C# and Object-Oriented Programming

Course Notes for

CIS 1222

These notes belong to

Name: ___________________________________

Mobile: ______________

Prepared and compiled by

Matthew Xuereb

Contact details

Name: Matthew Xuereb

Email: matthew.xuereb@um.edu.mt

URL: www.matthewxuereb.com

Copyright

Preface

Various books and websites were used for the compilation of these notes. The readers of these notes are encouraged to use other books and the web to find more C# and OOP examples.

Table of Contents

Object Oriented Programming using C# 4
Arrays, Strings and Collections 18
Exception Handling 28
Structs 32
Enums 33
Operator overloading 36
File Handling in C# 37
GUI Tutorial 40
Delegates and Events 41
Language Integrated Query (LINQ) 47
Multithreading 54
Reflection 56

Object-Oriented Programming using C#

Object oriented Programming is a type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can *inherit* characteristics from other objects.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify and maintain.

C# is a native object-oriented language, and one can say that in C# everything is defined as a class.

Classes and objects

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. On the other hand, a class is a blueprint or prototype from which objects are created. In short, **an object is an instance of a class**.

Declaring a class

A class is like a *record* in Pascal or a *struct* in the C programming language, in that it stores related data fields, where the data fields can be of different types. So, you could for example store a test string in one filed, and an integer in another field and a floating point in a third field. The difference between a class and a record (or *struct*) is that a class also defines methods to work on the data fields.

A C# class has the following structure:

```
class <class_identifier>
{
      …attributes…
      …constructors…
      …methods…
}
```

The *class_identifier* represents the name of the class. The body of the class consists of three types of elements:

- Properties (data fields or attributes)
- Constructors
- Methods

Note that these elements can be declared in any order. However, to keep the code as readable as possible it is advised to keep the order as shown above, i.e., first the properties, then the constructors and then the methods. All these elements are optional, that is a class body does not need to have any variables or constructors or methods at all.

The following example illustrates a class that represents a Person:

```
public class Person
{

}
```

Attributes/Properties

Attributes are variables therefore each attribute has an identifier and an associated data type. Every time an object is created, memory is allocated for each attribute of the object. Attributes are defined in a class in a similar way that we define variables, with one difference – an attribute can have an access modifier. The example below defines some attributes for the class Person:

```
public class Person
{
      // Attributes
      public string name;
      public string surname;
      public int age;
}
```

Note the *public* access modifier in the above attributes' declarations. In C# there are four basic types of access modifiers (*public*, *private*, *protected,* and *internal*). The access modifiers will be explained later. However, note that if no access modifier is included with an attribute as shown in same example below, by default C# treats them as if they were defined as *internal*.

```
public class Person
{
      // Attributes
      string name;
      string surname;
      int age;

}
```

Methods

Methods are the object-oriented language term for procedures, functions, or subroutines in other types of programming languages. Methods are containers of a group of C# programming statements that perform a specific task.

Each method has a name, a return data type, zero or more parameters and a body. The method body contains the statements that evaluate the input parameters, if any, and produces the return value. Note that methods that do not require a return value specify the **void** keyword instead of a return data type.

In the following example, a method called *DisplayMessage* is added to the class *Person*. This method is used to display a hello message on the screen.

```
public class Person
{
      // Attributes
      public string name;
      public string surname;
      public int age;

      // A method used to display a hello message
      public void DisplayMessage()
      {
            Console.WriteLine("Hello Person");
      }
}
```

As in the attribute declaration, the method declaration begins with the access modifier. In the above example it is *public*. Note that if the access modifier is omitted by default, it is taken as *public*. After the access modifier, the return type is written. In the above example it is *void*, i.e., this method is not going to return anything. Following the return data type there is the method identifier (name) followed by a (). The () is an empty set of parameters. This means that this method does not take any parameters.

The body of the method contains statements that perform the method's task. In this case this method prints the name, surname, and age of the person object on the screen. In the following example, another method *DisplayDetails* is added. This time this method is used to print the details of the person object on the screen.

```
public class Person
{
      // Attributes
      public string name;
      public string surname;
      public int age;

      // A method used to display a hello message
      public void DisplayMessage()
      {
            Console.WriteLine ("Hello Person");
      }

      // A method used to display the details of the person
      public void DisplayDetails()
      {
            Console.WriteLine("NAME    - " + name);
            Console.WriteLine("SURNAME - " + surname);
            Console.WriteLine("AGE     - " + age);
      }
}
```

Main methods

A class can have many methods, and a program can be composed on many classes. However, in a program there can only be one *Main* method. The *Main* method can be taught of as being the driver of the program. The reason is that in C# every program execution starts from the

Main method. The *Main* method can be either written in a separate class or else in a class such as the Person class written in this example.

The following code snippet illustrates the class Person with a *Main* method. In this main method an object (instance) of a class Person is created. The method *DisplayMessage()* is invoked. Notice the use of the “dot-notation”. The fields *name*, *surname* and *age* are then given a value and then the method *DisplayDetails()* is invoked.

```
public class Person
{
      // Attributes
      public string name;
      public string surname;
      public int age;

      // A method used to display a hello message
      public void DisplayMessage()
      {
            Console.WriteLine (“Hello Person”);
      }

      // A method used to display the details of the person
      public void DisplayDetails()
      {
            Console.WriteLine(“NAME    - “ + name);
            Console.WriteLine(“SURNAME - “ + surname);
            Console.WriteLine(“AGE     - “ + age);
      }

      public static void Main(string[] args)
      {
            Person p = new Person();
            p.name = "Matthew";
            p.surname = "Xuereb";
            p.age = 26;
            p.DisplaMessage();
            p.DisplayDetails();
      }
}
```

Exercises (OOP Classes and Objects)

1. Copy the above class and run it.

2. Add some other fields and method to the above class such as telephone number and address and test them.

Methods with parameters

As outlined above, a method can have parameters. Parameters allow a method to accept values that can be used by that method. The following code snippet illustrates a method named *DisplayGreeting()* that accepts a parameter *greeting* and makes use of it as shown in the code snippet below.

```
public void DisplayGreeting(string greeting){
      Console.WriteLine("{0} {1}. Welcome to C# programming",greeting,name);
}
```

Methods with return values

In the examples above, all the methods do not return any values, i.e., the return type is defined as *void*. In C# a method can return a value. To define a method that returns a value, the data type of the value to be returned by the method should be declared within the method instead of the *void* keyword. The *return* keyword is used to instruct the method to return the actual value.

```
      public string GetNameAndSurname()
      {
            string ans = name + " " + surname;
            return ans;
      }
```

Note that the return data type can be either a primitive data type (such as int, boolean, …) or a C# pre-defined object or else a programmer defined object.

Exercises (Methods)

3. Add the above *GetNameAndSurname()* method to the class Person and invoke it from the *Main* method to test is.

4. Create a new class and write a *Main* method. Write code in this *Main* method that creates three instances of *Person p1*, *p2* and *p3*. Experiment and play around with these three *Person* instances by setting the fields and invoking their methods.

5. Create a new class *Rectangle*. Define two fields (length and breadth) for this class. Write the following three methods for this class:

i. Method *DisplayDetails()*	A method to display the details
ii. Method *CalculateArea()*	A method to return the area
iii. Method *CalculatePer()*	A method to return the perimeter

   Write a *Main* method to test this class. In the main method create an instance of this *Rectangle*, set the data fields, and invoke the three methods.

Encapsulation

Encapsulation is the ability of an object to be a container for related properties and methods. Older languages did not enforce any property/method relationships. This often resulted in side effects where variables had their contents changed or reused in unexpected ways and spaghetti code that was difficult to unravel, understand and maintain.

Encapsulation is one of three fundamental principles in object-oriented programming. Data hiding is the ability of objects to shield variables from external access. It is a useful consequence of the encapsulation principle. Those variables marked as private can only be seen or modified using public accessor and mutator methods (get and set methods). This permits validity checking at run time. Access to other variables can be allowed but with tight control on how it is done. Methods can also be completely hidden from external use. Those that are made visible externally can only be called by using the object's front door.

Basically, when using the encapsulation technique, all the properties (data fields) are declared as **private** (and in some cases when there is inheritance as **protected**). Since these data fields are set to be private, they cannot be accessed for outside their class by invoking the using the dot notation (such as *p1.name*). Instead, getter and setter methods are used.

Getters

Getters are methods that are used to return a particular variable. For instance, suppose we have a class Person, and this class consists of three data fields *name*, *surname* and *age* that are set as private as shown below.

```
public class Person
{
      // Attributes
      private string name;
      private string surname;
      private int age;

}
```

Since the data fields are set as private their value cannot be accessed from outside that class. In order to be able to get the value of such data fields, getter methods are written as shown in the code snippet below. Note the naming convention used for the getter methods – every getter method starts with the word *Get* followed by the data field identifier starting with an uppercase letter.

```
public class Person
{
      // Attributes
      private string name;
      private string surname;
      private int age;

      public String GetName()
      {
            return name;
      }

      public String GetSurname()
      {
            return surname;
      }

      public int GetAge()
      {
            return age;
      }
}
```

Setters

Setters are methods used to set a particular value to a particular data field. Consider the above class *Person* example, the example below continues to build on the previous one, this time also adding the setter methods.

```
public class Person
{
      // Attributes
      private string name;
      private string surname;
      private int age;

      // Setters
      public void SetName(string name)
      {
            this.name = name;
      }

      public void SetSurname(string surname)
      {
            this.surname = surname;
      }

      public void SetAge(int age)
      {
            this.age = age;
      }

      // Getters
      public string GetName()
      {
            return name;
      }

      public string GetSurname()
      {
            return surname;
      }

      public int GetAge()
      {
            return age;
      }
}
```

The *this* keyword

Notice in the above example the use of the **this** keyword. The **this** keyword is a reference to the current object — the object whose method is being called. You can refer to any member of the current object from within an instance method by using the keyword **this.**

Usually the **this** keyword is used when a parameter is passed to a method and this parameter has the same identifier as another data field in that class. The **this** is used to access the data field of that class and not the parameter or variable of a particular method.

Encapsulation using Properties

Properties are a new language feature introduced with C#. Only a few languages support this property. Properties in C# helps in protecting a field in a class by reading and writing to it. The above (getter/setter) method itself is good but encapsulation can be accomplished much smoother with properties. Consider the following class *Person* example:

```
public class Person
{
        private string name;
        private string surname;
        private int age;

        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        public string Surname
        {
            get { return surname; }
            set { surname = value; }
        }

        public int Age
        {
            get {  return age; }
            set
            {
                if (value >= 0 && value <= 110)
                {
                    age = value;
                }
                else
                {
                    Console.Error.WriteLine("Invalid age");
                }
            }
        }
      …
      …
}
```

Exercises

6. Copy the above class *Person* example. Create a main method for it and create an instance of *Person* and experiment with the *getters* and *setters*.

7. Create a class *Rectangle*. Define two fields (length and breadth) for this class. Make sure to use the concept of encapsulation (i.e. include the getters and setters for the length and breadth fields). Write the following three methods for this class:

i.	Method *DisplayDetails()*	A method to display the details
ii.	Method *GetArea()*	A method to return the area
iii.	Method *GetPer()*	A method to return the perimeter

Write a main method to test this class. In the main method create an instance of the *Rectangle*, set the data fields using the setters and invoke the above three methods.

8. Create a class *Circle*. Define only one data field *radius* for this class and make sure to use the encapsulation principle. Write the following three methods for this class:

i.	Method *GetArea()*	A method the return the area
ii.	Method *GetCircumference()*	A method to get the circumference
iii.	Method *GetDiameter()*	A method to get the diameter
iv.	Method *DisplayDetails()*	A method to display all the details

Write a main method to test this class.

Constructors

A constructor is a special method for initialising a new instance of a class. The constructor method for a class will have the same name as the class. A class may have multiple constructors. In this case, each constructor will have the same name, but will have different arguments. A *no-argument constructor* is a constructor which does not take any arguments.

Let's take as an example the above *Person* class. To create an instance of person, first a Person has to be initialised using the **new** keyword and then the *name*, *surname* and *age* of that instance of person has to be set individually using the setter methods after the initialisation of the object. Constructors provide us with the capability to set such fields automatically during the initialisation of the object by passing the values of these fields as parameters to the object initialisation, for instance, consider the following code example with a constructor for class *Person*:

```
public class Person
{
        private string name;
        private string surname;
        private int age;

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

        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        public string Surname
        {
            get { return surname; }
            set { surname = value; }
        }
```

```
            public int Age
            {
                get {  return age; }
                set
                {
                    if (value >= 0 && value <= 110)
                    {
                        age = value;
                    }
                    else
                    {
                        Console.Error.WriteLine("Invalid age");
                    }
                }
            }
          …
          …
    }
```

The above class can be initiated as follows:

```
            Person p = new Person("Matthew","Xuereb",23);
```

Note that a class can have more than one constructor. For instance in the example below, class Person has three constructors, one of them empty, meaning that an instance of that class can be created without having to pass to it any parameters.

```
    public class Person
    {
          // Attributes
          private string name;
          private string surname;
          private int age;

          // Empty constructor
          public Person()
          {
          }

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

          // Another Constructor
          public Person(string name,string surname)
          {
                this.name = name;
                this.surname = surname;
          }
          …
          …
    }
```

Exercises (Constructors)

9. Modify the programs of the previous exercises and make them with constructors.

Inheritance

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality. An example of where this could be useful is with a school records system. You could create a generic person class with states and actions that are common to all persons. Then more specific classes could be defined for students, teachers and administrative staff.

The generic class is known as the parent (or superclass or base class) and the specific classes as children (or subclasses or derived classes). The concept of inheritance greatly enhances the ability to reuse code as well as making design a much simpler and cleaner process.

The inheritance symbol

C# uses the ***:*** symbol to indicate the inheritance relationship between a parent class and a child class. For example, using our person class, let's create another class *Teacher* that inherits from class person. Note that a teacher has all the properties of a person, plus some extra properties – in this case *subject* and *qualification*.

```
public class Teacher : Person
{
    private string subject;
    private string qualification;

    public string Subject
    {
      get { return subject; }
      set { subject = value; }
    }

    public string Qualifications
    {
      get { return qualifications; }
      set { qualifications = value; }
    }
}
```

Note that since the above class inherits from Person, class *Teacher* ***is a*** also a *Person*. Class teacher has all the public methods and attributes of class Teacher. That is, if an instance of *Teacher* is created, the programmer can invoke also the public methods of class *Person* as shown in the code snippet below:

```
Teacher t1 = new Teacher();
t1.setName("Matthew");
t1.setSurname("Xuereb");
t1.setAge(23);
t1.setQualification("BSc IT (Hons.)");
t1.setSubject("Computing");
```

Exercises

10. Copy the above class *Teacher* and create a main method to test the code snippet that makes use of this class as shown above.

11. Create a class *Student* that inherits from class *Person*. The properties of this class should be *className* (e.g. 3B), *optionalSubject1* (e.g. Biology), *optionalSubject2, optionalSubject3*. Make sure to create the getters and setters for all the properties. Create a main method to test this class.

12. Create a class *AdministrativeStaff* that inherits from class *Person*. The properties of this class should be *position* (e.g. Assistant Head) and *qualification*. Make sure to use encapsulation for the properties. Create a main method to test this class.

The protected access modifier

Till now the only two access modifiers used were *public* and *private*. However, suppose that you want to be able to access the fields of a particular class from its sub classes however you don't want to allow access of these fields from other classes. This can be done by using the *protected* access modifier. In the code snippet below, all the fields of class Person has been changed to *protected*.

```
public class Person
{
    protected string name;
    protected string surname;
    protected int age;

      …
      …
      …
}
```

The above *protected* fields can be accessed from subclasses of class *Person* as if they were fields of the same subclass. For example, in the code snippet below, a method *DisplayDetails()* in class *Teacher* is using these fields.

```
public class Teacher : Person
{
      …
      …
    public void DisplayDetails()
      {
        Console.WriteLine("DISPLAYING THE DETAILS OF A TEACHER:");
        Console.WriteLine("NAME: " + name);
        Console.WriteLine("Surname: " + surname);
        Console.WriteLine("AGE: " + age);
        Console.WriteLine("Qualification: " + qualification);
        Console.WriteLine("Subject: " + subject);
    }
      …
      …
}
```

Exercises

13. Change the fields of class Person to protected as shown above and create a *DisplayDetails()* method for subclasses *Teacher*, *Student* and *AdministrativeStaff*.

Constructors in inherited classes

Sometimes it is necessary to invoke the constructor of a superclass from a derived class. This is possible by using the *super* keyword. For example, suppose that the constructor of the derived class *Teacher* takes as arguments the name, surname, age, qualification and subject. To set the name, surname and age without invoking the *Person* (superclass) constructor, the *Teacher* constructor should be as shown in the code snippet below:

```
public Teacher(string name,string surname,int age, string quailification,string subject)
{
      this.name = name;
      this.surname = surname;
      this.age = age;
      this.qualification = qualification;
      this.subject = subject;
 }
```

To make the constructor contents shorter, the constructor of the superclass *Parent* can be invoked using the *base* keyword as shown below:

```
public Teacher(string name,string surname,int age, string quailification,string subject)
      : base(name,surname,age);
{
      this.qualification = qualification;
      this.subject = subject;
}
```

Exercises

14. Create constructors for the subclasses of *Person*. Make sure to make use of the *super* keyword.

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Any C# object that can pass more than an IS-A test is considered to be polymorphic. In C#, all objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared the type of a reference variable cannot be changed.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

We can say that in our previous examples, where classes *Teacher*, *Student* and *AdministrativeStaff* were inheriting from class *Teacher*, is a form of polymorphism. In fact, an instance of teacher can now be created either using the following code:

```
Teacher t = new Teacher(); // Not polymorphic
```

or

```
Person t = new Teacher();  // Polymorphic
```

Note that polymorphism is used a lot in conjunction with arrays and lists. These are explained in the next topics.

The *abstract* keyword

The *abstract* keyword is used to define an abstract class or an abstract method. An abstract class is a class that cannot be initiated while an abstract method is a method that resides in an abstract class and that does not contain any implementation. Consider the following example:

```
public abstract class Person
{
      …
      …
      public abstract double GetFixedSalary();
      …
      …
}
```

The above code snippet shows class Person with an abstract method *GetFixedSalary()*. This method is meant to return the yearly fixed salary of a person. However, as *Person* is a generic class, the implementation of such method can only be in the subclasses of Person as these are more specialised. If a class contains an abstract class, it has to be set to abstract. The reason being that now it is not possible to create an instance (object) of that class.

The subclasses of an abstract class such as *Person* must provide an implementation for the abstract methods. Note that the override keyword should be included in the definition of a method that is implementing an abstract class. Consider the following code snippet:

```
public class Teacher : Person
{
      …
      …
      public override double GetFixedSalary()
      {
            return 20750.00;
      }
      …
      …
}
```

In the above code snippet the abstract method *GetFixedSalary()* is implemented for class *Teacher*. Obviously, a different implementation of this method is required for all the classes that inherit from class *Person*. More information and examples about polymorphism and abstract classes will be given after arrays and lists are covered in the next topics.

Arrays, Strings and Collections

Arrays

Arrays can be informally defined as a group of variables containing values of the same type and that in some way or another they are related. An array has a fixed size that is defined before the array is created in memory. The elements of an array are accessed by indexing.

In C# arrays are reference objects, that is, they are allocated on the heap space of the memory. Arrays can be either single dimension or multi-dimensional arrays.

Declaring an array

Arrays can be declared using the follow syntax:

```
int[] nums = new int[5];
```

-- OR –

```
int[] nums;
nums = new int[5];
```

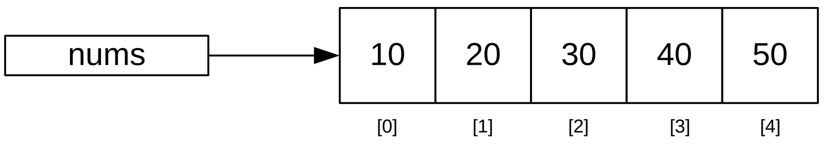
When an array is created, it initially contains the default values for the types of the elements of that array. For example if the array is an array of integers then the default value is 0 while if the array is an array of a particular object the default value is null.

Initialising an array

Arrays can be initialized at the same time as they are created. In this case the following syntax can be used:

```
int[] nums = {10,20,30,40,50};
```

The above declaration and initialisation creates the following grid like data structure in memory:



Accessing an array

Arrays are accessed using an index for each element. Note that if an array has a size of for example 5, the range of its index is from 0 to 4. Consider the following example:

```
int[] nums = {10,20,30,40,50};
Console.WriteLine(nums[0]);
Console.WriteLine(nums[1]);
```

The above example will output on the screen the contents of the first and second element of array num, that is, 10 and 20. To set values to an array the same technique is used together with the assignment operator (=) as shown in the code snippet below:

```
nums[0] = 11;
nums[1] = 22;
```

Many times it does not make sense to access each array's item individually. Instead a loop, traditionally a *for loop* is used to access an array. Consider the following two code snippets that are displaying the contents of the above *nums* array – one by accessing the elements individually and the other one using a *for loop*:

`int[] nums = {10,20,30,40,50};`
 	
`Console.Writeline(nums[0]);`
`Console.Writeline(nums[1]);`
`Console.Writeline(nums[2]);`
`Console.Writeline(nums[3]);`
`Console.Writeline(nums[4]);`
	`int[] nums = {10,20,30,40,50};`
	
	`for (int i = 0; i < nums.Length; i++)`
	`{`
	`    Console.WriteLine(nums[i]);`
	`}`

Accessing the items of the array individually	*Accessing the items of the array using a for loop*

Note that the *.Length* property return number of elements that there is in a particularly array. In C#, there is derivative of a *for loop*, the *foreach loop*. The *foreach loop* can be used to iterate over the elements in and array and a collection. This loop can be used to access an array as shown in the code snippet below:

```
int[] nums = {10,20,30,40,50};
foreach (int number in nums)
{
      Console.WriteLine(number);
}
```

Multidimensional arrays

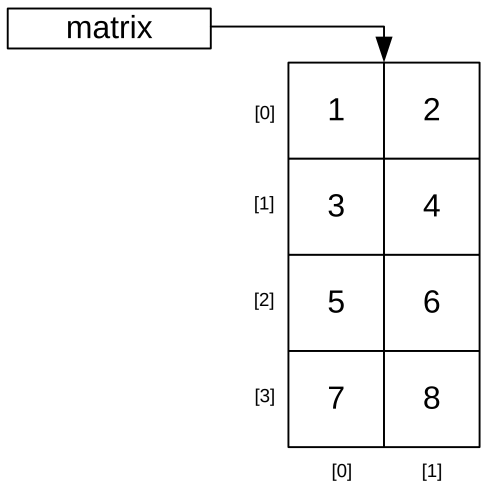
Multidimensional arrays are arrays with more than one dimension. For instance, the following code snippet declares a two-dimensional array:

```
int[,] matrix = new int[4,2];
```

-- OR --

```
int[,] matrix = {{1,2} , {3,4} , {5,6} , {7,8}};
```

The above declaration and initialisation create the following grid like data structure in memory:



The above *matrix* array is a 2D (two-dimensional) array and it has a first dimension of 4 and a second dimension of 2. Note that C# also supports 3D and 4D arrays. Multidimensional arrays are sometimes called rectangular arrays because elements can be written in a rectangular grid or cubes.

Jagged arrays

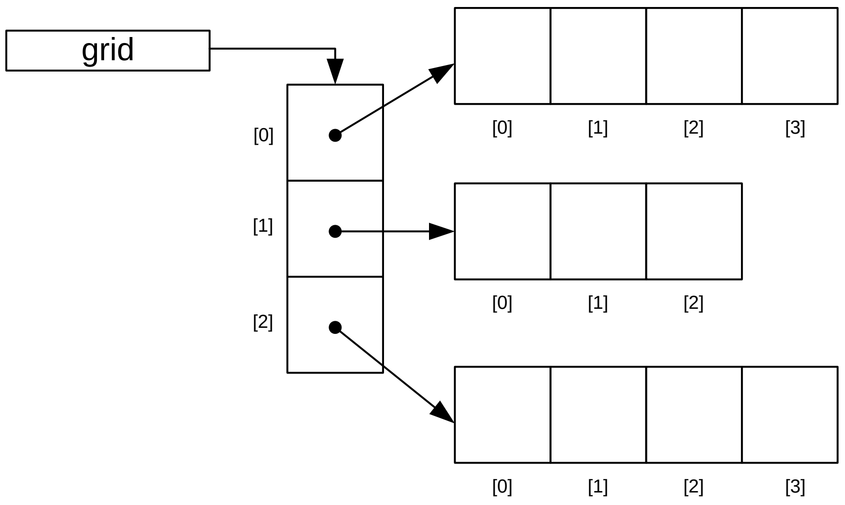
C# also supports jagged arrays. These can be seen as an array of arrays because in this case the array does not have to be rectangular. Consider the following example:

```
int[][] grid = new int[3][];
grid[0] = new int[4];
grid[1] = new int[3];
grid[2] = new int[4];
```

-- OR --

```
int[][] grid = {new int[4] , new int[2] , new int[4]};
```

The above declaration creates the following grid like data structure in memory:

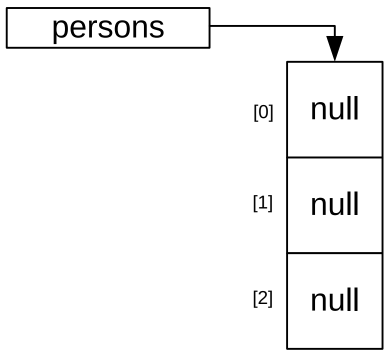


Arrays of objects

When creating an array of objects the elements of the array are initialised to null. Consider the following example:

```
Person[] myPersons = new Persons[3];
```

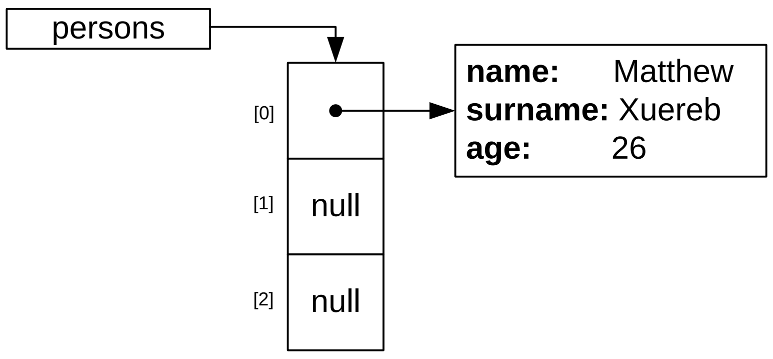
When the above code snippet is executed, although an array of Persons is created, it is not an array of *Person* object but simply an array of references to *Person* instances. Therefore, the items of the array will simply contain the null value as shown in the diagram below:



A new instance of Person must be initialised in each array item so that the element is really pointing to an instance of Person. Consider the code snippet below:

```
persons[0] = new Person("Matthew","Xuereb",26);
```

When the above line of code is executed, the memory will contain a data structure as shown in the diagram below:



Exercises:

15. Write a program that asks the user to enter 10 examination marks, store them in an array and then find the average mark, the minimum mark, and the maximum mark.

16. Write a program that asks the user to enter 5 telephone numbers and stores them in an array. The program should ask the user to enter another telephone number and checks if that number is stored in the array or not using a for loop or a foreach loop. If the entered telephone number is found in the array the program should print on the

screen FOUND, else NOT FOUND. (Store the telephone number as a string and not as an integer).

17. Create a class *Matrix* that will store in it a matrix. The constructor should accept a 2D array as an argument. This class should contain the following methods:

a. *void ShowMatrix()*	To display the matrix on the screen.
b. *Matrix MultiplyMatrix(Matrix m)*	To multiply the matrix with another matrix.

Test you *Matrix* class by writing a program that will perform the following matrix multiplication:

$$\begin{pmatrix} 4 & 3 & 6 \\ 5 & 9 & 8 \\ 4 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 4 & 9 \\ 2 & 0 & 7 \\ 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 28 & 28 & 63 \\ 47 & 36 & 116 \\ 4 & 8 & 25 \end{pmatrix}$$

Sorting and searching Arrays

Sorting and searching are functions that are available in the *System.Array* type. The *Sort()* function will sort the items of an array. The *IndexOf(), LastIndexOf()* and *BinarySearch()* functions are used to search items in the array. Consider the following examples:

```
int[] nums = {45,66,3,14,55};
Array.Sort(nums);
foreach (int number in nums)
{
     Console.WriteLine(number);
}
```

In the above code snippet, the contents of the *nums* array are sorted in ascending order. To sort in descending order, the *Array.Reverse()* method should be used just after the *Array.Sort()* is invoked as this will reverse all the contents of the array as shown in the code snippet below:

```
int[] nums = {45,66,3,14,55};
Array.Sort(nums);
Array.Reverse(nums);
```

The *Array.IndexOf()* searches for a particular item in an array. It returns the first found index of that item if found, otherwise it returns a -1. Consider the code snippet below:

```
string[] names = { "Paul", "Claire", "Sarah", "Matthew" };
Console.WriteLine(Array.IndexOf(names, "Sarah"));    // Outputs 2
Console.WriteLine(Array.IndexOf(names, "Michaela")); //Outputs -1
```

The *Array.LastIndexOf()* is like the *Array.IndexOf()* function however this returns the index of the last item in the lies if found. Consider the code snippet below:

```
int[] nums = { 45, 66, 3, 14, 55 , 3};
Console.WriteLine(Array.LastIndexOf(nums,3)); // Outputs 5
Console.WriteLine(Array.IndexOf(nums,1));     // Outputs -1
```

The *Array.BinarySearch()* performs a binary search on an array. Note that the array must be sorted in order for the *Array.BinarySearch()* function to operate correctly. This function returns the index of the item if found. Otherwise, a negative number is returned. Consider the following code snippet:

```
string[] names = { "Paul", "Claire", "Sarah", "Matthew"};
Array.Sort(names);
Console.WriteLine(Array.BinarySearch(names, "Paul"));
// Outputs 2
Console.WriteLine(Array.BinarySearch(names, "Michaela"));
//Outputs -3
```

Strings

In C# all strings are instances of the *System.String* type in the Common Language Runtime. Because of this, there are many built-in operations available that work with strings. The *String* class supports the following methods:

Method	**Description**
Compare()	Compares two strings.
CompareOrdinal()	Compares two string regions using an ordinal comparison.
CompareTo()	Compares the current instance with another instance.
EndsWith()	Determines whether a substring exists at the end of a string.
StartsWith()	Determines whether a substring exists at the beginning of a string.
IndexOf()	Returns the position of the first occurrence of a substring.
LastIndexOf()	Returns the position of the last occurrence of a substring.
Concat()	Concatenates two or more strings or objects together. If objects are passed, the *ToString()* function is called on them.
CopyTo()	Copies a specified number of characters from a location in this string into an array.
Insert()	Returns a new string with a substring inserted at a specific location.
Join()	Joins an array of strings together with a separator between each array element.
PadLeft()	Left aligns a string in field.
PadRight()	Right aligns a string in a field.
Remove()	Deletes characters from a string.
Replace()	Replaces all instances of a character with a different character.
Split()	Creates an array of strings by splitting a string at any occurrence of one or more characters.
Substrng()	Extracts a substring from a string.
ToLower()	Returns a lowercase version of a string.
ToUpper()	Returns an uppercase version of a string.
Trim()	Removes leading and trailing white space from a string.

TrimEnd()	Removes a string of characters from the end of a string.
TrimStart()	Removes a string of characters from the beginning of a string.

Consider the following code snippets:

```
string s1 = "   This is some text.      ";
Console.WriteLine(s1.Trim());
// OUTPUTS: This is some text.

string s2 = "Hello Luke!";
Console.WriteLine(s2.Replace("Luke", "Matthew"));
// OUTPUTS: Hello Matthew!

string s3 = "This is a String.";
Console.WriteLine(s3.ToUpper());
// OUTPUTS: THIS IS A STRING.
```

Collections

Collections are data structures that holds data in different ways for flexible operations. In C#, collection classes are defined as part of the *System.Collections* or *System.Collections.Generic* namespaces. Most collection classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialised data storage needs. The most important collections are the following:

- ArrayList and List
- HashTable
- Stack
- Queue

ArrayList

ArrayList is one of the mostly used data structures in object-oriented languages. An ArrayList contains a simple list of values. It implements the IList interface using an array and very easily we can add, insert, delete, view and more. It is very flexible because we can add without any size information, that is it will grow and shrink dynamically. Consider the following code snippet:

```
Person p1 = new Person("Matthew","Xuereb",26);
Person p2 = new Person("Luke","Borg",27);
Person p3 = new Person("Karen","Pisani",38);

ArrayList myPersons = new ArrayList();
myPersons.Add(p1);
myPersons.Add(p2);
myPersons.Add(p3);

foreach (Person p in myPersons)
{
     p.DisplayDetails();
}
```

There are various methods and properties that can be called on an ArrayList, namely:

- Add()
- Remove()
- RemoveAt()
- Insert()
- IndexOf()
- Count

List

The *List* collection is a variation of the *ArrayList* collection. The only difference is that while *ArrayList* takes any type of object, a *List* is typed and therefore can only take one type of object. For this reason, a *List* is less expensive to use and therefore when possible it is advisable to use it instead of *ArrayList*. Consider the following code snippet:

```
Person p1 = new Person("Matthew","Xuereb",26);
Person p2 = new Person("Luke","Borg",27);
Person p3 = new Person("Karen","Pisani",38);

List<Person> myPersons = new List<Person>();
myPersons.Add(p1);
myPersons.Add(p2);
myPersons.Add(p3);
```

HashTable

In C#, *HashTable* represents a collection of key/value pairs which maps keys to value. Any non-null object can be used as a key. We can retrieve items from a HashTable by providing the key. Consider the following code snippet:

```
Hashtable family = new Hashtable();

family.Add("Father", "Emanuel");
family.Add("Mother", "Mary");
family.Add("Children", "Michael, Karen, Matthew");

Console.WriteLine(family["Mother"]); // OUTPUTS: Mary
```

Stack

The Stack class represents a last-in-first-out (LIFO) Stack of Objects. Stack follows the push-pop operations. That is we can Push (insert) Items into Stack and Pop (retrieve) it back. Stack is implemented as a circular buffer. It follows the Last In First Out (LIFO) system. That is we can push the items into a stack and get it in reverse order. Stack returns the last item first. As elements are added to a Stack, the capacity is automatically increased as required through reallocation. Consider the following code snippet:

```
Stack<int> stack = new Stack<int>();
stack.Push(20);
stack.Push(30);
stack.Push(40);
```

```
while(stack.Count != 0)
{
      Console.Write("{0} ",stack.Pop());
}
// OUTPUTS: 40 30 20
```

Queue

The Queue works like FIFO system, a first-in, first-out collection of Objects. Objects stored in a Queue are inserted at one end and removed from the other. The Queue provide additional insertion, extraction, and inspection operations. We can Enqueue (add) items in Queue and we can Dequeue (remove from Queue) or we can Peek (that is we will get the reference of first item) item from Queue. Queue accepts null reference as a valid value and allows duplicate elements. Consider the following code snippet:

```
Queue<int> numbers = new Queue<int>();
numbers.Enqueue(55);
numbers.Enqueue(33);
numbers.Enqueue(88);

while (numbers.Count != 0)
{
      Console.Write("{0} ", numbers.Dequeue());
}
// OUTPUTS: 55 33 88
```

Exercises:

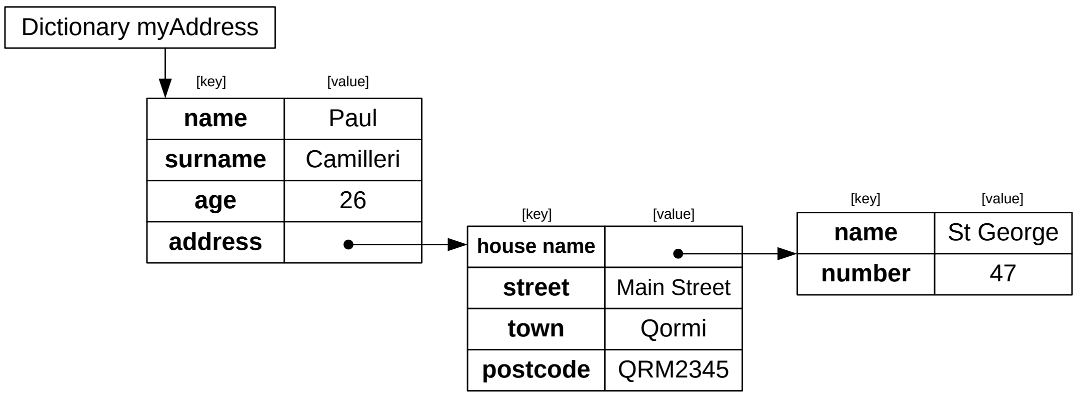
18. Repeat exercise 45 however this time use the *Array.BinarySearch()* function.

19. Write a program that asks the user to enter a simple mathematical expression in reverse polish notation (e.g. 1 4 +) and it works out the expression and outputs the total. You must use the Stack data structure.

    You can test your program with the following test data:

Input	Output
1 4 +	5
3 5 + 2 *	16
2 7 * 10 2 / +	19
3 4 + 5 +	12

20. Write a class called Dictionary. This class should be a variation of a hash table that can be used to store any kind of data. A value of a hash table can either store a value or else it can refer to another hash table as shown in the diagram below:



Implement the following methods:

void Put(object o)	To add a new item. The item can also be a primitive data type of else an instance of another hast table.
object Get(string key)	To return a particular item given its field.
string GetXML()	To return an XML version of the whole dictionary.

Also implement two constructors, an empty and parameter less constructor and a constructor that accepts a string with the XML version of the contents of the dictionary. The latter should populate the dictionary with the data that there is in the XML string that is passed as a parameter.

Exception Handling

Many times, in our programs errors occur or can occur. Some of these errors can be predicted, and therefore the programmer will know that they must be handled. However, there are occasions where such errors cannot be very well predicted. For instance, it is not possible to predict when there will be a file I/O error, run out of system memory, or encounter a database error. These things are generally unlikely, but they could still happen, and it is very important that we take care of them when they do occur. This is where exception handling comes in.

try..catch statements

Code that can throw any type of exception should be put in a *try..catch* statement. If an exception takes place while in a try block, then the program will not crash. Instead, the exception (error) will be caught by the program itself and program execution will continue from the *catch* block.

A *try..catch* statement has the following structure:

```
try
{
     ..
}
catch( _type of exception_  _exception reference_)
{
     ..
}
```

Consider the following example:

```
int[] nums = { 1, 2, 3, 4, 5 };
try
{
     Console.WriteLine(nums[4]);
     Console.WriteLine(nums[5]);
}
catch (Exception e)
{
     Console.WriteLine("An exception has occurred.");
}
Console.WriteLine("The program will now continue");
```

In the above code snippet, an exception is going to take place in the code that is written within the *try* block. The statement **`Console.WriteLine(nums[5]);`** is accessing an element in the array that does not exist and therefore an exception will take place. However, note that instead of crashing, once the exception occurs, the program flow is directed to *catch* block where an error message is prompted on the screen. The program will then continue its execution normally.

The code ***Exception e*** in the brackets of the *catch* block represent a reference to the exception that is effectively thrown by the program. Note that in this case *Exception* represents every type of exception. The reason being that Exception is the base class for every other type of exception that is supported by C#. However, exceptions can specifically caught by type as shown in the code snippet below:

```
try
{
     Console.WriteLine(nums[4]);
     Console.WriteLine(nums[5]);
}
catch (IndexOutOfRangeException e)
{
     Console.WriteLine("You are accessing out of range");
}
```

It is possible that a *try..catch* statement will have more than one catch statement so that if more than one exception can take place, these exceptions are caught according there type. Consider the code snippet below:

```
try
{
     ..
}
catch (IndexOutOfRangeException e)
{
     Console.WriteLine(e.ToString());
}
catch (FileNotFoundException e)
{
     Console.WriteLine(e.ToString());
}
catch (Exception e)
{
     Console.WriteLine(e.ToString());
}
```

Note that the ***e.ToString()*** statement will return information about the exception that occurred including the type of exception and line number.

finally blocks

An exception can leave a program in an inconsistent state by not releasing resources or doing some other type of cleanup. A *catch* block is a good place to figure out what may have gone wrong and try to recover, however it cannot account for all scenarios.

Sometimes whether or not a program succeeds, you need to perform clean up actions. These situations are good candidates for using a *finally* block. Code that is written in a *finally* block is executed independent of what occurred in the *try* block. Consider the following example:

```
FileStream outStream = null;
FileStream inStream = null;

try
{
     outStream = File.OpenWrite("DestinationFile.txt");
     inStream = File.OpenRead("BogusInputFile.txt");
}
catch(Exception ex)
{
     Console.WriteLine(ex.ToString());
}
finally
{
     if (outStream != null)
     {
          outStream.Close();
          Console.WriteLine("outStream closed.");
     }
     if (inStream != null)
     {
          inStream.Close();
          Console.WriteLine("inStream closed.");
     }
}
```

Throwing an exception

In C#, it is possible to throw an exception programmatically. The *throw* keyword is used for this purpose. Consider the flowing code snippet:

```
try
{
     throw new DivideByZeroException("Invalid Division");
}
catch(DivideByZeroException e)
{
     ..
}
```

In the above code snippet, a *DivideByZeroException* is thrown on purpose. Many times, this mechanism is used a lot in conjunction with user-defined exceptions.

User defined exceptions

In C#, it is possible to create tailor made exception class. In this case, class *Exception* must be the ultimate base class. So, the user-defined exception classes must inherit from either Exception class or one of its standard derived classes. Consider the following class:

```
public class MyException : Exception
{
     public MyException(string s)
     {
          Console.WriteLine("This is my tailor-made” +
                    exception with parameter --> " + s);
     }
}
```

The above class inherits from the *Exception* class and therefore in itself it is also an exception. Note that user defined exceptions can also have parameters. The following code snippets throws and catches the above user defined exception exception:

```
try
{
     throw new MyException("Hello");
}
catch (MyException e)
{
     Console.WriteLine(e.Message);
}
```

Exceptions should only be used to communicate exceptional conditions. They should not be used to communicate events that are expected, such as reaching the end of a file. If there is a good, predefined exception in the *System* namespace that describes the exception condition it should be used rather than defining a new exception class, and put specific information in the message.

Structs

Classes are used to implement most objects. However sometimes it may be desirable to create an object that behaves like one of the built-in types. In this case a value type is used and in C# this is done by declaring a ***struct***.

Structs act similarly to classes, but with few added restrictions. They cannot inherit from other types and other classes cannot inherit from them. Note that unlike classes, when a *struct* is created using the new keyword it is created and placed on the stack. Consider the following *Coordinate* example:

```
public struct Coordinate
{
    public int x;
    public int y;

    public Coordinate(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override string ToString()
    {
        return "(" + x + "," + y + ")";
    }
}

---------------------------------------------

public static void Main(string[] args)
{
     Coordinate c = new Coordinate(3, 4);
}
```

Structs should only be used for types that are just a piece of data - for types that could be used in a similar way to the built-in types. Even if more complex types can be implemented as value types, they probably should not be, since the value type semantics will probably not be expected by the user. The user will expect that a variable of the type could be *null*, which is not possible with value types.

Enums

An enumeration is a data type consisting of a set of named values that represent integral constants, known as enumeration constants. An enumeration also referred to as an enumerated type because you must list (enumerate) each of the values in creating a name for each of them. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values.

In other words, enums are strongly typed constants. They are essentially unique types that allow you to assign symbolic names to integral values. Consider the following definition of an *enum*:

```
public enum Days
{
     Mon, Tue, Wed, Thur, Fri, Sat, Sun
}
```

The above code snippet illustrates a definition of a enum with the days of the week. The above enum can be used as shown in the code snippet below:

```
Days today = Days.Mon;

switch (today)
{
     case Days.Mon:
          Console.WriteLine("Today is Monday");
          break;
     case Days.Tue:
          Console.WriteLine("Today is Tuesday");
          break;
     case Days.Wed:
          Console.WriteLine("Today is Wednesday");
          break;
     case Days.Thur:
          Console.WriteLine("Today is Thursday");
          break;
     case Days.Fri:
          Console.WriteLine("Today is Friday");
          break;
     case Days.Sat:
          Console.WriteLine("Today is Saturday");
          break;
     case Days.Sun:
          Console.WriteLine("Today is Sunday");
          break;
}
```

An enum is typically specified as shown above. However it may be customised by changing its base type and member values. By default, the underlying type of an enum is ***int***. This default may be changed by specifying a specific base when declaring the enum. You would specify a different base if the enum was used extensively and there was an opportunity for space savings by selecting a smaller type. Another reason may be if you wanted the underlying type of the enum to correspond to another type in your program and you wanted to explicitly cast between the two without loss of precision. Valid base types include ***byte***, ***sbyte***, ***short***, ***ushort***, ***int***, ***uint***, ***long***, and ***ulong***. Consider the following example:

```
public enum Days : byte
{
     Mon, Tue, Wed, Thur, Fri, Sat, Sun
}
```

Another modification you can make to an enum is to set the value of any enum member. By default, the first member of an enum takes the value of zero. If this value doesn't make sense for your enum, you can change it to one or some other number. Additionally, you can change any of the members of an enum to any value that is valid for its base type. Unassigned enum members have a value that is one more than their predecessor. Consider the following example:

```
public enum Days : byte
{
     Mon = 1, Tue, Wed, Thur, Fri, Sat, Sun
}

------

Console.Write("Enter the day of the week - ");
byte dayOfWeek = byte.Parse(Console.ReadLine());

Days today = (Days)dayOfWeek;

switch (today)
{
     case Days.Mon:
          Console.WriteLine("Today is Monday");
          break;
     case Days.Tue:
          Console.WriteLine("Today is Tuesday");
          break;
     case Days.Wed:
          Console.WriteLine("Today is Wednesday");
          break;
     case Days.Thur:
          Console.WriteLine("Today is Thursday");
          break;
     case Days.Fri:
          Console.WriteLine("Today is Friday");
          break;
     case Days.Sat:
          Console.WriteLine("Today is Saturday");
          break;
```

```
            case Days.Sun:
                 Console.WriteLine("Today is Sunday");
                 break;
      }
```

In the above example, a user integral input is casted to an enum.

Operator overloading

One of the nice features of C# is that you can give special meanings to operators, when they are used with user-defined classes. This is called operator overloading. The following set of operators is commonly overloaded for user-defined classes:

- = (assignment operator)
- + - * (binary arithmetic operators)
- += -= *= (compound assignment operators)
- == != (comparison operators)

To overload an operator, a static method with the *operator* keyword followed by the symbol of the operator should be defined. It is very important that this static method will have at least one of the parameters the same as the argument that is used for the operator overloading. Consider the following example:

```
public static Matrix operator *(Matrix m1, Matrix m2)
{
     return m1.MultiplyMatrix(m2);
}
```

The above method overloads the matrix multiplication that was implemented previously for *Question 46*.

C# enforces certain rules when you overload operators. One rule is that you must implement the operator overload in the type that will use it. This is sensible because it makes the type self-contained. Another rule is that you must implement matching operators. For example, if you overload ==, you must also implement !=. The same goes for <= and >=. When you implement an operator, its compound operator works also, for example, since the ***+*** operator for the ***Matrix*** type was implemented, you can also use the ***+=*** operator on ***Matrix*** types.

Exercises

21. Implement and test the above multiplication overloaded method in the matrix class of *Question 46*.

22. For the same Matrix class of *Question 46*, overload the *+*, *==* and *!=* operators and test them form a main method.

File Handling in C#

In C# there are various ways to handle files. In this note, code snippets of the basic functions required for file handling and examples of how to handle text files and serialisable files are given. Note that in practice, all file handling code should be surrounded with exception handling code so that if something fails, the program will not crash.

General file handling methods

Checking if a file exists

```
if(File.Exists("MyFile.txt"))
{
     Console.WriteLine("File exist");
}
else
{
     Console.WriteLine("File does not exist");
}
```

Creating a new directory (folder)

```
Directory.CreateDirectory("MySubDir");
```

Creating a new file

```
File.Create(FILE_DIR + "MyFile.bin");
```

Copying a file

```
File.Copy("MyFile.txt","MyFile2.txt",true);
```

Note the third argument in the above *Copy* method is used to indicate whether the [second] file should be overwritten or not if it already exists.

Delete a file

```
File.Delete("MyFile2.txt");
```

Moving a file

```
File.Move("MyFile.txt","MySubDir\\MyMovedFile.txt");
```

Listing files in a directory

```
string[] files = Directory.GetFiles(FILE_DIR);
foreach (String f in files)
{
     Console.WriteLine(f);
}
```

Listing sub directories in a directory

```
string[] dirs = Directory.GetDirectories(FILE_DIR);
foreach (String d in dirs)
{
     Console.WriteLine(d);
}
```

Text files

Writing text to a text file

```
StreamWriter myFile = File.CreateText("MyFile.txt");
myFile.WriteLine("This text will be stored in a text file");
myFile.Close();
```

Appending text to a text file

```
StreamWriter myFile = File.AppendText("MyFile.txt");
myFile.WriteLine("This text will be appended to the file");
myFile.Close();
```

Reading from a text file

```
string s;
StreamReader myFile = File.OpenText("MyFile.txt");
while ((s = myFile.ReadLine()) != null)
{
     Console.WriteLine(s);
}
myFile.Close();
```

Serialisation

Serialisation is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialisation.

Note that in two following two examples, the class *Person* is used.

Serialise an object to an XML file

```
const string FILE_PATH = "C:\\Users\\Matthew\\Test\\";

Person p = new Person("Matthew", "Xuereb", 26);
p.DisplayDetails();

XmlDocument xmlDocument = new XmlDocument();
XmlSerializer serializer = new XmlSerializer(p.GetType());

MemoryStream stream = new MemoryStream();
serializer.Serialize(stream, p);
stream.Position = 0;
xmlDocument.Load(stream);
xmlDocument.Save(FILE_PATH + "Person.xml");
stream.Close();
```

Deserialise an XML file to object

```
const string FILE_PATH = "C:\\Users\\Matthew\\Test\\";

XmlDocument xmlDocument = new XmlDocument();
xmlDocument.Load(FILE_PATH + "Person.xml");
string xmlString = xmlDocument.OuterXml;

StringReader read = new StringReader(xmlString);
XmlSerializer serialiser = new XmlSerializer(typeof(Person));
XmlReader reader = new XmlTextReader(read);
Person p = (Person)serialiser.Deserialize(reader);
reader.Close();
read.Close();
p.DisplayDetails();
```

GUI Tutorial

For information about C# Windows Forms please refer to the textbook (Deitel & Deitel) chapters 14 – 15 and to the demonstrations and examples that are given during the lectures.

Demo 1

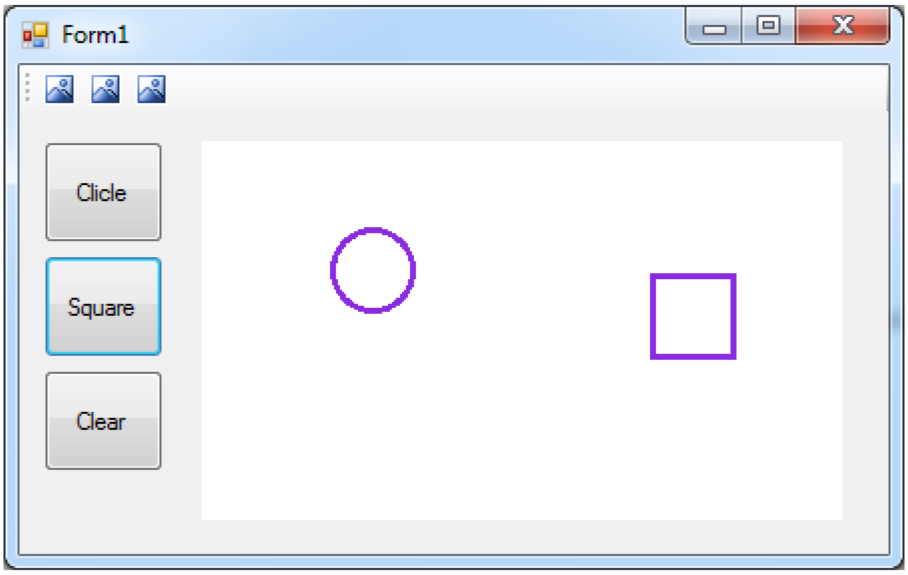
Implement a simple GUI based calculator. This calculator should allow the user to perform the most basic math operations – addition, subtraction, multiplication, and division. It should also provide the traditional memory functions **mc**, **m+**, **m-** and **mr**.

Demo 2

Implement a simple mobile phone directory. This telephone directory should be GUI based. The user should be able to add, edit, delete and search for records. Each record should include the mobile phone number and the name and surname of the owner of that mobile phone number. The search should allow the user to search for a particular mobile phone number using any data field. The program should save all the records either using a text file or an XML file.

Demo 3

Implement a simple drawing application. The program should have a shapes palette on the left-hand side of the window from where the user can select the shapes s/he wants to draw. In the middle of the application there should be a drawing panel where the user can click to draw the selected shape. Include a simple toolbar with options to start a new drawing panel, save a drawing and open a drawing. A sketch of this program is illustrated below:



Delegates and Events

Delegates

A *delegate* is a C# language element that allows a programmer to reference a method. Those who are familiar with C or C++, a *delegate* is basically a pointer to a function. However, this feature is totally new to other developers.

> Think about how you use methods right now. You write an algorithm that does its thing by manipulating the values of variables and calling methods directly by name. What if you wanted an algorithm that was very flexible, reusable, and allowed you to implement different functionality as the need arises? Furthermore, let's say that this was an algorithm that supported some type of data structure that you wanted to have sorted, but you also want to enable this data structure to hold different types. If you don't know what the types are, how could you decide an appropriate comparison routine? Perhaps you could implement an *if/then/else* or *switch* statement to handle well-known types, but this would still be limiting and require overhead to determine the type. You could solve this problem by passing a *delegate* to your algorithm and letting the contained method, which the *delegate* refers to, perform the comparison operation.
>
> http://www.csharp-station.com/Tutorial/CSharp/Lesson14

Consider the following example [Example adapted from http://www.csharp-station.com/Tutorial/CSharp/Lesson14]:

```
// Delegate declaration
public delegate int Comparer(object obj1, object obj2);

public class Name
{
     private string firstName = null;
     private string lastName = null;

     public string FirstName
     {
          get { return firstName; }
          set { firstName = value; }
     }

     public string LastName
     {
          get { return lastName; }
          set { lastName = value; }
     }

     public Name(string firstName, string lastName)
     {
          this.firstName = firstName;
          this.lastName = lastName;
     }
     // Delegate method handler
     public static int CompareFirstNames(object name1, object name2)
     {
          string n1 = ((Name)name1).FirstName;
          string n2 = ((Name)name2).FirstName;

          if (String.Compare(n1, n2) > 0)
          {
               return 1;
          }
          else if (String.Compare(n1, n2) < 0)
          {
               return -1;
          }
          else
          {
               return 0;
          }
     }

     public override string ToString()
     {
          return FirstName + " " + LastName;
     }
}

--------------

public class SimpleDelegate
{
     private Name[] names = new Name[5];

     public SimpleDelegate()
     {
          names[0] = new Name("Joe", "Borg");
          names[1] = new Name("Maria", "Camilleri");
          names[2] = new Name("Dylan", "Xuereb");
          names[3] = new Name("Karen", "Zammit");
          names[4] = new Name("Arnold", "Camilleri");
     }

     public void PrintNames()
     {
          foreach(Name name in names)
          {
               Console.WriteLine(name.ToString());
          }
     }
```

```
      public void Sort(Comparer compare)
      {
            object temp;
            for (int i = 0; i < names.Length; i++)
            {
                  for (int j = i; j < names.Length; j++)
                  {
                        if (compare(names[i], names[j]) > 0)
                        {
                              temp = names[i];
                              names[i] = names[j];
                              names[j] = (Name)temp;
                        }
                  }
            }
      }

      public static void Main(string[] args)
      {
            SimpleDelegate sd = new SimpleDelegate();

            // Delegate instantiation
            Comparer cmp = new Comparer(Name.CompareFirstNames);
            Console.WriteLine("\nBEFORE SORT:\n");
            sd.PrintNames();

            sd.Sort(cmp);
            Console.WriteLine("\nAFTER SORT:\n");
            sd.PrintNames();
      }
}
```

The first thing that the above program does is declare a *delegate*. *Delegate* declarations look somewhat like methods, except they have the *delegate* modifier, are terminated with a semi-colon (*;*), and have no implementation.

This *delegate* declaration defines the signature of a delegate handler method that this *delegate* can refer to. The delegate handler method, for the *Comparer delegate*, can have any name, but must have a first parameter of type *object*, a second parameter of type *object*, and return an *int* type. The following method from the above program shows a delegate handler method that conforms to the signature of the *Comparer delegate*:

```
public static int CompareFirstNames(object name1, object name2)
{
      …
}
```

The *CompareFirstNames* method calls *String.Compare* to compare the *FirstName* properties of the two *Name* instances. The *String* class has many convenience methods, such as *Compare*, for working with strings. Please don't allow the implementation of this method to interfere with learning how delegates work. What you should concentrate on is that *CompareFirstNames* is a handler method that a delegate can refer to, regardless of the code inside of that method.

To use a *delegate*, you must create an instance of it. The instance is created, similar to a class instance, with a single parameter identifying the appropriate delegate handler method, as shown below:

```
     Comparer cmp = new Comparer(Name.CompareFirstNames);
```

The *delegate*, *cmp*, is then used as a parameter to the *Sort()* method, which uses it just like a normal method. Observe the way the *delegate* is passed to the *Sort()* method as a parameter in the code below:

```
     sd.Sort(cmp);
```

Using this technique, any delegate handler method may be passed to the *Sort()* method at run-time. In other words you can define a method handler named *CompareLastNames()*, instantiate a new *Comparer delegate* instance with it, and pass the new *delegate* to the *Sort()* method as shown in the code snippets below:

```
// Delegate method handler
public static int CompareLastNames(object name1, object name2)
{
     string n1 = ((Name)name1).LastName;
     string n2 = ((Name)name2).LastName;

     if (String.Compare(n1, n2) > 0)
     {
          return 1;
     }
     else if (String.Compare(n1, n2) < 0)
     {
          return -1;
     }
     else
     {
          return 0;
     }
}

----------

Comparer cmp = new Comparer(Name.CompareLastNames);
```

Events

A C# *event* is a class member that is activated whenever the event it was designed for occurs. Anyone interested in the *event* can register and be notified as soon as the *event* fires. At the time an *event* fires, registered methods will be invoked.

Events and *delegates* work together to provide a program's functionality. It starts with a class that declares an *event*. Any class, including the same class that the *event* is declared in, may register one of its methods for the *event*. This occurs through a *delegate*, which specifies the signature of the method that is registered for the *event*. The *delegate* may be one of the pre-defined .NET *delegates* or one you declare yourself. Whichever is appropriate, you assign the *delegate* to the *event*, which effectively registers the method that will be called when the *event* fires.

Consider the following example [Example adapted from http://www.csharp-station.com/Tutorial/CSharp/Lesson14]:

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace EventsDemo
{
    public delegate void Startdelagate();

    public class EventsDemo : Form
    {
        // A custom event
        public event Startdelagate StartEvent;

        public EventsDemo()
        {
            Text = "Events Demo Window";
            Button myButton = new Button();
            myButton.Parent = this;
            myButton.Text = "Click Here";
            myButton.Location = new Point(
                (ClientSize.Width - myButton.Width) / 2,
                (ClientSize.Height - myButton.Height) / 2);

            // An EventHandler delagate is assigned to the
            // button's Click event
            myButton.Click += new EventHandler(OnMyButtonClicked);

            // Our custom "StartDelegate" delegate is assigned to
            // our custom "StartEvent" event
            StartEvent += new Startdelagate(OnStartEvent);

            // Fire our custom event
            StartEvent();

        }
        public void OnMyButtonClicked(object sender, EventArgs ae)
        {
            MessageBox.Show("You clicked my button!");
        }

        public void OnStartEvent()
        {
            MessageBox.Show("I Just Started!");
        }

        public static void Main(string[] args)
        {
            Application.Run(new EventsDemo());
        }
    }
}
```

As C# is a visual language, traditionally we design our graphical user interfaces using the drag and drop facility provided by *Visual Studio*. Therefore, code that involves events and delegates as shown above is generated automatically by the IDE.

Language Integrated Query (LINQ)

Language Integrated Query (LINQ) is a new feature in the C#.Net language that allows the programmer to query collections, xml files, database and other data structures using one standard query language. LINQ allows you to write query expressions, like SQL queries that retrieve information from a wide variety of data sources and not just databases.

LINQ to Objects

The term *LINQ to Objects* refers to the use of LINQ queries with any *Enumerable* or *Enumerable*<T> collection directly, without the use of an intermediate LINQ provider or API such as LINQ to SQL or LINQ to XML. You can use LINQ to query any enumerable collections such as List<T>, Array, or Dictionary<TKey, TValue>. The collection may be user-defined or may be returned by a .NET Framework API.

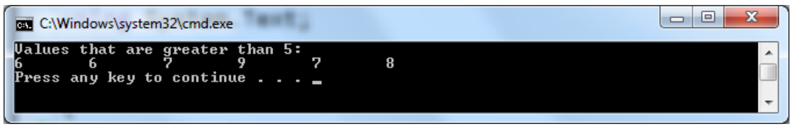
LINQ to Objects represents a new approach to collections. In the old way, you had to write complex *foreach* loops that specified how to retrieve data from a collection. In the LINQ approach, you write declarative code that describes what you want to retrieve.

In addition, LINQ queries offer three main advantages over traditional foreach loops:

- They are more concise and readable, especially when filtering multiple conditions.
- They provide powerful filtering, ordering, and grouping capabilities with a minimum of application code.
- They can be ported to other data sources with little or no modification.

Querying an array of integer values

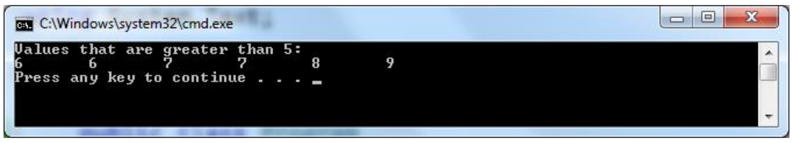
```
public static void Main(string[] args)
{
      int[] values = { 6, 3, 1, 5, 6, 7, 9, 3, 1, 5, 7, 8 };

      // A LINQ query that obtains values that are greater than 5 from the array
      var filtered = from value in values
                         where value > 5 select value;

      // Display the filtered results
      Console.WriteLine("Values that are greater than 5:");
      foreach (var item in filtered)
      {
            Console.Write("{0}\t", item);
      }
      Console.WriteLine
}
```



```
Values that are greater than 5:
6       6       7       9       7       8
Press any key to continue . . .
```

Querying an array of values and getting them sorted

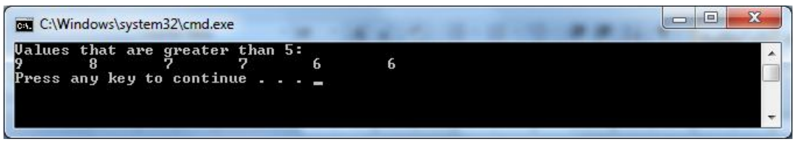
```
public static void Main(string[] args)
{
      int[] values = { 6, 3, 1, 5, 6, 7, 9, 3, 1, 5, 7, 8 };

      // A LINQ query that obtains values that are greater than 5 from the array
      // sorted
      var filtered = from value in values
                     where value > 5
                     orderby value
                     select value;

      // Display the filtered results
      Console.WriteLine("Values that are greater than 5:");
      foreach (var item in filtered)
      {
            Console.Write("{0}\t", item);
      }
      Console.WriteLine();
}
```



```
C:\Windows\system32\cmd.exe
Values that are greater than 5:
6       6       7       7       8       9
Press any key to continue . . .
```

Querying an array of values and getting them sorted in descending order

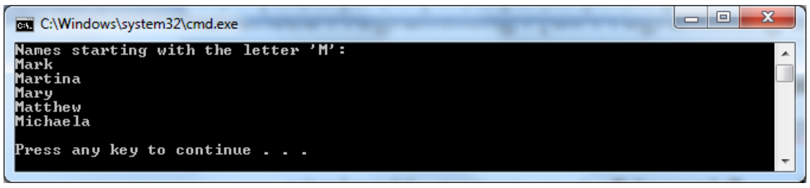
```
public static void Main(string[] args)
{
      int[] values = { 6, 3, 1, 5, 6, 7, 9, 3, 1, 5, 7, 8 };

      // A LINQ query that obtains values that are greater than 5 from the array
      // sorted in descending order
      var filtered = from value in values
                     where value > 5
                     orderby value descending
                     select value;

      // Display the filtered results
      Console.WriteLine("Values that are greater than 5:");
      foreach (var item in filtered)
      {
            Console.Write("{0}\t", item);
      }
      Console.WriteLine();
}
```



```
C:\Windows\system32\cmd.exe
Values that are greater than 5:
9       8       7       7       6       6
Press any key to continue . . .
```

Querying an array of names to get the names that starts with the letter “M”

```
public static void Main(string[] args)
{
      string[] names = { "Joseph" , "Matthew" , "Emanuel" , "Michaela" ,
                         "Mary" , "Karen" , "David" , "Paul" , "Mark" ,
                         "Martina" , "Daniela"};

      var startingWithM = from name in names
                                      where name[0] == 'M'
                                      orderby name
                                      select name;

      // Display the filtered results
      Console.WriteLine("Names starting with the letter 'M':");
      foreach (var name in startingWithM)
      {
            Console.WriteLine("{0}", name);
      }
      Console.WriteLine();
}
```



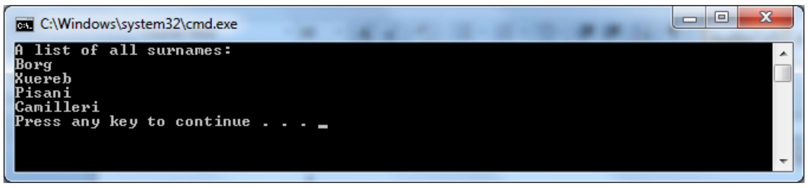
```
Names starting with the letter 'M':
Mark
Martina
Mary
Matthew
Michaela

Press any key to continue . . .
```

Querying an array of objects

```
public static void Main(string[] args)
{
      List<Person> myPersons = new List<Person>();
      myPersons.Add(new Person("Joseph","Borg",55));
      myPersons.Add(new Person("Matthew","Xuereb",26));
      myPersons.Add(new Person("Karen", "Pisani", 38));
      myPersons.Add(new Person("Damien", "Pisani", 9));
      myPersons.Add(new Person("Michaela", "Camilleri", 24));
      myPersons.Add(new Person("Jeremy", "Pisani", 12));

      // Get all persons with surname "Pisani" order by age
      var surnamePisani = from p in myPersons
                                      where p.Surname == "Pisani"
                                      orderby p.Age
                                      select p;

      Console.WriteLine("Surname \"Pisani\":");
      foreach (var p in surnamePisani)
      {
            Console.WriteLine(p.ToString());
      }
}
```

Querying an array of objects and getting distinct results

```
public static void Main(string[] args)
{
      List<Person> myPersons = new List<Person>();
      myPersons.Add(new Person("Joseph","Borg",55));
      myPersons.Add(new Person("Matthew","Xuereb",26));
      myPersons.Add(new Person("Karen", "Pisani", 38));
      myPersons.Add(new Person("Damien", "Pisani", 9));
      myPersons.Add(new Person("Michaela", "Camilleri", 24));
      myPersons.Add(new Person("Jeremy", "Pisani", 12));

      // Get all surnames
      var surnames = from p in myPersons select p.Surname;

      Console.WriteLine("A list of all surnames:");
      foreach (var s in surnames.Distinct())
      {
            Console.WriteLine(s);
      }
}
```


```
C:\Windows\system32\cmd.exe
A list of all surnames:
Borg
Xuereb
Pisani
Camilleri
Press any key to continue . . . _
```

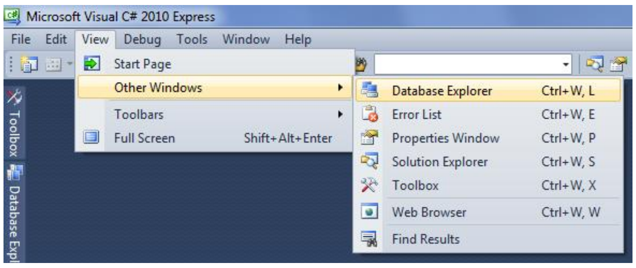
LINQ to SQL

LINQ to SQL provides a runtime infrastructure for managing relational data as objects without losing the ability to query. With this system you are free to manipulate the objects while LINQ to SQL stays in the background tracking your changes automatically.

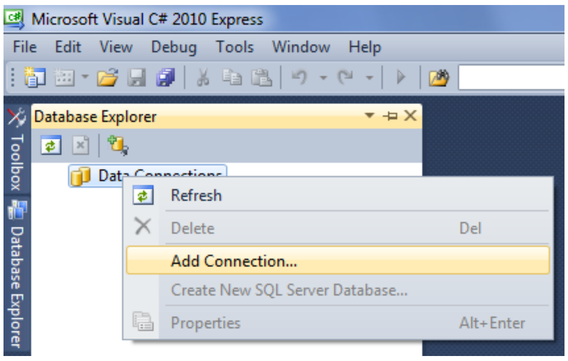
Before you can work with this system, you must create the LINQ to SQL classes. In the following example a database named *TestDatabase.mdf* is used. This is implemented on a MS SQL Server. A table called *Flights* that resides in this database is going to be used. This table contains three fields: *FlightNum*, *Origin*, *Destination*.

The following steps should be followed [When using Visual C# Express Studio 2010] to create the LINQ to SQL classes:

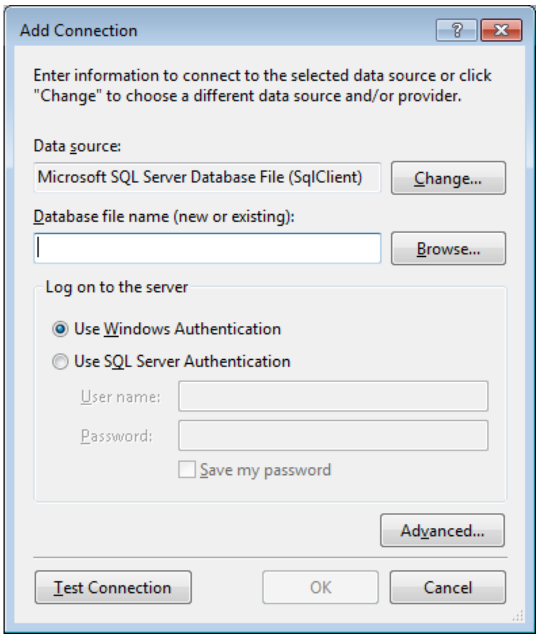
1. Select *Database Explorer* from the *Data* drop down menu.


Microsoft Visual C# 2010 Express
File Edit View Debug Tools Window Help
Start Page
Other Windows
Toolbars
Full Screen Shift+Alt+Enter
Database Explorer Ctrl+W, L
Error List Ctrl+W, E
Properties Window Ctrl+W, P
Solution Explorer Ctrl+W, S
Toolbox Ctrl+W, X
Web Browser Ctrl+W, W
Find Results
Toolbox
Database Expl

2. Right click on the *Database Connections* option and select *Add Connection*.



3. The *Add Connection* dialog box is displayed on the screen as shown below:



Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

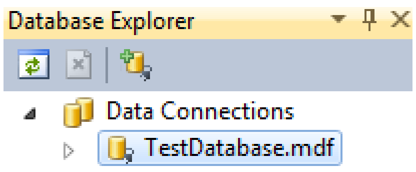
Data source:
Microsoft SQL Server Database File (SqlClient)

Database file name (new or existing):

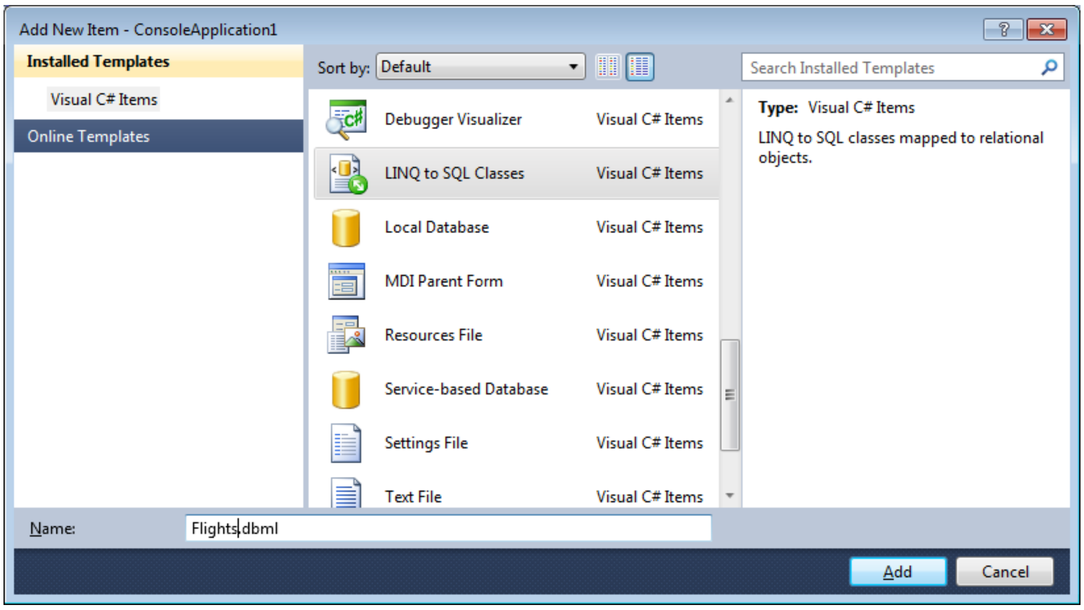
Log on to the server
Use Windows Authentication
Use SQL Server Authentication
User name:
Password:
Save my password

4. Select the correct Data Source (in this case it is a Ms SQL Server) and the database file. Click on Ok to continue.

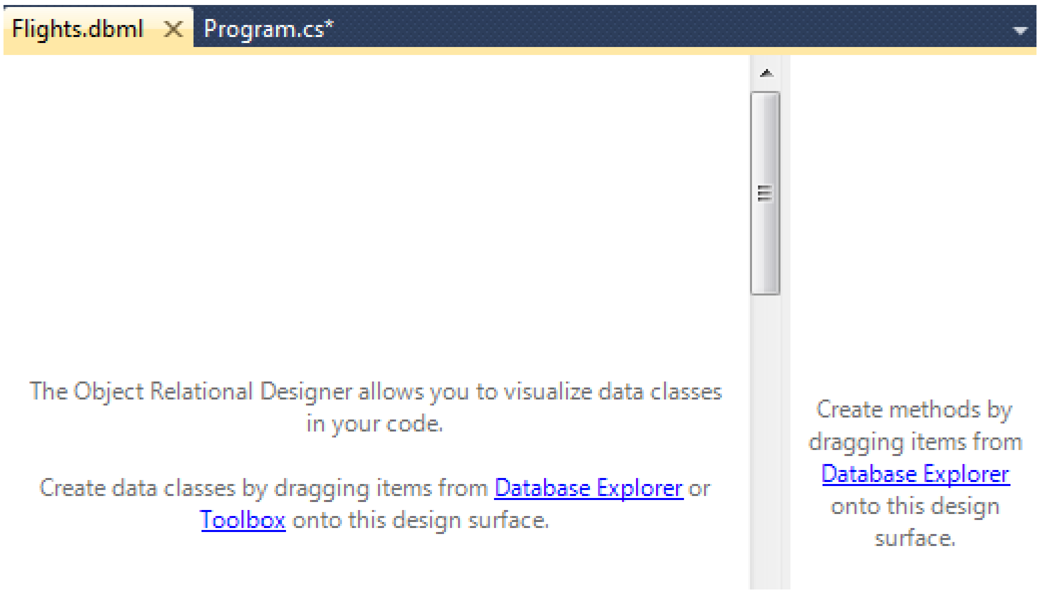
5. If the connection is successful it should be displayed under the Data Connections heading as shown in the screen shot below:



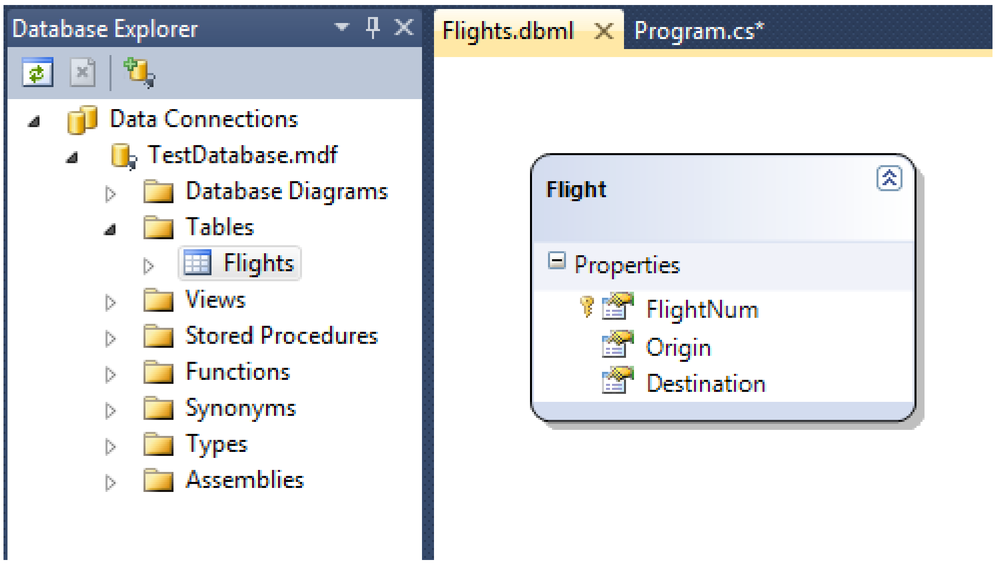
6. Create a new (or open an existing) project. Right click on the project icon on the Solution Explorer and select the *Add Item* option. Then select the option *LINQ to SQL Classes*. It is suggested to name this file with the same name of the SQL table ans shown in the screen shot below:


Add New Item - ConsoleApplication1
Installed Templates
Visual C# Items
Online Templates
Sort by: Default
Search Installed Templates
Debugger Visualizer Visual C# Items
LINQ to SQL Classes Visual C# Items
Local Database Visual C# Items
MDI Parent Form Visual C# Items
Resources File Visual C# Items
Service-based Database Visual C# Items
Settings File Visual C# Items
Text File Visual C# Items
Type: Visual C# Items
LINQ to SQL classes mapped to relational objects.
Name: Flights.dbml
Add
Cancel

7. The *Object Relational Designer* will be displayed on the screen as shown below:


Flights.dbml
Program.cs*
The Object Relational Designer allows you to visualize data classes in your code.
Create data classes by dragging items from Database Explorer or Toolbox onto this design surface.
Create methods by dragging items from Database Explorer onto this design surface.

8. Drag the tables that you want to use in your program from the *Database Explorer* to the *Object Relation Designer* as shown below:



9. The database table can now be accessed using an object oriented approach as explained below.

Querying a database table using LINQ

```
public static void Main(string[] args)
{
      FlightsDataContext database = new FlightsDataContext();

      var flights = from flight in database.Flights
                    select flight;
      Console.WriteLine("FlightNum\tOrigin\t\t\tDestination");
      foreach (var f in flights)
      {

      Console.WriteLine("{0}\t{1}\t{2}",f.FlightNum,f.Origin,f.Destination);
      }
}
```

Updating a database table using LINQ

```
public static void Main(string[] args)
{
      FlightsDataContext database = new FlightsDataContext();

      Flight f = new Flight
      {
            FlightNum = "KM301",
            Destination = "Rome",
            Origin = "Malta"
      };

      database.Flights.InsertOnSubmit(f);
      database.SubmitChanges();
      }
}
```

Multithreading

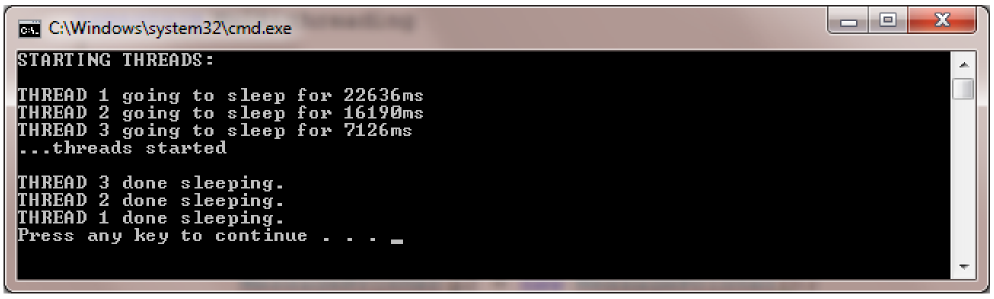
Multithreading is a behavior of a software application where multiple instances of codes are executed simultaneously within the application. Every application runs with at least one thread. A thread is nothing more than a process. The process performs sets of sequential steps, each step executing a line of code. Since the steps are sequential, each step takes a given amount of time. The time it takes to complete a series of steps is the sum of the time it takes to perform each programming step.

In C#, the System.Threading namespace provides classes and interfaces that support multithreaded programming and enable you to easily perform tasks such as creating and starting new threads, synchronizing multiple threads, suspending threads, and aborting threads.

A simple example

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

namespace Multithreading
{
    public class MessagePrinter
    {
        private int sleepTime;
        private static Random random = new Random();

        public MessagePrinter()
        {
            // Generate a random delay between 0 and 30 seconds
            sleepTime = random.Next(30000);
        }

        public void Print()
        {
            Thread current = Thread.CurrentThread;
            Console.WriteLine(current.Name + " going to sleep for " +
                              sleepTime + "ms");
            Thread.Sleep(sleepTime);
            Console.WriteLine(current.Name + " done sleeping.");
        }

    }
}

----------------
```

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

namespace Multithreading
{
    public class Program
    {
        public static void Main(string[] args)
        {
            MessagePrinter p1 = new MessagePrinter();
            Thread t1 = new Thread(new ThreadStart(p1.Print));
            t1.Name = "THREAD 1";

            MessagePrinter p2 = new MessagePrinter();
            Thread t2 = new Thread(new ThreadStart(p2.Print));
            t2.Name = "THREAD 2";

            MessagePrinter p3 = new MessagePrinter();
            Thread t3 = new Thread(new ThreadStart(p3.Print));
            t3.Name = "THREAD 3";

            Console.WriteLine("STARTING THREADS:\n");
            t1.Start();
            t2.Start();
            t3.Start();
            Console.WriteLine("...threads started\n");
        }
    }
}
```



```
STARTING THREADS:

THREAD 1 going to sleep for 22636ms
THREAD 2 going to sleep for 16190ms
THREAD 3 going to sleep for 7126ms
...threads started

THREAD 3 done sleeping.
THREAD 2 done sleeping.
THREAD 1 done sleeping.
Press any key to continue . . .
```

Reflection

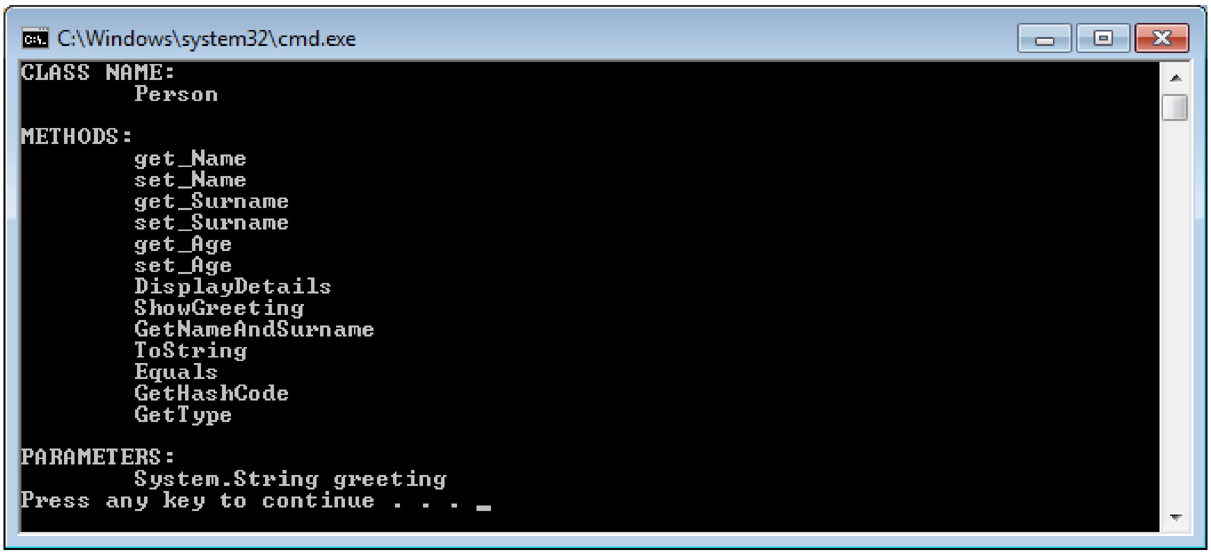
Reflection is the process by which a computer program can observe and modify its own structure and behavior. Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications. The use of reflection is an important technique for improving productivity. Reflection facilitates development of programs that are easily adapted to requirement changes.

By using Reflection in C#, one can find out details of an object, method, and create objects and invoke methods at runtime. The *System.Reflection* namespace contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types. When writing a C# code that uses reflection, the coder can use the *typeof* keyword to get the object's type or use the *GetType()* method to get the type of the current instance.

The following are some code snippets that are making use of reflection:

Example 1 – Getting information from a class

```
using System;
using System.Reflection;

namespace ReflectionDemo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Creating an object of type Person for demonstration purposes
            Person p = new Person();

            // Get the type information
            Type myObjectType = p.GetType();
            Console.WriteLine("CLASS NAME:\n\t" + myObjectType.Name);

            // List all methods in class
            MethodInfo[] myMethods = myObjectType.GetMethods();
            Console.WriteLine("\nMETHODS:");
            foreach (MethodInfo m in myMethods)
            {
                Console.WriteLine("\t" + m.Name);
            }

            // Show parameters of method "ShowGreeting"
            MethodInfo myMethodInfo =
                                    myObjectType.GetMethod("ShowGreeting");
            object[] parameters = myMethodInfo.GetParameters();
            Console.WriteLine("\nPARAMETERS:");
            foreach (object param in parameters)
            {
                Console.WriteLine("\t" + param);
            }
        }
    }
}
```

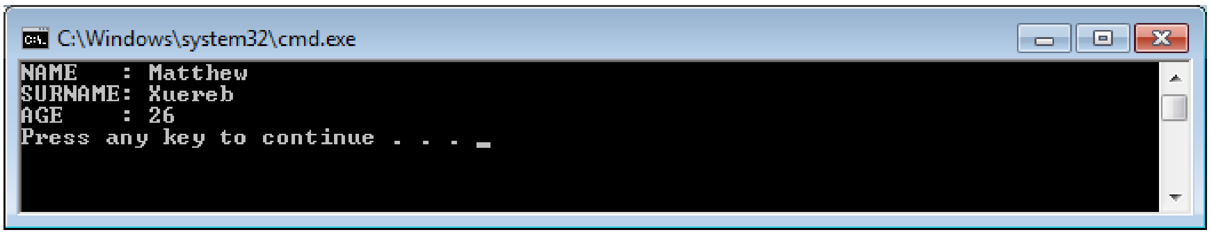


C:\Windows\system32\cmd.exe

```
CLASS NAME:
        Person

METHODS:
        get_Name
        set_Name
        get_Surname
        set_Surname
        get_Age
        set_Age
        DisplayDetails
        ShowGreeting
        GetNameAndSurname
        ToString
        Equals
        GetHashCode
        GetType

PARAMETERS:
        System.String greeting
Press any key to continue . . . _
```

Example 2 – Creating an object and invoking methods

```
using System;
using System.Reflection;

namespace ReflectionDemo
{
    public class Program
    {
        public static void Main(string[] args)
        {
            // Get the type reference. Person is used for demonstration purposes
            Type myObjectType = Type.GetType("ReflectionDemo.Person");

            // Create constructor arguments
            Type[] argumentsType ={
                                    typeof(string),
                                    typeof(string),
                                    typeof(int)
                                  };

            // Create constructor argument values
            object[] argumentsValues ={
                                    "Matthew",
                                    "Xuereb",
                                    26
                                  };

            // Create the constructor
            ConstructorInfo myConstructor =
                                myObjectType.GetConstructor(argumentsType);

            // Instantiate the object
            object myObject = myConstructor.Invoke(argumentsValues);

            // Invoke the "DisplayDetails" method
            myObject.GetType().GetMethod("DisplayDetails").Invoke(myObject,null);
        }
    }
}
```



```
NAME    : Matthew
SURNAME: Xuereb
AGE     : 26
Press any key to continue . . .
```