Session 11. Files and Generics

This session is divided into two main areas: files and generics. Files represent a mechanism through which we can provide persistence of data across multiple executions of the same program, i.e., you can save data and load it the next time you run a program. Although database technologies are more commonly used to persist data across multiple executions, files are still a relevant and important skill, but we also use them as a vehicle to explore serialisation. Serialisation provides us with the ability to represent an object (or collection of objects) as a stream/block of data, for which we explore binary, XML, and SOAP representations. This stream/block of data can then be stored in a file (as we examine), stored in a database, or transmitted over a network connection (commonly used for web services and some cloud computing technologies for example). Finally, we examine generics, which provides an ability for us to define methods and classes without referencing a particular data type. Modern programming libraries often require the use of generics, such as we have done with the List<> class, and you will need this knowledge progressing into a career in software development.

Session Objectives

In this session, you will learn:

- How file input and output can be used to add persistence to the data in an application;
- The difference between text files and binary files;
- The difference between sequential access and random access and how they can/cannot be applied to text files and binary files;
- The concept of serialisation and how to implement binary, SOAP, and XML serialisation; and
- The concept of generics and how to implement generic functions and classes in C#.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*
  Although we do not directly address this ULO in this session, we continue developing software that exploit object-oriented concepts, further developing our understanding of these concepts.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  The application of both files and generics are skills commonly used in the development of modern applications, and we continue to develop skills relevant to developing object-oriented solutions in the C# programming language.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  We will not be directly addressing this ULO, however further developing our

understanding of the C# programming language contributes to the skills required for this outcome.
- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*
  We will not be directly addressing this ULO, however further developing our understanding of, and practice in, the C# programming language, contributes to improving understanding of object-oriented designs.

Required Reading

Additional reading is required from the textbook Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014, as follows:

Required Reading: Chapter 17
This chapter covers files in some detail, however is limited to coverage of text files accessed sequentially and serialisation concepts.

Required Reading: Chapter 20
This chapter covers both generic functions and generic classes.

Required Tasks

The following tasks are mandatory for this week:

- Task 11.1. Text File I/O
- Task 11.2. Serialisation
- Task 11.3. Generics

11.1. Introduction

In this final session of content we examine two important topics in programming: files and generics. Files provide a way to provide survivability to the data that a user generates, by allowing the data they have entered/generated in a program to be written to a secondary storage device such as a disk. In Section 11.2 we examine text vs binary files, sequential vs random access, and examine serialisation as an alternative means of storing and retrieving objects from secondary storage.

Generics have emerged in recent years as an important tool for programmers, allowing programmers to define functionality for undefined/unknown data types. This is particularly important when examining collections of data (such as the `List<>` generic class), and modern object-oriented programming libraries will often incorporate such generic classes to allow for rapid development of applications. Thus, it is necessary to develop this skill to be a successful developer.

11.2. File Input and Output

Until now, the applications that we have considered begin execution with no data from previous runs/executions of that application, and upon terminating any data stored in the application's memory is discarded. From your own knowledge of

computing, it should already be clear to you that it is necessary to keep data for longer periods than this, referred to as ***persistence*** of data – when the application is closed or power is lost the data remains. To achieve persistence it is necessary to employ secondary storage, which provides a non-volatile storage medium, as opposed to RAM which does not retain its contents.

The two primary means of storing data on secondary storage are through the use of databases (commonly used for web sites), or through the ability to save files that can be loaded later (commonly used by GUI applications such as word processors). The topic of database support is covered primarily by the unit SIT103 Database and Information Retrieval. The ability to work with files however is a fundamental skill of a programmer, thus we examine them here.

A file is a logical element of storage stored on a hard disk, USB drive, optical disk, etc., and can be read from and/or written to similar to how we have used the Console object to date. Key to understanding files (and why they are similar to the Console) is the concept of a ***stream***, a concept repeated regularly throughout input and output support in many programming languages/libraries. Streams represent data as a sequence of bytes for which there are no clearly defined boundaries. If you consider input from the Console, there is no clearly defined structure to the data input by the user, however by attempting to read a particular type, e.g., an integer, and validating the input to match that data type, we can successfully interpret the information entered by the user.

11.2.1. File Types and Access Modes

There are two possible types of files: ***text files*** and ***binary files***. All of the input and output that we have seen to date has been in the form of text read from and/or written to the console, which is equivalent to the content of a text file. Text files are human readable, consisting of letters, numbers, symbols, and a number of characters that control how the text is displayed (tabs, new lines, etc.). Note however that text files are neither particularly efficient for storing data nor for the computer to work with. This is because of the need to convert between numeric formats and text.

Consider storing a student number in the memory of a computer, e.g., “214123456”. When stored as text, each character is stored separately. Characters are usually represented using either one byte per character (such as ASCII encoding) or two bytes per character (such as Unicode encoding used in Microsoft.Net). As such, the above student number would require a total of nine bytes (one byte encoding) or 18 bytes (two byte encoding) depending on which encoding is used. However, the same student number could also be accurately stored in a 32-bit/four byte integer, a saving of four or 12 bytes per student number.

Converting between numeric and textual representations of numeric data is also quite an expensive operation, which we have achieved using composite formatting and `ToString()` methods (numeric to textual), and the `Convert` class and `Parse()`/`TryParse()` methods (textual to numeric). Binary files eliminate the need for these conversions by

allowing numeric data to be stored in its native format (computer readable format), however this is at the cost of the file being human readable[1].

Reading and writing data from/to files can be accomplished either using sequential access or random access. The two access modes are distinguished by the ability to seek with random access. In particular, for sequential access, it is only possible to read a file from is beginning to its end, and similar when writing to a file. However, most modern storage mediums however allow the location of reading/writing to be changed within the file, known as seeking. Importantly however, even if the medium supports random access, it is still common to work with a file sequentially, as is usually the case with text files.

11.2.2. Text Files and Sequential Access

Before attempting to work with files in C#, we must first import the relevant namespace with the following statement at the top of our code file/s:

```
using System.IO;
```

There are a several ways to open text files using Microsoft.Net. We consider two ways to open a file:

```
FileStream variable_name = new FileStream(path, mode, access);
```

and

```
FileStream variable_name = File.Open(path, mode, access);
```

You will notice in these two examples that the same type of object is returned (a `FileStream`), and the same three parameters are used in the creation. In fact, these two ways are considered equivalent, thus you can safely select whichever one you prefer. The three parameters are as follows. The first parameter, *path*, specifies the filename to be opened and/or created (usually either a variable or a verbatim string literal). The second parameter, *mode*, specifies how the file should be opened and/or prepared, as follows for:

- `FileMode.Append` – (only for writing) opens an existing file or creates a new file and seeks to the end of the file;
- `FileMode.Create` – truncates an existing file or creates a new file;
- `FileMode.CreateNew` – creates a new file, throwing a `System.IO.IOException` if the file already exists;
- `FileMode.Open` – opens an existing file, throwing a `System.IO.FileNotfoundException` if the file does not exist;
- `FileMode.OpenOrCreate` – opens an existing file or creates a new file if it does not exist;

[1] Actually most of the content of binary files is often readable, however a text editor is not adequate – many hexadecimal editors/hex editors display the numeric value of each byte in the file (in hexadecimal format), and also the textual representation of that byte (if relevant).

- `FileMode.Truncate` – opens an existing file, zeroing its length (truncating the file) in the process.

The final parameter, *access*, determines whether the file is open for reading or writing:

- `FileAccess.Read` – opens a file only for reading;
- `FileAccess.Write` – opens a file only for writing;
- `FileAccess.ReadWrite` – opens a file for both reading and writing (see Section 11.2.3 for an example).

The `FileStream` object on its own is not particularly useful for text files, so we use two other classes: `StreamReader` and `StreamWriter`, which are used for reading and writing text files, respectively. Objects of these classes are created after opening the file, as follows:

```
StreamReader variable_name = new StreamReader(filestream_variable);
```

and

```
StreamWriter variable_name = new StreamWriter(filestream_variable);
```

Working with these classes is very similar to working with the Console class, except that the reading and writing methods are split into the appropriate class, i.e., objects of `StreamReader` type support the `Read()` and `ReadLine()` methods, whereas objects of `StreamWriter` type support the `Write()` and `WriteLine()` methods.

The last step in working with files is to close the files by invoking the `Close()` method of the `StreamReader` or `StreamWriter` object. Closing files is important for two reasons: (1) any data stored waiting in a buffer is immediately written if necessary, and (2) any operating system resources are freed.

An example of how to create and write to a text file is shown in Figure 11.1. The example begins by creating a file. Everything the user types in at the console is then written to the file until the user types in the text 'end', upon which the file is closed. A related example for reading from a text file is shown in Figure 11.2, where a text file is opened and its contents are displayed on the console.

```
FileStream fs = File.Open(@"test.txt", FileMode.Create, FileAccess.Write);
StreamWriter sw = new StreamWriter(fs);

Console.Write("Text or 'END' to terminate: ");
string text = Console.ReadLine();
while (text.ToUpper() != "END")
{
    sw.WriteLine(text);
    Console.Write("Text or 'END' to terminate: ");
    text = Console.ReadLine();
}

sw.Close();
```

Figure 11.1: Example of Writing to a Text File

```
FileStream fs = File.Open(@"test.txt", FileMode.Open, FileAccess.Read);
```

```
StreamReader sr = new StreamReader(fs);

Console.WriteLine("File contents:");
while ((text = sr.ReadLine()) != null)
    Console.WriteLine(text);

sr.Close();
```

Figure 11.2: Example of Reading from a Text File

11.2.3. Binary Files and Random Access

As noted in Section 11.2.1, random access involves the use of seeking in the file. Seeking is the process of changing the location to read from/write to the file. For an analogy of seeking, consider a young child reading from a book. Children are often taught to use their finger to track their current location in the book, i.e., to effectively point to the word they are currently reading. The equivalent of seeking would be observed if the child were to lift their finger and move it to another page and continue reading from there. However the random seeking that may be seen in a child reading a book is not useful when trying to read or write data that can be comprehended by the computer, thus we must first consider the question seek to where?

The answer to this question begins by defining the concept of a ***record***. A record is the grouping/collection of data in a file that represents one logical element stored in that file. For example, a file can contain a number of book records, where each record contains the relevant data for an individual book. Thus, a record roughly corresponds to an object in an object-oriented program. To be successful with random access, it is necessary that a record be fixed size, i.e.,

If the records are a fixed length, then it is possible to determine the address of the nth record by the formula , where n is the number of the record (first record = 1, second record = 2, etc.), L is the length of an individual record, and the result (A) is the address of the record in the file. Thus, given a record size of 50 bytes, to read the fifth record we would first seek to address 200 in the file and then read a book record.

This answers the question of how to calculate the address for a record, but the question remains of how to know which record contains the data to be looked for. For this purpose, there are two alternatives: use an index or use hashing. An index is where a small part of the record (such as a key field) is stored separately along with the record number in the file. This index can then be either kept in memory or read much more efficiently from disk (much less data to read). Hashing involves applying an algorithm to calculate a numeric value from one or more other data elements, potentially allowing a record to be accessed with a single seek without the need for an index. Both indexes and hashing are beyond the scope of this unit.

The `Main()` method for an example program exploiting random access files is shown in Figure 11.3. In this example, 12 `Student` objects are created consisting of student

number, name, birth year, month, and day fields. These students are then written out to the file by invoking the static `ToStream()` method of the `Student` class that has been written for this purpose. Once all students have been written, the program enters a simple loop to randomly recall one of those records and display it. Upon pressing the ENTER key, a random number is generated to identify which record is to be retrieved. The `Seek()` method of the `FileStream` object is then invoked to move to that record, and the static `FromStream()` method of the `Student` class is used to read the record back in and then the `Student` is displayed.

```
static void Main(string[] args)
{
    FileStream testFile = new FileStream(@"testFile.dat", FileMode.Create,
FileAccess.ReadWrite);

    // Student names taken from Top 100 singles on Australian charts for 2000
    // Source: http://www.aria.com.au/pages/aria-charts-end-of-year-charts-top-100-
singles-2000.htm
    Student.ToStream(testFile, new Student(1001, "Anastacia",        1991,  1,  1));
    Student.ToStream(testFile, new Student(1002, "Wheatus",          1992,  2,  2));
    Student.ToStream(testFile, new Student(1003, "Bomfunk MCs",      1993,  3,  3));
    Student.ToStream(testFile, new Student(1004, "Madonna",          1994,  4,  4));
    Student.ToStream(testFile, new Student(1005, "Destiny's Child", 1995,  5,  5));
    Student.ToStream(testFile, new Student(1006, "Bardot",           1996,  6,  6));
    Student.ToStream(testFile, new Student(1007, "*N Sync",          1997,  7,  7));
    Student.ToStream(testFile, new Student(1008, "Madison Avenue",   1998,  8,  8));
    Student.ToStream(testFile, new Student(1009, "Spiller",          1999,  9,  9));
    Student.ToStream(testFile, new Student(1010, "Mary Mary",        2000, 10, 10));
    Student.ToStream(testFile, new Student(1011, "Mandy Moore",      2001, 11, 11));
    Student.ToStream(testFile, new Student(1012, "Chris Franklin",   2002, 12, 12));

    Random rnd = new Random();
    Student s;
    Console.Write("Press ENTER or type anything and ENTER to quit: ");
    string input = Console.ReadLine();
    while (input == "")
    {
        int which = rnd.Next(0, 12); // random number from 0-11
        testFile.Seek(which * Student.RecordLength, SeekOrigin.Begin);
        Student.FromStream(testFile, out s);
        Console.WriteLine("{0,2}: {1}", which, s);
        Console.Write("Press ENTER or type anything and ENTER to quit: ");
        input = Console.ReadLine();
    }

    testFile.Close();
}
```

Figure 11.3: Example Program using Random Access

The `ToStream()` method is shown in Figure 11.4, which introduces a number of new concepts. Note that it is not expected that you completely understand this example code. The key problem faced by working with random access files is the need for the record written to be of fixed length. All simple data types are fixed size, so these can be written straightforwardly. The problem comes from storing strings in the file, which for storing a name are of variable length. Thus, to store the variable length the string must first be converted to a fixed length field, where an array of characters is used.

```
public static void ToStream(Stream stream, Student student)
{
    BinaryWriter bw = new BinaryWriter(stream, Encoding.Unicode);

    bw.Write(student._ID);                  // Write out _ID

    // Write out the length of _Name
    int nameLength = student._Name.Length;
    if (nameLength > MAX_NAME_LENGTH)
        nameLength = MAX_NAME_LENGTH;
    bw.Write(nameLength);

    // Prepare to write out _Name
    char[] name = student._Name.ToCharArray();
    int length = Math.Min(student._Name.Length, MAX_NAME_LENGTH);
    if (name.Length != MAX_NAME_LENGTH)
    {
        char [] tmp = new char[MAX_NAME_LENGTH];
        int i = 0;
        while (i < length)
        {
            tmp[i] = name[i];
            i++;
        }
        while (i < MAX_NAME_LENGTH)
        {
            tmp[i++] = '\0';
        }
        name = tmp;
    }

    bw.Write(name, 0, name.Length);         // Write out _Name
    bw.Write(student._BirthYear);           // Write out _BirthYear
    bw.Write(student._BirthMonth);          // Write out _BirthMonth
    bw.Write(student._BirthDay);            // Write out _BirthDay
}
```

Figure 11.4: Method to Write a Record to a Binary File using Random Access

The `ToStream()` method works by first creating an object of type `BinaryWriter`, which allows us to write out the binary data from (fixed length) numeric types, rather than converting them to (variable length) textual data. The next step in the method is then to write out the length of the name field that is being stored. The length of the name stored in the object may be less than, equal to, or greater than the length of the text field in the record, defined by the `MAX_NAME_LENGTH` constant. If the length of the name stored in the object is greater than the field width, then the name will be truncated in order to fit. Otherwise, any remaining characters are set to null (`'\0'`) before the name field is written out. Finally, the birth year, month, and day fields are also written out.

This completes the example code contained in this session to show how random access files binary files are used. The `FromStream()` method is similar to the `ToStream()` method except data is being read instead of written, and this method along with the remainder of the `Student` class will be available in CloudDeakin for you to examine further.

11.2.4. Object Serialisation

In examining files we have so far considered how to read and write text files using sequential access and binary files using random access. In each case however, we have only seen how to read and write the individual fields of an object. An alternative approach is ***object serialisation***, which allows us to convert an entire object into a sequence of bytes to be written to a file and covert the same sequence back later. Three types of object serialisation are provided by Microsoft.Net: binary serialisation, XML serialisation, and SOAP serialisation. Each of these has their own advantages and disadvantages, but for the purpose of this unit they are mostly equivalent, except you should note that XML serialisation only considers the `public` elements of a class (`private` and `protected` elements are ignored).

The first step in using serialisation is to mark the classes as serialisable, which is done by adding the `[Serializable]` attribute to each class whose objects will be serialised/deserialised, as follows:

```
using System; // Serializable attribute is defined in System namespace

...

[Serializable]
class class_name
{
    ...
}
```

Once this attribute is added to all classes to be serialised, the actual serialisation code is relatively straightforward.

Binary Serialisation

Required `using` statements:

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

Writing/serialising an object:

```
FileStream fs = File.Open(@"filename", FileMode.Create, FileAccess.Write);
BinaryFormatter bf = new BinaryFormatter();
bf.Serialize(fs, object_name);
fs.Close();
```

Reading/deserialising an object:

```
FileStream fs = File.Open(@"filename", FileMode.Open, FileAccess.Read);
BinaryFormatter bf = new BinaryFormatter();
object_name = (class_name) bf.Deserialize(fs);
fs.Close();
```

SOAP Serialisation

Required `using` statements[2]:

```
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
```

Writing/serialising an object:

```
FileStream fs = File.Open(@"filename", FileMode.Create, FileAccess.Write);
SoapFormatter sf = new SoapFormatter();
sf.Serialize(fs, object_name);
fs.Close();
```

Reading/deserialising an object:

```
FileStream fs = File.Open(@"filename", FileMode.Open, FileAccess.Read);
SoapFormatter sf = new SoapFormatter();
object_name = (class_name)sf.Deserialize(fs);
fs.Close();
```

XML Serialisation

Required `using` statements:

```
using System.Runtime.Serialization;
using System.Xml.Serialization;
```

Writing/serialising an object:

```
FileStream fs = File.Open(@"filename", FileMode.Create, FileAccess.Write);
XmlSerializer xs = new XmlSerializer(typeof(class_name));
xs.Serialize(fs, object_name);
fs.Close();
```

Reading/deserialising an object:

```
FileStream fs = File.Open(@"filename", FileMode.Open, FileAccess.Read);
XmlSerializer xs = new XmlSerializer(typeof(class_name));
object_name = (class_name)xs.Deserialize(fs);
fs.Close();
```

Task 11.1. Text File I/O

Objective: Learning to work with text files is the first step towards mastering the use of files in applications. For this task we will begin simply, by writing two short programs to write and then read information for a single class.

Begin this task by writing a small `Book` class, with the following elements:

[2] Note that you may receive an error message from Visual Studio when adding the second using statement for SOAP indicating that the namespace does not exist. If you receive this error message, you need to manually add a reference to the relevant library. To do so, in Visual Studio select Add Reference from the Project menu, then select Framework on the left hand side (under Assemblies), then select the `System.Runtime.Serialization.Formatters.Soap` and place a tick in the checkbox that appears next to it, then click the OK button.

- Instance variables and encapsulating properties for author, title, edition, publisher, and year of publication;
- A custom constructor with matching parameters used to initialise each of the above elements;
- A `ToString()` method that displays information in the format:
  author`, "`*title*`", Edition `*edition*`, `*publisher*`, `*year*`.`
  e.g.,
  ```
  Deitel, P.J., and Deitel, H.M., "Visual C# 2012: How to Program", Edition
  5, Pearson Education, 2013.
  ```

You are now required to write two short programs. Your first program should do the following:

- Create an array containing a number of `Book` objects representing different books;
- Create a file `"books.txt"` which contains the information for these books.

The second program should do the following:

- Open the file created in the first program
- Recreate the array of book objects from the first program, and displays them using a `foreach` loop and the `Book.ToString()` method.

Note: Using files is no different to using the Console – you should write data to the file in the same manner as you would expect to read it from the Console.

Hint: To recreate the array as required in the second program you will first need to know the size of the array!

Task 11.2. Serialisation

Objective: Serialisation is a much quicker way to read and write data to files, although it is restricted to sequential access only. For this task, we explore serialisation as an alternative to traditional file input and output.

Modify your solution/s to Task 11.1 to use SOAP serialisation instead of traditional file input and output.

Once you have completed this task, open your data file in a text editor such as Notepad. What do you notice about this code?

11.3. Generics

Often in software development there is a need to deal with a number of objects of different type/s and/or of unknown type/s. For example, consider writing a class that manages a queue of some type of object, e.g., a queue of print jobs, a queue of transactions, a queue of commands entered, and so on. In these examples, the concept of the queue does not change at all, only the type of data being placed in the queue.

The simplest way to define methods and classes to deal with different types of objects is use the `object` type – recall from Section 6.2 that every type in a C# application inherits from the `object` / `System.Object` class. Thus, we can refer to any object using references of this base class type. There is a major disadvantage to this approach however. Consider an array of objects, as follows:

```
object[] objArray = new object[3];
objArray[0] = 3;
objArray[1] = '3';
objArray[2] = "three";
```

The above code is legal in the C# programming language, however it is also suggestive of the major problem of using the object class. In particular, there is no checking of the types of data being added to the array of objects. Although adding objects of different types causes no issue at all, retrieving objects from such collections is usually both inefficient and error prone.

Generics allow both methods (***generic methods***) and classes (***generic classes***) to be defined where the data type used is deferred until the method or class is actually used. Consider the following method that is written using generics

```
TYPE Minimum<TYPE>(TYPE first, TYPE second)
    where TYPE : IComparable<TYPE>
{
    TYPE result = first;
    if (second.CompareTo(first) < 0)
        result = second;
    return result;
}
```

The method takes two parameters, first and second, which are compared and the minimum value of the two is returned. This method is similar to many methods we have already developed, except that the data type of the parameters and return type are indicated as `TYPE` instead of a data type we are familiar with. This is because the data type is specified as generic, indicated by `<TYPE>` appearing after the method name. Several different data types can be specified using a comma separated list inside the angled brackets ('<' and '>').

Note that the deferred data type (whose name is usually capitalised) can be used for parameters, the return value, and local variables, and that a constraint has been placed on the unspecified `TYPE`, requiring it to implement the interface `IComparable<>`, as follows:

```
where TYPE : IComparable<TYPE>
```

Note that it is also possible to apply generics to whole classes, using the following syntax:

```
class  class_name<type_name[, ...]>
       [where  type_name : class_or_interface
       [...]]
{
```

```
        …
}
```

The syntax for creating objects from generic classes is already familiar to us, as we have been using the generic `List<>` type since Session 3, as follows:

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

Note that it is also possible to nest the use of generic types as well, as follows:

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

Also note in the example above how a space appears between the two greater than symbols ('`>`') when specifying the type. Historically this space was necessary otherwise the compiler read the code as the right-shift operator ('`>>`'), instead of closing the data type, however modern versions of the Microsoft C# compiler no longer require this (other compilers and/or other languages using a similar syntax for generics may still require the space however).

Task 11.3. Generics

Objective: Although the real use for generics becomes clear when considering collections of data (a key topic of SIT221 Classes, Libraries and Algorithms), they are rapidly becoming a critical skill for all developerst – regardless of your knowledge of data structures and algorithms, there is a wide variety of collections that use generics both in Microsoft.Net but also in other languages such as C++ (in the form of the Standard Template Library). In this task, we will explore the development of a generic class.

For this task, you are required to write a class that finds the largest and smallest values of some data type. For this purpose, your class must satisfy the following requirements:

- Exploits generics with a single deferred type with an `IComparable<>` constraint;
- A `List<>` should be created (`private`) to contain objects of the deferred type;
- A method `Add()` should be provided to allow the objects to be added to the `List<>`;
- A method `Minimum()` should return the minimum value stored in the `List<>`; and
- A method `Maximum()` should return the maximum value stored in the `List<>`.

Finally, write a `Main()` method that demonstrates your generic class being used.