7 Generic Collections and how to Serialize them

Introduction

This chapter will start by introducing the reader to generic methods it will then go on to introduce the reader to an essential part of the .NET framework :- the classes that implement generic collections. Finally it will introduce the idea of serialization and show how different collections can be serialised as this is a very common task.

Objectives

By the end of this chapter you will be able to….

- Understand the concept of Generic Methods
- Understand the concepts of Collections and the different types of Collection
- Understand the concept of Serialiazation and understand how to serialise the different collections.

This chapter consists of twelve sections :-

1) An Introduction to Generic Methods
2) An Introduction to Collections
3) Different Types of Collections
4) Lists
5) HashSets
6) Dictionaries
7) A Simple List Example
8) A More Realistic Example Using Lists
9) An Example Using Sets
10) An example Using Dictionaries
11) Serializing and De-serializing Collections
12) Summary

7.1 An Introduction to Generic Methods

We have seen previously how methods are identified at run time by their signature i.e. the name of the method and the list of parameters the method takes.

Thus we can have two methods with the same name. Shown below are two methods that find a highest value… one finds the highest value given two integer numbers, the other finds the highest value of two double numbers.

Download free eBooks at bookboon.com

```
static public int Highest(int o1, int o2)
{
     if (o1 > o2)
          return o1;
     else
          return o2;
}
static public double Highest(double o1, double o2)
{
     if (o1 > o2)
          return o1;
     else
          return o2;
}
```

Given the following code the CLR engine will invoke the first of these methods then the second as the correct method to implement is identified by its name and the type of its parameters.

```
int n1 = 1;
int n2 = 2;
Console.WriteLine("The highest is " + Highest(n1, n2));

double n3 = 1.1;
double n4 = 2.2;
Console.WriteLine("The highest is " + Highest(n3, n4));
```

Given the following definition of a Student class…

```
class Student
{
     String name;
     public Student(String name)
     {
          this.name = name;
     }
     override public string ToString()
     {
          return name;
     }
}
```

Download free eBooks at bookboon.com

We could even define a version of this method to find the highest student (alphabetically).

```
static public Student Highest(Student o1, Student o2)
{
     if (o1.ToString().CompareTo(o2.ToString()) > 0)
          return o1;
     else
          return o2;
}
```

And invoke this using ..

```
Student s1 = new Student("Alan");
Student s2 = new Student("Clare");
Console.WriteLine("The highest is " + Highest(s1, s2));
```

In doing this we have created three methods that essentially do exactly the same thing only using different parameter types. This leads us to the idea that it would be nice to create just one method that would work with objects of any type.

The method below is a first attempt to do this :-

```
static public Object Highest(Object o1, Object o2)
{
     if (o1.ToString().CompareTo(o2.ToString()) > 0)
          return o1;
     else
          return o2;
}
```

This method takes any two 'Objects' as parameters. 'Object' is the base class of all other classes thus this method can take any two objects of any more specific type and treat these as of the base type 'Object'. This is polymorphism.

The method above then converts these two 'Object's to strings and compares these strings.

Thus this one method can be invoked three times using the following code….

```
int n1 = 1;
int n2 = 2;
Console.WriteLine("The highest is " + Highest(n1, n2));

double n3 = 1.1;
double n4 = 2.2;
Console.WriteLine("The highest is " + Highest(n3, n4));

Student s1 = new Student("Student A");
Student s2 = new Student("Student B");
Console.WriteLine("The highest is " + Highest(s1, s2));
```

Download free eBooks at bookboon.com

In these cases respectively the CLR engine will treat int, double, and Student objects as the most general type of thing available 'Object'. It will then convert this object to a string and compare the strings.

This will work however in many situations creating methods which take parameters of type 'Object' is flawed or at least very limited.

Inside these methods we do not know what type of object was actually passed as a parameter and hence in the example above we do not know what type of object is actually being returned. When two students object are passed the object returned is a student but we cannot invoke Student specific methods on this object unless we first cast the object returned to a Student.

Download free eBooks at bookboon.com
Click on the ad to read more

Assume that we want to invoke an 'AwardMerit() method on the 'Student' returned via the Highest() method. We can do this is we first cast the returned 'Object' onto a 'Student' object. E.g.

```
(Student)Highest(studentA,studentB).AwardMerit();
```

However the compiler cannot be certain that the returned object is a 'Student' and the compiler cannot detect the potentially critical error that would occur is we invoked Highest() on two integer numbers and then tried to cast the returning integer onto an object of type 'Student'.

This leads us to the idea that we would like to be able to create a method that will take parameters of ANY type and return values that are again of ANY type but where we will define these types when we invoke these methods. Such methods do exist and they are called Generic Methods.

Generic methods are methods where the parameter types are not defined until the method is invoked. Parameter types are specified each time the method is invoked and the compiler can thus still check the code is valid.

In other words a generic method uses a parameterized type – a data type that is determined by a parameter.

In the code below a generic method Highest() has been defined as a method that takes two objects as parameters of unspecified type :-

```
static public T Highest<T>(T o1, T o2)
{
     if (o1.ToString().CompareTo(o2.ToString()) > 0)
          return o1;
     else
          return o2;
}
```

We can use this method and each time we invoke it we define the type of object being compared. If two students are compared the compiler will know that the object being returned is also type student and will therefore know it is legal to invoke student specific methods on this object.

A list of generic data types contained between angle brackets that follow the method's name is provided. If multiple generic types are used by a method, their names are separated by commas.

In the example above, the identifer T can stand for any data type. So, when T is used within the brackets in the method's parameter list to describe data, it might be an int, double, 'Student' or any other data type. The only requirement is that the method must work no matter what type of object is passed to it. All objects have a ToString() method because every data type inherits the ToString() method form the 'Object' class, or contains its own overriding version.

Download free eBooks at bookboon.com

In the case above only one generic data type is specified. This is used to define the type of both parameters and the return value.

The generic data type identifier for a generic method can be any legal C# identifier, but by convention it is usually an upper case letter. "T" is used to stand for "type".

When a generic method is defined i.e. one that works with any data type the type is specified when the method is used. Thus in the code below, while Highest() is a generic method, the compiler can see that a 'Student' object is being passed as a parameter and thus the object being returned must also be of type 'Student'.

```
Student s1 = new Student("Student A");
Student s2 = new Student("Student B");
Student highestStudent = Highest(s1, s2));
```

Given the object being returned is of type 'Student' it must be OK to store this in a variable of type Student and it must be OK to invoke and Student methods on the object returned.

Generic methods can therefore that work with any type data but the compiler can still check for type errors (as the type is specified each time the method is invoked).

Generic classes have been created, based on the same mechanisms as generic methods and these are particularly useful because there were used by the creators of the .NET libraries to create generic collections.

While we won't be making much use of generic methods and generic classes ourselves in this book we will be making significant use of generic collections, but before using these we need a basic understanding of collections themselves.

7.2 An Introduction to Collections

Most software systems need to store various groups of entities rather than just individual items. Arrays provide one means of doing this, but in C# collections are much more varied and flexible forms of grouping.

In C# collections are classes which serve to hold together groups of other objects. The .NET libraries provides several important classes defined in the System.Collections namespace that provide several different type of collection and associated functionality. This work has been developed and significantly improved upon by the creation of generic collections as defined in the System.Collections.Generic namespace. As generic collections are far more useful than non-generic collections the use of non-generic collections is normally discouraged and this book will focus entirely on generic collections.

Download free eBooks at bookboon.com

7.3 Different Types of Collections

There are three basic type of collection (List, HashSet and Dictionary) defined in the collections namespace.

Activity 1

Go online to msdn.microsoft.com. This is the website for the Microsoft Developer Network and the contains details of the Application Programmer Interface (API) for the .NET framework (it also contains much more).

Follow the links for
    Library (tab near top of page)
    .NET Development (on left pane)
    .NET Framework 4
    .NET Framework Class Library

At this point you will see all of the namespaces (packages) in the .NET framework on the left with a description of each on the right)

Follow the link for
    System.Collections and then
    System.Collections.Generic

Look at the class documentation in the main pane and identify any other collections that you may find useful.

Feedback 1

One of the classes you may have identified is SortedDictionary … this is just like a dictionary that we will cover but one where the elements are automatically sorted for us. We will use a sorted dictionary in the large case study (Chapter 11).

Other useful collections include LinkedList, Queue, and Stack. Discussions of these are beyond the scope of this text but if you have an understanding of Queues, or Linked Lists then it is worth knowing that these have been implemented for you as part of the .NET framework.

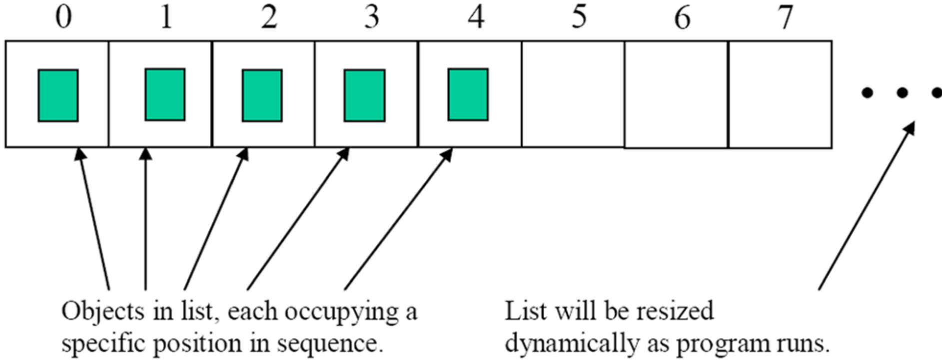
We will look at the List, HashSet and Dictionary classes, discuss what they do and show some of the more common useful methods that exist.

7.4 Lists

Lists are the most commonly used type of collection – where you might have used an array, a List will often provide a more convenient method of handling the data.

Lists are very general-purpose data structures, with each item occupying a specific position. They are in many ways like arrays, but are more flexible as they are automatically resized as data is added. They are also much easier to manipulate than arrays as many useful methods have been created that do the bulk of the work for you. Lists store items in a particular sequence (though not necessarily sorted into any meaningful order) and duplicate items are permitted.

Download free eBooks at bookboon.com



7.5 HashSets

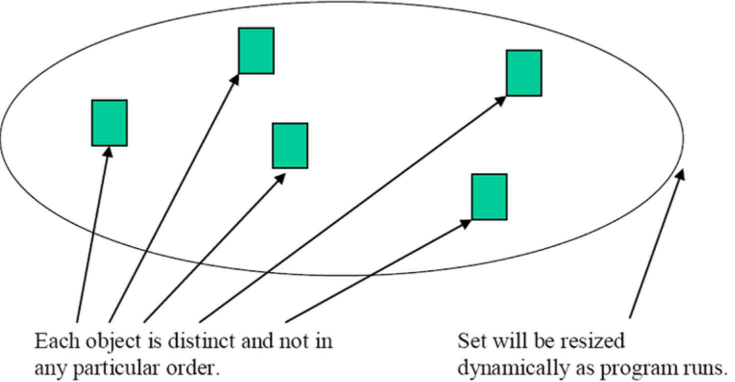
A HashSet is one implementation of a set. A set is like a 'bag' of objects rather than a list. They are based on the mathematical idea of a 'set' – a grouping of 'members' that can contain zero, one or many distinct items.

Unlike a List, duplicate items are not permitted in a set and a Set does not arrange items in order.

Download free eBooks at bookboon.com

Click on the ad to read more

Like lists, sets are resized automatically as items are added



Many of the operations available for a List are also available for a Set, but

- we CANNOT add an object at a specific position since the elements are not in any order
- we CANNOT 'replace' an item for the same reason (though we can add one and delete another)
- retrieving all the items is possible but the sequence is indeterminate
- it is meaningless to find what position an element is in.

7.6 Dictionaries

Dictionaries are rather different from Lists and Sets because instead of storing individual objects they store pairs of objects. The pair is made up of a 'key' and a 'value'. The key is something which identifies the pair. The value is a piece of information or an object associated with the key.

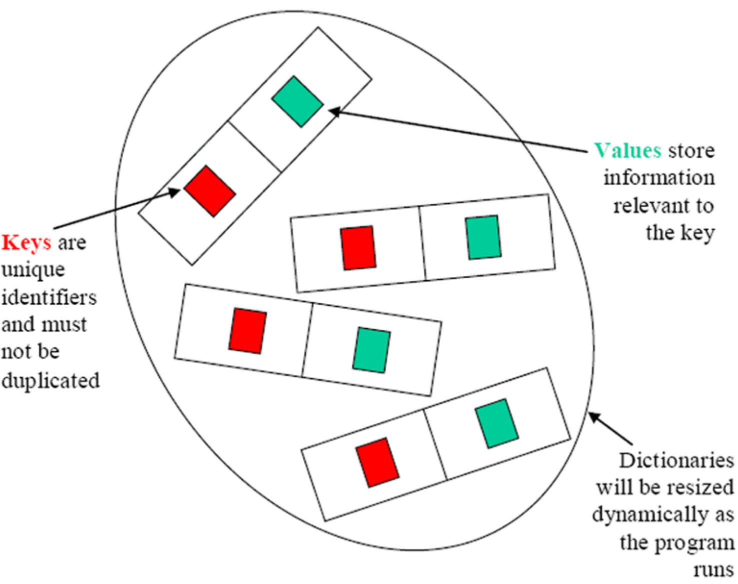
In language dictionaries that we are all familiar with the 'key' is the word you look up and the 'value' the meaning of that word. Of course in dictionaries that we are familiar with the entries are sorted into work (or key) order. If that was not the case it would be very difficult to find a word in a dictionary.

Another example would be an address book: in an address book the keys would be people's names and the values their address, phone, email etc. There is only one value for each key, but since values are objects they can contain several pieces of data.

Duplicate keys are not permitted, though duplicate values are. So in the previous example if you looked up two people in the address book you may find them living at the same address – but one person would not have two homes.

Like a set, a Dictionary does not arrange items in order (but a SortedDictionary does, in order of keys) and like lists and sets, dictionaries are resized automatically as items are added or deleted.

Download free eBooks at bookboon.com



Activity 2

For each of the following problems write down which would be most appropriate :- a list, a set or a dictionary.

1) We want to record the members of a club.
2) We want to keep a record of the money we spend (and what it was spent on).
3) We want to record bank account details – each identified by a bank account number

Feedback 2

1) For this we would use a Set. Members can be added and removed as required and the members are in no particular order.
2) For this we would use a List – this would record the items bought in the order in which they were purchased. Importantly as lists allows duplicate items we could buy two identical toys (perhaps for birthday presents for two different children),
3) We would not have two identical Bank accounts so a Set seems appropriate however as each is identified by a unique account number a Dictionary would be the most appropriate choice.

Download free eBooks at bookboon.com

7.7 A Simple List Example

In this section of the book we will demonstrate a simple use of the List collection class.

> Activity 3
>
> Go online to msdn.microsoft.com. Find API for the List class defined in the System.Collections.Generic namespace.
>
> List three of the methods you think would be most useful.

> Feedback 3
>
> Some of the clearly useful methods include…
>
> Add() – which adds an object onto the end of the list,
> Clear() – which removes all the elements from the list,
> Contains() – which returns true if this list contains the specified object,
> Insert() – which inserts an element at the specified position,
> Remove() - which removes the first occurrence of an object from a list and
> Sort() – which sorts the element of the list.
>
> Note some methods are overloaded such as Sort() which can either sort the entire list or sort just part of the list specified by a range.

Click on the ad to read more

The code below shows an example of the creation of a list. In this case a list of names i.e. Strings.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ListOfNames
{
     class Program
     {

          // create names which will store a list of names
          private static List<String> names;

          static void Main(string[] args)
          {
               names = new List<String>(); // Create empty list

               string name;
               for (int i = 0; i < 3; i++) //Enter three names
               {
                    name = Console.ReadLine();
                    names.Add(name);          // Add each name onto list
               }

               foreach (String n in names) // Itterate through list
                                             displaying each element.
               {
                    Console.WriteLine(n);
               }

               Console.WriteLine("Please enter name to search for");
               String searchname = Console.ReadLine();
               if (names.Contains(searchname)) // Demonstrate Contains
                                                    method works.
               {
                    Console.WriteLine("Name found");
               }
               else
               {
                    Console.WriteLine("Name not found");
               }
               Console.ReadLine();
          }
     }
}
```

The code above is fairly self explanatory however perhaps a few parts are worthy of explanation in particular the creation of the list.

Firstly as we are using System.Collections.Generic we are using a generic List class i.e. this can be a list of any object but the type must be specified as the list is created.

Download free eBooks at bookboon.com

Thus the code segment below creates 'names' a variable that will hold a List of Strings. Note the type is specified within the angled brackets < > after the word 'List'.

```
    private static List<String> names;
```

This was created as a Static variable simply for the reason that we are using this directly from within 'Main' and we won't actually be creating an object of this class.

When we invoke the constructor for the List class we must again specify the type of list being created i.e. a list of strings.

```
        names = new List<String>(); // Create empty list
```

In the remainder of the code we a) iteratively add three names to the list, b) display the list and c) use the Contains() method to search for a specific name.

Each element of the list can be accessed in much the same way as elements of an array are accessed (e.g. Console.WriteLine(names[0]) and C# provides an improved 'for loop' construct that will iterate through the elements of the list giving us access to each element as it does so.

Thus as the code below iterates through the list 'n' is set to each element in turn.

```
foreach (String n in names)
{
    Console.WriteLine(n);
}
```

Note elements in a list are stored in order hence when the list is displayed the order remains the same.

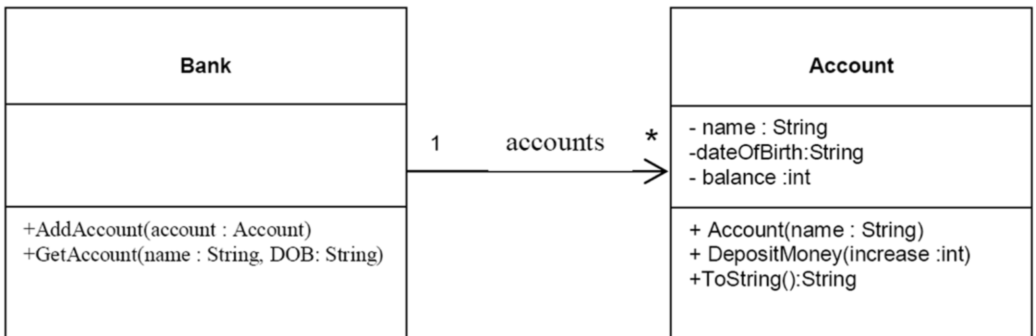
While this program stores and manipulates a list of Strings the program could just as easily store a list of Publication – or any object constructed from a class we create.

7.8 A More Realistic Example Using Lists

A more realistic example of lists would come where we are storing more complex objects of our own devising (not just Strings).

Download free eBooks at bookboon.com

Consider the UML class diagram below.



The diagram above represents the basics of a banking system (admittedly a very simplistic banking system). In this system a class is used to simulate a Bank and maintain a collection of Accounts… for the moment we will use a List to manage this collection.

Each element of this list will represent a single account object. One Bank object will maintain a list all accounts.

Each time an account is created the initial balance will be zero but the owner will be able to deposit money when required. So we can focus on the essential purpose of this exercise, to demonstrate the use of a list, we won't bother adding other obviously required functionality (to withdraw money etc).

Download free eBooks at bookboon.com

Click on the ad to read more

When each individual account is created we will add it to the list of accounts and we will also allow an individual account to be retrieved so that the account holder can deposit money into their account.

We will demonstrate this with a Main method that runs through a fixed routine a) adding a few accounts, b) displaying these accounts and c) retrieving an account, adding money to this account and displaying the list of accounts again.

We will need an accessor method in Bank so that the Main method can access the list of accounts to display them. In C# we will implement this using a public property.

The code for the Account class is shown below and should need no explanation.

```
class Account
{
     private String name;
     public String Name
     {
          get { return name; }
     }

     private String dateOfBirth;
     public String DateOfBirth
     {
          get { return dateOfBirth; }
     }

     private int balance;

     public Account(String name, String dob)
     {
          this.name = name;
          this.dateOfBirth = dob;
          balance = 0;
     }

     public void DepositMoney(int increase)
     {
          balance = balance + increase;
     }

     public override String ToString()
     {
          return "Name: " + name + "\nBalance : " + balance;
     }
}
```

Download free eBooks at bookboon.com

The code for the Bank class is given below and this demonstrates how easy it is to use the collection class 'List'.

```
class Bank
{
     private List<Account> accounts;
     public List<Account> Accounts
     {
           get { return accounts;}
     }

     public Bank()
     {
          accounts = new List<Account>();
     }

     public void AddAccount(Account account)
     {
          accounts.Add(account);
     }

     public Account GetAccount(String name,String dob)
     {
          Account FoundAccount = null;

          foreach (Account a in accounts)
          {
               if ((a.Name == name) && (a.DateOfBirth==dob))
               {
                    FoundAccount = a;
               }
          }
          return FoundAccount;
     }

}
```

Firstly the constructor creates a list of accounts. As with any list this automatically resizes itself, so unlike an array we do not need to worry about running out of space.

Also when elements are removed the remaining elements are moved so that the blank spaces are deleted. While this is not hard to do with arrays it is a time consuming process that is handled automatically for us when we use Lists.

Adding an account becomes a trivial task of invoking the Add() method.

Finally retrieving an individual account is fairly simple to do by iterating through the list until the account we are looking for is found.

Note this list manages a collection of complex objects and when any object is retrieved from the list the methods associated with that object can be invoked on that object.

This can be demonstrated via the following 'Main' method.

Download free eBooks at bookboon.com

```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;

     a = new Account("Alice","06/12/1963");
     b.AddAccount(a);
     a = new Account("Bert", "14/08/1990");
     b.AddAccount(a);
     a = new Account("Claire","1/1/2000");
     b.AddAccount(a);

     // Display the accounts
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Account a1 = b.GetAccount("Bert", "14/08/1990");
     a1.DepositMoney(100);

     // Display the accounts again
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Console.ReadLine();

}
```

LIGS University
based in Hawaii, USA
is currently enrolling in the Interactive Online BBA, MBA, MSc, DBA and PhD programs:

- enroll by October 31st, 2014 and
- save up to 11% on the tuition!
- pay in 10 installments / 2 years
- Interactive Online education
- visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education.
More info here.

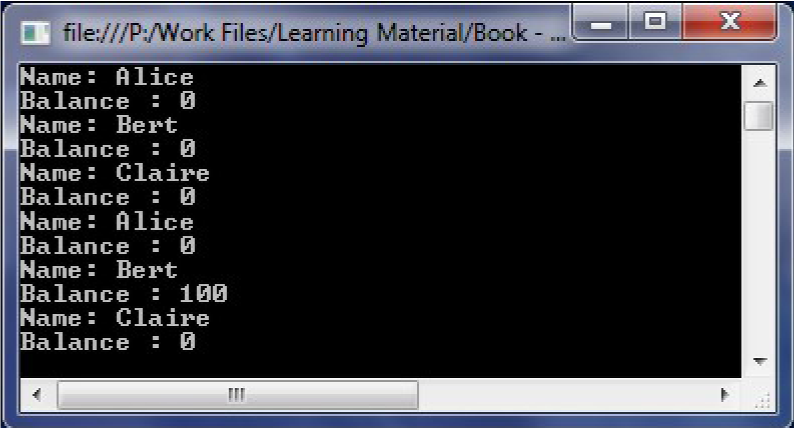
Download free eBooks at bookboon.com

Click on the ad to read more

The code above performs the following actions :-

- Firstly it invokes the constructor for the Bank class which in turn creates an empty list of Accounts.
- It then creates three accounts and adds these to the list in the Bank object.
- The list of accounts is then displayed
- An individual bank account is then retrieved from the list and money is deposited into this account.
- Finally the list of accounts is displayed again.

The output from running this program is shown below….



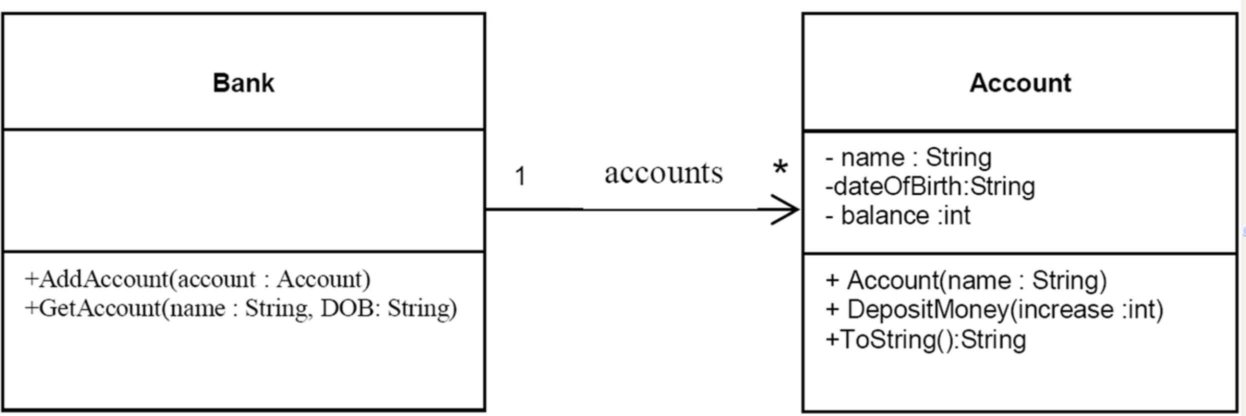
7.9 An Example Using Sets

In the example above we used a list to store a collection of bank accounts. This is not the most sensible choice as allows duplicated to be created and the bank would get very confused if two identical accounts were created (when money was deposited into an account which account should it be added to?).

Furthermore the list stores objects in a particular order – in this case the order the account were created. But this order is irrelevant as it will not help us find a particular account belonging to an individual.

The essential problem remains the same – to store a collection of bank accounts.

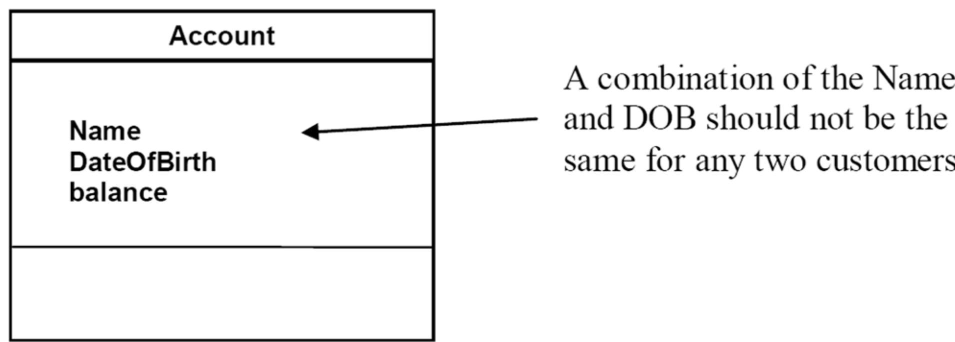
Download free eBooks at bookboon.com



However a much better choice for this collection would be to store the collection of accounts as a set, because sets do not allow duplicates to be created.

Consider a set of bank accounts :-

Now consider two bank accounts one for Mr Smith and one for Mrs Jones. To be certain that no two accounts are duplicates we would give each a unique account number – and we will do so later. For now we will assume that no two customers born on the same day will have the same name.



It should not be possible to create a second account where the account holder has the same name and DOB as a previous account holder. However unless told otherwise C# will treat the two objects below as different objects as both objects have different names (Account1 and Account2) and thus while sets do not allow duplicates objects both of these accounts could be created and added to a set because the computer does not recognize them as duplicates..

Download free eBooks at bookboon.com

Account1	Account2
Mr Smith 1/1/1990 £1,200	Mr Smith 1/1/1990 £0

To overcome this problem we need to override the Equals() method defined in the Object class to say these objects are the same if the Name and DOB are the same.

We can do this as below….

```
public override bool Equals(object obj)
{
     Account a = (Account)obj;
     return ( (name==a.Name) && (dateOfBirth==a.DateOfBirth));
}
```

Download free eBooks at bookboon.com
Click on the ad to read more

This overrides Object.Equals() for the Account class

Even though it will always be an Account object passed as a parameter, we have to make the parameter an Object type (and then cast it to Account) in order to override the Equals() method inherited from Object which has the signature

public bool Equals (Object obj)

(You can check this in the by looking for the 'Object' class in the 'System' namespace of the .NET API)

If we gave this method a signature with an Account type parameter it would not override Object.Equals(Object obj). It would in fact overload the method by providing an alternative method. As the system is expecting to use a method with a signature of public bool Equals (Object obj) it would not use a method with a signature of public bool Equals (Account obj).

Therefore we need to override public bool Equals (Object obj) and cast the parameter to an Account before extracting the name and DOB of the account holder to compare them with those of the current object.

One additional complication concerns how objects are stored in sets. To do this C# uses a hash code. Two accounts with the same name and DOB should generate the same hash code however currently this will not be the case as the hash code is generated using the object name (e.g. Account1). Thus two accounts, Account 1 and Account 2 could still be stored in the set even if the name and DOB is the same.

To overcome this problem we need to override the GetHashCode() method so that a hash code is generated using the Name and DOB rather than the object name (just as we needed to override the Equals() method).

We can ensure that the hash code generated is based on the name and DOB by overriding the GetHashCode() method as shown below….

```
public override int GetHashCode()
{
     return (name + dateOfBirth).GetHashCode();
}
```

The simplest way to redefine GetHashCode() for an object is to join together the instance values which are to define equality as a single string and then take the hashcode of that string. In this case equality is defined by the name of the account holder and DOB which taken together we assume will be unique.

It looks a little strange, but we can use the GetHashCode() method on this String even though we are overriding the GetHashCode() method for objects of type Account.

GetHashCode() is guaranteed to produce the same hash code for two Strings that are the **same**. Occasionally the same hash code may be produced for **different** key values, but that is not a problem.

Download free eBooks at bookboon.com

By overriding Equals() and GetHashCode() methods C# will prevent objects with duplicate data (in this case with duplicate name and dates of birth) from being added to the set.

The full code for the class Account is given below.

```
class Account
{
     private String name;
     public String Name
     {
          get { return name; }
     }

     private String dateOfBirth;
     public String DateOfBirth
     {
          get { return dateOfBirth; }
     }

     private int balance;

     public Account(String name, String dob)
     {
          this.name = name;
          this.dateOfBirth = dob;
          balance = 0;
     }

     public void DepositMoney(int increase)
     {
          balance = balance + increase;
     }

     public override String ToString()
     {
          return "Name: " + name + "\nBalance : " + balance;
     }

     public override bool Equals(object obj)
     {
          Account a = (Account)obj;
          return ( (name==a.Name) && (dateOfBirth==a.DateOfBirth));
     }

     public override int GetHashCode()
     {
          return (name + dateOfBirth).GetHashCode();
     }
}
```

The code above is identical to the previous version of the Account class (when we used lists) except for the addition the overridden methods Equals() and GetHashCode(). When storing any object in a set we must override both of these methods to prevent duplicates being stored.

Download free eBooks at bookboon.com

The code for the Bank class is identical apart from the firs few lines which creates a typed HashSet instead of a typed List.

```
class Bank
{
     private HashSet<Account> accounts;
     public HashSet<Account> Accounts
     {
           get { return accounts;}
     }

     public Bank()
     {
          accounts = new HashSet<Account>();
     }

     public void AddAccount(Account account)
     {
          accounts.Add(account);
     }

     public Account GetAccount(String name,String dob)
     {
          Account FoundAccount = null;

          foreach (Account a in accounts)
          {
               if ((a.Name == name) && (a.DateOfBirth==dob))
               {
                     FoundAccount = a;
               }
          }
          return FoundAccount;
     }

}
```

Download free eBooks at bookboon.com

Finally the Main method used to demonstrate this is shown below…

```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;

     a = new Account("Bert", "06/12/1963");
     b.AddAccount(a);

     // try to create and add a duplicate account-this shouldn't work
     a = new Account("Bert", "06/12/1963");
     b.AddAccount(a);

     a = new Account("Alice", "14/08/1990");
     b.AddAccount(a);
     a = new Account("Claire", "1/1/2000");
     b.AddAccount(a);



     // Display the accounts
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Account a1 = b.GetAccount("Bert", "06/12/1963");
     a1.DepositMoney(100);

     // Display the accounts again
     foreach (Account acc in b.Accounts)
     {
          Console.WriteLine(acc.ToString());
     }

     Console.ReadLine();

}
```

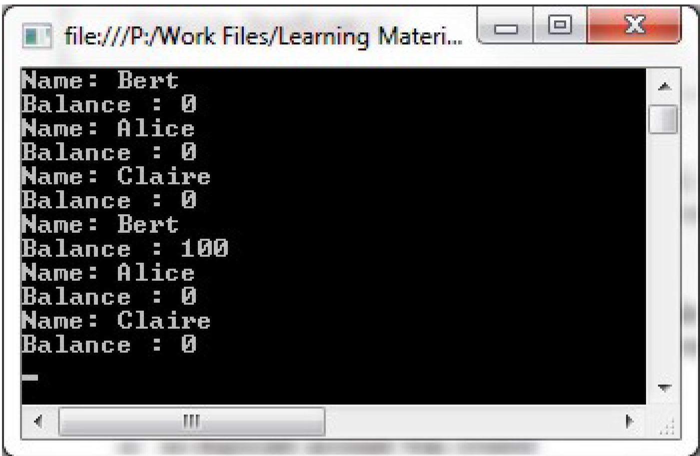
This Main method is very similar to the Main method used to test lists. Several accounts are created, displayed, an account is retrieved and after a deposit into this account has been made the list is displayed again.

Two changes should be noted. First an attempt is made to create a duplicate account – this should not work. The list of accounts has been created in a non alphabetical order.

By looking at the program output, shown below, we can see that :-

a) no duplicate account was created, so overriding Equals() and GetHashCode() worked.
b) the accounts are listed in the order created but not in an meaningful order. Entries in sets are not stored in any order and therefore we cannot be sure which order they will be retrieved in.

Download free eBooks at bookboon.com



Activity 4

Earlier we found that some of the useful methods for Lists included…

Add() – which adds an object onto the end of the list,
Clear() – which removes all the elements from the list,
Contains() – which returns true if this list contains the specified object,
Insert() – which inserts an element at the specified position,
Remove() - which removes the first occurrence of an object from a list and
Sort() – which sorts the element of the list.

Go online to msdn.microsoft.com. Find API for the HashSet class defined in the System.Collections.Generic namespace and find out which of the methods in the List class have equivalent methods in the HashSet class.

Feedback 4

You should find Methods Add(), Clear() Contains() and Remove() methods also exist for a HashSet.

Insert() and Sort() do not exist. As sets are unsorted objects you cannot sort them and as they have no specific order it make no sense to try to insert an object at a specified position.

There is much commonality between the different types of collection. Having learnt how to use one it is relatively easy to learn another.

Download free eBooks at bookboon.com

Activity 5

The code below will find a String in a set of Strings (called StringSet). Amend this so this will work find a Publication in a set of Publications (called PublicationSet).

```
public void FindString(String s)
{
     boolean found;
     found = StringSet.Contains(s);
     if (found)
     {
          Console.WriteLine("Element " + s + " found in set");
     }
     else
     {
          Console.WriteLine("Element " + s + " NOT found in set");
     }
}
```

Maastricht University

Join the best at the Maastricht University School of Business and Economics!

Top master's programmes
- 33rd place Financial Times worldwide ranking: MSc International Business
- 1st place: MSc International Business
- 1st place: MSc Financial Economics
- 2nd place: MSc Management of Learning
- 2nd place: MSc Economics
- 2nd place: MSc Econometrics and Operations Research
- 2nd place: MSc Global Supply Chain Management and Change

Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012

Maastricht University is the best specialist university in the Netherlands (Elsevier)

Visit us and find out why we are the best!
Master's Open Day: 22 February 2014

www.mastersopenday.nl

Feedback 5

```
public void FindPublication(Publication p)
{
     boolean found;
     found = PublicationSet.Contains(p);
     if (found)
     {
          Console.WriteLine("Element " + p + " found in set");
     }
     else
     {
          Console.WriteLine("Element " + p + " NOT found in set");
     }
}
```

Activity 6

Consider the set of Publications that would need to be created for the code in Activity 5 to work and answer the following questions.

1) Could such a set be used to store a collection of books?

2) Could it store a combination of books and magazines?

3) If a book was found in the set what would the following line of code do?

Console.WriteLine("Element " + p + " found in set");

Feedback 6

1) Yes. Books are a subtype of publication and could be stored in a set of publications.

2) Yes. For the same reasons we could store a combination of books and magazines objects (or any other type of publication).

3) 'p' would invoke the ToString() method on the publication. The CLR engine would determine at run time that this was in fact a book and assuming the ToString() method had been overridden for book, to return the title and author, this would make up part of the message displayed. Thus polymorphically the message would change depending upon which type of publication was found in the collection.

7.10 An Example Using Dictionaries

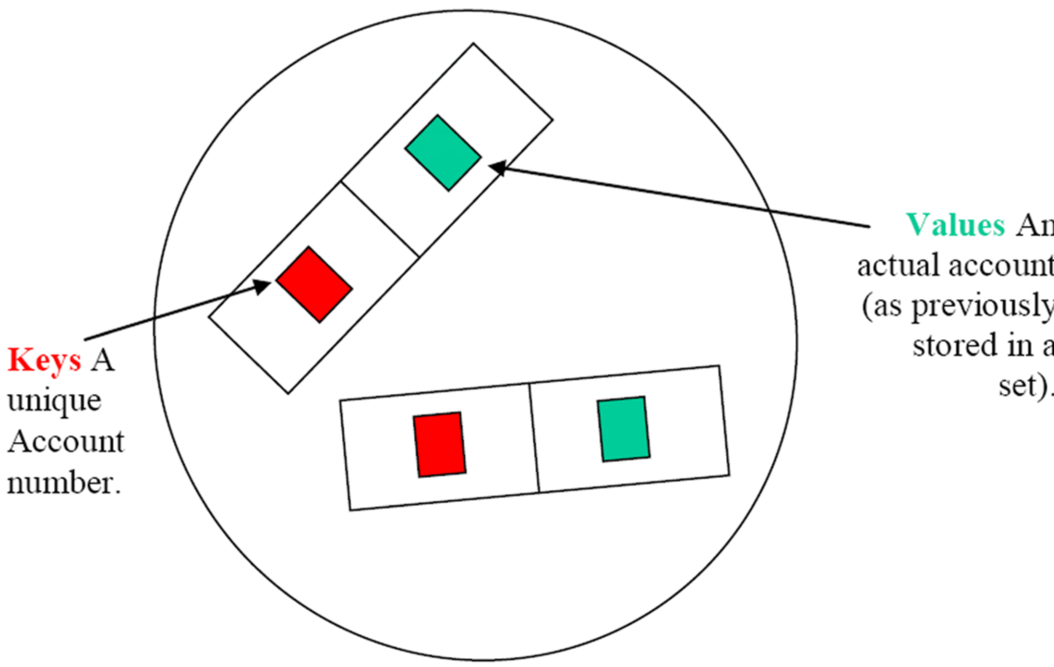
In the previous example we assumed no bank accounts would be created for two customers with the same name and DOB. However it is possible that two customers could have the same name and DOB.

Download free eBooks at bookboon.com

To make accounts truly unique we could amend the definition of Account to contain an account number that we will ensure is unique and we could continue to store these using a set. However as we will often retrieve account using an account number a dictionary would be a more efficient collection for this application.

A dictionary is like a set but where each item stored is made up of a key/value pair.

In this case the key would be a unique account number and the value would be the associated bank account object.



In the previous example we stored a collection of Accounts in a set, we do not need to make any changes to the Account class if we wish to store these in a dictionary.

We could remove the overridden methods for Equals() and GetHashCode() from the Account class as a dictionary can contain duplicate values. Alternatively we could leave these methods in should we wish to.

In a dictionary it is the keys that are unique.

Download free eBooks at bookboon.com

There are some significant changes to the Bank class that revolve around the fact that it is a Dictionary not a set that is being created. The code for the Bank class is shown below :-

```
class Bank
{
     private Dictionary<int,Account> accounts;
     public Dictionary<int,Account> Accounts
     {
           get { return accounts;}
     }

     public Bank()
     {
          accounts = new Dictionary<int, Account>();
     }

     public void AddAccount(int number,Account account)
     {
          try
          {
               accounts.Add(number, account);
          }
          catch (Exception)
          {
          }
     }

     public Account GetAccount(int number)
     {
          Account a;
          accounts.TryGetValue(number, out a);
          return a;
     }
}
```

In the code above when creating the dictionary you can see that two data types are specified :-. Firstly the type for the dictionary key is specified – in our case we will use a simple 'int' for an account number. Secondly the data type for the value is specified – in our case the value is a complete account object. Hence the segment of code below :-

```
     private Dictionary<int,Account> accounts;
```

The dictionary class defines two very useful methods for us, Add() and TryGetValue().

The Add() methods requires two parameters, the 'Key' and the 'Value'. Hence we must provide both the account number and the account object to be added to the dictionary :-

```
          accounts.Add(number, account);
```

Download free eBooks at bookboon.com

Note this method will generate an exception if the 'key' is not unique and this error should be trapped and dealt with ….though in this simple case we have decided to take no action should this occur.

The TryGetValue() method requires an input argument, the key it is searching for and sets the value of the second argument to the value associated with that key. In our example it sets the second parameter to the account associated with the account number. Note this will be null if the account number does not exist. See code below :-

```
        accounts.TryGetValue(number, out a);
```

Download free eBooks at bookboon.com

Click on the ad to read more

The Main method used to test this dictionary is given below :-

```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;
     int an;     // account number;

     a = new Account("Bert", "06/12/1963");
     b.AddAccount(1,a);

     // try to create and add a duplicate account-this shouldn't work
     a = new Account("Bert", "06/12/1963");
     b.AddAccount(1, a);

     a = new Account("Alice", "14/08/1990");
     b.AddAccount(2, a);
     a = new Account("Claire", "1/1/2000");
     b.AddAccount(3, a);

     // Display the accounts
     foreach (KeyValuePair<int, Account> kvp in b.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;

          Console.WriteLine("Account number: " + an +
                         "\nAccount details: " + a.ToString());
     }

     a = b.GetAccount(1);
     a.DepositMoney(100);

     foreach (KeyValuePair<int, Account> kvp in b.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;
          Console.WriteLine("Account number: " + an +
                         "\nAccount details: " + a.ToString());
     }

     Console.ReadLine();
}
```

Functionally the 'Main' method above is virtually identical to the Main method used to test the Set example.

The code above performs the following operations :-

- Several accounts are created, including one attempt to create a duplicate,
- The collection of accounts is displayed, in this case including an account number,
- an account is retrieved and amended and then
- the collection is displayed again.

To keep this example short we have not protected against the possibility that the account being retrieved cannot be found.

Download free eBooks at bookboon.com

The following lines of code which display the contents of the collection are perhaps worthy of a fuller explanation :-

```
foreach (KeyValuePair<int, Account> kvp in b.Accounts)
{
    an = kvp.Key;
    a = kvp.Value;
        Console.WriteLine("Account number: " + an +"\nAccount details: " + a.ToString());
}
```

Each object in the collection is made up of a key/value pair. Thus when we iterate around the collection each iteration returns a key/value pair.

Above we define kvp a variable made up of a key/value pair where the key is an 'int' i.e. an account number and the value is an Account object We split this pair into its component parts and store these in the respective variables ('an' and 'a'). We then display the account number and invoke the ToString() method on the account, using the result to display details of the account.

The output from this program is shown below:-

```
file:///P:/Work Files/Learning Material/Book - ...
Account number: 1
Account details: Name: Bert
Balance : 0
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0
Account number: 1
Account details: Name: Bert
Balance : 100
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0
```

As we would expect the system did not allow us to create accounts with duplicate account numbers.

Download free eBooks at bookboon.com

If you look at the Dictionary class in the System.Sollections.Generic namespace of the .NET framework you will see a host of other useful methods that have been created to manage Dictionaries. These include Clear(), ContainsKey(), ContainsValue() and Remove().

7.11 Serializing and De-serializing Collections

Collections, Lists, Sets and Dictionaries among others, are very powerful flexible mechanisms for storing collections of objects.

They automatically resize themselves and contain methods that save significant programming effort when adding members, retrieving members, removing members, searching for members, sorting collections etc.

They become even more powerful when combined with the serialization / de-serialization facilities provided by the .NET framework.

Using traditional file handling routines textual data can be stored and retrieved from files.

Serialization allows whole objects to be stored with one simple command (once an appropriate file has been opened).

However C# objects frequently contain references to other objects, they create what is known formally as a 'graph' – a network of connections. The serialization mechanism follows these references and also serializes the objects referenced… and objects those objects reference… etc.

Empowering People.
Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

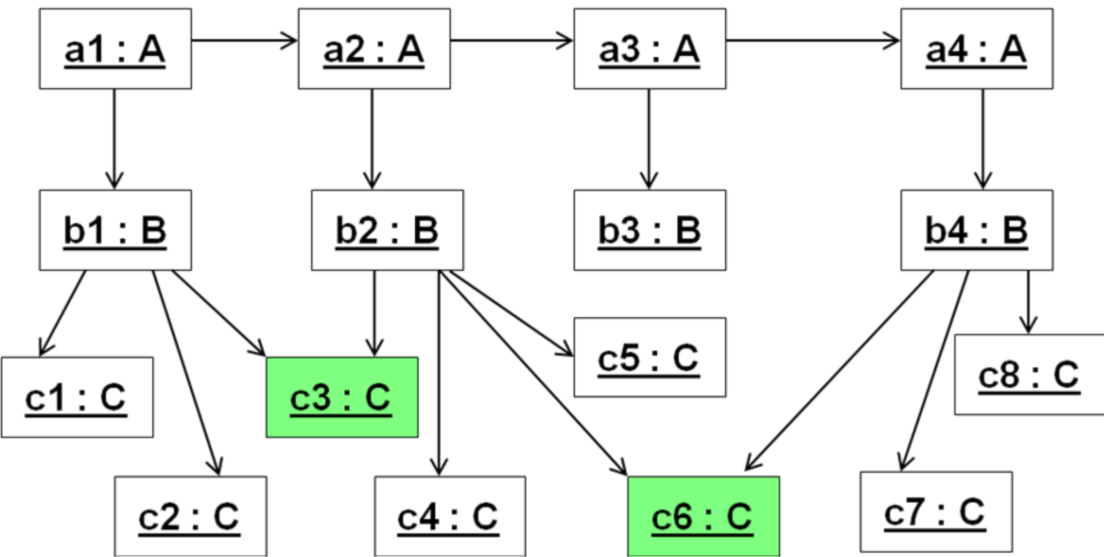
BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

Download free eBooks at bookboon.com

An object graph is shown below :-



In this graph we can see that object a1 refers to objects a2 and object b1. These in turn refer to other objects. Thus if we were to fully save (or serialise) a1 we would need to store all the details of this object including details of all the other objects this refers to… and all the objects these refer to and so on.

Serialisation will follow these links automatically.

Notice that c3 and c6 are referred to by two objects. However, the serialization mechanism is smart enough to realise that the same object is occurring when it encounters it for the second time and instead of writing out a duplicate it simply writes out a reference to the original object. By the same means a 'cycle' in the graph – i.e. a circle or references returning to the same place, will not cause the serialization mechanism to go into an infinite loop!

This mechanism assumes that classes A, B and C are all serializable. Some classes are not. For instance those that rely on reading from a file cannot be serialised as the file may not exist when we attempt to deserialize an object of that class. In practice most of the classes we create will be serializable.

To demonstrate the power of the serialization mechanism we will amend the bank account system developed in section 7.10 where a dictionary was used to store a collection of accounts.

With one instruction we will store an object of type Bank. In doing so the system will also store the collection Accounts and in doing so it will store ALL Account objects.

When we retrieve the Bank object all Account objects will also be retrieved and the collection reconstructed.

Download free eBooks at bookboon.com

In order to do this no changes need to be made to the Account class or the Bank class though we must tell the compiler these classes can be serialiazed.

To do this we place the keyword serializable in front of each class as shown below….

```
[Serializable]
class Account
{
    // code from body of class omitted
}
[Serializable]
class Bank
{
    // code from body of class omitted
}
```

In the 'Main' method that we will use to invoke the serialization \ de-serialization process, and to test the results, we need to add additional using statements as we are using additional parts of the .NET framework (see below).

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
```

A complete 'Main' method that will demonstrate this working is provided below. Much of this is either self explanatory or contains code we are already familiar with. The new parts that relate to the serialiazation / de-serialization will be explained afterwards.

Download free eBooks at bookboon.com

```
static void Main(string[] args)
{
     Bank b = new Bank();
     Account a;
     int an;     // account number;

     a = new Account("Bert", "06/12/1963");
     b.AddAccount(1, a);
     a = new Account("Alice", "14/08/1990");
     b.AddAccount(2, a);
     a = new Account("Claire", "1/1/2000");
     b.AddAccount(3, a);

     Console.WriteLine("Stored data");
     foreach (KeyValuePair<int, Account> kvp in b.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;

          Console.WriteLine("Account number: " + an +
                              "\nAccount details: " + a.ToString());
     }

     FileStream outFile = new FileStream("AccountData",
                                  FileMode.Create, FileAccess.Write);
     BinaryFormatter bFormatter = new BinaryFormatter();
     bFormatter.Serialize(outFile, b);
     outFile.Close();
     outFile.Dispose();

     Bank b2 = new Bank();
     FileStream inFile = new FileStream("AccountData", FileMode.Open,
                                                  FileAccess.Read);
     b2 = (Bank)bFormatter.Deserialize(inFile);
     inFile.Close();
     inFile.Dispose();

     Console.WriteLine("\nRetrieved data");
     foreach (KeyValuePair<int, Account> kvp in b2.Accounts)
     {
          an = kvp.Key;
          a = kvp.Value;

          Console.WriteLine("Account number: " + an +
                         "\nAccount details: " + a.ToString());
     }

     Console.ReadLine();

}
```

The code above performs the following steps :-

- Firstly several bank accounts are created, added to the collection and the collection is then displayed.
- Next an output file is created
- A binary formatter is created as this is used by the serialization process.

Download free eBooks at bookboon.com

- With one line the collection 'b' is serialised and in doing so all account objects inside this collection are serialized (see code below).

```
bFormatter.Serialize(outFile, b);
```

- In order to prove this has worked a new, empty Bank object is created 'b2'.
- The de-serialization process is invoked and the output cast into a Bank type. The results are then stored in the object 'b2' (see code below).

```
b2 = (Bank)bFormatter.Deserialize(inFile);
```

Finally the contents of 'b2' are displayed to show they are consistent with the data originally created.

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now

Go to www.helpmyassignment.co.uk for more info

The results of running this program are shown below.

```
file:///P:/Work Files/Learning Material/Book - C#/C...
Stored data
Account number: 1
Account details: Name: Bert
Balance : 0
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0

Retrieved data
Account number: 1
Account details: Name: Bert
Balance : 0
Account number: 2
Account details: Name: Alice
Balance : 0
Account number: 3
Account details: Name: Claire
Balance : 0
```

7.12 Summary

The .NET Collections classes provide ready-made methods for storing collections of objects. These almost completely make the use of arrays redundant!

There are 'untyped' collections and 'typed' collections. Typed collections, or generic collections, were developed based upon the idea of generic methods and generic classes. The use of generic collections are much more useful than untyped collections and thus we have not shown the use untyped collections here.

Collection classes include List, HashSet and Dictionary. Each of these define appropriate methods, many of which are common across all of the collection classes.

Special attention is required when defining objects to be stored in Sets (or as keys in Dictionaries) to define the meaning of 'duplicate'. For these we need to override the Equals() and GetHashCode() methods inherited from Object.

Serialization is a very powerful mechanism that allows the entire contents of a collection and all related objects to be stored with one simple command.

A further example of the use of collections will be provided in Chapter 11, a larger case study at the end of this book. This will demonstrate the use of lists, sets and dictionaries for a small but more realistic example application. The code for this will be available to download and inspect if required.

Download free eBooks at bookboon.com