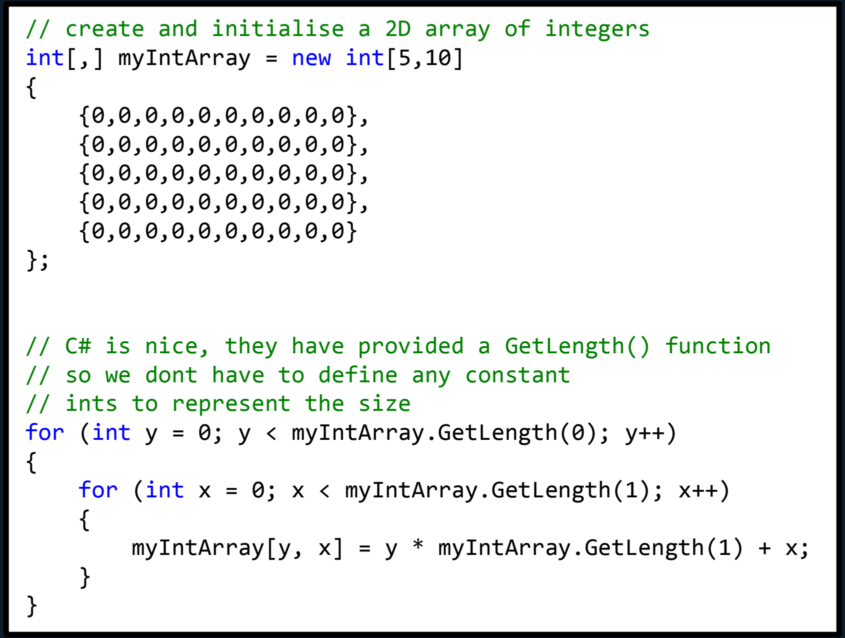


```
// create and initialise an array of integers
int[] myIntArray = new int[10]
{
    0,1,2,3,4,5,6,7,8,9
};

// C# is nice, they have provided a length value
// so we dont have to define any constant
// ints to represent the size
for (int i = 0; i < myIntArray.Length; i++)
{
    myIntArray[i] *= 10;
}
```

C# 2D Arrays

- 2D arrays are simple – picture a grid of items (specify the number of rows and columns)

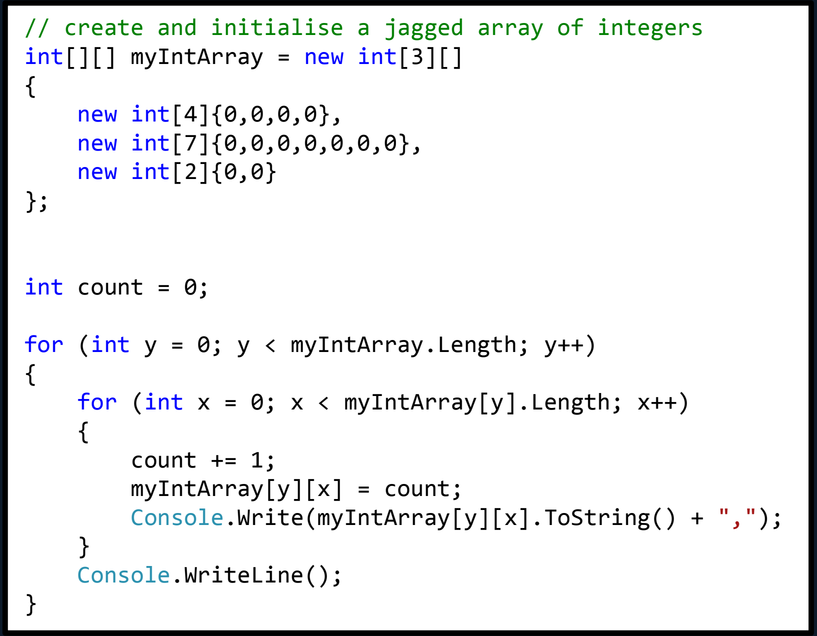
- 3D and beyond arrays have a similar syntax (though it is hard to visualize 4D and above)



```
// create and initialise a 2D array of integers
int[,] myIntArray = new int[5,10]
{
    {0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0}
};


// C# is nice, they have provided a GetLength() function
// so we dont have to define any constant
// ints to represent the size
for (int y = 0; y < myIntArray.GetLength(0); y++)
{
    for (int x = 0; x < myIntArray.GetLength(1); x++)
    {
        myIntArray[y, x] = y * myIntArray.GetLength(1) + x;
    }
}
```

C# Jagged Arrays

- Think of this as an array of arrays – each sub array can have its own length



```
// create and initialise a jagged array of integers
int[][] myIntArray = new int[3][]
{
    new int[4]{0,0,0,0},
    new int[7]{0,0,0,0,0,0,0},
    new int[2]{0,0}
};


int count = 0;

for (int y = 0; y < myIntArray.Length; y++)
{
    for (int x = 0; x < myIntArray[y].Length; x++)
    {
        count += 1;
        myIntArray[y][x] = count;
        Console.Write(myIntArray[y][x].ToString() + ",");
    }
    Console.WriteLine();
}
```

But I want more control!

- If arrays aren’t your thing, there are many useful container types in the .NET framework

- These can be found in:
  - System.Collections
  - System.Collections.Generic
  - System.Collections.ObjectModel
  - System.Collections.Specialized

Tip:
Take a look at what's available in the Object Browser
View->ObjectBrowser (ctrl + alt + j)

You will find your standard container types in mscorelib, a few more in System and some in System.Core

.NET Containers

- Standard container types, you'll use these most often
  - List<type>
  - LinkedList<type>
  - Dictionary<key,value>

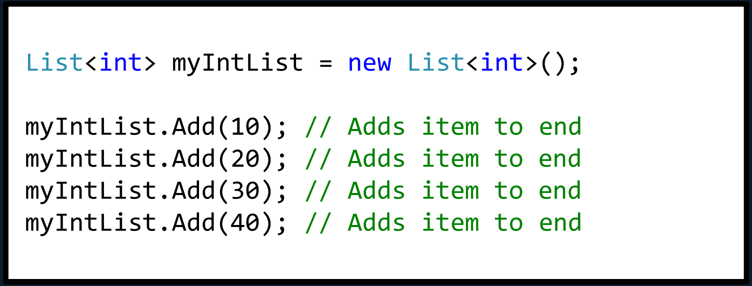
- These are comparable to the C++ STL Containers:
  - std::vector<type>
  - std::list<type>
  - std::unordered_map<key,value>

List<type>

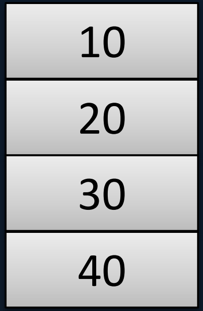
- List<type> provides a continuous chunk of memory that grows and shrinks, similar to an std::vector

- Provides random access with O(1) complexity to elements within the collection

- However, it is expensive to insert and remove from the middle



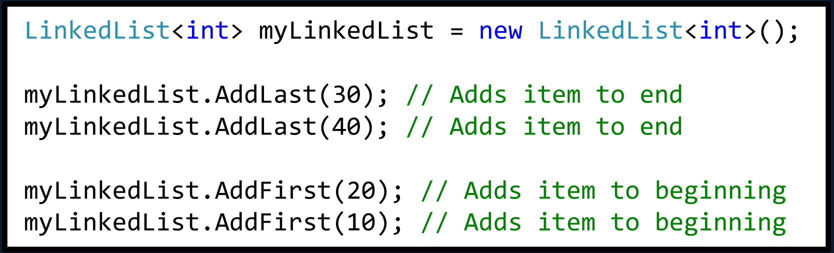
```
List<int> myIntList = new List<int>();

myIntList.Add(10); // Adds item to end
myIntList.Add(20); // Adds item to end
myIntList.Add(30); // Adds item to end
myIntList.Add(40); // Adds item to end
```



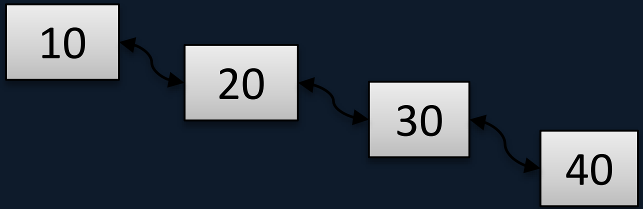
10
20
30
40

LinkedList<type>

- LinkedList<type> provides a double linked list data type, just like the std::list
- Does not provide random access to items. You can iterate through using foreach
  - O(n) complexity
- Cheap to insert and remove items



```
LinkedList<int> myLinkedList = new LinkedList<int>();

myLinkedList.AddLast(30); // Adds item to end
myLinkedList.AddLast(40); // Adds item to end

myLinkedList.AddFirst(20); // Adds item to beginning
myLinkedList.AddFirst(10); // Adds item to beginning
```

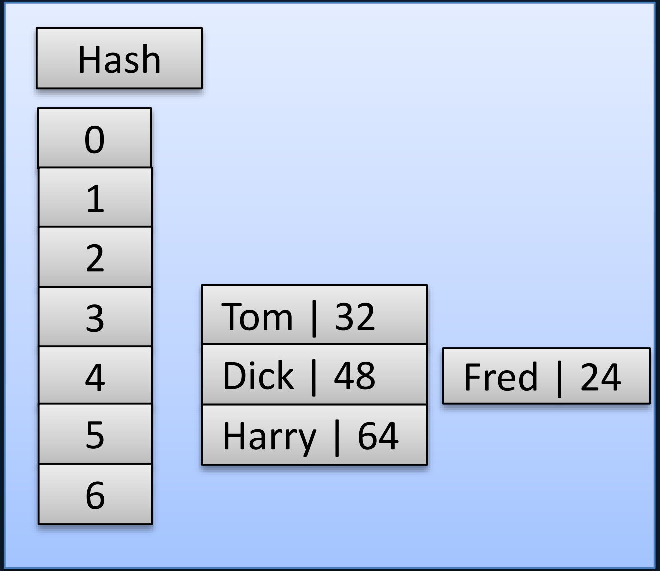


Dictionary<key,value>

- Dictionary<key,value> implements a hash map, just like std::unordered_map in C++11 or std::hash_map in C++99

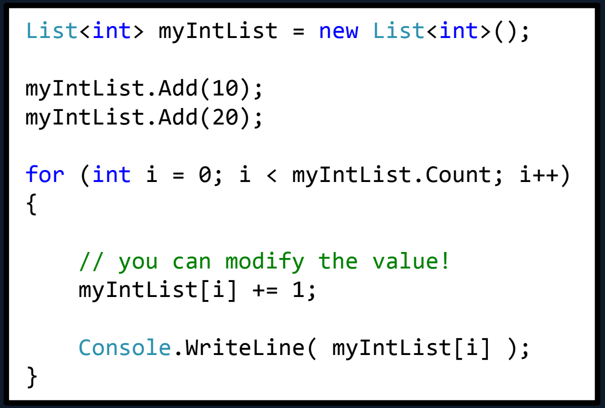
NOTE: Not to be confused with std::map which implements a red/black tree as its underlining data storage.

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

// can add items like this
playerAges.Add(“Tom", 32);
playerAges.Add(“Dick", 48);
playerAges.Add(“Harry", 64);

// or like this
playerAges[“Fred"] = 24;
```

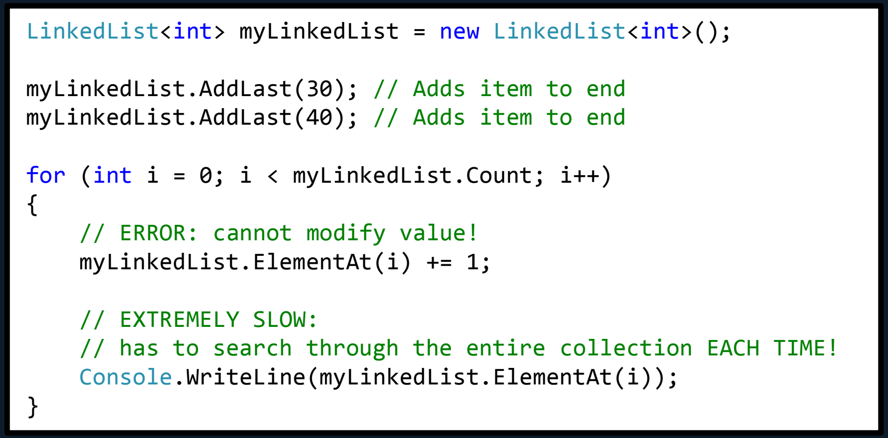


Iterating through collections – for loop

- As a list has random access, you can use a for loop
- A linked list does not, therefore you must use a foreach instead



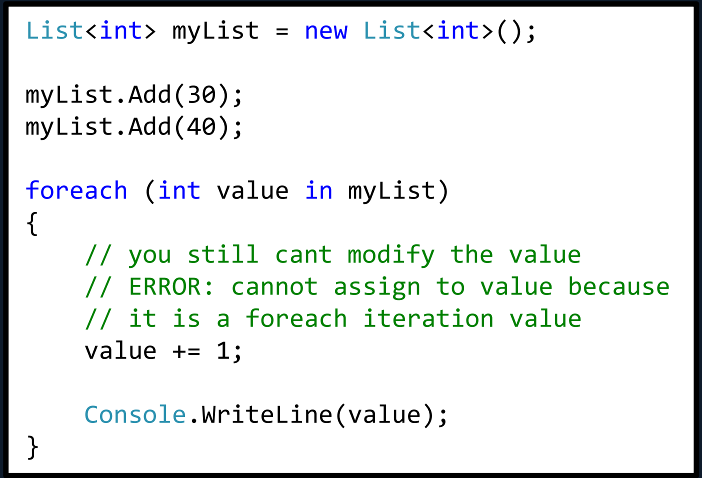
```
List<int> myIntList = new List<int>();

myIntList.Add(10);
myIntList.Add(20);

for (int i = 0; i < myIntList.Count; i++)
{

    // you can modify the value!
    myIntList[i] += 1;

    Console.WriteLine( myIntList[i] );
}
```



```
LinkedList<int> myLinkedList = new LinkedList<int>();

myLinkedList.AddLast(30); // Adds item to end
myLinkedList.AddLast(40); // Adds item to end

for (int i = 0; i < myLinkedList.Count; i++)
{
    // ERROR: cannot modify value!
    myLinkedList.ElementAt(i) += 1;

    // EXTREMELY SLOW:
    // has to search through the entire collection EACH TIME!
    Console.WriteLine(myLinkedList.ElementAt(i));
}
```

Iterating through collections - foreach

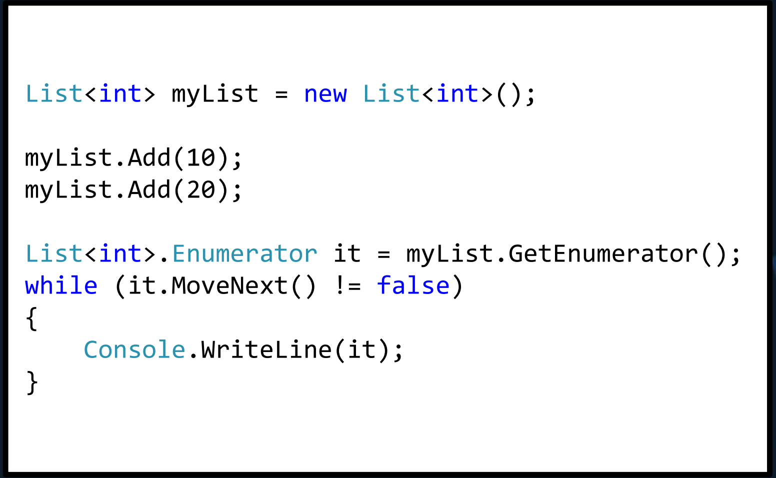
- .NET collections don’t use iterators the way C++ does
- They all implement IEnumerable<T> and can be iterated over using the foreach statement
- Restrictions:
  - You cannot modify the list (add, insert or remove)
  - You cannot modify the value



```
List<int> myList = new List<int>();

myList.Add(30);
myList.Add(40);

foreach (int value in myList)
{
    // you still cant modify the value
    // ERROR: cannot assign to value because
    // it is a foreach iteration value
    value += 1;

    Console.WriteLine(value);
}
```

Iterating through collections - Enumerator

- If you want to manually control iteration, you can call GetEnumerator() on the collection which will return an IEnumerator<T> object.
- IEnumerator<T>.MoveNext() is roughly equivalent to ++ on a C++ iterator, and Current is roughly equivalent to the pointer-dereference operator (*iter)
- Restrictions:
  - Cannot modify the list (add, insert or remove)
  - Cannot modify the value



```
List<int> myList = new List<int>();

myList.Add(10);
myList.Add(20);

List<int>.Enumerator it = myList.GetEnumerator();
while (it.MoveNext() != false)
{
    Console.WriteLine(it);
}
```

Value vs Reference

- Recap: structures are a value type, classes are a reference type
  - Ints, floats and the other default primitive types are value types too

Reference Type B = Reference Type A

Reference Type B now refers to the same memory as Reference Type A (like assigning pointers)

Value Type B = Value Type A

Value Type B is a copy of Value Type A

```
class CFoo
{ public int val; }

CFoo foo1 = new CFoo();
CFoo foo2 = foo1;

foo1.val = 10;    foo2.val = 20;

Console.WriteLine(foo1.val); // prints 20
Console.WriteLine(foo2.val); // prints 20
```

```
struct SFoo
{ public int val; }

SFoo foo1 = new SFoo();
SFoo foo2 = foo1;

foo1.val = 10;    foo2.val = 20;

Console.WriteLine(foo1.val); // prints 10
Console.WriteLine(foo2.val); // prints 20
```

Canberra Institute of Technology

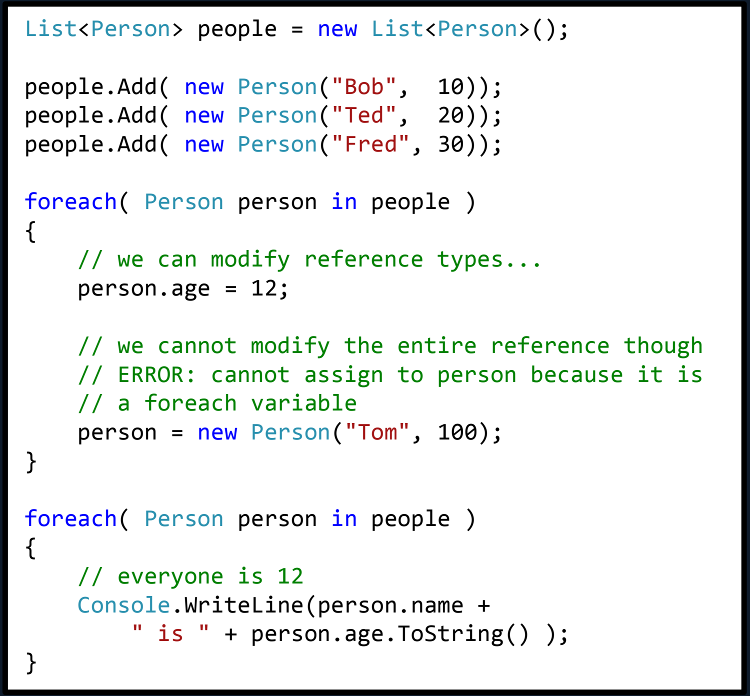
aie

SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

List of Value types

- A list of value types doesn’t help much when iterating through with a foreach loop, foreach loops are pretty much read only

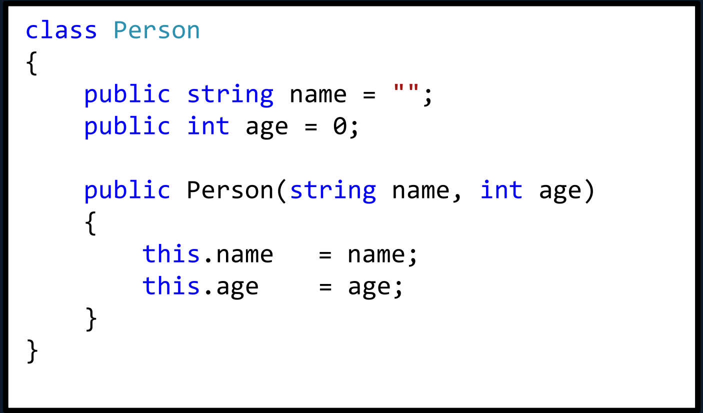
- Rather than using value types, you could store a list of referene types



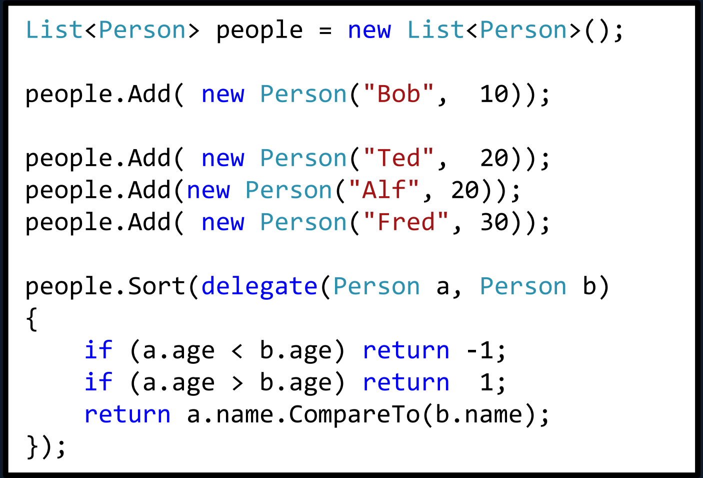
```
List<Person> people = new List<Person>();

people.Add( new Person("Bob",  10));
people.Add( new Person("Ted",  20));
people.Add( new Person("Fred", 30));

foreach( Person person in people )
{
    // we can modify reference types...
    person.age = 12;

    // we cannot modify the entire reference though
    // ERROR: cannot assign to person because it is
    // a foreach variable
    person = new Person("Tom", 100);
}

foreach( Person person in people )
{
    // everyone is 12
    Console.WriteLine(person.name +
        " is " + person.age.ToString() );
}
```

Sorting

- Some of the container types provide a sort method
- For the sort function to work, it needs to be able to compare objects
  - There are a couple of methods of achieving this
- Method 1: Provide an anonymous function
  - Return -1 if a < b
  - Return +1 if a > b
  - Return 0 if they are the same



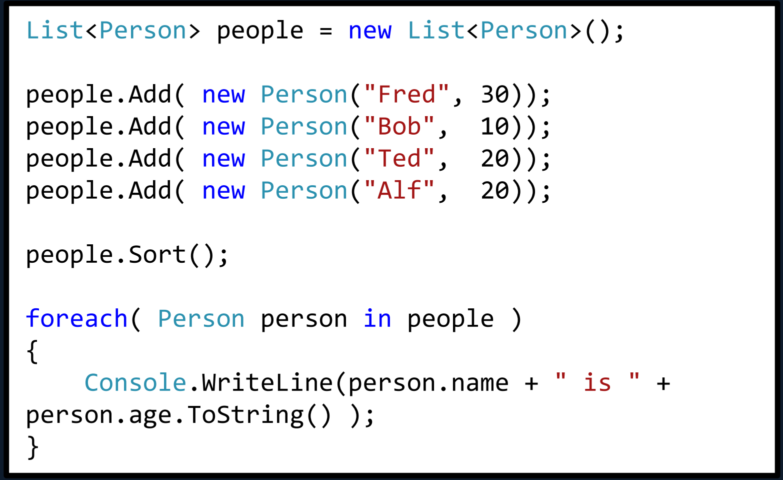
```
class Person
{
    public string name = "";
    public int age = 0;

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



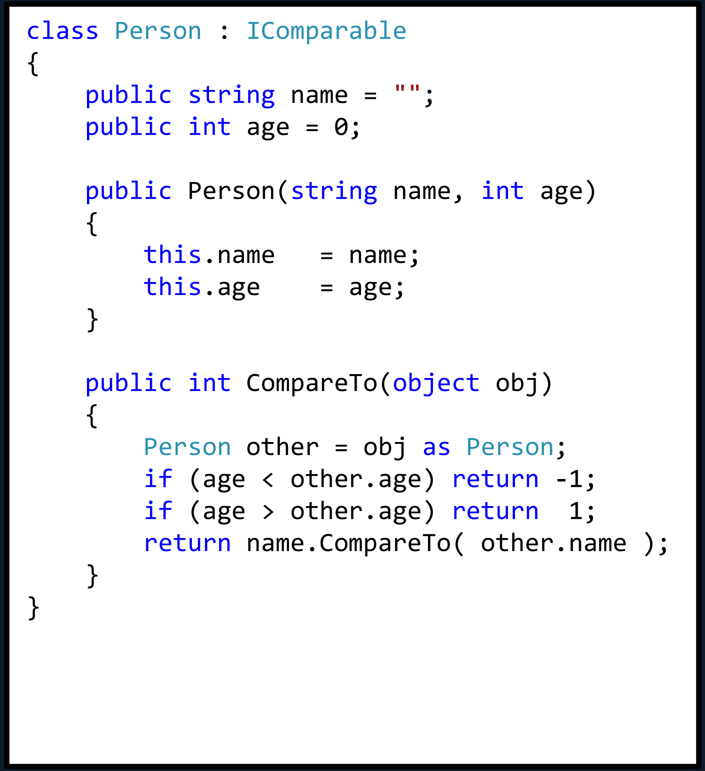
```
List<Person> people = new List<Person>();

people.Add( new Person("Bob",  10));

people.Add( new Person("Ted",  20));
people.Add(new Person("Alf", 20));
people.Add( new Person("Fred", 30));

people.Sort(delegate(Person a, Person b)
{
    if (a.age < b.age) return -1;
    if (a.age > b.age) return  1;
    return a.name.CompareTo(b.name);
});
```

Sorting

- Method 2: have your type inherit from IComparable



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

people.Add( new Person("Fred", 30));
people.Add( new Person("Bob",  10));
people.Add( new Person("Ted",  20));
people.Add( new Person("Alf",  20));

people.Sort();

foreach( Person person in people )
{
    Console.WriteLine(person.name + " is " +
person.age.ToString() );
}
```



```
class Person : IComparable
{
    public string name = "";
    public int age = 0;

    public Person(string name, int age)
    {
        this.name   = name;
        this.age    = age;
    }

    public int CompareTo(object obj)
    {
        Person other = obj as Person;
        if (age < other.age) return -1;
        if (age > other.age) return  1;
        return name.CompareTo( other.name );
    }
}
```

Summary

- There are a lot of different container types, and plenty more stuff to lookup!

- We looked at the following:
  - List, LinkedList, Dictionary
  - Foreach loops
  - Enumerators
  - Values vs References
  - Sorting lists

References

- Microsoft, 2014, *Collections*
  - https://msdn.microsoft.com/en-us/library/ybcx56wz.aspx

- .NETPerls, 2014, *C# Collections*
  - http://www.dotnetperls.com/collections