Programming in CSharp

by
Willi-Hans Steeb
International School for Scientific Computing

and
E.J. Dembskey

Version: 2009-01-06

email addresses of the authors:

steeb_wh@yahoo.com
steebwilli@gmail.com
whsteeb@uj.ac.za
evan.dembskey@gmail.com
demskeye@tut.ac.za

Contents

Preface

The book gives a collection of C# programs.
Without doubt, this book can be extended. If you have comments or suggestions, we would be pleased to have them. The email addresses of the author are:

whsteeb@uj.ac.za
steeb_wh@yahoo.com

The web sites of the author is:

http://issc.uj.ac.za

Chapter 1

Introduction

CSharp is designed for the .NET framework. The .NET framework is object oriented. CSharp has a great set of tools for the object oriented programmer. CSharp was standardised as ECMA-334 and ECMA-335 in August 2000 by Microsoft, Hewlett-Packard and Intel and as ISO/IEC 23270 by ISO/IEC. CSharp is the first component oriented language in the C/C++ family. Component concepts are first class:

Properties, methods, events
Design-time and run-time attributes
integrated documentation using XML

CSharp can be embedded in web pages (ASP.NET). In C++ and Java primitive date types (`int`, `double`, etc) are magic and do not interoperate with objects. In Smalltalk and Lisp primitive types are objects, but at great performance cost. CSharp unifies this with no performance cost. CSharp also adds new primitive data types, for example `decimal`. Collections work for all types.

In CSharp, `private` is the default accessibility. The accessibility options are:

`public` - accessible to all
`private` - accessible to containing class
`protected` - accessible to containing or derived classes
`internal` - accessible to code in same assembly
`protected internal` - means `protected` or `internal`

In the default accessibility, scope is restricted to the containing code block.

Classes can be marked as `public` or `internal`. By default classes are `private`.

Type members in CSharp are:

Fields: The state of an object or type
Methods: Constructors, Functions, Properties (smart fields)
Members come in two basic forms
Instance - per object data and methods (default)
Static - per type data and methods (use the `static` keyword).

Constructors are used to initialize fields. We can implement simpler constructors in terms of more complex ones with the `this` keyword. We can indicate which base constructor to call by using the `base` keyword. Type constructors are used to initialize `static` fields for a type. We use the `static` keyword to indicate a type constructor.

All types in the system are derived from class `object`. The class `object` contains the functions `string ToString()` and `bool Equals()` which should be overriden when we write our own class.

We use the `virtual` keyword to make a method virtual. In a derived class, an override method is marked with the `override` keyword.

CSharp has built in support for events. This is useful for dealing with objects in an event driven operating system. More than one type can register interest in a single event. A single type can register interest in any number of events.

CSharp supports interfaces using the `interface` keyword. Our types can implement interfaces. We must implement all methods. Interfaces can contain methods but no fields with properties and events included.

CSharp also provides type conversion, for example if `char c = 'a';` we can convert to an integer `int i = (int) c;` via the ASCII table.

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. Comments using such syntax are called documentation comments. The XML generation tool is called the documentation generator. This generator could be the C# compiler itself.

The names given to variables, methods etc. by the programmer are referred to as *identifiers*. An identifier has to: begin with a letter, or begin with an underscore. It cannot be the same as built-in keywords. Identifiers are case-sensitive.

The following is a list of keywords. Keywords are reserved identifiers that hold a special meaning to the CSharp compiler. We can use the same name as a keyword identifier as long as we prefix the name with an `@` symbol.

```
abstract    event       new         struct
as          explicit    null        switch
base        extern      object      this
bool        false       operator    throw
break       finally     out         true
byte        fixed       override    try
case        float       params      typeof
catch       for         private     uint
char        foreach     protected   ulong
checked     goto        public      unchecked
class       if          readonly    unsafe
const       implicit    ref         ushort
continue    in          return      using
decimal     int         sbyte       virtual
default     interface   sealed      volatile
delegate    internal    short       void
do          is          sizeof      while
double      lock        stackalloc
else        long        static
enum        namespace   string
```

The following is a list of contextual keywords. These provide a specific meaning in the code, but are not reserved words. Some have meanings in two or more contexts, for example, partial and where.

```
from                    partial (method)
get                     select
group                   set
into                    value
join                    var
let                     where (generic type constraint)
orderby                 where (query clause)
partial (type)          yield
```

Do avoid using class names duplicated in heavily used namespaces. For example, do not use the following for a class name:

```
System      Collections   Forms    UI
```

There are a number of preprocessor directives available. While the compiler lacks a separate preprocessor, directives are processed as if there was one. They aid in conditional compilation. Unlike with C and C++, these directives cannot be used to create macros.

```
#if                     #else
#elif                   #endif
```

Languages: C#, VB.NET, VJ#, Jscript, Cobol, Pascal
Common Language Specification (CLS)
Common Type System (CTS)
ASP.NET | Windows Forms XML Web Services Web Forms
Data and XML Classes
Base Framework Classes
Common Language Runtime (CLR)
Operating System

Table 1.1: .NET Framework Stack

```
#define                 #undef
#warning                #error
#line
#region
#endregion
#pragma  #pragma warning
#pragma checksum
```

C# cannot be run without installing the Common Language Runtime, which is Microsoft's implementation of the Common Language Infrastructure. In order to compile .NET programs a .NET compiler must be installed. Together these form part of the .NET Framework. The .NET Framework is installed automatically when Windows Server 2003, Windows XP SP2, Vista or later is installed. It is also installed when Visual Studio 2003, 2005 or 2008 is installed. The .NET Framework is also available as a separate download from Microsoft. It is best to install the .NET Framework SDK. An express version of Visual Studio 2008 is available for free download from http://www.microsoft.com/Express/.

The current version of the .Net Framework is version 3.5, but many machines still run 1.1, 2.0 and 3.0.

The .NET Framework consists of several components, the most important of which are the runtime and Class Library. See Table 1.1 for an overview of the stack. Version 3.5 adds several capabilities to the stack: the Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF), Windows CardSpace and Language Integrated Query (LINQ). SilverLight is a web-based subset of WPF.

The runtime, or Common Language Runtime (CLR) provides a virtual machine for the execution of .NET code. In addition, CLR provides services that include memory management, thread management, component lifetime management, garbage

collection and default error handling. A benefit of the CLR is that it provides these execution services to all .NET applications without any additional effort on the part of the programmer.

When a .NET program is compiled, the compiler generates an .exe file. However, this .exe file does not contain directly executable instructions. Instead it contains Common Intermediate Language (CIL). When the .exe is executed it does not interact with the operating system directly but with the CLR. This is similar to Java, with CIL being analogous to bytecode and CLR with the Java Virtual Machine(JVM).

When a .NET application is first executed, the CLR performs a compile of the CIL and creates a directly executable file from it. This is known as **Just In Time** compilation. This theoretically overcomes the performance overhead associated with using virtual machines.

The Class Library is a collection of extensible class libraries organised hierarchically into namespaces. The Class Library covers a wide range, including user interface, data access and database connectivity (ADO.NET), cryptography, web application development (ASP.NET), numeric algorithms, XML and network communications. The Class Library is language-independent.

For example, the **System** namespace is at the top of the namespace hierarchy. It contains all the types that represent the base data types such as text, numbers and dates. The **Systems.Windows.Forms** namespace allows the programmer to take advantage of the Windows forms engine to create graphical user interface objects easily.

The Common Language Specification (CLS) is a specification for creating or porting programming languages to that they are .NET compatible. The CLS uses uses a Common Type System (CTS). This is a component of the CLR and provides a common set of data types that are consistent between all .NET languages.

Chapter 2

CSharp Basics

2.1 Introduction

Compile and link the programs with

```
csc filename.cs
```

This generates an execute file named `filename.exe`. This is also known as an executable assembly. The most common compiler options are:

- `/checked` Explicitly request checking for entire component
- `/r:` Reference libraries
- `/out:` Output file
- `/target:` Component type (Library, module, .exe)
- `/define:` Define preprocessor symbols
- `/doc:` Defines documentation output
- `/version:` Define version
- `/reference:` Record dependency information
- `/debug+` Generate debug code (creates .exe and .pdb)
- `/main` Define class which contains entry main

In `C#` everything must be inside a class. Thus also the `Main` method must be contained inside a class. In the first example this is `Hello1`. The method name, `Main`, is reserved for the starting point of a program. `Main` is often called the entry point (starting address) of the program. In front of the word `Main` is a static modifier. The static modifier explains that this method works in this specific class only, rather

than an instance of the class. This is necessary, because when a program begins, no object instances exists.

To avoid fully qualifying classes throughout the program, we can use the `using` directive. We write to the screen, where `\n` provides a `newline`. Note that `C#` is case-sensitive. A comment is indicated by `//`.

```
// cshello.cs

using System;

class Hello1
{
  public static void Main()
  {
  Console.WriteLine("Hello Egoli\n");    // \n newline
  Console.WriteLine("Good Night Egoli");
  Console.Write("midnight");
  } // end Main
}
```

The namespace declaration, `using System;` indicates that we are referencing the `System` namespace. Namespaces contain groups of code that can be called upon by `C#` programs. With the `using System;` declaration, we are telling our program that it can reference the code in the `System` namespace without pre-pending the word `System` to every reference. The `System.Console` class contains a method

```
WriteLine()
```

and a method

```
Write()
```

that can be used to display a string to the console. The difference is that the `Console.Write(.)` statement writes to the Console and stops on the same line, but the `Console.WriteLine(.)` goes to the next line after writing to the Console. Strings are embedded in double quotes, for example `"name"`.

An exception is an error condition or unexpected behavior encountered by an executing program during runtime. We write to the screen using exception handling with a `try catch finally` block. The try block is used around statements that might throw exceptions. The catch block defines exception handlers. Code in a finally block is always executed. Use it to release resources, for example to close any streams or files that were opened in the try block. Some common exceptions are listed here:

- **System.ArithmeticException:** A base class for exceptions that occur during arithmetic operations, such as System.DivideByZeroException
- **System.ArgumentException:** Thrown when an argument to a method is invalid
- **System.ArrayTypeMismatchException:** Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
- **System.DivideByZeroException:** Thrown when an attempt to divide an integral value by zero occurs.
- **System.IndexOutOfRangeException:** Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
- **System.InvalidCastException:** Thrown when an explicit conversion from a base type or interface to derived types fails at run time.
- **System.MulticastNotSupportedException:** Thrown when an attempt to combine two non-null delegates fails, because the delegate type does not have a void return type.
- **System.NullReferenceException:** Thrown when a null reference is used in a way that causes the referenced object to be required.
- **System.OutOfMemoryException:** Thrown when an attempt to allocate memory (via new) fails.
- **System.OverflowException:** Thrown when an arithmetic operation in a checked context overflows.

To access command line parameters we change the signature of the `Main(void)` to `Main(string[] args)`. The expression `string[] args` defines an array of strings.

```
// hello2.cs

using System;

class Hello2
{
  public static void Main(string[] args)
  {
  try { Console.WriteLine("Hello {0}",args[0]); }
  catch(Exception e) { Console.WriteLine(e); }
     Console.WriteLine("Good Night");
  } // end Main
}
```

After compiling the program we run it with, for example,

```
Hello2 James
```

To read from the keyboard we use the method `ReadLine()`. The command `Length` returns the number of elements in the array. We are also using an `if-else` construct.

```
// hello3.cs

using System;

class Hello3
{
  public static void Main(string[] args)
  {
  if(args.Length > 0)
  {
  Console.WriteLine("Hello {0}",args[0]);
  }
  else
  {
  Console.WriteLine("Enter your name: ");
  string name = Console.ReadLine();
  Console.WriteLine("Hello {0}",name);
  }
  Console.WriteLine("Good Night");
  } // end Main
}
```

In the case of compiling using Visual Studio or some other IDE the problem of the program's console window disappearing too quickly to view its output might occur. This problem can be solved using the `Read()` method to pause the window. The program above can be modified like so:

```
  ...
  Console.WriteLine("Good Night");
  Console.Read(); // Pauses the program until a key is pressed
  } // end Main
}
```

2.2 Basic Data Types

Basic (primitive) data types are

```
bool,
byte, sbyte, char,
short, ushort, int, uint, long, ulong
float, double, decimal
```

The range of `byte` is `0..255` and for `sbyte` is `-128..127`. `short` and `ushort` are 2 bytes (16 bits), where `ushort` is unsigned. `int` and `uint` are 4 bytes. The floating point number `float` is 4 bytes and the floating point number `double` is 8 bytes. `char` data type (2 bytes) in C# contains Unicode characters similar to Java. CSharp is a strongly typed language and therefore variables must be declared with an available type and must be initialized with a value (or reference) of the same type.

Built-in types are also `string` and `object`.

C#	.NET Type	Description
bool	System.Boolean	true/false
byte	System.Byte	unsigned byte value
sbyte	System.SByte	signed byte value
char	System.Char	a single character
short	System.Int16	16 bit signed integer
ushort	System.UInt16	16 bit unsigned integer
int	System.Int32	32 bit signed integer
uint	System.UInt32	32 bit unsigned integer
long	System.Int64	64 bit signed integer
ulong	System.UInt64	64 bit unsigned integer
float	System.Single	32 bit floating point
double	System.Double	64 bit floating point
decimal	System.Decimal	a high precision double
string	System.String	string of characters
object	System.Object	a generic type

At compile time the C# compiler converts the C# types into their corresponding .NET types described in the above table. Apart from the above basic types the user may define his own types using `enum`, `struct` and `class`.

```
// datatypes.cs

using System;

class Datatypes
{
  public static void Main()
  {
  bool b = true;          // boolean data type
  Console.WriteLine(!b);  // ! is the logical NOT
```

```
  // character 2 bytes Unicode
  char e = 'a';           // character 'a' ASCII value 97
  Console.WriteLine(e);
  char f = '\0';          // null character ASCII value 0
  Console.WriteLine(f);

  // default integer type is int
  byte c = 255;           // unsigned byte 0...255
  c++;
  Console.WriteLine(c);
  sbyte d = -125;         // signed byte -128...127
  d--;
  short g = -10000;       // short -32768...32767 (2 bytes)
  Console.WriteLine(g);
  ushort h = 20000;       // unsigned short 0...65535 (2 bytes)
  Console.WriteLine(h);
  int i = -100000;        // signed int -2147483648...2147483647
  Console.WriteLine(i);
  uint j = 200000;        // unsigned int 0...4294967295
  Console.WriteLine(j);
  long k = -234567899;    // signed long 64 bits -2^63...2^63-1
  Console.WriteLine(k);
  ulong l = 3456789123;   // unsigned long 64 bits 0...2^64-1
  Console.WriteLine(l);

  // default floating point number is double
  // (float) type conversion from double to float
  float m = (float) 3.145; // float 32 bits
  Console.WriteLine(m);
  double n = 3.14159;      // double 64 bits
  Console.WriteLine(n);
  // decimal 128 bits
  decimal p = (decimal) 2.89124357865678; // type conversion
  Console.WriteLine(p);
  }
}
```

Note that `(data type)` is the type conversion operator.

2.3 ASCII Table

ASCII (American Standard Code for Information Interchange) is a character set and a character encoding based on the Roman alphabet as used in modern English. ASCII codes represent text in computers, in other communication equipment, and in control devices that work with text. ASCII specifies a correspondence between digit

bit patterns and the symbol of a written language. ASCII is, strictly, a seven-bit code, meaning that it uses the bit patterns representable with seven binary digits (a range of 0 to 127) to represent character information. ASCII reserves the first 32 codes (numbers 0-31 decimal) for control characters. The capital `'A'` is 65 (decimal) and the small `'a'` is 97. Furthermore `'0'` is 48 and `'9'` is 57. Space is 32.

We are doing type conversion from `char -> int` using the ASCII table. The *type conversion* operator is `(data-type)`. The *null character* is `'\0'`.

```
// Escape.cs

using System;

class Escape
{
  public static void Main()
  {
  char c1 = '\b';     // backspace
  int i1 = (int) c1;  // ASCII value
  Console.WriteLine("i1 = " + i1); // 8

  char c2 = '\f';     // form feed
  int i2 = (int) c2;  // ASCII value
  Console.WriteLine("i2 = " + i2); // 12

  char c3 = '\n';     // newline
  int i3 = (int) c3;  // ASCII value
  Console.WriteLine("i3 = " + i3); // 10

  char c4 = '\r';     // carriage return
  int i4 = (int) c4;  // ASCII value
  Console.WriteLine("i4 = " + i4); // 13

  char c5 = '\t';     // horizontal tab
  int i5 = (int) c5;  // ASCII value
  Console.WriteLine("i5 = " + i5); // 9

  char c6 = ' ';     // blank
  int i6 = (int) c6; // ASCII value
  Console.WriteLine("i6 = " + i6); // 32
  } // end Main
}
```

2.4 Arithmetic Operations

The arithmetic operations are

`++, --, +, -, *, /, %`

where `++` is the increment by 1 and `--` is the decrement by -1. The operation `%` provides the remainder in integer division.

Note that we have *integer division* (for example $17/4 = 4$) and *floating point division* depending on the data types.

```
// arithmetic.cs

using System;

class Arithmetic
{
  public static void Main()
  {
  byte c = 255;
  c++;
  Console.WriteLine(c);
  sbyte d = -125;  // signed byte
  d--;
  Console.WriteLine(d);

  int i = -100000;     // signed int
  int j = 15002;
  int k = i + j;
  Console.WriteLine(k);

  long m = -234567899; // signed long
  long n = 345;
  long p = m*n;
  Console.WriteLine(p);

  int r1 = 27;
  int r2 = 5;
  int r3 = r1/r2;  // integer division
  Console.WriteLine("r3 = " + r3);
  int r4 = r1%r2;  // remainder
  Console.WriteLine("r4 = " + r4);

  float f1 = (float) 3.145; // type conversion
  float f2 = (float) 2.81;
```

```
  float f3 = f1*f2;
  Console.WriteLine(f3);

  double d1 = 3.14159;
  double d2 = 4.5;
  double d3 = d1/d2;
  Console.WriteLine(d3);

  decimal p1 = (decimal) 2.89124357865678; // type conversion
  decimal p2 = (decimal) 3.14159;          // type conversion
  decimal p3 = p1 + p2;
  Console.WriteLine(p3);
  }
}
```

2.5 Control Statements

Control statements control the program flow. For example, selection statements such as `if ... else` and `switch` use certain criteria to select a course of certain action within the program.

```
// myIf.cs

using System;

class myIf
{
  public static void Main()
  {
  int i;
  Console.WriteLine("Enter integer: ");
  string line = Console.ReadLine();
  i = System.Convert.ToInt32(line); // convert string of digits to int
  if(i > 5) // if true do the next command else skip it
  Console.WriteLine("The number is larger than 5");
  else
  Console.WriteLine("The number is smaller than 5 or equal to 5");
  } // end Main
}
```

The `for` loop applied to a one-dimensional array.

```
// forloop.cs

using System;
```

```
class forloop
{
  public static void Main()
  {
  double[] numbers = { 1.1, 2.2, 3.3, 4.4 };
  int i = 0;
  double sum = 0.0;

  for(i=0;i<numbers.Length;i++) // Length provides length of array
  {
  sum += numbers[i]; // sum = sum + numbers[i];
  }
  Console.WriteLine("sum = {0}",sum);

  int[] x = new int[3]; // declare array and allocate memory
  x[0] = 4; x[1] = 3; x[2] = 7;
  int intsum = 0;

  for(i=0;i<x.Length;i++)
  {
  intsum += x[i]; // shortcut for intsum = intsum + x[i];
  }
  Console.WriteLine(intsum);
  } // end Main
}
```

Another example for the `for` loop applied to numbers read from the keyboard. Note that `!` is the logical NOT and `ToInt32(string)` converts a string of digits to an integer.

```
// forloop1.cs

using System;

class ForLoop
{
  public static void Main()
  {
  int sum = 0; // initialize sum to 0
  String line;
  for(line=Console.In.ReadLine();line!="";line=Console.In.ReadLine())
  {
  sum += System.Convert.ToInt32(line);
  } // end for loop
```

```
  Console.WriteLine(sum.ToString() + "\n");
  } // end Main
}
```

The `foreach` loop can be applied to an array of strings. We count how often the string `"otto"` is in the array of strings.

```
// foreachloop.cs

using System;

class foreachloop
{
  public static void Main()
  {
  string[] namelist = { "willi","otto","carl","john","otto" };
  int count = 0;
  string n = "otto";

  foreach(string name in namelist) // keyword in
  {
  if(name==n) // compare strings for equality case sensitive
  {
  count++;
  }
  } // end foreach
  Console.WriteLine("count = {0}",count);
  } // end Main
}
```

The `while` loop

```
// whileloop.cs

using System;

class whileloop
{
  public static void Main()
  {
  int[] numbers = { 1, 2, 3, 4 };
  int i = 0;
  int sum = 0;

  while(i < numbers.Length)
  {
```

```
  sum += numbers[i];
  i++;
  }
  Console.WriteLine("sum = {0}",sum);
  } // end Main
}
```

The `do-while` loop applied to an array of floating point numbers.

```
// dowhileloop.cs

using System;

class whileloop
{
  public static void Main()
  {
  double[] numbers = { 1.1, 2.3, 3.5, 4.5 };
  int i = 0;
  double sum = 0.0;

  do
  {
  sum += numbers[i];
  i++;
  }
  while(i < numbers.Length);
  Console.WriteLine("sum = {0}",sum);
  } // end Main
}
```

If one has a large decision tree and all the decisions depend on the value of the same variable we use a `switch` statement instead of a series of `if ... else` constructions. The `switch` statement transfers control to one of several case-labeled statements, depending on the value of the `switch` expression. Note that if the `break` is omitted, execution will continue over the remaining statements in the switch block. The `break` statement can also be used to break out of an iteration loop.

```
// switch.cs

using System;

class Myswitch
{
  public static void Main()
  {
```

```
  string  st = "bella";
  for(int i=0;i<st.Length;i++)
  {
  char c = st[i];
  switch(c)
  {
  case 'a': Console.WriteLine("character is a 'a' ");
            break;
  case 'b': Console.WriteLine("character is a 'b' ");
            break;
  default: Console.WriteLine("character is not an 'a' or a 'b' ");
           break;
  }
  } // end for loop

  int[] numbers = { 3,4,6,1,4,-3,1,6};

  for(int j=0;j<numbers.Length;j++)
  {
  switch(numbers[j])
  {
  case 4: Console.WriteLine("number at position {0} is 4",j);
          break;
  case 6: Console.WriteLine("number at position {0} is 6",j);
          break;
  default: Console.WriteLine("number at position {0} is not 4 or 6",j);
           break;
  } // end switch
  } // end for loop
  } // end Main
}
```

2.6 Logical Operations

The logical operators in CSharp, C, C++ and Java are

```
&& logical AND
|| logical OR
!  logical NOT

// logical.cs

using System;

class MyLogica
```

```
{
  public static void Main()
  {
  int j;
  Console.WriteLine(" Enter an integer: ");
  string line = Console.ReadLine();
  j = System.Convert.ToInt32(line);

  if((j%2 == 0) && (j < 10))
  Console.WriteLine("The integer is even and smaller than 10");
  else
  Console.WriteLine("The integer is either odd or larger than 10 or both");

  Console.WriteLine();
  int k;
  Console.WriteLine("Enter an integer: ");

  line = Console.ReadLine();
  k = System.Convert.ToInt32(line);

  if((k > 0) || (k < 0))
  Console.WriteLine("The integer is nonzero");
  else
  Console.WriteLine("The integer is zero");

  Console.WriteLine();
  int n;
  Console.WriteLine("Enter an integer: ");

  line = Console.ReadLine();
  n = System.Convert.ToInt32(line);

  if(n == 0) Console.WriteLine("The integer is zero");
  else
  Console.WriteLine("The integer is nonzero");
  }
}
```

2.7 Pointers

A *pointer* is a data type whose value refers directly to ("points to") another value stored elsewhere in the computer memory using its address. Thus the pointer has an address and contains (as value) an address. Obtaining the value that a pointer referes to is called *dereferencing*. The dereference operator is `*`. Pointers in CSharp

must be declared `unsafe`.

```
// Pointers1.cs

using System;

class Pointers
{
  public static unsafe void Main()
  {
  int i = 15;
  int* p = &i; // declare pointer and assignment to address of i
  int j = 15;
  int* q = &j;
  bool b1 = (i==j);
  Console.WriteLine("b1 = " + b1); // true
  bool b2 = (p==q);
  Console.WriteLine("b2 = " + b2); // false

  // dereferencing pointers
  int r = *q;
  Console.WriteLine("r = " + r); // 15
  }
}
```

We are using pointer to pass by reference (see section 2.10).

2.8 Recursion

Recursion plays a central role in computer science. A recursive function is one whose definition includes a call to itself. A recursion needs a stopping condition. We use recursion to find the Fibonacci numbers.

```
// recursion.cs

using System;

class recur
{
  public static ulong fib(ulong n)
  {
  if(n==0) return 0;
  if(n==1) return 1;
  return fib(n-1) + fib(n-2);
  } // end fib
```

```
  public static void Main()
  {
  ulong n = 10;
  ulong result = fib(n);
  Console.WriteLine("Result = {0}",result);
  } // end Main
}
```

2.9 Jump Statements

C# also has an goto for jumping to labels. We use the goto to jump to the labels L1, L2, L3, L4 if a condition is met.

```
// mygoto.cs

using System;

class Mygoto
{
  public static void Main()
  {
  Random r = new Random(51);
  L3:
  int a = r.Next(100);
  int b = r.Next(100);
  int result;
  L1:
  Console.WriteLine("{0} + {1} =",a,b);
  string s = Console.ReadLine();
  result = Convert.ToInt32(s);
  if(result == (a+b))
  goto L2;
  Console.WriteLine("sorry you are not correct: try again");
  goto L1;
  L2:
  Console.WriteLine("congratulations you are correct");

  Console.WriteLine("Want to add again: Press y for yes and n for no: ");
  string t = Console.ReadLine();
  if(t == "y") goto L3;
  if(t == "n")
  {
  Console.WriteLine("bye, see you next time around");
  goto L4;
```

```
  }
  L4:
  int e = 0;
  }
}
```

The method `Exit()` is used for program termination. It is part of the class `Environment`.

```
// password.cs

using System;

class password
{
  public static void Main()
  {
  string password = "XYA";
  int i,j,k;
  for(i=65;i<91;i++)
  {
  for(j=65;j<91;j++)
  {
  for(k=65;k<91;k++)
  {
  char c1 = (char) i;
  char c2 = (char) j;
  char c3 = (char) k;
  char[] data = { c1,c2,c3 };
  string s = new string(data); // converting array of char to string
  bool found = password.Equals(s);
  if(found == true)
  {
  Console.WriteLine("Password = {0}",s);
  Environment.Exit(0);
  }
  }
  }
  }
  }
}
```

2.10 Pass by Value, Pass by Reference

Arguments to functions can be passed either by value or by reference. When an argument is passed by value, a copy of the argument is produced, and the associated

parameter is the same as a local variable for the function. This means, changes to the parameter variable will have no effect on the original argument value. Functions that receive variables as parameters get local copies of those variables, not the originals. The alternative, pass by reference, is indicated by the presence of the keyword `ref` in the argument list. When arguments are passed by reference, the parameter variable is an alias for the argument value. Thus, change in the parameter also alter the original argument. In `C#` we can also use pointers and the dereference operator to pass by reference. Thus functions that receive pointers to variables gain access to the original variables associated with the pointers.

In the next program we pass by value.

```
// Pitfall.cs

using System;

public class Pitfall
{
  public static void add(int i)
  {
  int n = i + i;
  i = n;
  Console.WriteLine("i inside function add = " + i); // 20
  } // end add

  public static void Main()
  {
  int i = 10;
  add(i);
  Console.WriteLine("i in Main = " + i); // 10
  }
}
```

Note that without `static` in method `add(int)`: error message: An object reference is required for the nonstatic method `Pitfall.add(int)`.

In the next program we use pointers to pass by reference. We rotate integer numbers.

```
// pointers2.cs

using System;

public class Rotate
{
  public static unsafe void rot(int* p,int* q,int* r)
```

```
  {
  int t = *r;
  *r = *p; *p = *q; *q = t;
  }

  public static unsafe void Main(string[] args)
  {
  int a = 10; int b =  12; int c = 17;
  rot(&a,&b,&c);
  Console.WriteLine("a = {0} and b = {1} and c = {2}",a,b,c);
  } // end main
}
```

In the following program we use the keyword `ref` to pass by reference. We rotate three integers.

```
// references.cs

using System;

public class Rotate
{
  public static void rot(ref int p,ref int q,ref int r)
  {
  int t = r;
  r = p; p = q; q = t;
  }

  public static void Main()
  {
  int a = 10; int b = 12; int c = 17;
  rot(ref a,ref b,ref c);
  Console.WriteLine("a = {0} and b = {1} and c = {2}",a,b,c);
  } // end Main
}
```

In the following program we pass the first argument by reference and the second by value.

```
// passing.cs

using System;

public class Passing
{
  static void change(ref string sa,string sb)
```

```
  {
  sa = "yyy";
  sb = "222";
  }

  public static void Main()
  {
  string s1 = "xxx";
  string s2 = "111";
  change(ref s1,s2);
  Console.Write("s1 = {0} and s2 = {1}",s1,s2); // => s1 = yyy s2 = 111
  }
}
```

The `out` keyword explicitely speciefies that a variable should be passed *by reference* to a method, and set in that method. A variable using this keyword must not be initialized before the method call.

```
// Divide.cs

using System;

class Divider
{
  public static int Divide1(int dividend,int divisor,out int r)
  {
  int quot = dividend/divisor;
  r = dividend - quot*divisor;
  return quot;
  } // end Divide1

  public static void Divide2(int dividend,int divisor,out int quot,out int r)
  {
  quot = dividend/divisor;
  r = dividend - quot*divisor;
  } // end Divide2

  public static void Main()
  {
  int r;
  int q = Divide1(123,14,out r);
  Console.WriteLine("Quotient = {0} and Remainder = {1}",q,r);

  int s;
  int t;
```

```
  Divide2(145,3,out s,out t);
  Console.WriteLine("Quotient = {0} and Remainder = {1}",s,t);
  } // end Main
}
```

2.11 Arrays

An array is a data structure that contains a number of variables. These are accessed through computed indices. C# supports one-dimensional arrays, multidimensional arrays (rectangular arrays) and arrays of arrays (*jagged arrays*). As C, C++ and Java C# arrays are zero indexed. This means the array indexes start as zero. When declaring arrays, the square bracket `[]` must come after the type, not the identifiers, for example `int[] table`. The size of the array is not part of its type as it is in the C language. Thus

```
int[] numbers = new int[20];
```

We can initialise arrays very simply:

```
int[] numbers = {0, 1, 2, 3, 5};
```

This is identical to the complete initialisation statement:

```
int[] numbers = new int[] {0, 1, 2, 3, 5};
```

In C# arrays are actually objects. `System.Array` is the abstract base type of all array types. The class `Array` contains methods for sorting and searching.

```
// myArray.cs

using System;

class myArray
{
  public static void Main()
  {
  int[] numbers = { 4, 12345, 890, 23456789 };
  int prod = numbers[2]*numbers[0];
  Console.Write("prod = " + prod);
  Console.Write("\n");
  int numb = Array.BinarySearch(numbers,4);
  Console.Write("numb = " + numb);
  Console.Write("\n");
  double[] d = new double[3];
  d[0] = 1.1; d[1] = 3.4; d[2] = 8.9;
  int dpos = Array.BinarySearch(d,8.9);
  Console.Write("dpos = " + dpos);
```

```
  Console.Write("\n");

  string[] slist = { "otto", "uli", "carl", "marius", "jacob" };

  int pos1 = Array.BinarySearch(slist,"carl");
  Console.Write("pos1 = {0}",pos1);
  Console.WriteLine();

  Array.Sort(slist); // sorting the array

  int pos2 = Array.BinarySearch(slist,"carl");

  Console.Write("pos2 = {0}",pos2);
  Console.WriteLine();

  for(int j=0;j<slist.Length;j++)
  {
  Console.WriteLine("{0} {1}",j,slist[j]);
  }
  }
}
```

To create multidimensional arrays the array initializer must have as many levels of nesting as there are dimensions in the array. Thus:

```
int[,] numbers = {{0, 2}, {4, 6}, {8, 10}, {12, 17}, {16, 18}};
```

The outermost nesting level corresponds to the leftmost dimension. The innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. Thus the example above creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension. For example

```
int[,] numbers = new int[5,2];
```

and initialises the array with:

```
numbers[0,0] = 0; numbers[0,1] = 2;
numbers[1,0] = 4; numbers[1,1] = 6;
numbers[2,0] = 8; numbers[2,1] = 10;
numbers[3,0] = 12; numbers[3,1] = 14;
numbers[4,0] = 16; numbers[4,1] = 18;
```

We can also create jagged arrays, which are arrays of arrays. The element arrays do not all have to be the same.

```
// twodim.cs

using System;

public class TwoDim
{
  public static void Main()
  {
  int[][] matrix = new int[2][];  // rows
  matrix[0] = new int[2];         // columns
  matrix[1] = new int[2];         // columns
  matrix[0][0] = 2; matrix[0][0] = 4;
  matrix[1][0] = 7; matrix[1][1] = 3;
  int i = 1;
  Console.WriteLine("matrix[" + i + "][" + i + "] = " + matrix[i][i]);

  double[,] myarray;
  myarray = new double[2,3];
  myarray[0,0] = 3.1;
  myarray[0,1] = 4.7;
  myarray[0,2] = 3.3;
  myarray[1,0] = 2.7;
  myarray[1,1] = 1.1;
  myarray[1,2] = 7.3;
  double r = myarray[0,1]*myarray[1,2];
  Console.WriteLine("r = " + r);
  } // end Main
}
```

2.12 Bitwise Operations

Consider the integer number 17 (base 10, i.e. $17 = 1 \cdot 10^1 + 7 \cdot 10^0$. It can be written in binary as

$$17 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \,.$$

Thus the *binary representation* of 17 would be 10001. If 17 is considered as data type `int` (32 bits) we have the binary representation

00000000000000000000000000010001

The *bitwise operation* in CSharp, C, C++ and Java are:

```
&  AND
|  OR  (inclusive OR)
^  XOR (exclusive OR)
~  NOT
```

```
// bitwise.cs

using System;

class MylBitwise
{
  public static void Main()
  {
  int r3 = 4; int r4 = 5;
  int r5 = r3 & r4; // bitwise AND
  Console.WriteLine("Binary AND of 4 and 5 gives {0}",r5);

  int r6 = 7; int r7 = 11;
  int r8 = r6 | r7; // bitwise OR
  Console.WriteLine("Binary OR 7 and 11 gives {0}",r8);

  int r9 = r6 ^ r7; //bitwise XOR
  Console.WriteLine("Binary XOR of 7 and 11 gives {0}",r9);

  int x = 125;
  int r10 = x^x;
  Console.WriteLine("Binary XOR of 125 with itself gives {0}",r10);

  int r11 = ~r9;
  Console.WriteLine("Binary NOT of r9 gives {0}",r11);

  int r12 = 4;
  int r13 = ~r12; // one's complement
  int r14 = ++r13;
  Console.WriteLine("two's complement of 4 {0}",r14);
  }
}
```

2.13 Shift Operation

We can use shift operation for fast integer multiplication by 2, 4, 8, ... and fast integer division by 2, 4, 8, ... The shift operations are << and >>.

```
// MyShift.cs

using System;

class Shift
{
  public static void Main()
```

```
  {
  int i = 17;  // 17 in binary 10001
  int j = i >> 1;   // integer divison by 2
  Console.WriteLine("j = " + j); // => 8
  int k = i >> 2;   // integer divison by 4
  Console.WriteLine("k = " + k); // => 4

  int m = 9;
  int n = m << 1; // multiplication by 2
  Console.WriteLine("n = " + n);  // => 18
  int p = m << 2; // multiplication by 4
  Console.WriteLine("p = " + p);  // => 36
  }
}
```

2.14 Commmand-Line Arguments

C# enables us to access command-line arguments by supplying and using the following parameters in function `Main`

```
Main(string[] args)
```

Basically a C# program consists of a class with a static member method (class function) called `Main`. When the C# compiler (`csc.exe`) compiles a C# program it marks the `Main` method as the entrypoint in the generated IL code. The `Main` method may accept an array of string as its arguments (though this is optional). This array will always contain the command line arguments passed to the program by its user. We start counting from zero. The following program shows an application.

```
// CommandLine.cs

using System;

class CommandLine
{
  public static void Main(string[] args)
  {
  Console.WriteLine("Hello{0}",args[0]);
  Comsole.WriteLine("Goodbye.");
  }
}
```

We run the program, for example

```
CommandLine World
```

2.15 Boxing and UnBoxing Types

Boxing refers to converting a value type to an object type, and unboxing refers to the opposite. Boxing is carried out implicitly in C#, whereas we have to use type casting to unbox to an appropiate data type. For example

```
int i;
Console.WriteLine("i={0}",i);
```

The `WriteLine()` method requires an object, so in the above statement integer `i` is implicitly boxed to an object and passed to the `WriteLine` method. An example for unboxing is

```
int i;
object obj = i; // boxing is implicit
int j;
j = (int) obj;  // to unbox we use type cast
```

Typically unboxing is done in a `try` block. If the object being unboxed is `null` or if the unboxing cannot succeed because the object is of a different type, an `InvalidCastException` is thrown.

2.16 Delegates

A delegate essentially creates a name for a the specific type/signature of a method. Delegates are type safe function pointers. One must first declare a delegate.

```
// delegates.cs

using System;

// declare delegate with the signature of the
// encapsulated method
delegate void MyDelegate(string m,int a,int b);

class Application
{
  public static void Main()
  {
  MyDelegate md = new MyDelegate(FirstMethod);
  md += new MyDelegate(SecondMethod);
  md("message A",4,5);
  md("message B",7,11);
  } // end Main
```

```
  static void FirstMethod(string s1,int x1,int y1)
  {
  Console.WriteLine("1st method: " + s1);
  int sum1 = x1 + y1;
  Console.WriteLine("sum1 = " + sum1);
  }

  static void SecondMethod(string s2,int x2,int y2)
  {
  Console.WriteLine("2st method: " + s2);
  int sum2 = x2 + y2;
  Console.WriteLine("sum2 = " + sum2);
  }
}
```

The output is

```
1st method message A
sum1 = 9
2st method message A
sum2 = 9
1st method message B
sum1 = 18
2st method message B
sum2 = 18
```

2.17 Types

The `typeof` command is an operator. It resolves at compile time and operates over a type. To check whether an object is compatible to a specific type is to apply the `is` keyword.

```
// myTypeof.cs

using System;
using System.Text;

class myTypeof
{
  public static void Main()
  {
  double b = 3.14;
  string s = "xxx";
  StringBuilder sb = new StringBuilder("123456789");
```

```
  Type at = typeof(double);
  Console.WriteLine("at = {0}",at);
  Type st = typeof(string);
  Console.WriteLine("st = {0}",st);
  Type sbt = typeof(StringBuilder);
  Console.WriteLine("sbt = {0}",sbt);

  if(s is string) Console.WriteLine(at);
  if(s is StringBuilder) Console.Write("s is of the StringBuilder");
  else Console.Write("s is not of StringBuilder");

  if(b is int) Console.WriteLine("b is int");
  }
}
```

2.18 Reflection

Exposing and utilizing types at runtime is called reflection. The type of an object is stored as an instance of `System.Type` class the reference to which can be obtained using one of the following methods.

1. From the declaration type: If declaration `AType var` is legal then `System.Type` representing `AType` can be obtained using the `typeof` operator as:

```
Type t = typeof(AType);
```

2. From an instance: Type of an instance `obj` can be obtained using `GetType` method defined in `System.Object` as

```
Type t = obj.GetType();
```

3. From the type name within current assembly: `System.Type` offers a static method called `GetType` to obtain a `Type` from a fully qualified name of the type. The name will be searched in the current assembly

```
Type t = Type.GetType("FullyQualifiedTypeName");
```

4. From the type name within any assembly: First load the assembly and obtain a reference to it. This reference can be used to obtain the `Type` with a given name:

```
using System.Reflection;
Assembly asm = Assembly.LoadFrom("AssemblyName");
Type t = asm.GetType("FullyQualifiedTypeName");
```

The program `showtypes.cs` displays all the `Types` defined in an assembly whose name is passed in as first command line argument:

```
// showtypes.cs

using System;
using System.Reflection;

class ShowTypes
{
  public static void Main(string[] args)
  {
  Assembly asm = Assembly.LoadFrom(args[0]);
  Type[] types = asm.GetTypes();
  foreach(Type t in types) Console.WriteLine(t);
  }
}
```

We would run the program as, for example

```
showtypes datatypes.exe
```

Pass complete path to any .NET `exe` or `dll` to see the types declared in it.

The next program `showmembers.cs` takes the assembly name and type name as its command line arguments and displays all the members defined in that type of assembly.

```
// showmembers.cs

using System;
using System.Reflection;

class ShowMembers
{
  public static void Main(string[] args)
  {
  Assembly asm = Assembly.LoadFrom(args[0]);
  Type t = asm.GetType(args[1]);
  MemberInfo[] members = t.GetMembers();
  foreach(MemberInfo m in members) Console.WriteLine(m);
  }
}
```

2.19 Generics

C# Generics are similar to C++ Templates. They were introduced in version 2.0 of the C# language and the common language runtime (CLR). Generics introduce

to the .NET Framework the concept of type parameters. This makes it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated. The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace.

Some of the classes in

```
System.Collection.Generic
```

are:

```
Dictionary<TKey,TValue>         represents a collection of keys
                                and values
LinkedList<T>                   represents a doubly linked list
List<T>                         represents a strongly typed list
                                of objects
Queue<T>                        represents a firs-in, first out
                                collection of objects
SortedDictionary<TKey,TValue>   represents a collection of key/value
                                pairs that are sorted on the key
Stack<T>                        represents a varaible size last-in
                                first-out collection of instances
```

The first example shows an application of `Stack<T>`. It shows how the `Push` and `Pop` methods are used.

```
// MyStack0.cs

using System;
using System.Collections.Generic;

class MyStack0
{
   public static void Main()
   {
   Stack<int> stackint = new Stack<int>();
   stackint.Push(5);
   stackint.Push(8);
   int i = stackint.Pop();
   Console.WriteLine("i = " + i);   // 8
   stackint.Push(11);
   bool b0 = stackint.Contains(8);  // false
   Console.WriteLine("b0 = " + b0);
   bool b1 = stackint.Contains(11);
   Console.WriteLine("b1 = " + b1);
```

```
   int j = stackint.Peek();    // Peek does not Pop
   Console.WriteLine("j = " + j);  // 11
   Stack<string> stackstring = new Stack<string>();
   stackstring.Push("Abba");
   stackstring.Push("Baab");
   int r = stackstring.Count;
   Console.WriteLine("r = " + r); //  2
   } // end Main
}
```

The next program shows an application of `List<T>`. The method `Add` will add an element to the `List`.

```
// permutation.cs

using System;
using System.Collections;
using System.Collections.Generic;

public class Test
{
  private static void Swap(ref char a,ref char b)
  {
  if(a==b) return;
  a ^= b;  b ^= a; a ^= b; // using XOR operation for swapping
  }

  private static List<string> _permutations = new List<string>();

  // Recursive Function
  private static void Permute(char[] list,int k,int m)
  {
  if(k==m)
  {
  _permutations.Add(new string(list));
  }
  else
  {
  for(int i=k;i<=m;i++)
  {
  Swap(ref list[k],ref list[i]);
  Permute(list,k+1,m);
  Swap(ref list[k],ref list[i]);
  }
  }
```

```
  }

  [STAThread]
  static void Main(string[] args)
  {
    try
    {
    string str = "ola";
    char[] list = str.ToCharArray();
    Permute(list,0,list.Length-1);
    foreach(string perm in _permutations)
    {
    Console.WriteLine(perm);
    }
    }
    catch (Exception ex)
    {
    Console.WriteLine("Error: " + ex.Message);
    }
    } // end Main
}
```

The next program shows an application of `Dictionary<TKey,TValue>`.

```
// Dictionary1.cs

using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

public class Test
{
   private static Dictionary<char,char> _dict = new Dictionary<char,char>();

   private static string Substitution(string str)
   {
   StringBuilder sb = new StringBuilder();
   for(int i=0;i<str.Length;i++)
   {
   if(Char.IsWhiteSpace(str[i])) sb.Append(str[i]);
     else sb.Append(_dict[str[i]]);
   }
   return sb.ToString();
}
```

```
[STAThread]
static void Main(string[] args)
{
   try
   {
   // Build the dictionary
   // original alphabet A:
   // A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   string a = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   // substitution alphabet B:
   // D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
   string b = "DEFGHIJKLMNOPQRSTUVWXYZABC";
   for(int i=0;i<a.Length;i++)
   {
   _dict[a[i]] = b[i];
   }
   string res = Substitution("PLEASE CONFIRM RECEIPT");
   // SOHDVH FRQILUP UHFHLSW
   Console.WriteLine("Result=" + res);
   }
   catch (Exception ex)
   {
   Console.WriteLine("Error: " + ex.Message);
   }
   }
}
```

An application of the `SortedDictionary<TKey,TValue>` is given in the next program. The method `Add` will add a `TKey, TValue` element to the `SortedDictionary`. The method `Count` will find the number of `TKey, TValue` pairs in the `SortedDictionary`.

```
// MyDictionary.cs

using System;
using System.Collections.Generic;

class MyDictionary
{
  // declare key dictionary object
  public static SortedDictionary<int,string> items =
     new SortedDictionary<int,string>();

  static void Main(string[] args)
  {
```

```
  items.Add(1,"willi");
  items.Add(2,"ola");
  items.Add(7,"bulli");

  bool b1 = items.ContainsKey(7);
  Console.WriteLine("b1 = " + b1);

  bool b2 = items.ContainsValue("ola");
  Console.WriteLine("b2 = " + b2);

  int no = items.Count;
  Console.WriteLine("no = " + no);

  ICollection<int> k = items.Keys;

  foreach(int i in k)
    Console.WriteLine("{0},name: {1:k}",i,items[i]);
  }
}
```

We can also write our own generics. An example for the stack is given below. It shows how the `Pop` and `Push` methods are implemented and used

```
// MyStack1.cs

using System;
using System.Collections.Generic;

class MyStack<T>
{
   int MaxStack = 20;
   T[] StackArray;
   int StackPointer = 0;

   public MyStack() { StackArray = new T[MaxStack]; }

   public void Push(T x)
   {
   if(StackPointer < MaxStack) StackArray[StackPointer++] = x;
   }

   public T Pop()
   {
   return (StackPointer > 0) ? StackArray[--StackPointer] : StackArray[0];
   }
```

```
   public void Print()
   {
   for(int i=StackPointer-1;i>=0;i--)
     Console.WriteLine("Value: {0}",StackArray[i]);
   }
}

public class MainClass
{
   public static void Main()
   {
   MyStack<int> stackint = new MyStack<int>();
   stackint.Push(5);
   stackint.Push(8);
   int i = stackint.Pop();
   stackint.Print();    // 5

   MyStack<string> stackstring = new MyStack<string>();
   stackstring.Push("Abba");
   stackstring.Push("Baab");
   stackstring.Print(); // Baab Abba
   } // end Main
}
```

2.20 Indexers

An indexer is a member that enables an object to be indexed in the same way as an array (one or higher dimensional). Indexers have the same `this` and have a set of arguments in rectangular brackets. Two examples are given below for a one-dimensional array and a two-dimensional array.

```
// indexers1.cs

using System;

public class Number
{
   double[] Numbers;

   public double this[int i]
   {
   get { return Numbers[i];  }
   }
```

```
   public Number()  // default constructor
   {
   Numbers = new double[3];
   Numbers[0] = 1.23;
   Numbers[1] = 3.14;
   Numbers[2] = 2.83;
   }
}  // end class Number

public class MyMain
{
   public static void Main()
   {
   Number nbr = new Number();
   for(int i=0;i<3;i++)
      Console.WriteLine("Number {0}:{1}", 1+1,nbr[i]);
      Console.WriteLine();
   } // end Main
}  // end class MyMain

// indexers2.cs

using System;

public class Number
{
   double[,] Numbers;

   public double this[int i,int j]
   { get { return Numbers[i,j]; } }

   public Number()
   {
   Numbers=new double[3,3];
   Numbers[0,0]=1.23;
   Numbers[0,1]=3.14;
   Numbers[0,2]=2.83;
   Numbers[1,0]=4.56;
   Numbers[1,1]=7.54;
   Numbers[1,2]=9.23;
   Numbers[2,0]=8.34;
   Numbers[2,1]=7.43;
   Numbers[2,2]=5.24;
   }
}
```

```
public class MyMain
{
   static void Main()
   {
   Number nbr = new Number();
     for(int i=0;i<3;i++)
     {
     for(int j=0;j<3;j++)
     Console.WriteLine("Number {0},{1}: {2}",i,j,nbr[i,j]);
     }
     Console.WriteLine();
   } // end Main
}
```

Chapter 3

String and StringBuilder

3.1 String Class

The most importent built-in class is the `string` class. The `string` class and the `StringBuilder` class provide ways to perform string manipulations. The `string` class provides an immutable object

```
string s = "Hello";
```

which means that once the value of the string instance is set, it cannot be changed. Even though it appears that the application is changing the value of the string instance, it is actually returning a new instance of the `string` class in memory.

```
// mystring.cs

using System;

class mystring
{
  public static void Main()
  {
  string s1 = "otto";
  int length = s1.Length;
  Console.WriteLine(s1);     // => otto
  Console.WriteLine(length); // => 4

  string s2 = "willi";
  s2 = s2.Replace('l','t');
  Console.WriteLine(s2);    // witti

  string s3 = "Johannesburg";
  string result = s3.Substring(3,5);
  Console.WriteLine(result);  // => annes
```

```
  string s4 = "olli&ulli&ruedi";
  string[] textarray = s4.Split('&');
  Console.WriteLine(textarray[1]); // => ulli

  string s5 = string.Join(":",textarray);
  Console.WriteLine(s5); // => olli:ulli:ruedi

  char[] s6 = { 'o', 'p', 'a' };
  string s7 = new string(s6);
  Console.WriteLine(s7);   // => opa

  string s8 = "xeNa";
  string s9 = s8.ToUpper();
  Console.WriteLine(s9); // => XENA

  string s10 = "WILLI HANS";
  string s11 = s10.ToLower();
  Console.WriteLine("s11 = " + s11); // => willi hans

  // use + to concatenate strings
  string s12 = "Carl-";
  string s13 = "Otto";
  string s14 = s12 + s13;
  Console.WriteLine("s14 = " + s14);

  // use Equals() to compare strings
  // case sensitive
  bool b1 = s12.Equals(s13);
  Console.WriteLine("b1 = " + b1); // => False

  // can also use overloaded == to compare strings
  // case sensitive
  bool b2 = (s12 == s13);
  Console.WriteLine("b2 = " + b2); // => False

  // copy a string
  string s15 = string.Copy(s14);
  Console.WriteLine("s15 = " + s15);
  } // end Main
}
```

Arrays of strings can be implemented as follows.

```
// stringarrays.cs
```

```
using System;

class stringarrays
{
  public static void Main()
  {
  // one-dimensional array of strings
  string[] keywords = new string[] { "as", "do", "if", "in" };
  Console.WriteLine(keywords[3]); // => in

  // one-dimensional array of strings
  string[] names = { "willi", "ola", "xena" };
  Console.WriteLine(names[2]); // => xena

  // two-dimensional array of strings
  string[,] strArry = {{"1","one"}, {"2","two"}, {"3","three"}};
  Console.WriteLine(strArry[0,0]); // => 1
  Console.WriteLine(strArry[2,0]); // => 3
  } // end Main
}
```

3.2 Convert Class

Using the `Convert` class we can convert string to numbers (integers and floating point) and numbers (integers and floating point) to strings. The methods are

```
string ToString(T x) // any numerical type
bool ToBoolean(string s)
byte ToByte(string s)
char ToChar(string s)
short ToInt16(string s)
int ToInt32(string s)
long ToInt64(string s)
float ToSingle(string s)
double ToDouble(string s)
decimal ToDecimal(string s)
```

An example is

```
// MyConvert.cs

using System;

public class MyConvert
```

```
{
  public static void Main()
  {
  int i = 34;
  string s1 = Convert.ToString(i);
  Console.WriteLine("s1 = " + s1);
  double x = 3.14159;
  string s2 = Convert.ToString(x);
  Console.WriteLine("s2 = " + s2);

  bool b = Convert.ToBoolean("true");
  Console.WriteLine("b = " + b);
  char c = Convert.ToChar("x");
  Console.WriteLine("c = " + c);
  int j = Convert.ToInt32("12345");
  Console.WriteLine("j = " + j);
  string s3 = "3.14159";
  double y = Convert.ToDouble(s3);
  Console.WriteLine("y = " + y);
  }
}
```

3.3 StringBuilder Class

The `StringBuilder` class represents a mutable string a characters. It is called mutable because it can be modified once it has been created by using the methods `Append()`, `Insert()`, `Remove()` and `Replace()`. The `StringBuilder` class is defined in the `System.Text` namespace. Thus we have to add the following line in our application.

```
using System.Text;
```

Using the `StringBuilder` class.

```
// mystringbuilder.cs

using System;
using System.Text; // for StringBuilder

class mystringbuilder
{
  public static void Main()
  {
  string s = "carl";
  StringBuilder b1 = new StringBuilder(s.Length+12);
```

```
  b1.Append("carl");
  b1.Append("-uli");
  Console.WriteLine(b1); // => carl-uli
  b1.Remove(3,2);
  Console.WriteLine(b1); // => caruli

  StringBuilder b2 = new StringBuilder("A.C");
  b2.Insert(2,"B.");
  Console.WriteLine(b2); // => A.B.C

  b2.Replace('.',':');
  Console.WriteLine(b2); // =>  A:B:C

  StringBuilder b3 = new StringBuilder("stringbuilder");
  b3.Remove(4,9);
  Console.WriteLine(b3); // => stri
  }
}
```

Another application of the `StringBuilder` class is given by generating the *Thue-Morse sequence*. We apply recursion, i.e. the method `mythuemorse` calls itself.

```
// thuemorse.cs

using System;
using System.Text;

class ThueMorse
{
  public static void Main()
  {
  for(int i=0;i<7;i++)
  Console.WriteLine(mythuemorse(i));
  }

  public static StringBuilder mythuemorse(int n)
  {
  if(n==0) return new StringBuilder("0",50);
  StringBuilder tm = mythuemorse(n-1);
  StringBuilder tm2 = new StringBuilder("",30);
  for(int i=0;i<tm.Length;i++)
  if(tm[i] == '0') tm2.Append("01"); else tm2.Append("10");
  return tm2;
  }
}
```

Chapter 4

Built-in Classes

4.1 DateTime Class

To get the dates and time we use the `DateTime` class. The `DateTime` class stores both a full date and the full time. The two `static` members are `Now` and `Today`. `Now` contains the date and time for the moment the call is made. `Today` returns the current date. For formatting we could use `d` which would be the short date `mm/dd/yyyy`, for example `9/24/2005` and `D` would be `Saturday, September 24, 2005`.

```
// myDate.cs

using System;

class myDate
{
  public static void Main()
  {
  DateTime dt = DateTime.Now;
  Console.WriteLine("Date Time output: {0}",dt); // 7/29/2006 3:18:03 PM
  string s = dt.ToString();
  Console.WriteLine("s = " + s);

  DateTime today = DateTime.Today;
  Console.WriteLine("Today is: {0}",today);

  Console.WriteLine("Time output: {0}:{1}:{2}",dt.Hour,dt.Minute,dt.Second);
  Console.WriteLine("Date output: {0}\\{1}\\{2}",dt.Year,dt.Month,dt.Day);

  DateTime m1 = DateTime.Now;
  int milli1 = m1.Millisecond;
  Console.WriteLine("milli1 = " + milli1);
  for(uint j=0;j<10000000;j++)
  { j = j*2; j = j/2; j++; j--; }
```

```
  DateTime m2 = DateTime.Now;
  int milli2 = m2.Millisecond;
  Console.WriteLine("milli2 = " + milli2);
  int diff = milli2 - milli1;
  Console.WriteLine("diff = " + diff);

  // Formats
  DateTime currTime = DateTime.Now;
  Console.WriteLine("d: {0:d}",currTime); // 9/24/2005
  Console.WriteLine("D: {0:D}",currTime); // Saturday, September 24, 2005
  }
}
```

4.2 Array Class

The `Array` class

```
class Array : IClonable, ICollection, IEnumerable, IList
```

cotains methods for manipulations of arrays. For example

```
int BinarySearch(Array array,object value);
int BinarySearch(Array array,int index,int length,object value);
public virtual object Clone();
void Clear(Array array,int index,int length);
void Copy(Array source,Array destination,int length);
int IndexOf(Array array,object value);
void Reverse(Array array);
void Reverse(Array array,int index,int length);
```

An example is

```
// MyArray.cs

using System;
//using System.Collections.IList;

class MyArray
{
  public static void Main()
  {
  int[] a1 = { 669, 56, 45, 877, 123, 456, 777 };
  int p1 = Array.BinarySearch(a1,877);

  Console.WriteLine("p1: {0}", p1);
```

```
  string[] s1 = new string[] { "aaba", "baaba", "alla", "ana" };
  Array.Sort(s1);
  for(int i=0;i<s1.Length;i++)
  { Console.WriteLine("s1[" + i + "]=" + s1[i]); }

  Array.Reverse(s1);
  for(int i=0;i<s1.Length;i++)
  { Console.WriteLine("s1[" + i + "]=" + s1[i]); }

  Array.Clear(a1,3,2);
  for(int i=0;i<a1.Length;i++)
  { Console.WriteLine("a1[" + i + "]=" + a1[i]); }

  int j = Array.IndexOf(a1,123);
  Console.WriteLine("j = " + j); // j = -1 why?
  }
}
```

4.3 ArrayList Class

An `ArrayList` is used in situations where we want an array-like list, but cannot restrict the number of elements it may contain. The method `Add()` adds an element to the list. An application of the `ArrayList` class is given below.

```
// arraylist.cs

using System;
using System.Collections;

class arraylist
{
  public static void Main(string[] arg)
  {
  ArrayList alist = new ArrayList();
  foreach(string s in arg) alist.Add(s);

  for(int i=0;i<alist.Count;i++)
  {
  Console.Write("{0}",alist[i]);
  }
  Console.Write("\n");
  Console.Write("Argument Count:{0}\n",alist.Count);

  string s1 = "Xena";
```

```
  alist.Insert(3,s1);
  alist.RemoveAt(1);

  ArrayList nlist = new ArrayList();
  nlist = alist;

  bool test = alist.Equals(nlist);
  Console.WriteLine("{0}",test);
  } // end main
}
```

4.4 ListDictionary Class

The `ListDictionary` class stores a key and a value. The method

`public void Add(key,value)`

adds an element to the `ListDictionary`. The method

`void Remove(object key);`

removes an element in the dictionary.

```
// Dictionary.cs

using System;
using System.Collections;
using System.Collections.Specialized;

class Dictionary
{
  public static void Main()
  {
  ListDictionary population = new ListDictionary();
  population.Add("Johannesburg",9345120);
  population.Add("CapeTown",3500500);
  population.Add("Durban",550500);

  foreach(DictionaryEntry name in population)
  {
  Console.WriteLine("{0} = {1}",name.Key,name.Value);
  }
  population.Remove("Durban");
  foreach(DictionaryEntry name in population)
  {
  Console.WriteLine("{0} = {1}",name.Key,name.Value);
```

```
  }
  } // end Main()
}
```

4.5 Class IEnumerator

An enumerator object implements `IEnumerator` and $IEnumerator < T >$, where `T` is the yield type of the iterator block. It implements `System.IDisposable`. Memebers are `MoveNext`, `Current` and `Dispose`. The method

```
public virtual IEnumerator GetEnumerator()
```

returns a `System.Collections.IEnumerator` for the current instance.

```
// Enumerator.cs

using System;
using System.Collections;

public class ArrayGetEnumerator
{
  public static void Main()
  {
  string[,] strArry = {{"1","one"}, {"2","two"}, {"3","three"}};
  Console.Write("The elements of the array are: ");
  IEnumerator sEnum = strArry.GetEnumerator();

  while(sEnum.MoveNext())
    Console.Write(" {0}",sEnum.Current);
  }
}
```

4.6 Mathematics Class

The `Math` class is *sealed*. A sealed class cannot be used for inheritance. Additionally, all the classes and data members are static, so we cannot create an object of type `Math`. Instead, we use the members and methods with the class name. The `Math` class also includes two constants, `PI` and `E`.

The following table shows a partial list of the `Math` methods.

Method	Returns	Argument Data Type	Return Data Type
Abs(x)	Absolute value of x	Overloaded	Return matches argument type
Atan(x)	Angle in Radians whose tangent is x	Double	Double
Cos(x)	Cosine of x where x is in radians	Double	Double
Exp(x)	Value of e raised to the power of x	Double	Double
Log(x)	Natural log of x, where $x >= 0$	Double	Double
Max($x1, x2$)	Larger of the two arguments	Overloaded	Return matches argument type
Min($x1, x2$)	Smaller of the two arguments	Overloaded	Return matches argument type
Pow($x1, x2$)	Value of $x1$ raised to the power of $x2$	Double	Double
Round(x, y)	Value of x rounded to y decimal places	Overloaded	Return matches
Sin(x)	Sine of x where x is in radians	Double	Double
Sqrt(x)	Square root of x where $x <= 0$	Double	Double
Tan(x)	Tangent of x where x is in radians	Double	Double

For example, to round a double to 2 significant digits:

```
double x = 3.1415;
double xr = Math.Round(x,2);

// distance.cs

using System;

class Distance
{
  public static void Main()
  {
  double r = 6371.0;
  double alpha1, alpha2, beta1, beta2, temp, theta, distance;
  char dir;
  string source, dest, line;
  Console.WriteLine("Enter the name of the source: ");
  source = Console.ReadLine();
```

```
Console.WriteLine("Enter the latitude of {0}: ", source);
Console.WriteLine("degrees: ");
line = Console.ReadLine();
beta1 = (double) System.Convert.ToInt32(line);
Console.WriteLine("minutes: ");
line = Console.ReadLine();
temp = (double) System.Convert.ToInt32(line);
beta1 += temp/60.0;
Console.WriteLine("N/S: ");
line = Console.ReadLine();
dir = line[0];
if(dir == 'S' || dir == 's') beta1 = -beta1;
Console.WriteLine("Enter the longitude of {0}: ",source);
Console.WriteLine("degrees: ");
line = Console.ReadLine();
alpha1 = (double) System.Convert.ToInt32(line);
Console.WriteLine("minutes: ");
line = Console.ReadLine();
temp = (double) System.Convert.ToInt32(line);
alpha1 += temp/60.0;
Console.WriteLine("W/E: ");
line = Console.ReadLine();
dir = line[0];
if(dir == 'E' || dir == 'e') alpha1 = -alpha1;
Console.WriteLine("Enter the name of the destination: ");
dest = Console.ReadLine();
Console.WriteLine("Enter the latitude of {0}: ",dest);
Console.WriteLine("degrees: ");
line = Console.ReadLine();
beta2 = (double) System.Convert.ToInt32(line);
Console.WriteLine("minutes: ");
line = Console.ReadLine();
temp = (double) System.Convert.ToInt32(line);
beta2 += temp/60.0;
Console.WriteLine("N/S: ");
line = Console.ReadLine();
dir = line[0];
if(dir == 'S' || dir == 's') beta2 = -beta2;
Console.WriteLine("Enter the longitude of {0}: ",dest);
Console.WriteLine("degrees: ");
line = Console.ReadLine();
alpha2 = (double) System.Convert.ToInt32(line);
Console.WriteLine("minutes: ");
line = Console.ReadLine();
temp = (double) System.Convert.ToInt32(line);
```

```
  alpha2 += temp/60.0;
  Console.WriteLine("W/E: ");
  line = Console.ReadLine();
  dir = line[0];
  if(dir == 'E' || dir == 'e') alpha2 = -alpha2;
  alpha1 *= Math.PI/180.0; alpha2 *= Math.PI/180.0;
  beta1 *= Math.PI/180.0; beta2 *= Math.PI/180.0;
  temp = Math.Cos(beta1)*Math.Cos(beta2)*Math.Cos(alpha1-alpha2)
         + Math.Sin(beta1)*Math.Sin(beta2);
  theta = Math.Acos(temp);
  distance = r*theta;
  Console.WriteLine("The distance between {0} and {1} is {2} km",
                    source,dest,distance);
  }
}
```

4.7 Random Class

The next program shows an application of the `Random` class.

```
// random.cs

using System;

class LearnRandom
{
  public static void Main()
  {
  Random r = new Random();
  Console.WriteLine("Random sequence(no seed,limit 0..int.MaxVal)");
  Console.WriteLine();
  for(int i=0;i<10;i++)
  Console.WriteLine("random no: {0}",r.Next());
  Console.WriteLine("Random sequence(no seed,limit 0..10)");
  for(int i = 0;i<10;i++)
  Console.WriteLine("random no: {0}",r.Next(10));
  Console.WriteLine("Random sequence(no seed,limit 50..100)");
  for(int i=0;i<10;i++)
  Console.WriteLine("random no: {0}",r.Next(50,100));
  Console.WriteLine("Random sequence(no seed, limit 0..1)");
  for(int i=0;i<10;i++)
  Console.WriteLine("random no: {0}",r.NextDouble());
  Console.WriteLine("Random sequence in byte array(no seed,limit 0..1)");
  byte[] b = new byte[5];
  r.NextBytes(b);
```

```
  for(int i=0;i<b.Length;i++)
  Console.WriteLine("random byte: {0}",b[i]);
  }
}
```

4.8 Point Classes

The `Point` structure represents an ordered pair xy of integers. The `PointF` structure represents an ordered pair of floating point x- and y- coordinates that define a point in a two-dimensional plane.

```
// pointt.cs

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

class PointTest
{
  public static void Main()
  {
  Point p = new Point(23,13);
  Console.WriteLine("Our Point is: " + p);
  int xcoord = p.X;
  int ycoord = p.Y;
  Console.WriteLine("The x coordinate is " + xcoord);
  Console.WriteLine("The y coordinate is " + ycoord);
  Point q = new Point(xcoord,ycoord);
  bool b = p.Equals(q);
  Console.WriteLine(" the points are equal: " + b);

  PointF pF = new PointF((float)23.3333,(float)13.666666666667);
  Console.WriteLine("Our Point is: " + pF);
  float xcoordF = pF.X;
  float ycoordF = pF.Y;
  Console.WriteLine("The x coordinate is " + xcoordF);
  Console.WriteLine("The y coordinate is " + ycoordF);
  PointF qF = new PointF(xcoordF,ycoordF);
  b = pF.Equals(qF);
  Console.WriteLine(" the points are equal: " + b);
  }
}
```

4.9 Class BitArray

For dealing with bit string we use the class `BitArray`, where 1 is identified with `true` and 0 is identified with `false`. The constructor

```
BitArray a = new BitArray(16);
```

sets all elements to `false`.

```
// BitArrayTest.cs

using System;
using System.Collections;

public class BitArrayTest
{
  static void Main()
  {
  BitArray a = new BitArray(16);
  BitArray b = new BitArray(16);

  a[3] = a[4] = a[5] = true;
  b[4] = b[5] = b[6] = true;

  BitArray c = a.And(b);
  Console.WriteLine("Bitarray c");
  int i;
  for(i=0;i<16; i++)
  Console.WriteLine(c[i]);
  Console.WriteLine("");

  BitArray d = (BitArray) c.Clone();
  Console.WriteLine("Clone c into d");
  for(i=0;i<16;i++)
    Console.WriteLine(d[i]);

  BitArray e = a.Not();
  Console.WriteLine("Not a");
  for(i=0;i<16;i++)
  Console.WriteLine("e[" + i + "]=" + e[i]);

  a.SetAll(true);
  Console.WriteLine("a.SetAll(true)");
  for(i=0;i<16;i++)
  Console.WriteLine("f[" + i + "]=" + a[i]);
```

```
  BitArray g = d.Not();
  for(i=0;i<16;i++)
  Console.WriteLine("g[" + i + "]=" + g[i]);

  BitArray f = d.Xor(g);
  Console.WriteLine("Xor d to get f");
  for(i=0;i<16;i++)
  Console.WriteLine("f[" + i + "]=" + f[i]);
  }
}
```

4.10 Object Class

The `Object` class supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes. This is the ultimate superclass of all classes in the .NET Framework; it is the root of the type hierarchy. The C# syntax is

```
[Serializable]
public class Object
```

Public static (non-instance) members of this type are safe for multithreaded operations. Instance members are not guaranteed to be thread-safe. Languages typically do not require a class to declare inheeritance from `Object` since the inheritance is implicit.

Since all classes in the .NET Framework are derived from `Object`, every method defined in the `Object` class is available in all objects in the system. Derived classes can and do override some of these methods, including

```
Object.Equals - support comparisons between objects
Object.Finalize - performs cleanup operations before an object is
                  automatically reclaimed
Object.GetHashCode - generates a number corresponding to the value
                     of the object to support the use of a hash table
Object.ToString - manufactures a human-readable text that describes
                  an instance of the class
```

The default constructor is called by derived class constructors to initialize state in this type. Initializes a new instance of the `Object` class.

The method `GetHashCode()` serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.

The method `GetType()` gets the type of the current instance.

The following code compares the current instance with another object.

```
// object1.cs

using System;

public class object1
{
  public static void Main()
  {
  Object obj1 = new Object();
  Object obj2 = new Object();
  Console.WriteLine(obj1.Equals(obj2)); // => false
  obj2 = obj1;
  Console.WriteLine(obj1.Equals(obj2)); // => true
  }
}
```

The following example shows a `Point` class that overrides the `Object.Equals()` method to provide value equality and a class `Point3D`, which is derived from the `Point` class. The `Object.Equals()` method uses `Object.GetType()` to determine whether the run-time types of the two objects are identical.

```
// object2.cs

using System;

class Point : Object
{
  protected int x, y;

  public Point() { this.x = 0; this.y = 0; }

  public Point(int X,int Y)
  { this.x = X; this.y = Y; }

  public override bool Equals(Object obj)
  { // check for null and compare run-time types
  if(obj == null || GetType() != obj.GetType()) return false;
  Point p = (Point) obj;
  return (x == p.x) && (y == p.y);
  }
```

```
  public override int GetHashCode()
  { return x^y; }  // ^ XOR operation
} // end class Point

  class Point3D : Point
  {
  int z;

  public Point3D(int X,int Y,int Z)
  {
  this.x = X; this.y = Y; this.z = Z;
  }

  public override bool Equals(Object obj)
  { return base.Equals(obj) && z == ((Point3D) obj).z; }

  public override int GetHashCode()
  { return base.GetHashCode() ^ z; }
} // end class Point3D

public class object1
{
  public static void Main()
  {
  Point p1 = new Point(5,7);
  Point p2 = new Point(6,8);
  bool b1 = p1.Equals(p2);
  Console.WriteLine("b1 = " + b1);

  int h1 = p1.GetHashCode();
  Console.WriteLine("h1 = " + h1);

  Point3D p3 = new Point3D(5,6,9);
  Point3D p4 = new Point3D(5,6,9);
  bool b2 = p3.Equals(p4);
  Console.WriteLine("b2 = " + b2);

  int h2 = p3.GetHashCode();
  Console.WriteLine("h2 = " + h2);
  }
}
```

The method

```
protected object MemberwiseClone();
```

creates a shallow copy of the current object, i.e. it returns a shallow copy of the current object. The following code shows how to copy an instance of a class using `Object.MemberwiseClone()`.

```
// object3.cs

using System;

class MyBaseClass
{
  public static string CompanyName = "KXK";
  public int age;
  public string name;
}

class MyDerivedClass : MyBaseClass
{
  public static void Main()
  {
  // create an instance of MyDerivedClass and assign values
  // to its fields
  MyDerivedClass m1 = new MyDerivedClass();
  m1.age = 42;
  m1.name = "John";

  // performs a shallow copy of m1 and assign it to m2
  MyDerivedClass m2 = (MyDerivedClass) m1.MemberwiseClone();
  Console.WriteLine("age = " + m2.age);
  Console.WriteLine("name = " + m2.name);
  }
}
```

4.11 Environment Class

The `Environment` class provides information about, and means to manipulate, the current environment and platform. The class is sealed, i.e. it cannot be inherited.

`Version` gets a Version object that describes the major, minor, build, and revision numbers of the common language runtime. `CommandLine` provides the command line for this process. `CurrentDirectory` provides the directory from which this process starts. `ExitCode` gets or sets the exit code of the process. `StackTrace` provides current stack trace information. `SystemDirectory` provides as string the fully qualified path of the system directory. `TickCount` provides the number of milliseconds the time passed since the system started. The method `GetCommandLineArgs()` returns a string array containing the command line arguments of the current process. The

method `Exit()` terminates this process and gives the underlying operating system the specified exit code. The class also contain the methods `ToString` and *Equals*.

```
// enviro.cs

using System;

class MyEnviro
{
   public static void Main()
   {
   object ver = Environment.Version;
   Console.WriteLine("ver = " + ver);  // 2.0.50727.312

   string com = Environment.CommandLine;
   Console.WriteLine("com = " + com); // enviro

   string cd = Environment.CurrentDirectory;
   Console.WriteLine("cd = " + cd);  // C:\csharp

   int ex = Environment.ExitCode;
   Console.WriteLine("ex = " + ex); // 0

   string st = Environment.StackTrace;
   Console.WriteLine("st = " + st); // at System.Environment ....
                                       ... at MyEnviro.Main()
   string sd = Environment.SystemDirectory;
   Console.WriteLine("sd = " + sd); // C:\Windows\System32

   int t = Environment.TickCount;
   Console.WriteLine("t = " + t);  // 300223

   string[] comline = Environment.GetCommandLineArgs();
   Console.WriteLine("comline = " + comline); // System.String[]

   int i = 7;
   int x = 15/2;
   if(x==i) Environment.Exit(0);  // true
   Console.WriteLine("did we reach this point?");
   }
}
```

Chapter 5

Object-Oriented Programming

5.1 Write your own class

In CSharp we can write our own class. Object-oriented programming is essentially programming in terms of smaller units called objects. An object oriented program is composed of one or more objects. Each object holds some data (fields or attributes) as defined by its class. The class also defines a set of functions (also called methods or operations) which can be invoked on its objects. Generally the data is hidden within the objects (instances) and can be accessed only through functions defined by its class (encapsulation). One or more objects (instances) can be created from a class by a process called instantiation. The process of deciding which attributes and operations will be supported by an object (i.e. defining the class) is called abstraction. We say the state of the object is defined by the attributes it supports and its behaviour is defined by the operations it implements. The term passing a message to an object means invoking its operation. Sometimes the set of operations supported by an object is also referred to as the interface exposed by this object.

C# also introduces *properties*. Properties act like methods to the creator of the class but look like fields to the client of the class.

Polymorphism refers to the ability of objects to use different types without regard to the details.

Constructors are methods that are invoked when objects are instantiated. The CLR defines one, but it is better practice to code the constructors that are required. All objects are created using `new`.

A Default Constructor Generated by compiler if none exists. They have the same name as the class, no return type, no arguments are required, all fields are initialized to zero, have public accessibility. Note that writing any constructor stops default creation.

A Private Constructor prevents unwanted objects from being created. Instance

methods cannot be called. Static methods can be called. Commonly used to implement procedural functions.

A Static Constructor is called by class loader at run time. It is used to initialize static members and is guaranteed to be called before instance constructor. It cannot have parameters. Use this to declare an access modifier.

A default constructor may look like the following example.

```
public Person1() {  }
```

The following program shows an example for class `Person`. The `Person` is described by his name, age and sex. Thus we have three attributes:

`myName, myAge, mySex.`

These `private` data members are only visible inside the body of class `Person1`.

```
// Person1.cs

using System;

class Person1
{
  private string myName = "N/A";
  private int myAge = 0;
  private char mySex = 'n';

  // default constructor
  public Person1() {  }

  // constructor
  public Person1(string name,int age,char sex)
  {
  this.myName = name;
  this.myAge = age;
  this.mySex = sex;
  }

  // declare a Name property of type string
  public string Name
  {
  get { return myName; }
  set { myName = value; }
  }
```

```
public int Age
{
get { return myAge; }
set { myAge = value; }
}

public char Sex
{
get { return mySex; }
set { mySex = value; }
}

public override string ToString()
{
return "Name = " + Name + ", Age = " + Age + ", Sex = " + Sex;
}

public override bool Equals(object o)
{
if((myName==((Person1) o).myName) &&
   (myAge==((Person1) o).myAge) &&
   (mySex==((Person1) o).mySex))
return true;
else return false;
}

public static void Main()
{
// create a new Person object using default constructor
Person1 person1 = new Person1();

// Set some values on the person object
person1.Name = "Joe";
person1.Age = 99;
person1.Sex = 'm';
Console.WriteLine("Person details - {0}",person1);

Person1 person2 = new Person1();
person2.Name = "Jane";
person2.Age = 31;
person2.Sex = 'f';
Console.WriteLine("Person details - {0}",person2);

Person1 person3 = new Person1();
person3.Name = "Jane";
```

```
  person3.Age = 31;
  person3.Sex = 'f';
  Console.WriteLine("Person details - {0}",person3);

  bool same = person2.Equals(person3);
  Console.WriteLine("same person = " + same);

  Person1 person4 = new Person1("otto",42,'m');
  Console.WriteLine("Person details: " + person4.myName);
  Console.WriteLine("Person details: " + person4.mySex);

  // array of persons
  Person1[] pmany = new Person1[2];
  pmany[0] = new Person1("Carl",23,'m');
  pmany[1] = new Person1("Ola",7,'f');
  Console.WriteLine("name of person[0]: " + pmany[0].myName);
  Console.WriteLine("sex of person[1]: " + pmany[1].mySex);
  } // end Main
}
```

`class Person1` defines three private member variables, `myName`, `myAge`, `mySex`. These variable are visible only inside the body of `class Person1`. To access these variable outside the class we use the special property methods `Name`, `Age`, and `Sex`. The CSharp compiler translates for example the `Name` property to a pair of methods, namely

```
public string get_Name()
{ return myName; }

public void set_Name(string value)
{ myName = value; }
```

The `class Person1` contains the `Main()` method. In some application it would be better to keep the `class Person1` without the `Main()` method and have an extra file which calls the `Person` objects. The file `Person.cs` only contains the class `Person` and no `Main()` method.

```
// Person.cs

using System;

class Person
{
  public string myName = "N/A";
  public int myAge = 0;
  public char mySex = 'n';
```

```
  // default constructor
  public Person() {  }

  // constructor
  public Person(string name,int age,char sex)
  {
  this.myName = name;
  this.myAge = age;
  this.mySex = sex;
  }

  // declare a Name property of type string
  public string Name
  {
  get { return myName; }
  set { myName = value; }
  }

  public int Age
  {
  get { return myAge; }
  set { myAge = value; }
  }

  public char Sex
  {
  get { return mySex; }
  set { mySex = value; }
  }

  public override string ToString()
  {
  return "Name = " + Name + ", Age = " + Age + ", Sex = " + Sex;
  }

  public override bool Equals(object o)
  {
  if((myName==((Person) o).myName) &&
     (myAge==((Person) o).myAge) &&
     (mySex==((Person) o).mySex))
  return true;
  else return false;
  }
}
```

To generate the `Person.dll` file we run

```
csc /t:library Person.cs
```

The file `PersonMain.cs` contains the `Main()` method.

```
// PersonMain.cs

using System;

class PersonMain
{
  public static void Main()
  {
  // create a new Person object using default constructor
  Person person1 = new Person();

  // Set some values on the person object
  person1.Name = "Joe";
  person1.Age = 99;
  person1.Sex = 'm';
  Console.WriteLine("Person details - {0}",person1);

  Person person2 = new Person();
  person2.Name = "Jane";
  person2.Age = 31;
  person2.Sex = 'f';
  Console.WriteLine("Person details - {0}",person2);

  Person person3 = new Person();
  person3.Name = "Jane";
  person3.Age = 31;
  person3.Sex = 'f';
  Console.WriteLine("Person details - {0}",person3);

  bool same = person2.Equals(person3);
  Console.WriteLine("same person = " + same);

  // increment the age of person3
  person3.Age++;
  Console.WriteLine("person3 new age: " + person3.Age);

  Person person4 = new Person("otto",42,'m');
  Console.WriteLine("Person details: " + person4.myName);
  Console.WriteLine("Person details: " + person4.mySex);
```

```
  // array of persons
  Person[] pmany = new Person[2];
  pmany[0] = new Person("Carl",23,'m');
  pmany[1] = new Person("Ola",7,'f');
  Console.WriteLine("name of person[0]: " + pmany[0].myName);
  Console.WriteLine("sex of person[1]: " + pmany[1].mySex);
  } // end Main
}
```

To get the `PersonMain.exe` file we run

```
csc /r:Person.dll PersonMain.cs
```

However, we can also use the short-cut

```
csc Person.cs PersonMain.cs
```

to generate the `PersonMain.exe` file.

5.2 Override Methods

When we write our own class in most cases we should `override` the methods

```
string ToString(),   bool Equals(),   object Clone()
```

An example is given below

```
// Point3D.cs

using System;

public class Point3D
{
  public double X;
  public double Y;
  public double Z;

  public Point3D() { } // default constructor

  public Point3D(double x,double y,double z)
  {
  this.X = x;
  this.Y = y;
  this.Z = z;
  }
```

```
  // square of distance between two Points
  public double distance(Point3D p1,Point3D p2)
  {
  return (p1.X-p2.X)*(p1.X-p2.X)+(p1.Y-p2.Y)*(p1.Y-p2.Y)+(p1.Z-p2.Z)*(p1.Z-p2.Z);
  }

  public override string ToString()
  {
  return String.Format("({0},{1},{2})",X,Y,Z);
  }

  public override bool Equals(object o)
  {
  if((X==((Point3D) o).X) && (Y==((Point3D) o).Y) && (Z==((Point3D) o).Z))
  return true;
  else return false;
  }

  public object Clone()
  {
  object o = new Point3D(X,Y,Z);
  return o;
  }
}
```

An application of this class is

```
// Point3DMain.cs

using System;

public class Point3DMain
{
  public static void Main()
  {
  double x = 2.1;
  double y = 3.1;
  double z = 4.5;
  Point3D p1 = new Point3D(x,y,z);
  Console.WriteLine("p1 = " + p1);

  double a = 2.1;
  double b = 3.1;
  double c = 4.5;
  Point3D p2 = new Point3D(a,b,c);
```

```
  bool r = p1.Equals(p2);
  Console.WriteLine("r = " + r);

  double k = 2.0;
  double m = 1.0;
  double n = 7.5;
  Point3D p3 = new Point3D(k,m,n);
  Point3D p4 = new Point3D(0.0,0.0,0.0);
  p4 = (Point3D) p3.Clone();   // type conversion
  Console.WriteLine("p4 = " + p4);

  Point3D q = new Point3D();
  double d = q.distance(p1,p3);
  Console.WriteLine("d = " + d);
  }
}
```

5.3 Inheritance

Inheritance is one of the primary concepts of object-oriented programming. It allows us to reuse existing code. In the next program we use *inheritence*. The `class Car` is derived from `class Vehicle`. In the `class Car` we use code from the `class Vehicle`. First we must declare our intention to use `Vehicle` as the `base` class of `Car`. This is accomplished through the `Car` class declaration

```
public class Car : Vehicle
```

Then the colon, `:` , and the keyword `base` call the `base` class constructor. `C#` supports single class inheritance only.

A sealed class protects against inheritence.

An abstract class can only be used as a base class of other classes. They cannot be instantiated and may contain abstract methods and accessors. It is not possible to modify an abstract class with the sealed modifier, thus the class cannot be inherited.

```
// MVehicle.cs

using System;

class Vehicle
{
  private int weight;
  private int topSpeed;
```

```
  private double price;

  public Vehicle() {}

  public Vehicle(int aWeight,int aTopSpeed,double aPrice)
  {
  weight = aWeight;
  topSpeed = aTopSpeed;
  price = aPrice;
  }

  public int getWeight() { return weight;}

  public int getTopSpeed() { return topSpeed; }

  public double getPrice() { return price; }

  public virtual void print()
  {
  Console.WriteLine("Weight: {0} kg",weight);
  Console.WriteLine("Top Speed: {0} km/h",topSpeed);
  Console.WriteLine("Price: {0} Dollar",price);
  }
} // end class Vehicle

class Car : Vehicle
{
  private int numberCylinders;
  private int horsepower;
  private int displacement;

  public Car() { }

  public Car(int aWeight,int aTopSpeed,double aPrice,int aNumberCylinders,
             int aHorsepower,int aDisplacement) : base(aWeight,aTopSpeed,aPrice)
  {
  numberCylinders = aNumberCylinders;
  horsepower = aHorsepower;
  displacement = aDisplacement;
  }

  public int getNumberCylinders() { return numberCylinders; }
  public int getHorsepower() { return horsepower; }
  public int getDisplacement() { return displacement; }
```

```
  public override void print()
  {
  base.print();
  Console.WriteLine("Cylinders: {0} ",numberCylinders);
  Console.WriteLine("Horsepower: {0} ",horsepower);
  Console.WriteLine("Diplacement: {0} cubic cm",displacement);
  }
}

class myCar
{
  public static void Main()
  {
  Vehicle aVehicle = new Vehicle(15000,120,30000.00);
  Console.Write("A vehicle: ");
  aVehicle.print();
  Console.WriteLine("");
  Car aCar = new Car(3500,100,12000.00,6,120,300);
  Console.Write("A car: ");
  aCar.print();
  Console.WriteLine("");
  }
}
```

An *interface* looks like a class, but has no implementation. The only thing it contains are definitions of events, indexers, methods, and/or properties. The reason interfaces only provide definitions is because they are inherited by classes and structs, which must provide an implementation for each interface member defined. Since interfaces must be defined by inheriting classes and structs, they must define a contract. The following program shows an example.

```
// Policies.cs

public interface Policy
{
  double Rate(double principal,int period);
}

public class BronzePolicy : Policy
{
  public double Rate(double p,int n)
  { return 7; }
}

public class SilverPolicy : Policy
```

```
{
  public double Rate(double p,int n)
  { return (p < 25000.0) ? 8 : 10; }
}

public class GoldPolicy
{
  public double Rate(double p,int n)
  { return (n < 5) ? 9 : 11; }
}

public class PlatinumPolicy : Policy
{
  public double Rate(double p,int n)
  {
  double r = (p < 50000.0) ? 10 : 12;
  if(n >= 3) r += 1;
  return r;
  }
}
```

Each `policy` provides a `Rate` method which returns the rate of interest for a given principal and for a given period. All policy classes implement the `Policy` interface except `GoldPolicy` which defines `Rate` without implementing `policy`. We compile `Policies.cs` to create `Policies.dll` via

```
csc /t:library Policies.cs
```

Using reflection we can also create an instance of a class and invoke its member methods at runtime. This feature can be used to write more generic (dynamically extensible) programs which receive names of classes at runtime.

The `investment.cs` program including `Main` accepts `principal`, `period` and `policy` name as arguments `args[0]`, `args[1]`, and `args[2]`. We compile `investment.cs` as

```
csc /r:Policies.dll investment.cs
```

We would execute the program, for example,

```
investment 60000 5 SilverPolicy,Policies
```

```
// investment.cs

using System;
using System.Reflection;
```

```
class Investment
{
  public static void Main(string[] args)
  {
  double p = Double.Parse(args[0]); // string to double
  int n = Int32.Parse(args[1]); // string to int
  Type t = Type.GetType(args[2]);
  Policy pol = (Policy) Activator.CreateInstance(t);
  double r = pol.Rate(p,n);
  double amount = p*Math.Pow(1.0+r/100.0,n);
  Console.WriteLine("you will get {0:#.00}",amount);
  }
}
```

5.4 Overloading Methods

Methods in a class can be overloaded. The methods of a class may have the same name
if they have different numbers of parameters, or
if they have different parameter types, or
if they have different parameter kinds.

We overload the method `max()` to find the maximum of numbers.

```
// methods.cs

using System;

public class Maximizer
{
  public class Maximum
  {
  public static double max(double p,double q)
  {
  if(p > q) return p;
  return q;
  }

  public static double max(double p,double q,double r)
  {
  return max(max(p,q),r);
  }

  public static double max(double[] list)
```

```
  {
  if(list.Length == 0) return 0;
  double max = list[0];
  foreach(double val in list)
  {
  if(val > max) max = val;
  }
  return max;
  }

  public static void Main(string[] args)
  {
  Console.WriteLine("maximum of 4.5 and 4.7 is {0}",max(4.5,4.7));
  Console.WriteLine("maximum of 3.1, 2.4, 4.9 is {0}",max(3.1,2.4,4.9));
  double[] array = new double[4];
  array[0] = 3.1; array[1] = 5.7; array[2] = 3.9; array[3] = 2.1;
  Console.WriteLine("maximum element in array = {0}",max(array));
  } // end main
  }
}
```

Exercise. Add a method `max(int,int,int,int)` to the program that find the maximum of four integer numbers.

5.5 Operator Overloading

Operators can be overloaded (*operator overloading*) such that

`+, -, *, \, %, []`

In the next program we overload `+` to add two complex numbers.

```
// Complex.cs

using System;

public class Complex
{
  public double real;  // real part of complex number
  public double imag;  // imaginary part of number

  public Complex(double real,double imag)
  {
  this.real = real;
  this.imag = imag;
```

```
  }

  public static Complex operator + (Complex c1,Complex c2)
  {
  c1.real = c1.real + c2.real;
  c1.imag = c1.imag + c2.imag;
  return c1;
  }
}

class ComplexMain
{
  public static void Main()
  {
  Complex r1 = new Complex(1.0,2.0);
  Complex r2 = new Complex(2.0,1.0);
  r1 = r1 + r2;
  Console.WriteLine("Real: {0} Imaginary: {1}i",r1.real,r1.imag);
  }
}
```

Exercise. Overload in the program also * for the multiplication of two complex numbers. The multiplication of two complex numbers $z_1 = a + ib$ and $z2 = c + id$ (with a, b, c, d real) is given by $z_1 * z_2 = a * c - b * d + i(ad + bc)$.

5.6 Structures and Enumerations

In CSharp we can also use *structures*. Structures, or *structs*, contain properties, methods, fields, operators, nested types, indexers and structors. Structs do not support inheritence or destructors. The objects are stored on the stack compared to classes which are stored on the heap. Structs are useful for small data structures such as complex numbers, key-value pairs in a dictionary or points in a coordinate system. They should only be used for types that are small and simple.

Enumerations are an alternative to constants when grouping related constants together. For example, when considering temperature we can have the following.

```
enum Temp
{
  AbsoluteZero = 0,
  CosmicBackground = 2.75,
  WaterFreexingPoint = 273,
  WaterBoilingPoint = 373,
}
```

The base type for enums is integer. The following example demonstrates how to use different types.

```
enum NumberChocolatesInBox :uint
{
  SmallBox = 20,
  MediumBox = 30,
  LargeBox = 50,
}
```

The program shows how *structs* and *enumeration* can be used.

```
// enumeration.cs

using System;

enum MemberType { Lions, Elefant, Jackals, Eagles, Dogs };

struct ClubMember
{
  public string Name;
  public int Age;
  public MemberType Group;
}

class enumeration
{
  public static void Main(string[] args)
  {
  ClubMember a; // Value types are automatically initialized
  a.Name = "John";
  a.Age = 13;
  a.Group = MemberType.Eagles;

  ClubMember b = a; // new copy of a is assigned to b
  b.Age = 17; // a.Age remains 13
  Console.WriteLine("Member {0} is {1} year old and belongs to group of {2}",
                    a.Name,a.Age,a.Group);
  } // end main
}
```

5.7 Delegates

A *delegate* in C# allows us to pass methods of one class to objects of other classes that can call those methods. We can pass method `m` in class A, wrapped in a dele-

gate, to class B and class B will be able to call method `m` in class A. We can pass both static and instance methods. This concept is familiar to C++ where one uses function pointers to pass functions as parameters to other methods in the same class or in another class. C# delegates are implemented in the .NET framework as a class derived from `System.Delegate`. Thus the use of delegates involves four steps.

1. Declare a delegate object with a signature that exactly matches the method signature that we are trying to encapsulate.
2. Define all the methods whose signature match the signature of the delegate object that we have defined in step 1.
3. Create delegate object and plug in the methods that we want to encapsulate.
4. Call the encapsulated methods through the delegate object.

The following C# program shows the above four steps.

```
// Delegate.cs

using System;

// Step 1. Declare a delegate with the signature of the
// encapsulated method
public delegate void MyDelegate(string input);

// Step 2. Define methods that match with the signature
// of delegate declaration
class Class1
{
  public void delegateMethod1(string input)
  {
  Console.WriteLine("delegateMethod1: the input to the method is {0}",input);
  }

  public void delegateMethod2(string input)
  {
  Console.WriteLine("delegateMethod2: the input to the method is {0}",input);
  }
} // end class Class1


// Step 3. Create delegate object and plug in the methods
class Class2
{
  public MyDelegate createDelegate()
  {
  Class1 c2 = new Class1();
```

```
  MyDelegate d1 = new MyDelegate(c2.delegateMethod1);
  MyDelegate d2 = new MyDelegate(c2.delegateMethod2);
  MyDelegate d3 = d1 + d2;
  return d3;
  }
} // end class Class2

// Step 4. Call the encapsulated method through the delegate
class Class3
{
  public void callDelegate(MyDelegate d,string input)
  {
  d(input);
  }
} // end class Class3

class Driver
{
  public static void Main()
  {
  Class2 c2 = new Class2();
  MyDelegate d = c2.createDelegate();
  Class3 c3 = new Class3();
  c3.callDelegate(d,"calling the delegate");
  } // end Main
}
```

The output is:

```
delegateMethod1: the input to the method is calling the delegate
delegateMethod2: the input to the method is calling the delegate
```

Another example is given below. Here we use delegates to compare strings.

```
// MyDelegate.cs

using System;

// this is the delegate declaration
public delegate int Comparer(object obj1,object obj2);

public class Name
{
  public string FirstName = null;
  public string LastName = null;
```

```
  public Name(string first,string last)
  {
  FirstName = first;
  LastName = last;
  }

  public static int CompareFirstNames(object name1,object name2)
  {
  string n1 = ((Name) name1).FirstName; // type conversion
  string n2 = ((Name) name2).FirstName; // type conversion

  if(string.Compare(n1,n2) > 0) { return 1; }
  else if(string.Compare(n1,n2) < 0) { return -1; }
  else { return 0; }
  } // end method CompareNames()

  public override string ToString()
  { return FirstName + " " + LastName; }
} // end class Name

class SimpleDelegate
{
  Name[] names = new Name[4];

  public SimpleDelegate()
  {
  names[0] = new Name("John","Smithlone");
  names[1] = new Name("Carl","Xenon");
  names[2] = new Name("Rolf","Cooper");
  names[3] = new Name("Ola","Jones");
  }

  static void Main()
  {
  SimpleDelegate sd = new SimpleDelegate();

  // the delegate instantiation
  Comparer cmp = new Comparer(Name.CompareFirstNames);

  Console.WriteLine("\nBefore Sort:\n");
  sd.PrintNames();

  // observe the delegate argument
  sd.Sort(cmp);
```

```
  Console.WriteLine("\nAfter Sort:\n");

  sd. PrintNames();
  }

  public void Sort(Comparer compare)
  {
  object temp;

  for(int i=0;i<names.Length;i++)
  {
  for(int j=i;j<names.Length;j++)
  {
  // using delegate "compare" just like a normal method
  if(compare(names[i],names[j])>0)
  {
  temp = names[i];
  names[i] = names[j];
  names[j] = (Name) temp;
  } // end if
  }
  }
  } // end method Sort

  public void PrintNames()
  {
  Console.WriteLine("Names:\n");

  foreach(Name name in names)
  {
  Console.WriteLine(name.ToString());
  }
  } // end method PrintNames()
}
```

Chapter 6

Streams and File Manipulations

6.1 Introduction

A stream is a sequence of bytes. A byte is 8 bits. The `Stream` class and its subclasses provide a generic view of data sources and repositories, isolating the programmer from the specific details of the operating system and underlying devices. Streams involve the following three operations:

1) Streams can be read from. Reading is the transfer of data from a stream into a data structure, such as an array of bytes.

2) Streams can be written to. Writing is the transfer of data from a data structure into a stream.

3) Streams can support seeking. Seeking is the query and modifying of the current position within a stream.

6.2 Binary File Manipulations

`System.IO.BinaryWriter` and `System.IO.BinaryReader` can be used for writing and reading primitive data types

```
byte, short, int, long, float, double, bool, char
```

and also the data type `string` and `StringBuilder` to a stream (binary file manipulation).

```
// BinaryIO1.cs

using System;
using System.IO;
using System.Text;

class BinaryIO1
{
  private static void WriteData(int i,double d,char c,bool b)
  {
  Stream s = File.OpenWrite("info.dat");
  BinaryWriter bw = new BinaryWriter(s);
  bw.Write(i);
  bw.Write(d);
  bw.Write(c);
  bw.Write(b);
  bw.Close();
  s.Close();
  }

  private static void ReadData()
  {
  Stream s = File.OpenRead("info.dat");
  BinaryReader br = new BinaryReader(s);
  int i = br.ReadInt32();
  double d = br.ReadDouble();
  char c = br.ReadChar();
  bool b = br.ReadBoolean();
  br.Close();
  s.Close();
  Console.WriteLine("{0},{1},{2},{3}",i,d,c,b);
  }

  public static void Main(string[] args)
  {
  WriteData(12345,3.14159,'y',true);
  ReadData();
  }
}

// BinaryIO2.cs

using System;
using System.IO;
using System.Text;
```

```
class BinaryIO2
{
  private static void WriteData(string t,StringBuilder sb)
  {
  Stream s = File.OpenWrite("info.dat");
  BinaryWriter bw = new BinaryWriter(s);
  // write length-prefixed string
  bw.Write(t);
  bw.Write(sb.ToString());
  bw.Close();
  s.Close();
  }

  private static void ReadData()
  {
  Stream s = File.OpenRead("info.dat");
  BinaryReader br = new BinaryReader(s);
  string t = br.ReadString();
  string sb = br.ReadString();
  br.Close();
  s.Close();
  Console.WriteLine("{0},{1}",t,sb);
  }

  public static void Main(string[] args)
  {
  StringBuilder sb = new StringBuilder("Hello World!");
  WriteData("Hello Egoli",sb);
  ReadData();
  }
}
```

6.3 Text File Manipulation

Reading a text file. The class `FileStream` is useful for reading and writing files on a file system, as well as other file-related operating system handles (including pipes, standard input, standard output, and so on). `FileStream` buffers input and output for better performance. The `FileStrean` class can open a file in one of two modes, either synchronously or asynchronously, with signigicant performance consequences for the synchronous methods (`FileStream.Read` and `FileStream.Write`) and the asynchronous methods `FileStream.BeginRead` and `FileStream.BeginWrite`). Both sets of methods will work in either mode; however, the mode will affect the performance of these methods. `FileStream` defaults to opening files synchronously, but

provides a constructor to open files asynchronously. `FileStream` objects support random access to files using the `FileStream.Seek` method. The `Seek` method allows the read/write position to be moved to any position within the file. This is done with byte offset reference point parameters. The byte offset is relative to the seek reference point, which can be the beginning, the current position, or the end of the underlying file, as represented by the three properties of the `SeekOrigin` class.

We convert the array of bytes into an ASCII text for displaying it on the console.

```
// readfile.cs

using System;
using System.IO;
using System.Text; // for encoding

class ReadFile
{
  public static void Main(string[] args)
  {
  Stream s = new FileStream(args[0],FileMode.Open);
  int size = (int) s.Length;
  byte[] buffer = new byte[size];
  s.Read(buffer,0,buffer.Length);
  s.Close();
  string text = Encoding.ASCII.GetString(buffer);
  Console.WriteLine(text);
  }
}
```

Writing to a file (text mode). We append the text provided by the line

```
string text = "all the good men and women" + "\r\n";
```

to the file provided by `args[0]`.

```
// writefile.cs

using System;
using System.IO;
using System.Text; // for encoding

class WriteFile
{
  public static void Main(string[] args)
  {
  Stream s = new FileStream(args[0],FileMode.Append,FileAccess.Write);
```

```
  string text = "all the good men and women" + "\r\n"; // append end
                                                      // of line characters
  byte[] buffer = Encoding.ASCII.GetBytes(text);
  s.Write(buffer,0,buffer.Length);
  s.Close(); // also flushes the stream
  }
}
```

Writing the date and time into a file using the `DateTime` class..

```
// placeorder.cs

using System;
using System.IO;

class PlaceOrder
{
  public static void Main(string[] args)
  {
  DateTime dt = DateTime.Now;
  string today = dt.ToString("dd-MMM-yyyy");
  string record = today + "|" + String.Join("|",args);
  StreamWriter sw = new StreamWriter("order.txt",true);
  sw.WriteLine(record);
  sw.Close();
  }
}
```

The useful classes in reading/writing text files are: `StreamReader` and `StreamWriter`. Given the file `datain.txt` with the two lines

```
Green 4750 3000.40
Bakley 3451 2800.50
```

the following program reads the file line by line and displays it on the Console and also writes it to the output file `dataout.txt`.

```
// Filemanipulation.cs

using System;
using System.IO;

class FileManipulation
{
  static void Main(string[] args)
  {
```

```
  FileInfo fr = new FileInfo(@"c:\csharp\datain.txt");
  StreamReader reader = fr.OpenText();
  string sline;
  string[] substrings;
  char[] separators = { ' ',',','\t' };
  string name;
  int ID;
  float gpa;
  StreamWriter writer =
       new StreamWriter(@"c:\csharp\dataout.txt",false);
  sline = reader.ReadLine();
  while(sline != null)
  {
  substrings = sline.Split(separators,3);
  name = substrings[0];
  ID = int.Parse(substrings[1]);
  gpa = float.Parse(substrings[2]);
  Console.WriteLine("{0} {1} {2}",name,ID,gpa);
  writer.WriteLine("{0} {1} {2}",name,ID,gpa);
  sline = reader.ReadLine();
  }
  writer.Flush();
  writer.Close();
  }
}
```

6.4 Byte by Byte Manipulation

For reading and writing bytes we use the methods `ReadByte()` and `WriteByte()`. The type conversion to `char` is necessary in the following program. What happens if we remove `(char)` ?

```
// BytebyByte.cs

using System;
using System.IO;

public class BytebyByte
{
  public static void Main(string[] args)
  {
  FileStream s = new FileStream(args[0],FileMode.Open,FileAccess.Read,
                                FileShare.Read);
  BufferedStream bs = new BufferedStream(s);
  while(bs.Length > bs.Position)
```

```
  {
  Console.Write((char) bs.ReadByte()); }
  }
  bs.Close();
  s.Close();
}
```

In the following example we copy byte by byte a file into another file.

```
// TestFile.cs

using System;
using System.IO;

class TestFile
{
  static void Copy(string from,string to,int pos)
  {
  try
  {
  FileStream sin = new FileStream(from,FileMode.Open);
  FileStream sout = new FileStream(to,FileMode.Create);
  sin.Seek(pos,SeekOrigin.Begin);

  int ch = sin.ReadByte();

  while(ch >= 0)
  {
  sout.WriteByte((byte) ch); // type conversion
  ch = sin.ReadByte();
  }
  sin.Close();
  sout.Close();
  }
  catch (FileNotFoundException e)
  {
  Console.WriteLine("--file {0} not found",e.FileName);
  }
  }

  public static void Main(string[] arg)
  {
  Copy(arg[0],arg[1],0);
  }
}
```

In the next program we count the number of curly brackets in a file.

```
// Oops1.cs

using System;
using System.IO;

class Oops
{
  public static void Main()
  {
  char cl = '{';
  int countcl = 0;
  char cr = '}';
  int countcr = 0;

  FileStream fin = new FileStream("data.dat",FileMode.Open,FileAccess.Read);
  BufferedStream bs = new BufferedStream(fin);

  while(bs.Length > bs.Position)
  {
  char ch = (char) bs.ReadByte();
  if(ch == cl) countcl++;
  if(ch == cr) countcr++;
  } // end while
  fin.Close();
  Console.WriteLine("countcl = " + countcl);
  Console.WriteLine("countcr = " + countcr);
  } // end Main
}
```

Exercise. Extend the program so that it also counts the number of rectangular brackets (left and right) and parentheses (left and right).

6.5 Object Serialization

Often we want to store a complete object in a stream. This is achieved by a process called object serialization. By *serialization* we mean converting an object, or a connected graph of objects (a set of objects with some set of references to each other), stored within memory into a linear sequence of bytes. This string of bytes contains all of the information that was held in the object started with.

```
// employee.cs

using System;
```

```
using System.Runtime.Serialization;

[Serializable]
public class Employee
{
  public string Name;
  public string Job;
  public double Salary;

  public Employee(string name,string job,double salary)
  {
  Name = name;
  Job = job;
  Salary = salary;
  }

 public Employee() { }

  // we override the ToString() method of System.Object.
  // This method is invoked whenever an Employee object
  // is to be converted to a string
  public override string ToString()
  {
  return String.Format("{0} is a {1} and earns {2}",Name,Job,Salary);
  }
} // end class Employee end file
```

In the following program we apply now object serialization to `Employee`.

```
// binsertest1.cs

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

class BinarySerializatioTest1
{
  static BinaryFormatter bf = new BinaryFormatter();

  private static void WriteEmployee(Employee emp)
  {
  Stream s = File.OpenWrite("emp.bin");
  bf.Serialize(s,emp);
  s.Close();
  }
```

```
  private static void ReadEmployee()
  {
  Stream s = File.OpenRead("emp.bin");
  Employee emp = (Employee) bf.Deserialize(s);
  s.Close();
  Console.WriteLine(emp); // displays emp.ToString()
  }

  public static void Main(string[] args)
  {
  Employee emp = new Employee("Jack","Clerk",44000);
  WriteEmployee(emp);
  ReadEmployee();
  }
}
```

If we have more than one employee, i.e. an array we extend the program `binsertest1.cs` as follows

```
// serialization.cs

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

class Program
{
  // declare binary formatter to read/write serialized text from file
  static BinaryFormatter bf = new BinaryFormatter();

  static void Main(string[] args)
  {
  // declare array to hold employee objects
  Employee[] employees = new Employee[2];

  // create first and second emplyee object in the array
  employees[0] = new Employee("John Miller","Salesman",15000);
  employees[1] = new Employee("Jack White","Manager",16000);

  // serialize employee list to file
  Serialize(employees);

  // derserialize employee list
  DeSerialize();
```

```
  }

  static void Serialize(Employee[] employees)
  {
  // serialize to file
  Stream stream = File.OpenWrite("employees.bin");
  bf.Serialize(stream,employees);
  stream.Close();
  Console.WriteLine("Employee list has been serialized to file");
  Console.WriteLine();
  }

  static void DeSerialize()
  {
  Console.WriteLine("Reading employee list from file");
  Console.WriteLine();

  // deserialize from file
  Stream stream = File.OpenRead("employees.bin");
  Employee[] emps = (Employee[])bf.Deserialize(stream);
  stream.Close();

  // loop through the array and display each employee
  foreach(Employee e in emps) Console.WriteLine(e);
  Console.WriteLine();
  }
}
```

6.6 XML Documents

The .NET framework provides an XML parser for reading and modifying XML documents. Given below is the file `emp.xml` containing information (`id`,`name`, `job`,`salary`) about employees. There are five employees.

```
<?xml version="1.0"?>
<employees>
  <employee id="101">
    <name> John </name>
    <job> manager </job>
    <salary> 72000 </salary>
  </employee>
  <employee id="102">
    <name> Jane </name>
    <job> steno </job>
    <salary> 23000 </salary>
```

```
  </employee>
  <employee id="103">
    <name> Jim </name>
    <job> salesman </job>
    <salary> 36000 </salary>
  </employee>
  <employee id="104">
    <name> Jill </name>
    <job> clerk </job>
    <salary> 45000 </salary>
  </employee>
  <employee id="105">
    <name>Jeff</name>
    <job>CEO</job>
    <salary>105000</salary>
  </employee>
</employees>
```

The program `listemp.cs` displays `id`, and `salary` of each employee in the file `emp.xml`.

```
// listemp.cs

using System;
using System.Xml;

class ListEmployees
{
  public static void Main()
  {
  XmlDocument doc = new XmlDocument();
  // load employee.xml in XmlDocument
  doc.Load("emp.xml");
  // obtain a list of all employee element
  XmlNodeList list = doc.GetElementsByTagName("employee");
  foreach(XmlNode emp in list)
  {
  // obtain value of id attribute of the employee tag
  string id = emp.Attributes["id"].Value;
  // obtain value of salary subtag of the employee tag
  string sal = emp["salary"].FirstChild.Value;
  Console.WriteLine("{0}\t{1}",id,sal);
  }
  }
}
```

The program `addemp.cs` takes informations from the command line (`id`, `name`, `job`, `salary`) and makes a new entry in the file `emp.xml`.

```
// addemp.cs

using System;
using System.Xml;

class AddEmployee
{
  public static void Main(string[] args)
  {
  XmlDocument doc = new XmlDocument();
  doc.Load("emp.xml");
  // create a new employee element and value of its attribute
  XmlElement emp = doc.CreateElement("employee");
  emp.SetAttribute("id",args[0]);
  // create a name tag and st its value
  XmlNode name = doc.CreateNode("element","name","");
  name.InnerText = args[1];
  // make name tag a subtag of newly created employee tag
  emp.AppendChild(name);
  XmlNode job = doc.CreateNode("element","job","");
  job.InnerText = args[2];
  emp.AppendChild(job);
  XmlNode sal = doc.CreateNode("element","salary","");
  sal.InnerText = args[3];
  emp.AppendChild(sal);
  // add the newly created employee tag to the root (employees) tag
  doc.DocumentElement.AppendChild(emp);
  // save the modified document
  doc.Save("emp.xml");
  }
}
```

We execute the program with (for example)

```
addemp 105 willi president 50000
```

Chapter 7

Graphics

7.1 Drawing Methods

The `Windows.Forms namespace` contains classes for creating Windows based applications. In this namespace we find the `Form` class and many other controls that can be added to forms to create user interfaces. The `System.Drawing` namespace provides access to GDI + basic graphics functionality. The `Graphics` object provides methods for drawing a variety of lines and shapes. Simple or complex shapes can be rendered in solid or transparent colors, or using user-defined gradient or image textures. Lines, open curves, and outline shapes are created using a `Pen` object. To fill in an area, such as a rectangle or a closed curve, a `Brush` object is required. The drawing methods are

```
DrawString(), DrawLine(), DrawRectangle(),
DrawEllipse(), DrawPie(), DrawPolygon(),
DrawArc()
```

and `DrawImage()`. Using `DrawImage()` we can display an image.

The method `DrawString()` takes five arguments

```
DrawString(string,Font,Brush,int X,int Y)
```

The first argument is a `string`, i.e. the text we want to display. The last two arguments is the position where we put the string. The method `DrawEllipse()` is given by

```
DrawEllipse(System.Drawing.Pen,float x,float y,float width,float height)
```

Every method in the `Graphics` class has to be accessed by creating an object of that class. We can easily update the above program to render other graphical shapes like `Rectangle`, `Ellipse`, etc. All we have to do is to apply the relevant methods appropriately.

Using the `Pen` class we can specify colour of the border and also the thickness. The `Pen` class is applied for drawing shapes. The `Brush` is applied for filling shapes such as `SolidBrush` and `HatchStyleBrush`.

The default Graphics unit is Pixel. By applying the `PageUnit` property, we can change the unit of measurement to `Inch` and `Millimeter`.

```
// HelloGraphics.cs

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

public class Hello : Form
{
  public Hello()
  {
  this.Paint +=  new PaintEventHandler(f1_paint);
  }

  private void f1_paint(object sender,PaintEventArgs e)
  {
  Graphics g = e.Graphics;
  g.DrawString("Hello C#",new Font("Verdana",20),new SolidBrush(Color.Tomato),
              40,40);

  g.DrawRectangle(new Pen(Color.Pink,3),20,20,150,100);
  }

  public static void Main()
  {
  Application.Run(new Hello());
  }
}
```

The method

```
DrawPolygon(System.Drawing.Pen,new Point[]{
            new Point(x,y),new Point(x,y),
            new Point(x,y),new Point(x,y),
            new Point(x,y),new Point(x,y)});
```

draws a polygon for a given set of points. The following program shows how do draw two triangles using this method.

```
// MyPolygon.cs

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

class Triangle : Form
{
  public Triangle()
  {
  this.Paint += new PaintEventHandler(draw_triangle);
  }

  public void draw_triangle(Object sender,PaintEventArgs e)
  {
  Graphics g = e.Graphics;
  Pen pen1 = new Pen(Color.Blue,2);
  Pen pen2 = new Pen(Color.Green,2);

  g.DrawPolygon(pen1,new Point[]{new Point(150,50),
                    new Point(50,150),new Point(250,150)});
  g.DrawPolygon(pen2,new Point[]{new Point(50,155),
                    new Point(250,155),new Point(250,255)});
  }

  public static void Main()
  {
  Application.Run(new Triangle());
  }
}
```

The following program shows how to use

```
FillRectangle(Brush,float x,float y,float width,float height);
FillEllipse(Brush,float x,float y, float width,float height);
FillPie(Brush,float x,float y,float width,float height,float angleX,float angleY);
```

```
// Fill.cs

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

public class Fill : Form
{
  public Fill()
```

```
  {
  this.Paint += new PaintEventHandler(fillpaint);
  }

  private void fillpaint(object sender,PaintEventArgs e)
  {
  Graphics g = e.Graphics;
  g.FillRectangle(new SolidBrush(Color.Red),15,15,100,150);
  g.FillEllipse(new SolidBrush(Color.Blue),50,50,150,120);
  g.FillPie(new SolidBrush(Color.Yellow),200,200,40,40,0,90);
  }

  public static void Main()
  {
  Application.Run(new Fill());
  }
}
```

7.2 Color Class

The `Color` class provides "constants" for common colors such as `White`, `Black`, `Blue`, `Red`, `Green`, `Pink`. To create a `Color` structure from the specified 8-bit `Color` (red,green,blue) we call for example

```
Color myColor = Color.FromArgb(0,255,125);
```

The alpha-value is implicit 255 (no transparency). To create a `Color` structure from the four ARGB component (alpha,red,green,blue) we call for example

```
Color myColor = Color.FromArgb(51,255,0,0);
```

Alpha is also known as transparency where 255 is totally solid and 0 is totally transparent.

```
// draw.cs

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

public class Draw : Form
{
  public Draw() { } // default constructor

  protected override void OnPaint(PaintEventArgs e)
```

```
  {
  FontFamily fontFamily = new FontFamily("Times New Roman");
  Font font = new Font(fontFamily,24,FontStyle.Bold,GraphicsUnit.Pixel);
  PointF pointF = new PointF(30,10);
  SolidBrush solidbrush =
   new SolidBrush(Color.FromArgb(51,255,0,0));
  e.Graphics.DrawString ("Hello",font,solidbrush,pointF);

  Pen myPen = new Pen(Color.Red);
  myPen.Width = 50;
  e.Graphics.DrawEllipse(myPen,new Rectangle(33,45,40,50));
  e.Graphics.DrawLine(myPen,1,1,45,65);
  e.Graphics.DrawBezier(myPen,15,15,30,30,45,30,87,20);
  }

  public static void Main()
  {
  Application.Run(new Draw());
  }
}
```

In C# the user can choose a color by applying the `ColorDialog` class appropriatly. First we have to create an object of `ColorDialog` class

```
ColorDialog cd = new ColorDialog();
```

An example is given in the followsing program

```
// ColorD.cs

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

public class ColorD : Form
{
  Button b = new Button();
  TextBox tb = new TextBox();
  ColorDialog clg = new ColorDialog();

  public ColorD()
  {
  b.Click += new EventHandler(b_click);
  b.Text = "OK";
  tb.Location = new Point(50,50);
  this.Controls.Add(b);
```

```
  this.Controls.Add(tb);
  }

  public void b_click(object sender,EventArgs e)
  {
  clg.ShowDialog();
  tb.BackColor = clg.Color;
  }

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

7.3 Button and EventHandler

The `Button` class defines a `Click` event of type `EventHandler`. Inside the `Button` class, the `Click` member is exactly like a private field of type `EventHandler`. However, outside the `Button` class, the `Click` member can only be used on the left-hand side of the `+=` and `-=` operators. The operator `+=` adds a handler for the event, and the `-=` operator removes a handler for the event.

```
// Winhello.cs

using System;
using System.Windows.Forms;
using System.ComponentModel;
using System.Drawing;

class WinHello : Form
{
  private Button bnclick;

  public WinHello()
  {
  Text = "Hello World";
  Size = new Size(400,400);
  bnclick = new Button();
  bnclick.Text = "Click.Me";
  bnclick.Size = new Size(60,24);
  bnclick.Location = new Point(20,60);
  bnclick.Click += new EventHandler(bnclick_Click);
  Controls.Add(bnclick);
  Closing += new CancelEventHandler(WinHello_Closing);
```

```
  }

  private void bnclick_Click(object sender,EventArgs ev)
  {
  MessageBox.Show("Hello Egoli!!!!","Button Clicked",
                  MessageBoxButtons.OK,MessageBoxIcon.Information);
  }

  private void WinHello_Closing(object sender,CancelEventArgs ev)
  {
  if(MessageBox.Show("Are you sure?","Confirm exit",
               MessageBoxButtons.YesNo,MessageBoxIcon.Question)
               == DialogResult.No) ev.Cancel = true;
  }

  // Initialize the main thread
  // using Single Threaded Apartment (STA) model
  public static void Main()
  {
  Application.Run(new WinHello());
  }
}
```

We can create a Font selection dialog box using the `FontDialog` class. The following program gives an example

```
// Fonts.cs

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

public class Fonts : Form
{
  Button b = new Button();
  TextBox tb = new TextBox();
  FontDialog flg = new FontDialog();

  public Fonts()
  {
  b.Click += new EventHandler(b_click);
  b.Text = "OK";
  tb.Location = new Point(50,50);
  this.Controls.Add(b);
  this.Controls.Add(tb);
```

```
  }

  public void b_click(object sender,EventArgs e)
  {
  flg.ShowDialog();
  tb.Font = flg.Font;
  }

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

}
```

7.4 Displaying Images

Next we provide a program that displays images using the method `DrawImage`. The file formats could be `bmp`, `gif` or `jpeg`.

```
// mydrawim.cs

using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

public class Texturedbru : Form
{
  public Texturedbru()
  {
  this.Text = "Using Texture Brushes";
  this.Paint += new PaintEventHandler(Text_bru);
  }

  public void Text_bru(object sender,PaintEventArgs e)
  {
  Graphics g = e.Graphics;
  Image bgimage = new Bitmap("forest.bmp");
  g.DrawImage(bgimage,20,20,1000,600);
  }

  public static void Main()
  {
  Application.Run(new Texturedbru());
```

```
  }
}
```

Another option to display the picture is

```
// textbru.cs

using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

public class Texturedbru : Form
{
  Brush bgbrush;

  public Texturedbru()
  {
  this.Text = "Using Texture Brushes";
  Image bgimage = new Bitmap("forest.bmp");
  bgbrush = new TextureBrush(bgimage);
  this.Paint += new PaintEventHandler(Text_bru);
  }

  public void Text_bru(object sender,PaintEventArgs e )
  {
  Graphics g = e.Graphics;
  g.FillEllipse(bgbrush,50,50,500,300);
  g.FillEllipse(bgbrush,150,150,450,300);
  g.FillRectangle(bgbrush,350,450,100,130);
  }

  public static void Main()
  {
  Application.Run(new Texturedbru());
  }
}
```

7.5 Overriding OnPaint

The next program shows how to override `OnPaint()`. We create a `Button` and clicking on it switches the color of the circle from red to blue and vice versa.

```
// MyOnPaint.cs
```

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

class Draw : Form
{
   private SolidBrush b = new SolidBrush(Color.Red);

   public Draw()
   {
   Button button1 = new Button();
   button1.Text = "Click Me";
   button1.Location = new Point(210,220);
   button1.Click += new EventHandler(HandleClick);
   Controls.Add(button1);
   }

   protected override void OnPaint(PaintEventArgs e)
   {
   int diameter = 200;
   int x = 50;
   int y = 30;

   if(b.Color == Color.Blue)
   {
   b.Color = Color.Red;
   }
   else
   {
   b.Color = Color.Blue;
   }
   e.Graphics.FillEllipse(b,x,y,diameter,diameter);
   } // end OnPaint

   void HandleClick(object sender,EventArgs e)
   {
   this.Refresh();
   }

   public static void Main()
   {
   Application.Run(new Draw());
   }
}
```

Another example we consider a rotation line. We call the `Invalidate()` method to force a repaint at the end of the frame.

```
// Rotate.cs

using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;
using System;

class TimerVector : Form
{
   float rot;
   public static void Main()
   {
   Application.Run(new TimerVector());
   }

   private void TimerEvent(Object myObject,EventArgs myEventArgs)
   {
   Invalidate();
   }

   protected override void OnPaint(PaintEventArgs pea)
   {
   Graphics g = pea.Graphics;
   GraphicsState gs = g.Save();
   Pen myPen = new Pen(Color.Black,3);
   float x1 = ClientSize.Width/4;
   float y1 = ClientSize.Height/4;
   float x2 = (ClientSize.Width*3)/4;
   float y2 = (ClientSize.Height*3)/4;
   float centerx = ClientSize.Width/2;
   float centery = ClientSize.Height/2;
   g.TranslateTransform(-centerx,-centery);
   g.RotateTransform(rot,MatrixOrder.Append);
   g.TranslateTransform(centerx,centery,MatrixOrder.Append);
   g.SmoothingMode = SmoothingMode.AntiAlias;
   g.DrawLine(myPen,x1,y1,x2,y2);
   g.Restore(gs);
   rot = (rot + 1)%360;
   }

   public TimerVector()
   {
```

```
   rot = 0.0f;
   Timer t = new Timer();
   t.Interval = 100;
   t.Tick += new EventHandler(TimerEvent);
   t.Start();
   }
}
```

Another example is

```
// Flicker.cs

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

class Flicker : Form
{
  private Timer t = new Timer();
  private Size playerSize = new Size(50,50);
  private Point playerPosition = new Point(0,0);

  public Flicker()
  {
  ClientSize = new Size(500,500);
  SetStyle(ControlStyles.DoubleBuffer,true);
  SetStyle(ControlStyles.UserPaint,true);
  SetStyle(ControlStyles.AllPaintingInWmPaint,true);
  t.Interval = 40;
  t.Tick += new EventHandler(TimerOnTick);
  t.Enabled = true;
  this.KeyDown += new KeyEventHandler(OnKeyPress);
  }

  private void TimerOnTick(object sender,EventArgs e)
  {
  this.Refresh();
  this.Text = DateTime.Now.ToString();
  this.Text += " " + this.PlayerPosition.ToString();
  }

  private Point PlayerPosition
  {
  get { return this.playerPosition; }
```

```
set
{
if(value.X < 0)
{
this.playerPosition.X = this.ClientSize.Width-this.playerSize.Width;
}
else if(value.X+this.playerSize.Width > this.ClientSize.Width)
{
this.playerPosition.X = 0;
}
else
{
this.playerPosition.X = value.X;
}

if(value.Y < 0)
{
this.playerPosition.Y = this.ClientSize.Height-this.playerSize.Height;
}
else if(value.Y+this.playerSize.Height > this.ClientSize.Height)
{
this.playerPosition.Y = 0;
}
else
{
this.playerPosition.Y = value.Y;
}
}
}

private void OnKeyPress(object sender,KeyEventArgs e)
{
if(e.KeyValue == 37)
{
this.PlayerPosition =
 new Point(this.PlayerPosition.X-this.playerSize.Width,this.PlayerPosition.Y);
}
if(e.KeyValue == 38)
{
this.PlayerPosition =
 new Point(this.PlayerPosition.X,this.PlayerPosition.Y-this.playerSize.Width);
}
if(e.KeyValue == 39)
{
this.PlayerPosition =
```

```
   new Point(this.PlayerPosition.X+this.playerSize.Height,this.PlayerPosition.Y);
  }

  if(e.KeyValue == 40)
  {
  this.PlayerPosition =
   new Point(this.PlayerPosition.X,this.PlayerPosition.Y+this.playerSize.Height);
  }
  }

  protected override void OnPaint(PaintEventArgs e)
  {
  e.Graphics.FillRectangle(new SolidBrush(Color.Red),this.PlayerPosition.X,
                           this.playerPosition.Y,this.playerSize.Width,
                           this.playerSize.Height);
  }

  public static void Main()
  {
  Application.Run(new Flicker());
  }
}
```

Chapter 8

Events

Event handling is familiar to any developer who has programmed graphical user interfaces (GUI). When a user interacts with a GUI control (e.g., clicking a button on a form), one or more methods are executed in response to the above event. Events can also be generated without user interactions. Event handlers are methods in an object that are executed in response to some events occuring in the application. To understand the event handling model of the .NET framework, we need to unterstand the concept of delegate. A delegate in C# allows us to pass methods of one class to objects to other classes that can call those methods. We can pass method `m` in `class A`, wrapped in a delegate, to `class B` and `class B` will be able to call method `m`. We can pass both static and instance methods.

An event handler in C# is a delegate with a special signature, given below

```
public delegate void MyEventHandler(object sender,MyEventArgs e)
```

The first parameter (sender) in the above declaration specifies the object that fired the event. The second parameter (e) holds data that can be used in the event handler. The class `MyEventArgs` is derived from the class `EventArgs`. `EventArgs` is the base class of more specialized classes, like `MouseEventArgs`, `ListChangedEventArgs`, etc.

Thus an *event* is a notification sent by a sender object to a listener object. The listener registers an event handler method with the sender. When the event occurs the sender invokes event handler methods of all its registered listerners. The event handler is registered by adding `+=`. The `-=` operator removes the handler from the event.

As an example we set up two classes `A` and `B` to see how this event handling mechanism works in .NET framework. The delegates require that we define methods with the exact same signature as that of the delegate declaration. Class `A` will provide event handlers (methods with the same signature as that of the event declaration). It will create the delegate objects and hook up the event handler. Class `A` will then pass the delegate objects to class `B`. When an event occurs in class `B`, it will execute the event handler method of class `B`.

```
// MyHandler.cs

using System;

// Step 1. Create delegate object
public delegate void MyHandler1(object sender,MyEventArgs e);
public delegate void MyHandler2(object sender,MyEventArgs e);

// Step 2. Create event handler methods
class A
{
  public const string m_id = "Class A";
  public void OnHandler1(object sender,MyEventArgs e)
  {
  Console.WriteLine("I am in OnHandler1 and MyEventArgs is {0}",e.m_id);
  }

  public void OnHandler2(object sender,MyEventArgs e)
  {
  Console.WriteLine("I am in OnHandler2 and MyEventArgs is {0}",e.m_id);
  }

  // Step 3. Create delegates, plug in the handler and register
  // with the object that will fire the events
  public A(B b)
  {
  MyHandler1 d1 = new MyHandler1(OnHandler1);
  MyHandler2 d2 = new MyHandler2(OnHandler2);
  b.Event1 += d1;
  b.Event2 += d2;
  }
} // end class A

// Step 4. Call the encapsulated method through the delegate (fires events)
class B
{
  public event MyHandler1 Event1;
  public event MyHandler2 Event2;
  public void FireEvent1(MyEventArgs e)
  {
  if(Event1 != null) { Event1(this,e); }
  }
  public void FireEvent2(MyEventArgs e)
  {
  if(Event2 != null) { Event2(this,e); }
```

```
  }
} // end class B

public class MyEventArgs
{
  public string m_id;
} // end class MyEventArgs

class Driver
{
  public static void Main()
  {
  B b = new B();
  A a = new A(b);
  MyEventArgs e1 = new MyEventArgs();
  MyEventArgs e2 = new MyEventArgs();
  e1.m_id = "Event args for event 1";
  e2.m_id = "Event args for event 2";
  b.FireEvent1(e1);
  b.FireEvent2(e2);
  } // end Main
}
```

The next program shows GUI event handling in C#. We create two buttons each of them fires an event.

```
// evthand.cs

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

class MyForm : Form
{
  MyForm()
  {
  // button1 top left corner
  Button button1 = new Button();
  button1.Text = "Button";
  button1.Click += new EventHandler(HandleClick1);
  Controls.Add(button1);

  Button button2 = new Button();
  button2.Location = new Point(40,40);
  button2.Size = new Size(60,40);
```

```
  button2.Text = "New Button";
  button2.Click += new EventHandler(HandleClick2);
  Controls.Add(button2);
  }

  void HandleClick1(object sender,EventArgs e)
  {
  MessageBox.Show("The click event fire!");
  }

  void HandleClick2(object sender,EventArgs e)
  {
  Console.WriteLine("Console click fire!");
  }

  public static void Main()
  {
  Application.Run(new MyForm());
  }
}
```

Chapter 9

Processes and Threads

9.1 Processes

A *process* in its simplest form is a running application. The `Process` class provides access to local and remote processes and enables us to start and stop local system processes.

The `Process.Start()` method starts a process resource by specifying the name of a document or application file and associates the resource with a new process component. Thus we can start other applications from within our application.

The program below launches `notepad.exe`, `winhlp32.exe` and asynchronously counts up from 0 to 100001, then destroys the `notepad.exe` process using the `Kill()` method. It counts again from 0 to 2000 and then kills the `winhlp32.exe` process. Each process has an `Id` number.

```
// process1.cs

using System;
using System.Diagnostics;

class Processtest
{
  public static void Main()
  {
  Process p = Process.Start("notepad.exe");
  string name = p.ProcessName;
  int id = p.Id;
  DateTime started = p.StartTime;
  Console.WriteLine("name = " + name);
  Console.WriteLine("id = " + id);
  Console.WriteLine("StartTime = " + started);
```

```
  Process w = Process.Start("winhlp32.exe");

  for(int i=0;i<100001;i++) Console.WriteLine(i);
  p.Kill();
  for(int j=0;j<2000;j++) Console.WriteLine(j);
  w.Kill();
  }
}
```

If we also want to load a file (for example `process2.cs`) with `notepad.exe` we change this line to

```
Process p = Process.Start("notepad.exe","c:\\csharp\\process2.cs");
```

The detect process completion we use the method

```
WaitForExit()
```

9.2 Threads

9.2.1 Introduction

A *thread* is an execution stream within a process. A thread is also called a lightweight process. It has it own execution stack, local variables, and program counter. There may be more than one thread in a process. Threads are part of the same process which execute concurrently. In .NET Base Class Library (BCL) the `System.Threading` namespace provides various classes for executing and controlling threads.

The following program `Threads0.cs` is the simple example. We create a `Thread` instance by passing it an object of `ThreadStart` delegate which contains a reference to our `ThreadJob` method. Thus the code creates a new thread which runs the `ThreadJob` method, and starts it. That thread counts from 0 to 1 while the main thread counts from 0 to 5.

```
// Threads0.cs

using System;
using System.Threading;

public class Threads0
{
  static void Main()
  {
  ThreadStart job = new ThreadStart(ThreadJob);
  Thread thread = new Thread(job);
```

```
  thread.Start();

  int i = 0;
  while(i < 6)
  {
  Console.WriteLine("Main thread: {0}",i);
  i++;
  } // end while
  } // end Main

  static void ThreadJob()
  {
  int i = 0;
  while(i < 2)
  {
  Console.WriteLine("Other thread: {0}",i);
  i++;
  } // end while
  } // end ThreadJob
}
```

An output could be

```
Main thread:  0
Main thread:  1
Main thread:  2
Main thread:  3
Main thread:  4
Main thread:  5
Other thread: 0
Other thread: 1
```

At another run the output could be

```
Other thread: 0
Other thread: 1
Main thread:  0
Main thread:  1
Main thread:  2
Main thread:  3
Main thread:  4
Main thread:  5
```

9.2.2 Background Thread

In the program `threadtest2.cs` the `Thread t` is set to the background thread

```
t.IsBackgound = true;
```

The thread will terminate when the application terminates.

```
// threadtest2.cs

using System;
using System.Threading; // for Thread class

class ThreadTest2
{
   public static void SayHello()
   {
   for(int i=1;;i++)
   {
   Console.WriteLine("Hello {0}",i);
   }
   }

   public static void Main()
   {
   Thread t = new Thread(new ThreadStart(SayHello));
   t.IsBackground = true;
   t.Start();
   for(int i=1;i<1001;i++)
   {
   Console.WriteLine("Bye {0}",i);
   }
   }
}
```

9.2.3 Sleep Method

Once a thread has been started it is often useful for that thread to pause for a fixed period of time. Calling `Thread.Sleep` causes the thread to immediately block for a fixed number of milliseconds. The `Sleep` method takes as a parameter a timeout, which is the number of milliseconds that the thread should remain blocked. The `Sleep` method is called when a thread wants to put itself to sleep. One thread cannot call `Sleep` on another thread. Calling `Thread.Sleep(0)` causes a thread to yield the remainder of its timeslice to another thread. Calling `Thread.Sleep(Timeout.Infinite)` causes a thread to sleep until it is interrupted by another thread that calls `Thread.Interrupt`. A thread can also be paused by calling `Thread.Suspend`. When a thread calls `Thread.Suspend` on itself, the call blocks until the thread is resumed by another thread. When one thread calls `Thread.Suspend` on another thread, the call is a non-blocking call that causes the

other thread to pause.

The following program `Threads0.cs` is the simple example. We create a `Thread` instance by passing it an object of `ThreadStart` delegate which contains a reference to our `ThreadJob` method. Thus the code creates a new thread which runs the `ThreadJob` method, and starts it. That thread counts from 0 to 9 fairly fast (about twice a second) while the main thread counts from 0 to 4 fairly slowly (about once a second). The way they count at different speeds is by each of them including a call to `Thread.Sleep()`, which just makes the current thread sleep (do nothing) for the specified period of time (milliseconds). Between each count in the main thread we sleep for 1000ms, and between each count in the other thread we sleep for 500ms.

```
// ThreadsSleep.cs

using System;
using System.Threading;

public class ThreadsSleep
{
  static void Main()
  {
  ThreadStart job = new ThreadStart(ThreadJob);
  Thread thread = new Thread(job);
  thread.Start();

  int i = 0;
  while(i < 5)
  {
  Console.WriteLine("Main thread: {0}",i);
  Thread.Sleep(1000);
  i++;
  } // end while
  } // end Main

  static void ThreadJob()
  {
  int i = 0;
  while(i < 10)
  {
  Console.WriteLine("Other thread: {0}",i);
  Thread.Sleep(500);
  i++;
  } // end while
  } // end ThreadJob
}
```

A typical output could be

```
Main thread:  0
Other thread: 0
Other thread: 1
Main thread:  1
Other thread: 2
Other thread: 3
Other thread: 4
Main thread:  2
Other thread: 5
Main thread:  3
Other thread: 6
Other thread: 7
Main thread:  4
Other thread: 8
Other thread: 9
```

At another run this could change.

9.2.4 Join Methods

The method `Join()` blocks the calling thread until a thread terminates.

```
// joiningthread.cs

using System;
using System.Threading;

class JoiningThread
{
  public static void Run()
  {
  for(int i=1;i<3;i++)
  Console.WriteLine("Hello {0}",i);
  }

  public static void Main()
  {
  Thread t = new Thread(new ThreadStart(Run));
  t.Start();
  for(int i=1;i<6;i++)
  Console.WriteLine("Welcome {0}",i);
  t.Join();
  Console.WriteLine("Goodbye");
  }
```

```
}
```

The output is

```
Welcome 1
Welcome 2
Welcome 3
Welcome 4
Welcome 5
Hello 1
Hello 2
```

9.3 Monitor

While one thread is in a read/increment/write operation, no other threads can try to do the same thing. This is where monitors come in. Every object in .NET has a monitor associated with it. A thread can enter (or acquire) a monitor only if no other thread has currently "got" it. Once a thread has acquired a monitor, it can acquire it more times, or exit (or release) it. The monitor is only available to other threads again once it has exited as many times as it was entered. If a thread tries to acquire a monitor which is owened by another thread, it will block until it is able to acquire it. There may be more than one thread trying to acquire the monitor, in which case when the current owner thread releases it for the last time, only one of the threads will acquire it - the other one will have to wait for the new owner to release it too. The `Pulse(object)` method notifies a thread in the waiting queue of a change in the locked object's state. The method `Wait(object)` waits for the Monitor Pulse.

```
// waitingthread.cs

using System;
using System.Threading;

class WaitingThread
{
  static object obj = new object();

  public static void thread1()
  {
  for(int i=1;i<6;i++)
  Console.WriteLine("Welcome: {0}",i);
  Monitor.Enter(obj);
  Monitor.Pulse(obj);
  Monitor.Exit(obj);
  }
```

```
  public static void thread2()
  {
  for(int i=1;i<15;i++)
  Console.WriteLine("Hello: {0}",i);
  Monitor.Enter(obj);
  Monitor.Pulse(obj);
  Monitor.Exit(obj);
  }

  public static void thread3()
  {
  for(int i=1;i<8;i++)
  Console.WriteLine("Good Night: {0}",i);
  Monitor.Enter(obj);
  Monitor.Wait(obj);
  Monitor.Pulse(obj);
  Monitor.Exit(obj);
  }

  public static void Main()
  {
  ThreadStart job1 = new ThreadStart(thread1);
  ThreadStart job2 = new ThreadStart(thread2);
  ThreadStart job3 = new ThreadStart(thread3);
  Thread t1 = new Thread(job1);
  Thread t2 = new Thread(job2);
  Thread t3 = new Thread(job3);
  t1.Start();
  t2.Start();
  t3.Start();
  }
}
```

If `thread3` runs last after thread1 and thread2 have completed it freeze at

```
Good Night: 7
```

9.4 Synchronization

When two or more threads share a common resource access needs to be serialized in a process called synchronization. Synchronization is done with the `lock()` operation. The first program is without the `lock` operation. In the second program the `lock` operation is introduced.

```
// syncthreads1.cs

using System;
using System.Threading;

class Account
{
  private double balance = 5000;

  public void Withdraw(double amount)
  {
  Console.WriteLine("WITHDRAWING {0}",amount);
  if(amount > balance) throw new Exception("INSUFFICIENT FUNDS");
  Thread.Sleep(10); // do other stuff
  balance -= amount;
  Console.WriteLine("BALANCE {0}",balance);
  }
}

class SymnThread
{
  static Account acc = new Account();

  public static void Run()
  {
  acc.Withdraw(3000);
  }

  public static void Main()
  {
  new Thread(new ThreadStart(Run)).Start();
  acc.Withdraw(3000);
  }
}
```

Now we introduce the lock.

```
// syncthreads2.cs

using System;
using System.Threading;

class Account
{
  private double balance = 5000;
```

```
  public void Withdraw(double amount)
  {
  Console.WriteLine("WITHDRAWING {0}",amount);
  lock(this)
  {
  if(amount > balance) throw new Exception("INSUFFICIENT FUNDS");
  Thread.Sleep(10); // do other stuff
  balance -= amount;
  }
  Console.WriteLine("BALANCE {0}",balance);
  }
}

  class SymnThread
  {
  static Account acc = new Account();

  public static void Run()
  {
  acc.Withdraw(3000);
  }

  public static void Main()
  {
  new Thread(new ThreadStart(Run)).Start();
  acc.Withdraw(3000);
  }
}
```

9.5 Deadlock

The second major problem of multi-threading is that of deadlocks. Simply put, this is when two threads each holds a monitor that the other one wants. Each blocks, waiting for the monitor that it's waiting for to be released - and so the monitors are never released, and the application hangs (or at least those threads involved in the deadlock hang). An example is given below.

```
// Deadlock.cs

using System;
using System.Threading;

public class Deadlock
{
```

```
  static readonly object firstLock = new object();
  static readonly object secondLock = new object();

  static void ThreadJob()
  {
    Console.WriteLine("\t\t\t\tLocking firstLock");
    lock(firstLock)
    {
    Console.WriteLine("\t\t\t\tLocked firstLock");
    // Wait until we are fairly sure the first thread
    // has grabbed secondlock
    Thread.Sleep(1000);
    Console.WriteLine("\t\t\t\tLocking secondLock");
    lock(secondLock)
    {
    Console.WriteLine("\t\t\t\tLocked secondLock");
    }
    Console.WriteLine("\t\t\t\tReleased secondLock");
    }
    Console.WriteLine("\t\t\t\tReleased firstLock");
  } // end method ThreadJob

  public static void Main()
  {
    new Thread(new ThreadStart(ThreadJob)).Start();
    // wait until we are fairly sure the other thread
    // has grabbed firstlock
    Thread.Sleep(500);
    Console.WriteLine("Locking secondLock");
    lock(secondLock)
    {
    Console.WriteLine("Locked secondLock");
    Console.WriteLine("Locking firstLock");
    lock(firstLock)
    {
    Console.WriteLine("Locked firstLock");
    }
    Console.WriteLine("Released firstLock");
    }
    Console.WriteLine("Released secondLock");
  } // end Main
}   // end class
```

9.6 Interlocked Class

An operation is *atomic* if it is indivisible - in other words, nothing else can happen in the middle. Thus with an atomic write, we cannot have another thread reading the value half way through the write, and ending up "seeing" half of the old value and half of the new value. Sinilarly, with an atomic read, we cannot have another thread changing the value half way through the read, ending up with a value which is neither the old nor the new value. For example, for a `long` (64 bits) on a 32 bit machine, if one thread is changing the value from 0 to 0x0123456789ABCDEF, there is no guarantee that another thread will not see the value as 0x0123456700000000 or 0x0000000089ABCDEF.

The `Interlocked` class provides a set of methods for performing atomic changes: exchanges (optionally performing a comparison first), increments and decrements. The `Exchange` and `CompareExchange` methods act on a variables of type `int`, `object`, or float; the `Increment` and `Decrement` methods act on variables of type `int` and *long*.

```
// interlocked.cs

using System;
using System.Threading;

public class MyInterlocked
{
  static long count = 0;

  public static void Main()
  {
  ThreadStart job = new ThreadStart(ThreadJob);
  Thread thread = new Thread(job);
  thread.Start();

  for(long i=0;i<5;i++)
  {
  Interlocked.Increment(ref count);
  Console.WriteLine("I am in for loop in main");
  }
  thread.Join();
  Console.WriteLine("final count: {0}",count);
  } // end Main

  static void ThreadJob()
  {
  long i = 0;
```

```
  while(i < 5)
  {
  Interlocked.Increment(ref count);
  Console.WriteLine("I am in ThreadJob");
  i++;
  }
  } // end ThreadJob
}
```

First the `for` loop in `Main` will run to the end and then the `while` loop will be done.

9.7 Thread Pooling

We can use thread pooling to make much more efficient use of multiple threads, depending on our application. Many applications use multiple threads, but often those threads spend a great deal of time in the sleeping state waiting for an event to occur. Other threads might enter a sleeping state and be awakend only periodically to poll for a change or update status information before going to sleep again. Using thread pooling provides our application with a pool of *worker threads* that are managed by the system, allowing us to concentrate on application tasks rather than thread management. An example is given below.

```
// ThreadsSum2.cs

using System;
using System.Threading;

class ThreadsSum
{
  public static void Sum1(Object o)
  {
  int sum1 = 0;
  int[] a1 = (int[]) o;
  for(int i=0;i<a1.Length;i++)
  {
  sum1 += a1[i];
  }
  Console.WriteLine("sum1 = " + sum1);
  }

  public static void Sum2(Object o)
  {
  int sum2 = 0;
  int[] a2 = (int[]) o;
  for(int i=0;i<a2.Length;i++)
```

```
  {
  sum2 +=  a2[i];
  }
  Console.WriteLine("sum2 = " + sum2);
  }

  public static void Main()
  {
  int[] a1 = { 2, 3, 4 };
  int[] a2 = { 4, 5, 7 };

  if(ThreadPool.QueueUserWorkItem(new WaitCallback(Sum1),a1))
   Console.WriteLine("Sum1 queued");
  if(ThreadPool.QueueUserWorkItem(new WaitCallback(Sum2),a2))
   Console.WriteLine("Sum2 queued");

  Thread.Sleep(10); // Give other threads a turn
  }
}
```

9.8 Threading in Windows Forms

How to handle threading in a UI? There are two rules for Windows Forms:

1) Never invoke any method or property on a control created on another thread other than `Invoke`, `BeginInvoke`, `EndInvoke` or `CreateGraphics`, and `InvokeRequired`. Each control is effectively bound to a thread which runs its message pump. If we try to access or change anything in the UI (for example changing the Text property) from a different thread, we run a risk of our program hanging or misbehaving in other ways. We may get away with it in some cases. Fortunately, the `Invoke`, `BeginInvoke` and `EndInvoke` methods have been provided so that we can ask the UI thread to call a method in a safe manner.

2) Never execute a long-running piece of code in the UI thread. If our code is running in the UI thread, that means no other code is running in that thread. That means we won't receive events, our controls won't be repainted, etc. We can execute long-running code and periodically call `Application.DoEvents()`. It means we have to consider re-entrancy issues etc, which are harder to diagnose and fix than "normal" threading problems. We have to judge when to call `DoEvents`, and we can't use anything which might block (network access, for instance) without risking an unresponsive UI. There are message pumping issues in terms of COM objects as well.

If we have a piece of long-running code which we need to execute, we need to create a new thread (or use a thread pool thread if we prefer) to execute it on, and make

sure it doesn't directly try to update the UI with its results. The thread creation part is the same as any other threading problem. It is interesting going the other way - invoking a method on the UI thread in order to update the UI. There are two different ways of invoking a method on the UI thread, one synchronous (`Invoke`) and one asynchronous (`BeginInvoke`). They work in much the same way - we specify a delegate and (optionally) some arguments, and a message goes on the queue for the UI thread to process. If we use `Invoke`, the current thread will block until the delegate has been executed. If we use `BeginInvoke`, the call will return immediately. If we need to get the return value of a delegate invoked asynchronously, we can use `EndInvoke` with the `IAsyncResult` returned by `BeginInvoke` to wait until the delegate has completed and fetch the return value.

There are two options when working out how to get information between the various threads involved. The first option is to have state in the class itself, setting it in one thread, retrieving and processing it in the other (updating the display in the UI thread, for example). The second option is to pass the information as parameters in the delegate. Using state somewhere is necessary if we are creating a new thread rather than using the thread pool - but that doesn't mean we have to use state to return information to the UI. However, creating a delegate with lots of parameters feels clumsy, and is in some ways less efficient than using a simple `MethodInvoker` or `EventHandler` delegate. These two delegates are treated in a special (fast) manner by `Invoke` and `BeginInvoke`. `MethodInvoker` is just a delegate which takes no parameters and returns no value (like `ThreadStart`), and `EventHandler` takes two parameters (a sender and an EventArgs parameter and returns no value. However if we pass an `EventHandler` delegate to `Invoke` or `BeginInvoke` then even if we specify parameters ourself, they are ignored - when the method is invoked, the sender will be the control we have invoked it with, and the `EventArgs` will be `EventArgs.Empty`.

Here is an example which shows several of the above concepts.

```
// ThreadingForms.cs

using System;
using System.Threading;
using System.Windows.Forms;
using System.Drawing;

public class Test : Form
{
  delegate void StringParameterDelegate(string value);
  Label statusIndicator;
  Label counter;
  Button button;

  readonly object stateLock = new object();
```

```
int target;
int currentCount;

Random rng = new Random();

Test()
{
Size = new Size(180,120);
Text = "Test";

Label lbl = new Label();
lbl.Text = "Status:";
lbl.Size = new Size(50,20);
lbl.Location = new Point(10,10);
Controls.Add(lbl);

lbl = new Label();
lbl.Text = "Count:";
lbl.Size = new Size(50,20);
lbl.Location = new Point(10,34);
Controls.Add(lbl);

statusIndicator = new Label();
statusIndicator.Size = new Size(100,20);
statusIndicator.Location = new Point(70,10);
Controls.Add(statusIndicator);

counter = new Label();
counter.Size = new Size(100,20);
counter.Location = new Point(70,34);
Controls.Add(counter);

button = new Button();
button.Text = "Run";
button.Size = new Size(50,20);
button.Location = new Point(10,58);
Controls.Add(button);
button.Click += new EventHandler(StartThread);
}

void StartThread(object sender,EventArgs e)
{
button.Enabled = false;
lock(stateLock)
{
```

```
target = rng.Next(100);
}
Thread t = new Thread(new ThreadStart(ThreadJob));
t.IsBackground = true;
t.Start();
}

void ThreadJob()
{
MethodInvoker updateCounterDelegate = new MethodInvoker(UpdateCount);
int localTarget;
lock(stateLock)
{
localTarget = target;
}
UpdateStatus("Starting");

lock(stateLock)
{
currentCount = 0;
}
Invoke(updateCounterDelegate);
// Pause before starting
Thread.Sleep(500);
UpdateStatus("Counting");
for(int i=0;i<localTarget;i++)
{
lock(stateLock)
{
currentCount = i;
}
// Synchronously show the counter
Invoke(updateCounterDelegate);
Thread.Sleep(100);
}
UpdateStatus("Finished");
Invoke(new MethodInvoker(EnableButton));
}

void UpdateStatus(string value)
{
if(InvokeRequired)
{
// We're not in the UI thread, so we need to call BeginInvoke
BeginInvoke(new StringParameterDelegate(UpdateStatus),new object[]{value});
```

```
  return;
  }
  // Must be on the UI thread if we've got this far
  statusIndicator.Text = value;
  }

  void UpdateCount()
  {
  int tmpCount;
  lock(stateLock)
  {
  tmpCount = currentCount;
  }
  counter.Text = tmpCount.ToString();
  }

  void EnableButton()
  {
  button.Enabled = true;
  }

  static void Main()
  {
  Application.Run(new Test());
  }
}
```

State is used to tell the worker thread what number to count up to. A delegate taking a parameter is used to ask the UI to update the status label. The worker thread's principal method actually just calls `UpdateStatus`, which uses `InvokeRequired` to detect whether or not it needs to "change thread". If it does, it then calls `BeginInvoke` to execute the same method again from the UI thread. This is quite a common way of making a method which interacts with the UI thread-safe. The choice of `BeginInvoke` rather than `Invoke` here was just to demonstrate how to invoke a method asynchronously. In real code, we would decide based on whether we needed to block to wait for the access to the UI to complete before continuing or not. It is quite rare to actually require UI access to complete first, so we should use `BeginInvoke` instead of `Invoke`. Another approach might be to have a property which did the appropriate invoking when necessary. It is easier to use from the client code, but slightly harder work in that we would either have to have another method anyway, or get the `MethodInfo` for the property setter in order to construct the delegate to invoke. In this case we actually know that `BeginInvoke` is required because we are running in the worker thread anyway. We do not call `EndInvoke` after the `BeginInvoke`. Unlike all other asynchronous methods we do not need to call `EndInvoke` unless we need the return value of the delegate's method. `BeginInvoke`

is also different to all of the other asynchronous methods as it doesn't cause the delegate to be run on a thread pool thread. State is used again to tell the UI thread how far we have counted so far. We use a `MethodInvoker` delegate to execute `UpdateCount`. We call this using `Invoke` to make sure that it executes on the UI thread. This time there's no attempt to detect whether or not an `Invoke` is required. If we call `BeginInvoke` it will have a different effect than calling the method directly as it will occur later, rather than in the current execution flow, of course. Again, we actually know that we need to call Invoke here anyway. A button is provided to let the user start the thread. It is disabled while the thread is running, and another `MethodInvoker` delegate is used to enable the button again afterwards. All state which is shared between threads (the current count and the target) is accessed in locks in the way described earlier. We spend as little time as possible in the lock, not updating the UI or anything else while holding the lock. This probably doesn't make too much difference here. It would be disastrous to still have the lock in the worker thread when synchronously invoking `UpdateCount` - the UI thread would then try to acquire the lock as well, and we end up with deadlock. The worker thread is set to be a background thread (`IsBackground=true;`) so that when the UI thread exits, the whole application finishes. In other cases where we have a thread which should keep running even after the UI thread has quit, we need to be careful not to call `Invoke` or `BeginInvoke` when the UI thread is no longer running - we will either block permanently (waiting for the message to be taken off the queue, with nothing actually looking at messages) or receive an exception.

9.9 Asynchronous Programming Model

When a caller invokes a method, the call is synchronous that is the caller has to wait for the method to return before the remaining code can be executed. .NET has an inbuilt support for asynchronous method invocation. Here the caller can issue a request for invocation of a method and concurreently execute the remaining code. For every delegate declared in an assembly the compiler emits a class (subclass of `System.MulticastDelegate`) with `Invoke`, `BeginInvoke` and `EndInvoke` methods. For example, consider a delegate declared as

```
delegate int MyWorker(char c,int m);
```

The compiler will emit the `MyWorker` class

```
class MyWorker : System.MulticastDelegate
{
  public int Invoke(char c,int m);
  public System.IAsyncResult BeginInvoke(char c,int m,System.AsyncCallback cb,
                                         object asyncState);
  public int EndInvoke(System.IAsyncResult result);
}
```

The `BeginInvoke` and `EndInvoke` methods can be used for asynchronous invocation of a method pointed by `MyWorker` delegate. In the program below the method `DoWork` (it displays character `c` `m` number of times) is invoked asynchronously with character `'+'`, in the next statement this method is directly invoked with character `'*'`. Both `'+'` and `'*'` are displayed (1000 times each) on the Console simultaneously. As an asynchronous call executes within its own background thread, we have used `Console.Read()` to pause the main thread.

```
// Asynchronous.cs

using System;

delegate int MyWorker(char c,int m);

class AsncTest
{
   static MyWorker worker;

   static int DoWork(char c,int m)
   {
   int t = Environment.TickCount; // returns number of milliseconds
                                  // elapsed since the system started
   for(int i=1;i<=m;i++) Console.Write(c);
   return (Environment.TickCount - t);
   }

   public static void Main()
   {
   Console.WriteLine("Start");
   worker = new MyWorker(DoWork);
   worker.BeginInvoke('+',1000,null,null); // asynchronous call
   DoWork('*',1000); // synchronous call
   Console.Read(); // pause until user presses a key
   }
}
```

9.10 Timers

There are various different timers available in .NET, each of which basically calls a delegate after a certain amount of time has passed. All the timers implement `IDispossible`. so we have to make sure to dispose when we are not using them anymore.

```
// Timers.cs
```

```
using System;
using System.Threading;

public class Timers
{
  public static void Main()
  {
  Console.WriteLine("Started at {0:HH:mm:ss.fff}",DateTime.Now);
  // Start in three seconds, then fire every second
  using(Timer timer = new Timer(new TimerCallback(Tick),null,3000,1000))
  {
  // wait for 10 seconds
  Thread.Sleep(10000);
  // then go slow for another 10 seconds
  timer.Change(0,2000);
  Thread.Sleep(10000);
  }
  // the timerm will now have been disposed automatically due
  // to the using statement so there won't be any other threads running
  } // end Main()

  static void Tick(object state)
  {
  Console.WriteLine("Ticked at {0:HH:mm:ss.fff}",DateTime.Now);
  }
}
```

9.11 Interrupt and Abort

There are two methods in the `Thread` class which are used for stopping threads - `Abort` and `Interrupt`. Calling `Thread.Abort` aborts that thread as soon as possible. Aborting a thread which is executing unmanaged code has no effect until the CLR gets control again. Calling `Thread.Interrupt` is similar, but less drastic. This causes a `ThreadInterruptedException` exception to be thrown the next time the thread enters the `WaitSleepJoin` state, or immediately if the thread is already in that state.

```
// Interruptthread.cs

using System;
using System.Threading;

class SleepingThread
{
  public static void Run()
```

```
  {
  for(int i=1;i<6;i++)
  Console.WriteLine("Welcome {0}",i);

  try
  {
  Thread.Sleep(5000);
  }
  catch(ThreadInterruptedException e)
  {
  Console.WriteLine("Sleep Interrupted");
  }
  Console.WriteLine("Goodbye");
  }

  public static void Main()
  {
  Thread t = new Thread(new ThreadStart(Run));
  t.Start();
  for(int i=1;i<16;i++)
  Console.WriteLine("Hello {0}",i);
  t.Interrupt();
  }
}
```

The program `threadtest1.cs` uses the `Abort` method.

```
// threadtest1.cs

using System;
using System.Threading; // for Thread class

class ThreadTest1
{
   public static void SayHello()
   {
   for(int i=1;;i++)
   {
   Console.WriteLine("Hello {0}",i);
   }
   }

   public static void Main()
   {
   Thread t = new Thread(new ThreadStart(SayHello));
```

```
   t.Start();
   for(int i=1;i<1001;i++)
   {
   Console.WriteLine("Bye {0}",i);
   }
   t.Abort();
   }
}
```

Chapter 10

Sockets Programming

10.1 Introduction

Most interprocess communication uses the *client server model*. These terms refer to the two processes which will communicating with each other. One of the processes, the client, connects to the other process, the server, typically to make a request for information. A good analogy is a person who makes a phone call to another person.

The client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps for establishing a socket on the client side are:

1. Create a socket
2. Connect the socket to the address of the server
3. Send and receive data

The steps for establishing a socket on the server side are:

1. Create a socket
2. Bind the socket to an address. For the server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections.
4. Accept a connection. This call typically blocks until a client connects with the server.

5. Send and receive data.

Thus a socket is a communication mechanism. A socket is normally identified by an integer which may be called the socket descriptor. The socket mechanism was first introduced in the 4.2 BSD Unix system in 1983 in conjunction with the TCP/IP protocols that first appeared in the 4.1 BSD Unix system in late 1981.

Thus besides the IPAddress we also need a port number (2 bytes) which is arbitrary except for the well know port numbers associated with popular applications.

Formally a socket is defined by a group of four numbers. There are

1) The remote host identification number or address (IPAddress)
2) The remote host port number
3) The local host identification number or address (IPAddress)
4) The local host port number

10.2 Transmission Control Protocol

TCP is reliable connection oriented protocol.

Example 1. When the server program is run it will indicate at which IPAddress it is running and the port it is listening to. Now run the client program so that we establish a connection with the server. When the connection is established the server will display the IPAddress and port from where it has accepted the connection. The client will ask for the string which is to be transmitted to the server. The server on reciept of the string will display it, send an acknowledgement which will be received by the client.

```
// Server1.cs

using System;
using System.Text;
using System.Net;
using System.Net.Sockets;

class Server1
{
  public static void Main()
  {
  try
  {
  // use local IP address and use the same in the client
  IPAddress ipAd = IPAddress.Parse("152.106.40.84");
  TcpListener listener = new TcpListener(ipAd,2055);
  // start listening at the specified port 2055
  listener.Start();

  Console.WriteLine("Server is running at port 2055...");
  Console.WriteLine("The local End point is: " + listener.LocalEndpoint);
  Console.WriteLine("Waiting for a connection...");

  Socket s = listener.AcceptSocket();
  Console.WriteLine("Connection accepted from " + s.RemoteEndPoint);

  byte[] b = new byte[100];
  int k = s.Receive(b);
  Console.WriteLine("Received...");
  for(int i=0;i<k;i++)
     Console.Write(Convert.ToChar(b[i]));

  ASCIIEncoding asen = new ASCIIEncoding();
  s.Send(asen.GetBytes("The string was received by the server."));
  Console.WriteLine("\nSent Acknowledgement");
  // clean up
  s.Close();
  listener.Stop();
  } // end try
  catch(Exception e)
  {
  Console.WriteLine("Error... " + e.StackTrace);
  } // end catch
  } // end Main
} // end class Server1
```

```
// Client1.cs

using System;
using System.IO;
using System.Text;
using System.Net;
using System.Net.Sockets;

class Client1
{
  public static void Main()
  {
  try
  {
  TcpClient tcpclnt = new TcpClient();
  Console.WriteLine("Connecting...");
  tcpclnt.Connect("152.106.40.84",2055);

  Console.WriteLine("Connected");
  Console.Write("Enter the string to be transmitted: ");
  String str = Console.ReadLine();
  Stream stm = tcpclnt.GetStream();
  ASCIIEncoding asen = new ASCIIEncoding();
  byte[] ba = asen.GetBytes(str);
  Console.WriteLine("Transmitting...");
  stm.Write(ba,0,ba.Length);
  byte[] bb = new byte[100];
  int k = stm.Read(bb,0,100);

  for(int i=0;i<k;i++)
    Console.Write(Convert.ToChar(bb[i]));
  // clean up
  tcpclnt.Close();
  } // end try
  catch(Exception e)
  {
  Console.WriteLine("Error... " + e.StackTrace);
  } // end catch
  } // end Main
} // end class Client1
```

Example 2. In our second example we include an `xml`-file (`Server2.exe.config`) on the Server side we can query from the Client. On the Server side we first compile

```
csc /D:LOG Server2.cs
```

The code between `#if LOG` and `endif` will be added by the compiler only if the symbol `LOG` is defined during compilation (conditional compilation). Next we compile the Client side

```
csc Client2.cs
```

Then on the Server side we start running the `exe`-file `Server2` and finally we start the `exe`-file `Client2` on the Client side.

```
// Server2.cs

using System;
using System.Threading;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Configuration;

class EmployeeTCPServer
{
  static TcpListener listener;
  const int LIMIT = 5; // 5 concurrent clients

  public static void Main()
  {
  IPAddress ipAd = IPAddress.Parse("152.106.40.84");
  listener = new TcpListener(ipAd,2055);
  listener.Start();
  #if LOG
      Console.WriteLine("Server mounted,listening to port 2055");
  #endif
  for(int i=0;i<LIMIT;i++)
  {
  Thread t = new Thread(new ThreadStart(Service));
  t.Start();
  } // end for loop
  } // end Main

  public static void Service()
  {
  while(true)
```

```
  {
  Socket soc = listener.AcceptSocket();
  #if LOG
      Console.WriteLine("Connected: {0}",soc.RemoteEndPoint);
  #endif
  try
  {
  Stream s = new NetworkStream(soc);
  StreamReader sr = new StreamReader(s);
  StreamWriter sw = new StreamWriter(s);
  sw.AutoFlush = true; // enable automatic flushing
  sw.WriteLine("{0} Employees available",ConfigurationManager.AppSettings.Count);
  while(true)
  {
  string name = sr.ReadLine();
  if(name == "" || name == null) break;
  string job = ConfigurationManager.AppSettings[name];
  if(job == null) job = "No such employee";
  sw.WriteLine(job);
  } // end while
  s.Close();
  } // end try
  catch(Exception e)
  {
  #if LOG
  Console.WriteLine(e.Message);
  #endif
  } // end catch
  #if LOG
  Console.WriteLine("Disconnected: {0}",soc.RemoteEndPoint);
  #endif
  soc.Close();
  }
  }
} // end class Server2
```

The file (xml-file) `Server2.exe.config` is given by

```
<configuration>
   <appSettings>
      <add key="john" value="manager"/>
      <add key="jane" value="steno"/>
      <add key="jim" value="clerk"/>
      <add key="jack" value="salesman"/>
   </appSettings>
</configuration>
```

The Client2.cs is given by

```
// Client2.cs

using System;
using System.IO;
using System.Net.Sockets;

class Client2
{
  public static void Main()
  {
  TcpClient client = new TcpClient("152.106.40.84",2055);
  try
  {
  Stream s = client.GetStream();
  StreamReader sr = new StreamReader(s);
  StreamWriter sw = new StreamWriter(s);
  sw.AutoFlush = true;
  Console.WriteLine(sr.ReadLine());
  while(true)
  {
  Console.Write("Name: ");
  string name = Console.ReadLine();
  sw.WriteLine(name);
  if(name == "") break;
  Console.WriteLine(sr.ReadLine());
  } // end while
  s.Close();
  } finally
  {
  // code in finally block is guranteed to execute irrespective
  // whether any exception occurs or does not occur in the try block
  client.Close();
  } // end finally
  } // end Main
} // end class Client2
```

10.3 User Datagram Protocol

UDP is not very reliable (but fast) connectionless protocol.

Example. The `UDPServer1.cs` file is given by

```
// UDPServer1.cs

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Configuration;

class ServerUDP
{
   public static void Main()
   {
   UdpClient udpc = new UdpClient(2055);
   Console.WriteLine("Server started servicing on port 2055");
   IPEndPoint ep = null;
   while(true)
   {
   byte[] rdata = udpc.Receive(ref ep);
   string name = Encoding.ASCII.GetString(rdata);
   string job = ConfigurationManager.AppSettings[name];
   if(job==null) job = "No such employee";
   byte[] sdata = Encoding.ASCII.GetBytes(job);
   udpc.Send(sdata,sdata.Length,ep);
   } // end while
   } // end Main
}
```

The xml-file `UDPServer1.exe.config` is given by

```
<configuration>
   <appSettings>
      <add key="john" value="manager"/>
      <add key="jane" value="steno"/>
      <add key="jim" value="clerk"/>
      <add key="jack" value="salesman"/>
   </appSettings>
</configuration>
```

The UDPClient1.cs file is given by

```
// UDPClient1.cs

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class ClientUDP
{
   public static void Main()
   {
   UdpClient udpc = new UdpClient("152.106.40.84",2055);
   IPEndPoint ep = null;
   while(true)
   {
   Console.Write("Name: ");
   string name = Console.ReadLine();
   if(name == "") break;
   byte[] sdata = Encoding.ASCII.GetBytes(name);
   udpc.Send(sdata,sdata.Length);
   byte[] rdata = udpc.Receive(ref ep);
   string job = Encoding.ASCII.GetString(rdata);
   Console.WriteLine("job = " + job);
   } // end while
   } // end Main
}
```

We include an xml-file (UDPServer1.exe.config) on the Server side we can query from the Client. On the Server side we first compile

```
csc UDPServer1.cs
```

Next we compile the Client side

```
csc UDPClient1.cs
```

Then on the Server side we start running the exe-file UDPServer1 and finally we start the exe-file UDPClient1 on the Client side.

Chapter 11

Remoting

11.1 Introduction

Object running in one Virtual Machine (VM) should be able to access (call functions on) objects that are in a different Virtual Machine. Usually this means the two processes are running on two different computers. To set up remoting we proceed as follows:

```
Instantiate target object (on server)
Setup our channels
Register our object (on server)
Get a reference to object (on client)
Call method on server from the client
```

For example on the Server side:

```
public class Server {
   public static void Main() {

     Target t = new Target();
     RemotingServices.Marshal(t,"me");

     IChannel c = new HttpChannel(8080);
     ChannelServices.RegisterChannel(c);

     Console.WriteLine("Server ready. ");
     Console.ReadLine();
     }
}
```

On the Client side we have

```
public class Client {
   public static void Main() {

     string url = "http://localhost:8080/me";
     Type to = typeof(Target);
     Target t = (Target) RemotingServices.Connect(to,url);
     Console.WriteLine("Connected to server.");
     try {
       string msg = Console.ReadLine();
       t.sendMessage(msg);
       } catch(Exception e) {
         Console.WriteLine("Error " + e.Message);
       }
       }
}
```

A shared class (i.e. it is on the Client and Server side) is

```
public class Target : System.MarshalByRefObject
{
  public void sendMessage(string msg)
  { System.Console.WriteLine(msg); }
}
```

Reference types passed via remoting must be serializable or marshal-by-reference.

Serializable == pass-by-value

Copy of object is made.

MarshalByRefObject == pass-by-reference

proxy object is passed.

An example for the second case is:

```
public class Counter : MarshalByRefObject {
  int count = 0;
  public void incCount() { count++; }
  public int getCount()  { return count; }
}
```

An example for the first case is:

```
[Serializable]
public class Counter {
  int count = 0;
  public void incCount() { count++; }
  public int getCount() { return count; }
}
```

We have to import:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http; // for Http
using System.Runtime.Remoting.Channels.Tcp;  // for Tcp
```

Shared objects must be available to both the client and the server. Make a separate assembly for it

```
csc /t:library Shared.cs
csc /r:Shared.dll Server.cs
csc /r:Shared.dll Client.cs
```

There are three main items in the process of remoting. The Interface, the Client and the Server. The Interface is a `.dll` file (generated from a `.cs` file) which is on the Client and Server side. The Server provides an implementation of the Interface and makes it available to call. The Client calls the Server using the Interface.

Example. The client provides a string to the server and the server returns the length of the string.

We first create an interface with the file `MyInterface.cs`.

```
// MyInterface.cs

public interface MyInterface
{
  int FunctionOne(string str);
}
```

`MyInterface.cs` is on the Server and Client side. We compile to `MyInterface.dll` on the Client and Server side with

```
csc /t:library MyInterface.cs
```

On the Server side we now create our class which we invoke from a remote machine (client). `RemoteObject.cs` is on the Server side and is the implementation of `MyInterface.cs`.

```
// RemoteObject.cs

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

public class MyRemoteClass : MarshalByRefObject, MyInterface
{
  public int FunctionOne(string str)
  {
  return str.Length;  // length of string
  }
}
```

Any class derived from `MarshalByRefObject` allows remote clients to invoke its methods. Now on the Server side we compile

```
csc /t:library /r:MyInterface.dll RemoteObject.cs
```

This creates the file `RemoteObject.dll` on the Server side.

Next we provide our Server on the Server side.

```
// ServerTcp.cs

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class MyServer
{
  public static void Main()
  {
  TcpChannel m_TcpChan = new TcpChannel(9999);
  ChannelServices.RegisterChannel(m_TcpChan,false);
  RemotingConfiguration.RegisterWellKnownServiceType(
      Type.GetType("MyRemoteClass,RemoteObject"),
      "FirstRemote",WellKnownObjectMode.SingleCall);
      System.Console.WriteLine("Press ENTER to quit");
      System.Console.ReadLine();
  } // end Main
}  // end class MyServer
```

First we create a `TcpChannel` object which we use to transport messages across our remoting boundary. We select `9999` as the TCP port to listen on. This could cause problems with firewalls. We use

```
ChannelServices.RegisterChannel
```

to register our `TcpChannel` with the channel services. Then we use

```
RemoteConfiguration.RegisterWellKnownServiceType
```

to register our `RemoteClass` object as a well-known type. Now this object can be remoted.

We are using the string `FirstRemote` here which will be used as part of the URL that the client uses to access the remote object. We use `SingleCall` mode here which means a new instance is created for each remote call.

Remark. One could have also used `Singleton` in which case one object would have been instantiated and used by all connecting clients.

Next we generate the `exe` file for `ServerTcp` via

```
csc /r:RemoteObject.dll ServerTcp.cs
```

Now we write our Client program `ClientTcp.cs` on the client side.

```
// ClientTcp.cs

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class MyClient
{
  public static void Main()
  {
  TcpChannel m_TcpChan = new TcpChannel();
  ChannelServices.RegisterChannel(m_TcpChan,false);
  MyInterface m_RemoteObject =
   (MyInterface) Activator.GetObject(typeof(MyInterface),
   "tcp://192.168.0.3:9999/FirstRemote");
   Console.WriteLine(m_RemoteObject.FunctionOne("willi hans steeb"));
   // for LocalHost
   // "tcp://LocalHost:9999/FirstRemote");
  } // end Main
}  // end class MyClient
```

Just as in the Server we have a `TcpChannel` object, though in this case we do not have to specify a port. We also use

```
ChannelService.RegisterChannel
```

to register the channel. We use

```
Activator.GetObject
```

to obtain an instance of the `MyInterface` object. The IP address of the remote machine (Server) is `192.168.0.3` with the port `9999`. The string `FirstRemote` which forms part of the URL was that we passed to

```
RemotingConfiguration.RegisterWellKnownServiceType
```

on the Server.

To obtain the `ClientTcp.exe` file we run

```
csc /r:MyInterface.dll ClientTcp.cs
```

To run everything on the Server side we enter the `exe` file at the command line

```
ServerTcp
```

Then on the Client side we enter the `exe` file at the command line

```
ClientTcp
```

Then we are provide with the length of the string. In the present case 16.

Instead of using Tcp we can also use Http. Then the files `ServerTcp.cs` and `ClientTcp.cs` are replaced by `ServerHttp.cs` and `ClientHttp.cs` shown below.

```
// ServerHttp.cs
// on Server side

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

public class MyServer
{
  public static void Main()
  {
  HttpChannel chan = new HttpChannel(8080);
  ChannelServices.RegisterChannel(chan,false);
  RemotingConfiguration.RegisterWellKnownServiceType(
      Type.GetType("MyRemoteClass,RemoteObject"),
      "FirstRemote",WellKnownObjectMode.SingleCall);
      System.Console.WriteLine("Press ENTER to quit");
      System.Console.ReadLine();
  } // end Main
}  // end class MyServer
```

```
// Client.cs
// on client side

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;

public class MyClient
{
  public static void Main()
  {
  HttpChannel chan = new HttpChannel();
  ChannelServices.RegisterChannel(chan,false);
  MyInterface m_RemoteObject =
   (MyInterface) Activator.GetObject(typeof(MyInterface),
   "http://152.106.41.42:8080/FirstRemote");
   Console.WriteLine(m_RemoteObject.FunctionOne("willi hans"));
   // for LocalHost
   // "tcp://LocalHost:9999/FirstRemote");
  } // end Main
}  // end class MyClient
```

Chapter 12

Accessing Databases

12.1 Introduction

The .NET Framework uses ADO.NET to access databases. ADO.NET is a set of managed classes within the .NET Framework. It operates on a disconnected data access model. When an application requests data a new database connection is created and destroyed when the request is completed. A disconnected data access model is frugal with resources, which is desirable.

One important advantage of ADO.NET is that information is stored and transferred in XML.

ADO.NET allows access to many different databases. There are two basic types of connections. SQLClient is used for Microsoft's SQL Server and OLEDB is used for all other database formats.

In order to use SQL with SQL Server or SQL Server Express the following namespace must be specified.

```
using System.Data.SqlClient;
```

In order to use OLEDB the following namespace must be specified.

```
using System.Data.OleDb;
```

When designing a database, the correct datatypes must be chosen. Note the following.

- Choosing an incorect datatype can degrade performance, because of data conversion.
- Choosing too small a datatype cannot meet the system need and at some point your system may become useless.
- Choosing too large a datatype can waste space and increase capital expenses.

Accessing a database requires that a .NET Data Provider is set up. A Data Provider consists of

- Connection
- Command
- DataReader
- DataAdapter

A **Dataset** can be understood as a virtual database table that resides in RAM. A DataSet can contain a set of tables with all the metadata necessary to represent the structure of the original database. It can be manipulated and updated independantly of the database. A DataSet is not always required.

The **Connection** creates the actual connection to the database. The Connection object contains all the information needed to open the connection, for example the userid and password. A Connection String is required to connect to a database. The easiest way to find the correct connection string for a DBMS is to search the Web. A good site to start with is http://www.connectionstrings.com.

In order to open and close a database connection the Connection methods Open() and Close() are used respectively.

The **Command** executes an instruction against the database. For example, this could be a SQL query or a stored procedure. There are three options for executing a Command.

1. ExecuteReader: Used for accessing data. Can return multiple rows. Read-only, forward-only.
2. ExecuteScalar: Used to retrieve a value from a single field.
3. ExecuteNonQuery: Used for data manipulation, such as Delete, Update and Insert.

A **DataReader** is used for fast and efficient forward-reading, read-only database access. It returns individual rows directly to the application. The data is not cached.

In the case that the data is read-only and rarely updated, then a **DataAdapter** can be used. A DataAdapter handles retrieving and updating data. It is a middle-layer between the database (Data Provider) and the disconnected DataSet. It decouples the DataSet from the database and allows a single DataSet to represent more than one database. The data adapter fills a DataSet object when reading the data and writes in a single batch when writing changes back to the data base.

An important control is the **SqlDataSource**. It allows connections to most relational databases. The default provider is for Microsoft SQL Server, but providers for other databases are provided. For example, for Oracle.

An important concept is that of **Data-Bound controls**. They provide a link between the data source controls and the user. Any data source control can be selected and linked to any data-bound controls.

12.2 Examples

The following program shows how to connect to a Microsoft SQL Server 2005 or Sql Express database and retrieve some basic information about the server. Care must be taken is creating an appropriate connection string (CON_STRING in these examples). Knowledge of SQL is required.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

namespace csDB
{
class OpenSQLDB
{
public const string CON_STRING = "Data Source=mySqlDB;Initial
Catalog=dbTest;User ID=JoeUser;Password=User1234";

static void Main(string[] args)
{
Console.WriteLine("Connecting to db\n");
SqlConnection connection = new SqlConnection(CON_STRING);
connection.Open();
Console.WriteLine("Server Version: {0}",
connection.ServerVersion);
Console.WriteLine("Database: {0}",
connection.Database);
Console.WriteLine("Data Source: {0}",
connection.DataSource);
Console.WriteLine("Stats Enabled: {0}",
connection.StatisticsEnabled);
connection.Close();
} // end Main
} // end class
} // end namespace
```

Embedding the user ID and password, as in the above code, is not a good idea; a change in the database means the code will have to be modified and recompiled. In addition, it is not secure. ASP.NET solves this problem by using the *web.config* file.

The following is an extract from a web.config file.

```
<configuration>
<appSettings>
<add key="DBConnect"
value="server=mySqlDB.newdb,net;initial
catalog=dbTest;user id=JoeUser;pwd=User1234;"/>
<appSettings>

...
<\configuration>
```

The following line of code fetches the connection string from web.config.

```
string CON_STRING=ConfigurationSettings.Appsettings["DBConnect"];
```

The basic method of retrieving data using a Data Reader is as follows. A standard SQL query is used and submitted to the database via a SqlCommand object. In the case of data retrieval the SELECT statement is used.

```
SqlCommand cmDB = new SqlCommand();
cmDB.Connection = connection ;
cmDB.CommandText = "SELECT * FROM tblCustomer";
SqlDataReader myReader = cmDB.ExecuteReader();

while(myReader.Read())
{
       // Do something with the data
       // In the following code, the myReader[0]
       // accesses the FIRST field of data in the
       // the selected record. myReader[1] then
       // accesses the SECOND field.
       // ------------------------------------
       // string strTmp = myReader[0].ToString() +
       // " " + myReader[1].ToString();
}
myReader.Close();
```

A wildcard search can be achieved as follows. The assumption is that a TextBox called txtSearch is used to input the search parameters. A Trim() is used to remove whitespace.

```
 string strSearch =
   "SELECT * FROM tblCustomer where fldCustName LIKE '" +
   TextBox1.Text.Trim() + "%'"; // SQL uses a % for wildcard
 myReader = cmDB.ExecuteReader();
```

The following code shows how to access particular fields using a Data Reader.

```
// Access SQL EXPRESS db and retrieve data and put field
// values into variables
 string CON_STRING =
        "Data Source=ejd\\sqlexpress;" +
        "Initial Catalog=dbCustomer;Integrated Security=True";
 string strSQL =
        "SELECT * FROM tblCustomer WHERE" +
        " fldCustSurname = 'Hardy' AND fldCustName = 'Frank'";

 int id; string strName, strSurname; float fltCred, fltBal;

 SqlConnection connection = new SqlConnection(CON_STRING);
 connection.Open();
 SqlCommand cmDB = new SqlCommand();
 cmDB.Connection = connection;
 cmDB.CommandText = strSQL;
 SqlDataReader rdr = cmDB.ExecuteReader();

 while (rdr.Read())
 {
      id = (int) rdr["fldCustomerID"];
      strName = (string) rdr["fldCustName"];
      // etc etc
      TextBox1.Text = id + " " + strName;

 }

 rdr.Close();
 connection.Close();
```

The following code shows how to insert a row into a database, using a SQL INSERT command.

```
 String CON_STRING
    = @"Data Source=mname\sqlexpress;" +
    "Initial Catalog=dbCustomer" +
    ";Integrated Security=True";
 SqlConnection connection = new SqlConnection(CON_STRING);
```

```
connection.Open();
String MyString
   = @"INSERT INTO tblCustomer(fldCustName, fldCustSurname)" +
   "VALUES('Angus', 'MacBeth')";
SqlCommand MyCmd = new SqlCommand(MyString, connection);
MyCmd.ExecuteScalar();
connection.Close();
```

It is likely that the C# programmer will have a need to extract some data from a TextBox, either on a Windows or Web Form, and insert that value into a table. In the case of wanting to insert the values from two TextBoxes called txtFirstName and txtSurname, you would have the following statement in C#.

```
String MyString
   = @"INSERT INTO tblCustomer(fldCustName, fldCustSurname)"
    + "VALUES('"
    + txtFirstName.Text.Trim()
    + "', '"
    + txtSurname.Text.Trim()
    + "')";
```

The following code shows how to edit a row in a database, using a SQL UPDATE command.

```
string CON_STRING =
   @"Data Source=mname\sqlexpress;" +
   "Initial Catalog=dbCustomer;Integrated Security=True";

SqlConnection connection = new SqlConnection(CON_STRING);
connection.Open();
String MyString =
   "UPDATE tblCustomer SET fldCustCred = '1000' " +
   "WHERE fldCustName = 'Angus'";
SqlCommand MyCmd = new SqlCommand(MyString, connection);

MyCmd.ExecuteNonQuery();
connection.Close();
```

The following code shows how to delete data in a database, using a SQL DELETE command. This program uses a TRY... CATCH block for exception handling.

```
string CON_STRING =
   "Data Source=mname\\sqlexpress;" +
   "Initial Catalog=dbCustomer;" +
   "Integrated Security=True";
```

```
SqlConnection connection = new SqlConnection(CON_STRING);

try
{
   connection.Open();
   String MyString = @"DELETE FROM tblCustomer " +
   "WHERE fldCustName = 'Greg'";
   SqlCommand MyCmd = new SqlCommand(MyString, connection);

   MyCmd.ExecuteScalar();

}
catch (SqlException sqlexp)
{
 // SqlException catches SQL specific exceptions
 // Use it when accessing SQL databases
}
catch (Exception ex)
{
   TextBox1.Text = ex.Message;
}
finally
{
  // The following code checks if the database is still open
  // If it is, then it is closed
    if(connection.State == ConnectionState.Open) connection.Close();
}
```

Chapter 13

ASP.NET

13.1 Introduction

ASP.NET is Microsoft's Web technology built on the .NET Framework. Of all the available .NET languages, C# is perhaps the best language to write ASP.NET applications. This chapter outlines some additional code that is required when programming web applications.

Each Web page, or Web Form, is essentially a standalone program. Web Forms have an extension of .aspx, and the C# code that is associated with a particular page has an extension of .aspx.cs. The .aspx file is essentially a HTML file, with .NET specific extentions. The .aspx.cs file requires a number of libraries to be included. These are typically as follows

```
using System;
using System.Data;
using System.Configuration;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Web.UI.HtmlControls;
```

13.2 Page Lifecycle

When a Web Page loads, it follows an eight stage lifecyle.

1. Page request - This occurs before the page life cycle begins. When a page is requested, ASP.NET determines whether the page needs to be parsed and compiled (which starts the lifecycle), or whether a cached version of the page can be sent.

2. Start - Here page properties such as Request and Response are set. It is also determined whether the request is a postback or a new request and sets the IsPostBack property appropriately. The page's UICulture property is also set.

3. Page initialization - Here controls on the page are created and each control's UniqueID property is set. If the current request is a postback, the postback data has not yet been loaded and control property values have not been restored to the values from view state. Themes are also applied.

4. Load - If the current request is a postback, control properties are loaded with information recovered from view state and control state.

5. Validation - The Validate method of all validator controls is called, which sets the IsValid property of individual validator controls and of the page.

6. Postback event handling - Event handlers are called if the request is a postback.

7. Rendering - View state is saved for the page and all controls before rendering is performed. During the rendering phase, the page calls the Render method for each control, providing a text writer that writes its output to the OutputStream of the page's Response property.

8. Unload - This is called after the page has been rendered and sent to the client. Page properties such as Response and Request are unloaded and any cleanup is performed.

13.3 Controls

A control is a component that is used to create GUI interfaces in ASP.NET and Windows development. An example of a control is a TextBox, which is used for text input. Properties and Methods are typically accessed by use of the "dot" operator.

13.3.1 TextBox

A TextBox is used for text input. The following HTML code shows how a TextBox is typically created.

```
<asp:TextBox ID="TextBox1" runat="server" Style="z-index: 100; left:
0px; position: absolute; top: 0px"></asp:TextBox>
```

The following code shows how to retrieve text from a TextBox.

```
string strTB = TextBox1.Text;
```

The following code shows how to insert text into a TextBox.

```
TextBox1.Text = "Hello World";
```

13.3.2 DropDownList

A DropDownList is a similar to a TextBox, except that it has predefined input options. The following HTML code shows how a DropDownList is typically created.

```
<asp:DropDownList ID="DropDownList1" runat="server" Style="z-index:
  102; left: 0px; position: absolute; top: 0px">
</asp:DropDownList>
```

The following code shows how to add options to a DropDownList.

```
DropDownList1.Items.Add("Anselm");
DropDownList1.Items.Add("Frank");
DropDownList1.Items.Add("Joe");
```

To retrieve the option that the user has selected, the following code can be used.

```
string strDDL = DropDownList1.SelectedItem.ToString();
```

13.3.3 CheckBox

13.3.4 CheckBoxList

13.3.5 RadioButton

13.3.6 RadioButtonList

13.3.7 Navigation

When using a Menu or Treeview Controlin ASP.NET for navigation, a file called Web.sitemap is required. Below is an example of a Web.sitemap file.

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
    <siteMapNode url="default.aspx" title="Home"  description="">
        <siteMapNode url="default2.aspx" title="Services"
         description="" />
        <siteMapNode url="default3.aspx" title="Contacts"
         description="" />
    </siteMapNode>
</siteMap>
```

13.4 State Management

Web Forms can be thought of a seperate applications. That leaves the problem of maintaining state between the Web Forms of a single application. The Session object can achieve this. The code below shows how to save a variable using the Session object.

```
string strTmp = "Some string";
Session["sessionTmp"] = strTmp;
Response.Redirect("default2.aspx");
```

The following code retrieves the data from the Session object.

```
string strRetSess = Session["sessionTmp"].ToString();
```

A Query String can be created using a HyperLink control.

```
HyperLink1.NavigateUrl = "default.aspx?name=evan";
```

The Query String can be retrieved by using the Request object.

```
String strTmp = Request.QueryString["name"];
```

For the longer term storage of data, cookies can be used. Cookies are useful for storing information between visits.

The HttpCookie class is used to create cookies. .Expire is used to set the expiry date and time. .Value is used to store information in the cookie. Below is some code for creating cookies.

```
//Create a new cookie, passing the name into the constructor
HttpCookie cookie = new HttpCookie("Names");

//Set the cookies value (In this case from a DropDownList)
cookie.Value = DropDownList1.SelectedItem.ToString();

//Set the cookie to expire in 1 minute
DateTime dtNow = DateTime.Now;
TimeSpan tsMinute = new TimeSpan(0, 0, 1, 0);
cookie.Expires = dtNow + tsMinute;

//Add the cookie
Response.Cookies.Add(cookie);

// Redirect to another Web Form
Response.Redirect("default2.aspx");
```

Data from a cookie is retrieved using the Request object.

```
// Cookie name
String strCookieName = "Names";

//Grab the cookie
HttpCookie cookie = Request.Cookies[strCookieName];

//Check to make sure the cookie exists
if (null == cookie)
{
    Response.Write("Cookie not found. <br><hr>");
}
else
{
    //Write the cookie value
    String strCookieValue = cookie.Value.ToString();
}
```

A cookie can be deleted in the following manner.

```
cookie.Expires = DateTime.Now.AddDays(-1);
Response.Cookies.Add(cookie);
```

13.5 Page Load

In the Page Load event, we employ the following code to ensure that some code is only executed the first time a Web Form is loaded. This is useful for adding items to a DropDownList. Outside of the !IsPostBack, the items will be added to the DropDownList every time the Web Form is loaded.

```
if(!IsPostBack)
{
// Code in this block will only execute once
StateDropDownList.Items.Add{"Gauteng"};
StateDropDownList.Items.Add{"Northern Province"};
}
```

Bibliography

[1] http://www.codeproject.com/csharp

[2] C# Language Specification

Index