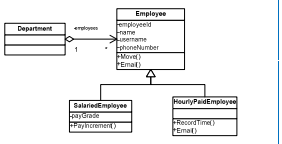
Object Oriented Software Development

6. Arrays and collections

One-to-many relationships

- A department can have many employees in it
- One-to-many “has-a” relationship



- “has-a” relationship is implemented with instance variable
- This variable must be able to hold a **collection** of Employee objects

Collections of data

- Most software applications need to work with collections of similar objects, for example:
  - List of transactions to display in a bank application
  - List of sprites in a game
- Often need to perform actions such as:
  - Listing all the objects in a collection
  - Adding and removing objects
  - Finding a specific object in a collection

Arrays

- Simple way of storing collection of objects of the same type

```
int[] myIntegers = new int[10];
myIntegers[0] = 42;

char[] myChars = { 'H', 'e', 'l', 'l', 'o' };
```

- Contents can be any type, including reference types and value types
- Arrays are stored on the heap
- If content type is a reference type, the array stores references to objects

Looping through an array

- **for** and **foreach** loops:

```
for (int i = 0; i < myIntegers.Length; i++)
{
    myIntegers[i] = i;
}

Console.Write("Array contents: ");
foreach (int myInt in myIntegers)
{
    Console.Write("{0} ", myInt);
}
```

- **foreach** loop works with any collection which implements an interface called **IEnumerable**
- We will look at this again later

Arrays and polymorphism

- An array can store instances of its declared type or subclasses of that type

```
Employee[] employees = new Employee[10];
HourlyPaidEmployee emp1 = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
SalariedEmployee emp2 = new SalariedEmployee(2, "Susan", "susan",
    loc, "5678", 6);
employees[0] = emp1;
employees[1] = emp2;

SalariedEmployee semp = (SalariedEmployee)employees[1] ;
int grade = semp.PayGrade;
```

- May need to cast to get object back out as its original type

More than one dimension

- Arrays can have two dimensions

```
int[,] numbers = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };
string[,] siblings = new string[2, 2] {
    { "Mike", "Amy" },
    { "Mary", "Albert" }
    };
```

- Can have more than two dimensions
- Can be rectangular or jagged
  - Jagged array = array of arrays
  - Arrays may be different sizes

Limitations of arrays

- Once created, the size of an array is fixed
- Array does not provide ways of adding, removing or finding elements
- To use an array you need to write code to access elements
- Often better to use classes which are designed to store data and provide methods for accessing the data
- Data structures, or collections

Abstract data types

- An ADT is a specification for the set of operations which can be performed on the data stored in a data structure
- **"Abstract"** because the specification is independent of the concrete implementation
- Could have many implementations of the same ADT which work differently "under the hood"
- We will look first at the **list** ADT and two ways of implementing it

List

- A list is a sequential data structure
- Addition and removals can be made at any position in the list
- Lists are normally in the form of $a_1,a_2,a_3.....a_n$. The size of this list is n.
- The first element of the list is a_1,and the last element is a_n
- The position of element a_i in a list is i.

List operations

- A list data structure should be able to provide the following operations:
  - Create a list
  - Find an item in the list
  - Insert an item in the list
  - Modify an item in the list
  - Delete an item from the list
  - Check whether list is full/empty
  - Traverse the list

Lists example

- **CollectionsDemo solution**
- **CollectionsDemo and SimpleCollections projects**
  - Data structures in separate project from program – can easily be reused by other programs
  - SimpleCollections is a **class library** project
- CollectionsDemo – **Program.cs**
- SimpleCollections – **ISimpleList.cs, SimpleArrayList.cs, SimpleLinkedList.cs**

Implementing a list

- We can express these requirements as an interface
- Lists will implement this interface

```
public interface ISimpleList
{
    int Size
    {
        get;
    }

    bool IsFull();

    bool IsEmpty();

    void Add(int index, Object o);

    void Add(Object o);

    bool Remove(Object o);

    int IndexOf(Object o);

    Object Get(int index);

    Object Set(int index, Object o);
}
```


Data type of items

- Note that in interface code, the item type is Object
- Polymorphism – each list item can be:
  - An instance of Object
  - An instance of any type which derives from object...
  - ...which is essentially any type at all as all types in .NET derive from Object
- Reusable – can be used to store any type of items in any application

Implementation details

- There are two common ways of actually storing the data in a list
- ArrayList
  - Items are stored in an array
- LinkedList
  - Items are stored as a chain of connected nodes
  - Each node has a data item and a reference to the next node

Does it matter which one?

- Both versions can be written to implement the interface, and both will work perfectly well
- In fact, we don't even need to write our own implementations as the .NET Framework library already has a wide selection of tried and test collection classes, including lists
- But, different implementations have different performance in particular situations
- Need to understand characteristics of structures to make the best choice

Simple implementations

- We will look at the code for simplified implementations of ArrayList and LinkedList
- This will help us to understand how the two versions work and identify where there might be performance differences
- Wouldn't use these in real programs – Framework versions are more complex, but more capable and robust
- Use Framework classes unless you need specific custom functionality

SimpleArrayList

- Items stored in array of type Object
- Capacity is the number of items which can be stored
- Size is the number of items currently stored

```
public class SimpleArrayList : ISimpleList
{
    private Object[] entries;
    private int capacity;
    private int size;

    public SimpleArrayList()
    {
        entries = new Object[10];
        this.capacity = 10;
        size = 0;
    }
```

instance variables

constructor

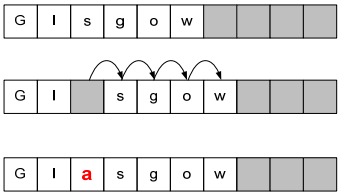
Adding items to SimpleArrayList

- Adding a new item at the end of the list is simple
- Number of items stored so far = *size*
- Index of last stored = *size - 1*
- Store the new item at index = *size*

```
public void Add(Object o)
{
    if (!IsFull())
    {
        entries[size] = o;
        size++;
    }
}
```


Adding items to SimpleArrayList

- Adding a new item at a specified index involves more work
- Need to shuffle items up to make space



Adding items to SimpleArrayList

- Start at first empty position (size)
- Copy previous entry into current one
- Move down one position
- Copy previous entry into current one
- Repeat until target position reached
- Put new item in target position

```
public void Add(int index, Object o)
{
    if (index > size)
        index = size;
    if (!IsFull())
    {
        for (int i = size; i > index; i--)
            entries[i] = entries[i - 1];
        entries[index] = o;
        size++;
    }
}
```

SimpleLinkedList

- Each item is stored in an object of type Node
- Contains item and reference to next Node
- List only needs reference to first Node, can find others from there by following references
- No capacity limit, list can grow

Node class

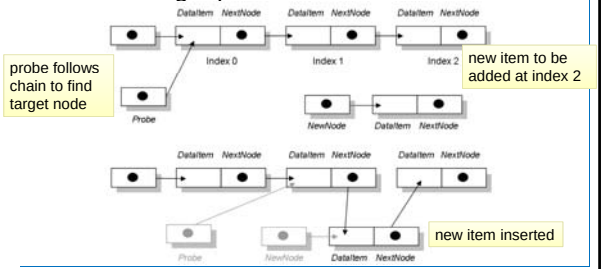
```
public class Node
{
    public Object dataItem;
    public Node nextNode;
}
```

```
public class SimpleLinkedList : ISimpleList
{
    private Node head;
```

reference to head node, initially set to null by constructor

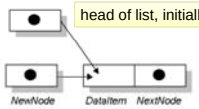
Adding items to SimpleLinkedList

- To add a new item at a specified index break chain at target position and insert new node



Adding items to SimpleLinkedList

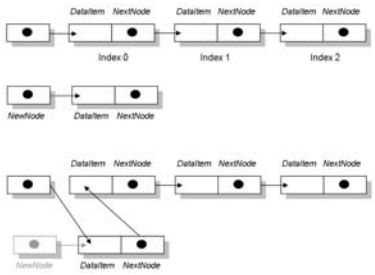
- Special case when inserting first item in empty list
  - Simply set head to point to new node



- Adding at end is not a particularly special case
  - Need to probe all the way along the list to find last item then insert new node

Adding items to SimpleLinkedList

- Special case when inserting at start of list



Adding items to SimpleLinkedList

- Implementation for general case

```
public void Add(int index, Object o)
{
    Node newNode = new Node();
    Node probe;

    newNode.dataItem = o;

    probe = head;
    int i = 0;
    while ((i < index - 1) && (probe.nextNode != null))
    {
        probe = probe.nextNode;
        i++;
    }
    newNode.nextNode = probe.nextNode;
    probe.nextNode = newNode;
```


Other operations

- How would deleting be implemented in each case?
- How would searching be implemented?
- How would modifying be implemented?
- Look at sample code to see how other operations are implemented

Which list should we use?

- Need to judge likely effect of list characteristics on performance
  - Real applications may store a large number of data
  - Consider the way the data will be added and accessed
- Will new items be added frequently?
- Where in the list will they be added?
- Will we need to access items frequently?
- Where in the list will we need to access items?

ArrayList

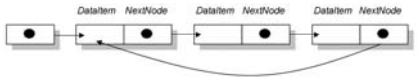
- Time to access does not depend on the size of the list
- To add an element at the end of the list, the time taken does not depend on the size
- Time taken to add an element at any other point in the list does depend on the size
- Additions near the start of the list take longer than additions near the middle or end
- Deletes near the start of the list take longer

LinkedList

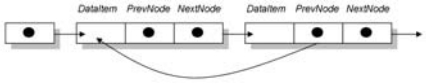
- Time to access an element depends on the index because each element of the list must be traversed until the required index is found
- Time to add an element does not depend on the size of the list, as no shifts are required
- Time to add does depend on the index
- Additions near the end of the list take longer than additions near the middle or start
- Deletes near the end of the list take longer

Variations on LinkedList

- Circularly linked list
  - The tail of the list always points to the head
  - Can start anywhere and reach whole list



- Doubly linked list
  - Permits searching in both directions



Capacity of lists

- LinkedList is **dynamic**
  - No limit on capacity
- ArrayList is **static**
  - Capacity defined by length of underlying array
  - Could be a big disadvantage, but...
- .NET Framework implementation of ArrayList can resize itself dynamically
- When array gets close to being full the ArrayList creates a new, larger array and copies items into it

Traversing a collection

- Traversing means passing through the collection visiting each item in turn
- The details of how this works depend on the way the collection is implemented
- Traversing an ArrayList simply involves passing along the array
- Traversing a LinkedList involves following references to reach each node
- Collection should provide a way of traversing

foreach loop and IEnumerable

- In C# the **foreach-in** loop is the usual way to traverse a collection
- Collection needs to provide help for the foreach loop to work
- Collection must implement **IEnumerable** interface

IEnumerable interface

- Defines a contract
- If a collection type implements IEnumerable interface then it **agrees** to provide the mechanism which can **enumerate**, or loop though, the collection
- Defines one method – **GetEnumerator**, which returns an instance of a class which in turn implements **IEnumerator**
- Most .NET collection types and arrays implement this interface

Enumerators

- The **IEnumerator** interface defines an **enumerator** object which traverses a collection in a way specific to that collection
- Enumerator object can be used by foreach loop or can be used directly in your code
- Defines:
  - **MoveNext** method
  - **Reset** method
  - **Current** property
- See example in lab exercises

More data structures

- Lists are suitable for many applications
- However, there are other data structures which have characteristics which make them more suited to specific applications, including:
  - Stack
  - Queue
  - Dictionary

Stack ADT

- A stack is a special case of a list
- Addition and removals can only be made at one end
- Last-in, first-out (LIFO)
- **Push** - adds a new item onto the stack
- **Pop** – removes and returns last item added
- Can be implemented with array or linked list

Stack applications

- Page visited history in a Web browser
- Undo sequence in a text editor
- Reversing a string
- Method calls – memory stack
- Balancing symbols – e.g. check for matching brackets in a C# code file

Queue ADT

- A queue is also special case of a list
- Addition can only be made at the tail and removals made at the head
- First-in, first-out (FIFO)
- **Enqueue** - adds a new item onto the queue
- **Dequeue** – removes and returns first item added
- Can be implemented with array or linked list

Queue applications

- Waiting lists
- Message queues
- Access to shared resources, e.g. printer
- Buffers for transferring data asynchronously e.g., file IO, sockets

Dictionary ADT

- A **dictionary** is a searchable collection of **key-item** pairs
- Also known as **map**, **associative list**
- Keys are used to locate items
- Collection can be sorted or unsorted
- May or may not allow more than one item with the same key – depends on implementation

Dictionary operations

- **Put**(key, item) - add a new key/item pair
- **Get**(key) – get the item with the specified key
- **Set**(key, item) – set the item for the specified key
- **Remove**(key) – remove the item with the specified key
- **ContainsKey**(key) – check whether the specified key exists

Dictionary data types

- Most reusable case is when key and item data types are both Object
- Can store any type and use any type as key because of polymorphism
- Sometimes it is convenient to have a customised implementation restricted to specific data types
  - e.g. .NET Framework includes a StringDictionary class

Boxing/unboxing with collections

- Collections often have item type Object
- Don't need to create a separate collection class for each data type which might be stored
- A consequence of this is that value types are converted to object references (boxed) when stored in such a collection

```
SimpleArrayList sal = new SimpleArrayList();
sal.Add(3);
int i = (int)sal.Get(0);
```

boxing

unboxing – cast to value type

Disadvantages of boxing

- Overhead of creating object references affects performance
- Need to write code to explicitly cast to value type
- Not **type-safe** – can add the "wrong" type of objects into the collection, compiler will not pick this up

Generics

- Generic collection classes give the best of both worlds
  - One class allows storage of any data type
  - Can specify what type it is to hold
  - Type-safe – compiler will not allow code which stores wrong type

```
SimpleGenericArrayList<int> sgal =
    new SimpleGenericArrayList<int>();
sgal.Add(3);
int j = sgal.Get(0);
SimpleGenericArrayList<Employee> sgal2 =
    new SimpleGenericArrayList<Employee>();
```

same collection class used for different types

no boxing needed

no unboxing, no cast needed

Generic list example

- **CollectionsDemo solution**
- **SimpleCollections project**
- **SimpleGenericArrayList.cs**

Creating a generic collection class

- Declare placeholder for generic type – T
  - Placeholder will be replaced with specific type when application is compiled
  - References to T instead of Object

```
public class SimpleGenericArrayList<T>
{
    private T[] entries;
    private int capacity;
    private int size;

SimpleGenericArrayList<Employee> sga =
    new SimpleGenericArrayList<Employee>();
```

generic type

becomes **Employee[] entries** when compiled

specific type

Generic collection class members

```
public SimpleGenericArrayList()
{
    entries = new T[10];
    this.capacity = 10;
    MakeEmpty();
}

public void Add(int index, T o)
{
    if (index > size)
        index = size;
    if (!IsFull())
    {
        for (int i = size; i > index; i--)
            entries[i] = entries[i - 1];
        entries[index] = o;
        size++;
    }
}
```

references to T instead of Object

Generic collection class members

- Can't generally set a reference of generic type to **null**
- Type at compile time might be a value type which can't be null
- Use **default** instead – null for reference type, default value for value type

```
public T Get(int index)
{
    if (IsEmpty())
        return default(T);
    else
        return entries[index];
}
```

Framework collections

- The .NET framework provides a range of collections in three namespaces
- System.Collections
- System.Collections.Generic
  - generic versions – names not always consistent with non-generic variants, e.g. ArrayList is equivalent to List<T>
- System.Collections.Specialized
  - a range of collections with specialized characteristics

Framework collections

- Use framework collection where possible instead of writing your own
- Consider characteristics and likely impact on performance when choosing
- Use generic collections where possible
- Only create custom collections to match very specific, unusual requirements

Framework collections example

- **FrameworkCollectionsDemo solution**
- **ListGenericDemo project**
- **Program.cs**
- **DictionaryGenericDemo project**
- **Program.cs**

Using List<T>

- Create

```
List<DateTime> aList = new List<DateTime>();
```

- Add

```
aList.Add(new DateTime(1968, 4, 5));
```

- Get value

```
DateTime dt = aList[1];
```

- Iterate

```
foreach (DateTime dt in aList)
{
    Console.WriteLine("{0}", dt.ToShortDateString());
}
```


Using Dictionary<T,U>

- Create

```
Dictionary<string, double> dict = new Dictionary<string, double>();
```

- Add

```
dict.Add("Chai", 18.0);
```

- Get value

```
double val = dict["Chai"];
```

Note that .NET implementation uses **Add** rather than Put, and array-like index notation for lookup rather than Get

Using Dictionary<T,U>

- Iterate
  - Iterate through Dictionary as **KeyValuePair** collection

could also use **implicit type**: `var entry in dict`

```
foreach (KeyValuePair<string,double> entry in dict)
{
    Console.WriteLine("{0} price: {1:c}", entry.Key, entry.Value);
}
```

  - Get dictionary values from **Values** property(can get **Keys** similarly)

```
foreach (double value in dict.Values)
{
    Console.WriteLine("{0:c}", value);
}
```

Further reading

- There are other collection types
- Many sources of information online
- Recommended:
  - An Extensive Examination of Data Structures using C# 2.0 (MSDN)
  - http://msdn.microsoft.com/en-us/library/ms364091%28v=VS.80%29.aspx

Further reading

- The code in the CollectionsDemo download contains example implementations of the following collection types:
  - Stack
  - Queue
  - Dictionary – see extra slides for information on ways to implement a Dictionary
- Implementation details of these will not be in the exam

What's next?

- We will go on to look in detail at how to implement a UML model using C# classes, and how collections can help with this