

Modern Programming Tools and Techniques-II

DCAP505

Edited by:
Kumar Vishal



L OVELY
P ROFESSIONAL
U NIVERSITY



MODERN PROGRAMMING TOOLS & TECHNIQUES - II

Edited By
Kumar Vishal

Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Lovely Professional University
Phagwara

SYLLABUS

Modern Programming Tools & Techniques - II

Objectives: To impart the skills needed to implement Network enabled technologies. Student will learn: dot net framework classes, multi-language support, platform independence, console and windows applications development, implementing Object oriented concepts in dot net, file handling in dot net, database application development using dot net and data transmission technology.

Sr. No.	Description
1.	Introduction: What is C#, Why C#, Evolution of C#, Characteristics of C#, Difference of C# from C++ and Java, Writing a C# program
2.	Variables and Data Types: Variables and Data Types, Boxing & Un-boxing, Operators & Expressions Decision Making and Looping: If, If else if. While, do while, for loop.
3.	Handling Arrays: Declaring Arrays. System.Array class Methods: Declaring Methods, Invoking Methods, Pass by Value, Pass by Reference
4.	Classes & Object in C#: Defining Classes, object, methods. Constructors, Using Polymorphism, Inheritance in classes.
5.	Interfaces: Meaning and Implementation Namespaces: Meaning and its working. Using System Namespace and Object class.
6.	Exception Handling: Exceptions, Multiple Catch Statements, Using Finally Statement, Nested Try Blocks
7.	Windows Programming: Using Controls- textboxes, listbox, buttons, datetime picker, comboboxes etc Common Dialog Boxes: OpenFileDialog, SaveFileDialog, ColorDialog, MessageBox Class and DialogResult Class.
8.	File Input Output: Working with Files and Directories. System.IO.
9.	ADO.NET: Accesing Database with ADO.NET. Executing Insertion, deletion, updation and select command with databases.
10.	XML Basics: What is XML? Data Representation through XML. Working with XMLReader and XMLWriter Classes.

CONTENT

Unit 1:	Introduction to C# <i>Kumar Vishal, Lovely Professional University</i>	1
Unit 2:	Variables and Datatypes <i>Kumar Vishal, Lovely Professional University</i>	28
Unit 3:	Decision Making and Looping <i>Kumar Vishal, Lovely Professional University</i>	72
Unit 4:	Handling Arrays <i>Kumar Vishal, Lovely Professional University</i>	102
Unit 5:	Methods <i>Kumar Vishal, Lovely Professional University</i>	121
Unit 6:	Classes & Objects in C# <i>Kumar Vishal, Lovely Professional University</i>	143
Unit 7:	Polymorphism and Inheritance <i>Kumar Vishal, Lovely Professional University</i>	190
Unit 8:	Interfaces <i>Anil Sharma, Lovely Professional University</i>	211
Unit 9:	Exception Handling <i>Anil Sharma, Lovely Professional University</i>	244
Unit 10:	Window Programming <i>Anil Sharma, Lovely Professional University</i>	262
Unit 11:	Common Dialog Boxes <i>Anil Sharma, Lovely Professional University</i>	281
Unit 12:	File Input Output <i>Anil Sharma, Lovely Professional University</i>	297
Unit 13:	ADO.Net <i>Anil Sharma, Lovely Professional University</i>	318
Unit 14:	XML Basics <i>Anil Sharma, Lovely Professional University</i>	340

Unit 1: Introduction to C#

Notes

CONTENTS

Objectives

Introduction

- 1.1 What is C#?
- 1.2 Why C#
- 1.3 Evolution of C#
- 1.4 Characteristics of C#
- 1.5 .NET Platform
 - 1.5.1 Common Language Runtime (CLR)
 - 1.5.2 Common Type System (CTS)
 - 1.5.3 Common Language Specification (CLS)
 - 1.5.4 Class Library (CL)
- 1.6 Writing a C# Program
 - 1.6.1 Comparison between C++ and C#
 - 1.6.2 Comparison between Java and C#
- 1.7 Summary
- 1.8 Keywords
- 1.9 Review Questions
- 1.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan C#
- Describe C# evolution
- Demonstrate characteristics of C#
- Recognize the difference between C# from C++ and Java
- Write a C# program

Introduction

This unit will introduce you to C# which Microsoft introduced as an attempt to have a single computer language that has all the features that a modern computer would like it to have. It has several features and capabilities, which enable programmers to overcome the limitations of other programming languages such as Visual Basic, C and C++ and Java. You will also learn about the .NET platform which is the most important component essential to C# programming as the

Notes

codes compiled by C# program can run only on the .NET platform. You will be introduced to some of the fundamental aspects and programming basics of C.

The panacea that Microsoft offers for all the above-mentioned traumas of a programmer is the “.NET solution”. The .NET (pronounced Dot Net) framework is completely a radical change in the way the programmers have been developing applications. .NET framework launched by Microsoft provides a runtime and integration environment for different languages like C#, Jscript, C and C++ along with Web Services platform. It has four main components:

1. **.Net Framework and Visual Studio.Net:** It is a multi-language collection of programming tools that allows programs to write programs in different languages and to club them together to do a task.
2. **Server Infrastructure:** It is a collection of applications used to build and deploy XML services. For example, Mobile Information Server2001 to be used for mobile phones.
3. **Smart Devices:** It allows the use of handheld devices, laptops and PCs in the .Net environment.
4. **.Net Experiences:** It provides a personal and integrated experience using XML Web Services through smart devices.

C# is one of the programming languages included in the Visual Studio.Net version 7.0. The .Net Platform has Common Runtime Engine and Class Libraries, therefore although C# is a new language, it has complete access to the rich class libraries of tools such as Visual Basic and Visual C++. C# itself does not include a class library.

The .NET Framework is a new computing platform designed to simplify application development in the highly distributed environment of the Internet. The .NET Framework has two main components: the common language runtime and the .NET Framework class library.

The common language runtime is the foundation of the .NET Framework. You can think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict safety and accuracy of the code. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code; code that does not target the runtime is known as unmanaged code.

The .NET Framework class library is a comprehensive, object-oriented collection of reusable classes that you can use to develop applications ranging from traditional command-line or Graphical User Interface (GUI) applications to applications based on the latest innovations provided by ASP.NET and Web Services.

The .NET Framework also provides several runtime hosts, which are unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software environment that can exploit both managed and unmanaged features. The .NET Framework SDK not only provides several runtime hosts, but also supports the development of third-party runtime hosts.



Example: ASP.NET hosts the runtime to provide a scalable, server-side environment for managed code. ASP.NET works directly with the runtime to enable .aspx pages and Web Services, both of which are discussed later in this topic.

The core features provided by the .NET platform are:

1. **Interoperability with existing code:** The existing COM binaries can be fully re-used into a .NET application and vice-versa.
2. **Total language integration:** .NET supports cross-language inheritance, cross-language exceptional handling and cross-language debugging features.

3. **Common run-time environment:** .NET is equipped with a common run-time environment that can be seamlessly shared by all the .NET-aware languages.
4. **Base class library:** The incompatible API has been dressed into a consistent system of APIs making calls to methods uniform. The object model has also been redesigned to offer a uniform object interaction.
5. **Simplified COM:** The complex interfaces like IclassFactory, Iunknown, IDL code and the difficult data-types like BSTR, SAFEARRAY etc. have been done away with in .NET.
6. **Simplified deployment mechanism:** The much horrifying requirement for an object to be registered into the system-registry before it could be used on a machine has been completely removed. The .NET run-time environment takes care of all these and more. Also, a multiple version of the same DLL can exist and be accessed at the same time.

1.1 What is C#?

The needs of a computer programmer and the features they demand from a programming language keep on changing with time. Once C was supposed to be answer to all the questions in programming. Though to some extent it is true, yet the need of simpler programming languages was immediately felt. This very fact has compelled the programming language designers to continuously revise the existing programming languages. The rapid pace of improvements in related technologies demands that programming tools must improve even more rapidly to catch up with them. This is a continuous process and in this process comes a point where a revised version of a language is so much detached from the original counterpart that it takes form of an altogether new language.

Due to ongoing struggle for creating the ultimate programming language we have ended up having so many programming languages today - one catering to a specific need better than the others.

Latest in the sequel of this evolutionary development, Microsoft introduced a new programming language represented symbolically as C# and pronounced see-sharp. In the recent past it was observed that both languages C and C++ were very popular among programmers due to the programming-power they provided to them. However, Visual Basic was found to be popular because of its simplicity. Some languages have much desirable features while lack in other useful features. In short, no single computer language has all the features that a modern computer would like it to have. C# is an attempt of Microsoft to provide just that.

The designers of C# have lifted good features from the existing programming languages and alleviated them from arcane drudgery by reducing the limitations or removing them completely. Let us have a look at what limitations in existing programming languages are and how C# enables the programmers to overcome them.

- C# has retained the code-simplicity of Visual Basic. VB programmers would not feel the migration-pain while shifting to C#. VB suffered severely from its limited OOP capabilities. C# offers true OOP capabilities to the VB programmers without sacrificing the simplicity of the code.
- C# has a significant resemblance with the syntax of C++ and Java. It has retained the usability of OOP and API that C++ provides but has done away with the much error-prone memory management, much difficult pointer manipulations and incomprehensible code-structures. C++ programmers have long waited for these enhancements.
- Java programmers felt handicapped when they wanted to use the existing codes written in alien languages. Though Java offers platform-independence it is not truly language-independent. A Java application has to be written in Java end-to-end. It has very limited capability to use non-Java API's. C# heralds a good news to Java programmers by providing seamless integration with millions of lines of codes written in non-Java languages.

Notes

- Internet programming has always been a difficult enterprise. Because of the availability of great numbers of incompatible and semi-compatible technologies Internet programmers never had a blissful moment. C# does not differentiate between a normal application from Internet application. This very fact is enough a reason for the Internet programmers to migrate to C#.
- The most important component that is essential to C# programming is the .NET platform. The codes produced by compilation of a C# program can run only on the .NET platform. However, as you will see in the forthcoming sections, this very aspect is not a limitation as it may sound but a means to achieve total platform-independence.



Did u know? **What does the Microsoft's new release of Visual Studio includes?**

Microsoft's new release of Visual Studio includes all the necessary tools for C# programming on .NET platform. It also includes VB.NET, ASP.NET apart from other language tools. The studio has been named Visual Studio.NET version 7.0. The studio is an extension to the previous version and hence all the previous versions of tools and libraries are also accessible to the programmers.

1.2 Why C#

C and C++ are the most widely used programming languages in today's scenario. But they are complex and less user-friendly than Visual Basic. Therefore a Language was required which could combine the power of C and C++ and user friendliness of Visual Basic along with the portability of Java.

The .NET Framework class library is a collection of reusable classes, or types, that tightly integrate with the common language runtime. The class library builds on the object-oriented nature of the runtime, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the learning curve associated with using a new piece of code. In addition, third-party components can integrate seamlessly with the classes in the .NET Framework.



Example: The .NET Framework collection classes implement a set of interfaces, which you can use to develop your own collection classes. Your collection classes will then blend seamlessly with the classes in the .NET Framework.

As you would expect from an object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios.



Task Answer the following questions:

1. What is .Net Framework? How it is related with Visual Studio.Net?
2. Define the architecture of .Net.

1.3 Evolution of C#

Birth of a language may be an event but it is only through a series of evolutionary transformations that it attains maturity. C# is no exception.

Here's a brief history of the (C#) and the (.NET). The .NET CLR is based on Colusa software's language-neutral OmniVM, a part of their Omniware product. OmniVM was based on research carried out by Steven Lucco at Carnegie Mellon University. Steven co-founded Colusa Software in February 1994 in Berkeley, California. Omniware was released in August 1995. Colusa started working with Microsoft in February 1996. Microsoft acquired Colusa Software in the year 1996. Colusa Virtual Machine was christened as Intermediate Language (IL) subsequently.

Self Assessment

Fill in the blanks:

1. VB programmers would not feel the migration- pain while shifting to
2. .NET supports cross-language inheritance, cross-language exceptional handling and cross-language features.
3. The .NET Framework has two main components: the common language runtime and the .NET Framework
4. The class library builds on the nature of the runtime, providing types from which your own managed code can derive functionality.
5. The .NET Framework class library is a collection of reusable classes, or types, that tightly integrate with the runtime.

1.4 Characteristics of C#

The main design goal of C# was simplicity rather than pure power. It has type safety and automatic garbage collection in return. C# can make your code more stable and productive overall, meaning that you can more than make up that lost power in the long run. C# offers several key benefits for programmers:

- Simplicity
- Consistency
- Modernity
- Object-orientation
- Type-safety
- Scalability
- Version support
- Compatibility
- Flexibility

Let's look at each of the ways that C# stands to improve your coding life.

Simplicity

What's one of the most annoying things about working in C++? It has to use the -> pointer indicator, when to use the: for a class member, and when to use the dot. And the compiler knows when you get it wrong. In C#, everything is represented by a dot. Whether you're looking at members, classes, name-spaces, references, or what have you, you don't need to track which operator to use.

Notes

The second most annoying thing about working in C and C++ is figuring out exactly what type of data type to use. In C#, a Unicode character is no longer a `wchar_t`, it's a `char`. A 64-bit integer is a `long`, not an `__int64`. And a `char` is a `char` is a `char`. There's no more `char`, `unsigned char`, `signed char`, and `wchar_t` to track.

The third most annoying problem that you run across in C and C++ is integers being used as Booleans, causing assignment errors when you confuse `=` and `==`. C# separates these two types, providing a separate `bool` type that solves this problem. A `bool` can be `true` or `false`, and can't be converted into other types. Similarly, an integer or object reference can't be tested to be `true` or `false` - it must be compared to zero (or to `null` in the case of the reference). If you wrote code like this in C++:

```
int i;
if (i) . . .
```

You need to convert that into something like this for C#:

```
int i;
if (i != 0) . . .
```

Another programmer-friendly feature is the improvement over C++ in the way `switch` statements work. In C++, you could write a `switch` statement that fell through from case to case.

For example, this code

```
switch (i)
{
case 1:
    FunctionA();

case 2:
    FunctionB();
    Break;
}
```

would call both `FunctionA` and `FunctionB` if `i` was equal to 1. C# works like Visual Basic, putting an implied `break` before each case statement. If you really do want the case statement to fall through, you can rewrite the `switch` block like this in C#:

```
switch (i)
{
case 1:
    FunctionA();
    goto case 2;
case 2:
    FunctionB();
    Break;
}
```

Consistency

Notes

C# unifies the type system by letting you view every type in the language as an object. Whether you're using a class, a struct, an array, or a primitive, you'll be able to treat it as an object. Objects are combined into namespaces, which allow you to access everything programmatically. This means that instead of putting includes in your file like this

```
#include <stdlib.h>
#include <studio.h>
#include <string.h>
```

You include a particular namespace in your program to gain access to the classes and objects contained within it as:

```
using System;
```

In COM+, all classes exist within a single hierarchical namespace. In C#, the using statement lets you avoid having to specify the fully qualified name when you use a class.



Example: The System namespace contains several classes, including Console.

Console has a WriteLine method that, as you might expect, writes a line to the system console. If you want to write the output part of a Hello World program in C#, you can say:

```
System.Console.WriteLine("Hello World!");
```

This same code can be written as:

```
Using System;
Console.WriteLine("Hello World!");
```

That's almost everything you need for the C# Hello World program. A complete C# program needs a class definition and a Main function. A complete, console-based Hello World program in C# looks like:

```
using System;
class HelloWorld
{
    public static int Main(String[] args)
    {
        Console.WriteLine("Hello, World!");
        return 0;
    }
}
```

The first line makes System – the COM+ base class namespace – available to the program. The program class itself is named HelloWorld (code is arranged into classes, not by files). The Main method (which takes arguments) is defined within HelloWorld. The COM+ Console class writes the friendly message, and the program is finished.

One final point about classes. If you have classes with the same name in more than one namespace, C# lets you define aliases for any of them so you don't have to fully qualify them. Suppose you have created a class NS1.NS2.ClassA that looks like this:

```
namespace NS1.NS2
{
```

Notes

```
class ClassA {}  
}
```

You can then create a second namespace, NS3 that derives the class N3.ClassB from NS1.NS2.ClassA like this:

```
namespace NS3  
{  
    class ClassB: NS1.NS2.ClassA {}  
}
```

If this construct is too long for you, or if you're going to repeat it several times, you can use the alias A for the class NS1.NS2.ClassA with the using statement like so:

```
namespace NS3  
{  
    using A = NS1.NS2.ClassA;  
    class ClassB: A {}  
}
```

This effect can be accomplished at any level of an object hierarchy. For instance, you could also create an alias for NS1.NS2 like this:

```
namespace NS3  
{  
    using C = NS1.NS2;  
    class ClassB: C.A {}  
}
```

Modernity

C and C++ provide reliable transportation, but lack some of the features that programmers look for. This is one of the reasons many developers have adopted the Java language over the past few years.

C# goes back to the basics and emerges with several features that programmers longed for in C++. Garbage collection is one example-everything gets cleaned up when it's no longer referenced. However, garbage collection can have a price. It makes problems caused by certain risky behavior (using unsafe casts and stray pointers, for example) far harder to diagnose and potentially more devastating to a program. To compensate for this, C# implements type safety to ensure application stability. Of course, type safety also makes your code more readable, so others on your team can see what you've been up to-you take the bad with the good.

C# has a richer intrinsic model for error handling than C++. Have you ever really got deep into a coworker's code? It's amazing-there are dozens of unchecked HRESULTs all over the place, and when a call fails, the program always ends up displaying an "Error: There was an error" message.



Caution C# improves on this situation by providing integral support for throw, try...catch, and try...finally as language elements. True, you could do this as a macro in C++, but now it's available right out of the box.

Borrowing from languages like SQL, C# implements built-in support for data types like decimal and string, and lets you implement new primitive types that are as efficient as the existing ones.

You'll also be happy to see that C# takes a more modern approach to debugging. The traditional way to write a debug-able program in C++ was to sprinkle it with `#ifdefs` and indicate that large sections of code would only be executed during the debugging process. You would end up with two implementations—a debug build and a retail build, with some of the calls in the retail build going to functions that do nothing. C# offers the conditional keyword to control program flow based on defined tokens.

Finally, C# is designed to be easy to parse, so vendors can create tools that allow source browsing and two-way code generation.

Object Oriented

C++ is object oriented. That's why C# ditches multiple inheritance in favor of native support for the COM+ virtual object system. Encapsulation, polymorphism, and inheritance are preserved without all the pain.

C# ditches the entire concept of global functions, variables, and constants. Instead, you can create static class members, making C# code easier to read and less prone to naming conflicts.

And speaking of naming conflicts, have you ever forgotten that you created a class member and redefined it later on in your code? By default, C# methods are non-virtual, requiring an explicit virtual modifier. It's far harder to accidentally override a method, it's easier to provide correct versioning, and the vtable doesn't grow as quickly. Class members in C# can be defined as private, protected, public, or internal. You retain full control over their encapsulation.

Methods and operators can be overloaded in C#, using a syntax that's a lot easier than the one used by C++. However, you can't overload global operator functions—the overloading is strictly local in scope. The overloading of method F below is an example of what this looks like:

```
interface ITest
{
    void F();           // F()
    void F(int x);     // F(int)
    void F(ref int x); // F(ref int)
    void F(out int x); // F(out int)
    void F(int x, int y); // F(int, int)
    int F(string s);   // F(string)
    int F(int x);      // F(int)
}
```

The COM+ component model is supported through the implementation of delegates—the object-oriented equivalent of function pointers in C++.



Task Interfaces support multiple inheritance. Explain with proper program.

Classes can privately implement internal interfaces through explicit member implementations, without the consumer ever knowing about it.

Notes

Type Safety

Several features that promote proper code execution (and more robust programs) in Visual Basic have been included in C#. For example, all dynamically allocated objects and arrays are initialized to zero. Although C# doesn't automatically initialize local variables, the compiler will warn you if you use one before you initialize it. When you access an array, it is automatically range checked. Unlike C and C++, you can't overwrite unallocated memory.

In C# you can't create an invalid reference. All casts are required to be safe, and you can't cast between integer and reference types. Garbage collection in C# ensures that you don't leave references dangling around your code. Hand-in-hand with this feature is overflow checking. Arithmetic operations and conversions are not allowed if they overflow the target variable or object. Of course, there are some valid reasons to want a variable to overflow. If you do, you can explicitly disable the checking.

As I've mentioned, the data types supported in C# are somewhat different from what you might be used to in C++. For instance, the char type is 16 bits.



Notes Certain useful types, like decimal and string, are built in. Perhaps the biggest difference between C++ and C#, however, is the way C# handles arrays.

C# arrays are managed types, meaning that they hold references, not values, and they're garbage collected. You can declare arrays in several ways, including as multidimensional (rectangular) arrays and as arrays of arrays (jagged). Note in the following examples that the square brackets come after the type, not after the identifier as in some languages.

```
int[ ] intArray;           // A simple array
int[ , , ] intArray;      // A multidimensional array
                           // of rank 3 (3 dimensions)
int[ ][ ] intArray        // A jagged array of arrays
int[ ][ , , ][ , ] intArray; // A single-dimensional array
                           // of three-dimensional arrays
                           // of two-dimensional arrays
```

Arrays are actually objects; when you first declare them they don't have a size. For this reason, you must create them after you declare them. Suppose you want an array of size 5. This code will do the trick:

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

If you do this twice, it automatically reallocates the array. Therefore

```
int[] intArray;
intArray = new int[5];
intArray = new int[10];
```

results in an array called intArray, which has 10 members. Instantiating a rectangular array is similarly easy:

```
int[,] intArray = new int[3,4];
```

However, instantiating a jagged array needs a bit more work. You might expect to say `new int[3][4]`, but you really need to say:

Notes

```
int[][] intArray = new int[3][];
For (int a = 0; a < intArray.Length; a++) {
    intArray[a] = new intArray[4];
}
```

You can initialize a statement in the same line you create and instantiate it by using curly brackets:

```
int[] intArray = new int[5] {1, 2, 3, 4, 5};
```

You can do the same thing with a string-based array:

```
string[] strArray = new string[3] {"MSJ", "MIND", "MSDNMag"};
```

If you mix brackets, you can initialize a multidimensional array:

```
int[,] intArray = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
```

You can also initialize a jagged array:

```
int[][] intArray = new int[][] {
    new int[] {2,3,4}, new int[] {5,6,7}
};
```

If you leave out the new operator, you can even initialize an array with implicit dimensions:

```
int[] intArray = {1, 2, 3, 4, 5};
```

Arrays are considered objects in C#, and as such they are handled like objects, not like an addressable stream of bytes. Specifically, arrays are automatically garbage collected, so you don't need to destroy them when you're finished using them. Arrays are based on the C# class `System.Array`, so you can treat them conceptually like a collection object, using their `Length` property and looping through each item in the array. If you define `intArray` as shown earlier, the call

```
intArray.Length
```

would return 5. The `System.Array` class also provides ways to copy, sort, and search arrays.

C# provides a `foreach` operator, which operates like its counterpart in Visual Basic, letting you loop through an array. Consider this snippet:

```
int[] intArray = {2, 4, 6, 8, 10, -2, -3, -4, 8};
foreach (int i in intArray)
{
    System.Console.WriteLine(i);
}
```

This code will print each number in `intArray` on its own line of the system console. The `System.Array` class also provides a `GetLength` member function, so the preceding code could also be written like this (remember, arrays are zero-based in C#):

```
for (int i = 0; i < intArray.GetLength(); i++)
{
    System.Console.WriteLine(intArray(i));
}
```

Notes

Scalability

C and C++ require all sorts of often-incompatible header files before you can compile all but the simplest code. C# gets rid of these frequently aggravating headers by combining the declaration and definition of types. It also directly imports and emits COM+ metadata, making incremental compiles much easier.

When a project gets large enough, you might want to split up your code into smaller source files. C# doesn't have any restrictions about where your source files live or what they're named. When you compile a C# project, you can think of it as concatenating all the source files, then compiling them into one big file. You don't have to track, which headers go where, or which routines belong in which source file. This also means that you can move, rename, split, or merge source files without breaking your compile.



Task Analyze the use of scalability in C sharp.

Version Support

DLL Hell is a problem for users and programmers alike. C# was designed to make versioning far easier by retaining binary compatibility with existing derived classes. When you introduce a new member in a base class as one that exists in a derived class, it doesn't cause an error. However, the designer of the class must indicate whether the method is meant as an override or as a new method that just hides the similar inherited method.

C# works with a namespace model. Classes and interfaces in class libraries must be defined in hierarchical namespaces instead of in a flat model. Applications can explicitly import a single member of a namespace, so there won't be any collisions when multiple namespaces contain similarly named members. When you declare a namespace, subsequent declarations are considered to be part of the same declaration space.

Compatibility

Four types of APIs are common on the Windows platform and C# supports all of them. The old-style C APIs have integrated support in C#. Applications can use the N/Direct features of COM+ to call C-style APIs. C# provides transparent access to standard COM and OLE Automation APIs and supports all data types through the COM+ runtime. Most importantly, C# supports the COM+ Common Language Subset specification. If you've exported any entities that aren't accessible from another language, the compiler can optionally flag the code.



Example: For instance, a class can't have two members run Job and run job because a case-insensitive language would choke on the definitions.

When you call a DLL export, you need to declare the method, attach a sysimport attribute, and specify any custom marshaling and return value information that overrides the COM+ defaults. The following shows how to write a Hello World program that displays its message of cheer in a standard Windows message box.

```
class HelloWorld
{
    [sysimport(dll = "user32.dll")]
    public static extern int MessageBoxA(int h, string m,
```

```
string c, int type);
```

Notes

```
public static int Main()
{
    return MessageBoxA(0, "Hello World!", "Caption", 0);
}
}
```

Each COM+ type maps to a default native data type, which COM+ uses to marshal values across a native API call. The C# string value maps to the LPSTR type by default, but it can be overridden with marshaling statements like so:

```
using System;
using System.InteropServices;

class HelloWorld
{
    [DllImport("user32.dll")]
    public static extern int MessageBoxW(
        int h,
        [MarshalAs(UnmanagedType.LPWStr)] string m,
        [MarshalAs(UnmanagedType.LPWStr)] string c,
        int type);

    public static int Main()
    {
        return MessageBoxW(0, "Hello World!", "Caption", 0);
    }
}
```

In addition to working with DLL exports, you can work with classic COM objects in several ways: create them with `CoCreateInstance`, query them for interfaces, and call methods on them.

If you want to import a COM class definition for use within your program, you must take two steps. First, you must create a class and use the `comimport` attribute to mark it as related to a specific GUID. The class you create can't have any base classes or interface lists, nor can it have any members.

```
// declare FilgraphManager as a COM classic coclass
[comimport, guid("E436EBB3-524F-11CE-9F53-0020AF0BA770")]
class FilgraphManager
{
}
```

After the class is declared in your program, you can create a new instance of it with the new keyword (which is equivalent to the `CoCreateInstance` function).

Notes

```
class MainClass
{
    public static void Main()
    {
        FilgraphManager f = new FilgraphManager();
    }
}
```

You can query interfaces indirectly in C# by attempting to cast an object to a new interface. If the cast fails, it will throw a `System.InvalidCastException`. If it works, you'll have an object that represents that interface.

```
FilgraphManager graphManager = new FilgraphManager();
IMediaControl mc = (IMediaControl) graphManager;
mc.Run(); // If the cast succeeds, this line will work.
```

Flexibility

It's true that C# and COM+ create a managed, type-safe environment. However, it's also true that some real-world applications need to get to the native code level either for performance considerations or to use old-style, unmodernized APIs from other programs. I've discussed ways to use APIs and COM components from your C# program. C# lets you declare unsafe classes and methods that contain pointers, structs, and static arrays. These methods won't be type-safe, but they will execute within the managed space so you don't have to marshal boundaries between safe and unsafe code.

These unsafe features are integrated with the COM+ EE and code access security in COM+. This means that a developer can pin an object so that the garbage collector will pass over them when it's doing its work. (Sort of like a mezuzah for your code.)



Did u know? **What happens to the unsafe code?**

Unsafe code won't be executed outside a fully trusted environment. Programmers can even turn off garbage collection while an unsafe method is executing.

Remote Execution

If it is determined that an object's identity cannot be shared then a remoting boundary is put in place. A remoting boundary is implemented by the CLR using proxies. A proxy represents an object on one side of the remoting boundary and all instance field and method references are forwarded to the other side of the remoting boundary. A proxy is automatically created for objects that derive from `System.MarshalByRefObject`.

The CLR has a mechanism that allows applications running from within the same operating system process to be isolated from one another. This mechanism is known as the application domain. A class in the base class library encapsulates the features of an application domain known as `AppDomain`. A remoting boundary is required to effectively communicate between two isolated objects. Because each application domain is isolated from another application domain, a remoting boundary is required to communicate between application domains.

Self Assessment**Notes**

Fill in the blanks:

6. A integer is a long, not an __int64. And a char is a char is a char.
7. Objects are combined into, which allow you to access everything programmatically.
8. C# has a richer intrinsic model for error handling than
9. Each COM+ type maps to a default native data type, which COM+ uses to marshal values across a native
10. A proxy is automatically created for objects that derive from

1.5 .NET Platform

Dot Net (.NET) is the Microsoft's latest offering for all the programmers belonging to every school of thought. Dot Net has introduced radical changes in the programming paradigm albeit only to make the task simpler. Moreover, it does not take too much effort from any programmer to switch over to this platform.

Soon after the official announcement of the first release, as many as 30 different programming languages have initiated .NETting themselves. No matter whether you program in C, C++, Java, Cobol, Smalltalk, Pascal or any other language, you can always harness the huge advantages that .NET has to offer. The languages, which have reincarnated themselves into .NET version are referred to as .NET aware languages.

.NET is itself not a programming language. It is rather a programming framework that provides all the goodies of all the programming languages under one roof. You can write one module of an application in Java, another in Cobol, pick up a module written in C and integrate them all to make a single application. Could you imagine anything like this before!

.NET platform provides both integrated development environment and a uniform runtime environment. The framework is a new programming platform aptly suited for application development and deployment in highly distributed environment of such as the Internet.

A basic understanding of .NET framework is desirable to start programming in C#. .NET Framework has four main components:

1.5.1 Common Language Runtime (CLR)

Language Runtime or simply runtime is an interface that provides services to a program while executing its code. You probably already would know about Java runtime - Java Virtual Machine (JVM); Visual Basic 6.0 runtime module - msvbvm60.dll etc.

These runtimes provide the necessary linkages to the existing codes that ultimately make the execution possible. It is responsible for loading and managing the code during its execution. CLR is the very core of the .NET Framework. It acts as an agent that manages code at execution time. It provides services such as memory management; thread management, remoting and enforcing strict safety and accuracy of the code. In relation to runtime, codes can be either managed or unmanaged.

Notes



Did u know? **What is managed code?**

Code that targets the runtime is known as managed code; code that does not target the runtime is known as unmanaged code.

Since .NET has to support several .Net aware programming languages, it must provide a common runtime that manages codes written in all these languages. CLR does exactly that - it provides a common language runtime environment. It is this common runtime that makes .NET operating system independent.

The .NET Common Language Runtime is implemented as a dynamic link library - mscorlib.dll. The programs written for .NET runtime will run on any operating system where .NET runtime is available.

1.5.2 Common Type System (CTS)

Every programming language deals with certain data types. However, the data types are often incompatible between languages. For instance, an integer type data may occupy different number of bytes in the memory in different languages. This inconsistency must be resolved for a common language runtime.

CTS is the specification for such a common data type. It formally specifies the syntax and semantics of how different data types must be declared in the .NET aware programming languages so that CLR may support them while execution. In order that all these languages may share their codes among each other, they must conform to these specifications.

Specifically CTS defines the following common types:

- **CTS Class Types:** A class is the core implementation of object-oriented programming paradigm. It encapsulates data and methods and makes inheritance possible. Every .NET aware language must support a class structure. The declaration must also specify whether a class can be sub-classed or not; whether the class implements any interface or not; whether the class is abstract or not and the visibility of the class. All the classes implicitly derive from the base class System.Object.
- **CTS Structure Types:** A structure is a data type that comprises of a group of other data types and yet can be treated as a single unit. Structure in .NET world is very similar to structures in other platforms. .NET also allows constructors in a structure which can be used to initialize its members at the time of structure creation.
- **CTS Interface Types:** Interfaces are abstract classes. They cannot be instantiated. They simply specify the methods without providing any code for the same. It is up to the classes and sub-interfaces to implement an interface by providing code for it. .NET interfaces can derive from multiple interfaces.
- **CTS Enumeration Type:** An enumeration data type allows programmers to group a number of data items in such a way that they can be accessed by their cardinal number. Every enumeration type must derive from System.Enum base class.
- **CTS Delegate Types:** If you are aware of language C, you would also probably be aware of function pointers. A function pointer is an address in the memory where a function is stored. A function loaded in the memory may be called by its name or through its function pointer. The problem with C function pointer is that it happens to be just an address and therefore it may not always be safe to use. .NET delegate type is loosely equivalent to C function pointer. However, in contrast, it is a class rather than just an address. It must derive from MulticastDelegate base class. For this reason safety of function calling is guaranteed.

- **CTS Intrinsic Data Types:** .NET defines clearly and rigorously all the atomic data types. As expected even these intrinsic data types are also classes. Each .NET aware language must conform to these specifications.

Notes



Task Analyze the basic difference between CLR and CTS.

1.5.3 Common Language Specification (CLS)

Every programming language has its own syntactical structure of writing programs. For instance some languages concatenate two strings using + and some using &. While some languages use return reserved word to return a value from a function, some simply use assignment. The respective compilers will flag errors if a program does not adhere to the specific syntactic rules of that language.

.NET provides a common set of rules that all .Net aware languages must conform to. Their compilers have been designed to support these rules. This common set of rules is called CLS.

1.5.4 Class Library (CL)

The libraries accompanied with each language need not be thrown away. .NET provides mechanism through which these libraries can be used in a .NET aware program. Strictly speaking there is no class library in C#. However, it can use any class provides with the .NET framework.

.Net organizes all these binary classes into neatly packed assemblies and access them using namespace. These namespaces are uniformly accessible to all the .NET aware languages. Here is a synopsis of how the System.Console class is accessed by VB.NET, C# and .NET aware C++ (aka Managed C++ or simply MC++).

Code in VB.NET

```
Imports System
Public Module Test
```

```
Sub Main()
```

```
    Console.WriteLine("Welcome to the .NET world")
```

```
End Sub
```

```
End Module
```

Code in C#

```
using System;
```

```
public class Test()
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        Console.WriteLine("Welcome to the .NET world");
```

```
    }
```

```
}
```

Notes

```
Code in MC++  
  
using namespace System;  
  
void main()  
{  
  
    Console.WriteLine("Welcome to the .NET world");  
  
}
```

.NET programmers have access to a rich class library organized into namespaces. A program on .NET platform must use at least System namespace which defines Object root class. In order to use a class, structure, methods, enumeration or any such construct packed into a namespace the program must specify the reference to the namespace. As you have seen in the example codes given above, VB.NET uses Imports clause; C# uses using clause; and MC++ uses using namespace clause.



Task Answer the following questions:

1. What are some of limitations of other programming languages that C# enables programmers to overcome?
2. What is .Net Framework? Explain the architecture of .Net.
3. What are the different functionalities associated with Common Language Runtime (CLR)?
4. What are the common data types specified by the component CTS of .Net framework?
5. What do you mean by a namespace? List down some of the Namespace available in .Net.
6. What are .NET aware languages?

Self Assessment

Fill in the blanks:

11. .NET platform provides both integrated development environment and a uniform environment.
12. The .NET Common Language Runtime is implemented as a dynamic link library

1.6 Writing a C# Program

To learn the basics of a programming language, it is always better to write a simple program first. Let us begin with a very simple working program in C#. The program listed below displays - Welcome to the World of C Sharp - on the screen. We will save this program in a text file - FirstProgram and attach a file name extension - cs.

```
/*  
  
This is our first C Sharp program.  
  
The program displays a message on the screen.  
  
*/
```

```
class FirstProgram
{
    public static void Main()
    {
        System.Console.WriteLine("Welcome to the World of C Sharp");
    }
}
```

The program can be saved on the disk as a text file - FirstProgram.cs - using any text editor like Microsoft Windows Notepad. The text file of the program is called its source code. In order to run the program you must compile the source code (.cs) into executable code (.exe). The command that compiles this (.cs) file into (.exe) file is csc. Type the following at the DOS console prompt to invoke the compiler.

```
csc /r:System.dll FirstProgram.cs
```

This will create the executable file - FirstProgram.exe. Type the name of the exe file - FirstProgram - at the prompt to run the program. You should see the following output.

“Welcome to the World of C Sharp”

This is the general method of creating source and executable codes. However, .NET compliant language tools also come into GUI (Graphical User Interface) form. The IDE (Integrated Development Environment) provides easy to use user-friendly editor, compiler, debugger and loader. We will learn the use of IDE in subsequent sections.

Before we move any further, let us examine the different parts of this elementary program that does nothing useful than displaying a message on the screen. The part being discussed is presented in bold fonts.

```
Comment Line(s)
/*
This is our first C Sharp program.
The program displays a message on the screen.
*/
class FirstProgram
{
    public static void Main()
    {
        System.Console.WriteLine("Welcome to the World of C Sharp");
    }
}
```

Like in C and C++ everything enclosed between `/*` and `*/` is considered as a comment. While compilation the compiler ignores the comment. Comments are a useful way of documenting a program.



Did u know? **What is block commenting?**

The comments enclosed between `/*` and `*/` pairs can be extended to multiple lines. This is called block commenting.

Notes

C# provides yet another comment syntax that comments a single line. Anything written after `'//'` on a line is treated as a comment. Thus, the block comment of the FirstProgram can also be written as shown below.

```
//This is our first C Sharp program.  
//The program displays a message on the screen.  
class FirstProgram  
{  
    public static void Main()  
    {  
        System.Console.WriteLine("Welcome to the World of C Sharp");  
    }  
} //End of the program
```

Note that a line comment has also been introduced in the last line of the program.

Class Definition

```
/*  
This is our first C Sharp program.  
The program displays a message on the screen.  
*/  
class FirstProgram  
{  
    public static void Main()  
    {  

```

Like in any other object-oriented language such as C++ and Java, in C# also the main component of a program is a class that encapsulates data and methods. A C# class has the following form.

```
class classname  
{  
    .....  
    .....  
}
```

A class in C# is defined using `- class -` keyword. The calssname is a user-defined identifier and it simply identifies a class. In our example program there is a single class - FirstProgram. All the methods and codes are inside the class within a pair of braces - { - and - }. The braces - { - and - } signify the start and the end of the class definition respectively.

The Main Method

```
/*  
This is our first C Sharp program.
```

The program displays a message on the screen.

Notes

```
*/
class FirstProgram
{
    public static void Main()
    {
        System.Console.WriteLine("Welcome to the World of C Sharp");
    }
}
```

An application (or program) may have many classes and a class may have a number of methods. A method executes only when it is called. A special method of a C# class is the Main method. Main method is the starting point of execution of an application. Therefore, one class in the application must have a Main method.



Notes When the application is executed, its Main method executes first automatically.

In case your application does not have a class with the Main method, the compiler will report an error message - No Start Point Defined.

The syntax of the Main method is given below.

```
public static void Main()
{
    ...
    ...
}
```

The Main method also has several parts. Let us understand each one of them in brief.

- **public:** A C# method may or may not be accessible to other methods. Accessibility of a method is specified by an access modifier. `public` is an access modifier that implies that the method is accessible from all the other methods. Since the Main method should be accessible to everyone in order for the .NET Runtime to be able to call it automatically it is always defined as `public`.
- **static:** This indicated that the method is a Class Method. Hence it can be called without making a instance of the class first.
- **void:** A method is always expected to return a value to its caller. However, when no value is returned the value returned must be indicated as `void` type. Nobody stops you from letting your function return any other data type.
- **Main():** A method may have any valid name and the programmer is free to use any name for his methods. However, the Main method (with capitalized M) has a special meaning. This is the main entry point of the class. Programmers write their executing codes in this method, which may create other objects and call accessible methods on them.

Notes

The System.Console.WriteLine Method

WriteLine method is a static member of the Console class of the System namespace. Namespace is a way to package many related classes under one identity. What the method it does is obvious. It prints on the console whatever is provided to it as argument.

As it is clear from the example, writing a C# program is basically writing a class. Every class is derived from the Object class in C#. The base class of every other class in C# is the class Object. It is defined under System namespace. It is the top most class found at the root of the .NET class hierarchy.

1.6.1 Comparison between C++ and C#

Since C# has close resemblance with the C++ syntax, it is only expectable that the two languages will have remarkable similarities. Indeed the two are similar in many ways (see the list below).

Similarities

Both C# and C++ are object-oriented as also case-sensitive. In both the languages one cannot use a variable before declaring it explicitly. Besides, objects are instantiated from their classes using new keyword.

Dissimilarities

However, the similarities end here. The points of differences are that while the entry point in C++ is void main (), in C# everything happens in a class. While in C++ a library needs to be explicitly referenced, in C# there is an assembly - mscorlib.dll - which implicitly loads all the required libraries. While C++ uses double colon (::) for a resolution operator, C# uses a period (.). Here is an example.

In C++ MyClass::MyMethod();

In C# MyClass.MyMethod();

1.6.2 Comparison between Java and C#

Similarities

C# has a few notable similarities with Java. Both implement the concept of automatic garbage collection.

In both the languages, source code is converted to intermediate code - in C# it is converted to MSIL whereas in java it is converted into bytecode. Both use JIT compiler to convert the intermediate code to native code at runtime. Class is the action-joint for both the languages. Neither of the two languages uses pointers or header-files and uninstantiated variables. Exception handling concept is very similar to both the languages.

Both are type safe languages. In both the cases, the Object class is base class for all other classes. Both the languages have their exclusive execution environment - CLR in C# and JVM in Java.

Dissimilarities

C#, being a newer language than Java, has introduced a number of improvements over Java. The points of differences between the two are therefore due to these improvements.

Java relies on code interpretation while C# is never interpreted. This makes C# code execute faster than that in Java. Java does not support Win32 and COM platforms whereas C# does. Unlike Java structure type, structure data type in C# is value type and not reference type and therefore have no inheritance. C# allows the programmer to mark certain codes to be unsafe to allow it to have memory access. In C# all exceptions are unchecked.

While all the methods in Java are by default virtual, in C# they must to be explicitly declared virtual. C# provides more numerical precision in its data types than that in Java. There is no need of wrapper classes to use the value data types as reference data types in C# unlike Java.



Task Answer the following questions:

1. Explain the different ways of adding comments in a C# program.
2. Illustrate the comment line operators of C# using suitable example.
3. Explain the Main Method with its parts. What happens when an application does not have a class with the Main method?
4. Compare and contrast between C# and C++.
5. Compare and contrast between C# and Java.
6. What are some of the applications that can be developed on the .NET Framework?

Self Assessment

Fill in the blanks:

13. C# has a few notable similarities with Java. Both implement the concept of automatic collection.
14. .NET compliant language tools also come into form.
15. Namespace is a way to package many related under one identity.



Caselet

Computers are language-neutral...

If you are a talented computer programmer, there is nothing to stop you from learning several computer languages and becoming 'multilingual,' encourages Richard Mansfield in *Programming: A beginner's guide* (www.tatamcgrawhill.com). "Some people excel at this, just as some find it easy to learn multiple human languages. Clearly, you'll benefit in the marketplace – at least you'll qualify for a greater number of jobs – if you know how to program in multiple languages."

To an extent, different languages suit different tasks, the author explains. He mentions how, for example, VB shines as a teaching language, for general-purpose programming, and when working with databases; and how PERL, being quite flexible, can be particularly good for manipulating strings.

"And, of course, you might also want to explore C#, a gateway into the C languages... Microsoft's C# is probably the best place to start."

Contd...

Notes

A common misunderstanding, Mansfield rues, is that computer languages can't resemble English. The computer isn't like 'a Tibetan villager with whom you must learn to communicate in Tibetan if you hope to communicate at all,' he says.

"Some people think computers have a language built in, as if they were born in Machineland or something. A computer has no native high-level language, merely a simple set of built-in, low-level capabilities such as the capacity to turn memory cells on or off or add numbers together." The computer is language-neutral like babies, analogizes Mansfield. "Raise a Tibetan in Tucson and she will speak English."

The book concludes on a note of reassurance to amateurs. If you decide to pursue programming alone, as a hobby, you can nonetheless still make good money at it, the author cheers. "A thriving community of independent programmers publish shareware... Take a look at www.download.com or www.majorgeeks.com." He cites WinZip, Paint Shop, and the McAfee security programs as instances of commercial success, with humble origins as shareware.

Recommended Read.

Law, Technology, and Society

If you are looking for an example of a grassroots national movement, check the discussion on the Centre for Technology and Society (CTS) in *Implementing the World Intellectual Property Organization's Development Agenda*, edited by Jeremy de Beer (www.idrc.ca).

The CTS was founded in 2003, in the Rio de Janeiro School of Law, and is currently Brazil's leading academic institution dealing with the interplay of law, technology, and society, informs Pedro Paranaguá in one of the essays included in the book.

"The CTS manages extensive research and educational programmes, employing an interdisciplinary approach. Its collaborators include anthropologists, journalists, computer scientists, economists, and media executives, as well as law professors and researchers."

Among the initiatives that the CTS has undertaken, using the Berkman Centre for Internet and Society at Harvard as a model, is the promotion, for the first time in Latin America, of Harvard's Internet Law Seminar (I-Law), "which brought to Brazil important IP (intellectual property) legal scholars including Jonathan Zittrain, Lawrence Lessig, Yochai Benkler, Charles Nesson, John Perry Barlow, among others." The repercussions of the I-Law/ Brazil continue to gain attention in the Brazilian legal-technical community, Paranaguá observes.

With the financial support of the Financiadora de Estudos e Projectos, the CTS has created a group to provide legal support for free and open source software research, he adds. "As a strategy for spreading the free software legal concepts, the CTS has created a national contest for research projects dealing with practical problems connected with free software businesses."

The contest has been 'ported' to Brazil's nineteen open source licence models, such as the Mozilla and the BSD licences, the author informs. "These licences had no Portuguese language version, and the CTS accredited these Brazilian versions."

Worth a Detailed Study

Assemblies of Parts

An inventor fails 999 times, and if he succeeds once, he's in, says Charles Kettering. For, "an inventor treats his failures simply as practice shots." Perhaps, with all the right tools in place, the 'Inventor' software reverses the proportion; so much so, you may find it 'a breeze

Contd...

to work with,' with only 'a handful ways to make it difficult,' as Thom Tremblay assures in Autodesk Inventor 2010: No experience required (www.wileyindia.com).

"One common bad habit is not properly controlling where files are kept. Inventor keeps track of where things are supposed to be, using a project file." The project file, as he explains, allows one to control where Inventor looks for templates, what styles are available, and where the files are stored, including standard parts such as bolts.

"This opens up a lot of possibilities, such as using different project files to switch templates with different title blocks when you work on jobs for multiple customers. Over time, more and more Inventor users establish one project file and sort jobs as folders under that project file."

Most users don't create individual parts, Tremblay notes. "They create assemblies of parts. That is why Autodesk Inventor was written with the assembly in mind."

Though the size and complexity of assemblies may vary, at their core they are collections of components that are fastened, welded, or in some other way stuck together, he says.

For the Hands-on Design Professional.

Academic Publishing

Online publishing of books, unlike journals, is still in its infancy and there remain prejudices about the quality of online material and a reluctance to read large amounts of text online, opines Sarah Caro in How to Publish Your PhD (www.sagepublications.com). An example of success that she mentions is that of Rice University Press, 'which has completely reinvented itself as an online academic press.'

On the online platform, scholarly works are freed from the restrictions of paper and ink, and can include not just words but unlimited visual material including pictures, computer graphics, video clips and live Internet links and audio material of all kinds, explains Caro.

"The possibilities are endless and very exciting for those engaged in serious scholarship in areas such as performing arts, art history and cultural studies."

A few issues that may yet require attention are the rigour of the review process, and the difficulty in comparisons between the multimedia and text presentations.

"The fact remains, however, that the hardcopy, single-authored monograph does offer all the benefits of traditional academic publishing. It provides an excellent showcase for your talents," Caro avers.

Useful addition to the researchers' shelf.

dmurali@thehindu.co.in

Tailpiece

"We could recover our investment in electronic surveillance cameras..."

"Through improvements in workplace productivity?"

"No, by selling critical market intelligence data about human behaviour!"

1.7 Summary

- C# several features and capabilities, which enable programmers to overcome the limitations of other programming languages such as Visual Basic, C and C++ and Java.
- The .Net Framework is a new computing platform designed to simplify application development in the highly distributed environment of the Internet.

Notes

- The four components of .Net framework are .NET Common Language Runtime, Common Type, Common Language Specification and Class Library.
- Net is equipped with a common run time environment that can be seamlessly shared by all the .Net aware languages.
- C# unifies the type system by letting you view every type in the language as an object.
- C# is a modern, type safe programming language, object oriented language that enables programmers to quickly and easily built solutions for the Microsoft .Net platform.
- The .Net framework also provides the collection of classes and tools to aid in development and consumption of web services applications.
- In order to use C# and the .Net framework classes, you first need to install either the .Net framework SDK, or else Visual Studio.NET.
- There are two types of comments – one line comment (//) and multiple line comment (/* ...*/).
- A special method of a C# class is the main method which is the starting point of execution of an application. It has several parts such as public, static, void and Main().
- What makes the .NET Framework suitable for developers is its class library, which is a collection of pre-compiled reusable classes and types.

1.8 Keywords

.NET Platform: It is a programming framework that provides both integrated development environment and a uniform runtime environment and this platform is aptly suited for application development and deployment in highly distributed environment such as the Internet.

Common Language Infrastructure (CLI): It is an open specification developed by Microsoft that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework.

Common Language Run Time: It is the heart of .Net framework and is responsible for overall execution of a .Net program.

Common Type System (CTS): It is used by every language built on the .NET Framework, the CTS specifies no particular syntax or keywords, but instead defines a common set of types that can be used with many different language syntaxes.

Namespaces: Net framework base classes are organized in two folders and units in which C# classes are organized are called Namespaces.

Visual Studio .Net: It is a Multilanguage collection of programming tools that allow programmers to write programs in different languages.

1.9 Review Questions

1. Explain the relevance of the following terms with respect to C#.
 - (a) Run-anywhere
 - (b) Objects owned
 - (c) Lifestyle choice
 - (d) Type safe
2. Explain how .NET attains language independence in addition to platform independence.

3. Compare and contrast between C#, C++ and Java.
4. What are the different functionalities associated with CLR?
5. How a Web Service is different from a Web based application?
6. Why is C# is called as backward compatible language?
7. Can a C# class have more than one main method? If yes, what purpose does it serve?
8. What is the purpose of aliasing a namespace?
9. Illustrate the comment line operators of C# using suitable example.
10. Write a class echo that heads your name from keyboard and prints it back on the monitor.
11. Illustrate different mathematical function available in Math namespace.
12. List down the advantages and limitations of using Aliases.
13. Write the syntax of declaring an alias.
14. Which mathematical function is used to check whether a number is positive or negative?

Notes

Answers: Self Assessment

- | | |
|--------------------|-------------------------------|
| 1. C# | 2. debugging |
| 3. class library | 4. object-oriented |
| 5. common language | 6. 64-bit |
| 7. Namespaces | 8. C++ |
| 9. API call | 10. System.MarshalByRefObject |
| 11. Runtime | 12. mscorere.dll |
| 13. garbage | 14. GUI |
| 15. classes | |

1.10 Further Readings



Books

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

[http://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](http://en.wikipedia.org/wiki/C_Sharp_(programming_language))

<http://www.functionx.com/csharp/>

Unit 2: Variables and Datatypes

CONTENTS

Objectives

Introduction

2.1 Value Types

2.2 Datatypes

2.2.1 Integral Types

2.2.2 Floating Point Types

2.2.3 The Bool Type

2.2.4 Enumeration Types

2.2.5 Reference Types

2.3 Variables

2.3.1 Variable Categories

2.3.2 Declaration of Variables

2.3.3 Initialization of Variables

2.4 Boxing and Unboxing

2.4.1 Boxing Conversions

2.4.2 Unboxing Conversions

2.5 Operators

2.5.1 Unary Operators

2.5.2 Binary Operators

2.5.3 Conversion Operators

2.5.4 Postfix Increment and Decrement Operators

2.5.5 New Operator

2.5.6 Arithmetic Operators

2.5.7 Relational Operators

2.5.8 Logical Operators

2.5.9 Assignment Operators

2.5.10 Increment and Decrement Operators

2.5.11 Conditional Operator

2.5.12 Bitwise Operators

2.5.13 Checked and Unchecked Operators

2.6 Expressions

2.7 Summary

2.8 Keywords

2.9 Review Questions

2.10 Further Readings

Objectives

Notes

After studying this unit, you will be able to:

- Explain the Variables and Datatypes
- Describe Boxing and Unboxing
- Demonstrate the Operators and Expressions

Introduction

A datum (singular of data) is a value in the context of processing. Data can assume one among various different forms. The type of processing that can be operated on a datum determines. Thus a data of numeric type allows arithmetic operation on itself while a data of text type does not. Note that the data does not depend on the data itself. It depends on the operations one wants to put the data to. For instance, a phone number may be taken as text type instead of numeric type since one would not subject a phone number for addition, subtraction or any other arithmetical operation.

Data is usually stored on storage devices like a hard disk or a compact disk. However, for processing purposes it must be brought and stored in the main memory (RAM). A variable, which could be referenced by a suitable name, is employed to store data in the RAM. Note that the type of the variable provides the necessary semantics to the value stored in it. Thus, the value 5 stored in a variable of integer type is different from the value 5 stored in a character type variable.

When a variable is created in the memory by a program it needs to know what type of data it is supposed to store. For this reason every programming language defines various types of data it supports and provides means to create and manipulate them.

Data type thus determines following aspects of a variable:

1. The domain of values it can store
2. The operations that can be applied on the data stored in the variable
3. The size (in terms of number of bytes) of the variable in the memory.

Programs are primarily written to process data. Therefore, a programming language must provide a variety of data types to support processing of different types of data the users may have. Data types fall into two different categories:

- **Primitive data types:** Primitive data types are the fundamental data types which cannot be divided into simpler data types. For this reason, they are also known as atomic data types. The range of the values assignable to primitive data types as also their size and operations are pre-defined in the language itself.
- **User defined data types:** Primitive data types are not sufficient for many programming problems. Therefore, a programming language also provides mechanism to create new data types by modifying and grouping primitive data types. These data types, which are defined by programmers, are called user defined data types.

Different data types can be used in three different forms – value type, reference type and pointer type.

- **Value type:** In this form the data type stores real data. When the data are queried by different function a local copy of these memory cells are created. It guarantees that changes made to data in one function don't change them in some other function.

Notes

- **Reference type:** Reference type data do not store the actual data. They store reference which indirectly access a data.
- **Pointer type:** Pointer type does not exist in C#. However, it does provide mechanism to use pointer types in unsafe code.

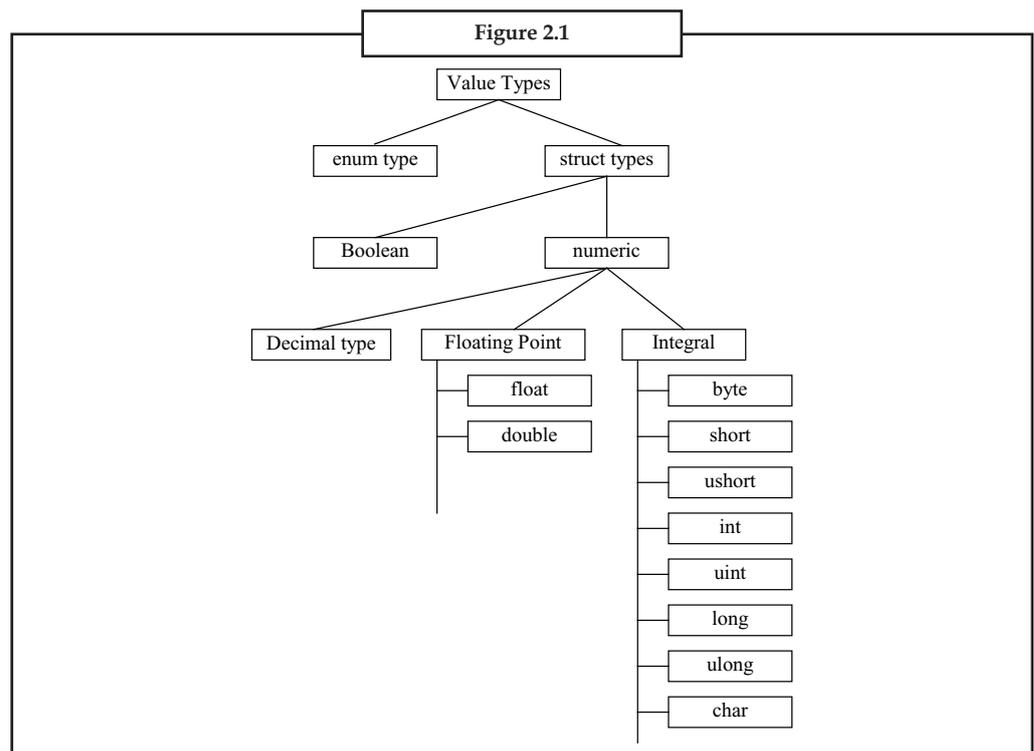
The most important improvement in C# data type system is that it is unified. A value of any type can be treated as an object. Every type in C# directly or indirectly derives from the object class type, the root base class for all other classes. Values of reference types are treated as objects simply by viewing the values as type object.



Caution Values of value types can be treated as objects as well as mere values by performing boxing and unboxing operations.

2.1 Value Types

C# provides two categories of value types – struct type or enumeration type. A set of predefined struct types is also available. It is called the simple types. The simple types are identified through reserved words, and are further subdivided into numeric types, integral types, and floating-point types.



Value types are also classes. They implicitly inherit from class object. However, value types are sealed and cannot be extended to form sub-classes. A variable of a value type always contains a value of that type. Unlike reference types, it is not possible for a value of a value type to be null or to reference an object of a more derived type.

Assignment to a variable of a value type creates a copy of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

Notes

All value types implicitly declare a public parameterless constructor called the default constructor. The default constructor returns a zero-initialized instance known as the default value for the value type. For an enum-type E, the default value is 0. For a struct-type, the default value is the value produced by setting all value type fields to their default value and all reference type fields to null.



Notes Like any other constructor, the default constructor of a value type is invoked using the new operator.

Consider the code listed below. Here both i and j variables are initialized to zero.

```
class TestOne
{
    void TestFunction()
    {
        int i = 0;
        int j = new int();
    }
}
```

Since every value type implicitly has a public parameterless constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is however permitted to declare parameterized constructors (see the code listed below).

```
struct Point3D
{
    int x, y, z;
    public Point3D(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Having defined this, look at the following statements:

```
Point3D p1 = new Point3D();
```

```
Point3D p2 = new Point3D(0, 0, 0);
```

These statements create Point3D objects p1 and p2 with x, y and z initialized to zero.

Simple Types

C# provides a set of predefined struct types called the simple types. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the System namespace, as described in the table below.

Notes

Reserved word	Aliased type
Sbyte	System.SByte
Byte	System.Byte
Short	System.Int16
Ushort	System.UInt16
Int	System.Int32
UInt	System.UInt32
Long	System.Int64
Ulong	System.UInt64
Char	System.Char
Float	System.Single
Double	System.Double
Bool	System.Boolean
Decimal	System.Decimal

Because a simple type aliases a struct type, every simple type has members.



Example: int has the members declared in System.Int32 and the members inherited from System.Object, and the following statements are permitted:

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();       // System.Int32.ToString() instance method
string t = 123.ToString();     // System.Int32.ToString() instance method
```

The simple types differ from other struct types in that they permit certain additional operations:

- Most simple types permit values to be created by writing literals. For example, 123 is a literal of type int and 'a' is a literal of type char. C# makes no provision for literals of other struct types, and values of other struct types are ultimately always created through constructors of those struct types.
- When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile-time. Such an expression is known as a constant-expression. Expressions involving operators defined by other struct types always imply run time evaluation.
- Through const declarations it is possible to declare constants of the simple types. It is not possible to have constants of other struct types, but a similar effect is provided by static read-only fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator.

Self Assessment

Fill in the blanks:

1. Assignment to a of a value type creates a copy of the value being assigned.
2. All value types implicitly declare a public constructor called the default constructor.

3. Expressions involving operators defined by other types always imply run time evaluation.
4. Value types are sealed and cannot be extended to form

2.2 Datatypes

2.2.1 Integral Types

C# offers several integral data type, each different in their size and range of values it can store. The various integral types, their sizes and range of values are:

Data type	Size (in bytes)	Range of values
sbyte	1	-128 to 127
byte	1	0 to 255
short	2	-32768 to 32767
ushort	2	0 to 65535
int	4	-2147483648 to 2147483647
uint	4	0 to 4294967295
long	8	9223372036854 to 9223372036854775807
ulong	8	0 to 18446744073709551615
char*	2	0 to 65535

* Though char type is classified as an integral type, it differs from the other integral types in the following ways:

- No implicit conversions exist from other types to the char type. In particular, even though the sbyte, byte, and ushort types have ranges of values that can be fully represented using the char type, implicit conversions from sbyte, byte, or ushort to char are not allowed.
- char type constants must be written as character-literals. Character constants can only be written as integer-literals in combination with a cast. For example, (char)10 is the same as '\x000A'.

2.2.2 Floating Point Types

Two floating types are supported by C# – float and double. The float and double types are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats, which provide the following sets of values:

Positive Zero and Negative Zero

In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two.

Positive Infinity and Negative Infinity

Infinities are produced by such operations as dividing a non-zero number by zero. For example 1.0 / 0.0 yields positive infinity, and -1.0 / 0.0 yields negative infinity.

The Not-a-Number Value

Not-a-number values (NaN's) are produced by invalid floating-point operations, such as dividing zero by zero.

Notes

The following observations are worth taking a note:

- The float type can represent values ranging from approximately 1.5×10^{-45} to 3.4×10^{38} with a precision of 7 digits.
- The double type can represent values ranging from approximately 5.0×10^{-324} to 1.7×10^{308} with a precision of 15-16 digits.
- If one of the operands of a binary operator is of a floating-point type, then the other operand must be of an integral type or a floating-point type, and the operation is evaluated as follows:
 - ❖ If one of the operands is of an integral type, then that operand is converted to the floating-point type of the other operand.
 - ❖ If a floating-point operation is invalid, the result of the operation becomes NaN.
 - ❖ If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.



Task Give the example of positive zero and negative infinity.

2.2.3 The Bool Type

C# does not use numerical values for representing true and false Boolean values like C. There is bool type data that represents boolean logical quantities in C# with only two possible values - true and false. No conversion exists between bool type and other data types.

It is important to note that C# differs from C and C++ in this regard. In C and C++ languages, a zero integral value or a null pointer can be converted to the boolean value false, and a non-zero integral value or a non-null pointer can be converted to the boolean value true. In C#, such conversions are accomplished by explicitly comparing an integral value to zero or explicitly comparing an object reference to null.

Thus the following code is incorrect in C#

```
bool one = 0;
bool two = -4;
bool three = 1;
```

The following assignments are only valid assignments in C#

```
bool one = true;
bool two = false;
```

Also, while in C and C++ is correct,

```
a = TRUE;
if( a ) { };
```

the same is incorrect in C#. In C# one has to explicitly include logical operator as shown below.

```
a = true;
If( a != true) { };
```

2.2.4 Enumeration Types

Notes

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying integer type. Enumeration types are defined through enumeration declarations as shown below.

```
enum Teams
{
    India, England, Australia, Pakistan
}
```

The keyword `enum` declares an enum type. The above declaration declares an enum type named `Teams` whose members are `India`, `England`, `Australia` and `Pakistan`.

Each enum type has a corresponding integral type called the underlying type of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. Note that `char` cannot be used as an underlying type.



Note An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

Look at the following code.

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

This code declares an enum with an underlying type of `long`. A programmer might choose to use an underlying type of `long` for greater range than that of integer.

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum-members can have the identical name.

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. Consider the following code.

```
enum Direction: uint
{
    East = -1,
    West = -2,
    North = -3,
    South = -4;
}
```

Notes

The above code has a declaration error. The constant values -1, -2, -3 and -4 are not in the range of the underlying integral type uint.

More than one members of an enum type may share the same associated value as shown in the code listed below.

```
enum Direction
{
    East,
    West,
    North,
    Sunrise = East
}
```

Here, two enum members - East and Sunrise have the same associated value, i.e., East. The individual members of the enum type are accessed as usual by dot operator as shown below.

```
Direction.East
Direction.West etc.
```

You can assign the associated value to an enum either implicitly or explicitly. If the declaration of the enum member has a constant-expression initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly. If the member is the first enum member declared in the enum type, its associated value is zero. Otherwise, the associated value of the enum member is obtained by increasing the associated value of the previous enum member by one. This increased value must be within the range of values that can be represented by the underlying type.

Consider the following code.

```
using System;

enum Direction
{
    East,
    West = 15,
    North,
    South
}

class Test
{
    static void Main()
    {
        Console.WriteLine(ShowDirection(Direction.East));
        Console.WriteLine(ShowDirection(Direction.West));
        Console.WriteLine(ShowDirection(Direction.North));
    }
}
```

```
Console.WriteLine(ShowDirection(Direction.South));
}

static string ShowDirection(Direction c)
{
    switch (c)
    {
        case Direction.East:
            return String.Format("East = {0}", (int)c);
        case Direction.West:
            return String.Format("West = {0}", (int)c);
        case Direction.North:
            return String.Format("North = {0}", (int)c);
        case Direction.South:
            return String.Format("South = {0}", (int)c);
        default:
            return "Invalid direction";
    }
}
}
```

This program prints out the enum member names and their associated values. The output is:

East = 0

West = 15

North = 16

South = 17



Notes The enum member East is automatically assigned the value zero since it has no initializer and is the first enum member. The enum member West is explicitly given the value 15 and the enum members North and South are automatically assigned the value one greater than the member that textually precedes it, i.e., 16 and 17.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

Consider the following code.

```
enum CircularExample
{
    One = Two,
    Two
}
```

Notes

This code is invalid because the declarations of One and Two are circular. One depends on Two explicitly, and Two depends on One implicitly. Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility-an enum member is accessible if its containing enum type is accessible.



Task Answer the following question:

1. Define data. What are the categories of data types?
2. Explain the different forms that data types can take.
3. Explain the two categories of value types C# provides.
4. What is the difference between values types and reference types?
5. Explain the concept of Enumeration with the help of suitable example.
6. What are the advantages of enumeration data type?

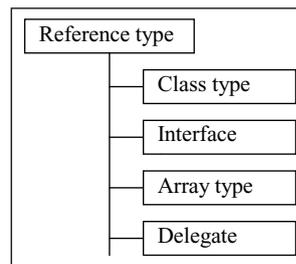
Self Assessment

Fill in the blanks:

5. C# offers several integral data type, each different in their and range of values it can store.
6. No implicit conversions exist from other types to the type.
7. Each enum type has a corresponding type called the underlying type of the enum type.

2.2.5 Reference Types

A reference type is a class type, an interface type, an array type, or a delegate type.



A reference type value is a reference to an instance of the type, the latter known as an object. The special value null is compatible with all reference types and indicates the absence of an instance.

Class Types

A class type defines a data structure that contains data members (constants, fields, and events), function members (methods, properties, indexers, operators, constructors, and destructors), and nested types.



Did u know? **What is the basic property of class type?**

Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using object-creation-expressions.

Object Class Type

It is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the object class type.

The object keyword is simply an alias for the predefined System.Object class. Writing the keyword object is exactly the same as writing System.Object, and vice versa.

String Type

The string type is a sealed class type that inherits directly from object. Instances of the string class represent Unicode character strings. Values of the string type can be written as string literals.

The string keyword is simply an alias for the predefined System.String class. Writing the keyword string is exactly the same as writing System.String, and vice versa.

A character that follows a backslash character (\) in a regular-string-literal-character must be one of the following characters: ', ", \, 0, a, b, f, n, r, t, u, U, x, v. Otherwise, a compile-time error occurs.

Following are some examples of valid string literals.

```
string a = "hello, there";
string b = @"hello, there";
string c = "hello \t there";
string d = @"hello \t there";
string e = "Mahesh said \"Hello\" to me";
string f = @"Mahesh said \"Hello\" to me";
string g = "\\nic\share\file.txt";
string h = @"\\nic\share\file.txt";
string i = "first\nsecond\nthird";
string j = @"first second third";
```

The last string literal, *j*, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as newline characters, are preserved verbatim.



Note Each string literal does not necessarily result in a new string instance.

When two or more string literals that are equivalent according to the string equality operator appear in the same program, these string literals refer to the same string instance (see the following program).

```
class TestTwo
```

Notes

```
{  
  
    static void Main()  
    {  
  
        object a = "jumbo";  
        object b = "jumbo";  
        Console.WriteLine(a == b);  
  
    }  
}
```

The output is True because the two literals refer to the same string instance.

The string class derives directly from the object, and is sealed i.e. user cannot derive from it. The string keyword is simply an alias for the predefined System.String class.

String Manipulation in C#

System.String provides a number of properties and methods through which strings may be manipulated in various ways. One of the properties is

Length: This property returns the length of the current string.

Some of the useful string-methods are:

Concat(): This is a static method of String class which returns a new string by concatenating (joining) two strings.

CompareTo(): Compares two strings.

Copy(): Returns a new copy of the existing string.

Format(): This method is used to format the string.

Insert(): This method inserts a string into another string.

PadLeft(): This method appends a string with some character to the left side.

PadRight(): This method appends a string with some character to the rightside.

Remove(): Removes a substring from a string.

Replace(): Replaces a substring with some other substring.

ToUpper(): Coverts to upper case.

ToLower(): Coverts to lower case.

Although, string data type is reference type, the equality operator (== and !=) compare the string objects and not the memory they reference. Also, addition operator (+) has been overloaded as concatenating operator for strings.



Example: String arrays are indexed as will be evident from the following example.

```
using System;  
  
class TestThird  
{  
  
    public static void Main()  
    {
```

```

        string[] names = {"Rajesh", "Vibhor", "Geeta", "Jyoti"};
        foreach (string person in names)
        {
            Console.WriteLine("{0} ", person);
        }
    }
}

```

Within the “foreach” parentheses is an expression composed of two elements divided by the keyword “in”. The right-hand side is the collection you want to use to access each element. The left-hand side holds a variable with a type identifier compatible with whatever type the collection returns.

Every time through this loop the collection is queried for a new value. As long as the collection can return a value, this value will be put into the read-only variable and the expression will return true, thus causing the statements in the “foreach” block to be executed. When the collection has been fully traversed, the expression will evaluate to false and control will transfer to the first executable statement following the end of the “foreach” block.

Interface Types

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Array Types

An array is a data structure that contains a number of variables which are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

Delegate Types

A delegate is a data structure that refers to a static method or to an object instance and an instance method of that object.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores not only a reference to the method’s entry point, but also a reference to the object instance for which to invoke the method.

Self Assessment

Fill in the blanks:

8. A reference type value is a reference to an instance of the type, the latter known as an
9. When the collection has been fully traversed, the expression will evaluate to and control will transfer to the first executable statement following the end of the “foreach” block.

2.3 Variables

During its operation a computer program has to store data values in the memory. The memory location where a data value can be stored and retrieved is called a variable. The interpretation of the value stored in a variable is determined by its type. The C# compilers are designed to check the data compatibility at the compile time itself. C# compiler guarantees that values stored in variables are always of the appropriate type. For this reason C# is called a type-safe language.

The value stored in a variable keeps on changing as the processing progresses. The value can be changed through assignment or through increment (++) and decrement (--) operators.



Caution Variable must have some value stored in it before its value can be obtained.

2.3.1 Variable Categories

For convenience C# variables can be put into the following categories.

Static Variables

A static variable is created when the type in which it is declared is loaded, and ceases to exist when the program terminates. The modifier `static` before the declaration indicates that the variable is static.

The initial value of a static variable is the default value of the variable's type. For purposes of definite assignment checking, a static variable is considered initially assigned.

Instance Variables

A field declared without the `static` modifier is called an instance variable. An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed. The initial value of an instance variable of a class is the default value of the variable's type. For purposes of definite assignment checking, an instance variable of a class is considered initially assigned.

An instance variable of a structure has exactly the same lifetime as the structure variable to which it belongs. The initial assignment state of an instance variable of a struct is the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so too are its instance variables, and when a struct variable is considered initially unassigned, its instance variables are likewise unassigned.



Task The initial value of an instance variable of a class is the default value of the variable's type. Analyze.

Array Elements

The elements of an array are created when an array instance is created, and cease to exist when there are no references to that array instance. The initial value of each of the elements of an array is the default value of the type of the array elements. For purposes of definite assignment checking, an array element is considered initially assigned.

Value Parameters

Notes

A parameter declared without a `ref` or `out` modifier is a value parameter. A value parameter comes into existence upon invocation of the function member (method, constructor, accessor, or operator) to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the function member. For purposes of definite assignment checking, a value parameter is considered initially assigned.

Reference Parameters

A parameter declared with a `ref` modifier is a reference parameter. A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters.

- A variable must be definitely assigned before it can be passed as a reference parameter in a function member invocation.
- Within a function member, a reference parameter is considered initially assigned.
- Within an instance method or instance accessor of a structure type, the `this` keyword behaves exactly as a reference parameter of the structure type.

Output Parameters

A parameter declared with an `out` modifier is an output parameter. An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of an output parameter is always the same as the underlying variable. The following definite assignment rules apply to output parameters.

A variable need not be definitely assigned before it can be passed as an output parameter in a function member invocation.

Following a function member invocation, each variable that was passed as an output parameter is considered assigned in that execution path. Within a function member, an output parameter is considered initially unassigned.

Every output parameter of a function member must be definitely assigned before the function member returns.



Note Within a constructor of a structure type, the `this` keyword behaves exactly as an output parameter of the structure type.

Local Variables

A local variable is declared by a local-variable-declaration, which may occur in a block, a `for`-statement, a `switch`-statement, or a `using`-statement. A local variable comes into existence when control enters the block, `for`-statement, `switch`-statement, or `using`-statement that immediately contains the local variable declaration. A local variable ceases to exist when control leaves its immediately containing block, `for`-statement, `switch`-statement, or `using`-statement.

Notes

A local variable is not automatically initialized and thus has no default value. For purposes of definite assignment checking, a local variable is considered initially unassigned. A local-variable-declaration may include a variable-initializer, in which case the variable is considered definitely assigned in its entire scope, except within the expression provided in the variable-initializer.

Within the scope of a local variable, it is an error to refer to the local variable in a textual position that precedes its variable-declarator.

Consider the following code example.

```
class Test
{
    public static int i;
    int j;
    void F(int[] Books, int aVar, ref int bVar, out int cVar)
    {
        int k = 1;
        cVar = aVar - bVar++;
    }
}
```

Here in this example:

- i is a static variable
- j is an instance variable
- Books[0] is an array element
- aVar is a value parameter
- bVar is a reference parameter
- cVar is an output parameter



Task Answer the following questions:

1. What are the different categories of variables? Explain with the help of suitable examples.
2. What is the difference between Instance variable and class variable?
3. How reference parameters are different from value parameters?
4. How output parameter is different from reference parameter?

2.3.2 Declaration of Variables

Variables are declared before they are used. C# being a type safe language makes this rule obligatory. The general syntax of variable declaration is given below.

```
<data_type> <variable_1>, <variable_2>, ..., <variable_n>
```

For instance, the following code declares a variable sum as integers data type.

```
int sum;
```

2.3.3 Initialization of Variables

Notes

Variables, when created, must store valid values. Assigning a value to a newly created variable is known as variable initialization. If no explicit initialization is performed by the programmer, C# places default values to the variables. Here is a code that creates and initializes a variable sum.

```
int sum;
sum = 100;
```

A variable may be declared and initialized at the same time as shown below.

```
int sum = 100;
```

Default Values

As mentioned above, in case a variable is not initialized explicitly, a default value is stored automatically. The default value is always a valid value for the variable concerned. The numerical variables are initialized to 0 while character variables to space.

Constant Variables

Just as an identifier can be declared as a variable meant to store a varying data value of suitable type, so also it can be declared to store a fixed value. When an identifier retains the value once assigned to it, it is known as constant variable. The following code shows the declaration of a constant variable.

```
const real pi = 3.714
```

The keyword `const` makes the variable `pi` a constant variable. Once declared this way, `pi` represents the real value 3.714 wherever it appears in the program.

Self Assessment

Fill in the blanks:

10. A parameter declared with an `out` modifier is an parameter.
11. Variables, when created, must store values.

2.4 Boxing and Unboxing

Boxing and unboxing is a central concept in C#'s type system. It provides a binding link between *value-types* and *reference-types* by permitting any value of a *value-type* to be converted to and from type object. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

2.4.1 Boxing Conversions

A boxing conversion permits any *value-type* to be implicitly converted to the type object or to any *interface-type* implemented by the *value-type*. Let us consider an example where for any value type `B`, the boxing class would be declared as follows:

```
class B_Box
{
    B value;
}
B_Box(B b) {
```

Notes

```

        value = b;
    }
}

```

Boxing of a value *v* of type *T* now consists of executing the expression `new B_Box(v)`, and returning the resulting instance as a value of type `object`. Thus, the statements

```

int i = 123;
object box = i;

```

Conceptually correspond to

```

int i = 123;
object box = new int_Box (i);

```

Boxing classes like `B_Box` and `int_Box` above don't actually exist and the dynamic type of a boxed value isn't actually a class type. Instead, a boxed value of type *B* has the dynamic type *B*, and a dynamic type check using the `is` operator can simply reference type *B*. For example,

```

int i = 123;
object box = i;
if (box is int) {
    Console.WriteLine("Box contains an int");
}

```

will output the string "Box contains an int" on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a *reference-type* to type `object`, in which the value continues to reference the same instance and simply is regarded as the less derived type `object`. For example, given the declaration

```

struct Point
{
    public int x, y;

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

```

the following statements

```

Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.WriteLine(((Point)box).x);

```

The output would be 10.

because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` instead been declared a class, the value 20 would be output because `p` and `box` would reference the same instance.

2.4.2 Unboxing Conversions

Notes

An unboxing conversion permits an explicit conversion from type *object* to any *value-type* or from any *interface-type* to any *value-type* that implements the *interface-type*. An unboxing operation consists of first checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of the instance.

Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an object *box* to a *value-type* *B* consists of executing the expression `((B_Box)box).value`. Thus, the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new int_Box(123);
int i = ((int_Box)box).value;
```

For an unboxing conversion to a given *value-type* to succeed at run-time, the value of the source argument must be a reference to an object that was previously created by boxing a value of that *value-type*. If the source argument is null or a reference to an incompatible object, an `InvalidCastException` is thrown.

e.g.

```
using System;
```

```
class Test
{
    static void Main() {
        Console.WriteLine(3.ToString());
    }
}
```

calls the object-defined `ToString` method on an integer literal. The example

```
class Test
{
    static void Main() {
        int i = 1;
        object o = i; // boxing
        int j = (int) o; // unboxing
    }
}
```

An `int` value can be converted to `object` and back again to `int`.

This example shows both boxing and unboxing. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box. Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location.

2.5 Operators

An operator is a symbolic representation of an operation. Operators are used to compute and compare values, and test multiple conditions. An expression is a combination of variables, constants and operators written according to some rules. An expression evaluates to a value that can be assigned to variables and can also be used wherever that value can be used.

Individual constant, variables, array elements function references can be joined together by various operators to form expressions. C# includes a large number of operators, which fall into several different categories. In this section we examine some of these categories in detail. Specifically, we will see how arithmetic operators; unary operators, relational and logical operators, assignment operators and the conditional operator are used to form expressions.

The data items that operators act upon are called operands. Some operators require two operands, while others act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands.



Note An operator is a member that defines the meaning of an expression operator that can be applied to instances of the class.

Operators are declared using operator-declarations. There are three categories of operators: Unary operators, binary operators, and conversion operators.

The following rules apply to all operator declarations:

- An operator declaration must include both a public and a static modifier, and is not permitted to include any other modifiers.
- The parameter(s) of an operator must be value parameters. It is an error to for an operator declaration to specify ref or out parameters.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

All types referenced in an operator declaration must be at least as accessible as the operator itself.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus, the new modifier is never required, and therefore never permitted, in an operator declaration.

For all operators, the operator declaration includes a block which specifies the statements to execute when the operator is invoked. The block of an operator must conform to the rules for value-returning methods.

Operator Precedence and Associativity

When an expression contains multiple operators, the precedence of the operators control the order in which the individual operators are evaluated. For example, the expression $x + y * z$ is evaluated as $x + (y * z)$ because the $*$ operator has higher precedence than the $+$ operator. The precedence of an operator is established by the definition of its associated grammar production.



Example: An additive-expression consists of a sequence of multiplicative-expressions separated by + or - operators, thus giving the + and - operators lower precedence than the *, /, and % operators.

The following table summarizes all operators in order of precedence from highest to lowest:

Category	Operators
Primary	(x) x.y f(x) a[x] x++ x-- newtypeof sizeof checked unchecked
Unary	+ - ! ~ ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&&
Logical XOR	^
Logical OR	
Conditional AND	&&&
Conditional OR	
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= =

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are left-associative, meaning that operations are performed from left to right. For example, $x + y + z$ is evaluated as $(x + y) + z$.
- The assignment operators and the conditional operator (?:) are right-associative, meaning that operations are performed from right to left. For example, $x = y = z$ is evaluated as $x = (y = z)$.
- Precedence and associativity can be controlled using parentheses. For example, $x + y * z$ first multiplies y by z and then adds the result to x , but $(x + y) * z$ first adds x and y and then multiplies the result by z .



Task Answer the following questions:

1. Examine what is meant by operators' precedence and associativity?
2. What are the diverse classes of operators available in C#?

2.5.1 Unary Operators

The following rules apply to unary operator declarations, where T denotes the class or struct type that contains the operator declaration:

- A unary +, -, !, or ~ operator must take a single parameter of type T and can return any type.

Notes

- A unary ++ or -- operator must take a single parameter of type T and must return type T.
- A unary true or false operator must take a single parameter of type T and must return type bool.

The signature of a unary operator consists of the operator token (+, -, !, ~, ++, --, true, or false) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The true and false unary operators require pair-wise declaration. An error occurs if a class declares one of these operators without also declaring the other.



Did u know? **What are the properties of unary + operator?**

Unary + operators are predefined for all numeric types. The result of a unary + operation on a numeric type is just the value of the operand.

2.5.2 Binary Operators

A binary operator must take two parameters, at least one of which must be of the class or struct type in which the operator is declared. A binary operator can return any type.

The signature of a binary operator consists of the operator token (+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, or <=) and the types of the two formal parameters. The return type is not part of a binary operator's signature, nor are the names of the formal parameters.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the sametype for each parameter. The following operators require pair-wise declaration:

- operator == and operator !=
- operator > and operator <
- operator >= and operator <=

2.5.3 Conversion Operators

A conversion operator declaration introduces a user-defined conversion which augments the pre-defined implicit and explicit conversions.

A conversion operator declaration that includes the implicit keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments.

A conversion operator declaration that includes the explicit keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator. A class or struct is permitted to declare a conversion from a source type S to a target type T provided all of the following are true:

- S and T are different types.
- Either S or T is the class or struct type in which the operator declaration takes place.

- Neither S nor T is object or an interface-type.
- T is not a base class of S, and S is not a base class of T.

From the second rule it follows that a conversion operator must either convert to or from the class or struct type in which the operator is declared.



Example: It is possible for a class or struct type C to define a conversion from C to int and from int to C, but not from int to bool.

It is not possible to redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert from or to object because implicit and explicit conversions already exist between object and all other types. Likewise, neither of the source and target types of a conversion can be a base type of the other, since a conversion would then already exist.

User-defined conversions are not allowed to convert from or to interface-types. This restriction in particular ensures that no user-defined transformations occur when converting to an interface-type, and that a conversion to an interface-type succeeds only if the object being converted actually implements the specified interface-type.

The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) The implicit or explicit classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot declare both an implicit and an explicit conversion operator with the same source and target types.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

Consider the following code.

```
public struct Digit
{
    byte value;
public Digit(byte value)
{
    if (value < 0 || value > 9) throw new ArgumentException();
    this.value = value;
}
public static implicit operator byte(Digit d)
{
    return d.value;
}
public static explicit operator Digit(byte b)
{
    return new Digit(b);
}
}
```

Notes

Here, the conversion from Digit to byte is implicit because it never throws exceptions or loses information, but the conversion from byte to Digit is explicit since Digit can only represent a subset of the possible values of a byte.

2.5.4 Postfix Increment and Decrement Operators

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a get and a set accessor. If this is not the case, a compile-time error occurs.

Unary operator overload resolution is applied to select a specific operator implementation. Predefined ++ and -- operators exist for the following types: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, and any enum type. The predefined ++ operators return the value produced by adding 1 to the operand, and the predefined -- operators return the value produced by subtracting 1 from the operand.

The run-time processing of a postfix increment or decrement operation of the form x++ or x-- consists of the following steps:

If x is classified as a variable:

- x is evaluated to produce the variable.
- The value of x is saved.
- The selected operator is invoked with the saved value of x as its argument.
- The value returned by the operator is stored in the location given by the evaluation of x.
- The saved value of x becomes the result of the operation.

If x is classified as a property or indexer access:

- The instance expression (if x is not static) and the argument list (if x is an indexer access) associated with x are evaluated, and the results are used in the subsequent get and set accessor invocations.
- The get accessor of x is invoked and the returned value is saved.
- The selected operator is invoked with the saved value of x as its argument.
- The set accessor of x is invoked with the value returned by the operator as its value argument.
- The saved value of x becomes the result of the operation.

The ++ and -- operators also support prefix notation. The result of x++ or x-- is the value of x before the operation, whereas the result of ++x or --x is the value of x after the operation. In either case, x itself has the same value after the operation.



Note An operator ++ or operator -- implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

2.5.5 New Operator

Notes

The new operator is used to create new instances of types. There are three forms of new expressions:

- Object creation expressions are used to create a new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The new operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when new is used to create instances of value types.



Example:

```
struct SampleStruct
{
    public int x;
    public int y;
    public SampleStruct(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

class SampleClass
{
    public string name;
    public int id;
    public SampleClass() {}
    public SampleClass(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
}

class ProgramClass
{
    static void Main()
    {
        // Create objects using default constructors:
```

Notes

```

SampleStruct Location1 = new SampleStruct();
SampleClass Employee1 = new SampleClass();
// Display values:
Console.WriteLine("Default values:");
Console.WriteLine("  Struct members: {0}, {1}",
    Location1.x, Location1.y);
Console.WriteLine("  Class members: {0}, {1}",
    Employee1.name, Employee1.id);
// Create objects using parameterized constructors:
SampleStruct Location2 = new SampleStruct(10, 20);
SampleClass Employee2 = new SampleClass(1234, "Cristina Potra");
// Display values:
Console.WriteLine("Assigned values:");
Console.WriteLine("  Struct members: {0}, {1}",
    Location2.x, Location2.y);
Console.WriteLine("  Class members: {0}, {1}",
    Employee2.name, Employee2.id);
}
}
/*
Output:
Default values:
  Struct members: 0, 0
  Class members: , 0
Assigned values:
  Struct members: 10, 20
  Class members: Cristina Potra, 1234
*/

```

2.5.6 Arithmetic Operators

There are five arithmetic operators in C#. They are

Operator	Action
+	addition
-	Subtraction
*	multiplication
/	division
%	remainder after integer division

The % operator is sometimes referred to as the modulus operator.

There is no exponentiation operator in C#. You can write your own function to compute exponential value.

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined by the computer's character set.) The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as integer division. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-pointing quotient.

Suppose that a and b are integer variables whose values are 8 and 4, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	12
a - b	4
a * b	32
a / b	2
a % b	0



Note Notice the truncated quotient resulting from the division operation, since both operands represent integer quantities. Also, notice the integer remainder resulting from the use of the modulus operator in the last expression.

Now suppose that a1 and a2 are floating-point variables whose values are 14.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a1 + a2	16.5
a1 - a2	12.5
a1 * a2	29.0
a1 / a2	7.25

Finally, suppose that x1 and x2 are character-type variables that represent the character M and U, respectively. Some arithmetic expressions that make use of these variables are shown below, together with their resulting values (based upon the ASCII character set).

$$x1 + x2 = 162$$

$$x1 + x2 + '5' = 215$$



Note M is encoded as (decimal) 77, U is encoded as 85, and 5 is encoded as 53 in the ASCII character set.

Notes

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation toward zero; i.e., the resultant will always be smaller in magnitude than the true quotient.

Here is an illustration of the results that are obtained with floating-point operands having different signs. Let y1 and y2 be floating-point variables whose assigned values are 0.70 and 3.50. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
y1+y2	2.72
y1-y2	-4.28
y/y2	-0.2728

Operands that differ in type may undergo type conversion before the expression takes on its final value. In general, the final result will be expressed in the highest precision possible, consistent with the data type of the operands. The following rules apply when neither operand is unsigned.

If both operands are floating-point types whose precisions differ (e.g., a float and a double), the lower-precision operand will be converted to the precision of the other operand, and the result will be expressed in this higher precision. Thus, an operation between a float and double will result in a double; a float and a long double will result in a long double; and a double and a long double will result in a long double.

If one operand is a floating-point type (e.g., float, double or long double) and the other is a char or an int (including short int or long int), the char or int will be converted to the floating-point type and the result will be expressed as such. Hence, an operation between an int and a double will result in a double.

If neither operand is a floating-point type but one is long int, the other will be converted to long int and the result will be long int. Thus, an operation between a long int and an int will result in a long int.

If neither operand is a floating-point type or a long int, then both operands will be converted to int (if necessary) and the result will be int. Thus, an operation between a short into and an int will result in an int.

2.5.7 Relational Operators

Relational operators evaluate to true or false.

Operator	Description	Example	Explanation
==	Evaluates whether the operands are equal.	A==b	Returns true if the values are equal and false otherwise
!=	Evaluates whether the operands are not equal	a!=y	Returns true if the values are not equal and false otherwise
>	Evaluates whether the left operand is greater	a>b	Returns true if a is greater than b and false than the right operand
<	Evaluates whether the left operand is less than	a<b	Returns true if a is greater than or equal the right operand to b and false otherwise
>=	Evaluates whether the left operand is greater	a>=b	Returns true if a is greater than or equal to the right operand or equal to b and false otherwise
<=	Evaluates whether the left operand is less than	A<=b	Returns true if a is less than or equal to or equal to the right operand b and false otherwise.

2.5.8 Logical Operators

Use logical operators to combine the results of Boolean expressions.

Operator	Description	Example	Explanation
&&	Evaluates to true, if both the	<code>a>6&& y<20</code>	The result is true if condition 1 (<code>a>6</code>) conditions evaluate to true, and condition 2 (<code>y<20</code>) are both true. False otherwise If one of them is false, the result is false.
	Evaluate to true, if at least	<code>a>6 y < 20</code>	The result is true if either condition1 one of the conditions (<code>a>6</code>) and condition2 (<code>y<20</code>) or both evaluates to true and false if evaluate to true. If both the conditions none of the conditions are false, the result is false evaluate to true.

These operators (&&, ||) appear to be similar to the bit-wise & and | operators, except that they are limited to Boolean expressions only. However, the difference lies in the way these operators work. In the bit-wise operators, both the expressions are evaluated. This is not always necessary since:

false & a would always result in false

true | a would always result in true

Short circuit operators do not evaluate the second expression if the result can be obtained by evaluating the first expression alone.



Example: `a < 6 && y > 20`

The second condition (`b>20`) is skipped if the first condition is false, since the entire expression will anyway, be false. Similarly with the || operator, if the first condition evaluates to true, the second condition is skipped as the result, will anyway, be true. These operators, && and ||, are therefore, called short circuit operators.

If you want both the conditions to be evaluated irrespective of the result of the first condition, then you need to use bit-wise operators.



Task Answer the following question:

- Write a C# statement to find out the remainder when 23 is divided by 11.
- Evaluate the expression `a>5 && b<2` when `a=3` and `b=1`
- Define the term expression and explain the various types of expression in c#.
- Evaluate the expression `a%2==0 || a==1` when `a=4`.
- Evaluate `(a==2)&&(b==2) || (c>3)` when `a=2,b=2,c=2`.

2.5.9 Assignment Operators

Operators that store some value to a variable are known as assignment operators. Following are the assignment operators available in C#.

Operator	Description	Example	Explanation
=	Assign the value of the right operand	<code>a = b</code>	Assigns the value of b to a to the left
+=	Adds the operands and assigns the result	<code>a +=b</code>	Adds the of b to a to the left operand. The expression could also be written as <code>a = a+b</code>

Contd...

Notes

-=	Subtracts the right operand from the left	a -=b	Subtracts b from a operand and stores the result in the left Equivalent to a = a-b operand
=	Multiplies the left operand by the right	a=b	Multiplies the values a and b and stores operand and stores the result in the left the result in a operand Equivalent to a = a*b
/=	Divides the left operand by the right	a/=b	Divides a by b and stores the result in a operand and stores the result in the left Equivalent to a = a/b operand
%=	Divides the left operand by the right	a%=b	Divides a by b and stores the remainder operand and stores the remainder in a the left operand Equivalent to a = x%y

Any of the operators used as shown below:

A <operator>=y

can also be represented as

a=a <operator> b

that is, b is evaluated before the operation takes place.

You can also assign values to more than one variable at the same time. The assignment will take place from the right to the left.



Example: a = b = 0;

In the example given above, first b will be initialized and then a will be initialized.

2.5.10 Increment and Decrement Operators

The increment operator, ++, adds 1 to the existing value of the operand. It can be used in two ways - as a prefix and as a suffix. As prefix it takes the following form.

++var;

In this form the value of the variable is first incremented and then used in the expression as illustrated below:

```
var1=20;
var2 = ++var1;
```

This code is equivalent to the following set of codes:

```
var1=20;
var1 = var1+1;
var2 = var1;
```

In the end, both variables var1 and var2 store value 21.

The increment operator ++ can also be used as a postfix operator, in which the operator follows the variable.

var++;

In this case the value of the variable is used in the expression and then incremented as illustrated below:

```
var1 = 20;
var2 = var1++;
```

The equivalent of this code is:

```
var1 = 20;
var2=var1;
var1 = var1 + 1;          // Could also have been written as var1 += 1;
```

In this case, variable var1 has the value 21 while var2 remains set to 20.

In a similar fashion, the decrement operator can also be used in both the prefix and postfix forms.

If the operator is to the left of the expression, the value of the expression is modified before the assignment. Conversely, when the operator is to the right of the expression, the C++ statement, `var2 = var1++`; the original of var1 is assigned to var2. In the statement, `var2 = ++var1++`; the original value of var1 is assigned to var2. In the statement, `var2 = ++var1`; the incremented value of var1 is assigned to var2. The operators, '++' and '--', are best used in simple expressions like the ones shown.

2.5.11 Conditional Operator

C# has a ternary operator - that takes three operands to work - called conditional operator.

Operator	Description	Example	Explanation
(condition) va11, va12	Evaluates to va11 if the condition returns true and va12 if the condition returns false	a = (b>c) ? b:c	A is assigned the value in b, if b is greater than c, else a is assigned the value of c.

This example finds the maximum of two given numbers.

```
If (num1 > num2)
{
imax = num1;
}
else
{
imax = num2;
}
```

In the above program code, we determine whether num1 is greater than num2. The variable, imax is assigned the value, num1, if the expression, (num1 >num2), evaluates to true, and the value, num2, if the expression evaluates to false. The above program code can be modified using the conditional operator as:

```
Imax = (num1 > num2) ? num1 : num2;
```

The ?: Operator is called the ternary operator since it has three operands.

Notes



Task Answer the following questions:

1. Write the following expressions in shortcut notation.
 - (a) $a=a/(n+1)$
 - (b) $b=b*5$
2. Write an expression using conditional operator to print the highest number among a and b.
3. Evaluate the following expression. $a+=3>10?10:1;$
4. Evaluate the following expression:- $b=++a + a++;$ if $a=1$

2.5.12 Bitwise Operators

Data is stored internally in binary format (in the form of bits). A bit can have a value of one or zero. Eight bits form a byte. The following table displays the binary representation of digits 0 to 8.

Decimal	Binary	Equivalent
0		00000000
1		00000001
2		00000010
3		00000011
4		00000100
5		00000101
6		00000110
7		00000111
8		00001000

A bitwise operator works on a data item at the bit level.

Shift operator's work on individual bits in a byte. Using the shift operator involves moving the bit pattern left or right. You can use them only on integer data type and not on the char, bool, float, or double data types.

Operator	Description	Example	Explanation
>>	Shifts bits to the right, filling sign bit at the left	$a=10 \gg 3$	The result of this is 10 divided by 23. An explanation follows.
<<	Shifts bits to the left, filling zeros at the right	$a=10 \ll 3$	The result of this is 10 multiplied by 23. An explanation follows.

Shifting Positive Numbers

If the int data type occupies four bytes in the memory, the rightmost eight bits of the number 10 are represented in binary as

0 0 0 0 1 0 1 0

When you do a right shift by 3 ($10 \gg 3$), the result is

0 0 0 0 0 0 0 1

$10/2^3$, which is equivalent to 1.

When you do a left shift by 3 ($10 \ll 3$), the result is

0 1 0 1 0 0 0 0

$10 * 2^3$, which is equivalent to 80

Shifting Negative Numbers

For negative numbers, the unused bits are initialized to 1. Therefore, -10 is represented as:

Operator	Description	Example	Explanation
& (AND) combination	Evaluates to a binary value after a bit-wise AND on the operands	a & b	AND results in a 1 if both the bits are 1, any other results in a 0
! (OR) combination	Evaluates to binary value after a bit-wise OR on the two operands	a ! b	OR results in a 0 when both the bits are 0, any other combination results in a 1.
^ (XOR) if	Evaluates to a binary value after a bit-wise XOR on the two operands	a ^ b	XOR results in a 0 if both the bits are of the same value and 1 the bits have different values.
~	Converts all 1 bits to 0s and all 0 bits (inversion)	to 1s	Example given below.

In the example shown in the table of bit wise operators, a and b are integers and can be replaced with expressions that give a true or false (bool) result.



Example: When both the expressions evaluate to true, the result of using the & operator is true. Otherwise, the result is false.

The ~ Operator

If you use the ~ operator, all the 1s in the bits are converted to 0s and vice versa. For example, 10011001 would become 01100110.

2.5.13 Checked and Unchecked Operators

The checked and unchecked operators are used to control the overflow checking context for integral-type arithmetic operations and conversions.

The checked operator evaluates the contained expression in a checked context, and the unchecked operator evaluates the contained expression in an unchecked context. A checked-expression or unchecked-expression corresponds exactly to a parenthesized-expression, except that the contained expression is evaluated in the given overflow checking context.

Notes

The overflow checking context can also be controlled through the checked and unchecked statements. The following operations are affected by the overflow checking context established by the checked and unchecked operators and statements:

- The predefined ++ and -- unary operators, when the operand is of an integral type.
- The predefined - unary operator, when the operand is of an integral type.
- The predefined +, -, *, and / binary operators, when both operands are of integral types.
- Explicit numeric conversions from one integral type to another integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a checked context, if the operation is a constant expression, a compile-time error occurs. Otherwise, when the operation is performed at run-time, an `OverflowException` is thrown.
- In an unchecked context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

When a non-constant expression (an expression that is evaluated at run-time) is not enclosed by any checked or unchecked operators or statements, the effect of an overflow during the run-time evaluation of the expression depends on external factors (such as compiler switches and execution environment configuration).



Note The effect is however guaranteed to be either that of a checked evaluation or that of an unchecked evaluation.

For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow checking context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

Consider the following code.

```
class TestClass
{
    static int x = 1000000;
    static int y = 1000000;

    static int FunctionOne()
    {
        return checked(x * y);        // Throws OverflowException
    }

    static int FunctionTwo()
    {
        return unchecked(x * y);     // Returns -727379968
    }

    static int FunctionThree()
    {
```

```
return x * y;           // Depends on default
}
}
```

Here, no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the `FunctionOne()` method throws an `OverflowException`, and the `FunctionThree()` method returns `-727379968` (the lower 32 bits of the out-of-range result). The behavior of the `FunctionThree()` method depends on the default overflow checking context for the compilation, but it is either the same as `FunctionOne()` or the same as `FunctionTwo()`.

Consider the following code.

```
class TestClass
{
    const int x = 1000000;
    const int y = 1000000;
    static int FunctionOne()
    {
        return checked(x * y);           // Compile error, overflow
    }
    static int FunctionTwo()
    {
        return unchecked(x * y);       // Returns -727379968
    }
    static int FunctionThree()
    {
        return x * y;                   // Compile error, overflow
    }
}
```

Here, the overflows that occur when evaluating the constant expressions in `FunctionOne()` and `FunctionThree()` cause compile-time errors to be reported because the expressions are evaluated in a checked context. An overflow also occurs when evaluating the constant expression in `FunctionTwo()`, but since the evaluation takes place in an unchecked context, the overflow is not reported.

The checked and unchecked operators only affect the overflow checking context for those operations that are textually contained within the “(” and “)” tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. Consider the following code.

```
class TestClass
{
    static int Multiply(int x, int y)
    {
        return x * y;
    }
}
```

Notes

```
static int FunctionOne()
{
    return checked(Multiply(1000000, 1000000));
}
}
```

Here, the use of checked in FunctionOne() does not affect the evaluation of x * y in Multiply(), and x * y is therefore evaluated in the default overflow checking context.

The unchecked operator is convenient when writing constants of the signed integral types in hexadecimal notation. Consider the following code.

```
class TestClass
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);
    public const int HighBit = unchecked((int)0x80000000);
}
```

Both of the hexadecimal constants above are of type uint. Because the constants are outside the int range, without the unchecked operator, the casts to int would produce compile-time errors.

Self Assessment

Fill in the blanks:

- 12. The signature of an operator must differ from the signatures of all other declared in the same class.
- 13. Operands that differ in type may undergo before the expression takes on its final value.

2.6 Expressions

Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators include +, -, *, /, and new. Examples of operands include literals, fields, local variables, and expressions.

There are three types of operators:

- **Unary operators:** The unary operators take one operand and use either prefix notation (such as -x) or postfix notation (such as x++).
- **Binary operators:** The binary operators take two operands and all use infix notation (such as x + y).
- **Ternary operator:** Only one ternary operator, ?;, exists. The ternary operator takes three operands and uses infix notation (c? x: y).

The order of evaluation of operators in an expression is determined by the precedence and associativity of the operators. Certain operators can be overloaded. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

Object Creation Expressions

Notes

An object-creation-expression is used to create a new instance of a class-type or a value-type. The type of an object-creation-expression must be a class-type or a value-type. The type cannot be an abstract class-type.



Task Analyze why the optional argument-list is permitted only if the type is a class-type or a struct-type.

Array Creation Expressions

An array-creation-expression is used to create a new instance of an array-type. An array creation expression of first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression `new int[10, 20]` produces an array instance of type `int[,]`, and the array creation expression `new int[10][,]` produces an array of type `int[][,]`. Each expression in the expression list must be of type `int`, `uint`, `long`, or `ulong`, or of a type that can be implicitly converted to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second form, the rank of the specified array type must match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

```
new int[,] {{0, 1}, {2, 3}, {4, 5}};
```

exactly corresponds to

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}};
```

The result of evaluating an array creation expression is classified as a value, namely a reference to the newly allocated array instance. The run-time processing of an array creation expression consists of the following steps:

- The dimension length expressions of the expression-list are evaluated in order, from left to right. Following evaluation of each expression, an implicit conversion to type `int` is performed. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
- The computed values for the dimension lengths are validated. If one or more of the values are less than zero, an `IndexOutOfRangeException` is thrown and no further steps are executed.
- An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, an `OutOfMemoryException` is thrown and no further steps are executed.
- All elements of the new array instance are initialized to their default values.

Notes

- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer—in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array must be manually initialized. For example, the statement

```
int[][] a = new int[100][];
```

creates a single-dimensional array with 100 elements of type `int[]`. The initial value of each element is null. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

```
int[][] a = new int[100][5];           // Error
```

is an error. Instantiation of the sub-arrays must instead be performed manually, as in

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

When an array of arrays has a “rectangular” shape, that is when the sub-arrays are all of the same length, it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects—one outer array and 100 sub-arrays. In contrast,

```
int[,] = new int[100, 5];
```

creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement.

Delegate Creation Expressions

A delegate-creation-expression is used to create a new instance of a delegate-type. The argument of a delegate creation expression must be a method group or a value of a delegate-type. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate.



Note If the argument is a value of a delegate-type, it identifies a delegate instance of which to create a copy.

The compile-time processing of a delegate-creation-expression of the form `new D(E)`, where `D` is a delegate-type and `E` is an expression, consists of the following steps:

If *E* is a method group:

- If the method group resulted from a base-access, an error occurs.
- The set of methods identified by `E` must include exactly one method with precisely the same signature and return type as those of `D`, and this becomes the method to which the newly created delegate refers. If no matching method exists, or if more than one matching methods exists, an error occurs. If the selected method is an instance method, the instance expression associated with `E` determines the target object of the delegate.

- As in a method invocation, the selected method must be compatible with the context of the method group: If the method is a static method, the method group must have resulted from a simple-name or a member-access through a type. If the method is an instance method, the method group must have resulted from a simple-name or a member-access through a variable or value. If the selected method does not match the context of the method group, an error occurs.
- The result is a value of type D, namely a newly created delegate that refers to the selected method and target object.

Otherwise, if E is a value of a delegate-type:

- The delegate-type of E must have the exact same signature and return type as D, or otherwise an error occurs.
- The result is a value of type D, namely a newly created delegate that refers to the same method and target object as E.
- Otherwise, the delegate creation expression is invalid, and an error occurs.

The run-time processing of a delegate-creation-expression of the form `new D(E)`, where D is a delegate-type and E is an expression, consists of the following steps:

If E is a method group:

- If the method selected at compile-time is a static method, the target object of the delegate is null. Otherwise, the selected method is an instance method, and the target object of the delegate is determined from the instance expression associated with E.
- The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.
- If the instance expression is of a reference-type, the value computed by the instance expression becomes the target object. If the target object is null, a `NullReferenceException` is thrown and no further steps are executed.
- If the instance expression is of a value-type, a boxing operation is performed to convert the value to an object, and this object becomes the target object.
- A new instance of the delegate type D is allocated. If there is not enough memory available to allocate the new instance, an `OutOfMemoryException` is thrown and no further steps are executed.
- The new delegate instance is initialized with a reference to the method that was determined at compile-time and a reference to the target object computed above.

If E is a value of a delegate-type:

- E is evaluated. If this evaluation causes an exception, no further steps are executed.
- If the value of E is null, a `NullReferenceException` is thrown and no further steps are executed.
- A new instance of the delegate type D is allocated. If there is not enough memory available to allocate the new instance, an `OutOfMemoryException` is thrown and no further steps are executed.
- The new delegate instance is initialized with references to the same method and object as the delegate instance given by E.

The method and object to which a delegate refers are determined when the delegate is instantiated and then remain constant for the entire lifetime of the delegate. In other words, it is not possible to change the target method or object of a delegate once it has been created.

Notes



Caution It is not possible to create a delegate that refers to a constructor, property, indexer, or user-defined operator.

As described above, when a delegate is created from a method group, the signature and return type of the delegate determine which of the overloaded methods to select. Consider the following code.

```
delegate double DoubleFunc(double x);
class One
{
    DoubleFunc f = new DoubleFunc(Square);
    static float Square(float x)
    {
        return x * x;
    }
    static double Square(double x)
    {
        return x * x;
    }
}
```

Here, the One.f field is initialized with a delegate that refers to the second Square method because that method exactly matches the signature and return type of DoubleFunc. Had the second Square method not been present, a compile-time error would have occurred.

Self Assessment

Fill in the blanks:

- 12. A delegate-creation-expression is used to create a new of a delegate-type.
- 13. An array-creation-expression is used to create a new instance of an



Caselet

Microsoft Lines up Big Plans for Hyderabad Centre

MICROSOFT Corporation is on track to invest \$75 million in three years, starting last year, in its Hyderabad product development centre.

“We may even invest more in the R&D centre,” said Mr S.S. Somasegar, Corporate Vice- President, Windows Engineering Solutions & Services, Microsoft Corporation.

Without divulging any figures, he said, the investment would be mostly on people. The company is also on track to recruit 300 people for the centre by end 2003, from the present 150, he added.

Based at the company’s headquarters, Redmond, US, Mr Somasegar is responsible for the overall project and release management of the Microsoft Windows family of products.

Contd...

He also oversees the company's R&D centre in Hyderabad, which focuses more on delivering Net-related solutions.

In an informal chat with newsmen here, Mr Somasegar said that the company plans to leverage more on the Indian talent.

"If you ask whether we have leveraged the Indian talent compared to other firms, the answer is no. However, in future, we intend to leverage a lot more the vast pool of Indian resources. We are trying to figure out as to how we can use the Indian resources, not only in product development, but in other areas including consulting and in the company's internal applications."

According to Mr Somasegar, a few days ago, the Hyderabad centre rolled out VJ# (Visual J#), a development tool for Java-language developers building applications and services on the Microsoft .NET framework.

The VJ# was missing in Visual Studio (Visual Studio .NET empowers developers to design broad-reach Web applications for any device and on any platform).

Microsoft now provides the VJ# as an add-on to Visual Studio, he added. The launch of Visual J# .NET signals the completion and availability of all the Microsoft programming languages within Visual Studio .NET, including Visual C++ .NET, Visual C# .NET and Visual Basic .NET, he said.

Mr Somasegar said that Microsoft intends to make Hyderabad "a critical and the most important product centre for the company".

"We are working seriously to achieve that," he added.

2.7 Summary

- C# data type is unified. A value of any type can be treated as an object.
- Values of reference type can be treated as objects by simply viewing the values as type object.
- Values of value types can be treated as objects as well as mere values by performing boxing and unboxing operations
- C# provides two categories of value types – struct type or enumeration type.
- If no explicit initialization of variables is performed by the programmers, C# places default values to the variables.
- An operator is a symbol that tells C# to perform some of the operations, actions on one or more operands.
- Value can be assigned to more than one variable at the same time. Example a=b=0.
- When an expression contains multiple operators, the precedence of the operator controls the order in which the individual operator is evaluated.
- Data types can be converted from one form to another by typecasting.
- Unlike instance variables class variables are declared with the word static.
- Variables can hold either value types or reference types, or they can be pointers.
- Verbatim strings are defined by enclosing the required strings in the characters @" and".
- Boxing and unboxing enables a unified view of the type system where in a value of any type can ultimately be treated as an object.

Notes

- The possible value for Byte data type ranging from 0 to 255.
- If no explicit initialization of variables is performed by the programmers, C# places default values to the variables.

2.8 Keywords

Boxing: To convert value type data type in to reference type data type is called Boxing.

Datum: A datum, singular of data, is a value in the context of processing.

Explicit Type Conversions: The type conversions that are performed with the programmer's intervention are called as explicit conversions.

Implicit Type Conversion: The type conversions that are performed automatically by C# compiler are called implicit conversions.

Instance Variable: Variables that are declared inside the class declaration and outside the scope of any method are called instance variables.

Operators: C# provides a large set of operators, which are symbols that specify which operations to perform in an expression. C# predefines the usual arithmetic and logical operators, as well as a variety of others.

Reference Type: Variable, which holds a reference (pointer) to data rather than containing the actual data. Reference types include array, class, delegate, and interface.

Static Variables: Variables that are declared with the word static are called static variables where in all the objects of the class shares the only one copy of the variables.

Unboxing: To convert reference data type to value type data type is called Unboxing.

Value Type: .NET type containing actual data rather than a reference to data stored elsewhere in memory. Simple value types include Boolean, character, decimal, floating point, and integer.

Variable: It is a storage location in the memory and like a container that holds the value.

2.9 Review Questions

1. Can there be a working C# program without a single Main method? Justify your answer.
2. Differentiate between instance and static variables using a suitable example.
3. What are the different types of data types available in C#?
4. Explain the concept of reference parameter and output parameters with suitable examples.
5. Substantiate what is the .net framework system type of the data types available in C#?
6. Examine what is the purpose of boxing and unboxing conversions?
7. Explain the concept of reference parameter and output parameters with suitable examples.
8. What is the .net framework system type of the data types available in C#?
9. "Value can be assigns to more than one variable at the same time". Explain this statement through a program.
10. "If no explicit initialization of variables is performed by the programmers, C# places default values to the variables". Explain this statement through a program.

Answers: Self Assessment**Notes**

- | | |
|---------------------|------------------|
| 1. variable | 2. parameterless |
| 3. struct | 4. sub-classes |
| 5. size | 6. char |
| 7. integral | 8. object |
| 9. false | 10. output |
| 11. valid | 12. operators |
| 13. type conversion | 14. instance |
| 15. array-type | |

2.10 Further Readings**Books**

Beginning C: From Novice to Professional, Fourth Edition by Ivor Horton.

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Expert C# 2005 Business Objects, Second Edition by Rockford Lhotka.

Pro C# 2005 and the .NET 2.0 Platform, Third Edition by Andrew Troelsen.

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)

**Online links**

<http://csharp.net-tutorials.com/>

<http://www.csharp-station.com/Tutorials/Lesson01.aspx>

Unit 3: Decision Making and Looping

CONTENTS

Objectives

Introduction

3.1 Statements

3.1.1 Labeled Statement

3.1.2 End Points and Reachability

3.2 Blocks

3.3 Branching Statements

3.3.1 If-Statement and If-Else-If Statement

3.3.2 The Switch Statement

3.4 Looping Statements

3.4.1 The While Statement

3.4.2 The do Statement

3.4.3 The for Statement

3.5 Jump Statements

3.6 The Checked and Unchecked Statements

3.7 Summary

3.8 Keywords

3.9 Review Questions

3.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the If Statement
- Describe the If-Else-If statement
- Demonstrate the While Loop
- Recognize Do While loop
- Explain For loop

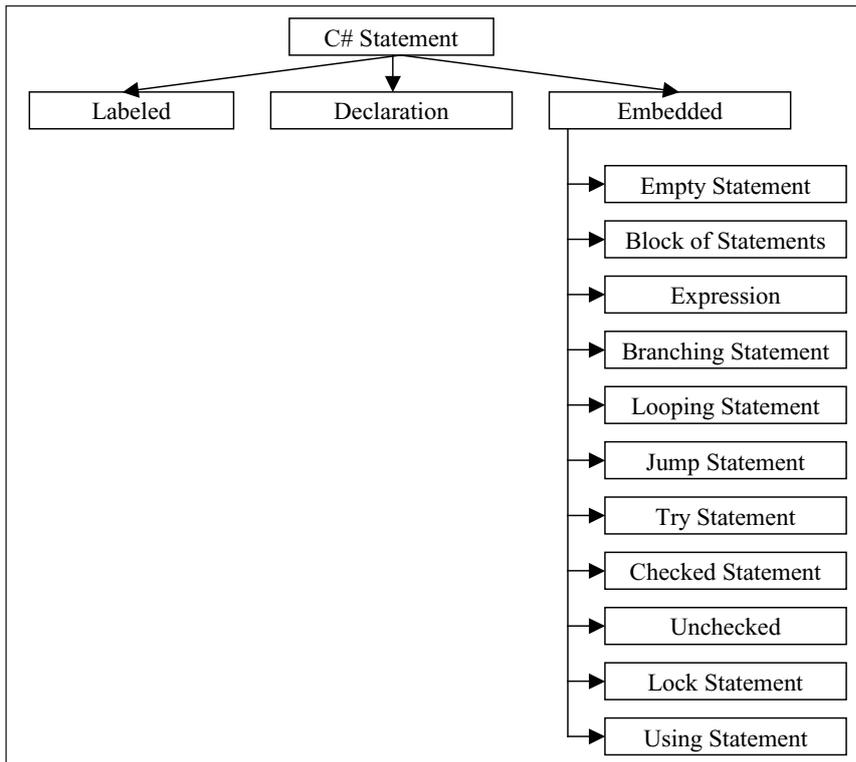
Introduction

This unit will introduce you to the various control statements in C#, blocks, statement lists, the various looping statements such as the while statement, the do statement, for statement etc. in which the looping allows repeated execution of statements. Also you will be introduced to jump statements, break statement, go to statements and lastly the checked and unchecked statements.

3.1 Statements

In the context of a programming language a complete sentence made up of reserved and user-defined words is called a statement. Statements serve a variety of purposes. Some statements indicate execution actions while some simply direct the compilation process. Just as a complete story consists of a number of coherent sentences, a complete program is made up of a number of coherent statements.

C# provides a rich variety of statements that includes all the familiar statements of languages C and C++. The types of statements available in C# are presented in the following tree diagram.



3.1.1 Labeled Statement

A labeled statement is any valid statement that is identifiable by a user-defined label attached to it. The syntax of a labeled-statement is given below.

```
identifier      :      statement
```

For example, consider the following C# function.

```
int abc(int x)
{
    if (x >= 0) goto doit;
    x = -x;
    doit: return x;
}
```

In the code-snippet listed above, identifier - doit - is a label and hence - doit: return x; - is a labeled-statement.

Notes

A label can be referenced from goto statements within the scope of the label. This means that goto statements can transfer control inside blocks and out of blocks, but never into blocks.

A labeled-statement permits a statement to be prefixed by a label. Labeled statements are permitted blocks, but are not permitted as embedded statements.



Caution A labeled statement declares a label with the name given by an identifier. The scope of a label is the block in which the label is declared, including any nested blocks. It is an error for two labels with the same name to have overlapping scopes.

Labels have their own declaration space and do not interfere with other identifiers. Consider the following code.

```
int FunctionOne(int x)
{
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

The label name - x- is valid and uses the name x as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label. In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable goto statement.

3.1.2 End Points and Reachability

A statement starts at a point and ends at another point. The end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement.



Example: When control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be reachable. Conversely, if there is no possibility that a statement will be executed, the statement is said to be unreachable.

Consider the following code.

```
void FunctionOne()
{
    Console.WriteLine("This is reachable");
    goto Label;
    Console.WriteLine("This is unreachable");
Label:
    Console.WriteLine("reachable through label");
}
```

Here, the second `Console.WriteLine` invocation is unreachable because there is no possibility that the statement will be executed. A warning is reported if the compiler determines that a statement is unreachable. However, it is specifically not an error for a statement to be unreachable.

The compiler performs flow analysis according to the reachability rules defined for each statement to determine whether a particular statement or end point. The flow analysis takes into account the values of constant expressions that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

Consider the following code.

```
void FunctionOne()
{
    const int i = 1;
    if (i == 2) Console.WriteLine("This is unreachable");
}
```

Here, the boolean expression of the `if` statement is a constant expression because both operands of the `==` operator are constants. The constant expression is evaluated at compile-time, producing the value `false`, and the `Console.WriteLine` invocation is therefore considered unreachable. However, if `i` is changed to be a local variable as is listed below the statement becomes reachable.

```
void FunctionOne()
{
    int i = 1;
    if (i == 2) Console.WriteLine("This is reachable");
}
```

The `Console.WriteLine` invocation is considered reachable, even though it will in reality never be executed. The block of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined. Consider the following code.

```
void FunctionOne(int x)
{
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

In the above example, the reachability of the second `Console.WriteLine` is determined as follows:

First, because the block of the `FunctionOne` method is reachable, the first `Console.WriteLine` statement is reachable.

Next, because the first `Console.WriteLine` statement is reachable, its end point is reachable.

Next, because the end point of the first `Console.WriteLine` statement is reachable, the `if` statement is reachable.

Finally, because the boolean expression of the `if` statement does not have the constant value `false`, the second `Console.WriteLine` statement is reachable.

Notes



Notes There are two situations in which it is an error for the end point of a statement to be reachable:

- Because the switch statement does not permit a switch section to “fall through” to the next switch section, it is an error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a break statement is missing.
- It is an error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it is typically an indication that a return statement is missing.

3.2 Blocks

Multiple statements can be blocked to make them behave as a single statement. A block consists of an optional statement-list, enclosed in braces. If the statement list is omitted, the block is said to be empty. A block may contain declaration statements. The scope of a local variable or constant declared in a block extends from the declaration to the end of the block. Within a block, the meaning of a name used in an expression context must always be the same.

Consider the following code.

```

Int FunctionOne(int x)
{
    if(x < 10)
        return(2 * x);
    else
    {
        System.Console.WriteLine("Blocked statements");
        Return(x);
    }
}
    
```

Here the two statements in the else part form a single block of statements. It is treated as a single statement in the context of else.

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable. The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

A statement list consists of one or more statements written in sequence.



Task Statement lists occur in blocks and in switch-blocks. Analyze.

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable goto statement.
- The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

A block of statements having no statement is called an empty-statement. It is a do-nothing statement. Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

Self Assessment

Fill in the blanks:

1. A labeled-statement permits a statement to be by a label.
2. A label can be referenced from statements within the scope of the label.
3. The invocation is considered reachable, even though it will in reality never be executed.
4. The scope of a or constant declared in a block extends from the declaration to the end of the block.
5. If a statement can possibly be reached by execution, the statement is said to be

3.3 Branching Statements

Execution of a group of statements takes place in sequential manner in their order of appearance. However, often it is required that based on some conditional criteria a specified statement is executed while other are omitted. Jumping to a statement depending on some selection basis is termed as branching.

C# offers two broad mechanism to accomplish branching in a program.

- if-statement
- switch-statement

3.3.1 If-Statement and If-Else-If Statement

Working of an - if - statement is straightforward. The general syntax of the - if - statement is given below.

```

    If (condition)
statement1;
    else
        statement2;
```

Notes

A condition is an expression or a statement that evaluates to either of the two Boolean values - true or false.



Note The indentation shown above is not mandatory.

It has been used on purpose. It enhances the readability of the code. However, nobody stops you from writing it as any one of the forms given below.

If (condition) statement1; else statement2;

Or

If (condition) statement1;

else statement2;

Or

If (condition) statement1;

Or

If (condition)

;

else

statement2;



Example:

```
using System;
```

```
namespace ConsoleApplication1
```

```
{
```

```
    class Program
```

```
    {
```

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

```
        {
```

```
            int number;
```

```
            Console.WriteLine("Please enter a number between 0 and 10:");
```

```
            number = int.Parse(Console.ReadLine());
```

```
            if(number > 10)
```

```
                Console.WriteLine("Hey! The number should be 10 or less!");
```

```
            else
```

```
                if(number < 0)
```

```

        Console.WriteLine("Hey! The number should be 0 or more!");
    else
        Console.WriteLine("Good job!");

    Console.ReadLine();
}
}
}

```

In all the cases what the - if - statement does is described in the following points.

1. The condition is evaluated
2. If the condition is found to be true then statement1 is executed and if it turns out to be false then statement2 is executed
3. Either of the statements - statement1 and statement2 - may be missing in which case no action takes place in that portion of the - if - statement.
4. The statements - statement1 and statement2 - may be a single statement or a block of statements. Multiple statements must be blocked in the else part otherwise only the first statement shall be takes to be associated with the else part. Consider the following code snippet.

```

if(condition)
    statement1;
else
    statement2;
    statement3;

```

The indentation of the above statements suggests that if the condition is false both the statements - statement1 and statement2 - should be executed. However, as far as C# is concerned it associates only statement2 to the else part. Therefore, irrespective of the value of the condition, statement3 will always get executed. If you intend to treat both the statements - statement1 and statement2 - in the else part you must block them equivalent to a single statement as shown below.

```

if(condition)
    statement1;
else
{
    statement2;
    statement3;
}

```

5. The - else - portion of the s - if - statement is associated with the nearest preceding - if - that does not have its own - else. Consider the following code snippet.

```

if(condition1)
    if(condition2)

```

Notes

```
        statement1;
    else
        statement2;
```

The indentation in the above code seems to suggest that the - else - part should be associated with the - if - with condition1. However, since - else - is associated with the nearest preceding - if - without an - else -, actually - else - here is associated with condition2.

3.3.2 The Switch Statement

An if-else statement essentially performs a two-way branching. It is suitable in cases where one of the two given statements is to be selected for execution. Though, it can also implement multi-way branching, the code becomes very involved. Take a look at the following code snippet.

```
        if(condition1) statement1;
        else
if(condition2) statement2;
                else
if(condition3) statement3;
                        else
if(condition4) statement4;
                                else
if(condition5) statement5;
```

As the number of conditional statement increases the code becomes difficult to read and maintain.

The switch statement provides a more elegant way to handle multi-way branching. executes the statements that are associated with the value of the controlling expression. The syntax is given below.

```
switch (switch_expression)
{
case Switch_Value1:
        FunctionOne();
        break;
case Switch_Value2:
        FunctionTwo();
        break;
case Switch_Value3:
        FunctionThree();
        break;
default:
        FunctionOther();
        break;
}
```

A switch-statement consists of the keyword `switch`, followed by a parenthesized expression (called the switch expression), followed by a switch-block. The switch-block consists of zero or more switch-sections, enclosed in braces. Each switch-section consists of one or more switch-labels followed by a statement-list.

The governing type of a switch statement is established by the switch expression. If the type of the switch expression is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or an enum-type, then that is the governing type of the switch statement. Otherwise, exactly one user-defined implicit conversion must exist from the type of the switch expression to one of the following possible governing types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`.



Note If no such implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

The constant expression of each case label must denote a value of a type that is implicitly convertible to the governing type of the switch statement. A compile-time error occurs if two or more case labels in the same switch statement specify the same constant value.

There can be at most one default label in a switch statement.

A switch statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a case label is equal to the value of the switch expression, control is transferred to the statement list following the matched case label.
- If no constant matches the value of the switch expression and if a default label is present, control is transferred to the statement list following the default label.
- If no constant matches the value of the switch expression and if no default label is present, control is transferred to the end point of the switch statement.
- If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the “no fall through” rule. Consider the following code.

```
switch (i)
{
    case 0:
        CaseZero();
        break;
    case 1:
        CaseOne();
        break;
    default:
        CaseOthers();
        break;
}
```

It is a valid code because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to “fall through” to the next switch section, therefore, the following code will show error.

Notes

```
switch (i)
{
case 0:
    CaseZero();
case 1:
    CaseZeroOrOne();
default:
    CaseAny();
}
```



Did u know? **What are the uses of switch statement?**

The switch statement is like a set of if statements. It's a list of possibilities, with an action for each possibility, and an optional default action, in case nothing else evaluates to true.

When execution of a switch section is to be followed by execution of another switch section, an explicit goto case or goto default statement must be used, as shown in the following code.

```
switch (i)
{
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}
```

Moreover, multiple labels can be used in a switch-section as shown in the following code.

```
switch (i)
{
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
}
```

```

        break;
    }

```

This code is perfectly legal. It does not violate the “no fall through” rule because the labels case 2: and default: are parts of the same switch-section.

The “no fall through” rule prevents a common class of bugs that occur in C and C++ when break statements are accidentally omitted. Also, because of this rule, the switch sections of a switch statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the switch statement above can be reversed without affecting the behavior of the statement (see the code ahead).

```

switch (i)
{
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}

```

The statement list of a switch section typically ends in a break, goto case, or goto default statement, but any construct that renders the end point of the statement list unreachable is permitted.



Example: A while statement controlled by the boolean expression true is known to never reach its end point.

Likewise, a throw or return statement always transfer control elsewhere and never reaches its end point. Thus, the following code is valid.

```

switch (i)
{
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}

```

The switch_expression is not always required to be a numeric type. It can also be string type. Consider the following code.

```

void Action(string action)
{

```

Notes

```
switch (action.ToLower())
{
    case "run":
        DoRun();
        break;
    case "save":
        DoSave();
        break;
    case "quit":
        DoQuit();
        break;
    default:
        InvalidAction(action);
        break;
}
```

Like the string equality operators, the switch statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a case label constant. As illustrated by the example above, a switch statement can be made case insensitive by converting the switch expression string to lower case and writing all case label constants in lower case.

When the governing type of a switch statement is string, the value null is permitted as a case label constant.



Note A switch-block may contain declaration statements. The scope of a local variable or constant declared in a switch block extends from the declaration to the end of the switch block.

Within a switch block, the meaning of a name used in an expression context must always be the same.

The statement list of a given switch section is reachable if the switch statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a case label in the switch section.
- The switch expression is a constant value that doesn't match any case label, and the switch section contains the default label.
- A switch label of the switch section is referenced by a reachable goto case or goto default statement.

The end point of a switch statement is reachable if at least one of the following is true:

- The switch statement contains a reachable break statement that exits the switch statement.
- The switch statement is reachable, the switch expression is a non-constant value, and no default label is present.

- The switch statement is reachable, the switch expression is a constant value that doesn't match any case label, and no default label is present.



Task Answer the following question:

1. What do you understand by labeled statement?
2. What is a block? How is it executed?
3. What are the various mechanisms provided in C# to accomplish branching in a program?
4. What is the problem of "Fall Through" associated with switch-case. How it can be overcome in C#?

Self Assessment

Fill in the blanks:

6. An if-else statement essentially performs a branching.
7. The switch expression is a value that matches a case label in the switch section.
8. The switch statement is, the switch expression is a constant value that doesn't match any case label, and no default label is present.
9. A switch-statement consists of the keyword switch, followed by a parenthesized expression (called the switch expression), followed by a

3.4 Looping Statements

Very often one or a group of statements are required to be executed a specified number of times or until some condition becomes true/false. Such iterative executions are made possible by looping statements. C# offers the many looping constructs like - while statement, do-statement, for-statement and foreach statement.

3.4.1 The While Statement

The while statement has two parts - looping condition and body. The looping condition is a Boolean expression that evaluates to either true or false. The body consists of either a single statement or a block of statements, as shown below.

```
while (boolean_expression)
    statement;
```

The statement executes the statement as long as the Boolean_expression is true. Once the Boolean_expression becomes false, the statement reaches its end.

A while statement is executed as follows:

- The boolean-expression is evaluated.
- If the boolean expression yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), control is transferred to the beginning of the while statement.

Notes

- If the boolean expression yields false, control is transferred to the end point of the while statement.

Within the embedded statement of a while statement, a break statement may be used to transfer control to the end point of the while statement (thus ending iteration of the embedded statement), and a continue statement may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the while statement).

The embedded statement of a while statement is reachable if the while statement is reachable and the boolean expression does not have the constant value false.

The end point of a while statement is reachable if at least one of the following is true:

- The while statement contains a reachable break statement that exits the while statement.
- The while statement is reachable and the boolean expression does not have the constant value true.

Consider the following function that prints numbers from 1 to the input argument.

```
void PrintSeries(int x)
{
    int i = 1;
    while(i <= x)
    {
        System.Console.WriteLine("{0}", i);
        i++;
    }
}
```



Note If the Boolean_expression happens to be false at the time of executing while statement, the statement will not be executed even once.

Consider the code listed below.

```
int i = 10;
while(i < 5)
    System.Console.WriteLine("Not executed even once");
```

In this case the WriteLine statement will not be executed even once because the condition is false at the time of the entry of while statement.

For the while loop to terminate the loop-controlling condition must be modified in the body. Here is an example of infinite loop.

```
while(true);
```

3.4.2 The do Statement

The do statement executes an embedded statement at least once depending on the given Boolean_expression. The syntax is given below.

do

```
statement
```

```
while(boolean-expression);
```

The statement is executed before the Boolean_expression is evaluated. Therefore, the statement is executed at least once. A do statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the boolean-expression is evaluated. If the boolean expression yields true, control is transferred to the beginning of the do statement. Otherwise, control is transferred to the end point of the do statement.

Within the embedded statement of a do statement, a break statement may be used to transfer control to the end point of the do statement (thus ending iteration of the embedded statement), and a continue statement may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the do statement).

The embedded statement of a do statement is reachable if the do statement is reachable.

The end point of a do statement is reachable if at least one of the following is true:

- The do statement contains a reachable break statement that exits the do statement.
- The end point of the embedded statement is reachable and the boolean expression does not have the constant value true.



Example: First, here is an example of using the do while loop to sum the values of the elements in an int array. The int array here is guaranteed to have four elements, so you can avoid checking the length before starting checking its elements in do while.

Program that uses do while loop [C#]

```
class Program
{
    static void Main()
    {
        int[] ids = new int[] { 6, 7, 8, 10 };
        //
        // Use do while loop to sum numbers in 4-element array.
        //
        int sum = 0;
        int i = 0;
        do
        {
            sum += ids[i];
            i++;
        } while (i < 4);

        System.Console.WriteLine(sum);
    }
}
```

Notes

}

Output

31

3.4.3 The for Statement

The for statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions. The syntax is given below.

```
for(initializer; condition; iterator)
statement;
```

The for loop has four parts. However all the four parts are optional and they may be missing in an implementation. The initializer is a comma separated list of initial values. The condition is the looping condition. The for loop executes as long as the looping-condition remains true. The scope of a local variable declared by a for-initializer starts at the variable-declarator for the variable and extends to the end of the embedded statement. The scope includes the for-condition and the for-iterator.

- The for-condition, if present, must be a boolean-expression.
- The for-iterator, if present, consists of a list of statement-expressions separated by commas.

A for statement is executed as follows:

- If a for-initializer is present, the variable initializers or statement expressions are executed in the order they are written. This step is only performed once.
- If a for-condition is present, it is evaluated.
- If the for-condition is not present or if the evaluation yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the expressions of the for-iterator, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the for-condition in the step above.
- If the for-condition is present and the evaluation yields false, control is transferred to the end point of the for statement.

Within the embedded statement of a for statement, a break statement may be used to transfer control to the end point of the for statement (thus ending iteration of the embedded statement), and a continue statement may be used to transfer control to the end point of the embedded statement (thus executing another iteration of the for statement).

The embedded statement of a for statement is reachable if one of the following is true:

- The for statement is reachable and no for-condition is present.
- The for statement is reachable and a for-condition is present and does not have the constant value false.

The end point of a for statement is reachable if at least one of the following is true:

- The for statement contains a reachable break statement that exits the for statement.
- The for statement is reachable and a for-condition is present and does not have the constant value true.

The following code prints the series of integers from 1 to 100.

```
for(int= i; i <= 100; i++)
    System.Console.WriteLine("{0},i);
```



Example:

```
using System;
class ForLoop
{
    public static void Main()
    {
        for (int i=0; i < 20; i++)
        {
            if (i == 10)
                break;

            if (i % 2 == 0)
                continue;
            Console.Write("{0} ", i);
        }
        Console.WriteLine();
    }
}
```

The foreach Statement

The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection. Here is the syntax.

```
foreach identifier statement;
```

The identifier of a foreach statement declares the iteration variable of the statement. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a foreach statement, the iteration variable represents the collection element for which iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to assign to the iteration variable or pass the iteration variable as a ref or out parameter.

The type of the expression of a foreach statement must be a collection type (as defined below), and an explicit conversion must exist from the element type of the collection to the type of the iteration variable.

A type C is said to be a collection type if all of the following are true:

- C contains a public instance method with the signature GetEnumerator() that returns a struct-type, class-type, or interface-type, in the following called E.
- E contains a public instance method with the signature MoveNext() and the return type bool.

Notes

- E contains a public instance property named Current that permits reading. The type of this property is said to be the element type of the collection type.

The System.Array type is a collection type, and since all array types derive from System.Array, any array type expression is permitted in a foreach statement. For single-dimensional arrays, the foreach statement enumerates the array elements in increasing index order, starting with index 0 and ending with index Length - 1. For multi-dimensional arrays, the indices of the rightmost dimension are increased first.

A foreach statement is executed as follows:

- The collection expression is evaluated to produce an instance of the collection type. This instance is referred to as c in the following. If c is of a reference-type and has the value null, a NullReferenceException is thrown.
- An enumerator instance is obtained by evaluating the method invocation c.GetEnumerator(). The returned enumerator is stored in a temporary local variable, in the following referred to as e. It is not possible for the embedded statement to access this temporary variable. If e is of a reference-type and has the value null, a NullReferenceException is thrown.
- The enumerator is advanced to the next element by evaluating the method invocation e.MoveNext().

If the value returned by e.MoveNext() is true, the following steps are performed:

- The current enumerator value is obtained by evaluating the property access e.Current, and the value is converted to the type of the iteration variable by an explicit conversion. The resulting value is stored in the iteration variable such that it can be accessed in the embedded statement.
- Control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), another foreach iteration is performed, starting with the step above that advances the enumerator.

If the value returned by e.MoveNext() is false, control is transferred to the end point of the foreach statement.

Within the embedded statement of a foreach statement, a break statement may be used to transfer control to the end point of the foreach statement (thus ending iteration of the embedded statement), and a continue statement may be used to transfer control to the end point of the embedded statement (thus executing another iteration of the foreach statement).

The embedded statement of a foreach statement is reachable if the foreach statement is reachable. Likewise, the end point of a foreach statement is reachable if the foreach statement is reachable.



Task Analyze the difference with suitable example between do...while and while statements.

Self Assessment

Fill in the blanks:

10. The while statement has two parts - looping and body.
11. Within the embedded statement of a do statement, a statement may be used to transfer control to the end point of the do statement.

12. The for statement is reachable and a for-condition is present and does not have the value false.

Notes

3.5 Jump Statements

Jump statements unconditionally transfer control to another location. The location to which a jump statement transfers control is called the target of the jump statement. There are five jump statements provided by C#.

break-statement

continue-statement

goto-statement

When a jump statement occurs within a block, and when the target of the jump statement is outside that block, the jump statement is said to exit the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening try statements. In the absence of such try statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening try statements, execution is more complex. If the jump statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement.



Note When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement.

This process is repeated until the finally blocks of all intervening try statements have been executed.

Consider the following code.

```
static void FunctionOne()
{
    while (true)
    {
        try
        {
            try
            {
                Console.WriteLine("Before break");
                break;
            }
            finally
            {
                Console.WriteLine("Innermost finally block");
            }
        }
    }
}
```

Notes

```
    }  
    finally  
    {  
        Console.WriteLine("Outermost finally block");  
    }  
}  
  
Console.WriteLine("After break");  
}
```

Here, the finally blocks associated with two try statements are executed before control is transferred to the target of the jump statement.

The break Statement

The break statement exits the nearest enclosing switch, while, do, for, or foreach statement. Its syntax consists of nothing but the word break.

```
break;
```

The target of a break statement is the end point of the nearest enclosing switch, while, do, for, or foreach statement. If a break statement is not enclosed by a switch, while, do, for, or foreach statement, a compile-time error occurs.

When multiple switch, while, do, for, or foreach statements are nested within each other, a break statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement must be used.

A break statement cannot exit a finally block. When a break statement occurs within a finally block, the target of the break statement must be within the same finally block, or otherwise a compile-time error occurs.

A break statement is executed as follows:

- If the break statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the break statement.

Because a break statement unconditionally transfers control elsewhere, the end point of a break statement is never reachable.



Task When multiple switch, while, do, for, or foreach statements are nested within each other, a break statement applies only to the innermost statement. Analyze

The continue Statement

The continue statement starts a new iteration of the nearest enclosing while, do, for, or foreach statement. The syntax consists of nothing but the word continue.

```
continue;
```

The target of a continue statement is the end point of the embedded statement of the nearest enclosing while, do, for, or foreach statement. If a continue statement is not enclosed by a while, do, for, or foreach statement, a compile-time error occurs.

When multiple while, do, for, or foreach statements are nested within each other, a continue statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement must be used.

A continue statement cannot exit a finally block. When a continue statement occurs within a finally block, the target of the continue statement must be within the same finally block, or otherwise a compile-time error occurs.

A continue statement is executed as follows:

- If the continue statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the continue statement.
- Because a continue statement unconditionally transfers control elsewhere, the end point of a continue statement is never reachable.

The goto Statement

The goto statement is another branching statement that transfers control of execution to a specified labeled statement. The general syntax of - goto - statement is given below.

```
goto identifier;
Or
goto case constant-expression;
Or
goto default;
```

The target of a goto identifier statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a goto statement to transfer control out of a nested scope, but not into a nested scope. Consider the following code.

```
class TestClass
{
    static void Main(string[] args)
    {
        int i = 0;
        while (true)
        {
            Console.WriteLine(i++);
            if (i == 10)
                goto done;
        }
    }
}
```

Notes

```

    }
    done:
        Console.WriteLine("Done");
    }
}

```

Here, a goto statement is used to transfer control out of a nested scope. The target of a goto case statement is the statement list of the switch section in the nearest enclosing switch statement that contains a case label with the given constant value. If the goto case statement is not enclosed by a switch statement, if the constant-expression is not implicitly convertible to the governing type of the nearest enclosing switch statement, or if the nearest enclosing switch statement does not contain a case label with the given constant value, a compile-time error occurs.

The target of a goto default statement is the statement list of the switch section in the nearest enclosing switches statement that contains a default label. If the goto default statement is not enclosed by a switch statement, or if the nearest enclosing switch statement does not contain a default label, a compile-time error occurs.



Did u know? **Why goto statement cannot exit a finally block?**

A goto statement cannot exit a finally block. When a goto statement occurs within a finally block, the target of the goto statement must be within the same finally block, or otherwise a compile-time error occurs.

A goto statement is executed as follows:

- If the goto statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the goto statement.

Because a goto statement unconditionally transfers control elsewhere, the end point of a goto statement is never reachable.

3.6 The Checked and Unchecked Statements

The checked and unchecked statements are used to control the overflow checking context for integral-type arithmetic operations and conversions. The syntax is given below.

```

checked    block
unchecked  block

```

The checked statement causes all expressions in the block to be evaluated in a checked context, and the unchecked statement causes all expressions in the block to be evaluated in an unchecked context.

The checked and unchecked statements are precisely equivalent to the checked and unchecked operators, except that they operate on blocks instead of expressions.

The lock Statement

The lock statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock. The syntax is given below.

```
lock(expression)
statement;
```

The expression of a lock statement must denote a value of a reference-type. An implicit boxing conversion is never performed for the expression of a lock statement, and thus it is an error for the expression to denote a value of a value-type.

A lock statement of the form

```
lock (x) ...
```

where *x* is an expression of a reference-type, is precisely equivalent to

```
System.Threading.Monitor.Enter(x);
try
{
    ...
}
finally
{
    System.Threading.Monitor.Exit(x);
}
```

except that *x* is only evaluated once.



Note The exact behavior of the `Enter` and `Exit` methods of the `System.Threading.Monitor` class is implementation-defined.

The `System.Type` object of a class can conveniently be used as the mutual-exclusion lock for static methods of the class (see the following code).

```
class CacheClass
{
    public static void Add(object x)
    {
        lock (typeof(CacheClass))
        {
            ...
        }
    }
    public static void Remove(object x)
    {
        lock (typeof(CacheClass))
        {
            ...
        }
    }
}
```

Notes

```
}
}
```

The using Statement

The using statement obtains one or more resources, executes a statement, and then disposes of the resource. The syntax is given below.

```
using (resource-acquisition) statement;
```

A resource is a class or struct that implements System.IDisposable, which includes a single parameter-less method named Dispose. Code that is using a resource can call Dispose to indicate that the resource is no longer needed. If Dispose is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of resource-acquisition is local-variable-declaration then the type of the local-variable-declaration must be System.IDisposable or a type that can be implicitly converted to System.IDisposable. If the form of resource-acquisition is expression then this expression must be System.IDisposable or a type that can be implicitly converted to System.IDisposable.

Local variables declared in a resource-acquisition are read-only, and must include an initializer.

A using statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is enclosed in a try statement that includes a finally clause. This finally clause disposes of the resource. If a null resource is acquired, then no call to Dispose is made, and no exception is thrown.



Example: Using statement of the form

```
using (R r1 = new R ())
{
    r1.FunctionOne ();
}
```

is precisely equivalent to

```
R r1 = new R ();
try
{
    r1.FunctionOne ();
}
finally
{
    if (r1 != null) ((IDisposable) r1).Dispose ();
}
```

Self Assessment

Fill in the blanks:

- When a jump statement occurs within a block, and when the target of the jump statement is outside that block, the jump statement is said to the block.

14. The expression of a lock statement must denote a value of a
15. If the nearest enclosing switch statement does not contain a label, a compile-time error occurs.

Notes



Caselet

Almost in the Maker's Shoes

ART is lies that tell the truth, said Picasso. Art is art; everything else is everything else, said Ad Reinhardt. Art completes what nature cannot bring to finish, is an Aristotle quote.

If art imitates life, one form of such exercise is animation. Peter Ratner's "3-D Human Modeling and Animation" has all the tools and know-how "to create digital characters that can move, express emotions and talk". The book demonstrates how you can use your artistic skills in figure drawing, painting, and sculpture to create animated human figures using the latest computer technology. "No one has been able to make computer graphics humans that have been mistaken for real ones in movies and photos when viewed at close range," states the preface. "Until technology evolves to the point that this becomes possible, creating an artistic representation of a human is still a worthwhile goal." A few frames from the book:

- The closer a character becomes to an everyday human, the more ordinary it will appear. Synthetic humans most often lack personality. Computer characters that try to mimic human movement through unedited motion-capture techniques generally look like puppets or store mannequins that have come to life. As contradictory as it sounds, when animators exaggerate the movements and expressions of their characters, they appear more lifelike and realistic.
- Generally, the average height of a man or a woman can be measured as seven heads tall. If your goal is to create and animate the ideal male and female, then consider modelling them eight heads tall. Stretch their proportions first and then use them as your guides. Superheroes are often portrayed as very tall with tiny heads.
- Some artists prefer to start with the smallest unit possible: the point, or vortex. After placing a series of these vertices, one can connect them as a spline or create polygons from them. Splines are flexible line segments defined by edit points or vertices. Sometimes splines are referred to as curves. A series of connected splines make a wire mesh. Adjoining wire meshes are patches. Thus spline modelling lends itself to the patch modelling method.
- Preparing the human model for facial expressions and dialogue is an essential part of the 3-D animation process. A base model with a neutral expression is the starting point. The base model should have at least three sets of parallel lines for each wrinkle. Approximately 56 shapes or morphs, which include the mouth shapes for dialogue, should be enough for the majority of facial expressions.
- Conventional art materials such as charcoal, paint, clay, fibre, and so on, are tactile. The artist who works with these has an emotional link that is often lacking when compared to the one who relies on hardware and software. The computer artist is forced to instill the quality of emotion into a medium that is for the most part cerebral. This is one of the greatest challenges. Without emotional content, the work will appear cold and removed from the human experience. Unlike other artists, computer animators work mostly in the mental realm to put feeling into their work.

Contd...

Notes

If only animators could breathe life into their creations, would life not be one big comics book to sit back and read?

Book courtesy: Wiley Dreamtech India P Ltd.

Onto the Next Generation of Web-based Technology

WEB services is an umbrella term. It describes a collection of industry-standard protocols and services used to facilitate a 'base-line level of interoperability' between applications. Which, in other words, means all different systems can get talking over the cyberspace. "Building XML Web Services for the Microsoft .NET Platform" by Scott Short deals with the basic building blocks of XML Web services, viz. Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), Universal Description, Discovery, and Integration (UDDI) and so on. There's more:

- WSDL documents can be intimidating at first glance. But the syntax of a WSDL document is not nearly as complex as that of an XML Schema document. A WSDL document is composed of a series of associations layered on top of an XML Schema document that describes a Web Service. These associations add to the size and the perceived complexity of a WSDL document. But once you look underneath the covers, WSDL documents are rather straightforward.
- HTTP is by nature a stateless protocol. Even with the introduction of the connection keep-alive protocol in HTTP 1.1, you cannot assume that all requests from a given client will be sent over a single connection. ASP.NET provides a state management service that can be leveraged by Web Forms and Web services.
- A well-known object accepts method requests without requiring the client to first formally instantiate the object. From the client's perspective, the object already exists and calls can be made to it without the need to create or initialise the object. This is the default behaviour of SOAP-based Web services.
- Digest authentication does not transfer the user's password in the clear; instead a hash, or digest, of the password and data provided by the server is used to authenticate the user.
- From a developer's perspective, .NET My Services eliminates many of the problems of securing data, providing encrypted transport channels, and reconciling disparate data sources. And all of this is achievable using XML Web services, so businesses are spared the drastic learning curve associated with new technologies.

Remember, they say Web services are the next big thing.

From OOP to more

LET us say you already understand OOP (object-oriented programming) concepts such as data abstraction, inheritance and polymorphism. And that you want to 'leverage the power of .NET Framework to build, package and deploy any kind of application'. Jeffrey Richter's "Applied Microsoft .NET Framework Programming" has the answers. Read on:

- As an application runs, the Common Language Runtime (CLR) maintains a 'snapshot' of the set of assemblies loaded by the application. When the application terminates, this information is compared with the information in the application's corresponding .ini file. If the application loaded the same set of assemblies that it loaded previously, the information in the .ini file matches the information in memory and the in memory information is discarded.
- Sometimes the add and remove methods the compiler generates are not ideal. For example, if you're adding and removing delegates frequently and you know that

Contd...

your application is single-threaded, the overhead of synchronising access to the object that owns the delegate can really hurt your application's performance.

- Compilers convert code that references an enumerated type's symbol to a numeric value at compile time. Once this occurs, no reference to the enumerated type exists in metadata and the assembly that defines the enumerated type doesn't have to be available at run time. If you have code that references the enumerated type – rather than just having references to symbols defined by the type – the assembly that defines the enumerated type will be required at run time.
- A try block doesn't have to have a finally block associated with it at all; sometimes the code in a try block just doesn't require any cleanup code. However, if you do have a finally block, it must appear after any and all catch blocks, and a try block can have no more than one finally block associated with it.
- If the CLR suspends a thread and detects that the thread is executing unmanaged code, the thread's return address is hijacked and the thread is allowed to resume execution. A pinned object is one that the garbage collector isn't allowed to move in memory.

3.7 Summary

- C# provides a rich variety of statements that include all the familiar statements of language C and C++.
- The switch statement is another convenient tool provided by C# to handle the situations in which multiple decisions to be made on an expression that can have multiple values.
- Conditional operator could be used as an alternate to the if else statement as both of them provides a control flow capacity.
- The For loop in C# is the simplest, fixed and entry controlled loop.
- The second type of loop, the while loop is an entry controlled loop as it tests the conditions first and if the condition is true, then only the control will enter into the loop body.
- An empty loop can also be configured using while statement and could be used as a time delay loop.
- Unlike the For and while loops, a do while loop always executes at least once.
- The Foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.
- Jump statement unconditionally transfer control without checking any condition.

3.8 Keywords

Blocks: Multiple statements blocked to make them behave as a single statement. A block may contain declaration statements.

Control Statement: The statement by which we can control the flow of the program execution is called as control statement.

Jump Statement: These statements unconditionally transfer control to another location referred to as the target of the jump statement. A jump statement may transfer control out of a block but it can never transfer control into a block.

Notes

Loop Body: The loop body consists of statement(s) that is supposed to be executed again and again as long as the condition expression evaluates to true.

Switch Statement: It is a multiple branch statement that successively tests the value of an expression against a list of case values and when a match is found, the statement associated with the particular case is executed.

3.9 Review Questions

1. Compare if-else statement with the switch statement. Explain how the switch statement provides a more elegant way to handle multi-way branching.
2. Translate a For loop into a while loop and vice-versa.
3. Explain with an example how a goto statement is used to transfer control out of a nested scope.
4. "Labeled statements are permitted blocks, but are not permitted as embedded statements". Explain.
5. "A compile-time error occurs if two or more case labels in the same switch statement specify the same constant value". Justify.
6. "The switch statement provides a more elegant way to handle multi-way branching". Explain.
7. "The embedded statement of a do statement is reachable if the do statement is reachable". Explain.
8. What are the various loops available in C#? Explain with the help of suitable codes.
9. "To transfer control across multiple nesting levels, a goto statement must be used." Is it true? Explain.
10. "A break statement cannot exit a finally block". Explain with an example.

Answer: Self Assessment

- | | |
|----------------------|--------------------|
| 1. prefixed | 2. goto |
| 3. Console.WriteLine | 4. local variable |
| 5. reachable | 6. two-way |
| 7. constant | 8. reachable |
| 9. switch-block | 10. condition |
| 11. break | 12. constant |
| 13. exit | 14. reference-type |
| 15. default | |

3.10 Further Readings

Notes



Books

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

<http://www.csharp-station.com/Tutorials/Lesson04.aspx>

<http://www.dotnetperls.com/foreach>

Unit 4: Handling Arrays

CONTENTS

Objectives

Introduction

4.1 Arrays

4.2 Declaring Array

4.3 The System.Array Class

4.3.1 ArrayList Class

4.3.2 The Array Class

4.4 Structures

4.4.1 Declaring Structures

4.4.2 Class vs Structure

4.5 Summary

4.6 Keywords

4.7 Review Questions

4.8 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the introduction to array
- Describe how to declare array
- Demonstrate the System.Array class

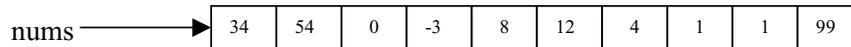
Introduction

This unit will deal with arrays and structures of C# programming. An array is a data structure that contains a number of variables, called the elements of the array, having same name, same type and same size but containing possibly different values. In this unit you will learn about C# arrays, single-dimensional array, multidimensional array, how they are accessed through indexes and how to go about creating arrays. In C#, array types are reference types derived from the abstract base type System.Array. The latter half of the unit deals with structures, which are a group of basic data type variables, which behaves as a unit. Further the structure will be contrasted with classes which structures resembles.

4.1 Arrays

If you have some experience in computer programming, then irrespective of the language(s) you used you would be well aware of arrays. An array is a data structure that contains a number of identical variables (variables having same name, same type and same size) accessible by a number called index. The number of elements an array has is called its rank.

Following is an array of 10 integer variables or simply put, an array of 10 integers.



Lets us understand different parts of this array.

1. Name of the array: nums. This is the name of the array. It could be any valid identifier allowed by a particular language.
2. The rank of the array: 10, i.e., it has 10 elements.
3. Type of the array: integer, i.e., its elements can store only one integer value each.
4. Indices of the array: 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10. Thus the 10 element of the array are:

nums[1] nums[2] nums[3] nums[4] nums[5]
 nums[6] nums[7] nums[8] nums[9] nums[10]

However, in some languages like the language C, the indexing begins at 0 instead of 1. In that case the elements will be referred to as:

nums[0] nums[1] nums[2] nums[3] nums[4]
 nums[5] nums[6] nums[7] nums[8] nums[9]

Still in some cases both the options are available as in Visual Basic wherein it could begin with either 0 or 1. The programmer has to specify the option in the beginning of the program. Even square brackets around the indices are language specific. In some languages (such as Visual Basic) parentheses are used in place of square brackets, as shown below.

nums(0) nums(1) nums(2) nums(3) nums(4)
 nums(5) nums(6) nums(7) nums(8) nums(9)

Also note that it is not necessary that the indices are always integers. They could be any other variable or expression that evaluate to a valid index value.



Did u know? **What is valid indexed value?**

An index value is valid for a particular array if its value lies between the lower and upper index values exclusive.

Thus, if ind is an integer variable whose current value is 3 then all the following are valid index value for the array given above provided the indices fall between the upper and lower index values inclusively.

nums[ind]
 nums[ind + 4]
 nums[ind - 2]
 nums[2 * ind]
 nums[ind++]

5. Values stored in the array in the example are: 34, 54, 0, -3, 8, 12, 4, 1, 1, 99. Assuming that the first index is 0, nums[0] stores 34, nums[5] stores 12 while nums[9] stores 99.
6. These elements behave just as any other integer type variable does. Thus, all the following expressions are valid.

nums[0] + 3 ;which evaluates to 37

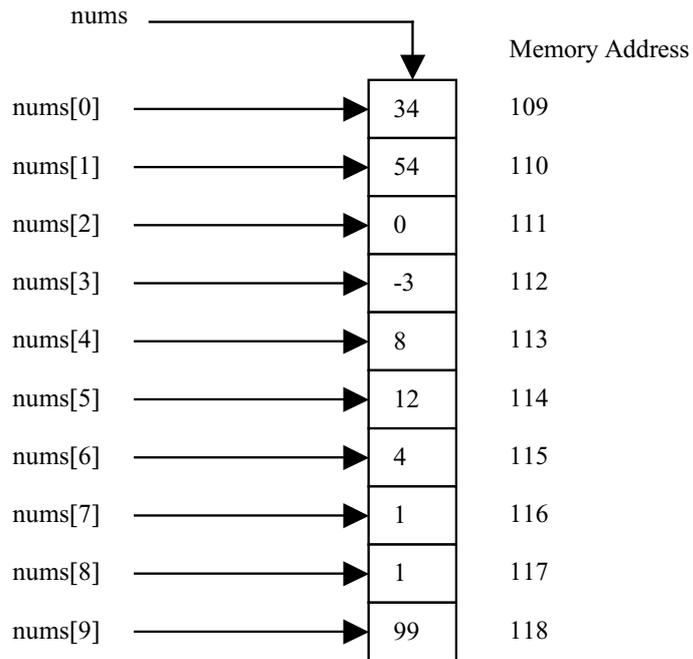
Notes

`nums[4] - nums[5]` ;which evaluates to -4

In fact, even the following expression is valid!

`nums[nums[2]] + 4` ;which evaluates to 38

- The array elements are created at compile time implying that you cannot change the number of elements in the array dynamically at runtime. This is referred to as static allocation of memory. Moreover the elements are allocated contiguous memory locations when created. Thus, assuming that the first element of the above array is allocated memory location - (say) 109, then the array will have the following memory map.



One of the important implications of contiguous memory allocation of the array elements is that the array elements can be manipulated with the help of pointers.

Pointer is a variable (or constant) that is capable of storing memory address of another variable. The name of the array itself is a pointer, a constant pointer to be exact, which stores the address of the first element. In the foregoing example, `nums` is a pointer type constant that stores 109, the address of the first element of the array. Thus, all the following references are valid.



Note A pointer is nothing but a variable that holds the memory address of another type. But in C# pointer can only be declared to hold the memory address of value types and arrays.

`nums` ;points to the first array element

`nums+1` ;points to the second array element

It is interesting to note that in most of the languages strings of characters are stored in this manner.

8. The number of integers used in the index to access an element of the array is called its dimension. Since in our case (the array `nums`) only one integer appears into the index, it is one-dimensional array. Arrays may be two-dimensional, three-dimensional or n-dimensional in which case it is called multi-dimensional array.

Self Assessment

Fill in the blanks:

1. An array is a data structure that contains a number of identical variables accessible by a number called
2. Even square brackets around the indices are specific.
3. The array elements are created at compile time implying that you cannot change the number of elements in the array dynamically at
4. Pointer is a variable (or constant) that is capable of storing memory address of another
5. The number of integers used in the index to access an element of the array is called its

4.2 Declaring Array

In C#, an array index starts at zero. That means the first item of an array starts at the 0th position. The position of the last item on an array will total number of items - 1. So if an array has 10 items, the last 10th item is at 9th position. In C#, arrays can be declared as fixed length or dynamic.

A fixed length array can store a predefined number of items.

A dynamic array does not have a predefined size. The size of a dynamic array increases as you add new items to the array. You can declare an array of fixed length or dynamic.



Caution You can even change a dynamic array to static after it is defined.

Let's take a look at simple declarations of arrays in C#. The following code snippet defines the simplest dynamic array of integer types that does not have a fixed size.

```
int[] intArray;
```

As you can see from the above code snippet, the declaration of an array starts with a type of array followed by a square bracket (`[]`) and name of the array.

The following code snippet declares an array that can store 5 items only starting from index 0 to 4.

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

The following code snippet declares an array that can store 100 items starting from index 0 to 99.

```
int[] intArray;
intArray = new int[100];
```

Array instances are created using array-initializer. While creating an array instance the rank and length of each dimension are specified explicitly which cannot be changed as long as the array exists. Elements of arrays created thus are always initialized to their default value.

Notes

Various valid forms of array creation syntax are given below.



Example: Create a one-dimensional array named - intArr - of 5 integers each element containing default values.

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



Example: Create a one-dimensional array named - names - of 15 strings each element containing default values.

```
string[ ] names = new string[15];
```

The array-initializer in this case is - int[] and string[]. The reserved word - new - creates an array each and assigns the references to intArr and names respectively. Here, in this example, the data type (int), number of elements (5) and dimension (1) are all specified along with the name of the array.

The elements of this array may be initialized with desired integers now explicitly as shown below.

```
intArr[0] = 4 intArr[1] = 6 intArr[2] = 1 intArr[3] = 9 intArr[4] = 3
```

However, one can initialize the array with given values even at the time of creation in the declaration itself. See the following example.



Example: Create a one-dimensional array named - aaa - of integers containing 5 given

Integers - 4, 6, 1, 9, 3.

```
int[ ] aaa = new int[ ] { 4, 6, 1, 9, 3 };
```

Or simply,

```
int[ ] aaa = { 4, 6, 1, 9, 3 };
```

When an array is created with values assigned to its elements, the number of elements need not be explicitly specified as is shown in the example given above. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of elements in the array-initializer determines the length of the array instance being created. In the example given above the length is 5 (indices from 0 to 4).

Creating a multi-dimensional array is similar to creating one-dimensional array. The array initializer specifies as many levels of nesting as there are dimensions in the array. The number of elements in each of the dimensions may be explicitly stated as shown in the example below.



Example: Create a two-dimensional array (say, 4 rows and 3 columns) named intArr of integers with default values.

```
int[ , ] intArr = new int[ 4, 3 ];
```

The indices of this array vary from 0 to 3 for rows and from 0 to 2 for columns. Different values can now be stored in the array by assigning the given values to the individual elements, as shown below.

```
intArr[ 0, 0 ] = 11    intArr[ 0, 1 ] = 3
intArr[ 1, 0 ] = 5    intArr[ 1, 1 ] = 2
intArr[ 2, 0 ] = 4    intArr[ 2, 1 ] = 9
intArr[ 3, 0 ] = 14   intArr[ 3, 1 ] = 7
```

In multi-dimensional arrays the outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The number of elements in each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. However, for each nested array initializer, the number of elements must be the same as the other array initializers at the same level.



Example: Create a two-dimensional array named - bbb - having the following integers arranged in rows and column shown below.

```

11    3
5     2
4     9
14    7

int[,] bbb = {{11, 3}, {5, 2}, {4, 9}, {14, 7}};

```

Note that in this example the number of elements in each dimension is not specified explicitly. The compiler computes it at compile time.

The number of elements specified in the array-initializer must always be a constant positive integer. It cannot be a variable. Moreover, the number of elements at each nesting level must match the corresponding dimension length. Thus, following array creation declarations are incorrect.

```

int i = 3;
int[] aaa = new int[i];

```

This declaration is incorrect because the number of elements specified (i) is not a constant. Note that the compiler must know the exact number of elements to be created at compile time. Since a variable (such as I) can take any value at run time, a variable cannot be employed as the specification of number of elements to create an array.

```

int[] ccc = new int[4] {10, 21, 2, 56, 77};

```

This declaration is incorrect because the number of elements specified (4) and the number of initial values in the list (5) do not match.

Self Assessment

Fill in the blanks:

6. A dynamic array does not have a size.
7. Array instances are created using
8. The number of elements in each dimension of the array is determined by the number of at the corresponding nesting level in the array initializer.
9. The compiler must know the exact number of elements to be created at time.
10. The array initializer specifies as many levels of nesting as there are in the array.

4.3 The System.Array Class

C# offers an in-built class - System.Array- that serves as the base class for all types of arrays. It provides methods for creating, copying, manipulating, searching, and sorting arrays. Here is the class prototype of System.Array class.

Notes

```
public abstract class Array : ICloneable, ICollection, IEnumerable, IList
```

This class is intended to be used as a base class by language implementations that support arrays. Only the system can derive from this type: derived classes of Array are not to be created by the developer.

Remember that an array is a collection of identically typed data elements that are accessed and referenced by sets of integral indices. The rank of an array is the number of dimensions in the array. Each dimension has its own set of indices. An array with a rank greater than one can have a different lower bound and a different number of elements for each dimension. Multidimensional arrays (i.e. arrays with a rank greater than one) are processed in row-major order.

The lower bound of a dimension is the starting index of that dimension. The length of an array is the total number of elements contained in all of its dimensions.

The type System.Array is the abstract base type of all array types. An implicit reference conversion exists from any array type to System.Array, and an explicit reference conversion exists from System.Array to any array type.

At run-time, a value of type System.Array can be null or a reference to an instance of any array type.



Note System.Array is not itself an array-type rather, it is a class-type from which all array-types are derived.

4.3.1 ArrayList Class

To support arrays of any kind, the Microsoft .NET Framework provides the ArrayList class. This class can be used to add, locate, or remove an item from a list. The class provides many other valuable operations routinely done on a list. It allows you to do the same functionality as dynamic memory allocation but is much simpler to implement. With ArrayList, you no longer need to worry about freeing up memory after allocation of memory and array bound overflow.

The capacity of an ArrayList is the number of elements the list can hold. As elements are added to an ArrayList, the capacity is automatically increased as required through reallocation. Indexes in this collection are zero-based.

ArrayList accepts a null reference (Nothing in Visual Basic) as a valid value and allows duplicate elements.

The primary operation performed on a list is to create one. After declaring a variable of type ArrayList, you can call the Add() method every time you need to add a new item to the list.

Let us see one example.

```
using System;

{
    class Sales
    {
        static void Main()
        {
            ArrayList lstItems = new ArrayList();
            StoreItem anItem;
            anItem = new StoreItem();
        }
    }
}
```

Notes

```
anItem.ItemNumber = 31882;
anItem.ItemName   = "Zembla Sling Shoe - Women";
anItem.UnitPrice  = 255.75;
lstItems.Add(anItem);
    anItem = new StoreItem();
anItem.ItemNumber = 70517;
anItem.ItemName   = "14-Karat Rain Diamond Earrings";
anItem.UnitPrice  = 1250.55;
lstItems.Add(anItem);
    anItem = new StoreItem();
anItem.ItemNumber = 36308;
anItem.ItemName   = "Men Khaki Pants";
anItem.UnitPrice  = 32.90;
lstItems.Add(anItem);
    anItem = new StoreItem();
anItem.ItemNumber = 11582;
anItem.ItemName   = "Children Summer Hat";
anItem.UnitPrice  = 18.75;
lstItems.Add(anItem);
    anItem = new StoreItem();
anItem.ItemNumber = 44005;
anItem.ItemName   = "Men Black Leather Shoe";
anItem.UnitPrice  = 225.45;
lstItems.Add(anItem);
    Console.WriteLine("Store Inventory");
    for(int i = 0; i < lstItems.Count; i++)
    {
        Console.WriteLine("Index: {0}", i);
        anItem = (StoreItem)lstItems[i];
        Console.WriteLine("Item Stock #: {0}", anItem.ItemNumber);
        Console.WriteLine("Description: {0}", anItem.ItemName);
        Console.WriteLine("Unit Price: {0:C}\n", anItem.UnitPrice);
    }
}
}
```

Notes



Example:

```
using System;
using System.Collections;
public class SamplesArrayList
{
    public static void Main()
    {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add("Hello");
        myAL.Add("World");
        myAL.Add("!");
        // Displays the properties and values of the ArrayList.
        Console.WriteLine("myAL");
        Console.WriteLine("\tCount:    {0}", myAL.Count);
        Console.WriteLine("\tCapacity: {0}", myAL.Capacity);
        Console.Write( "\tValues:" );
        PrintValues( myAL );
    }
    public static void PrintValues( IEnumerable myList ) {
        System.Collections.IEnumerator myEnumerator = myList.GetEnumerator();
        while ( myEnumerator.MoveNext() )
            Console.Write( "\t{0}", myEnumerator.Current );
        Console.WriteLine();
    }
}
/*
```

This code produces the following output.

```
myAL
Count:    3
Capacity: 16
Values:   Hello    World    !
```

4.3.2 The Array Class

An array is implemented as a class in System package of C#. The System.Array type is the base class from which all the array objects are derived. When you create an array using either array-initializer or using new method what is created is a subclass of System.Array base class. This class provides all the necessary functionalities of an array data type. A synopsis of some of the member methods of System.Array class is presented below.



Did u know? Explain the working of array class.

When you create an array, you are in fact declaring a variable of type Array. Based on this, since an array variable is an object of a class type, you can use the characteristics of the Array class to create an array and/or to manipulate the values stored in the variable.

System.Array.Clear()

This is a static method of the class. It clears the specified range of elements of the stored values. Numeric elements are assigned with 0 while reference type elements are assigned null. The syntax of the method is given below.

```
System.Array.Clear(arrayname, lowerindex, higherindex);
```



Example: Clear all the elements having index values between 3 to 9 inclusive of the array names.

```
System.Array.Clear(names, 3, 9);
```

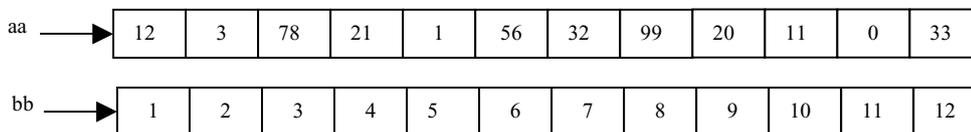
System.Array.Copy()

This method takes two array arguments and copies specified number of elements starting from specified index into another array from specified index position. The syntax is given below.

```
System.Array.Copy(SourceArr, SourceInd, DestArr, DestInd, Len);
```

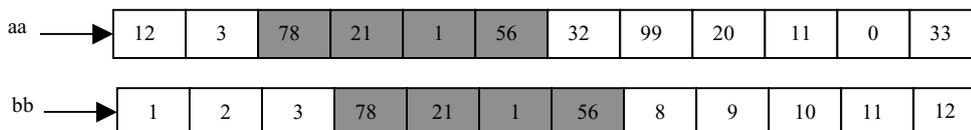


Example: Copy 4 elements starting from third position (index 2) of array aa into array bb starting at fourth position (index 3) as shown below.



```
System.Array.Copy(aa, 2, bb, 3, 4);
```

The result is shown below.



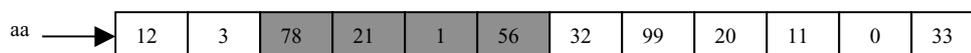
System.Array.Reverse()

This method reverses the order of a given length of the given array. The syntax is given below.

```
System.Array.Reverse(array, index, length);
```



Example: Reverse the order of 4 elements starting from third position (index 2) of the given array aa.



```
System.Array.Reverse(array, index, length);
```

Notes

The result is shown below.




Task Answer the following questions:

1. What do you mean by “Array”? How it can be declared and initialized in a C# program?
2. Explain the different ways to initialize Arrays.
3. Name some of the methods available in System.Array class.
4. What are the differences between a fixed size array and a variable size array?
5. Write a program using a variable size integer array.
6. How can you create a three dimensional array?

Self Assessment

Fill in the blanks:

11. The of an array is the number of dimensions in the array.
12. An array is implemented as a in System package of C#.
13. The capacity of an is the number of elements the list can hold.

4.4 Structures

A structure is a variable data type that contains a group of variables of different types including structure type and functions. In this sense structures are similar to classes. However, there is a significant difference between the two. Structures are value types while classes are reference types. Consequently, structures store data values directly and do not require heap allocation. On the other hand a class type variable contains a reference to the data stored elsewhere (called the object) on the heap memory.

Using structures a programmer can treat a group of related variables as a single unit. Thus, data items like complex numbers, coordinates of points etc. can be conveniently structured as structures. The primitive data types in C# such as int, double, and bool, are actually defined to be structures behind the scene.

4.4.1 Declaring Structures

The declaration of a structure defines its template. It does not create any variable. After one has declared a structure one can create variables of that structure type just as any variable is created. A structure is declared as shown below.

```
struct-declaration:
attributes modifiers struct identifier1 interfaces body; identifier2;
```

Where attributes is an optional set of attributes; modifiers is an optional set of modifiers; struct is mandatory keyword; identifier1 is the name of this structure type; interfaces is an optional list of interfaces that the structure may implement; body is mandatory part where the members of the

structure are specified; and identifier2 is an optional comma separated variable names which if present are created as this structure type.

1. A structure declaration may optionally include a sequence of structure modifiers. The modifiers may be one of the following.
 - (i) new
 - (ii) public
 - (iii) protected
 - (iv) internal
 - (v) private

These modifiers are common between classes and structures. Classes have a few more modifiers than listed above. The abstract and sealed modifiers are not applicable only in classes and not in structures.
2. A structure declaration may implement one or more interfaces similar to classes.
3. The body of a structure is enclosed within curly braces - { }. It is here that the members of the structures are specified.

The members of a structure can a group of the following types.

1. constant
2. field
3. method
4. property
5. event
6. indexer
7. operator
8. constructor
9. static-constructor
10. type



Task Answers the following questions:

1. What is a structure? How is it different from classes?
2. What are the different structure modifiers?

4.4.2 Class vs Structure

Classes and structures appear to be identical. However, it is not quite so. They differ more than they are similar. Some of the most important differences are mentioned hereunder.

Type

Structures are value types as against classes, which are reference types. A structure type variable directly contains the data of the structure. The same is not true with classes. A class type variable

Notes

does not have data embedded with its name. It is just a reference (or pointer) that is stored at a location. The data associated with the class (object) is stored on the heap memory to which the name of the class points.

One of the consequences of this difference is that two variables of class type may reference the same object, and therefore it is possible for operations on one variable to affect the object referenced by the other variable. Structure variables do not reference each other. They each have their own copy of the data hence it is not possible for operations on one to affect the other.

It is because of this difference that structure type variables cannot be assigned null values while a class type variable may be assigned a null value.



Example:

```
using System;
class One
{
    struct Point
    {
        public int x, y, z;
        public Point(int x, int y, int z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }
    }
    static void Main(string[] args)
    {
        Point a = new Point(100, 200, 300);
        Point b = a;
        Console.WriteLine("b.x before assignment " + b.x.ToString());
        a.x=400;
        Console.WriteLine("b.x after assignment " + b.x.ToString());
    }
}
```

The output of the above program is shown below.

```
b.x before assignment 100
b.x before assignment 100
```



Notes The value stored in b.x remains unaffected even though the value of a.x is changed. This is because the assignment b = a creates a structure type (the type of a) variable b which has its own copy of x, y and z.

Inheritance

Notes

A class may be sub-classed to form a new class. The new class inherits properties and methods from the base-class. Even structure type variables implicitly inherit from class object. Unlike a class a structure type cannot specify a base class though it may implement interfaces. Structure types cannot be abstract and are always implicitly sealed. Therefore, abstract and sealed modifiers are not permitted in a structure declaration. Inheritance is not applicable to structures. Therefore, the members of a structure cannot have protected or protected internal accessibility. For the same reason the function members of a structure cannot be abstract or virtual.

Assignment

Assignment to a variable of a structure type creates a copy of the value being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the object pointed by the reference.

Similar to assignment, a copy of the structure passed to or returned from a member function when the structure is passed as a value parameter. However, in order to pass a structure to a member function by reference a ref or out parameter must be used.

Default Values

In C# the variables are initialized with default values at the time of their creation unlike C wherein the initial value could be garbage. The reference type variables, including a class variable, are initialized with null. Structures being value type are initialized differently. For a structure the default initialization takes place member-wise, i.e., each field is initialized with default values. For instance, numeric fields are initialized with 0.

Consider the Point structure once again.

```
struct Point
{
    public int x, y, z;
    public Point(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

If a new Point structure variable is created using new key word, the individual fields (i.e., x, y and z) are initialized with 0.

```
Point[] point1 = new Point[];
```

The value of each of the variables point1.x, point1.y and point1.z shall be set to 0.

Boxing and Unboxing

While boxing and unboxing conversion of a class type variable the object remains the same only its reference changes to appropriate type. The same is not true with structures. Boxing and unboxing in this manner is not applicable to structures simply because structures are value

Notes

types, not reference type. Therefore, in both the boxing and unboxing operations a copy of the structures being boxed or unboxed is created which does not reflect the changes done on the original structure variable.

This

The reserved word - `this` - is treated as a value when used within a constructor or member function of a class. Therefore, `this` can be used to refer to the instance for which the function member was invoked. However, it is not possible to assign a value to `this` in a member function of a class.

On the contrary, in case of a structure, `this` corresponds to an out parameter and within and member function of a structure, `this` corresponds to a ref parameter. In either case, `this` is treated as a variable. Therefore, it is possible to assign values to it and modify the entire structure for which the member function was invoked. It can also be passed as a ref or out parameter to a member function.

Field Initializers

The fields of a structure get initialized to default values at the time of creation. Therefore, it is an error to initialize the fields with constants in the declaration. Consider the following code-snippet.

```
struct Point
{
    public int x = 100;    // Error, initializer not permitted
    public int y = 200;    // Error, initializer not permitted
    public int z = 300;    // Error, initializer not permitted
}
```

The same can be achieved by introducing a constructor function to the structure and passing to it the initial values at the time of creation of the structure. See the following code fragment.

```
struct Point
{
    public int x, y, z;
    public Point(int x, int y, int z);
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}

Point point1 = new Point(100, 200, 300);
```

Notes



Notes Field initialization is permitted for class variables (i.e., static variables). Thus the following code is correct.

```
struct Point
{
    public int x, y, z;
    static public NumberOfpoints = 0;    //no error!
    public Point(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

The Point class has been added with a static member - NumberOfPoints. This is a class variables and not instance variable, which is indicated by it being static. Therefore, it is no error to initialize with value.

Constructors

You have seen that a structure can have constructors and/or destructor just as a class has. However, unlike a class, a structure cannot declare a constructor having no parameters. In fact, every structure has a parameterless constructor implicitly provided which initializes the fields of the structure when created without calling its constructor. Besides, in case of a nested structure a constructor of the form base(...) is not allowed for the purpose of initialization as it happens in case of a class. A structure can also have static constructors which are similar to those of classes.

Destructors

While a class can have a explicit destructor member function a structure cannot have one.



Task Answers the following questions:

1. What are the main differences between a classes and structures?
2. Explain boxing and unboxing conversions. How do they differ in classes and structures?
3. What is the difference between default and parameterized constructor?

Notes

Self Assessment

Fill in the blanks:

- 14. While boxing and unboxing conversion of a class type variable the object remains the same only its changes to appropriate type.
- 15. Even structure type variables inherit from class object.



Caselet **Rational unveils new developer**

RATIONAL Software India has announced the launch of its latest developer offering – the Rational XDE Professional at Microsoft’s largest conference and exhibitions in the world – Tech Ed 2002 in Mumbai.

Rational XDE Professional, which supports Microsoft’s Visual Basic .NET and ASP.NET, has further widened Rational’s unique extended development environment by allowing users to work in a single environment, enabling them to build a rich, highly scalable business application in a more simplified and timely manner.

“Rational’s new support for Microsoft Visual Basic .NET and ASP.NET continues our strategy of expanding Rational’s unique ‘Extended Development Experience,’ which we introduced just six months ago with support for Visual C# .NET,” said Mr Eric Schurr, Chief Marketing Officer, Rational Software.

“The new support for these key Microsoft .NET technologies and the advanced features in Rational XDE Professional liberate developers from unproductive activities that stifle creativity, enabling them to focus on creating great code,” he said in a press release.

4.5 Summary

- An Array is a group of memory locations related by the fact that they all have the same name and same data type.
- An Array including more than one dimension is called a multidimensional array.
- In memory two-dimensional arrays are represent in two ways, as Row major form, Column major form.
- The System.Array class serves as the base class for all types of arrays. The System.Array is not itself an array type rather; it is a class type from which all array types are derived.
- A component of an array is referred to by its positions in the array whereas each component of a structure has a unique name.
- Structure members are stored in memory in the same order in which they are declared.
- The keywords struct can be used to declare a structure.
- In C# structure member can be accessed in the same way as class member access.
- A C# struct can also contain methods. The methods can be either static or non-static.
- In C# every value type implicitly has a public parameter less default constructor.
- There is no inheritance for structs as there is for classes. A struct cannot inherit form another struct or class and it cannot be the base class for a class.

4.6 Keywords

Boxing: It is an implicit conversion of a value type to the type object.

Default Constructor: Parameter less constructor is called as default constructor.

Element: Each constituted value of an array is called an element.

Multi-dimensional Array: Arrays having more than two dimensions are called as Multi-dimensional arrays.

Structure: A structure is a collection of different data types in such a way that they can be referenced as a single unit.

Unboxing: It is an explicit conversion from the type object to a value type.

4.7 Review Questions

1. Write a C# program to sort a two-dimensional array in order specified by user.
2. What are the differences between a fixed size array and a variable size array?
3. What will be the output of the following C# code?

```
using System;
public class ArrayTest
{
    public static void Main()
    {
        int d=0;
        int[][] myArray = new int[2][];
        myArray[0] = new int[5] {1,3,5,7,9};
        myArray[1] = new int[4] {2,4,6,8};
        for (int i=0; i < myArray.Length; i++)
            for (int j = 0 ; j < myArray[i].Length ; j++)
                d = d + myArray[i,j];
        Console.WriteLine("{0},d);
    }
}
```

4. Write a program using a variable size integer array.
5. Write a program to accept 12 numbers in two dimensional arrays in row major form.
6. Write a program to demonstrate the use of different method of ArrayList Class.
7. What are the main differences between a class and a structure with methods?
8. What are the advantages of enumeration data type?
9. Explain the usages of boxing and unboxing conversions.

Notes

10. Dry-run the following program to find out its output.

```
using System;

public class En
{
    public enum MON: byte
    {January, February, November, December, March}

    public static void Main()
    {
        Array dayArray = Enum.GetValues(typeof(En.MON));
        foreach (MON m in dayArray)
            Console.WriteLine("Number {1} of En.MON is {0}", m,
m.ToString("d"));
    }
}
```

Answers: Self Assessment

- | | |
|----------------------|----------------|
| 1. index | 2. language |
| 3. runtime | 4. variable |
| 5. dimension | 6. predefined |
| 7. array-initializer | 8. elements |
| 9. compile | 10. dimensions |
| 11. rank | 12. class |
| 13. ArrayList | 14. reference |
| 15. implicitly | |

4.8 Further readings



Books

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

<http://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx>

<http://www.functionx.com/csharp/Lesson24.htm>

Unit 5: Methods

Notes

CONTENTS

Objectives

Introduction

5.1 Declaring Methods

5.2 Invoking Methods

5.3 Nesting of Methods

5.4 Pass by Value and Pass by Reference

5.5 Variable Argument Lists

5.6 Method Overloading

5.7 Method Overriding

5.8 Summary

5.9 Keywords

5.10 Review Questions

5.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the introduction to Methods
- Describe how to invoke methods
- Demonstrate the Pass by Value
- Explain Pass by Reference

Introduction

A method is a member that implements a computation or action that can be performed by an object or class. Methods are the means through which an object's data members may be manipulated. The program logic is implemented by methods of objects in an application.

5.1 Declaring Methods

Methods are declared using method-declarations whose syntax is given below.

```
method-declaration:
```

```
method-header method-body
```

```
method-header:
```

```
attributesopt method-modifiersopt return-type member-name (formal-  
parameter-listopt)
```

Notes

```
method-modifiers:  
method-modifier  
method-modifier:  
new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern  
return-type:  
type  
void  
member-name:  
identifier  
method-body:  
block;
```

A method-declaration may include a set of attributes, a new modifier, an extern modifier, a valid combination of the four access modifiers, and a valid combination of the static, virtual, override, and abstract modifiers. In addition, a method that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract method can override a virtual one.

The return-type of a method declaration specifies the type of the value computed and returned by the method.



Caution The return-type is void if the method does not return a value.

The member-name specifies the name of the method. Unless the method is an explicit interface member implementation, the member-name is simply an identifier. For an explicit interface member implementation, the member-name consists of an interface-type followed by a "." and an identifier. The optional formal-parameter-list specifies the parameters of the method.

The return-type and each of the types referenced in the formal-parameter-list of a method must be at least as accessible as the method itself. For abstract and extern methods, the method-body consists simply of a semicolon. For all other methods, the method-body consists of a block, which specifies the statements to execute when the method is invoked.

The name and the formal parameter list of a method define the signature of the method. Specifically, the signature of a method consists of its name and the number, modifiers, and types of its formal parameters. The return type is not part of a method's signature, nor is the names of the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class.

The Main Method

The method with the name - Main - with upper case M, has a special significance. It is the entry point in any application. As such there must be at least one Main method in one class of the entire application. When an application is executed it is this method that gets the execution control automatically in the beginning.



Notes More than one class may have Main methods in an application. However, the programmer must specify which Main method is the entry point of the application at the compile time failing which the compiler generates error message.

Since Main method should be accessible outside the class that declares it, the method must be declared as public. However, C# allows Main methods to private as well.

Main method takes various forms depending on the purpose. Some of the valid declarations of Main method are listed below.

A Main method not taking any argument and returning no value:

```
public static void Main()
{
    //method body;
}
```

A Main method taking an integer type argument and returning no value:

```
public static void Main(int x)
{
    //method body;
}
```

A Main method not taking any argument and returning an integer type value:

```
public static int Main()
{
    //method body;
    return 0;
}
```

A Main method taking an integer type argument and returning an integer type value:

```
public static int Main(int x)
{
```

Notes

```
//method body;  
return 0;  
}
```

A Main method taking command line arguments and returning no value:

```
public static void Main(string [] args)  
{  
    //method body;  
}
```

A Main method taking command line arguments and returning an integer value:

```
public static int Main(string [] args)  
{  
    //method body;  
}
```

Self Assessment

Fill in the blanks:

1. The static, virtual, override, and modifiers are mutually exclusive except in one case.
2. The optional formal-parameter-list specifies the of the method.
3. Since Main method should be accessible outside the class that declares it, the method must be declared as.....
4. The and each of the types referenced in the formal-parameter-list of a method must be at least as accessible as the method itself.

5.2 Invoking Methods

A method, once declared in a class can be invoked (executed) on an object of that class as shown below.

```
public class abc  
{  
    public:  
        void method1()  
        {  
        }  
}
```

In this code snippet, a public class – abc – has been declared with a public method – method1 that takes no arguments and returns no value. This method can be invoked on an object of abc type as shown below.

```
abc Obj1;  
Obj1.method1();
```

Notes



Notes The method1 is available for public use and therefore can be invoked outside the class definition. Private and protected methods cannot be invoked outside the definition of a class.

Also, in order to invoke the method – method1 – an object of class - abc - must be created because method1 is an instance method not static. A static method may be invoked without having to create an object of that class as shown below.

```
public class abc
{
    public:
        static void method1()
        {
        }
    }
}
abc.method1();
```

5.3 Nesting of Methods

A method is said to be nested, if it invokes other methods from within its definition. Consider the following code snippet for illustration.

```
public class abc
{
    public:
        void method1()
        {
            //some statements;
            method2();
            //some other statements;
        }
}
void method2()
{
    //method2 body;
}
}
```

In the above example, method1 is a nested method. It is also possible that a method invokes itself in its definition. Such a method is known as recursive method.



Task Analyze method declaration, method definition, and method calling. Explain with the help of suitable examples.

Notes

Method Parameters

The parameters of a method, if any, are declared by the method's formal-parameter-list.

```

formal-parameter-list:
fixed-parameters
fixed-parameters , parameter-array
parameter-array
fixed-parameters:
fixed-parameter
fixed-parameters , fixed-parameter
fixed-parameter:
attributesopt parameter-modifieropt type identifier
parameter-modifier:
ref
out
parameter-array:
attributesopt params array-type identifier
    
```

The formal parameter list consists of one or more fixed-parameters optionally followed by a single parameter-array, all separated by commas.

A fixed-parameter consists of an optional set of attributes, an optional ref or out modifier, a type, and an identifier. Each fixed-parameter declares a parameter of the given type with the given name.



Did u know? **What is parameter array?**

A parameter-array consists of an optional set of attributes, a params modifier, an array-type, and an identifier. A parameter array declares a single parameter of the given array type with the given name.

The array-type of a parameter array must be a single-dimensional array type. In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. A method declaration creates a separate declaration space for parameters and local variables. Names are introduced into this declaration space by the formal parameter list of the method and by local variable declarations in the block of the method. All names in the declaration space of a method must be unique. Thus, it is an error for a parameter or local variable to have the same name as another parameter or local variable.

A method invocation creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the block of a method, their identifiers in simple-name expressions can reference formal parameters.

There are four kinds of formal parameters:

1. Value parameters, which are declared without any modifiers.
2. Reference parameters, which are declared with the ref modifier.

3. Output parameters, which are declared with the out modifier.
4. Parameter arrays, which are declared with the params modifier.

The ref and out modifiers are part of a method's signature, but the params modifier is not.

Self Assessment

Fill in the blanks:

5. A method is said to be....., if it invokes other methods from within its definition.
6. The array-type of a parameter array must be a array type.
7. A method declaration creates a separate declaration space for parameters and.....
8. A fixed-parameter consists of an optional set of....., an optional ref or out modifier, a type, and an identifier.

5.4 Pass by Value and Pass by Reference

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter – they have no effect on the actual argument given in the method invocation.

Reference Parameters

A parameter declared with a ref modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword ref followed by a variable-reference of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.



Example:

```
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
x = y;
        y = temp;
    }
}
```

Notes

```
    }
    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

produces the output

i = 2, j = 1

For the invocation of Swap in Main, x represents i and y represents j. Thus, the invocation has the effect of swapping the values of i and j.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location.



Example:

```
class A
{
    string s;
    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }
    void G() {
        F(ref s, ref s);
    }
}
```

The invocation of F in G passes a reference to s for both a and b. Thus, for that invocation, the names s, a, and b all refer to the same storage location, and the three assignments all modify the instance field s.

The Output Parameters

A parameter declared with an out modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword out followed by a variable-reference of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

Output parameters are typically used in methods that produce multiple return values.



Example:

```
class Test
{
    static void SplitPath(string path, out string dir, out string name)
    {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\\' || ch == \'/\'' || ch == \':\')
                break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

The example produces the output:

```
c:\Windows\System\hello.txt
```



Notes The dir and name variables can be unassigned before they are passed to SplitPath, and that they are considered definitely assigned following the call.

Parameter Arrays

A parameter declared with a params modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types string[] and string[][] can be used as the type of a parameter array, but the type string[,] can not. It is not possible to combine the params modifier with the ref and out modifiers.

Notes

A parameter array permits arguments to be specified in one of two ways in a method invocation:

1. The argument given for a parameter array can be a single expression of a type that is implicitly convertible to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
2. Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter of the same type.



Example:

```
class Test
{
    static void F(params int[] args) {
        Console.WriteLine("Array contains {0} elements:", args.Length);
        foreach (int i in args) Console.Write(" {0}", i);
        Console.WriteLine();
    }
    static void Main() {
        int[] a = {1, 2, 3};
        F(a);
        F(10, 20, 30, 40);
        F();
    }
}
```

produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of F simply passes the array a as a value parameter. The second invocation of F automatically creates a four-element int[] with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of F creates a zero-element int[] and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form. The expanded form of a method is available only if

the normal form of the method is not applicable and only if a method with the same signature as the expanded form is not already declared in the same type.



Example:

```
class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }
    static void F() {
        Console.WriteLine("F()");
    }
    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }
    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}
```

produces the output

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expanded form for a single object parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed. Consider the following example for illustration.

Notes



Example:

```
class Test
{
    static void F(params object[] args) {
        foreach (object o in a) {
            Console.WriteLine(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }
    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;

        F(a);

        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

produces the output

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of F, the normal form of F is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type object[]). Thus, overload resolution selects the normal form of F, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of F is not applicable because no implicit conversion exists from the argument type to the parameter type (type object cannot be implicitly converted to type object[]). However, the expanded form of F is applicable, and it is therefore selected by overload resolution. As a result, a one-element object[] is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an object[]).

Self Assessment

Fill in the blanks:

9. A parameter declared with no is a value parameter.
10. A is permitted to assign new values to a value parameter.
11. The argument given for a parameter array can be a single of a type that is implicitly convertible to the parameter array type.

5.5 Variable Argument Lists

Notes

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` can not. It is not possible to combine the `params` modifier with the `ref` and `out` modifiers.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

1. The argument given for a parameter array can be a single expression of a type that is implicitly convertible to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
2. Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter of the same type. Consider the following example for illustration.



Example:

```
class Test
{
    static void F(params int[] args) {
        Console.WriteLine("Array contains {0} elements:", args.Length);
        foreach (int i in args) Console.Write(" {0}", i);
        Console.WriteLine();
    }

    static void Main() {
        int[] a = {1, 2, 3};
        F(a);
        F(10, 20, 30, 40);
        F();
    }
}
```

produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F` automatically creates a four-element `int[]` with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `F` creates a zero-element `int[]`

Notes

and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
F(new int[] {10, 20, 30, 40});  
F(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form. The expanded form of a method is available only if the normal form of the method is not applicable and only if a method with the same signature as the expanded form is not already declared in the same type.



Example:

```
class Test  
{  
    static void F(params object[] a) {  
        Console.WriteLine("F(object[])");  
    }  
    static void F() {  
        Console.WriteLine("F()");  
    }  
    static void F(object a0, object a1) {  
        Console.WriteLine("F(object,object)");  
    }  
    static void Main() {  
        F();  
        F(1);  
        F(1, 2);  
        F(1, 2, 3);  
        F(1, 2, 3, 4);  
    }  
}
```

produces the output

```
F();  
F(object[]);  
F(object,object);  
F(object[]);  
F(object[]);
```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expanded form for a single object parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.



Example:

```
class Test
{
    static void F(params object[] args) {
        foreach (object o in a) {
            Console.WriteLine(o.GetType().FullName);
            Console.WriteLine(" ");
        }
        Console.WriteLine();
    }
    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

produces the output

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of `F`, the normal form of `F` is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type `object[]`). Thus, overload resolution selects the normal form of `F`, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no implicit conversion exists from the argument type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the expanded form of `F` is applicable, and it is therefore selected by overload resolution.



Did u know? **What is the result of the above explained example?**

One-element `object[]` is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an `object[]`).

Notes

Self Assessment

Fill in the blanks:

- 12. A parameter declared with a modifier is a parameter array.
- 13. When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its form.

5.6 Method Overloading

C# allows you to define different versions of a method in class, and the compiler will automatically select the most appropriate one based on the parameters supplied.

Declaring Operator Overloading

While display a integer we can write:

```
int x = 10;  
Console.WriteLine (x);
```

While display a integer we can write:

```
string Message = "Hello";  
Console.WriteLine (Message);
```

How is this possible? what parameter type does Console.WriteLine () take? If it were expecting to take a string, then the first of these two examples would give error, because there is no implicit cast from int to string.

If Console.WriteLine () was expecting to take any numeric data type however, then the second of these examples would give a compilation error there is no implicit cast from string to any numeric type. Yet fact, both of these lines will compile without error and run to give the expected results.

The reason is that there are two different Console.WriteLine() methods - one of them takes an int as parameter, the other one takes a string.



Did u know? **What is the use of Method Overloading in C#?**

A lot of programming languages supports a technique called default/ optional parameters. It allows the programmer to make one or several parameters optional, by giving them a default value. It's especially practical when adding functionality to existing code.

Add Method Overloading

In addition of two or three integers, we add two integer values as a method, add three integer values as a method, as well. i.e:

```
public class AddingNumbers  
{  
    public int Add(int a, int b)  
    {  
        return a+b;  
    }  
}
```

```

    }
    public int Add(int a, int b, int c)
    {
        return a+b+c;
    }
}

```

Calling Overloaded Methods

```

public int add(int x, int y)
public int add(int x, int y, int z)

```

We've also assumed that two overloaded methods.

Overload 1 requires two integers - exactly the parameters that we are passing in. The compiler will generate code that calls this overload.

Overload 2 requires three integers.

When to use Method Overloading?

Generally, you should consider overloading a method when you for some reason need a couple of methods that take different parameters, but conceptually do the same thing. `Console.WriteLine()` is a good example. Another good example method is a method `Sign()` in the `System.Math` base class, which returns the sign of a number, and is overloaded to work out the sign of double, a decimal and other signed types.

In general, you should not use overloads when two methods really do different things - otherwise you'll simply confuse the developers who have to use your classes.

5.7 Method Overriding

When an instance method declaration includes an override modifier, the method is said to be an override method. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of the method.

It is an error for an override method declaration to include any one of the new, static, or virtual modifiers. An override method declaration may include the abstract modifier. This enables a virtual method to be overridden by an abstract method.

The method overridden by an override declaration is known as the overridden base method. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class of `C`, starting with the direct base class of `C` and continuing with each successive direct base class, until an accessible method with the same signature as `M` is located.



Notes For purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal, or if it is internal and declared in the same program as `C`.

Notes

1. A compile-time error occurs unless all of the following are true for an override declaration:
2. An overridden base method can be located as described above.
3. The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
4. The overridden base method is not a sealed method.
5. The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method.

An override declaration can access the overridden base method using a base-access.



Example:

```
class A
{
    int x;
    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}
class B: A
{
    int y;
    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}
```

In the above example, the `base.PrintFields()` invocation in B invokes the `PrintFields` method declared in A. A base-access disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in B been written `((A)this).PrintFields()`, it would recursively invoke the `PrintFields` method declared in B, not the one declared in A.

Only by including an override modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method.



Example:

```
class A
{
    public virtual void F() {}
}
class B: A
{
    public virtual void F() {} // Warning, hiding inherited F()
}
```

In the example, the F method in B does not include an override modifier and therefore does not override the F method in A. Rather, the F method in B hides the method in A, and a warning is reported because the declaration does not include a new modifier.



Example:

```
class A
{
    public virtual void F() {}
}
class B: A
{
    new private void F() {}           // Hides A.F within B
}
class C: B
{
    public override void F() {}     // Ok, overrides A.F
}
```

In the above example, the F method in B hides the virtual F method inherited from A. Since the new F in B has private access, its scope only includes the class body of B and does not extend to C. The declaration of F in C is therefore permitted to override the F inherited from A.



Task Answer the following question briefly

```
Static void ABC(ref int x, string a, out string b)
{ -----
}
```

Can we pass an array as parameter? Justify your answer with example.

Self Assessment

Fill in the blanks:

14. It is an error for an override method declaration to include any one of the new, static, or modifiers.
15. The override declaration and the overridden base method have the declared accessibility.



Caselet

Co-op Institute to Launch Integrated IT, GIS Course

The Co-operative Institute of Management and Technology (CIMAT), an affiliate of the Kerala State Cooperative Bank, will launch an advanced training programme in Integrated IT and Geographical Information Systems (GIS).

Contd...

Notes

This will be first such integrated programme in the State promising exposure to students to a wide range of IT skills and also offer a complete GIS curriculum as standard for all modules, an official spokesman said here.

Ties Up With CO

CIMAT has tied up with the Technopark-based Geotrans Technologies, a leading GIS specialist, to source the curriculum, faculty and software for the programme.

To be open for admission from Monday, October 17, 2011, the programme will induct 25 students in the first batch.

GIS integrates hardware, software and data for capturing, managing, analysing, and displaying all forms of geographically-referenced information.

It allows for viewing, understanding, questioning, interpreting, and visualising data in many ways to reveal relationships, patterns, and trends in the form of maps, reports, and charts.

The six-month course can be selected through a modular structure that includes ASP.NET, VB.NET, C#, Core Java and J2EE. The idea is to bring the best of facilities, faculties and course material on to one platform and offer students a package that is unmatched, said Dr Chandramohan, Director, CIMAT.

The programme will also enable students to get placed in reputed companies across the country.

It is open for all students who have either completed or are pursuing engineering, MCA or graduate-in-science courses.

According to Mr Sudeep K. V., Vice-President, Geotrans, this will give students access to some of the best in terms of resources and industry interface, which is extremely important for freshers.

Career Options

With ample machine time, guest lectures, live projects, access to industry and placement options, the programme will cater to the needs of the technical and engineering graduates.

According to Mr Sudeep, career opportunities for GIS professionals are huge and extremely rewarding.

Apart from the six-month training programme, we will also offer domain specific programmes in GIS to the industry.

This will be custom built on the specific needs of various sectors and agencies such as the water, electricity and telecom utilities, agriculture and healthcare, Dr Chandramohan added.

5.8 Summary

- Methods are the means through which an object's data members may be manipulated.
- Method should have proper declaration, header and body.
- The return type of modifier specifies the type of value computed.
- There must be at least one main method in one class of the entire application.
- Invoking the other method in the definition of one method is called nesting of methods.

- The four kind of formal parameters are value parameter, reference parameter, Output parameter, and output parameters.
- A parameter declared with a params modifier is a parameter array.
- An override method overrides an inherited virtual method with the same signature.
- The override declaration and the overridden base method have the same declared accessibility.
- An override declaration can access the overridden base method using a base access.

5.9 Keywords

Fixed Parameter: An optional set of attributes, an optional reference or out modifier, a type and an identifier is called fixed parameter.

Method: A method is a member that implements a computation or action that can be performed by an object or class.

Overridden Base Class: The method overridden by an override declaration is called as the overridden base class.

Override Method: When an instance method declaration includes an override modifier, the method is said to be an override method.

Parameter: Values passé in a function is called a parameter.

5.10 Review Questions

1. Declare a class having some methods and show how these methods may be invoked.
2. What will be the output of the following code?

```
class Test
{
    static void F(params object[] args) {
        foreach (object o in a) {
            Console.WriteLine(o.GetType().FullName);
            Console.WriteLine(" ");
        }
        Console.WriteLine();
    }
    static void Main() {
        object[] a = {1, "Delhi", 93.1};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

Notes

3. State one usage each of different forms of Main method.
4. How is a value type parameter different from a reference type parameter?
5. Explain the meaning of method overriding.
6. WAP to calculate the factorial of a number using recursion. Explain.
7. WAP to calculate the sum of array when array is passed as a parameter. Discuss.
8. The name and the formal parameter list of a method define the signature of the method. Explain with example.
9. What are the different variations to write Main() method? Write a program, which consists of multiple Main() methods.
10. An override method declaration may include the abstract modifier. Discuss.

Answer: Self Assessment

- | | |
|--------------------|-----------------------|
| 1. abstract | 2. parameters |
| 3. public | 4. return-type |
| 5. nested | 6. single-dimensional |
| 7. local variables | 8. attributes |
| 9. modifiers | 10. method |
| 11. expression | 12. params |
| 13. expanded | 14. virtual |
| 15. same | |

5.11 Further Readings



Books

C# and the .NET Platform, Second Edition by Andrew Troelsen.

C# Programmer's Handbook by Gregory S. MacBeth

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

<http://csharp.net-tutorials.com/classes/method-overloading/>

<http://www.csharp-station.com/Tutorials/Lesson05.aspx>

Unit 6: Classes & Objects in C#

Notes

CONTENTS

Objectives

Introduction

6.1 Defining Class

6.2 Class Members

6.3 Objects

6.4 Methods

6.5 Constructors

6.6 Destructors

6.7 Summary

6.8 Keywords

6.9 Review Questions

6.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the introduction to classes
- Describe Defining classes
- Demonstrate Objects
- Recognize Methods
- Explain Constructor and Destructor

Introduction

This unit will introduce you to classes, which are the fundamental topic to be discussed in any Object-Oriented Programming (OOP) language. A class is a combination of related objects whereas each object is an instance or a copy of the corresponding class. Also we will deal with Methods and the various types of methods will be discussed in detail. Methods are blocks of code that perform some kind of action, or carry out functions such as printing, opening a dialog box, and so forth.

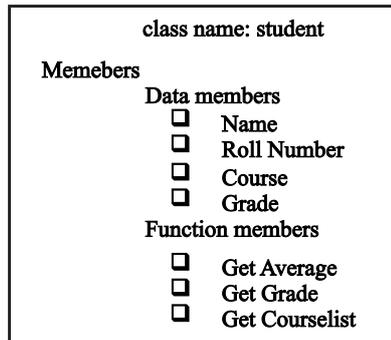
6.1 Defining Class

Class is the central concept of Object Oriented Approach in programming. In most of the object-oriented programming languages classes are implemented as a structure having additional features. Class implements the Object Oriented Approach in programming. A class is a data structure that may contain the following elements:

Notes

1. Data members such as constants, variables and events
2. Function members such as methods, properties, indexers, operators, constructors, and destructors
3. Nested types.

Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class. Consider the pictorial representation of a class named student.



A class-declaration consists of an optional set of attributes, followed by an optional set of class-modifiers, followed by the keyword `class` and an identifier that names the class, followed by an optional class-base specification, followed by a class-body, optionally followed by a semicolon.

A class-declaration syntax is given below.

```
class-modifiers class-name class-body
```

A class-modifier may be one or more of the following:

- `new`
- `public`
- `protected`
- `internal`
- `private`
- `abstract`
- `sealed`

The keyword `new` before a class name creates an object of the specified class type. If a class is declared as `public`, other classes can access it. Rest of the modifiers will be dealt with in the subsequent sections.

Class-name is any valid identifier that uniquely identifies each class. The class-body is group of statements enclosed within curly braces - { and }.

Thus, the student class may be written as:

```
Public class Student
{
    string name;
    int rollNo;
    string course;
    char grade;
```

```

    char GetGrade();

    float GetAverage();

    string[] GetCourseList();

}

```

Notes



Notes It is an error for the same modifier to appear multiple times in a class declaration.

The new modifier is only permitted on nested classes. It specifies that the class hides an inherited member by the same name. The public, protected, internal, and private modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers may not be permitted.

Abstract Classes

The abstract modifier is used to indicate that a class is incomplete and intended only to be a base class of other classes. An abstract class differs from a non-abstract class in the following ways:

1. An abstract class cannot be instantiated directly, and it is an error to use the new operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be null or contain references to instances of non-abstract classes derived from the abstract types.
2. An abstract class is permitted (but not required) to contain abstract members.
3. An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract members. Such implementations are provided by overriding the abstract members (see the following example).



Example:

```

abstract class AbstractOne
{
    public abstract void FunctionOne();
}

abstract class AbstractTwo:AbstractOne
{
    public void FunctionTwo() {}
}

class AbstractThree : AbstractTwo
{
    public override void FunctionOne()
    {
        // actual implementation of FunctionOne
    }
}

```

Notes

Here, the abstract class `AbstractOne` introduces an abstract method `FunctionOne`. Class `AbstractTwo` introduces an additional method `FunctionTwo`, but doesn't provide an implementation. `AbstractTwo` must therefore also be declared abstract. Class `AbstractThree` overrides `FunctionOne` and provides an actual implementation. Since there are no outstanding abstract members in `AbstractThree`, it is permitted (but not required) to be non-abstract.



Did u know? **What is the use of abstract modifier?**

The abstract modifier can be used with classes, methods, properties, indexers, and events.

Sealed Classes

The sealed modifier is used to prevent derivation from a class. An error occurs if a sealed class is specified as the base class of another class. A sealed class cannot be an abstract class.

The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

Class Base Specification

A class declaration may include a class-base specification which defines the direct base class of the class and the interfaces implemented by the class.

For instance, consider the following example.

```
Class DerivedClass : BaseClass, InterfaceName
{ }
```

Base Classes

When a class-type is included in the class-base, it specifies the direct base class of the class being declared. If a class declaration has no class-base, or if the class-base lists only interface types, the direct base class is assumed to be `object`. A class inherits members from its direct base class.

In the example code given below,

```
class One
{ }
class Two:One
{ }
```

`class One` is said to be the direct base class of `Two`, and `Two` is said to be derived from `One`. Since `One` does not explicitly specify a direct base class, its direct base class is implicitly `System.object`.

The direct base class of a class type must be at least as accessible as the class type itself. For example, it is an error for a public class to derive from a private or internal class.



Caution The direct base class of a class type must not be any of the following types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.

The base classes of a class are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. Referring to the example above, the base classes of B are A and object.

Except for class object, every class has exactly one direct base class. The object class has no direct base class and is the ultimate base class of all other classes.

When a class B derives from a class A, it is an error for A to depend on B. A class directly depends on its direct base class (if any) and directly depends on the class within which it is immediately nested (if any). Given this definition, the complete set of classes upon which a class depends is the transitive closure of the directly depends on relationship (see the example code given below).

```
class One: Two {}
class Two: Three {}
class Three: One {}
```

This is in error because the classes circularly depend on themselves. Likewise, the example,

```
class One: Two.Three {}
class Two: One
{
    public class Three {}
}
```

is in error because One depends on Two. Three (its direct base class), which depends on Two (its immediately enclosing class), which circularly depends on One. Note that a class does not depend on the classes that are nested within it. Consider the following example.

```
class One
{
    class Two: One {}
}
```

Class Two depends on One (because One is both its direct base class and its immediately enclosing class), but One does not depend on Two (since Two is neither a base class nor an enclosing class of One). Thus, the example is valid.

It is not possible to derive from a sealed class. Consider the example listed below.

```
sealed class One {}
class Two: One {}
```

This is an error. Class Two attempts to derive from the sealed class One which is not allowed.

A class-base specification may include a list of interface types, in which case the class is said to implement the given interface types.

Self Assessment

Fill in the blanks:

1. Class is the central concept of Approach in programming.
2. The keyword before a class name creates an object of the specified class type.

Notes

3. An abstract class cannot be instantiated directly, and it is an to use the new operator on an abstract class.
4. Except for class object, every class has exactly one class.

6.2 Class Members

The class-body of a class defines the members of the class. The members of a class consist of the members introduced by its class-member-declarations and the members inherited from the direct base class. The members of a class are divided into the following categories:

- Constants, which represent constant values associated with the class.
- Fields, which are the variables of the class.
- Methods, which implement the computations and actions that can be performed by the class.
- Properties, which define named attributes and the actions associated with reading and writing those attributes.
- Events, which define notifications that are generated by the class.
- Indexers, which permit instances of the class to be indexed in the same way as arrays.
- Operators, which define the expression operators that can be applied to instances of the class.
- Instance constructors, which implement the actions required to initialize instances of the class.
- Destructors, which implement the actions to perform before instances of the class are permanently discarded.
- Static constructors, which implement the actions required to initialize the class itself.
- Types, which represent the types that are local to the class.

Members that contain executable code are collectively known as the function members of the class.



Notes The function members of a class are the methods, properties, indexers, operators, constructors, and destructors of the class.

A class-declaration creates a new declaration space, and the class-member-declarations immediately contained by the class-declaration introduce new members into this declaration space. The following rules apply to class-member-declarations:

- Constructors and destructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- The name of a constant, field, property, event, or type must differ from the names of all other members declared in the same class.
- The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class.

- The signature of an constructor must differ from the signatures of all other constructors declared in the same class.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

The inherited members of a class are specifically not part of the declaration space of a class. Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member).

The New Modifier

A class-member-declaration is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to hide the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a new modifier to indicate that the derived member is intended to hide the base member.

If a new modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.



Caution It is an error to use the new and override modifiers in the same declaration.

Access Modifiers

A class-member-declaration can have any one of the five possible types of declared accessibility: public, protected internal, protected, internal, or private. Except for the protected internal combination, it is an error to specify more than one access modifier. When a class-member-declaration does not include any access modifiers, the declaration defaults to private declared accessibility.

Constituent Types

Types that are referenced in the declaration of a member are called the constituent types of the member. Possible constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or operator, and the parameter types of a method, indexer, operator, or constructor. The constituent types of a member must be at least as accessible as the member itself.

Static and Instance Members

Members of a class are either static members or instance members. Generally speaking, it is useful to think of static members as belonging to classes and instance members as belonging to objects (instances of classes).

When a field, method, property, event, operator, or constructor declaration includes a static modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member is referenced in a member-access of the form E.M, E must denote a type. It is an error for E to denote an instance.

Notes

- A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.
- A static function member (method, property, indexer, operator, or constructor) does not operate on a specific instance, and it is an error to refer to this in a static function member.

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a static modifier, it declares an instance member. An instance member is sometimes called a non-static member. Instance members have the following characteristics:

- When an instance member is referenced in a member-access of the form E.M, E must denote an instance. It is an error for E to denote a type.
- Very instance of a class contains a separate copy of all instance fields of the class.
- n instance function member (method, property accessor, indexer accessor, constructor, or destructor) operates on a given instance of the class, and this instance can be accessed as this.



Example: The rules for accessing static and instance members:

```
class TestClass
{
int x;

    static int y;
    void FunctionOne()
    {
        x = 1;           // Ok, same as this.x = 1
        y = 1;           // Ok, same as Test.y = 1
    }
    static void FunctionTwo()
    {
        x = 1;           // Error, cannot access this.x
        y = 1;           // Ok, same as Test.y = 1
    }
    static void Main()
    {
        TestClass t = new TestClass();
        t.x = 1;         // Ok
        t.y = 1;         // Error, cannot access static member
                        // through instance
        Test.x = 1;     // Error, cannot access instance member
                        // through type
        Test.y = 1;     // Ok
    }
}
```

The FunctionOne method shows that in an instance function member, a simple-name can be used to access both instance members and static members. The FunctionTwo method shows that in a static function member, it is an error to access an instance member through a simple-name. The Main method shows that in a member-access, instance members must be accessed through instances, and static members must be accessed through types.

Nested Types

Constants

A constant is a class member that represents a constant value. A constant type is a value that can be computed at compile-time. A constant-declaration introduces one or more constants of a given type.

A constant-declaration may include a set of attributes, a new modifier, and a valid combination of the four access modifiers. The attributes and modifiers apply to all of the members declared by the constant-declaration. Even though constants are considered static members, a constant-declaration neither requires nor allows a static modifier.

The type of a constant-declaration specifies the type of the members introduced by the declaration. The type is followed by a list of constant-declarators, each of which introduces a new member. A constant-declarator consists of an identifier that names the member, followed by an "=" token, followed by a constant-expression that gives the value of the member.

The type specified in a constant declaration must be sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, an enum-type, or a reference-type. Each constant-expression must yield a value of the target type or of a type that can be converted to the target type by an implicit conversion.



Task The type of a constant must be at least as accessible as the constant itself. Analyze.

A constant can itself participate in a constant-expression. Thus, a constant may be used in any construct that requires a constant-expression. Examples of such constructs include case labels, goto case statements, enum member declarations, attributes, and other constant declarations.

A constant-expression is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a reference-type other than string is to apply the new operator, and since the new operator is not permitted in a constant-expression, the only possible value for constants of reference-types other than string is null.

When a symbolic name for a constant value is desired, but when the type of the value is not permitted in a constant declaration or when the value cannot be computed at compile-time by a constant-expression, a read-only field may be used instead.

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type.



Example:

```
class One
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
```

Notes

```
}  
is equivalent to  
class One  
{  
    public const double X = 1.0;  
    public const double Y = 2.0;  
    public const double Z = 3.0;  
}
```

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. Consider the code listed hereunder.

```
class One  
{  
    public const int X = Two.Z + 1;  
    public const int Y = 10;  
}  
class Two  
{  
    public const int Z = A.Y + 1;  
}
```

The compiler first evaluates Y, then evaluates Z, and finally evaluates X, producing the values 10, 11, and 12. Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. Referring to the example above, if One and Two were declared in separate programs, it would be possible for One.X to depend on Two.Z, but Two.Z could then not simultaneously depend on One.Y.

Fields

A field is a member that represents a variable associated with an object or class. A field-declaration introduces one or more fields of a given type.

A field-declaration may include a set of attributes, a new modifier, a valid combination of the four access modifiers, a static modifier, and a read-only modifier. The attributes and modifiers apply to all of the members declared by the field-declaration.

The type of a field-declaration specifies the type of the members introduced by the declaration. The type is followed by a list of variable-declarators, each of which introduces a new member. A variable-declarator consists of an identifier that names the member, optionally followed by an "=" token and a variable-initializer that gives the initial value of the member.



Task The type of a field must be at least as accessible as the field itself. Explain with a example.

The value of a field is obtained in an expression using a simple-name or a member-access. The value of a field is modified using an assignment. The value of a field can be both obtained and

modified using postfix increment and decrement operators and prefix increment and decrement operators.

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type.



Example:

```
class One
{
    public static int X = 1, Y, Z = 100;
}
```

is equivalent to

```
class One
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}
```

Static and Instance Fields

When a field-declaration includes a static modifier, the fields introduced by the declaration are static fields. When no static modifier is present, the fields introduced by the declaration are instance fields. Static fields and instance fields are two of the several kinds of variables supported by C#, and are at times referred to as static variables and instance variables.

A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field. A static field comes into existence when the type in which it is declared is loaded, and ceases to exist when the type in which it is declared is unloaded.

Every instance of a class contains a separate copy of all instance fields of the class. An instance field comes into existence when a new instance of its class is created, and ceases to exist when there are no references to that instance and the destructor of the instance has executed.

When a field is referenced in a member-access of the form E.M, if M is a static field, E must denote a type, and if M is an instance field, E must denote an instance.

ReadOnly Fields

When a field-declaration includes a readonly modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor in the same class. Specifically, assignments to a readonly field are permitted only in the following contexts:

- In the variable-declarator that introduces the field (by including a variable-initializer in the declaration).
- Or an instance field, in the instance constructors of the class that contains the field declaration, or for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a readonly field as an out or ref parameter.

Notes

Attempting to assign to a readonly field or pass it as an out or ref parameter in any other context is an error.

A static readonly field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a const declaration or when the value cannot be computed at compile-time.



Example:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte red, green, blue;
    public Color(byte r, byte g, byte b)
    {
        red = r;
        green = g;
        blue = b;
    }
}
```

The Black, White, Red, Green, and Blue members cannot be declared as const members because their values cannot be computed at compile-time. However, declaring the members as static readonly fields has much the same effect.

Field Initialization

The initial value of a field is the default value of the field's type. When a class is loaded, all static fields are initialized to their default values, and when an instance of a class is created, all instance fields are initialized to their default values. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never "uninitialized".



Example:

```
class TestClass
{
    static bool b;
    int i;
    static void Main()
    {
        TestClass t = new TestClass();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

```
    }
}
```

This produces the output

```
b = False, i = 0
```

because `b` is automatically initialized to its default value when the class is loaded and `i` is automatically initialized to its default value when an instance of the class is created.

Variable Initializers

Field declarations may include variable-initializers. For static fields, variable initializers correspond to assignment statements that are executed when the class is loaded. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.



Example:

```
class TestClass
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";
    static void Main() {
        TestClass a = new TestClass();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

It produces the output

```
x = 1.414213562373095, i = 100, s = Hello
```

because an assignment to `x` occurs when the class is loaded and assignments to `i` and `s` occur when a new instance of the class is created.

The default value initialization occurs for all fields, including fields that have variable initializers. Thus, when a class is loaded, all static fields are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields are first initialized to their default values, and then the instance field initializers are executed in textual order.

It is possible for static fields with variable initializers to be observed in their default value state, though this is strongly discouraged as a matter of style (see the following example).



Example:

```
class TestClass
{
    static int a = b + 1;
    static int b = a + 1;
    static void Main() {
```

Notes

```
        Console.WriteLine("a = {0}, b = {1}", a, b);  
    }  
}
```

This program exhibits this behavior. Despite the circular definitions of a and b, the program is legal. It produces the output

a = 1, b = 2

because the static fields a and b are initialized to 0 (the default value for int) before their initializers are executed. When the initializer for a runs, the value of b is zero, and so a is initialized to 1. When the initializer for b runs, the value of a is already 1, and so b is initialized to 2.



Task Answers the following questions briefly

1. Name different types of members we can put in a class.
2. What do you understand by static and instance members?

Self Assessment

Fill in the blanks:

5. Static constructors, which implement the actions required to the class itself.
6. A class-member-declaration is permitted to declare a member with the same name or signature as an member.
7. The default value initialization occurs for all fields, including fields that have initializers.

6.3 Objects

A class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection. Client code is the code that uses these variables to call the methods and access the public properties of the object. In an object-oriented language such as C#, a typical program consists of multiple objects interacting dynamically.



Notes Static types behave differently than what is described here. For more information, see Static Classes and Static Class Members (C# Programming Guide).

Struct Instances vs. Class Instances

Because classes are reference types, a variable of a class object holds a reference to the address of the object on the managed heap. If a second object of the same type is assigned to the first object, then both variables refer to the object at that address. This point is discussed in more detail later in this topic.

Instances of classes are created by using the new operator.



Example: Person is the type and person1 and person 2 are instances, or objects, of that type.

C#

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    //Other properties, methods, events...
}
class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);
        // Declare new person, assign person1 to it.
        Person person2 = person1;
        //Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;
        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);
        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
```

Output:

```
person1 Name = Leopold Age = 6
person2 Name = Molly Age = 16
person1 Name = Molly Age = 16
*/
```

Notes

Because structs are value types, a variable of a struct object holds a copy of the entire object. Instances of structs can also be created by using the newoperator, but this is not required, as shown in the following example:



Example:

```
C#
public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);
        // Create new struct object. Note that struct can be initialized
        // without using "new".
        Person p2 = p1;
        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);
        // p1 values remain unchanged because p2 is copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);
        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
```

Output:

Notes

```
p1 Name = Alex Age = 9
p2 Name = Spencer Age = 7
p1 Name = Alex Age = 9
*/
```

The memory for both p1 and p2 is allocated on the thread stack. That memory is reclaimed along with the type or method in which it is declared. This is one reason why structs are copied on assignment. By contrast, the memory that is allocated for a class instance is automatically reclaimed (garbage collected) by the common language runtime when all references to the object have gone out of scope. It is not possible to deterministically destroy a class object like you can in C++. For more information about garbage collection in the .NET Framework, see Garbage Collection.



Notes The allocation and deallocation of memory on the managed heap is highly optimized in the common language runtime. In most cases there is no significant difference in the performance cost of allocating a class instance on the heap versus allocating a struct instance on the stack.

Object Identity vs. Value Equality

When you compare two objects for equality, you must first distinguish whether you want to know whether the two variables represent the same object in memory, or whether the values of one or more of their fields are equivalent. If you are intending to compare values, you must consider whether the objects are instances of value types (structs) or reference types (classes, delegates, arrays).

- To determine whether two class instances refer to the same location in memory (which means that they have the same identity), use the `staticEquals` method. (`System.Object` is the implicit base class for all value types and reference types, including user-defined structs and classes.)
- To determine whether the instance fields in two struct instances have the same values, use the `ValueType.Equals` method. Because all structs implicitly inherit from `System.ValueType`, you call the method directly on your object as shown in the following example:

```
C#
Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;
if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");
// Output: p2 and p1 have the same values.
```

The `System.ValueType` implementation of `Equals` uses reflection because it must be able to determine what the fields are in any struct. When creating your own structs, override the `Equals` method to provide an efficient equality algorithm that is specific to your type.

Notes

- To determine whether the values of the fields in two class instances are equal, you might be able to use the Equals method or the == operator. However, only use them if the class has overridden or overloaded them to provide a custom definition of what “equality” means for objects of that type. The class might also implement the IEquatable(Of T) interface or the IEqualityComparer(Of T) interface. Both interfaces provide methods that can be used to test value equality. When designing your own classes that override Equals, make sure to follow the guidelines stated in How to: Define Value Equality for a Type (C# Programming Guide) and Object.Equals(Object).

Self Assessment

Fill in the blanks:

8. Client code is the code that uses these variables to call the methods and access the properties of the object.
9. To determine whether the instance fields in two instances have the same values, use the ValueType.Equals method.

6.4 Methods

A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using method-declarations. A method-declaration may include a set of attributes, a new modifier, an extern modifier, a valid combination of the four access modifiers, and a valid combination of the static, virtual, override, and abstract modifiers. In addition, a method that includes the override modifier may also include the sealed modifier.

The static, virtual, override, and abstract modifiers are mutually exclusive except in one case. The abstract and override modifiers may be used together so that an abstract method can override a virtual one.



Did u know? **What does the return-type of a method declaration specifies?**

The return-type of a method declaration specifies the type of the value computed and returned by the method. The return-type is void if the method does not return a value.

The member-name specifies the name of the method. Unless the method is an explicit interface member implementation, the member-name is simply an identifier. For an explicit interface member implementation, the member-name consists of an interface-type followed by a “.” and an identifier. The optional formal-parameter-list specifies the parameters of the method.

The return-type and each of the types referenced in the formal-parameter-list of a method must be at least as accessible as the method itself. For abstract and extern methods, the method-body consists simply of a semicolon. For all other methods, the method-body consists of a block which specifies the statements to execute when the method is invoked.

The name and the formal parameter list of a method defines the signature of the method. Specifically, the signature of a method consists of its name and the number, modifiers, and types of its formal parameters. The return type is not part of a method’s signature, nor are the names of the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class.

Value Parameters

Notes

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter—they have no effect on the actual argument given in the method invocation.

Reference Parameters

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword `ref` followed by a variable-reference of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.



Example:

```
class TestClass
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

It produces the output

```
i = 2, j = 1
```

For the invocation of `Swap` in `Main`, `x` represents `i` and `y` represents `j`. Thus, the invocation has the effect of swapping the values of `i` and `j`.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location.

Notes



Example:

```
class One
{
    string s;
    void FunctionOne(ref string a, ref string b)
    {
        s = "One";
        a = "Two";
        b = "Three";
    }
    void FunctionTwo()
    {
        FunctionOne(ref s, ref s);
    }
}
```

The invocation of `FunctionOne` in `FunctionTwo` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`.

Output Parameters

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a variable-reference of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns. Output parameters are typically used in methods that produce multiple return values.



Example:

```
class TestClass
{
    static void SplitPath(string path, out string dir, out string name)
    {
        int i = path.Length;
        while (i > 0)
```

```

        {
            char ch = path[i - 1];
            if (ch == '\\\' || ch == '/' || ch == ':')
                break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }
}

static void Main()
{
    string dir, name;
    SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
    Console.WriteLine(dir);
    Console.WriteLine(name);
}
}

```

The example produces the output:

```

c:\Windows\System\
hello.txt

```



Notes The dir and name variables can be unassigned before they are passed to SplitPath, and that they are considered definitely assigned following the call.

Parameter Arrays

A parameter declared with a params modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` can not. It is not possible to combine the params modifier with the ref and out modifiers.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression of a type that is implicitly convertible to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Notes

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter of the same type.



Example:

```
class TestClass
{
    static void FunctionOne(params int[] args)
    {
        Console.WriteLine("Array contains {0} elements:", args.Length);
        foreach (int i in args) Console.Write(" {0}", i);
        Console.WriteLine();
    }
    static void Main()
    {
        int[] a = {1, 2, 3};
        FunctionOne(a);
        FunctionOne(10, 20, 30, 40);
        FunctionOne();
    }
}
```

It produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `FunctionOne` simply passes the array `a` as a value parameter. The second invocation of `FunctionOne` automatically creates a four-element `int[]` with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `FunctionOne` creates a zero-element `int[]` and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
FunctionOne(new int[] {10, 20, 30, 40});
FunctionOne(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form. The expanded form of a method is available only if the normal form of the method is not applicable and only if a method with the same signature as the expanded form is not already declared in the same type.



Example:

```
class TestClass
{
    static void FunctionOne(params object[] a)
    {
```

```

        Console.WriteLine("FunctionOne(object[])");
    }
    static void FunctionOne()
    {
        Console.WriteLine("FunctionOne()");
    }
    static void FunctionOne(object a0, object a1)
    {
        Console.WriteLine("FunctionOne(object,object)");
    }
    static void Main()
    {
        FunctionOne();
        FunctionOne(1);
        FunctionOne(1, 2);
        FunctionOne(1, 2, 3);
        FunctionOne(1, 2, 3, 4);
    }
}

```

It produces the output

```

FunctionOne();
FunctionOne(object[]);
FunctionOne(object,object);
FunctionOne(object[]);
FunctionOne(object[]);

```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expanded form for a single object parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.



Example:

```

class TestClass
{
    static void FunctionOne(params object[] args)

```

Notes

```
{
    foreach (object o in a)
    {
        Console.WriteLine(o.GetType().FullName);
        Console.WriteLine(" ");
    }
    Console.WriteLine();
}
static void Main()
{
    object[] a = {1, "Hello", 123.456};
    object o = a;
    FunctionOne(a);
    FunctionOne((object)a);
    FunctionOne(o);
    FunctionOne((object[])o);
}
}
```

It produces the following output.

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of `FunctionOne`, the normal form of `FunctionOne` is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type `object[]`). Thus, overload resolution selects the normal form of `FunctionOne`, and the argument is passed as a regular value parameter.

In the second and third invocations, the normal form of `FunctionOne` is not applicable because no implicit conversion exists from the argument type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the expanded form of `FunctionOne` is applicable, and it is therefore selected by overload resolution. As a result, a one-element `object[]` is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an `object[]`).

Static and Instance Methods

When a method declaration includes a static modifier, the method is said to be a static method. When no static modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is an error to refer to this in a static method. It is furthermore an error to include a virtual, abstract, or override modifier on a static method.

An instance method operates on a given instance of a class, and this instance can be accessed as `this`.

Virtual Methods

Notes

When an instance method declaration includes a virtual modifier, the method is said to be a virtual method. When no virtual modifier is present, the method is said to be a non-virtual method.

It is an error for a method declaration that includes the virtual modifier to also include any one of the static, abstract, or override modifiers.

The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be changed by derived classes. The process of changing the implementation of an inherited virtual method is known as overriding the method.

In a virtual method invocation, the run-time type of the instance for which the invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the compile-time type of the instance is the determining factor. In precise terms, when a method named *N* is invoked with an argument list *A* on an instance with a compile-time type *C* and a run-time type *R* (where *R* is either *C* or a class derived from *C*), the invocation is processed as follows:

- First, overload resolution is applied to *C*, *N*, and *A*, to select a specific method *M* from the set of methods declared in and inherited by *C*.
- Then, if *M* is a non-virtual method, *M* is invoked.
- Otherwise, *M* is a virtual method, and the most derived implementation of *M* with respect to *R* is invoked.

Or every virtual method declared in or inherited by a class, there exists a most derived implementation of the method with respect to that class. The most derived implementation of a virtual method *M* with respect to a class *R* is determined as follows:

- If *R* contains the introducing virtual declaration of *M*, then this is the most derived implementation of *M*.
- Otherwise, if *R* contains an override of *M*, then this is the most derived implementation of *M*.
- Otherwise, the most derived implementation of *M* is the same as that of the direct base class of *R*.



Example: The differences between virtual and non-virtual methods:

```
class One
{
    public void FunctionOne()
    {
        Console.WriteLine("One. FunctionOne");
    }
    public virtual void FunctionTwo()
    {
        Console.WriteLine("One. FunctionTwo");
    }
}
```

Notes

```

    }
    class Two: One
    {
        new public void FunctionOne()
        {
            Console.WriteLine("Two. FunctionOne");
        }
        public override void FunctionTwo()
        {
            Console.WriteLine("Two. FunctionTwo");
        }
    }
    class TestClass
    {
        static void Main()
        {
            Two b = new Two();
            One a = b;
            a.FunctionOne();
            b.FunctionOne();
            a.FunctionTwo();
            b.FunctionTwo();
        }
    }
}

```

In the example, One introduces a non-virtual method FunctionOne and a virtual method FunctionTwo. The class FunctionTwo introduces a new non-virtual method FunctionOne, thus hiding the inherited FunctionOne, and also overrides the inherited method FunctionTwo. The example produces the output:

```

One.FunctionOne
Two.FunctionOne
Two.FunctionTwo
Two.FunctionTwo

```



Notes The statement `One.FunctionTwo()` invokes `Two.FunctionTwo`, not `one.FunctionTwo`. This is because the run-time type of the instance (which is `Two`), not the compile-time type of the instance (which is `One`), determines the actual method implementation to invoke.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. Consider the following example.

```
class One
{
    public virtual void FunctionOne()
    {
        Console.WriteLine("One.FunctionOne");
    }
}
class Two: One
{
    public override void FunctionOne()
    {
        Console.WriteLine("Two.FunctionOne");
    }
}
class Three: Two
{
    new public virtual void FunctionOne()
    {
        Console.WriteLine("Three.FunctionOne");
    }
}
class Four: Three
{
    public override void FunctionOne()
    {
        Console.WriteLine("Four.FunctionOne");
    }
}
class TestClass
{
    static void Main()
    {
        Four d = new Four();
        One a = d;
        Two b = d;
        Three c = d;
        a.FunctionOne();
        b.FunctionOne();
    }
}
```

Notes

```

        c.FunctionOne();
        d.FunctionOne();
    }
}

```

Classes Three and Four classes contain two virtual methods with the same signature: The one introduced by One and the one introduced by Three. The method introduced by Three hides the method inherited from One. Thus, the override declaration in Four overrides the method introduced by Three, and it is not possible for Four to override the method introduced by One. The example produces the output:

```

Two.FunctionOne
Two.FunctionOne
Four.FunctionOne
Four.FunctionOne

```



Notes It is possible to invoke the hidden virtual method by accessing an instance of D through a less derived type in which the method is not hidden.

Override Methods

When an instance method declaration includes an override modifier, the method is said to be an override method. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of the method.

It is an error for an override method declaration to include any one of the new, static, or virtual modifiers. An override method declaration may include the abstract modifier. This enables a virtual method to be overridden by an abstract method.

The method overridden by an override declaration is known as the overridden base method. For an override method M declared in a class C, the overridden base method is determined by examining each base class of C, starting with the direct base class of C and continuing with each successive direct base class, until an accessible method with the same signature as M is located. For purposes of locating the overridden base method, a method is considered accessible if it is public, if it is protected, if it is protected internal, or if it is internal and declared in the same program as C.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method.

An override declaration can access the overridden base method using a base-access.



Example:

```
class One
{
    int x;
    public virtual void PrintFields()
    {
        Console.WriteLine("x = {0}", x);
    }
}

class Two: One
{
    int y;
    public override void PrintFields()
    {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}
```

Here the `base.PrintFields()` invocation in `Two` invokes the `PrintFields` method declared in `One`. A base-access disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in `Two` been written `((One)this).PrintFields()`, it would recursively invoke the `PrintFields` method declared in `Two`, not the one declared in `One`.

Only by including an override modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method.



Example:

```
class One
{
    public virtual void FunctionOne()
    {}
}

class Two: One
{
    public virtual void FunctionOne()
    {} // Warning, hiding inherited FunctionOne()
}
```

The `FunctionOne` method in `Two` does not include an override modifier and therefore does not override the `FunctionOne` method in `One`. Rather, the `FunctionOne` method in `Two` hides the method in `One`, and a warning is reported because the declaration does not include a new modifier.

Notes



Example:

```
class One
{
    public virtual void FunctionOne()
    {}
}
class Two: One
{
    new private void FunctionOne()
    {} // Hides One.FunctionOne within Two
}
class Three: Two
{
    public override void FunctionOne()
    {} // Ok, overrides One.FunctionOne
}
```

The FunctionOne method in Two hides the virtual FunctionOne method inherited from One. Since the new FunctionOne in Two has private access, its scope only includes the class body of Two and does not extend to Three. The declaration of FunctionOne in Three is, therefore, permitted to override the FunctionOne inherited from One.

Sealed Methods

When an instance method declaration includes a sealed modifier, the method is said to be a sealed method. A sealed method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of the method.

An override method can also be marked with the sealed modifier. Use of this modifier prevents a derived class from further overriding the method. The sealed modifier can only be used in combination with the override modifier.



Example:

```
class One
{
    public virtual void FunctionOne()
    {
        Console.WriteLine("One.FunctionOne");
    }
    public virtual void FunctionTwo()
    {
```

```
        Console.WriteLine("One.FunctionTwo");
    }
}
class Two: One
{
    sealed override public void FunctionOne()
    {
        Console.WriteLine("Two.FunctionOne");
    }
    override public void FunctionTwo()
    {
        Console.WriteLine("Two.FunctionTwo");
    }
}
class Three: Two
{
    override public void FunctionTwo()
    {
        Console.WriteLine("Three.FunctionTwo");
    }
}
```

Here, the class Two provides two override methods: an FunctionOne method that has the sealed modifier and a FunctionTwo method that does not. Two's use of the sealed modifier prevents Three from further overriding FunctionOne.

Abstract Methods

When an instance method declaration includes an abstract modifier, the method is said to be an abstract method. An abstract method is implicitly also a virtual method.

An abstract method declaration introduces a new virtual method but does not provide an implementation of the method. Instead, non-abstract derived classes are required to provide their own implementation by overriding the method. Because an abstract method provides no actual implementation, the method-body of an abstract method simply consists of a semicolon.

Abstract method declarations are only permitted in abstract classes. It is an error for an abstract method declaration to include either the static or virtual modifiers.



Example:

```
public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
```

Notes

```

{
    public override void Paint(Graphics g, Rectangle r)
    {
        g.drawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r)
    {
        g.drawRect(r);
    }
}

```

Here, the Shape class defines the abstract notion of a geometrical shape object that can paint itself. The Paint method is abstract because there is no meaningful default implementation. The Ellipse and Box classes are concrete Shape implementations. Because these classes are non-abstract, they are required to override the Paint method and provide an actual implementation.

It is an error for a base-access to reference an abstract method.



Example:

```

class One
{
    public abstract void FunctionOne();
}

class Two: One
{
    public override void FunctionOne()
    {
        base.FunctionOne();
        // Error, base.FunctionOne is abstract
    }
}

```

Here, an error is reported for the base.FunctionOne() invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. Consider the following code.

```

class One
{
    public virtual void FunctionOne()

```

```

    {
        Console.WriteLine("One.FunctionOne");
    }
}
abstract class Two: One
{
    public abstract override void FunctionOne();
}
class Three: Two
{
    public override void FunctionOne()
    {
        Console.WriteLine("THree.FunctionOne");
    }
}

```

Here, the class One declares a virtual method, the class Two override this method with an abstract method, and the class Three overrides to provide its own implementation.

External Methods

When a method declaration includes an extern modifier, the method is said to be an external method. External methods are implemented externally, using a language other than C#. Because an external method declaration provides no actual implementation, the method-body of an external method simply consists of a semicolon.

The extern modifier is typically used in conjunction with a DllImport attribute, allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

It is an error for an external method declaration to also include the abstract modifier. When an external method includes a DllImport attribute, the method declaration must also include a static modifier.

The example listed below demonstrates use of the extern modifier and the DllImport attribute.

```

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttributes sa);
    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);
    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);
    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}

```

Notes

Method Body

The method-body of a method declaration consists either of a block or a semicolon. Abstract and external method declarations do not provide a method implementation, and the method body of an abstract or external method simply consists of a semicolon. For all other methods, the method body is a block that contains the statements to execute when the method is invoked.

When the return type of a method is void, return statements in the method body are not permitted to specify an expression. If execution of the method body of a void method completes normally (that is, if control flows off the end of the method body), the method simply returns to the caller.

When the return type of a method is not void, each return statement in the method body must specify an expression of a type that is implicitly convertible to the return type. Execution of the method body of a value-returning method is required to terminate in a return statement that specifies an expression, or in a throw statement that throws an exception. It is an error if execution of the method body can complete normally. In other words, in a value-returning method, control is not permitted to flow off the end of the method body.



Example:

```
class One
{
    public int FunctionOne()
    {
        // Error, return value required
    }
    public int FunctionTwo()
    {
        return 1;
    }
    public int FunctionThree(bool b)
    {
        if (b)
        {
            return 1;
        }
        else
        {
            return 0;
        }
    }
}
```

Here, the value-returning FunctionOne method is in error because control can flow off the end of the method body. The FunctionTwo and FunctionThree methods are correct because all possible execution paths end in a return statement that specifies a return value.



Task Analyze and compare Sealed Method and Abstract Method.

Self Assessment

Notes

Fill in the blanks:

10. A parameter declared with an out modifier is an parameter.
11. A variable must be definitely assigned before it can be passed as a parameter.
12. When the return type of a method is void, return statements in the method body are not permitted to specify an.....
13. When a method declaration includes an extern modifier, the method is said to be an method.

6.5 Constructors

An instance constructor is a member that implements the actions required to initialize an instance of a class. Constructors are declared using constructor-declarations. A constructor-declaration may include a set of attributes and a valid combination of the four access modifiers.

The identifier of a constructor-declarator must name the class in which the constructor is declared. If any other name is specified, an error occurs. The optional formal-parameter-list of a constructor is subject to the same rules as the formal-parameter-list of a method. The formal parameter list defines the signature of a constructor and governs the process whereby overload resolution selects a particular constructor in an invocation. Each of the types referenced in the formal-parameter-list of a constructor must be at least as accessible as the constructor itself.

The optional constructor-initializer specifies another constructor to invoke before executing the statements given in the block of this constructor.

The block of a constructor declaration specifies the statements to execute in order to initialize a new instance of the class. This corresponds exactly to the block of an instance method with a void return type. Constructors are not inherited. Thus, a class has no other constructors than those that are actually declared in the class. If a class contains no constructor declarations, a default constructor is automatically provided.



Notes Constructors are invoked by object-creation-expressions and through constructor-initializers.

Constructor Initializers

All constructors (except for the constructors of class object) implicitly include an invocation of another constructor immediately before the first statement in the block of the constructor. The constructor to implicitly invoke is determined by the constructor-initializer:

- A constructor initializer of the form `base(...)` causes a constructor from the direct base class to be invoked. The set of candidate constructors consists of all accessible constructors declared in the direct base class. If the set of candidate constructors is empty, or if a single best constructor cannot be identified, an error occurs.
- A constructor initializer of the form `this(...)` causes a constructor from the class itself to be invoked. The set of candidate constructors consists of all accessible constructors declared in the class itself. If the set of candidate constructors is empty, or if a single best constructor

Notes

cannot be identified, an error occurs. If a constructor declaration includes a constructor initializer that invokes the constructor itself, an error occurs.

If a constructor has no constructor initializer, a constructor initializer of the form `base()` is implicitly provided. Thus, a constructor declaration of the form

```
C(...) {...}
```

is exactly equivalent to

```
C(...): base() {...}
```

The scope of the parameters given by the formal-parameter-list of a constructor declaration includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor.



Example:

```
class One
{
    public One(int x, int y)
}
}
class Two: One
{
    public Two(int x, int y): base(x + y, x - y)
}
}
```

A constructor initializer cannot access the instance being created. It is therefore an error to reference `this` in an argument expression of the constructor initializer, as it is an error for an argument expression to reference any instance member through a simple-name.

Constructor Execution

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the first statement in the block of a constructor.



Example:

```
using System.Collections;
class One
{
    int x = 1, y = -1, count;
    public One()
    {
        count = 0;
    }
    public One(int n)
```

Notes

```

{
    count = n;
}
}
class Two: One
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;
    public Two(): this(100)
{
    items.Add("default");
}
    public Two(int n): base(n - 1)
{
    max = n;
}
}

```

This example contains several variable initializers and also contains constructor initializers of both forms (base and this). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations isn't valid, but merely serves to illustrate the mechanism).

```

using System.Collections;
class One
{
    int x, y, count;
    public One()
    {
        x = 1;           // Variable initializer
        y = -1;         // Variable initializer
        object();       // Invoke object() constructor
        count = 0;
    }
    public A(int n)
    {
        x = 1;           // Variable initializer
        y = -1;         // Variable initializer
        object();       // Invoke object() constructor
        count = n;
    }
}

```

Notes

```
    }  
}  
class Two: One  
{  
    double sqrt2;  
    ArrayList items;  
    int max;  
    public Two(): this(100)  
    {  
        Two(100);                // Invoke Two(int) constructor  
        items.Add("default");  
    }  
    public Two(int n): base(n - 1)  
    {  
        sqrt2 = Math.Sqrt(2.0);    // Variable initializer  
        items = new ArrayList(100); // Variable initializer  
        One(n - 1);                // Invoke One(int) constructor  
        max = n;  
    }  
}
```



Notes Variable initializers are transformed into assignment statements, and that these assignment statements are executed before the invocation of the base class constructor. This ordering ensures that all instance fields are initialized by their variable initializers before any statements that have access to the instance are executed.



Example:

```
class One  
{  
    public One()  
{  
        PrintFields();  
    }  
    public virtual void PrintFields()  
{  
}  
}  
class Two: One  
{
```

```
        int x = 1;
        int y;
public Two()
{
    y = -1;
}
public override void PrintFields()
{
    Console.WriteLine("x = {0}, y = {1}", x, y);
}
}
```

When new Two() is used to create an instance of Two, the following output is produced:

```
x = 1, y = 0
```

The value of x is 1 because the variable initializer is executed before the base class constructor is invoked. However, the value of y is 0 (the default value of an int) because the assignment to y is not executed until after the base class constructor returns.

Default Constructors

If a class contains no constructor declarations, a default constructor is automatically provided. The default constructor simply invokes the parameterless constructor of the direct base class. If the direct base class does not have an accessible parameterless constructor, an error occurs. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. Thus, the default constructor is always of the form

```
protected C(): base() {}
```

or

```
public C(): base() {}
```

where C is the name of the class.



Example:

```
class Message
{
    object sender;
    string text;
}
```

Here, a default constructor is provided because the class contains no constructor declarations. Thus, the example is precisely equivalent to

```
class Message
{
    object sender;
    string text;
```

Notes

```
public Message(): base()
{
}
}
```

Private Constructors

When a class declares only private constructors it is not possible for other classes to derive from the class or create instances of the class (an exception being classes nested within the class). Private constructors are commonly used in classes that contain only static members.



Example:

```
public class Trig
{
    private Trig()
    {} // Prevent instantiation
    public const double PI = 3.14159265358979323846;
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

The Trig class provides a grouping of related methods and constants, but is not intended to be instantiated. It therefore declares a single private constructor. At least one private constructor must be declared to suppress the automatic generation of a default constructor.

Optional Constructor Parameters

The this(...) form of constructor initializer is commonly used in conjunction with overloading to implement optional constructor parameters.



Example:

```
class Text
{
    public Text(): this(0, 0, null)
    {}
    public Text(int x, int y): this(x, y, null)
    {}
    public Text(int x, int y, string s)
    {
        // Actual constructor implementation
    }
}
```

Here, the first two constructors merely provide the default values for the missing arguments. Both use a this(...) constructor initializer to invoke the third constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

```
Text t1 = new Text(); // Same as Text(0, 0, null)
Text t2 = new Text(5, 10); // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

Notes

Static Constructors

A static constructor is a member that implements the actions required to initialize a class. Static constructors are declared using static-constructor-declarations. A static-constructor-declaration may include a set of attributes.

The identifier of a static-constructor-declaration must name the class in which the static constructor is declared. If any other name is specified, an error occurs. The block of a static constructor declaration specifies the statements to execute in order to initialize the class. This corresponds exactly to the block of a static method with a void return type. Static constructors are not inherited.

Class loading is the process by which a class is prepared for use in the runtime environment. The loading process is mostly implementation-dependent, though several guarantees are provided:

- A class is loaded before any instance of the class is created.
- A class is loaded before any of its static members are referenced.
- A class is loaded before any types that derive from it are loaded.
- A class cannot be loaded more than once during a single execution of a program.
- If a class has a static constructor then it is automatically called when the class is loaded. Static constructors cannot be invoked explicitly.



Example:

```
class TestClass
{
    static void Main()
    {
        One.FunctionOne();
        Two.FunctionOne();
    }
}
class One
{
    static One()
    {
        Console.WriteLine("Init One");
    }
}
public static void FunctionOne()
{
    Console.WriteLine("One.FunctionOne");
}
```

Notes

```
}  
class Two  
{  
    static Two()  
{  
    Console.WriteLine("Init Two");  
    }  
public static void FunctionOne()  
{  
    Console.WriteLine("Two.FunctionOne");  
    }  
}
```

It could produce either the output:

```
Init One  
One.FunctionOne  
Init Two  
Two.FunctionOne
```

or the output:

```
Init Two  
Init One  
One.FunctionOne  
Two.FunctionOne
```

because the exact ordering of loading and therefore of static constructor execution is not defined.



Example:

```
class TestClass  
{  
    static void Main()  
{  
    Console.WriteLine("1");  
    Two.FunctionTwo();  
    Console.WriteLine("2");  
    }  
}  
class One  
{  
    static One()  
{  
    Console.WriteLine("Init One");  
    }  
}
```

```

    }
}
class Two: One
{
    static Two()
    {
        Console.WriteLine("Init Two");
    }
    public static void FunctionTwo()
    {
        Console.WriteLine("Two.FunctionTwo");
    }
}

```

It is guaranteed to produce the output:

```

Init One
Init Two
Two.FunctionTwo

```

because the static constructor for the class One must execute before the static constructor of the class Two, which derives from it.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.



Example:

```

class One
{
    public static int X = Two.Y + 1;
}
class Two
{
    public static int Y = One.X + 1;
}
static void Main()
{
    Console.WriteLine("X = {0}, Y = {1}", One.X, Two.Y);
}
}

```

It produces the output

```
X = 1, Y = 2
```

To execute the Main method, the system first loads class Two. The static constructor of Two proceeds to compute the initial value of Y, which recursively causes One to be loaded because the value of One.X is referenced. The static constructor of One in turn proceeds to compute the initial

Notes

value of X, and in doing so fetches the default value of Y, which is zero. One.X is thus initialized to 1. The process of loading One then completes, returning to the calculation of the initial value of Y, the result of which becomes 2.

Had the Main method instead been located in class One, the example would have produced the output

X = 2, Y = 1

Circular references in static field initializers should be avoided since it is generally not possible to determine the order in which classes containing such references are loaded.

6.6 Destructors

A destructor is a member that implements the actions required to destruct an instance of a class. Destructors are declared using destructor-declarations. A destructor-declaration may include a set of attributes.

The identifier of a destructor-declarator must name the class in which the destructor is declared. If any other name is specified, an error occurs.

The block of a destructor declaration specifies the statements to execute in order to destruct an instance of the class. This corresponds exactly to the block of an instance method with a void return type. Destructors are not inherited. Thus, a class has no other destructors than those that are actually declared in the class.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use the instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in an inheritance chain are called in order, from most derived to least derived.

Self Assessment

Fill in the blanks:

- 14. When a class declares only constructors it is not possible for other classes to derive from the class or create instances of the class.
- 15. The of a destructor-declarator must name the class in which the destructor is declared.



Caselet

Arithnet Technical offers Model-based Testing Tools

ARITHNET Technical Services, a subsidiary of Denmark-based ATS, providing Model-based Testing (MBT) solutions, is in the process of expanding its Indian operations. This will see a gradual ramp-up of its numbers to move to a support centre. The company plans to offer its testing solutions in an outsourced process model.

The leader of RedVest group of the Institute For Systems Programming of the Russian Academy of Sciences, Mr Alexander K. Petrenko, who is associated with product development in this space, told Business Line that the global market was gradually emerging towards a model-based testing approach. This would offer a whole range of applications for the technology products companies.

Contd...

Mr Balaswamy, General Manager of Arithnet, said there had been significant growth in MBT in the last few years due to the popularity of object oriented programming and models in software engineering. Explaining the advantages of MBT, Prof. Petrenko said in the traditional testing, a model of the application to be tested is implicit only in the testers mind. MBT takes the model out of the testers mind making it more useful for multiple testing tasks, shareable, reusable and describing more precisely the system to be tested.

These toolkits are integrated into popular integrated development environments such as Visual Studio from Microsoft or Forte from Sun Microsystems and support all phases of MBT model based testing– specifications, test design, test case generation from specifications, test execution and automatic analysis of outcome.

ATS has entered into a technical collaboration with the Institute for Systems Programming of the Russian Academy of Sciences, Moscow, for development of several software testing related products. While reaching out its MBT solutions, ATS provides services right from consultancy services, outsourced testing services as also training for corporations. “We are hosting interactive meetings with technology companies in Hyderabad to enable them to benefit from our solutions,” Mr Immanuel Selvaraj, International Marketing Manager, Arithnet, said.

6.7 Summary

- An abstract class cannot instantiate directly.
- Class A is said to be the direct base class of B if B is inherited from A.
- A static method does not operate on a specific instance, and it is an error to refer to this in a static method.
- When an instance method declaration includes an override modifier, the method is said to be an override method.
- An abstract method is implicitly also a virtual method.
- A property that includes both a get accessor and a set accessor is said to be read -write property.
- A property can be a static member whereas an indexer is always an instance member.
- A delegates declaration defines a class that extends the class System.Delegates.
- Delegates types are classified into two kinds: combinable and non-combinable.
- Once instantiated, delegates instances always refer to the same target object and method.

6.8 Keywords

Class: A set of related objects is called a class.

Data Abstraction: It refers to the act of representing essential features without including the background details or explanations.

Dynamic Binding: It refers to the linking of a procedure call to the code to be executed in response to the call.

Event: An event is a member that enables an object or class to provide notification.

Field: A field is a member that represent a variable associated with an object or class.

Notes

Indexer: An indexer is a member that enables an object to be indexed in the same way as an array.

Inheritance: The ability of a class to inherit the properties of another class is called inheritance.

Object: Objects are the basic runtime entity in an object-oriented system.

Polymorphism: Capability of the data to be processed in more than one form is called polymorphism.

Property: A property is a member that provides access to an attribute of an object or a class.

6.9 Review Questions

1. Substantiate the role of a constructor method in a class with example.
2. How is a static member different from other members?
3. Can you allow class to be inherited, but prevent the method from being over-ridden?
4. Declare a class having some methods and show how these methods may be invoked.
5. What will be the output of the following code?

```
class Test
{
    static void F(params object[] args)
    {
        foreach (object o in a)
        {
            Console.WriteLine(o.GetType().FullName);
            Console.WriteLine(" ");
        }
        Console.WriteLine();
    }
    static void Main()
    {
        object[] a = {1, "Delhi", 93.1};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

6. The name and the formal parameter list of a method define the signature of the method. Examine.
7. How is a value type parameter different from a reference type parameter?
8. When an instance method declaration includes an abstract modifier, the method is said to be an abstract method. Discuss this statement with an example.

9. WAP to calculate the factorial of a number using recursion. Explain.
10. WAP to calculate the sum of array when array is passed as a parameter. Explain.

Notes

Answer: Self Assessment

- | | |
|--------------------|----------------|
| 1. Object Oriented | 2. new |
| 3. error | 4. direct base |
| 5. initialize | 6. inherited |
| 7. variable | 8. public |
| 9. struct | 10. output |
| 11. Reference | 12. expression |
| 13. external | 14. private |
| 15. identifier | |

6.10 Further Readings



Books

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

<http://msdn.microsoft.com/en-us/library/x9afc042.aspx>

http://www.homeandlearn.co.uk/csharp/csharp_s10p1.html

Notes

Unit 7: Polymorphism and Inheritance

CONTENTS	
Objectives	
Introduction	
7.1 Inheritance	
7.2 Polymorphism	
7.3 Operator Overloading	
7.4 Summary	
7.5 Keywords	
7.6 Review Questions	
7.7 Further Readings	

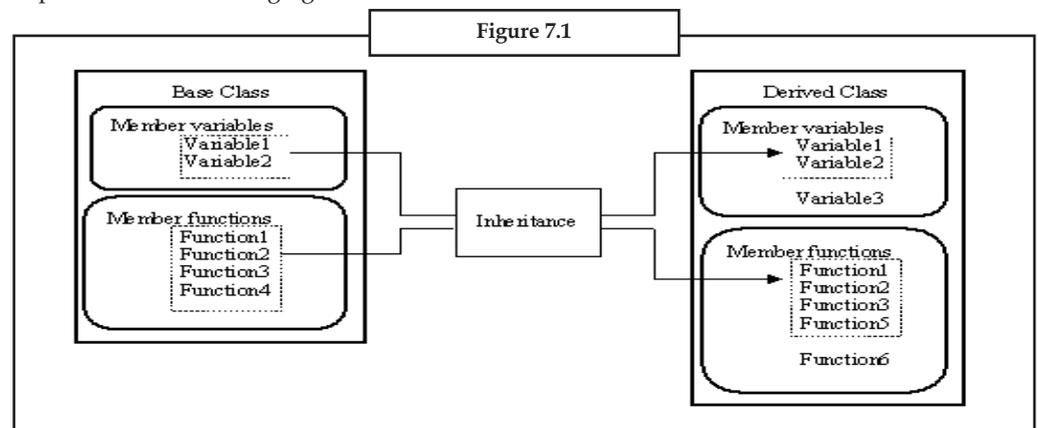
Objectives

After studying this unit, you will be able to:

- Describe Inheritance
- Demonstrate Polymorphism
- Recognize Operator Overloading

Introduction

Object oriented programming allows reusability of the codes written in a class. This is achieved through inheritance. A new class can be constructed using an existing class in such a way that the new class retains some or all the members of the existing class and may add one or more members to it. This process through which the members of an existing class are obtained by a new class is called inheritance much in the same way as a son inherits his father’s property. The new class being created is called the derived class and the existing class whose members are used in the new class is called the base class. The relationship between the base and derived classes is depicted in the following figure.



7.1 Inheritance

The figure 7.1 illustrates that a derived class can inherit features (member variables and member functions) of the base and can have its own features (member variables and member functions).

Inheritance follows certain rules that determine which features are inherited and which are not. These rules are summarized below.

- The derived class does not inherit private members of the base class.
- Inheritance is transitive. If C is derived from B, and B is derived from A, then C inherits the members declared in B as well as the members declared in A.
- A derived class extends its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Constructors and destructors are not inherited, but all other members are, regardless of their declared accessibility. However, depending on their declared accessibility, inherited members may not be accessible in a derived class.
- A derived class can hide inherited members by declaring new members with the same name or signature.



Notes However that hiding an inherited member does not remove the member-it merely makes the member inaccessible in the derived class.

- An instance of a class contains a copy of all instance fields declared in the class and its base classes, and an implicit conversion exists from a derived class type to any of its base class types. Thus, a reference to a derived class instance can be treated as a reference to a base class instance.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the run-time type of the instance through which the function member is invoked.
- Classes support single inheritance, and the type object is the ultimate base class for all classes.

Inheritance is one of the key concepts of Object Oriented Programming. By using the concept of inheritance, it is possible to create a new class from an existing one and add new features to it. Thus inheritance provides a mechanism for class level re usability. Obviously C# supports inheritance. The syntax of inheritance is very simple and straightforward.



Did u know? **What is Implementation Inheritance?**

When a class (type) is derived from another class(type) such that it inherits all the members of the base type it is Implementation Inheritance.

```
class BaseClass
{
}

class DerivedClass : BaseClass
```

Notes

```
{  
}
```

The classes shown in earlier examples all implicitly derive from object.



Example:

```
class One  
{  
    public void FunctionOne()  
{  
  
Console.WriteLine("One.Function");  
}  
}
```

It shows a class One that implicitly derives from object.



Example:

```
class Two : One  
{  
    public void FunctionTwo()  
{  
Console.WriteLine("Two.FunctionTwo");  
}  
}  
  
class TestClass  
{  
    static void Main()  
{  
        Two b = new Two();  
        b.FunctionOne();           // Inherited from One  
        b.FunctionTwo();          // Introduced in Two  
        One a = b;                // Treat a Two as an One  
        a.FunctionOne();  
    }  
}
```

The code shows a class Two that derives from One. The class Two inherits One's FunctionOne method, and introduces a FunctionTwo method of its own.

The operator ':' is used to indicate that a class is inherited from another class. Remember that in C#, a derived class can't be more accessible than its base class. That means that it is not possible to declare a derived class as public, if it inherits from a private class.



Example: The following code will generate a compile time error.

```
class BaseClass
{
}

public class DerivedClass : BaseClass
{
}
```

In the above case the BaseClass class is private. We try to inherit a public class from a private class.

In the code listed below DerivedClass inherits public members of the BaseClass x,y and Method(). The objects of the DerivedClass can access these inherited members along with its own member z.

```
using System;

class BaseClass
{
    public int x = 100;
    public int y = 200;
    public void Method()
    {
        Console.WriteLine("BaseClass Method");
    }
}

class DerivedClass : BaseClass
{
    public int z = 300;
}

class TestClass
{
    public static void Main()
    {
        DerivedClass d1 = new DerivedClass();
        Console.WriteLine("{0},{1},{2}",d1.x,d1.y,d1.z);
        // displays 10,20,30
        d1.Method();
        // displays 'BaseClass Method'
    }
}
```

Notes

Methods, properties, and indexers can be virtual, which means that their implementation can be overridden in derived classes (see the following code).

```
using System;

class One
{
    public virtual void FunctionOne()
    {
        Console.WriteLine("One.FunctionOne");
    }
}

class Two: One
{
    public override void FunctionOne()
    {
        base.FunctionOne();
        Console.WriteLine("Two.FunctionOne");
    }
}

class TestClass
{
    static void Main()
    {
        Two b = new Two();
        b.FunctionOne();
        One a = b;
        a.FunctionOne();
    }
}
```

The code shows a class One with a virtual method FunctionOne, and a class Two that overrides FunctionOne. The overriding method in Two contains a call base.FunctionOne() which calls the overridden method in One.

The abstract modifier allows a class to indicate that it is incomplete and that is intended only as a base class. A class using abstract modifier is called an abstract class.

An abstract class can specify abstract members-members that a non-abstract derived class must implement.



Example:

```
using System;

abstract class One
{

```

```

        public abstract FunctionOne ();
    }
class Two: One
{
    public override FunctionOne ()
    {
        Console.WriteLine("Two.FunctionOne");
    }
}
class TestClass
{
    static void Main()
    {
        Two b = new Two();
        Two.FunctionOne();
        One a = b;
        a.FunctionOne();
    }
}

```

The code introduces an abstract method `FunctionOne` in the abstract class `One`. The non-abstract class `Two` provides an implementation for this method.



Task Analyze the benefits of implementing inheritance in object programming.

Hiding through Inheritance

It must be noted carefully that inheritance may lead to name hiding. It happens when classes or structs redeclare names that were inherited from base classes. Following rules govern name hiding during inheritance.

- If the name of a constant, field, property, event, or type introduced in a class or struct is the same as that in the base class, the base class member's name becomes hidden.
- A method in a class having same name as a method in the base class, hides the method of the base class provided their signatures are same.
- An indexer in a class or struct hides all base class indexers with the same signature.
- Name hiding does not apply on operators. Thus, an operator in a derived class never hides the operator in the base class.

A warning is sounded by the compiler if the name hiding takes place in a class as shown in the code below.

```
class One
```

Notes

```
{
    public void FunctionOne()
}
}
class Two: One
{
    public void FunctionOne()
}
// Warning, hiding an inherited name
}
```

This code will generate a warning message since the compiler assumes that you did not intend to hide the name and that it was probably by mistake.



Caution It is just a warning and not an error.

The warning caused by hiding an inherited name can be eliminated through use of the new modifier as shown in the code listed hereunder.

```
class One
{
    public void FunctionOne()
}
}
class Two: One
{
    new public void FunctionOne()
}
// No warning this time
}
```

The new modifier directs the compiler that the name hiding is indeed intended and is not just by mistake. A declaration of a new member hides an inherited member only within the scope of the new member. Consider the code listed below.

```
class One
{
    public static void FunctionOne()
}
}
class Two : One
{
    new private static void FunctionOne()
}
// Hides One.FunctionOne in Two only
}
class Three: Two
```

Notes

```

{
    static void FunctionTwo() { FunctionOne(); }
// Invokes One.FunctionOne
}

```

In the example above, the declaration of FunctionOne in Two hides the FunctionONE that was inherited from One, but since the new FunctionOne in Two has private access, its scope does not extend to Three.



Notes The call FunctionOne() in Three.FunctionTwo is valid and will invoke One.FunctionOne.

Inheritance and Constructors

Inheritance does not allow constructors and destructors of the base class to be inherited by the derived class. However when we create an object of the derived class, the derived class constructor implicitly call the base class default constructor. Consider the following code.

```

using System;
class BaseClass
{
    public BaseClass()
    {
        Console.WriteLine("Base class default constructor");
    }
}
class DerivedClass : BaseClass
{
}
class TestClass
{
    public static void Main()
    {
        DerivedClass d1 =new DerivedClass();
        // Displays 'BaseClass default constructor'
    }
}

```



Notes DerivedClass constructor can call only the default constructor of BaseClass explicitly.

Notes

But they can call any BaseClass constructor explicitly by using the keyword base as shown in the code listed below.

```
using System;
class BaseClass
{
    public Base()
    {
        Console.WriteLine("BaseClass constructor One");
    }
    public BaseClass(int x)
    {
        Console.WriteLine("Base constructor Two");
    }
}
class DerivedClass : BaseClass
{
    public DerivedClass() : BaseClass(10)
    // implicitly call the BaseClass(int x)
    {
        Console.WriteLine("DerivedClass constructor");
    }
}
class TestClass
{
    public static void Main()
    {
        DerivedClass d1 = new DerivedClass();
        // Displays 'BaseClass constructor Two followed by 'DerivedClass
        Constructor''
    }
}
```



Task Discuss how inheritance may lead to name hiding. What are the rules that govern name hiding during inheritance?

Self Assessment**Notes**

Fill in the blanks:

1. A class can hide inherited members by declaring new members with the same name or signature.
2. The new class being created is called the derived class and the existing class whose members are used in the new class is called the class.
3. A class using abstract modifier is called an.....
4. Inheritance provides a mechanism for class level.....
5. Inheritance does not allow constructors and destructors of the base class to be inherited by the class.
6. A declaration of a new member hides an member only within the scope of the new member.
7. A method in a class having same name as a method in the base class, hides the method of the base class provided their are same.

7.2 Polymorphism

Polymorphism is the term derived from two greek words poly meaning many and morph meaning forms. Thus, polymorphism is a property of an object to exhibit multiple form.

In object-oriented paradigm, polymorphism refers to the objects belonging to different classes respond to the same function call but in different forms. For example, View function takes an argument and displays it on the screen.

- View("Hello there");
- View(46542.93);
- View(1234);

Therefore when the first method is called it will display "Hello there" on the screen. When the second method is called by passing float argument it will display 46542.93 and on the execution of the third method it will display 1234, which is an integer value. Here we can say that the function View is polymorphic.

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

1. At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
2. Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way.

Notes



Example: Suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface.

You do not know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called Shape, and derived classes such as Rectangle, Circle, and Triangle. Give the Shape class a virtual method called Draw, and override it in each derived class to draw the particular shape that the class represents. Create a List<Shape> object and add a Circle, Triangle and Rectangle to it. To update the drawing surface, use a foreach loop to iterate through the list and call the Draw method on each Shape object in the list. Even though each object in the list has a declared type of Shape, it is the run-time type (the overridden version of the method in each derived class) that will be invoked.



Example:

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
```

```
}  
}  
class Rectangle : Shape  
{  
    public override void Draw()  
    {  
        // Code to draw a rectangle...  
        Console.WriteLine("Drawing a rectangle");  
        base.Draw();  
    }  
}  
class Triangle : Shape  
{  
    public override void Draw()  
    {  
        // Code to draw a triangle...  
        Console.WriteLine("Drawing a triangle");  
        base.Draw();  
    }  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        // Polymorphism at work #1: a Rectangle, Triangle and Circle  
        // can all be used wherever a Shape is expected. No cast is  
        // required because an implicit conversion exists from a derived  
        // class to its base class.  
        System.Collections.Generic.List<Shape> shapes = new System.Collections.  
Generic.List<Shape>();  
        shapes.Add(new Rectangle());  
        shapes.Add(new Triangle());  
        shapes.Add(new Circle());  
  
        // Polymorphism at work #2: the virtual method Draw is  
        // invoked on each of the derived classes, not the base class.  
        foreach (Shape s in shapes)  
        {
```

Notes

```
s.Draw();  
}  
  
// Keep the console open in debug mode.  
Console.WriteLine("Press any key to exit.");  
Console.ReadKey();  
}  
  
}  
/*
```

Output:

```
Drawing a rectangle  
Performing base class drawing tasks  
Drawing a triangle  
Performing base class drawing tasks  
Drawing a circle  
Performing base class drawing tasks  
*/
```

In C#, every type is polymorphic because all types, including user-defined types, inherit from Object.

Self Assessment

Fill in the blanks:

8. methods enable you to work with groups of related objects in a uniform way.
9. Polymorphism refers to the objects belonging to different classes respond to the same but in different forms.
10. A derived class can override a base class member only if the base class member is declared as.....
11. Fields cannot be virtual; only methods, properties, events and can be virtual.

7.3 Operator Overloading

Normally an operator has a predefined meaning. The number of operands, their types and the computed result are all predefined for each operator. However, additional meanings can also be attached to the existing functions of an operator. Assigning additional meaning to an operator is called operator overloading.

In C# polymorphism is achieved by operator overloading. User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered.

A class or struct member declaration cannot introduce a member by the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. A class or struct furthermore permits the declaration of overloaded constructors and operators. For instance, a class, struct, or interface may contain multiple method declarations with the same name, provided these method declarations differ in their signature.

Not all operators are allowed to be overloaded. The overloadable unary operators are:

+ - ! ~ ++ -- true false

The overloadable binary operators are:

+ - * / % & | ^ << >> == != > < >= <=

All the rest of the operators cannot be overloaded. In particular, it is not possible to overload member access, method invocation, or the =, &&, |, ?, new, typeof, sizeof, and is operators. When a binary operator is overloaded, the corresponding assignment operator is also implicitly overloaded. For example, an overload of operator * is also an overload of operator * =.

Constructors are often overloaded. Overloading of constructors permits a class or struct to declare multiple constructors, provided the signatures of the constructors are all unique. Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided the signatures of the indexers are all unique.



Notes Operator itself (=) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, op denotes any overloadable unary operator. In the second entry, op denotes the unary ++ and -- operators. In the third entry, op denotes any overloadable binary operator.

Operator Notation	Functional Notation
op x	operator op(x)
X op	operator op(x)
X op y	operator op(x,y)

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator.



Example: The * operator is always a binary operator, always has the precedence level, and is always left associative.

Overriding Methods

The signature of an inherited method can be modified and the method can be re-implemented in a derived class. The process of re-defining a method is referred to as method-overriding. The override keyword is employed to override a method as shown below.

```
public override void InheritedFunction()
{ ... }
```

Notes

The method overrides an inherited virtual method with the same signature when an instance method declaration includes an override modifier. There is a difference between declaring a function virtual and overriding a function. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of the method. Therefore, an override method declaration will report an error if any one of the new, static, virtual, or abstract modifiers is included in the declaration.

Which base function is being overridden in the derived class is determined by examining each base class of the derived class, starting with the direct base class and continuing with each successive direct base class, until an accessible method with the same signature as the overridden method is located. Only the public, protected, protected internal and internal methods declared in the same project as the derived class are considered for the purpose of locating the overridden base method.

Following are some of the rules that govern method overriding. Violating any of them will cause the compiler report an error.

- An overridden base method could not be located.
- The overridden base method must be virtual, abstract, or override method itself.
- The overridden base method cannot be static.
- The override declaration and the overridden base method must have the same declared accessibility.



Example:

```
class One
{
    int x;
    public virtual void FunctionOne()
    {
        Console.WriteLine("x = {0}", x);
    }
}

class Two: One
{
    int y;
    public override void FunctionOne()
    {
        base.FunctionOne();
        Console.WriteLine("y = {0}", y);
    }
}
```

Here, the `base.FunctionOne()` invocation in `Two` invokes the `FunctionOne` method declared in `One`. A base-access disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in `Two` been written `((One)this).FunctionOne()`, it would recursively invoke the `FunctionOne` method declared in `Two`, not the one declared in `One`.

If a method declaration does not specify `override`, the method with the same signature as an inherited method will become hidden. Consider the following code.

```
class One
{
    public virtual void FunctionOne()
    {}
}
class Two: One
{
    public virtual void FunctionOne()
    {}           // Warning, hiding inherited FunctionOne()
}
```

Here, the `FunctionOne` method in `Two` does not include an `override` modifier and therefore does not override the `FunctionOne` method in `One`. Rather, the `FunctionOne` method in `Two` hides the method in `One`, and a warning is reported because the declaration does not include a new modifier.



Example:

```
class One
{
    public virtual void FunctionOne()
    {}
}
class Two: One
{
    new private void FunctionOne()
    {}           // Hides One.FunctionOne within Two
}
class Three: Two
{
    public override void FunctionOne()
    {}         // No error. overrides One.FunctionOne
}
```

Here, the `FunctionOne` method in `Two` hides the virtual `FunctionOne` method inherited from `One`. Since the new `FunctionOne` in `Two` has `private` access, its scope only includes the class body of `Two` and does not extend to `Three`. The declaration of `FunctionOne` in `Three` is therefore permitted to override the `FunctionOne` inherited from `One`.



Task Analyze the meaning of method-overriding. What are the rules that govern method-overriding?

Notes

Self Assessment

Fill in the blanks:

- 12. The process of re-defining a method is referred to as.....
- 13. In C# polymorphism is achieved by.....
- 14. A class or struct furthermore permits the declaration of overloaded constructors and.....
- 15. The method overrides an inherited virtual method with the same signature when an instance method declaration includes an.....



Caselet

IT helps align messages books2byte

Common language and speed are the two ways in which new digital media is dramatically shaping the practice of integrated communications, says a new book from Sage (www.sagepublications.com).

“Digitisation has created a shared language that allows employees, without technical skills, to exchange and share information (numbers, text and images) across different functions and geographical locations,” write Lars Thøger Christensen, Mette Morsing, and George Cheney in *Corporate Communications: Convention, Complexity, and Critique*.

As example of integrating language, the authors speak of how XML (extensible markup language) is capable of describing many different kinds of data and, consequently, to facilitate the sharing of data across different systems, particularly systems connected through the Net.

More important, one may say, is the other effect of digital media on communication, viz. speed, because IT (information technology) is making it faster and easier to compare messages across different media and different audiences. “The new information technologies allow organisations to identify inconsistencies in their messages and symbols and, accordingly, support their insistence on coherence in everything the organisation says and does,” the authors observe. Contemporary IT has thus “accentuated the call for integration within and across the organisation’s formal boundaries.”

For starters, a simple definition of ‘integrated communications,’ according to the book, is “the practice of aligning symbols, messages, procedures and behaviours in order for an organisation to communicate with clarity, consistency and continuity within and across formal organisational boundaries.”

A read that can add value to your messages.

The Connected President

In the first few weeks after Dr A.P.J. Abdul Kalam became President, he received an e-mail from a young girl in Agra. “It said, ‘Uncle, in our area, there is only one park and in that park, there is only one see-saw. And that has been out of order for the last ten days. Nobody bothers.’ And this e-mail came up for discussion in the morning meeting,” narrates P.M. Nair in *The Kalam Effect: My Years with the President* (www.harpercollins.com).

“‘So, what do we do?’ Kalam asked me in his typical style. I said, ‘Sir, I shall speak to the Collector.’ I did and the job was done. And Kalam gets an e-mail from the girl, ‘Uncle, thank you. The see-saw is working now.’ She added, ‘But uncle, when can I meet you, you are so good.’”

Contd...

Nair, who served as the President's secretary all through the five years of APJ's tenure, reminisces how Dr Kalam's keenness that all the letters and e-mails be attended to and answered cast 'a very heavy burden' on the entire Rashtrapati Bhavan establishment.

"Kalam's permanent refrain in all his speeches everywhere that anyone could send messages to him on his e-mail ID and that he would get a reply within 24/48 hours made the situation absolutely uncontrollable. His mailbox swelled out of proportion, bursting at the seams, and stretching us to the limit too in the bargain."

Inevitable, that was, because Rashtrapati Bhavan had become the 'People's Bhavan,' notes Nair, towards the conclusion. The five years saw 'a veritable electronic revolution in Rashtrapati Bhavan,' he adds. "Accessing the first citizen through e-mails that ran into a few hundreds almost every day became a way of life with people all over the country." The author also makes a mention of the state-of-the-art multimedia studio commissioned in the bhavan from which the President could address 'many a foreign audience in far-off countries.'

Interesting account.

Recipe Format

Herb Schildt is a well-known author of books on Java, C, C++ and C#, with over 3.5 million copies sold worldwide. "Because of the rapid revision cycles of those languages, I spend nearly all of my available time updating my books to cover the latest versions of those languages," confesses Schildt. However, a new book from TMH (www.tatamcgrawhill.com), was one of his most 'enjoyable projects': *Java Programming Cookbook*.

Based on the format of a traditional food cookbook, the author "distils the essence of many general-purpose techniques into a set of step-by-step recipes. Each recipe describes a set of key ingredients, such as classes, interfaces, and methods."

Good programming books should have two elements, insists Schildt: solid theory and practical application. "In the recipes, the step-by-step instructions and discussions supply the theory. To put that theory into practice, each recipe includes a complete code example... The examples eliminate the 'guesswork' and save you time."

He clarifies, however, that code examples in the book are not optimised for performance. "They are optimised for clarity and ease of understanding. Their purpose is to clearly illustrate the steps of recipe. In many cases you will have little trouble writing tighter, more efficient code."

An appetising addition to the avid programmers' shelf.

Musical Matchmaker

In an era when our celebrity-obsessed society hangs on every word and deed – both good and bad – of a handful of hot properties, it's easy to forget that hundreds of thousands of talented artists go almost unnoticed, rues Jim Champy in *Outsmart! How to do what your competitors can't* (www.pearsoned.co.in).

One of the cases discussed in the book is about Panos Panay, who saw a \$15-billion market in the untapped opportunity for musicians, and founded Sonicbids, to give every band an international stage, using technology. "The wedding band business, by itself, totals \$2.5 billion, plus there's another \$11 billion in bookings at small bars, clubs, coffeehouses, festivals, colleges, and private parties." Most talent agents collect a fee equivalent to just 10 per cent of their client's earnings, so they see it as a waste of time and energy to book, say, a wedding band that gets only \$800 for a night's work, explains Champy. Here's

Contd...

Notes

where Sonicbids leverages technology. "Some 10,000 promoters use the site to connect with Sonicbids' 1,20,000 musician members, a quarter of whom are from abroad.

The site facilitates connections via a community forum and an advanced search tool that enables members to narrow their quest by location, date, and genre."

Acting as musical matchmaker, Sonicbids works on the other side of the aisle too by prodding the musicians 'to show their best face to suitors' through an EPK (electronic press kit) that includes 'an MP3 sample of their music, photos, biographical material, and press clippings.' Promoters pay the musicians directly, and Sonicbids charges musicians, not commissions, but a membership fee of \$50 to \$100 a year.

An intangible that Panay offers both the promoters and the musicians is a proactive customer-service operation, finds Champy. "One of the problems you have in an online business is that you don't have a physical presence, like a restaurant where people can walk in and smell and touch. You don't have that kind of legitimacy by virtue of where you exist. We try to make up for that by the language we use, the way we communicate," reads a quote of Panay in the book.

The people who answer the phones are young and sympathetic to customers' needs, particularly those of struggling young musicians, writes Champy. "Aware of their insecurity, Panay insists that the musicians be treated respectfully. There is a standing rule, for example, that e-mails must be answered the same day they are received."

7.4 Summary

- Inheritance is one of the primary concepts of any object oriented programming.
- A class inherits the members of its direct base class.
- A derived class can hide inherited members by declaring new members with the same name or signature.
- The constructors and destructors are not inherited to a derived class from a base class.
- A virtual method in C# specifies an implementation of a method that can be polymorphically overridden derived method.
- Overloading is the technique to implement the polymorphism.
- It is possible to omit the keywords virtual from the derived class method or it is possible to declare the derived class method as new.
- Derived class constructor can call only the default constructor of base class explicitly.

7.5 Keywords

Base Class: The class whose features are used in the creation of a new class is called as Base Class.

Derived Class: the new class created through inheritance is called as derived class.

Inheritance: It is a process through which a new class can be created using the features of some other existing classes.

Operator Overloading: To provide multiple functions with the same operators name is called operator overloading.

Polymorphism: It refers to the objects belonging to different classes respond to the same function call but in different forms.

Notes

7.6 Review Questions

1. Explain various forms of inheritances with suitable examples.
2. Substantiate what are sealed methods? Where are they used? Explain with examples.
3. Illustrate multiple Inheritance with the help of a suitable examples.
4. Make the distinctions between multiple Inheritance and multilevel Inheritance. Explain with figures and example.
5. Write a few classes, demonstrating polymorphism.
6. Write a program to overload + (Plus) operator to add two complex numbers.
7. Write a program to define a class Automobile which contains number of wheels, engine number and body type. Now inherit a class Santro form automobile class and add the extra features in it.
8. Write a program in C# to demonstrate the concept of method overriding.
9. What do you mean by Inheritance? What are the rules that determine which features are inherited and which are not?
10. How Constructor behave in context with inheritance. Explain with the help of suitable example.

Answers: Self Assessment

- | | |
|--------------------------|-------------------------|
| 1. derived | 2. base |
| 3. abstract class | 4. re usability |
| 5. derived | 6. inherited |
| 7. signatures | 8. Virtual |
| 9. function call | 10. virtual or abstract |
| 11. indexers | 12. method-overriding |
| 13. operator overloading | 14. operators |
| 15. override modifier | |

7.7 Further Readings



Books

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde, Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

Notes

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

<http://msdn.microsoft.com/en-us/library/ms173152.aspx>

<http://www.exforsys.com/tutorials/csharp/inheritance-in-csharp.html>

Unit 8: Interfaces

Notes

CONTENTS

Objectives

Introduction

8.1 Interfaces

8.2 Base Interfaces

8.3 Interface Methods

8.4 Interface Properties

8.5 Interface Events

8.6 Interface Indexers

8.7 Interface Implementations

8.8 Interface Mapping

8.9 Interface Re-implementation

8.10 Namespaces

8.10.1 System Namespaces

8.10.2 Object Class

8.11 Summary

8.12 Keywords

8.13 Review Questions

8.14 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the Interface meaning and implementation
- Describe the Namespace meaning and its working
- Demonstrate the usage of system namespace
- Recognize Object Class

Introduction

This unit will introduce you to C# interfaces. An interface merely defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces. Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or struct that implement the interface.

8.1 Interfaces

At times programmers require to specify what all member must be there, if another programmer re-uses the class. The programmer needs only to specify the members and must leave their implementation to the programmer who would ultimately use the class. To allow such a contract between the programmers the concept of an interface has been introduced. An interface merely defines a contract between the producer and the consumer. The user may create classes and implement one or more interface. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Methods, properties, events, and indexers may be specified in an interface.



Notes The interface itself does not provide codes for the members that it defines. It only specifies the members that must be supplied by classes or interfaces that implement the interface.

Interface Declarations

In order to create an interface it must be declared. An interface is declared using interface keyword. An interface-declaration consists of an optional set of attributes, followed by an optional set of interface-modifiers, followed by the keyword interface and an identifier that names the interface, optionally followed by an optional interface-base specification, followed by a interface-body, optionally followed by a semicolon. For instance, see the following example.

```
Interface InterfaceOne
{
};
```

In the code listed above an interface whose name is InterfaceOne is declared. An interface-declaration may optionally include a sequence of following interface modifiers.

- new
- public
- protected
- internal
- private



Notes It is an error for the same modifier to appear multiple times in an interface declaration. The new modifier is only permitted on nested interfaces. It specifies that the interface hides an inherited member by the same name.

The public, protected, internal, and private modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers may be permitted.

Self Assessment

Notes

Fill in the blanks:

1. Methods, properties, events, and indexers may be specified in an.....
2. The public, protected, internal, and private modifiers control the of the interface.

8.2 Base Interfaces

An interface can also inherit from zero or more interfaces, which are called the explicit base interfaces of the interface. When an interface has more than zero explicit base interfaces then in the declaration of the interface, the interface identifier is followed by a colon and a comma-separated list of base interface identifiers.

```
Interface InterfaceTwo : InterfaceOne
{
};
```

The explicit base interfaces of an interface must be at least as accessible as the interface itself. For example, it is an error to specify a private or internal interface in the interface-base of a public interface.

It is an error for an interface to directly or indirectly inherit from itself.

The base interfaces of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on.



Example:

```
interface InterfaceOne
{
    void FunctionOne();
}

interface InterfaceTwo : InterfaceOne
{
    void FunctionTwo(string text);
}

interface InterfaceThree : InterfaceOne
{
    void FunctionThree(string[] items);
}

interface InterfaceFour : InterfaceTwo, InterfaceThree
{
};
```

Here the base interfaces of InterfaceFour are InterfaceOne, InterfaceTwo, and InterfaceThree. An interface inherits all members of its base interfaces.

The interface-body of an interface defines the members of the interface. The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

Notes

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, constructors, destructors, static constructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is an error for interface member declarations to include any modifiers. In particular, interface members cannot be declared with the `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static` modifiers.



Example:

```
public delegate void StringListEvent (IStringList sender);
public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

The code declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

An interface-declaration creates a new declaration space, and the interface-member-declarations immediately contained by the interface-declaration introduce new members into this declaration space. The following rules apply to interface-member-declarations:

- The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature of a method must differ from the signatures of all other methods declared in the same interface.
- The name of a property or event must differ from the names of all other members declared in the same interface.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to hide the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a new modifier to indicate that the derived member is intended to hide the base member.



Task An interface declaration may declare zero or more members. Explain with a program.

If a new modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the new modifier.

Self Assessment

Notes

Fill in the blanks:

3. The interfaces of an interface are the explicit base interfaces and their base interfaces.
4. The interfaces of an interface must be at least as accessible as the interface itself.
5. The signature of an indexer must differ from the signatures of all other indexers declared in the same.....
6. The members of an interface are specifically not part of the declaration space of the interface.

8.3 Interface Methods

Interface methods are declared using interface-method-declarations which can have attributes, return-type and formal parameter list. The attributes, return-type, and formal-parameter-list of an interface method declaration have the same meaning as those of a method declaration in a class. An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

8.4 Interface Properties

Interface properties are declared using interface-property-declarations. The attributes, type, and identifier of an interface property declaration have the same meaning as those of a property declaration in a class.

The accessors of an interface property declaration correspond to the accessors of a class property declaration, except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.



Did u know? What are Properties in C#?

Properties are a new language feature introduced with C#. They provide the opportunity to protect a field in a class by reading and writing to it through the property.

C# properties enable this type of protection while also letting you access the property just like it was a field. Let's take a look at how to provide field encapsulation by traditional methods.

```
using System;

public class PropertyHolder
{
    private int someProperty = 0;
    public int getSomeProperty()
    {
        return someProperty;
    }
    public void setSomeProperty(int propValue)
    {
```

Notes

```
        someProperty = propValue;
    }
}

public class PropertyTester
{
    public static int Main(string[] args)
    {
        PropertyHolder propHold = new PropertyHolder();
        propHold.setSomeProperty(5);

        Console.WriteLine("Property Value: {0}", propHold.
getSomeProperty());
        return 0;
    }
}
```

The propertyHolder class has the field we're interested in accessing. It has two methods, getSomeProperty and setSomeProperty. The getSomeProperty method returns the value of the someProperty field. The setSomeProperty method sets the value of the someProperty field.

The PropertyTester class uses the methods of the PropertyHolder class to get the value of the someProperty field in the PropertyHolder class. The Main method instantiates a new PropertyHolder object, propHold. Next it sets the someMethod of propHold to the value 5 by using the setSomeProperty method. Then the program prints out the property value with a Console.WriteLine method call. The argument used to obtain the value of the property is a call to the getSomeProperty method of the propHold object. It prints out "Property Value: 5" to the console.

This method of accessing information in a field has been good because it supports the object-oriented concept of encapsulation. If the implementation of someProperty changed from an int type to a byte type, this would still work.

The same thing can be accomplished much smoother with properties.

```
using System;

public class PropertyHolder
{
    private int someProperty = 0;

    public int SomeProperty
    {
        get
        {
            return someProperty;
        }
        set
    }
}
```

Notes

```

{
    someProperty = valued;
}
}
}
public class PropertyTester
{
    public static int Main(string[] args)
    {
        PropertyHolder propHold = new PropertyHolder();
        propHold.SomeProperty = 5;
        Console.WriteLine("Property Value: {0}", propHold.SomeProperty);
        return 0;
    }
}

```

The program listed above shows how to create and use a property. The PropertyHolder class has the "SomeProperty" property implementation. Notice that the first letter of the first word is capitalized. That's the only difference between the names of the property "SomeProperty" and the field "someProperty". The property has two accessors, get and set.



Caution The get accessor returns the value of the someProperty field.

The set accessor sets the value of the someProperty field with the contents of "valued".

The PropertyTester class uses the SomeProperty property in the PropertyHolder class. The first line of the Main method creates a PropertyHolder object named propHold. Next the value of the someProperty field of the propHold object is set to 5 by using the SomeProperty property. It's that simple – just assign the value to the property as if it were a field.

After that, the Console.WriteLine method prints the value of the someProperty field of the propHold object. It does this by using the SomeProperty property of the propHold object. Again, it's that simple – just use the property as if it were a field itself.

ReadOnly Property

Properties can be made read-only. This is accomplished by having only a get accessor in the property implementation. Since there is no set accessor provided for the property, its value cannot be written.

```

using System;

public class PropertyHolder
{
    private int someProperty = 0;
    public PropertyHolder(int propVal)
    {

```

Notes

```
someProperty = propVal;
}
public int SomeProperty
{
get
{
return someProperty;
}
}
}
public class PropertyTester
{
public static int Main(string[] args)
{
PropertyHolder propHold = new PropertyHolder(5);
Console.WriteLine("Property Value: {0}", propHold.SomeProperty);
return 0;
}
}
```

The PropertyHolder class has a SomeProperty property that only implements a get accessor. It leaves out the set accessor. This particular PropertyHolder class has a constructor which accepts an integer parameter.

The Main method of the PropertyTester class creates a new PropertyHolder object named propHold. The instantiation of the propHold object uses the constructor of the PropertyHolder that takes an int parameter. In this case, it's set to 5. This initializes the someProperty field of the propHold object to 5. Since the SomeProperty property of the PropertyHolder class is read-only, there is no other way to set the value of the someProperty field. If you inserted PropHold.SomeProperty = 7 into the listing, the program would not compile, because SomeProperty is read-only.



Notes When the SomeProperty property is used in the Console.WriteLine method, it works fine. This is because it's a read operation which only invokes the get accessor of the SomeProperty property.

Similar to read-only, write-only properties do not implement get accessor for that property. Consider the following program wherein a write-only property has been implemented.

```
using System;
public class PropertyHolder
{
private int someProperty = 0;
public int SomeProperty
```

Notes

```

{
set
{
someProperty = value;
Console.WriteLine("someProperty is equal to {0}", someProperty);
}
}
}

public class PropertyTester
{
public static int Main(string[] args)
{
PropertyHolder propHold = new PropertyHolder();
propHold.SomeProperty = 5;
return 0;
}
}

```

The above program shows how to create and use a write-only property. This time the get accessor is removed from the SomeProperty property of the PropertyHolder class. The set accessor has been added, with a bit more logic. It prints out the value of the someProperty field after it's been modified.

The Main method of the PropertyTester class instantiates the PropertyTester class with a default constructor. Then it uses the SomeProperty property of the propHold object to set the someProperty field of the propHold object to 5. This invokes the set accessor of the propHold object, which sets the value of it's someProperty field to 5 and then prints "someProperty is equal to 5" to the console.

8.5 Interface Events

Interfaces can also define events. Interface events are declared using interface-event-declarations. The attributes, type, and identifier of an interface event declaration have the same meaning as those of an event declaration in a class.

Self Assessment

Fill in the blanks:

7. Interface methods are declared using which can have attributes, return-type and formal parameter list.
8. The accessors of an interface property declaration correspond to the accessors of a class property declaration, except that the accessor body must always be a.....
9. Interface are declared using interface-event-declarations.

8.6 Interface Indexers

Indexers can also be members of an interface. Interface indexers are declared using interface-indexer-declarations. The attributes, type, and formal-parameter-list of an interface indexer declaration have the same meaning as those of an indexer declaration in a class.

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration, except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

Accessing Interface Members

Interface members are accessed through member access and indexer access expressions of the form `I.M` and `I[A]`, where `I` is an instance of an interface type, `M` is a method, property, or event of that interface type, and `A` is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup, method invocation, and indexer access rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be included in the program code to resolve the ambiguities.



Example:

```
interface InterfaceOne
{
    int Count { get; set; }
}

interface InterfaceTwo
{
    void Count(int i);
}

interface InterfaceThree : InterfaceOne, InterfaceTwo
{}

class ClassOne
{
    void Test(InterfaceTwo x)
    {
        x.Count(1);
    }
}

// Error, Count is ambiguous
x.Count = 1;

// Error, Count is ambiguous
((InterfaceTwo)x).Count = 1;

// Ok, invokes InterfaceTwo.Count.set
```

```

        ((InterfaceOne)x).Count(1);
// Ok, invokes InterfaceOne.Count
    }
}

```

Here, the first two statements cause compile-time errors because the member lookup of `Count` in `InterfaceTwo` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs—they merely consist of viewing the instance as a less derived type at compile-time.



Example:

```

interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}

interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);           // Error, both Add methods are applicable
        n.Add(1.0);        // Ok, only IDouble.Add is applicable
        ((IInteger)n).Add(1); // Ok, only IInteger.Add is a candidate
        ((IDouble)n).Add(1); // Ok, only IDouble.Add is a candidate
    }
}

```

Here, the invocation `n.Add(1)` is ambiguous because a method invocation requires all overloaded candidate methods to be declared in the same type. However, the invocation `n.Add(1.0)` is permitted because only `IDouble.Add` is applicable. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity.



Example:

```

interface InterfaceOne
{
    void FunctionOne(int i);
}

interface InterfaceTwo : InterfaceOne
{

```

Notes

```

        new void FunctionOne(int i);
    }
    interface InterfaceThree: InterfaceOne
    {
        void FunctionTwo();
    }
    interface InterfaceFour : InterfaceTwo, InterfaceThree
    {}
    class ClassOne
    {
        void Test(InterfaceFour d)
        {
            d.FunctionOne(1);
            // Invokes InterfaceTwo.FunctionOne

            ((InterfaceOne)d).FunctionOne(1);
            // Invokes InterfaceOne.FunctionOne

            ((InterfaceTwo)d).FunctionOne(1);
            // Invokes InterfaceTwo.FunctionOne

            ((InterfaceThree)d).FunctionOne(1);
            // Invokes InterfaceOne.FunctionOne
        }
    }

```

Here, the InterfaceOne.FunctionOne member is hidden by the InterfaceTwo.FunctionOne member. The invocation d.FunctionOne(1) thus selects InterfaceTwo.FunctionOne, even though InterfaceOne.FunctionOne appears to not be hidden in the access path that leads through InterfaceThree.



Task Analyze the differences between an interface and a class? Explain with an example.

8.7 Interface Implementations

Interfaces are codeless entities and hence cannot be instantiated as classes. However, interfaces may be implemented by classes and structs. To indicate that a class or struct implements an interface, the interface identifier is included in the base class list of the class or struct. Consider the following example.



Example:

```
interface InterfaceOne
{
    object ObjectOne();
}
interface InterfaceTwo
{
    int CompareTo(object other);
}
class ClassOne: InterfaceOne, InterfaceTwo
{
    public object ObjectOne() {...}
    public int CompareTo(object other) {...}
}
```

A class or struct that implements an interface also implicitly implements all of the interface's base interfaces. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list.

Self Assessment

Fill in the blanks:

10. Indexers can also be members of an
11. A class or struct that implements an interface also implicitly implements all of the interface's interfaces.

8.8 Interface Mapping

Note that a class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as interface mapping.

Interface mapping for a class or struct locates an implementation for each member of each interface specified in the base class list. The implementation of a particular interface member is determined by examining each class or struct starting with the current class and repeating for each successive base classes, until a match is located.

For purposes of interface mapping, a class member A matches an interface member B when:

- A and B are methods, and the name, type, and formal parameter lists of A and B are identical.
- A and B are properties, the name and type of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).

Notes

- A and B are events, and the name and type of A and B are identical.
- A and B are indexers, the type and formal parameter lists of A and B are identical, and A has the same accessors as B (A is permitted to have additional accessors if it is not an explicit interface member implementation).



Notes Notable implications of the interface mapping algorithm are:

Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.

Private, protected, and static members do not participate in interface mapping.



Example:

```
interface IinterfaceOne
{
    object Clone();
}
class ClassOne : InterfaceOne
{
    object InterfaceOne.Clone() {...}
    public object Clone() {...}
}
```

Here, the InterfaceOne.Clone member of ClassOne becomes the implementation of Clone in InterfaceOne because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member.



Example:

```
interface InterfaceOne
{
    void Paint();
}
interface InterfaceTwo
{
    void Paint();
}
class ClassOne : InterfaceOne, InterfaceTwo
{
```

```

    public void Paint() {...}
}

```

Here, the Paint methods of both the interfaces are mapped onto the Paint method in ClassOne. It is of course also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations.



Example:

```

interface InterfaceOne
{
    int P { get; }
}
interface InterfaceTwo: InterfaceOne
{
    new int P();
}

```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms:

```

class ClassOne : InterfaceTwo
{
    int InterfaceOne.P { get {...} }
    int InterfaceTwo.P() {...}
}
class ClassOne: InterfaceTwo
{
    public int P { get {...} }
    int InterfaceTwo.P() {...}
}
class ClassOne: InterfaceTwo
{
    int InterfaceOne.P { get {...} }
    public int P() {...}
}

```

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface.



Example:

```

interface IControl
{

```

Notes

```

        void Paint();
    }
    interface ITextBox: IControl
    {
        void SetText(string text);
    }
    interface IListBox: IControl
    {
        void SetItems(string[] items);
    }
    class ComboBox: IControl, ITextBox, IListBox
    {
        void IControl.Paint() {...}
        void ITextBox.SetText(string text) {...}
        void IListBox.SetItems(string[] items) {...}
    }

```

As is clear, it is not possible to have separate implementations for the IControl named in the base class list, the IControl inherited by ITextBox, and the IControl inherited by IListBox. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of ITextBox and IListBox share the same implementation of IControl, and ComboBox is simply considered to implement three interfaces, IControl, ITextBox, and IListBox.

The members of a base class participate in interface mapping.



Example:

```

interface Interface1
{
    void FunctionOne();
}
class Class1
{
    public void FunctionOne() {}
    public void FunctionTwo() {}
}
class Class2: Class1, Interface1
{
    new public void FunctionTwo() {}
}

```

Here, the method FunctionOne in Class1 is used in Class2's implementation of Interface1.

As it is compulsory for a class or a struct which implements an interface to implement all the methods and properties of the interface this is called interface mapping the signature and the arguments of all the members of the interface should map with that declared in the class.



Example:

```
interface Print
{
void setText(int number);
string getText();
}
public class Student: Print
{
    string name[ ]= new string[];
    public void setText( int e)
    {
        for( int a=0; a<e;a++)
        {
            System.Console.WriteLine("enter the name");
            Name[a]=System.Console.ReadLine();
        }
    }
public string getText(int f)
    {
        return name[f];
    }
    public static void Main()
    {
        System.Console.WriteLine("enter the number of students you want
to enter the name");
        int e=System.Console.ReadLine();
        Student s= new Student();
        s.setText(e);
        for(a=0;a<e;a++)
        System.Console.WriteLine("The name of student is:"+ s.getText(a));
        Teacher t=new Teacher()
        System.Console.WriteLine("Enter the number of subjects taken by the
teacher");
        int h=System.Console.ReadLine();
        t.setText(h);
    }
}
class Teacher: Print
```

Notes

```
{
    public void setText (int subject)
    {
        System.Console.WriteLine("This teacher takes "+subject+"
subjects");
    }
    public string getText(){}
}
```

The output would be:

```
enter the number of students you want to enter the name:
3
enter the name:
Anita
enter the name
Rekha
Enter the name:
Sulekha
The name of the student is Anita
The name of the student is Rekha
The name of the student is Sulekha
Enter the number of subjects taken by the teacher:
5
This teacher takes 5 subjects
```

In the above program we have used two different classes Student and Teacher, you will see that we have mapped both the methods of the interface Print in the classes as it implements the interface but the function performed by the methods are different in both the classes. It is not important to have the same definition or the function performed by the method in various classes, which implement the method. Also note that in the class Teacher the body of getText() method is not there because we are not going to do any function but we still have to map it and declare with empty body.

Interface Implementation Inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly re-implementing an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes.



Example:

```
interface IControl
{
    void Paint();
}
```

```

class Control: IControl
{
    public void Paint() {...}
}
class TextBox: Control
{
    new public void Paint() {...}
}

```

the `Paint` method in `TextBox` hides the `Paint` method in `Control`, but it does not alter the mapping of `Control.Paint` onto `IControl.Paint`, and calls to `Paint` through class instances and interface instances will have the following effects

```

Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes Control.Paint();

```

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the declarations above to

```

interface IControl
{
    void Paint();
}
class Control: IControl
{
    public virtual void Paint() {...}
}
class TextBox: Control
{
    public override void Paint() {...}
}

```

The following effects will now be observed

```

Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;

```

Notes

```
c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes TextBox.Paint();
```

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. It is however perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it.



Example:

```
interface IControl
{
    void Paint();
}
class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}
class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Here, classes derived from Control can specialize the implementation of IControl.Paint by overriding the PaintControl method.

Ambiguity in the methods of the interface.



Example:

```
public interface Home
{
    void address();
}
public interface Office
{
    void address(string name);
}
interface Employee: Home, Office {}

class company
```

Notes

```

{
    void employee(Employee e)
    {
        e.address("New Delhi");
    }
    e.address("Mumbai");
}

public static void Main()
{
    company c=new company();
    c.employee();
}
}

```

The output would be that it would give an error saying that the method is ambiguous because it implements interface Home and Office both of which have same method address. Therefore it won't know which method to use.

The improved program would be to explicitly specify which method to take from which interface.

```

public interface Home
{
    void address();
}

public interface Office
{
    void address(string name);
}

interface Employee: Home, Office {}

class company
{
    void employee(Employee e)
    {
        (( Home ) e ).address("New Delhi");
        (( Office ) e).address("Mumbai");
        System.Console.WriteLine("The home address is:" + ((Home) e).address());
        System.Console.WriteLine("The office address is:" + ((Office) e).address());
    }

    public static void Main()
    {
        company c=new company();
        c.employee();
    }
}

```

Notes

```
    }  
}
```

The output would be

```
The home address is:      New Delhi  
The office address is:   Mumbai
```

Now in this case we have specified the method to be taken from which interface.



Task Discuss with your tutor that how are interfaces implemented? Explain giving suitable examples.

Self Assessment

Fill in the blanks:

- 12. When a class implements interfaces that have the same base interface, there can be only one implementation of the base interface.

8.9 Interface Re-implementation

A class that inherits an interface implementation is permitted to re-implement the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface.



Example:

```
interface IControl  
{  
    void Paint();  
}  
class Control: IControl  
{  
    void IControl.Paint() {...}  
}  
class MyControl: Control, IControl  
{  
    public void Paint() {}  
}
```

The fact that Control maps IControl.Paint onto Control.IControl.Paint doesn't affect the re-implementation in MyControl, which maps IControl.Paint onto MyControl.Paint.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces.



Example:

```
interface IMethods
{
    void FunctionOne();
    void FunctionTwo();
    void FunctionThree();
    void FunctionFour();
}

class Base: IMethods
{
    void IMethods.FunctionOne() {}
    void IMethods.FunctionTwo() {}
    public void FunctionThree() {}
    public void FunctionFour() {}
}

class Derived: Base, IMethods
{
    public void FunctionOne() {}
    void IMethods.FunctionThree() {}
}
```

Here, the implementation of IMethods in Derived maps the interface methods onto Derived. FunctionOne, Base.IMethods.FunctionTwo, Derived.IMethods. FunctionThree, and Base.FunctionFour.

When a class implements an interface, it implicitly also implements all of the interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. Consider the following example.



Example:

```
interface IBase
{
    void FunctionOne();
}

interface IDerived: IBase
{
    void FunctionTwo();
}

class C: IDerived
{
    void IBase.FunctionOne() {...}
}
```

Notes

```
void IDerived.FunctionTwo() {...}
}
class D: C, IDerived
{
    public void FunctionOne() {...}
    public void FunctionTwo() {...}
}
```

Here, the re-implementation of IDerived also re-implements IBase, mapping IBase.FunctionOne onto D.FunctionOne.

Abstract Classes and Interfaces

Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods.



Example:

```
interface IMethods
{
    void FunctionOne();
    void FunctionTwo();
}
abstract class C: IMethods
{
    public abstract void FunctionOne();
    public abstract void FunctionTwo();
}
```

Here, the implementation of IMethods maps FunctionOne and FunctionTwo onto abstract methods, which must be overridden in non-abstract classes that derive from C.



Notes Explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods.



Example:

```
interface IMethods
{
    void FunctionOne();
    void FunctionTwo();
}
abstract class C: IMethods
```

```

{
    void IMethods.FunctionOne() { FF(); }
    void IMethods.FunctionTwo() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}

```

Here, non-abstract classes that derive from C would be required to override FF and GG, thus providing the actual implementation of IMethods.



Task Compare and contrast between an interface and an abstract class.

Self Assessment

Fill in the blanks:

- A of an interface follows exactly the same interface mapping rules as an initial implementation of an interface.

8.10 Namespaces

A programmer need not reinvent the wheel by creating all the basic classes in .NET platform. Most of the basic classes are provided by Microsoft as library classes, which form base classes for most of the user classes. All these classes can be (re)used in a C# program. These classes provide the kind of basic functionality such as creating windows and handling files.

.NET base classes are analogous to Win32 Application Program Interface. The .NET base classes are much simpler to use. User classes can derive from these base classes to provide specialist classes. Moreover, base classes from any .NET-aware language can be used in any other .NET-aware language. In particular, these base classes provide rich functionalities for the following tasks, among others:

- String handling
- Arrays, lists, maps etc.
- Accessing files and the file system
- Accessing the registry
- Security
- Windowing
- Windows messages
- Connecting to other computers and to the Internet
- Drawing
- Directory Access
- Database Access

Note that the base classes are not only available to C# programs but they can equally well be accessed from VB, C++ or any other .NET-aware language.

Notes

There are thousands of base classes included in .NET framework. These classes are organized in much the same way a file system is organized into folders. These units in which C# classes are organized are called namespaces. By putting the classes into namespaces we can group related classes together, and also avoid the risk of name duplication.

Programmers will eventually create their own classes during programming. They can also organize their classes into namespaces. If your company happens to define a class that has the same name as the class written by another organization, and there were no namespaces, there would be no way for a compiler to figure out which class a program is actually referring to. With namespaces, there isn't a problem because the two classes will be placed in different namespaces, which compares with, say, the Windows files system where files with the same name can be contained in different folders. It is also possible for namespaces to contain other namespaces, just as folders on your file system can contain other folders as well as files.

When Visual Studio generates your projects, it automatically puts your classes in a namespace. Say for example you use the developer environment to create a C# Windows Application project called ProjectOne. If you do this and look at the code generated for you, you'll see something like this.

```
namespace ProjectOne
{
    using System;
        using System.Drawing;
        using System.Collections;
        using System.ComponentModel;
        using System.Windows.Forms;
        using System.Data;
```

The initial namespace keyword indicates that everything following the opening curly brace is part of a namespace called ProjectOne. Later on in the file, a class called FormOne is declared as:

```
public class FormOne : System.Windows.Forms.Form
```

Since this class has been declared inside the namespace, its 'full' name is not FormOne but ProjectOne.FormOne, and any other code outside this namespace must refer to it as such. This name is correctly known as its fully qualified name.



Notes The class called Form from which FormOne is derived is itself defined inside the namespace WinForms, which in turn is defined inside the namespace System.

The purpose of the 'using' command is to provide a way to avoid having to write fully-qualified names everywhere, since the fully-qualified names can get quite long making the code hard to read. For example, consider the line

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

This line declares that I may later in the code use classes from the System.Windows namespace, without indicating the fully-qualified name - and the same applied for every other namespace mentioned in a using command. For example consider the line of code, also generated by the developer environment

```
public class FormOne : System.Windows.Forms.Form
```

Because of the earlier 'using' command, we could equally well write this as

```
public class FormOne : Form
```

In this case the compiler will locate the class by searching all the namespaces that have been mentioned in a 'using' command. If it finds a class named 'Form' in more than one of these namespaces, it will generate a compilation error. To avoid compilation error of this kind you must use the fully qualified name in your source code.

The only purpose of the 'using' command in this context is to save you typing and make your code simpler. It doesn't cause any other code or libraries to be added to your project. If your code uses base classes or any other classes that are defined in libraries or assemblies, you need to ensure separately that the compiler knows which assemblies to look in for the classes. If you are compiling from the Visual Studio developer environment, this is done through the project references, which are listed in the Solution Explorer window. When you initiate a project, some references are inserted automatically.



Did u know? **On which situation does the references depends?**

The references, however, depend on the type of project, and you can add others using the Project | Add Reference menu. An assembly is a container of many namespaces and vice versa.

In case you are compiling from the command line then the assembly mscorlib.dll, which contains some of the most important base classes, is referenced implicitly. You will need to indicate any other assemblies to be referenced with the command line parameter /r, supplying the full file system path to the assembly. For example for the above project, the appropriate command for command line compilation is:

```
C:\>csc ReadFile.cs
/r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.Drawing.dll
/r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.Windows.dll
/r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.Data.dll
/r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.Diagnostics.dll
/r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.dll
/r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\Microsoft.Win32.Interop.dll
```

Adding Comments

Comments in a program serve documentation purposes. In C# comments can be added to a program in two ways.

1. **One-line comment:** A line that begins with double slash (//) is considered to be a comment.
2. **Multiple-line comment:** All the lines written between (/*) and (*/) pair are considered to be comments.

Main Returning a Value (Working)

In the HelloWorld example program Main method was declared as:

```
public static void Main()
```

The void keyword in the declaration indicates that the Main method (or function) returns no value to the caller (Operating System in this case). However, a Main method may return a value of indicated type.

Notes



Notes The return value may, for instance, indicate whether the program terminated successfully.

Here is a class having a Main method that returns an integer type value.

```
using System;

public class HelloWorld
{
    public static int Main()
    {
        Console.WriteLine("Hello World! ");
        Return 1;
    }
}
```

Using Aliases for Namespace Classes

If you have a very long namespace and you want to use it several times in your code, then you can substitute a short word for the long namespace name which you can refer to in the code as often as you want. Such an alternative name is called an alias. The advantages of using aliases are that:

1. The code becomes easier to read and maintain and
2. It saves you typing out very long strings repeatedly

The syntax for declaring an alias is:

```
using Alias = Abcd.Pqrs.Xyzw.MySimpleProgram.Examples;
```

The following code snippet illustrates how you can use an alias.

```
namespace Abcd.Pqrs.Xyzw.MySimpleProgram.Examples
{
    using System;
    using MyEx = Abcd.Pqrs.Xyzw.MySimpleProgram.Examples;
    namespace BaseClass
    {
        public class clsExample1
        {
            public static void Main()
            {
                abc = new MyEx.abc();
                Console.WriteLine(abc.Add(3, 4));
            }
        }
    }
}
```

}

Here, the alias MyEx refers to this namespace Abcd.Pqrs.Xyz.MySimpleProgram.Examples.

Passing String Objects to WriteLine Method

WriteLine method has been sufficiently overloaded to take care of almost any type of argument passed to it. Even string type objects can be passed to it in a similar manner. Here is a code snippet that does just the same.

```
using System;

public class HelloWorld
{
    string nam = "Vikas"
    public static int Main()
    {
        Console.WriteLine(nam);
        Return 1;
    }
}
```

The output is clearly Vikas.

Command Line Method

Sometimes a program needs to be executed with a set of parameters at the command prompt. This is allowed in C# provided the Main method takes the following form.

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

When the Main method takes this form, parameters specified in the command line are stored in a string type array - args. Inside the program, these parameters are referred to as args[0], args[1] and so on.



Example:

```
using System;

class HelloThere
{
    public static void Main(string[] args)
    {
        for (int i = 0; i < args.Length; i++)
            Console.WriteLine(Arg{0}", args[i]);
    }
}
```

The program when run in the following manner having some command line parameters:

```
HelloThere Ashok Vinod Mukesh
```

Notes

Produces the following output:

```
Ashok  
Vinod  
Mukesh
```

Main with a Class

The Main method may be used as stand alone method doing all the program logic. However, in a true object-oriented program it always belongs to some object of a class. Here is an example of the Main method without a class.

```
using System;  
  
void Main()  
{  
    Console.WriteLine("Main without a class");  
}
```

This is a valid C# program and it demonstrates that it is upward compatible with C.

8.10.1 System Namespaces

The System namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.

Other classes provide services supporting data type conversion, method parameter manipulation, mathematics, remote and local program invocation, application environment management, and supervision of managed and unmanaged applications.

8.10.2 Object Class

The object class is the ultimate base class of every type. Because of its lofty position and significant influence on the code you write, you should have a good understanding of the object class and its members. You may also be interested in knowing why object exists in the first place. In this article, we look at these things with the goal of making you more aware of the rationale and use of the object class in your code.

Why an Object Class?

As the ultimate base class that all other types directly or indirectly derive from, the object class allows you to build generic routines, provides Type System Unification, and enables working with groups of types as collections. Common practice for implementing generic routines has been to write a method that accepts arguments of type object and allow users to pass in any type they want. This approach is difficult to maintain because of the lack of type safety and inefficiency for value types. I believe the practice of using the object class for generic routines will be replaced in many future scenarios by the use of Generics in C# v2.0.

Reference types inherit the object class either directly or through other reference types. Value types inherit implicitly from the object class through System.ValueType. In C#, the object class is a language specific alias for the .NET Base Class Library (BCL) type with the fully qualified name of System.Object.

Notes

The ability of both reference types and value types to be treated as objects supports Type System Unification. In many languages, built-in types such as int and float have no object-oriented properties and must be explicitly wrapped in objects to simulate object-oriented behavior. However, C# eliminates the requirement to wrap built-in types through the concept of value types that inherit from System.ValueType, which inherits from System.Object. This allows C# built-in types to be worked with in a manner similar to reference types. From an object-oriented perspective, under Type System Unification both reference types and value types are objects.

Of course you don't get Type System Unification for free. When assigning a value type to an object, behind the scenes, the system performs an extra operation referred to as Boxing. After a boxing operation has completed, a value type is considered boxed. Similarly, when copying a boxed value type back to a normal value type, behind the scenes, the system performs another operation referred to as Unboxing. While any value type can be boxed, it is not logical to assume that any reference type can be unboxed. Only objects that have been previously boxed can be unboxed.



Notes When a value type is boxed, a new object for that type is created in memory and the value of the value type is copied into that object. During unboxing the value in the object is copied into the value of a value type.

Self Assessment

Fill in the blanks:

14. The keyword in the declaration indicates that the Main method (or function) returns no value to the caller.
15. The ability of both types and value types to be treated as objects supports Type System Unification.



Caselet

Tata Infotech course on .Net Platform

TATA Infotech Ltd on Wednesday launched its C Sharp (C#) course based on the .NET technology platform from Microsoft for software professionals.

According to Mr Rahul Thapan, Head, Education Service Division, this course would introduce concepts on the .NET platform, Windows 2000, followed by details on C#.

"C# is a modern, object-oriented development language that enables programmers to build solution on the Microsoft .NET platform quickly and easily," he said.

The duration of the course is 106 hours and is priced at ₹ 11,900.

It is targeted at active professionals in the IT industry and professionals from other streams who want to update and build applications on the .NET platform.

Pic.: Mr Rahul Thapan, Head, Education Service Division, Tata Infotech, at a press meet in the Capital on Wednesday.

Notes

8.11 Summary

- An Interface merely defines a contract between the producer and the consumer.
- Interface can contain methods, properties, events, and indexers.
- An interface can inherit from zero or more interface, which are called the explicit base interfaces of the interface.
- It is an error for an interface to directly or indirectly inherit from itself.
- Interfaces may be implemented by classes and structs.
- A class or struct that implements an interface also implicitly implements all of the interface's base interface.
- A class that inherits an interface implementation is permitted to reimplement the interface by including it in the base class list.
- Like a non-abstract class, an abstract class must provide implementation of all members of the interface that are listed in the base class.
- It is compulsory for a class or a struct, which implements an interface to implements all the methods and properties of the interface.
- Without explicitly reimplementing an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes.

8.12 Keywords

Abstract Class: A class with the prefix word abstract is called as abstract class. Abstract class cannot be instantiated.

Interface: An interface contains only the signatures of methods, delegates or events. The implementation of the methods is done in the class that implements the interface.

Interface Mapping: The process of locating implementations of interface members in an implementing class or struct is known as interface mapping.

Interface Reimplementation: A class that inherits an interface implementation is permitted to re-implement the interface by including it in the base class.

8.13 Review Questions

1. Can a C# class have more than one main method? If yes, what purpose does it serve?
2. Examine the purpose of aliasing a namespace. Explain with a example.
3. What could be the members of an interface? What are the rules that apply to interface member declarations?
4. What purpose do C# properties serve? What are the advantages of properties? Explain with examples.
5. Using a suitable example, explain how read-only properties may be used.
6. Explain interface mapping. How interface mapping helps in making the C# a strong programming language?
7. Explain the concept of interface re-implementation with the help of suitable examples.
8. Substantiate the usage of Alias in namespace class. Also explain what is system namespace?

9. Interfaces may be implemented by classes and structs. Explain.
10. An Interface merely defines a contract between the producer and the consumer. Do you agree with this statement? Why or why not? Give reasons to support your answer.

Notes

Answers: Self Assessment

- | | |
|----------------------------------|------------------|
| 1. interface | 2. accessibility |
| 3. base | 4. explicit base |
| 5. interface | 6. inherited |
| 7. interface-method-declarations | 8. semicolon |
| 9. events | 10. interface |
| 11. base | 12. multiple |
| 13. re-implementation | 14. void |
| 15. reference | |

8.14 Further Readings



Books

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

<http://www.csharp-station.com/Tutorials/Lesson13.aspx>

<http://www.dotnetperls.com/namespace>

Unit 9: Exception Handling

CONTENTS

Objectives

Introduction

9.1 Exceptions

9.2 Handling Exceptions

9.3 Try and Catch Statements

9.3.1 Multiple Catch Blocks

9.3.2 Nesting Try Blocks

9.4 Throw Clause

9.5 Using Finally Statement

9.6 User Defined Exception

9.7 System.Exception Class

9.8 Summary

9.9 Keywords

9.10 Review Questions

9.11 Further Readings

Objectives

After studying this unit, you will be able to:

- Introduce to exceptions
- Describe multiple catch statements
- Demonstrate the using of Finally Statements
- Recognize Nested Try Blocks

Introduction

This unit will introduce you to what exceptions are and how they are handled in C3 by using system exceptions that are designed for the purpose of handling errors. Handling of runtime errors is one of the most crucial tasks ahead of any programmer. As a programmer one must make provisions in case exceptions occur for the program to terminate or provide an alternative course of action.

9.1 Exceptions

There are many ways in which a program under execution may commit error. For instance, a program may attempt to divide a number by zero at some point of time in execution. A runtime

error in a program is referred to as an exception. Clearly it is an undesirable trait of a program. Note that exceptions are errors occurring at runtime. The compiler will not detect them.

One can take extra caution to include codes to avoid such conditions that give rise to exceptions. However one may be careful, exceptions cannot be ruled out.

Therefore, the programmer must make provisions for the program to terminate gracefully or to take alternative course of action in case an exception does occur. Traditionally, as in C programming, if-else-if ladder used to be employed for this purpose. This approach often renders the program utterly unreadable. Object-oriented programming frameworks including .NET have introduced an elegant way of catching and handling exceptions.

If an exception has its origin in an application program it is called application level exception. On the other hand if it is originated in the system program on the top of which the application program in question runs it is called system level exception.

Here we shall be dealing with the way C# handles exceptions though the approach is quite general, idiosyncrasy apart.

C# provides a well-defined, structured, uniform, and type-safe way of handling exceptions at both levels - application and system. An exception is represented by a class in C#. Exception class comes under the system namespace. C# uses try-catch-throw mechanism to handle exceptions. Simply stated it reads as: try (run under supervision) a block of code; in case of an exception catching it; create an object of it; and then throw it for some other object to handle it. Thus, an exception must be represented by an instance of System.Exception or any other class derived from System.Exception class. This is in contrast to languages like C++ where this restriction is not applicable and therefore any value of any type can be used to represent an exception.



Notes Another notable improvement is regarding the finally block. A finally block always executes irrespective of the condition normal or exceptional in C# programs. System-level exceptions are defined by different classes. Thus we have exception classes corresponding to overflow, divide-by-zero, and null dereferences exceptions.

9.2 Handling Exceptions

A C# program may throw an exception. The thrown exception may be handled by the program itself, however, it is not necessary. If the program that throws an exception also includes code to handle it, the exception is known as checked-exception. On the contrary, if the program simply throws an exception and allows the runtime environment to take the necessary actions in response, it is called unchecked exception.

For instance, division by zero is an unchecked exception. It is the runtime environment that handles this exception implying that a programmer may rely on the runtime system for its handling. Similarly, when a program code references an array with an index for which no array element exists, it causes an unchecked exception.

Consider a numeric variable in a program whose value is being read from the keyboard. If the value entered is non-numeric. This will indicate an exception. This exception is an example of checked exception because of course you would not like to leave this to the runtime environment. Instead you would like to insert code in the program itself to take the necessary action such as prompting the user to re-enter a valid numerical value.

Notes



Example:

```
class ExceptionExample1
{
    int x = 100;

    public static void Main()
    {
        System.Console.WriteLine("Enter the divisor:");
        int y = System.Convert.ToInt32(System.Console.ReadLine());
        double r = x/y;
        System.Console.WriteLine("Result = "+r);
    }
}
```

Look at the output for a couple of runs.

```
Enter the divisor:
You entered 20. The output would be:
        Result = 5
Run it again.
```

An exception 'System.DivideByZeroException' has occurred in ExceptionExample1.exe.



Notes The in-built exception System.DivideByZeroException was thrown by the program and subsequently caught by the runtime environment. In case you decide that your program catches this exception you will have to insert necessary codes into your program.

Self Assessment

Fill in the blanks:

1. An exception is represented by a in C#.
2. class comes under the system namespace.
3. If the program that throws an exception also includes code to handle it, the exception is known as
4. If no exception handler for a given exception is present, the program stops executing with an.....

9.3 Try and Catch Statements

In a C# program, try defines a code block which is likely to throw an exception. It is the onus on the programmer to anticipate which statements are capable of throwing an exception and which the program needs to catch. The codes included in the try clause may or may not throw an exception.

A try clause is associated with a catch block. This block catches the exception thrown by the try block and performs the actions specified in there.

Let us apply this mechanism to handle the exception thrown in the above program snippet. We want to dispatch a meaningful error message if division by zero is attempted. Enclose the statements in a try block (see the listing below).



Example:

```
class Exception
{
    int x = 100;

    public static void Main()
    {
        try
        {
            System.Console.WriteLine("Enter the divisor:");
            int y = System.Convert.ToInt32(System.Console.ReadLine());
            double r = x/y;
            System.Console.WriteLine("Result = "+r);
        }
        catch (System.DivideByZeroException exp)
        {
            System.Console.WriteLine("Error: Division by zero not allowed!");
            System.Console.WriteLine("Action: Program will terminate!");
        }
    }
}
```

Now this program catches the exception thrown by the try code (in this case System System.DivideByZeroException). When this exception is thrown by the try block, an object of the System.DivideByZeroException exception class is created (exp in this case) which is caught by the associated catch block and the program terminates.



Did u know? What is the use of Catch Statement?

The catch clause can be used without arguments, in which case it catches any type of exception, and referred to as the general catch clause. It can also take an object argument derived from System.Exception, in which case it handles a specific exception.

Our catch clause simply writes two lines of message to the console in response before terminating. This is certainly a better way of handling the exception than the previous one. To see the difference, run the program and input 0 as divisor. The following will be the output.

Enter the divisor:0

You enter 0. The program gives you the following output.

Error: Division by zero not allowed!

Notes

Action: Program will terminate!

Well! Now your program is equipped to catch the `System.DivideByZeroException` exception. What about other exceptions that this try block may throw. The answer is simple. You can associate more than one catch block to a single try block. For example, our program expects that you enter an integer for the input. What happens when you enter a fraction number instead of integer? To see the effect run the program `ExceptionExample2.exe` and input 1.5. The output will be as shown below.

Enter the divisor:1.5

An exception '`System.FormatException`' has occurred in `ExceptionExample2.exe`.

This exception (`System.FormatException`) was not handled by the program therefore the runtime has issued this error message. If you want your program to trap this error as well, attach one more catch clause for this exception (see the listing for `ExceptionExample4` given below).



Example:

```
class Exception
{
    int x = 100;

    public static void Main()
    {
        try
        {
            System.Console.WriteLine("Enter the divisor:");
            int y = System.Convert.ToInt32(System.Console.ReadLine());
            double r = x/y;
            System.Console.WriteLine("Result = "+r);
        }
        catch (System.DivideByZeroException exp)
        {
            System.Console.WriteLine("Error: Division by zero not allowed!");
            System.Console.WriteLine("Action: Program will terminate!");
        }
        catch (System.FormatException exp)
        {
            System.Console.WriteLine("Error: Non-integer input not allowed!");
            System.Console.WriteLine("Action: Program will terminate!");
        }
    }
}
```

Now run the program ExceptionExample.exe with input 1.5. The output will be as shown below.

Notes

Enter the divisor:1.5

Error: Non-integer input not allowed!

Action: Program will terminate!

Here you have also captured FormatException in addition to DivideByZero exception. This way you can make your program handle other exceptions. If you do not know which other exceptions may be thrown, you can include specify all the other exceptions by the class - System.Exception. The modified program is listed as ExceptionExample below.



Example:

```
class Exception
{
int x = 100;
public static void Main()
{
try
{
System.Console.WriteLine("Enter the divisor:");
int y = System.Convert.ToInt32(System.Console.ReadLine());
double r = x/y;
System.Console.WriteLine("Result = "+r);
}
catch (System.DivideByZeroException exp)
{
System.Console.WriteLine("Error: Division by zero not allowed!");
System.Console.WriteLine("Action: Program will terminate!");
}
catch (System.FormatException exp)
{
System.Console.WriteLine("Error: Non-integer input not allowed!");
System.Console.WriteLine("Action: Program will terminate!");
}
catch (System.Exception exp)
{
System.Console.WriteLine("Error: Some error has occurred!");
System.Console.WriteLine("Action: Program will terminate!");
}
}
```

Notes



Notes The name of the exception object being thrown in all the cases has been kept as exp. This however is not necessary. You can very well pick the names exp1, exp2, exp3 etc. Which exp represents which object depends on the catch block that traps it.



Task Can a try block followed by multiple catch statement? Justify your answer.

9.3.1 Multiple Catch Blocks

A try block can throw multiple exceptions, which can handle by using multiple catch blocks. Remember that more specialized catch block should come before a generalized one. Otherwise the compiler will show a compilation error.

```
//C#: Exception Handling: Multiple catch
using System;
class MyClient
{
public static void Main()
{
int x = 0;
int div = 0;
try
{
div = 100/x;
Console.WriteLine("Not executed line");
}
catch(DivideByZeroException de)
{
Console.WriteLine("DivideByZeroException" );
}
catch(Exception ee)
{
Console.WriteLine("Exception" );
}
finally
{
Console.WriteLine("Finally Block");
}
Console.WriteLine("Result is {0}",div);
}
```

```
}  
}
```

9.3.2 Nesting Try Blocks

```
using System;  
class MainClass {  
    public static void Main() {  
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };  
        int d = 0;  
        try { // outer try  
            for(int i=0; i < 10; i++) {  
                try { // nested try  
                    Console.WriteLine(numer[i] + " / " +  
                                     numer[i] + " is " +  
                                     numer[i]/d);  
                }  
                catch (DivideByZeroException) {  
                    // catch the exception  
                    Console.WriteLine("Can't divide by Zero!");  
                }  
            }  
        }  
        catch (IndexOutOfRangeException) {  
            // catch the exception  
            Console.WriteLine("No matching element found.");  
            Console.WriteLine("Fatal error -- program terminated.");  
        }  
    }  
}  
  
Can't divide by Zero!  
No matching element found.  
Fatal error -- program terminated.
```

Notes

Self Assessment

Fill in the blanks:

5. It is possible to use more than one specific catch clause in the same statement.
6. Our catch clause simply writes two lines of message to the console in response before.....
7. If you want to re-throw the exception currently handled by a parameter-less catch clause, use the throw statement without
8. A try clause is associated with a block.
9. When catching an exception in C# you need not declare a for the exception instance if you do not need it.

9.4 Throw Clause

A C# program throws an exception whenever an abnormal condition arises in the program. However, the programmer is also at the liberty of throwing an exception if he desires so. This is accomplished by throw clause.

Throw clause when executed throws the specified exception unconditionally whether or not any exception has occurred. Look at the following program (ExceptionExample) in which the programmer throws an exception when the input does not match with the string stored in a variable.



Example:

```
class ExceptionExample5
{
    string name;
    public static void Main()
    {
        try
        {
            System.Console.WriteLine("Which is the highest mountain peak in the world?");
            name = System.Console.ReadLine();
            if (name != "Everest")
                throw new System.Exception();
            else
                System.Console.WriteLine("Right answer!");
        }
        catch(Exception e)
        {
            System.Console.WriteLine(e.ToString()+"Sorry! It is the Everest");
        }
    }
}
```

```

    }
}

```

Run the program ExceptionExample.exe. You will see the following.

```
Which is the highest mountain peak in the world? Kanchenjunga
```

The output will be:

```
Sorry! It is the Everest
```

You must have realized by now that throw serves more purposes than it meets the eyes.

9.5 Using Finally Statement

At times one wants to execute a piece of code irrespective of whether an exception occurred or not. Such codes are included in the finally clause.



Notes The catch clause code is executed only when an appropriate exception occurs but finally clause codes are always executed.



Example:

```

class ExceptionExample
{
    string name;
    public static void Main()
    {
        try
        {
            System.Console.WriteLine("Which is the highest mountain peak in the world?");
            name = System.Console.ReadLine();
            if (name != "Everest")
                throw new System.Exception();
            else
                System.Console.WriteLine("Right answer!");
        }
        catch(Exception e)
        {
            System.Console.WriteLine(e.ToString()+"Sorry! It is the Everest");
        }
        finally {
            System.Console.WriteLine("It is Nepal ");
        }
    }
}

```

Notes

Run the program ExceptionExample.exe with correct answer. The output will be:

```
Which is the highest mountain peak in the world? Everest
Right answer!
It is in Nepal
```

Run the program ExceptionExample.exe again this time with incorrect answer. The output will be:

```
Which is the highest mountain peak in the world?Kanchenjunga
Sorry! It is Everest
It is in Nepal
```

Observe that in both the cases, whether the exception was thrown or not, the finally clause has executed.



Task Analyze the use of finally statement? Explain with the help of suitable example.

Self Assessment

Fill in the blanks:

10. The statement in the C# language provides a way to generate or translate exceptions.
11. The throw statement, then, provides an essential ability to an exception or generate a new exception.
12. The block is useful for cleaning up any resources allocated in the try block.

9.6 User Defined Exception

Though in-built exceptions serve the purpose very well, there are situations where the programmer needs to create his own exceptions. All the user-defined exceptions class must derive from the System.Exception base class or any other derivable class which itself derives from System.Exception class. In the following example - ExceptionExample - this concept is elucidated. Here we will create an exception class ExceptionExample having two constructors - one without any parameter and the other having a parameter name.

```
public class MyException : System.Exception
{
public
MyException()
{
System.Console.WriteLine("The integer entered is more than 10");
}
}
class ExceptionExample7
```

Notes

```

{
    public static void Main()
    {
        int x;
        System.Console.WriteLine("Enter an integer");
        x = System.Convert.ToInt32(System.Console.ReadLine());
        try
        {
            if(x > 10) throw new MyException();
        }
        catch(MyException e)
        {
        }
    }
}

```

When you run the program `ExceptionExample.exe` and input integer 12, an exception of `MyException` class is thrown which is caught by the catch block and consequently you will see the following output on the console.

The integer entered is more than 10



Notes The catch clause does not print any message. It is the exception object that issues the message. If you want you can also add some code into the catch clause itself. Also note that by catching `System.Exception` class will catch all the exceptions occurring in the corresponding try block because all the exceptions are direct or indirect derivative of `System.Exception` class.

An exception may be thrown multiple times. Thus the catch block can again throw the caught exception as shown in the following program listing `ExceptionExample`. Here the caught exception is passed to the base class of the exception class in which the current exception was caught.



Example:

```

class ExceptionExample
{
    public static void Main()
    {
        int x;
        System.Console.WriteLine("Enter an integer");
        x = System.Convert.ToInt32(System.Console.ReadLine());
        try
        {

```

Notes

```
if(x > 10) throw new MyException();
    }
    catch(MyException e)
    {
        throw e;
    }
}
```

9.7 System.Exception Class

As you have seen every exception must derive from the base class System.Exception class. Here is a synopsis of some of the important members and methods.

- Message is a read-only property that contains a human-readable description of the reason for the exception.
- InnerException is a read-only property that contains the “inner exception” for this exception. If this is not null, this indicates that the current exception was thrown in response to another exception. The exception that caused the current exception is available in the InnerException property.
- The value of these properties can be specified in the constructor for System.Exception.

The public properties and methods of System.Exception are listed below.

- bool System.Exception.Equals(System.Object)
This method is inherited from System.Object class which incidentally is the root of all the classes in C#. The method compares the object passed as parameter to the current System.Exception object
- System.Exception GetBaseException()
When overridden in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.
- int System.Exception.GetHashCode()
This methods returns to Serves as a hash function for a particular type. It is suitable for use in hash algorithms and data structures like hash table.
- void GetObjectdata(System.Runtime.SerializationInfo info, System.Runtime.Serialization.StreamingContext context)
When overridden in a derived class, sets the System.Runtime.SerializationInfo with information about the exception.
- string HelpLink
Gets or sets the link to the help file associated with this exception.
- System.Exception InnerException
Gets the System.Exception instance that caused the current exception.
- string Message
Gets a message that describes the current exception.

- string Source
Gets or sets the name of the application or object that caused the exception.
- string Stacktrace
Gets a string representation of the frames on the call stack at the time the current exception was thrown.
- System.Reflection.MethodBase TargetSite
Gets the method that throws the current exception.
- Common Exception Classes
Though the list of in-built exceptions in C# is very large, here is a synopsis of some of the important ones.
- System.OutOfMemoryException
Thrown when an attempt to allocate memory (via new) fails.
- System.StackOverflowException
Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
- System.NullReferenceException
Thrown when a null reference is used in a way that causes the referenced object to be required.
- System.TypeInitializationException
Thrown when a static constructor throws an exception, and no catch clauses exists to catch in.
- System.InvalidCastException
Thrown when an explicit conversion from a base type or interface to a derived types fails at run time.
- 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.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.MulticastNotSupportedException
Thrown when an attempt to combine two non-null delegates fails, because the delegate type does not have a void return type.
- System.ArithmeticException
A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.
- System.DivideByZeroException
Thrown when an attempt to divide an integral value by zero occurs.
- System.OverflowException
Thrown when an arithmetic operation in a checked context overflows.

Notes



Task List down some of important in-built exception classes.

Self Assessment

Fill in the blanks:

- 13. All the user-defined exceptions class must derive from the base class.
- 14. When in a derived class, returns the System.Exception that is the root cause of one or more subsequent exceptions.
- 15. Though in-built exceptions serve the purpose very well, there are situations where the programmer needs to create his own.....



Caselet **Patterns Hiding in Mountains of Data**

LOOK at these recent reports. If the Food and Drug Administration of the US ties up with major health insurers and reviews patient histories, such a study can reveal possible safety red flags about drugs in use; but according to Harvard Medical School professor Jerry Avorn, cited in www.boston.com, such data mining would cost about \$500,000 per drug.

A tax compliance software called GoSystem has a data mining engine by the name FormSource; it helps identify planning opportunities from the simplest to the most complex returns, as www.webcpa.com informs.

A few weeks ago, Fujitsu and France Telecom agreed to launch a joint research project on grid computing technology. According to www.rednova.com, the first phase of the project will focus on facilitating analysis of France Telecom’s huge volume of data, such as for data mining and customer billing.

The common thread in the above stories is ‘data mining’. The topic, therefore, is hot; and so is the second edition of Data Mining Techniques by Michael J.A. Berry and Gordon S. Linoff, published by Wiley Dreamtech India P Ltd (www.wileydreamtech.com).

The book is aimed at data mining practitioners, not software developers, clarifies the intro. Therefore, “ideas are presented in non-technical language with minimal use of mathematical formulas and arcane jargon.”

While data warehouse provides the company with a big memory, you achieve intelligence only when you are able to comb through the bits, notice patterns, devise rules, come up with new ideas, figure out the right questions, and predict the future; to do which you need data mining, advise the authors.

Data mining comes in two flavours, not shallow and deep, but directed and undirected. The former “attempts to explain or categorise some particular target field such as income or response”.

The latter is more challenging; it finds patterns among groups of records “without the use of a particular target field or collection of predefined classes.”

Contd...

For their target audience - that is, those in marketing, sales, and customer relationship management - the authors provide numerous real-world examples.

Here is one such, to explain why data may be at the wrong level of detail when using declining customer usage as an early warning before the customer leaves a cellular operator. "For seven months, the subscriber used about 100 minutes per month. Then, in the eighth month, usage went down to about half that. In the ninth month, there was no usage at all."

If you inferred that the customer switched loyalty only in the ninth month, you may be wrong. "Looking at minutes of use by day instead of by month would show that the customer continued to use the service at a constant rate until the middle of the month and then stopped completely, presumably because on that day, he began using a competing service."

"If you torture data sufficiently, it will confess to almost anything," is a Fred Menger quote. But with Data Mining Techniques to help, you will need to torture data less, to know more.

'Secrets' of Developing Accounting Package

WE know that accountants live in a different world, disclosing their secretly sacred work in a format that many don't understand. To know their minds, try a different route: Learn the 'secrets of developing an accounting package'! Which is what Bharati and Krishna reveal in Database Programming Using VB.Net & SQL Server 2000, from VK Publishers (www.vkinfotek.com).

The first chapter running to about a hundred pages explains the components of .Net framework, user interface elements of VB.Net, and OOP implementation.

What follows is a crash course in accounting. Double entry lies at the core of accounting, say the authors, and more flatteringly for bean-counters, it is extolled as "the finest discovery of human intellect."

Auditors should particularly focus on the inputs about SQL, so they can issue a command that may possibly read: `USE FinAccounting SELECT *FROM AccountsTable WHERE AccountName LIKE 'Osama%'`

There are only four types of 'new accounts' - viz. creditor, debtor, bank and general. This may not gel with how your munimji operates, yet the discussion that runs across the book guides one to create a chart of accounts, develop 'dialog' boxes, 'filter' to make subsets of datasets, build a 'tran class' as field vs transaction matrix, and so forth.

To the question, "what is a transaction?" an accountant may respond that it is the financial interaction between people.

Not so the duo, Bharati and Krishna: "A transaction is a series of actions that must either succeed, or fail, as a whole. If one of the actions fails, then the entire transaction fails and all the changes made to the database so far, must be reversed or rolled back."

The book, I suppose, can serve a dual purpose: Techies can know how accounting works, and likewise, accountants too get into the minds of system developers.

Tailpiece

"They found that the criminal used a laptop to track his victim and murder..."

"Oh!"

"And the case has been transferred to the IT cops to find the criminal's address from the hard disk!"

Notes

9.8 Summary

- Trapping and handling of runtime errors is one of the most crucial tasks ahead of any programmer. Programmers must make provisions for the program to terminate gracefully or take alternative course of action in case an exception does occur.
- C# provides three keywords try, catch and finally to do exception handling.
- The finally can be used for doing any clean up process.
- If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.
- If there is no exception occurred inside the try block, the control directly transfers to finally block.
- The user-defined exception classes must inherit from either Exception class or one of its standard derived classes.
- If code catches an exception that it isn't going to handle, consider whether it should wrap that exception with additional information before re-throwing it.

9.9 Keywords

Exception: An exception is an error condition or unexpected behavior encountered by an executing program during runtime.

Exception Handling: It is an in-built mechanism in .NET framework to detect and handle run time errors.

Finally Clause: The finally block is useful for cleaning up any resources allocated in the try block as well as running any code that must execute even if there is an exception. Control is always passed to the finally block regardless of how the try block exits.

Throw Statement: The throw statement is used to signal the occurrence of an anomalous situation (exception) during the program execution.

Try and Catch Statement: The try-catch statement consists of a try block followed by one or more catch clauses, which specify handlers for different exceptions.

9.10 Review Questions

1. How is an 'Exception' different from 'Error'? Explain with examples.
2. Describe the members of 'Exception Class'.
3. 'Exception' have multiple 'Catch Statements'. If yes explain.
4. Write a program to explain the concept of Exceptional Handling.
5. Write a program to accept 10 numbers in an array and print the sorted list via incorporating the concept of exception handling.
6. If an exception has its origin in an application program it is called application level exception. Discuss.
7. Substantiate Exception. Write a program to explain the concept of Exceptional Handling.
8. Explain checked and unchecked exception. Explain the Throw clause with the help of suitable example.

9. A C# program throws an exception whenever an abnormal condition arises in the program. Demonstrate this statement with the help of a suitable program.
10. Message is a read-only property that contains a human-readable description of the reason for the exception. Discuss.

Notes

Answers: Self Assessment

- | | |
|----------------------|------------------|
| 1. class | 2. Exception |
| 3. checked-exception | 4. error message |
| 5. try-catch | 6. terminating |
| 7. Arguments | 8. catch |
| 9. variable | 10. throw |
| 11. rethrow | 12. finally |
| 13. System.Exception | 14. Overridden |
| 15. exceptions | |

9.11 Further Readings



Books

CLR via C#, Second Edition (Paperback) by Jeffrey Richter, Publisher: Microsoft Press; 2nd edition (February 22, 2006)

C# Programming Language, The (2nd Edition) (Microsoft .NET Development Series) (Hardcover) by Anders Hejlsberg, Scott Wiltamuth, Peter Golde Publisher: Addison-Wesley Professional; 2 edition (June 9, 2006)

Essential C# 2.0 (Microsoft .NET Development Series) (Paperback) by Mark Michaelis, Publisher: Addison-Wesley Professional (July 13, 2006)

Programming in the Key of C# (Paperback) by Charles Petzold, Publisher: Microsoft Press; 1 edition (July 30, 2003)



Online links

[http://msdn.microsoft.com/en-us/library/dszsf989\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/dszsf989(v=vs.71).aspx)

[http://msdn.microsoft.com/en-us/library/zwc8s4fz\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/zwc8s4fz(v=vs.71).aspx)

Unit 10: Window Programming

CONTENTS

Objectives

Introduction

10.1 Using Controls

10.2 The Text Box Control

10.3 The List Box and Combo Box Controls

10.3.1 Important Properties

10.3.2 Important Events

10.3.3 Methods

10.4 Radio Buttons and Check Boxes

10.5 Scroll Bars

10.6 The DateTimePicker Control

10.7 Summary

10.8 Keywords

10.9 Review Questions

10.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the basic controls
- Describe Text Boxes
- Demonstrate the List Boxes
- Recognize Buttons
- Explain Datetime Picker
- Scan the Combo Boxes

Introduction

Visual Basic provides a number of controls to facilitate application interface design. These controls can be found in the Visual Basic toolbox. Command buttons, label controls, scroll bars, text boxes are a few of the basic controls that are always required while developing a GUI based application. This unit covers a brief introduction to the basic controls, using them, their properties and methods. Although sufficient number of examples has been given to support the explanations, still readers are advised to be a bit creative and imaginative by letting their imagination free and then design the applications.

10.1 Using Controls

Controls are the basic foundation of any application providing Graphical User Interface. (GUI) Some controls are used to take an input from the user and other controls like command button and text boxes elicit responses. There are few controls that work for you behind the scene like timer control, a common dialog box, an image list control etc. Using controls is not a difficult task if you use them sincerely. Each control has its own properties, methods and events. Although most of them have few common properties, methods and events, but each control also has its own properties, methods and events. These controls can be easily placed on the forms by selecting them from the toolbox and then performing drag operation. You can also double click a control icon in the tool box to add it to a form.

Toolbox window contains all the basic controls when you open a new project. You can add new controls from the Components option in the Project menu.

Besides these controls there are several advance controls that come with Visual Basic 7.0. In addition you can develop your own ActiveX controls and use them just like any other custom controls and even distribute them to other developers through a variety of methods. Some controls provided with Visual



Notes Basic work with Multimedia, OLE and some controls can even be utilised in building Internet applications.

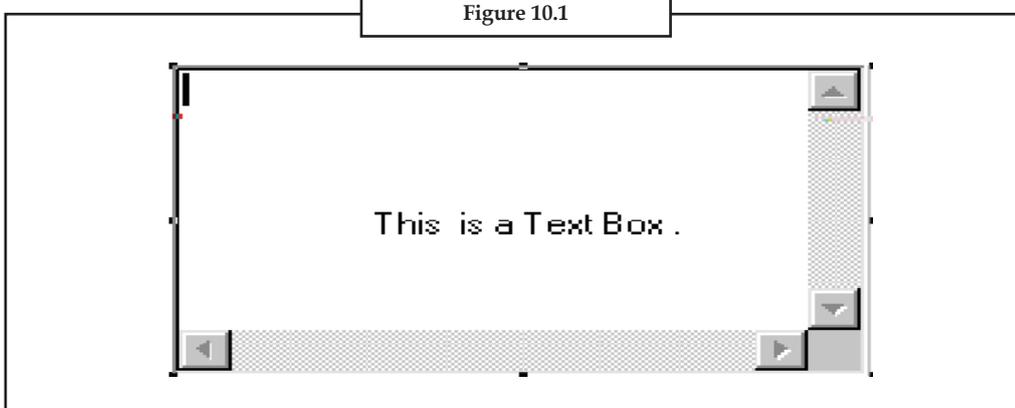
This unit covers only few basic controls. Other advanced controls and ActiveX controls have been discussed in the upcoming units.

10.2 The Text Box Control

Generally almost each and every application uses a Text Box to accept an input from the user as it has all the required properties and events that can be used in an application. The Text Box control has the Text property as the default property. You can add Scrollbars to it, set Font for the text, set its backcolor. Multiline property enables you to allow typing in multilines.

Textbox Control is also a data aware control i.e., it can be bound to a data control to display field values. The Password-char property-allows you to specify the characters to be used while you don't want to display the actual text. You can even enable or disable a Text Box at runtime using its Enabled property. You can even stop a user from editing text using Locked property of the Text Box.

Figure 10.1



Notes



Did u know? **What are the things which are offered by the Text Box?**

TextBox controls offer a natural way for users to enter a value in your program. For this reason, they tend to be the most frequently used controls in the majority of Windows applications.

Let us now see few important properties of a Text Box.

Important Properties

Text	Displays the text in the Text Box.
PasswordChar	Specifies the character to be displayed if the Text Box is used to accept the password.
ScrollBars	Allows ScrollBars in the Text Box, only if Multiline property is set to True.
DataField	Specifies the field name of the table connected to the Text Box through data control to retrieve the information.
MaxLength	Specifies the maximum number of characters the Text Box will accept.
DataSource	Specifies the name of the Data Control.
Multiline	Specifies whether a Text Box accepts a new line on pressing Enter key or giving linefeed character, or not.
Locked	Makes a Text Box read only if set to True.
Visible	Makes the Text Box invisible if set to False.

Important Methods

FontSize, FontBold, FontItalic, FontStrikethru, FontUnderline are some more properties associated with the control. Move, Refresh are few methods applied on the Text Box control.

Important Events

TextChanged	Triggers when any text is added or deleted in the Text Box.
KeyPress	Triggers when the Text Box is the active control and any key is pressed from the keyboard.
Click	Triggers when the user clicks on the Text Box.

Capturing the Key Strokes

Visual Basic provides an easy way to capture all the key strokes, especially in Text Boxes. This can easily be done by writing few codes on the KeyPress events of any object which itself returns the ASCII code of the key pressed in KEYASCII variable. Almost all the controls in Visual Basic including forms have KeyPress event.



Example:

```
Private Sub TextBox_KeyPress (KeyAscii as Integer)
    Dim KeyPressedIs As String
```

```

KeyPressedIs = KeyASCII
MsgBox "KeyPressed is" + KeyPressedIs
End Sub

```

Here chr is a function which changes an ASCII code to its equivalent character.

Self Assessment

Fill in the blanks:

1. TextBox controls, which have a great many properties and events, are also among the most complex controls.
2. Text can be entered into the text box by assigning the necessary to the text property of the control
3. If the user needs to display multiple lines of text in a TextBox, set the property to True.
4. If you set the Multiline property to True, you can set the alignment using the property.
5. The Text property is the one you'll reference most often in code, and conveniently it's the property for the TextBox control.

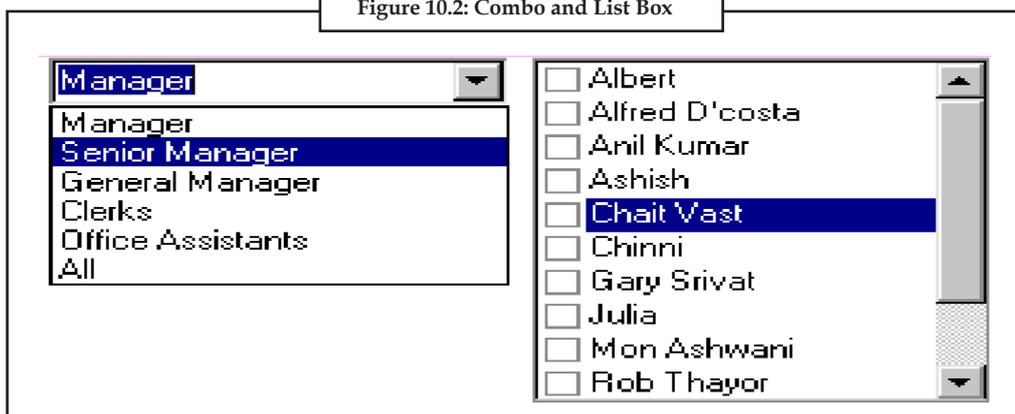
10.3 The List Box and Combo Box Controls

A List box is an ideal way of presenting users with a list of data. A list box control displays a list of items and allows you to select one of them.

A ScrollBar is automatically added to a list box control if the number of items exceed its size. The user can't edit the data in the list box.

A combo Box is simply a drop down list box. It combines the features of a text box and a list box, both. A combo box allows you to select an item from the list and even add a new item.

Figure 10.2: Combo and List Box



Notes Combo boxes are so-named because they “combine” the features found in both text boxes and list boxes. Combo boxes are also commonly referred to as “drop-down boxes” or “drop-down lists”.

Notes

10.3.1 Important Properties

Columns

This property controls the number of columns in a List Box.

Value

- 0 (Default) single column List Box with vertical scrolling.
- 1 single column List Box with horizontal scrolling.
- >1 Multiple column List Box.

ListCount

The ListCount property gives the number of items in the List Box.

ListIndex

Returns the index number of the selected item.

Text

Text property returns the currently selected item stored as string.



Notes Like a ListBox, and unlike the other combo box styles, the user can only select a value that is in the list they cannot type in a new value.

Style

Style property in Combo Box and List Box returns or sets the value indicating display type and behaviour of the control. Settings for a Combo Box are

Constant	Value	Description
VbComboDropDown	0	Drop down ComboBox.
VbComboSimple	1	Simple Combo Box includes a Text Box and a list which doesn't drop down.
VbDropDownList	2	Drop down list that allows selection.

Settings for a List Box are:

Constant	Value	Description
VbListBoxStandard	0	Standard List Box.
VbListBoxCheckbox	1	Displays a check box with the list items.

List

It is a string array that contains all the items on the list. Each item in this array can be accessed by its index number at run time.

Thus

Notes

List1.Text and List1.List(ListIndex) will give the same string.



Caution The ListBox portion of the combo box remains hidden until the combo box receives focus and the user clicks the down arrow on the text box.

Multiselection

This property determines how the user can select items from the List Box.

Value	Type of Selection
0	No multi selection allowed.
1	Simple multi selection.
2	Extended multi selection.

10.3.2 Important Events

Click and double click are the important events for the List Box control.

10.3.3 Methods

AddItem

1. Adds an item to a list or Combo Box.
2. List1.AddItem "Ashish Srivastava"

OR

Name = "Ashish Srivastava"

List1.AddItem Name

RemoveItem

Removes an item from the List Box or Combo Box if the index number of the item is supplied to the method.



Example:

List1.RemoveItem(Index number)

Combo1.RemoveItem(Index number)

Clear

Clears the list or Combo box.

List1.Clear

Combo1.Clear

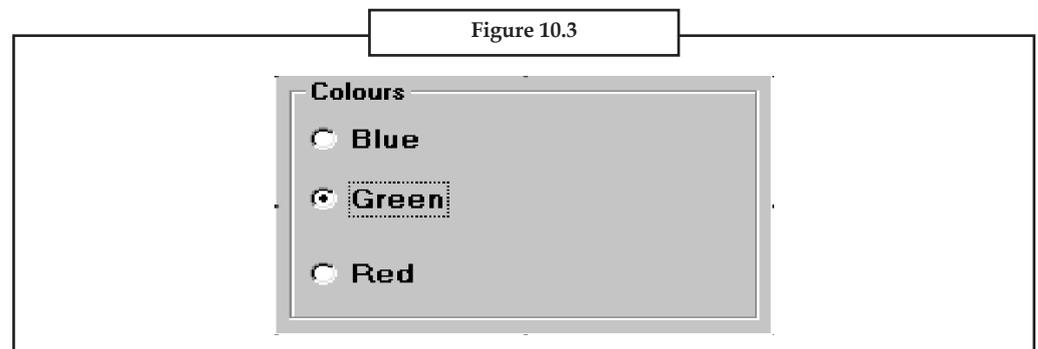
Notes

Self Assessment

Fill in the blanks:

- 6. A is automatically added to a list box control if the number of items exceed its size.
- 7. A combo Box is simply a list box.
- 8. It is a string array that contains all the items on the.....
- 9. Style property in Combo Box and List Box returns or sets the value indicating type and behaviour of the control.

10.4 Radio Buttons and Check Boxes



Option buttons are also called as Radio Buttons. Option buttons always work together- when the user chooses one button, all the other buttons in the group are turned off. Thus, if one form contains more than one group, then it should be kept in different frame controls to identify the group of option buttons separately.



Did u know? **What are the uses of check boxes?**

The Check box is a very useful control in Visual Basic 2008. It allows the user to select one or more items by checking the checkbox/checkboxes concerned.

A Check Box control is rather similar to an option button. The Value property of both the controls is tested to check the current state. Check boxes are valid as a single control whereas a single option button is probably counts-intuitive. Check Boxes are not mutually exclusive.

Important Properties

Value: Specifies whether the Option button or Check Box has been checked or not. Value is either True or False for Option Buttons and values for Check Boxes are

- 0 Unchecked
- 1 Checked
- 2 Grayed

Enabled: Specifies whether the Option button or the Check Box is enabled or disabled.

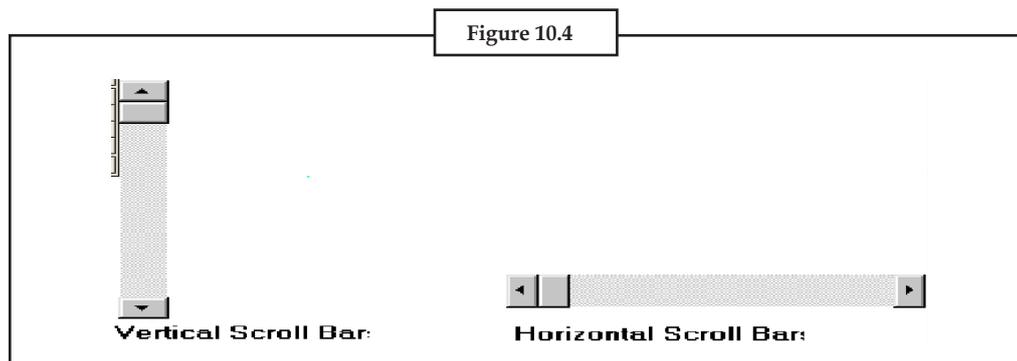
Self Assessment

Notes

Fill in the blanks:

10. A Check Box control is rather similar to an.....
11. Check boxes are valid as a control whereas a single option button is probably counts-intuitive.
12. are not mutually exclusive.

10.5 Scroll Bars



Visual Basic provides two types of scroll bars—horizontal and vertical. These scroll bars are provided for a different purpose unlike those used in wordpad documents. Basically these are used to accept values from a numerical range from the user, as the one used in Windows Custom Color Palette Application. Scroll bars have 9 events and 26 properties of which 5 are important.

Important Properties

Min	An integer value that specifies the smallest value for a scroll bar.
Max	An integer value that specifies the largest value for a scroll bar.
Value	Returns the current position of the bar on the scrolling track. Returns an integer.
SmallChange	Specifies the change in the value property when a user clicks at the scrolling arrow. Can also be called as the distance covered by the bar on the track on one click. This value is between 1 and 32,767. Available at run and design time.
LargeChange	Specifies the change in the value property when a user clicks on the scrolling track. Default large change is 1.



Example: Using Option Buttons, Check Boxes and Scroll Bars

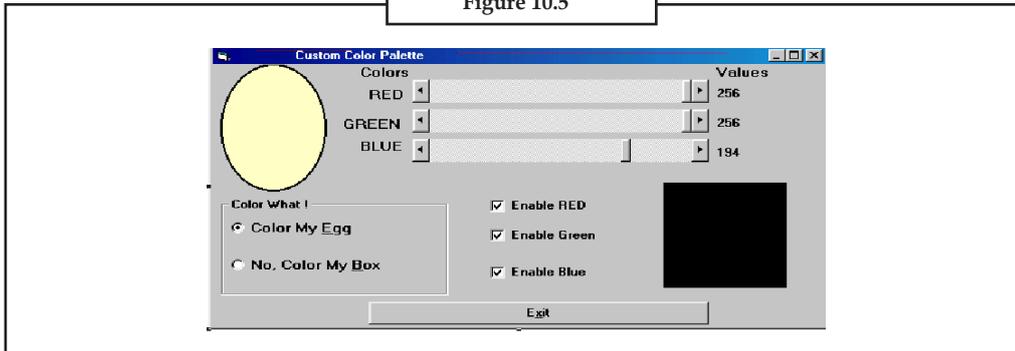
Notes

Properties to be Set at Design Time

Name	Type	Property	Value
Form 1	Form	Caption	"Custom Color Palette"
Command 1	Command Button	Caption	"E&XIT"
CHKGreen	Check Box	Caption	"Enable Green"
CHKRed	Check Box	Caption	"Enable Red"
Frame 1	Frame	Caption	"Color What!"
OptBox	Option Button	Caption	"No, Color My &Box"
OptEgg	Option Button	Caption	"Color My &Egg"
HScroll1	HScroll Bar	Large Change	10
HScroll1	HScroll Bar	Max	256
HScroll1	HScroll Bar	Small Change	5
HScroll2	HScroll Bar	Large change	10
HScroll2	HScroll Bar	Max	256
HScroll2	HScroll Bar	Small Change	5
HScroll3	HScroll Bar	Large Change	10
HScroll3	HScroll Bar	Max	256
HScroll3	HScroll Bar	Small Change	5
Box	Shape	BackStyle	1 (opaque)
Box	Shape	Shape	1 (Square)
Egg	Shape	BackStyle	1 (opaque)
Egg	Shape	Shape	2 (oval)
Label1	Label	Caption	"0"
Label 2	Label	Caption	"0"
Label 3	Label	Caption	"0"
-	-	-	-
-	-	-	-
Label 8	Label	Caption	"0"
Label 1	Label	Auto Size	"True"
Label 2	Label	Auto Size	"True"
Label 3	Label	Auto Size	"True"
-	-	-	-
-	-	-	-
Label 8	Label	Auto Size	"True"

After setting these properties on your form, arrange the controls on the form as shown in figure 10.5.

Figure 10.5



At the General Selection declare the following variables:

```
Dim r,g,b as Integer
Private Sub ChkBlue_Click()
    If ChkBlue.Value = 1 Then
        HScroll13.Enabled = True
    Else
        HScroll13.Enabled = False
    End If
End Sub
Private Sub ChkGreen_Click()
    If ChkGreen.Value = 1 Then
        HScroll12.Enabled = True
    Else
        HScroll12.Enabled = False
    End If
End Sub
Private Sub ChkRed_Click()
    If ChkRed.Value = 1 Then
        HScroll11.Enabled = True
    Else
        HScroll11.Enabled = False
    End If
End Sub
Private Sub Command1_Click()
    End
End Sub
Private Sub Form_Load()
    OptEgg.Value = True
    HScroll11.Enabled = False
```

Notes

```
HScroll2.Enabled = False
HScroll3.Enabled = False
Label1.Caption = " "
Label2.Caption = " "
Label3.Caption = " "
Label4.Caption = "Red"
Label5.Caption = "Green"
Label6.Caption = "Blue"
End Sub

Private Sub HScroll1_Change()
    r = HScroll2.Value
    g = HScroll2.Value
    b = HScroll3.Value
    Egg.BackColor = RGB (r,g,b)
    Box.BackColor = RGB (r,g,b)
    Label1.Caption = HScroll11.Value
End Sub

Private Sub HScroll2_Change()
    r = HScroll11.Value
    g = HScroll2. Value
    b = HScroll3. Value
    Egg. BackColor = RGB (r,g,b)
    Box.BackColor = RGB (r,g,b)
    Label2.Caption = HScroll2.Value
End Sub

Private Sub HScroll3_Change()
    r = HScroll11.Value
    g = HScroll2.Value
    b = HScroll3.Value
    Egg.BackColor = RGB (r,g,b)
    Box.BackColor = RGB (r,g,b)
    Label3.Caption = HScroll3.Value
End Sub

Private Sub OptBox_Click()
    If OptBox.Value then
        Egg.FillStyle = 0
        Box.FillStyle = 1
    End If
End Sub
```

```

End If
End Sub
Private Sub Opt_Egg_Click()
    If OptEgg.Value Then
        Box.FillStyle = 0
        Egg.FillStyle = 1
    End If
End Sub

```

Self Assessment

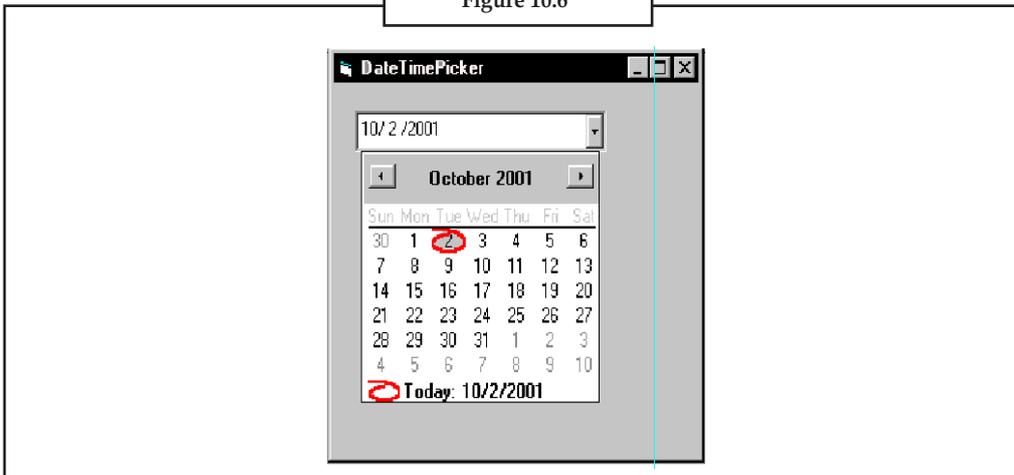
Fill in the blank:

- 13. specifies the change in the value property when a user clicks on the scrolling track.

10.6 The DateTimePicker Control

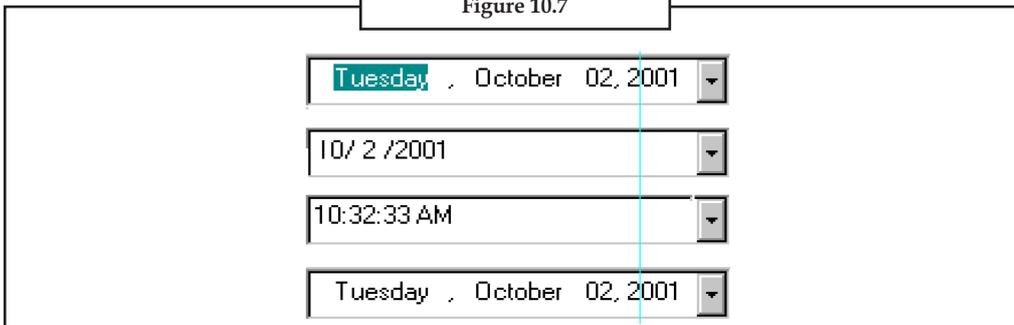
The DateTimePicker control can be called as the combination of a ComboBox and a MonthView control. The DateTimePicker control is very much similar to MonthView control but however these two controls are not same. The DateTimePicker control lets you select both date and time.

Figure 10.6



The DateTimePicker control provides four different formats for values.

Figure 10.7



Notes

The DateTimePicker control displays a drop down calendar like the MonthView control, but does not provide enough properties to control the display settings of the control. This control can be very useful and may provide a powerful interface in application that needs date, time inputs from the user. Now lets explore the properties, methods and events of this control.

Properties of The DtPicker Control

Although DateTimePicker or say DTPicker control has very few properties that let you control the display settings of the calendar, most of them resemble the MonthView control's properties.

Colour Properties

The CalendarBackColor, CalendarForeColor controls the back colour and the colour of the dates of the calendar and resembles the MonthBackColor and ForeColor properties respectively. The CalendarTrailingForeColor specifies the colour of dates from previous or trailing months.



Notes The CalendarTitleForeColor and CalendarTitleBackColor specifies the fore colours and back colours of the TitleBar that shows the month and year.

MinDate and MaxDate

MinDate and MaxDate properties control the range of date that can be displayed, similar to the MonthView control.

Value

The Value property returns the date or time selected. Similarly, Month, Year, Day, Hour, Minute and Second properties return specific information from the DateTimePicker control's value property.

DayOfWeek

Starting day of the week can be specified by DayOfWeek property, instead of Sunday which is default. This property is similar to StartOfWeek property used in the MonthView control.

UpDown

Up and Down arrow keys can be used for exploring in the Calendar, if the UpDown Property is set to true.

CheckBox

The CheckBox property when set to True displays a checkbox that shows whether any date is selected or not. If no date is selected the checkbox is unmarked and marked when the date is selected.

Format

This property is used to determine the format of the date and time to be displayed in the DateTimePicker control. Format property has the following syntax

`Object.Format = Integer`

The settings for integer values are :

Notes

Constant	Value	Example
DtpLongDate	0	"Monday, May 17 1979"
DtpShortDate	1	"11/12/76"
DtpTime	2	"10:20:55"
DtpCustom	3	Custom format specified in the CustomFormat property.

CustomFormat

Returns or sets the value that specifies the format used by the control when displaying date or time in the box.



Task CustomFormat property works only when the value of the Format property is set to DtpCustom. Explain with example.

Specifications in the CustomFormat property includes format characters which are given below. The property values are case sensitive.

Format Characters	Description
d	One-or-Two-digit Day.
dd	Two digit day
ddd	Week Day abbreviation of three characters e.g., MON,TUE etc.
dddd	Full Week Day name
h	One-or-Two Digit hour in 12 hour format, e.g., 6,1,11 etc.
hh	Two digit hour in 12 hour format, e.g., 09,12,06 etc.
H	One-or-Two Digit hour in 24 hours format.
m	One-or-Two Digit minute
mm	The Two-digit minute.
M	One-or-Two digit month number.
MM	Always two digit month number, e.g., 01,02,07
MMM	Month abbreviation of three characters.
MMMM	Full month name.
S	One-or-two digit Second.
SS	Always two digit second.
t	One letter AM/PM abbreviation.
tt	Two-letter AM/PM abbreviation.
X	Callback field. The control uses the other valid format characters and asks the user to fill in the "X" portion. Multiple "X" characters can be used in series to signify unique callback fields.

Notes	y	One-digit year.
	YY	Last Two digits of the year.
	yyy	Full year with the century.

As discussed above, the DTPicker control uses the other valid format characters and asks the user to fill in the "X" portion, called callback field. In order to use this callback field you must set the Format property to DtpCustom and CustomFormat property with a new string containing format.



Task Match the following

1. Load Picture	a. KeyPress
2. AutoSize	b. Grayed
3. Image Box	c. Label
4. List Box	d. Scroll()
5. Check Box	e. Loads a picture

For example, you want the date to be displayed in the following format, where the month names are in Spanish language.

Julio, Saturday, 1976

The first step is to set the format property to DtpCustom and CustomFormat property with the following format

xxxx, dddd, yyy

The "x" included here determines the Custom name for the month.

Events of DtPicker Control

Everytime when the new value is asked by the user, FormatSize event is fired which has callbackField as string and size as integer arguments. This event allows you to set the maximum allowable size of the formatted string.

Dim NewMonthString(12) As String

At the Load event of form write the following code. Declare the NewMonthString array at the general section of the form.

```

NewMonthString(0) = "Enero"
NewMonthString(1) = "Febrero"
NewMonthString(2) = "Margo"
NewMonthString(3) = "April"
NewMonthString(4) = "Mayo"
NewMonthString(5) = "Julio"
NewMonthString(6) = "Jimio"
NewMonthString(7) = "Agosto"
NewMonthString(8) = "Septiembre"
NewMonthString(9) = "Octbre"
    
```

Notes

```
NewMonthString(10) = "Noviembre"
NewMonthString(11) = "Diciembre"
```

This code will store all the Spanish names of the months to an array `NewMonthString`. To set the size of the format write the following code at `FormatSize` Event Procedure.

```
Private Sub DtPicker_FormatSize(ByVal CalbackField as String, Size As Integer)
    Dim MonthLen As Integer
    If callbackfield = "xxxx" then
        MonthLen = 0
        For x = 0 to 11
            If MonthLen < Len(NewMonthString(x)) then
                MonthLen = Len(NewMonthString(x))
            End if
        Next i
    End if
    Size = MonthLen
End Sub
```

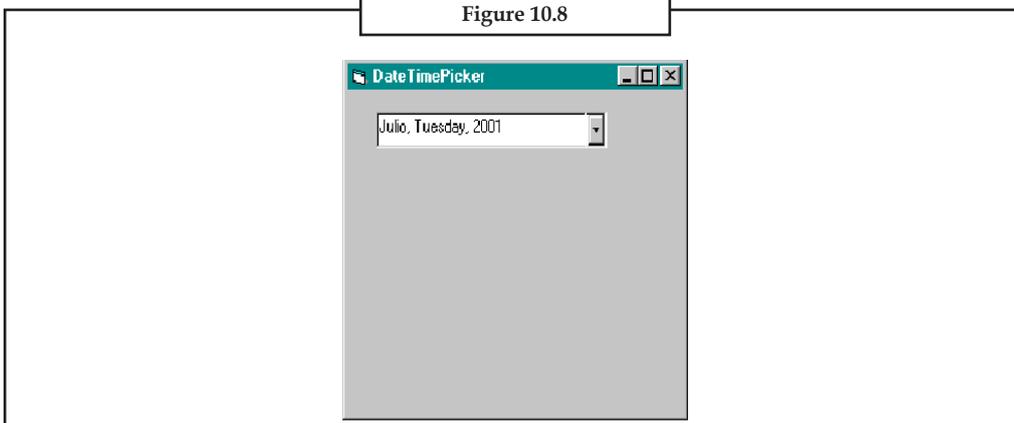
The above code fixes the size argument with new value which is the length of new string containing month. The value of size may vary depending upon the length of the month name. You have already declared one string type array of twelve elements in the general declaration section of your form module with the name `NewMonthString`.

Now the only code you need to write before running your application is for setting the new value of the month name. This can be done by writing the following code at `Format` event. The `Format` event has `callbackField` argument as string, `FormattedString` as string.

```
Private Sub DtPicker_Format(ByVal CallbackField as String, FormattedString as String)
    If callbackField = "xxxx" then
        FormattedString = NewMonthString(DtPicker1.Month)
    End if
End Sub
```

This code will store the currently selected month which is an integer returned by `DtPicker1.Month`.

Figure 10.8



Notes


Task Match the following

1. Timer	f. Change()
2. Specifies the change in the value property	g. Stretch
3. Returns the ASCII value of the key pressed	h. displays a list
4. Common in Scroll Bars and Text Box control	i. Smallchange
5. When user scrolls	j. Interval

Self Assessment

Fill in the blanks:

- 14. MinDate and MaxDate properties control the range of date that can be displayed, similar to the.....
- 15. Everytime when the new value is asked by the user, event is fired which has callbackField as string and size as integer arguments.


Caselet **Visual Basic .Net Tour**

Microsoft India Corporation Pvt Ltd said on Thursday that it has commenced a five-city Visual Basic .Net tour in India as a part of its world Tour.

The tour is aimed at familiarising developers and students with the advanced concepts for building applications that take full advantage of their existing Visual Basic .Net skills.

At the tour, Microsoft is previewing its upcoming Visual Basic .Net 2005, which is the next version of Visual Basic .Net. In addition, it is introducing Visual Basic .Net 2003 Power Pack - a set of controls released by Microsoft earlier this year, that provide enhanced user interface elements and enable developers to create more interesting client based applications.

10.7 Summary

- Using label controls in the applications and setting access keys for a Text Box using label controls.
- Using Text Box controls and capturing the key strokes on any control.
- Working with List Box and Combo Box controls. Adding and removing items to and from them.
- Using Option buttons and Check Boxes in the applications.
- Using Scroll Bars through an example.
- Using Timer Control and creating application with the help of a Timer Control.
- Adding pictures on the form using Image Control.

10.8 Keywords

Notes

CalendarTrailingForeColor: Specifies the colour of dates from previous or trailing months.

Controls: Controls are the basic foundation of any application providing Graphical User Interface. (GUI)

GUI: Graphical User Interface.

10.9 Review Questions

1. How can you make a label control work like a Text Box? Explain with an example.
2. How do you set an Access key for a Combo Box control?
3. Create an application that displays current system time and date. Add option buttons to select the format of the time and date to be displayed.
4. How can you create a scrollable form in Visual Basic?
5. Create an executable file of the "Flying Message" program after making changes to it so that it may work like a screen saver. **Hint:** You must add some code at form's KeyPress event and check mouse events for stopping the execution. Save the Exe in Windows System Directory with .scr extension and select it to from the Properties dialog box of Desktop.
6. WordWrap property of a Label control works only if AutoSize property is set to True. Do you agree with statement? Why or why not?
7. An Access key can be set for a List Box control by adding an '&' to a label control's caption property and setting its TabIndex property to one less than the TabIndex property of the List Box control. Do you agree with statement? Why or why not?
8. All controls have a caption property of the List Box control. Discuss.
9. A Timer control's Caption property is available at design time only. Do you agree with statement? Why or why not?
10. Cls property clears the contents of a List Box and a Combo Box. Explain.

Answers: Self Assessment

- | | |
|-----------------|-----------------------|
| 1. intrinsic | 2. string |
| 3. MultiLine | 4. Alignment |
| 5. Default | 6. ScrollBar |
| 7. drop down | 8. list |
| 9. display | 10. option button |
| 11. single | 12. Check Boxes |
| 13. LargeChange | 14. MonthView control |
| 15. FormatSize | |

Notes

10.10 Further Readings



Books

Database Access With Visual Basic 6, Jeffrey P. McManus, Sams

MCSD Training Guide - Visual Basic 6 Exams, Howard Hawhee, New Riders

Microsoft Visual Basic 6 Professional (step by step), Michael Halvorson, Microsoft Press

Programming Microsoft Visual Basic 6.0, Francesco Balena, Microsoft Press
Sams Teach Yourself Visual Basic 2005 in 24 Hours: complete starter kit, James D. Foxall, Sams Publishing

Visual Basic 6 from the Ground Up, Gary Cornell, Osborne McGraw Hill



Online links

http://www.vbtutor.net/vb2008/vb2008_lesson17.html

<http://www.vb6.us/tutorials/visual-basic-combo-box-tutorial>

Unit 11: Common Dialog Boxes

Notes

CONTENTS

Objectives

Introduction

11.1 Displaying Dialogs

11.2 Creating a Modal Dialog Box

11.3 The Message Box

11.4 Common Dialog Boxes

11.4.1 Types of Common Dialog Boxes

11.4.2 File Open and Save as Dialog Box

11.4.3 Color Dialog Box

11.4.4 Font Dialog Box

11.4.5 The Print Dialog Box

11.4.6 The InputBox

11.5 Dialog Result Class

11.6 Summary

11.7 Keywords

11.8 Review Questions

11.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the Common Dialog boxes
- Describe OpenFileDialog
- Demonstrate the SaveFileDialog
- Recognize ColorDialog
- Explain MessageBox
- Scan the DialogResult Class

Introduction

All the GUI applications keep prompting and talking to the users, using dialog boxes. Dialog boxes have been the only interface through which an application can talk to its users, and it will be so till Speech Oriented Softwares become feasible.

Notes

This unit covers creating dialog boxes, using them and creating Modal and Modeless dialog boxes. Windows operating system comes with default common dialog boxes that can be even used by other applications. This unit also introduces you with the MsgBox and Input statements for calling the default windows message box and input box. An endeavor to make you familiar with Windows Common Dialog Boxes, calling them and using them is also a part of this unit.

11.1 Displaying Dialogs

Dialog boxes are specifically used for accepting input from the user or displaying a message to the user. Windows operating system comes with default dialog boxes like Font dialog box, Printer dialog box, Color dialog box etc. Most of the applications, specially those vendored by Microsoft, borrow these dialog boxes from Windows whenever they need them. There are three kinds of dialog boxes

- Modal Dialog Box (Application Modal Dialog Box)
- Modeless Dialog Box
- System Modal Dialog Box

Modal Dialog Box

Modal dialog boxes can also be called as Application Modal Dialog Box. These dialog boxes insist that you to respond to them before continuing in the same application. A Modal dialog box, while in action, stops running your code until it is closed or hidden. Although you can call Default Dialog Box as a Modal Dialog Box, you can also create one of your own. The Show method of form uses VbModal style to load the form as a Modal Dialog Box. The general syntax you follow will be

```
FormName.Show VbModal
```

Besides Visual Basic's default Modal Message Box, you can create a Custom Modal Dialog Box, which we shall learn later.

Modeless Dialog Box

Modeless dialog boxes do not need to be closed to loose their focus. They are just like any other form in your application and loose their focus as soon as you click some other Window outside the application. A Find and Replace dialog box, Document Windows in Word application are few examples of Modeless dialog boxes.



Caution Form is loaded by default in this state if no arguments are supplied with Show method.

System Modal Dialog Box

A System Modal Dialog Box restricts the user from continuing work on the system unless it gets unloaded. Screen Saver with passwords is the pretty example of a Modal Dialog Box. Creating such a dialog box is a bit tricky, however you can have a default message box in System Modal mode. The show method of forms does not give you such style. In order to call a default message box in a system Modal style, use the following syntax

```
MsgBox "Message String", VbOKonly + VbSystemModal, "Title"
```

Self Assessment**Notes**

Fill in the blanks:

1. A....., while in action, stops running your code until it are closed or hidden.
2. Modeless dialog boxes do not need to be to loose their focus.
3. A System Modal Dialog Box the user from continuing work on the system unless it gets unloaded.
4. A Find and Replace dialog box, in Word application are few examples of Modeless dialog boxes.

11.2 Creating a Modal Dialog Box

As discussed above, Modal boxes are Application Modal dialog boxes and restrict the user from working on other forms in the application till they are on the screen. To create a modal a Model Dialog Box, follow these steps.

- Add a new standard EXE project.
- Add a new standard form. Now you have two standard forms.
- Place one Image box, one label control and two command buttons on the form named Form1 with Caption property set to "OK" and "Cancel" respectively.
- On Form2 place two command buttons with Caption property set to "Show Message" and "Exit" and with their default names.
- Now add the following code at click event of Command1 of Form2 to call the Modal Form.

```
Private Sub Command1_Click()
    Load Form1
    Form1.Image1.Picture = LoadPicture_
    ("C:\Program Files\Microsoft Visual Studio\COMMON\Graphics\Icons\" +_
    "Misc\Exclem.Ico")
    Form1.Caption = "Warning!"
    Form1.Show 1, Me
    Form1.label1.Caption = "This is Warning message!!"
End Sub
```

- Add the following code at click event of Command2 of Form2.

```
Private Sub Command2_Click()
    End
End Sub
```

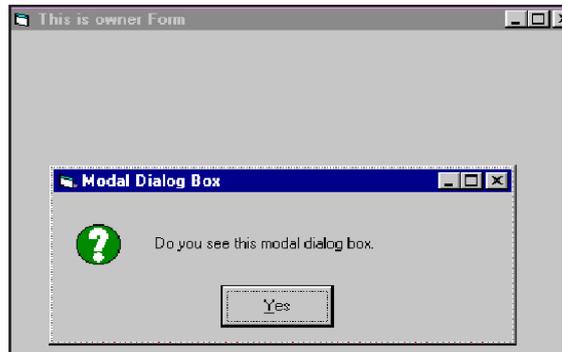
- Move to Form1 and add this code at click event of Command1

```
Private Sub Command1_Click()
    Unload Me
End Sub
```

Notes

- Set the Cancel Property of Command2 at Form2 to False and add the following code

```
Private Sub Command1_Click()  
End  
End Sub
```



- Save and run this application, it creates Form2 as the modal dialog box. The code line Form1.Show 1, Me. shows Form1 and loads it as Modal Dialog Box and Me keyword here specifies that the owner form for Form2 will be Form1. You cannot have a form as a Modal dialog in an MDI form.



Task Analyze practically what have you understood from the above mentioned steps for making the Model Dialog Box.

11.3 The Message Box

Message and Input dialog boxes are two default dialog boxes that you get readymade in Visual Basic to interact with the users while your application is running.

Message boxes are generally used to display warnings, messages or to accept user's choice. Message box can be called using MsgBox statements and MsgBox() function. MsgBox() function returns the user's response. Message box can be modal dialog box on a system modal dialog box. Message boxes can be called within your programs using the following code

```
MsgBox (Prompt [, buttons][,title][,helpfile, Context])
```

Each item inside brackets has been explained below.

Prompt

- Message that will be displayed within the Message box.
- It can be 1024 characters long.
- You can even display a multiline message. You can add a linefeed character to display the message in two lines.

Buttons

- Command buttons, icons that will be included in the message box.
- Each button and icon has its own Visual Basic constant.

- Default button is the OK button.
- The Buttons argument settings are

Constants	Value	Description
BUTTON CONSTANTS		
VbOKOnly	0	Display OK button only.
VbOKCancel	1	Display OK, Cancel buttons only.
VbAbortRetryIgnore	2	Display Abort, Retry and Ignore buttons only.
VbYesNoCancel	3	Display Yes, No and Cancel buttons only.
VbYesNo	4	Display Yes, No buttons only.
VbRetryCancel	5	Display Retry and Cancel Buttons only.
ICON CONSTANTS		
VbCritical	16	Display Critical message icon.
VbInformation	64	Display Information message icon.
VbExclamation	48	Display Warning message icon.
VbQuestion	32	Display Warning Query icon.
DEFAULT BUTTON CONSTANTS		
VbDefaultButton1	0	First Button is default.
VbDefaultButton2	256	Second Button is default.
VbDefaultButton3	512	Third Button is default.
VbDefaultButton4	768	Fourth Button is default.
BEHAVIOUR CONSTANTS		
VbApplicationModal	0	Insists the user to respond to the Message Box before continuing work in the current application.
VbSystemModal	4096	All the applications are suspended until the user responds to the Message Box.
VbMsgBoxHelpButton	16384	Add help button to the Message Box.
VbMsgBoxSetForeground	65536	Specifies the Message Box window as the foreground window.
VbMsgBoxRight	524288	Text is kept Right aligned.
VbMsgBoxRtlReading	1048576	If specified then text flows from right to left for Hebrew and Arabic system.



Example: MsgBox "This is an example", VbOKCancel, "Example"

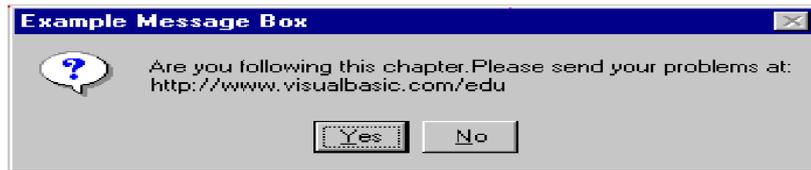
These constants can also be used in combination of more than one.



Example: ButtonConst = VbYesNo + VbQuestion + VbDefaultButton1

MsgBox "Are you following this session", ButtonConst, "Example Message Box"

Notes



Title

- The Caption that the message box will bear.

Helpfile and Context Number

To use help for an individual dialog box, include the path and the name of a windows compatible helpfile and an integer representing the help context you want to display.

Using the MsgBox Return Values

The message box returns an integer that can be stored in any integer type value. The specific return type is a `VbMsgBoxResult`, also an integer. Defined constants for the `VbMsgBoxResult` return values of the `MsgBox` function are

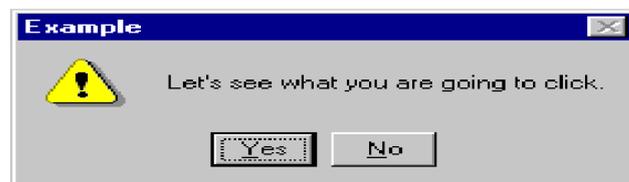
Constant	Value	Definition
<code>VbOK</code>	1	When user's response is OK.
<code>VbCancel</code>	2	When user's response is Cancel.
<code>VbAbort</code>	3	When user's response is Abort.
<code>VbRetry</code>	4	When user's response is Retry.
<code>VbIgnore</code>	5	When user's response is Ignore.
<code>VbYes</code>	6	When user's response is Yes.
<code>VbNo</code>	7	When user's response is No.



Example:

```
Private Sub Command1_Click
    Dim Message As String
    Buttons=VbExclamation + VbAbortRetryIgnore+VbDefaultButton1
    VbMsgBoxResult=MsgBox("Let's see what you're going to click",
    Buttons, "Example")

```



```
Select Case VbMsgBoxResult
    Case VbAbort
        Message = "Abort"

```

```

CaseVbRetry
    Message = "Retry"
CaseVbIgnore
    Message = "Ignore"
End Select
MsgBox "I Got it, you have clicked" + Message+"Button", VbInformation +
VbOK_,"Example"

```



Self Assessment

Fill in the blanks:

5. Message box can be called using MsgBox statements and function.
6. Message box can be modal dialog box on a modal dialog box.
7. The returns an integer that can be stored in any integer type value.
8. To use help for an dialog box, include the path and the name of a windows compatible helpfile and an integer representing the help context you want to display.

11.4 Common Dialog Boxes

The common dialog control provides an interface between Visual Basic and the routines in the Windows dynamic-link library. COMMDDL.DLL is an important file which should be present in your Windows System directory. It cannot be resized on the form and is not visible at run time.

11.4.1 Types of Common Dialog Boxes

There are five main types of common dialog boxes that can be called through Common Dialog Control.

1. File Open and Save As Dialog box
2. Color Dialog box
3. Font Dialog box
4. Print Dialog box
5. Help Dialog box

Help dialog box has not been discussed in this unit as working with help files is out of the scope of this book. File Open and Save As Dialog Box, Color Dialog Box, Font Dialog Box and Printer Dialog Box, all can be called using the same common dialog box control.



Notes You must set its preferably properties at run time.

Notes

11.4.2 File Open and Save as Dialog Box

This dialog box displays the list of drives and folders on your system. User can select and specify the filename from this dialog box. This box is used to get the path and file name from the user only. Opening and saving a file is done by the program itself.



Calling a File Open and Save as Dialog Box

- ShowOpen Calls a file dialog box with caption Open.
- ShowSave Calls a file dialog box with caption Save As.

Important Properties to be used in Calling a File Open and Save As Dialog Box

Properties	Description
DefaultExt	Sets the default extension for the files in the dialog box.
FileName	Returns the path and full name of the file selected.
FileTitle	Returns the file name only.
Filter	Filters the kind of file according to the extensions given. Suppose .BMP is given as argument to this property, the dialog box will display the BMP files only. Multiple extensions can be given using (pipe) symbol.
FilterIndex	Used to specify look and behaviour of the dialog box.
InitDir	Specifies the initial directory.
MaxFileSize	Sets the maximum size of the file name including all the path information.

Important Flags for the File Dialog Boxes

Flags	Characteristics
CdIOFNAllowMultiSelect	Used to allow the users multiple selection of files from the box.
CdIOFNCreatePrompt	Pops up a message box that inquires to create a new file, when a new name is given by the user in the file name.
CdIOFNFileMustExist	Pops up a message box if a user tries to enter a file or path and CdIOFNPathMustExist that doesn't exist.

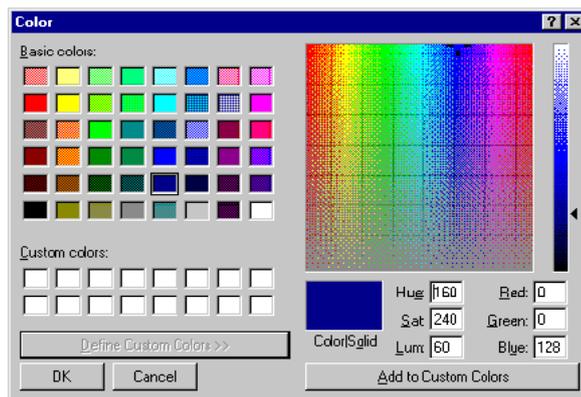
		Notes
CdIOFNHideReadOnly	The first flag hides the Read only check box in the dialog CdIOFNReadOnly box.	
	The second flag makes the Read only check box checked when the dialog first appears.	
CdIOFNOverviewPrompt	If you use ShowSave and the file already exists, Visual Basic will pop up a message box asking the user to confirm that he or she wants to overwrite the file.	
CdIOFNShareAware	If this flag is set, Windows will ignore sharing violation errors.	
CdIOFNHelpButton	Displays Help button in the dialog box.	

11.4.3 Color Dialog Box

The color dialog box is opened using the Open Color method of the common dialog box control. The following statement will open the color dialog box.

```
CommonDialog1.ShowColor
```

A typical color dialog box with custom colors option opened looks like the one shown below in figure.



The Color property returns the selected color. To set the initial color selection, set this Color property and set the Flags property to `cdICCRGBInit`. If you set the Flags property to `cdCCIFullOpen` the dialog box is opened with the custom colors selection as shown above in figure. If you set the Flags property to `cdCCPreventFullOpen` the user will be prevented from using the custom colors.



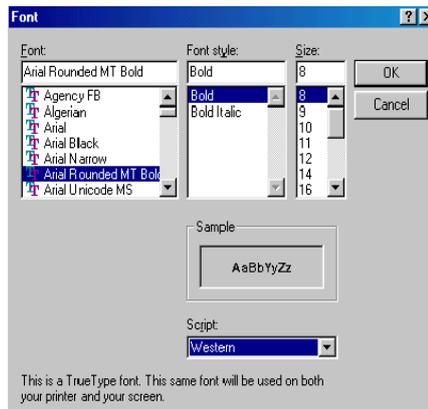
Example: `Form1.BackColor = CommonDialog1.Color`

This code line will change the Color of the form's background by the color selected by the user from a common dialog box.

11.4.4 Font Dialog Box

Shows the various fonts available on the system. Various options like Subscript, Superscript, Colors etc can be added to the Font Dialog box using Flag property.

Notes



Important Flags

Flag	What It Does
cdlCFHelpButton	Determines whether the dialog box displays a Help button.
cdlCFEffects	Determines whether you want to allow strikeout, underline and color effects.
cdlCFBoth	Shows both screen and printer fonts.
cdlCFApply	Determines whether the dialog box enables the Apply button.
cdlCFANSIOOnly	Determines whether the dialog box displays only the fonts that include the Windows character set.
cdlCFNoVectorFonts	Determines whether the dialog box should not display vector-based fonts.
cdlCFNoSimulations	Determines whether the dialog box will not allow graphic device interface (GDI) font simulations.
cdlCFLimitSize	Determines whether the dialog box should show only font sizes between those specified by the Max and Min properties.
cdlCFFixedPitchOnly	Determines whether the dialog box should display only fixed-pitch fonts.
cdlCFWYSIWYG	Determines whether the dialog box should show only fonts common to both the screen and printer.
cdlCFForceFontExit	Determines whether a message box pops up if a user selects a font style that doesn't exist.
cdlCFScalableOnly	Determines whether the dialog box will allow the user to only select scalable fonts such as True Type.
cdlCFTTOnly	Specifies that the dialog box should allow the user to select only True Type fonts.
cdlCFNoFaceSel	This is returned if no font name is selected.
cdlCFNoStyleSel	This is returned if no font style is selected.
cdlCFNoSizeSel	This is returned if no font size is selected.



Did u know? **What are basic properties of common dialog boxes?**

Common Dialog not only replaces three controls (Drive, Directory and FileList), but also is easier to program. It is supported in Visual Basic and VBA as well. The new VB.NET has the same functionality in the OpenFileDialog class.

Font Name property returns the selected font. Font Bold property returns whether Bold is selected. Font Italic property returns whether Italic is selected. Font Strikethru property returns whether strikethrough is selected. Font Underline property returns whether Underline is selected. All these properties are Boolean properties.

Some Important Properties Associated with Fonts

Property	Use
Color	Only used for color printers.
FontBold,FontItalic	True/False properties. If the cdLCFEffects flag is set,
FontStrikeThru	you can allow the user to choose these properties.
FontUnderline	
FontName	Sets or returns the font name.
FontSize	Sets or returns the font size.
Max, Min	These reflect the range of the point sizes shown in the size box. You need to have the cdLCFLimitSize flag set before you can use these properties.

11.4.5 The Print Dialog Box

A common dialog control can be used to show the Print Dialog Box. ShowPrinter is the method used to call Print Dialog Box. Flag's property controls the appearance and flexibility of the dialog box.

Important Properties

Property	Use
Copies	Sets or returns the number of copies the user wants.
FromPage, ToPage	Specifies what pages are wanted.
hDC	This is the device context number. It is used for API function calls.
Max, Min	Specifies the maximum and minimum pages the user can put in the Print Range frame.
PrinterDefault	Set this to True and the user can click the Setup button to change the WIN.INI file.

Important Flags

Flag	What It Does
cdlPDAllPages	Returns the value or sets the All Pages option button.
cdlPDCollate	Returns the value or sets the Collate check box.

Notes

Notes	cdlPDDisablePrintToFile	Disables the Print to File check box.
	cdlPDHidePrintToFile	Hides the Print to File check box.
	cdlPDNoPageNums	Returns the value or sets the page option button.
	cdlPDNoSelection	Disables the Selection option button.
	cdlPDNoWarning	Prevents Visual Basic from issuing a warning message when there is no default printer.
	cdlPDPageNums	Returns the value or sets the pages option button.
	cdlPDPrintSetup	Displays Print Setup dialog box rather than the Printer dialog box.
	cdlPDPrintToFile	Returns the value or sets the Print to the File check box.
	cdlPDReturnDC	Returns a device context for the printer selection from the hDC property of the dialog box.
	cdlPDReturn Default	Returns the default printer name.
	cdlPDRReturnIC	Returns an information context for the printer selection value from the hDC property of the dialog box.
	cdlPDSelection	Returns the value or sets the Selection option button.
	cdlPDHelpButton	Determines whether the dialog box displays the Help button.
	cdlPDUseDevModelCopies	Sets support for multiple copies.

11.4.6 The InputBox

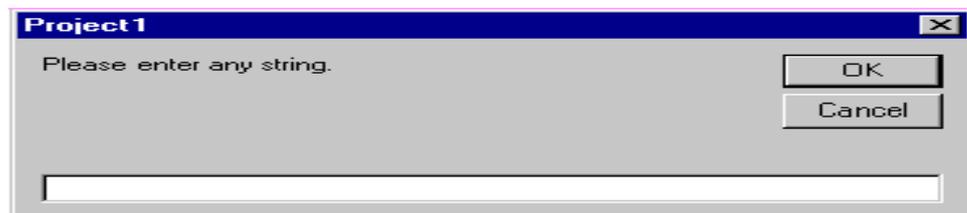
It is used to collect an input from the user. The syntax for the InputBox is

```
InputBox (Prompt[,title[,default][,xpos][,ypos][,helpfile, Context])
```



Example:

```
Private Sub Command1_Click
Dim NewText as string
NewText = InputBox ("Enter Any String")
Form1.Print NewText
End Sub
```



InputBox has only two default Buttons—OK and Cancel. HelpButton is included if the helpfile and context parameters are passed to it. XPos and YPos specify the position of the input box where it will appear, however it is movable.

11.5 Dialog Result Class

Notes

Defines constants that represent common results returned from dialog boxes.

Enum

```
|
+--DialogResult

package com.ms.wfc.ui

public class DialogResult
extends Enum
```

Remarks: If you assign a DialogResult value to the dialogResult property of a Buttoncontrol, clicking the button closes the parent form and sets the form.sdialogResult property to that value (if the form was displayed using theshowDialog method).

Self Assessment

Fill in the blanks:

9. is an important file which should be present in your Windows System directory.
10. is the method used to call Print Dialog Box.
11. has only two default Buttons-OK and Cancel.
12. specify the position of the input box where it will appear, however it is movable.
13. If you set the Flags property to the user will be prevented from using the custom colors.
14. CdIOFNHideReadOnly is the first flag hides the Read only in the dialog CdIOFNReadOnly box.
15. is used to specify look and behaviour of the dialog box.



Caselet

Academic Curriculum does not meet Industry Expectations

INDIAN universities churn out more than half-a-million engineers every year. Are all of them employable? No, because “academic curriculum does not match industry expectations,” said Mr N. Ram Subramani, Chairman and CEO of Maples ESM Technologies Ltd, Chennai.

He was talking to the students of computer applications and management in a seminar organised recently under the aegis of BL club by the MCA department of Madha Engineering College, Kandrathur, Chennai.

Mr Subramani listed the following as the broad categories of opportunities: PC computing application development (front end), enterprise computing, BPO, CAD/CAM and multimedia/animation.

Contd...

Notes

“By the time you study Unix & C, the requirement from the industry will move to C, C++. Similarly, by the time you complete, C, C++, the requirement will be VB, Oracle, then to VB, ASP, XMC and so on,” he said, listing the problems in PC computing application development.

He added that this technology is used predominantly in the small office home office (soho) or the lower mid-segment. It is highly overcrowded, offers low salary and no job stability.

Mr Subramani also drew the attention of the participants to the organisational issues in PC computing technology: It changes every six months, creates instability, organisations do not get any return on investment, change management is expensive, there is no scalability and frequent virus attacks hamper business.

Operating system, database and network are key to the health of any IT-driven organisation. They are like the brain, heart and nervous systems of the human beings. If any one of the above fails, it brings down the whole user community. Here, Enterprise Systems Management (ESM) plays an important role. ESM is a critical technology. It also helps in remote management of the organisations at different locations, said Mr Subramani.

On the trends in outsourcing from India, he said, “Post 9/11, only 20 per cent of the jobs are carried out on site, while the rest are outsourced; \$43/hour is the cost of labour for on-shore work whereas \$11/hour is the cost of work that is outsourced, resulting in 300 per cent profit to the outsourcing organisations.” Therefore, the opportunity for outsourcing from India is huge.

Mr Subramani said emphatically that BPO jobs are not for engineers/MCA and MBA degree-holders but for under-graduates. Americans or foreigners do not give CAD/CAM jobs to Indians, as intellectual property rights are not respected and the designs can be copied.

On the multimedia/animation front, he said Indians do not have the resources to invest in multimedia films like Hollywood does. Tom & Jerry’s 5-minute episodes cost \$5 million (₹ 25 crore). For this, nearly 45,000 frames are shot at kindergarten schools by doctoral degree holders in psychology.

In his final remarks, Mr Subramani said, “Even if you are Bill Gates, Indian companies will not take you if you are not good at communications. Here, technology comes after communications.”

Many graduates and post-graduates today are not “industry-ready”. Students have to take up additional training in IT skills as well as improve soft skills, he said.

Earlier, the college Principal, Dr R. Venkataswami, in his inaugural address, drew the students’ attention to the need for improving communication skills. He encouraged them to read business newspapers, as it would help them improve communication skills, besides imparting business knowledge.

The Head of the Department (MCA), Prof C. Joseph Callistus, and faculty from both departments were also present.

11.6 Summary

- Various modes of dialog boxes and difference between Modal and Modeless dialog box.
- Creating and using dialog boxes.

- Calling Windows Default Message Box and Input Box and using them in applications.
- Calling and using Windows default File Open, Save As, Font and Printer dialog box. Properties of these common dialog boxes and important flags that can be used to add the functionality and feasibility to Windows common dialog boxes while calling.

Notes

11.7 Keywords

InitDir: Specifies the initial directory.

MaxFileSize: Sets the maximum size of the file name including all the path information.

11.8 Review Questions

1. Modal dialog boxes can also be called as Application Modal Dialog Box. Explain.
2. Screen Saver with passwords is the pretty example of a Modal Dialog Box. Analyze.
3. Message boxes are generally used to display warnings, messages or to accept user's choice. Discuss.
4. Demonstrate the steps for making the Model Dialog Boxes. Explain the advantages and disadvantages Model Dialog Boxes.
5. MsgBox() function returns the user's response. Examine.
6. The color dialog box is opened using the Open Color method of the common dialog box control. Discuss.
7. Font Underline property returns whether Underline is selected. Explain.
8. A true type font is available both on the printer and the screen. Do you agree with statement? Why or why not? Give reasons to support your answer.
9. You can call File open dialog box that allows the user to select multiple files. Do you agree with statement? Why or why not? Give reasons to support your answer.
10. A file open dialog box does not display the system files. Do you agree with statement? Why or why not? Give reasons to support your answer.

Answers: Self Assessment

- | | |
|--------------------------|---------------------|
| 1. Modal dialog box | 2. closed |
| 3. restricts | 4. Document Windows |
| 5. MsgBox() | 6. System |
| 7. message box | 8. individual |
| 9. COMMdlg.DLL | 10. ShowPrinter |
| 11. InputBox | 12. XPos and YPos |
| 13. cdlCCPreventFullOpen | 14. check box |
| 15. FilterIndex | |

Notes

11.9 Further Readings



Books

Database Access With Visual Basic 6, Jeffrey P. McManus, Sams

MCSD Training Guide - Visual Basic 6 Exams, Howard Hawhee, New Riders

Microsoft Visual Basic 6 Professional (step by step), Michael Halvorson, Microsoft Press

Programming Microsoft Visual Basic 6.0, Francesco Balena, Microsoft Press

Sams Teach Yourself Visual Basic 2005 in 24 Hours: complete starter kit, James D. Foxall, Sams Publishing

Visual Basic 6 from the Ground Up, Gary Cornell, Osborne Mcgraw Hill



Online links

[http://msdn.microsoft.com/en-us/library/256tssz7\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/256tssz7(v=vs.90).aspx)

<http://vbadud.blogspot.com/2007/06/visual-basic-common-dialog.html>

Unit 12: File Input Output

Notes

CONTENTS

Objectives

Introduction

12.1 Files

12.2 Working with Files

12.2.1 Types of File Systems

12.2.2 Writing and Reading Files, Text Files and Binary Files

12.3 System IO

12.3.1 Input Device

12.3.2 Output Device

12.4 Summary

12.5 Keywords

12.6 Review Questions

12.7 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the working of Files and Directories
- Describe System IO

Introduction

Another part of the operating system is the file manager. While the memory manager is responsible for the maintenance of primary memory, the file manager is responsible for the maintenance of secondary storage (e.g., hard disks). Each file is a named collection of data stored in a device. The file manager implements this abstraction and provides directories for organizing files. It also provides a spectrum of commands to read and write the contents of a file, to set the file read/write position, to set and use the protection mechanism, to change the ownership, to list files in a directory, and to remove a file. The file manager provides a protection mechanism to allow machine users to administer how processes executing on behalf of different users can access the information in files. File protection is a fundamental property of files because it allows different people to store their information on a shared computer, with the confidence that the information can be kept confidential.



Notes In addition to these functions, the file manager also provides a logical way for users to organize files in secondary storage.

12.1 Files

A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc. Sometimes you may import a file from elsewhere, for example from another computer. If you want to enter your own text or data, you will start by creating a file. Whether you copied a file from elsewhere or created your own, you will need to return to it later in order to edit its contents.

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sector, generally 512 bytes each. The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used. Most file systems address data in fixed-sized units called "clusters" or "blocks" which contain a certain number of disk sectors (usually 1-64). This is the smallest logical amount of disk space that can be allocated to hold a file.

However, file systems need not make use of a storage device at all. A file system can be used to organize and represent access to any data, whether it be stored or dynamically generated (e.g. from a network connection).

Whether the file system has an underlying storage device or not, file systems typically have directories which associate file names with files, usually by connecting the file name to an index into a file allocation table of some sort, such as the FAT in an MS-DOS file system, or an inode in a Unix-like file system. Directory structures may be flat, or allow hierarchies where directories may contain subdirectories. In some file systems, file names are structured, with special syntax for filename extensions and version numbers. In others, file names are simple strings, and per-file metadata is stored elsewhere.

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as an exact byte count. The time that the file was last modified may be stored as the file's timestamp. Some file systems also store the file creation time, the time it was last accessed, and the time that the file's metadata was changed. (Note that many early PC operating systems did not keep track of file times.) Other information can include the file's device type (e.g., block, character, socket, subdirectory, etc.), its owner user-ID and group-ID, and its access permission settings (e.g., whether the file is read-only, executable, etc.).

The hierarchical file system was an early research interest of Dennis Ritchie of Unix fame; previous implementations were restricted to only a few levels, notably the IBM implementations, even of their early databases like IMS. After the success of Unix, Ritchie extended the file system concept to every object in his later operating system developments, such as Plan 9 and Inferno.



Did u know? **What are the facilities provided by the Traditional file systems?**

Traditional file systems offer facilities to create, move and delete both files and directories. They lack facilities to create additional links to a directory (hard links in Unix), rename parent links (".." in Unix-like OS), and create bidirectional links to files.

Traditional file systems also offer facilities to truncate, append to, create, move, delete and in-place modify files. They do not offer facilities to prepend to or truncate from the beginning of a file, let alone arbitrary insertion into or deletion from a file. The operations provided are highly asymmetric and lack the generality to be useful in unexpected contexts.



Example: Interprocess pipes in Unix have to be implemented outside of the file system because the pipes concept does not offer truncation from the beginning of files.

Secure access to basic file system operations can be based on a scheme of access control lists or capabilities. Research has shown access control lists to be difficult to secure properly, which is why research operating systems tend to use capabilities. Commercial file systems still use access control lists.

Self Assessment

Fill in the blanks:

1. The length of the data contained in a file may be stored as the number of allocated for the file or as an exact byte count.
2. Directory structures may be flat, or allow hierarchies where directories may contain.....
3. also offer facilities to truncate, append to, create, move, delete and in-place modify files.
4. Secure access to basic file system operations can be based on a scheme of lists or capabilities.
5. Research has shown access control lists to be difficult to properly, which is why research operating systems tend to use capabilities.

12.2 Working with Files

A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data.

More formally, a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.



Caution File systems share much in common with database technology, but it is debatable whether a file system can be classified as a special-purpose database (DBMS).

12.2.1 Types of File Systems

File system types can be classified into disk file systems, network file systems and special purpose file systems.

- **Disk file systems:** A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer. Examples of disk file systems include FAT, FAT32, NTFS, HFS and HFS+, ext2, ext3, ISO 9660, ODS-5, and UDF. Some disk file systems are journaling file systems or versioning file systems.
- **Flash file systems:** A flash file system is a file system designed for storing files on flash memory devices. These are becoming more prevalent as the number of mobile devices is increasing, and the capacity of flash memories catches up with hard drives.

While a block device layer can emulate a disk drive so that a disk file system can be used on a flash device, this is suboptimal for several reasons:

- ❖ **Erasing blocks:** Flash memory blocks have to be explicitly erased before they can be written to. The time taken to erase blocks can be significant, thus it is beneficial to erase unused blocks while the device is idle.

Notes

- ❖ *Random access:* Disk file systems are optimized to avoid disk seeks whenever possible, due to the high cost of seeking. Flash memory devices impose no seek latency.
- ❖ *Wear leveling:* Flash memory devices tend to wear out when a single block is repeatedly overwritten; flash file systems are designed to spread out writes evenly.

Log-structured file systems have all the desirable properties for a flash file system. Such file systems include JFFS2 and YAFFS.

- **Database file systems:** A new concept for file management is the concept of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar metadata. Example: dbfs.
- **Transactional file systems:** Each disk operation may involve changes to a number of different files and disk structures. In many cases, these changes are related, meaning that it is important that they all be executed at the same time. Take for example a bank sending another bank some money electronically. The bank's computer will "send" the transfer instruction to the other bank and also update its own records to indicate the transfer has occurred. If for some reason the computer crashes before it has had a chance to update its own records, then on reset, there will be no record of the transfer but the bank will be missing some money.

Transaction processing introduces the guarantee that at any point while it is running, a transaction can either be finished completely or reverted completely (though not necessarily both at any given point). This means that if there is a crash or power failure, after recovery, the stored state will be consistent. (Either the money will be transferred or it will not be transferred, but it won't ever go missing "in transit".)

This type of file system is designed to be fault tolerant, but may incur additional overhead to do so.



Task Journaling file systems are one technique used to introduce transaction-level consistency to file system structures. Analyze.

- **Network file systems:** A network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server.



Example: Network file systems include clients for the NFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

- **Special purpose file systems:** A special purpose file system is basically any file system that is not a disk file system or network file system. This includes systems where the files are arranged dynamically by software, intended for such purposes as communication between computer processes or temporary file space.

Special purpose file systems are most commonly used by file-centric operating systems such as Unix. Examples include the procfs (/proc) file system used by some Unix variants, which grants access to information about processes and other operating system features.

Deep space science exploration craft, like Voyager I & II used digital tape based special file systems. Most modern space exploration craft like Cassini-Huygens used Real-time operating system file systems or RTOS influenced file systems. The Mars Rovers are one such example of an RTOS file system, important in this case because they are implemented in flash memory.

- **Flat file systems:** In a flat file system, there are no subdirectories-everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc. While simple, this system rapidly becomes inefficient as the number of files grows, and makes it difficult for users to organise data into related groups. Like many small systems before it, the original Apple Macintosh featured a flat file system, called Macintosh File System. Its version of Mac OS was unusual in that the file management software (Macintosh Finder) created the illusion of a partially hierarchical filing system on top of MFS. This structure meant that every file on a disk had to have a unique name, even if it appeared to be in a separate folder. MFS was quickly replaced with Hierarchical File System, which supported real directories.

12.2.2 Writing and Reading Files, Text Files and Binary Files

You can read from (or write to) a text file using the Open statement, Close statement and various other statements for actually reading/writing.

The Open statement has clauses for what type of access you want to the file, so you can specify if you want to read it as text, write to it as binary, and so on. You can also set locking options, to stop other programs from opening the file. For more details, go to the Index of VB help and see "Open statement".

For all of these file access functions you need to specify a file number, which most people use 1 for. You should always use a number which is not already in use (as using the same will cause an error), luckily the next free number can be found using the FreeFile function as in the example below.

An example of reading a file:

```
Dim sFileText as String
Dim iFileNo as Integer
iFileNo = FreeFile
'open the file for reading
Open "C:\Test.txt" For Input As #iFileNo
'change this filename to an existing file! (or run the example below first)
'read the file until we reach the end
Do While Not EOF(iFileNo)
Input #iFileNo, sFileText
'show the text (you will probably want to replace this line as appropriate to
your program!)
MsgBox sFileText
Loop
'close the file (if you dont do this, you wont be able to open it again!)
Close #iFileNo
```

(note: an alternative to Input # is Line Input # , which reads whole lines).

An example of writing a file:

```
Dim sFileText as String
Dim iFileNo as Integer
iFileNo = FreeFile
'open the file for writing
Open "C:\Test.txt" For Output As #iFileNo
```

Notes

```
'please note, if this file already exists it will be overwritten!  
'write some example text to the file  
Print #iFileNo, "first line of text"  
Print #iFileNo, " second line of text"  
Print #iFileNo, "" `blank line  
Print #iFileNo, "some more text!"  
'close the file (if you dont do this, you wont be able to open it again!)  
Close #iFileNo
```

(Note: an alternative to Print # is Write # , which adds commas between items, and also puts the "character around strings).

You can read files in a much faster way too, by simply loading the entire file at once. To do this you need to use binary mode, like so:

Function FileText(ByVal filename As String) As String

Dim handle As Integer

'ensure that the file exists

```
If Len(Dir$(filename)) = 0 Then  
Err.Raise 53 ` File not found  
End If
```

' open in binary mode

handle = FreeFile

Open filename\$ For Binary As #handle

' read the string and close the file

FileText = Space\$(LOF(handle))

Get #handle, , FileText

Close #handle

End Function

Reading/ writing files can also be achieved using FSO (FileSystemObject) as shown below. Please note that using FSO requires an extra reference (under "Project" >"References"), and therefore extra files to be installed with your application.

Option Explicit

'Set a reference to "Microsoft Scripting Runtime"

Private Sub Command1_Click()

'declare and initiate required objects

Dim fs As FileSystem Object

Dim ts As TextStream

Set fs = New FileSystemObject

'To write Set ts = fs.OpenTextFile("C:\mytestfile.txt", ForWriting, True)

ts.WriteLine "I Love"

ts.WriteLine "VB Forums"

ts.Close

'To Read

If fs.FileExists("C:\mytestfile.txt") Then

Set ts = fs.OpenTextFile("C:\mytestfile.txt")

```

Do While Not ts.AtEndOfStream
MsgBox ts.ReadLine
Loop
ts.Close
End If
'clear memory used by FSO objects
Set ts = Nothing
Set fs = Nothing
End Sub

```

To be able to open a file and read the data from a storage unit of a computer, such as a hard drive and able to save the data into the storage unit are important functions of a computer program. In fact, the ability to store, retrieve and modify data makes a computer a powerful tool in database management.

In this unit, we will learn how to manage data that is stored as a text file. Using text file is an easy way to manage data, although it is not as sophisticated as full fledged database management software such as SQL Server, Microsoft Access and Oracle. Visual Basic 2008 allows the user to create a text file, save the text file as well as read the text file. It is relatively easy to write code for the above purposes in VB2008 compared to VB6.

Reading and writing to a text file in VB2008 required the use of the StreamReader class and the StreamWriter class respectively. StreamReader is a tool that enables the streaming of data by moving it from one location to another so that it can be read by the user. For example, it allows the user to read a text file that is stored in a hard drive. On the other hand, the StreamWriter class is a tool that can write data input by the user to a storage device such as the hard drive.

Reading a Text File

In order to read a file from the hard disk or any storage device, we need to use the StreamReader class. To achieve that, first of all we need to include the following statement in the program code:

```
Imports System.IO
```

This line has to precede the whole program code as it is higher in hierarchy than the StreamReader Class. In Fact, this is the concept of object oriented programming where StreamReader is part of the namespace System.IO . It has to be put on top of the whole program(i.e. above the Public Class Form 1 statement). The word import means we import the namespace System.IO into the program. Once we have done that , we can declare a variable of the StreamReader data type with the following statement:

```
Dim FileReader As StreamReader
```

If we don't include the Imports System.IO, we have to use the statement

```
Dim FileReader As IO.StreamReader
```

each time we want to use the StreamReader class.

Now, start a new project and name it in whatever name you wish. Now, insert the OpenFileDialog control into the form because we will use it to read the file from the storage device. The default name of the OpenFileDialog control is OpenFileDialog1, you can use this name or you can rename it with a more meaningful name. The OpenFileDialog control will return a DialogResult value which can determine whether the user clicks the OK button or Cancel button . We will also insert a command button and change its displayed text to 'Open'. It will be used by the user to open and read a certain text file. The following statement will accomplish the task above.

```
Dim results As DialogResult
```

```
results = OpenFileDialog1.ShowDialog
```

Notes

```
If results = DialogResult.OK Then
'Code to be executed if OK button was clicked
Else
'Code to be executed if Cancel button was clicked
End If
End Sub
```

Next, we insert a textbox and set its Multiline property to true. It is used for displaying the text from a text file. In order to read the text file, we need to create a new instant of the StreamReader and connect it to a text file with the following statement:

```
FileReader = New StreamReader(OpenFileDialog1.FileName)
```

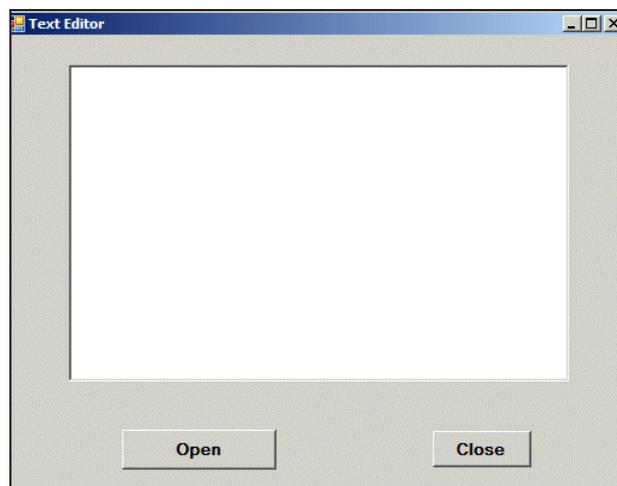
In addition, we need to use the ReadToEnd method to read the entire text of a text file. The syntax is:

```
TextBox1.Text = FileReader.ReadToEnd()
```

Lastly, we need to close the file by using the Close() method. The entire code is shown in the box below:

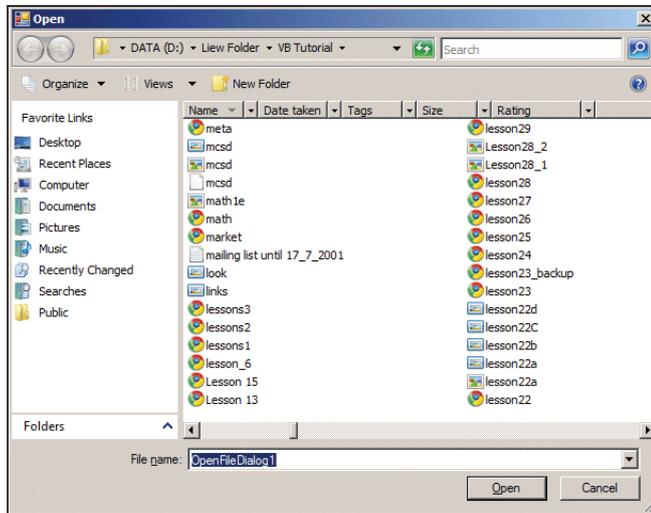
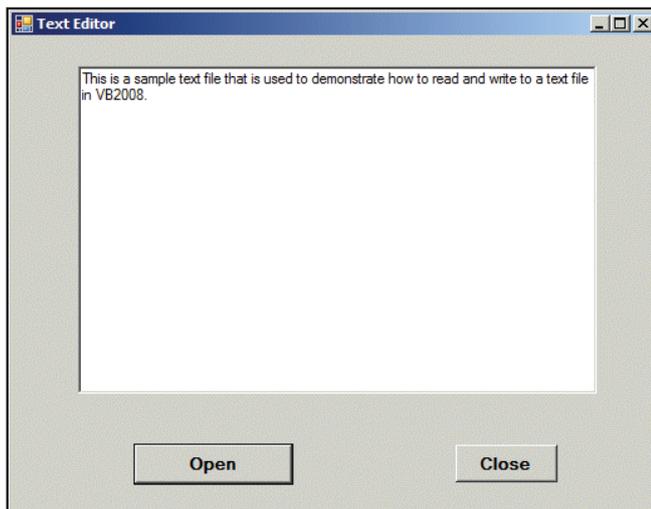
```
The Code
Imports System.IO
Public Class Form1
Private Sub BtnOpen_Click(ByVal sender As System.Object, ByVal eAs System.
EventArgs) Handles BtnOpen.Click
Dim FileReader As StreamReader
Dim results As DialogResult
results = OpenFileDialog1.ShowDialog
If results = DialogResult.OK Then
FileReader = New StreamReader(OpenFileDialog1.FileName)
TextBox1.Text = FileReader.ReadToEnd()
FileReader.Close()
End If
End Sub
```

The Design Interface



The Open Dialog box

Notes

*The Output Interface**Writing to a Text File*

Writing a text file means storing the text entered by the user via a textbox into a storage device such as a hard drive. It also means saving the file. To accomplish this task, we need to deploy the StreamWriter Class. You also need to insert the SaveFileDialog control into the form as it is used to save the data into the storage unit like a hard drive. The default name for the SaveFileDialog control is SaveFileDialog1. The Code is basically the same as the code for reading the file, you just change the StreamReader to StreamWriter, and the method from ReadToEnd to Write. The code is shown as following:

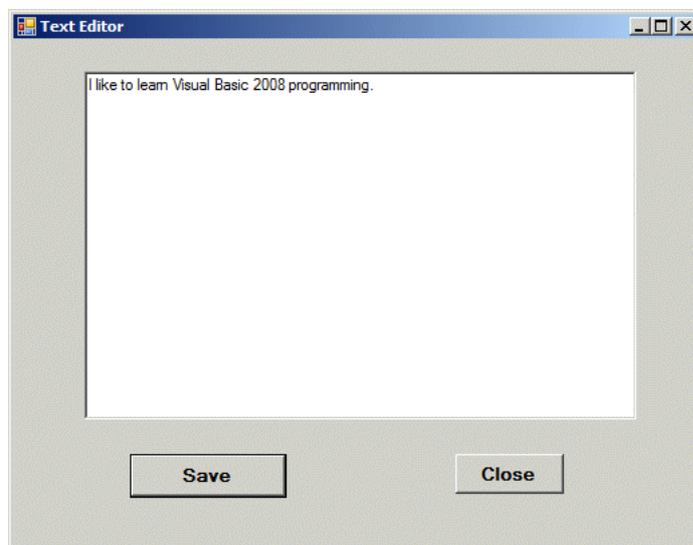
The code

```
Imports System.IO
Public Class Form1
Private Sub BtnSave_Click(ByVal sender As System.Object, ByVal eAs System.
EventArgs)
```

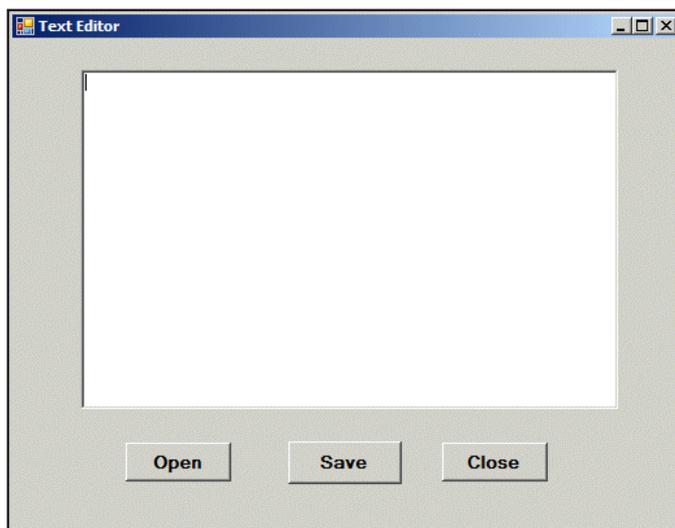
Notes

```
Dim FileWriTer As StreamWriter
Dim results As DialogResult
results = SaveFileDialog1.ShowDialog
If results = DialogResult.OK Then
FileWriTer = New StreamWriter(SaveFileDialog1.FileName, False)
FileWriTer.Write(TextBox1.Text)
FileWriTer.Close()
End If
End Sub
```

The Output Interface



When you click the save button, the program will prompt you to key in a file name and the text will be save as a text file. Finally, you can combine the two programs together and create a text editor that can read and write text file, as shown in the diagram below.



Binary Files

Notes

The nitty-gritty of writing a Binary file is that you can put the data into the file any way you want. When a binary file is opened the file pointer is positioned at byte 1. In other words it is at the beginning of the file.

You can write as much as you want into the file while it is open. After each write operation is complete, the file pointer is positioned at the byte immediately after the last data that was written.



Example: If you open the file and write 10 bytes of data, the data begins at byte 1 and ends at byte 10. The pointer is now at byte 11. You can change the pointer in code but that's where it will be if you do nothing. The same thing occurs when reading the file. If you close the file with no further writes to it then the subsequent file will be 10 bytes plus the size of a terminator for each piece of data you stored.

One thing that makes Binary access superior to Random access files is that the string fields can be any length at all. Strings are Visual Basic's largest data type not including objects or User Defined Types. They will quickly eat up memory and disk space, so it's nice to be able to store only as many characters as necessary.

For example, a Random access file must be written using fixed-length records as follows:

```
Option Explicit
Private Type tType
ID As Long
Name As String * 25
Address As String * 50
City As String * 25
State As String * 2
ZIP As String * 10
End Type
```



Notes Note that each record will require the full storage size defined for the string fields.

For Binary access we don't need to define the length of the string. If the string is 0 bytes, then 0 bytes get stored (plus some overhead to define the field as a string). The overhead also is used in Random access files, so on that level the files are the same.

```
Option Explicit
Private Type tType
ID As Long
Name As String
Address As String
City As String
State As String
ZIP As String
End Type
```

Now we're not only saving space, but a string field can be longer if needed. So we have two advantages because we are no longer limited by the length of strings.

Notes

Self Assessment

Fill in the blanks:

- 6. A file system is a set of data types that are implemented for the storage.
- 7. A file system is a file system designed for storing files on flash memory devices.
- 8. Special purpose file systems are most commonly used by operating systems such as Unix.
- 9. A network file system is a file system that acts as a client for a remote file access....., providing access to files on a server.

12.3 System IO

Input/output, or I/O, refers to the communication between an information processing system (such as a computer), and the outside world - possibly a human, or another information processing system. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. The term can also be used as part of an action; to “perform I/O” is to perform an input or output operation. I/O devices are used by a person (or other system) to communicate with a computer. For instance, keyboards and mice are considered input devices of a computer, while monitors and printers are considered output devices of a computer. Devices for communication between computers, such as modems and network cards, typically serve for both input and output.

Note that the designation of a device as either input or output depends on the perspective. Mice and keyboards take as input physical movement that the human user outputs and convert it into signals that a computer can understand. The output from these devices is input for the computer. Similarly, printers and monitors take as input signals that a computer outputs. They then convert these signals into representations that human users can see or read. (For a human user the process of reading or seeing these representations is receiving input.)

In computer architecture, the combination of the CPU and main memory (i.e. memory that the CPU can read and write to directly, with individual instructions) is considered the heart of a computer, and from that point of view any transfer of information from or to that combination.



Example: To or from a disk drive, is considered I/O.

The CPU and its supporting circuitry provide I/O methods that are used in low-level computer programming in the implementation of device drivers.

Higher-level operating system and programming facilities employ separate, more abstract I/O concepts and primitives.



Example: Most operating systems provide application programs with the concept of files.

The C and C++ programming languages, and operating systems in the Unix family, traditionally abstract files and devices as streams, which can be read or written, or sometimes both. The C standard library provides functions for manipulating streams for input and output.

I/O devices allow your managed system to gather, store, and transmit data. I/O devices are found in the server unit itself and in expansion units and towers that are attached to the server. I/O devices can be embedded into the unit, or they can be installed into physical slots.

Not all types of I/O devices are supported for all operating systems or on all server models.

Notes

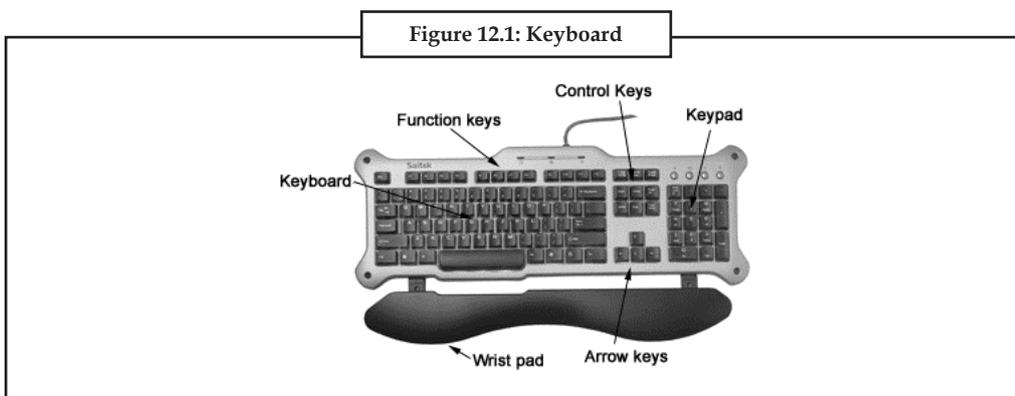


Example: Switch Network Interface (SNI) adapters are supported only on certain server models, and are not supported for i5/OS® logical partitions.

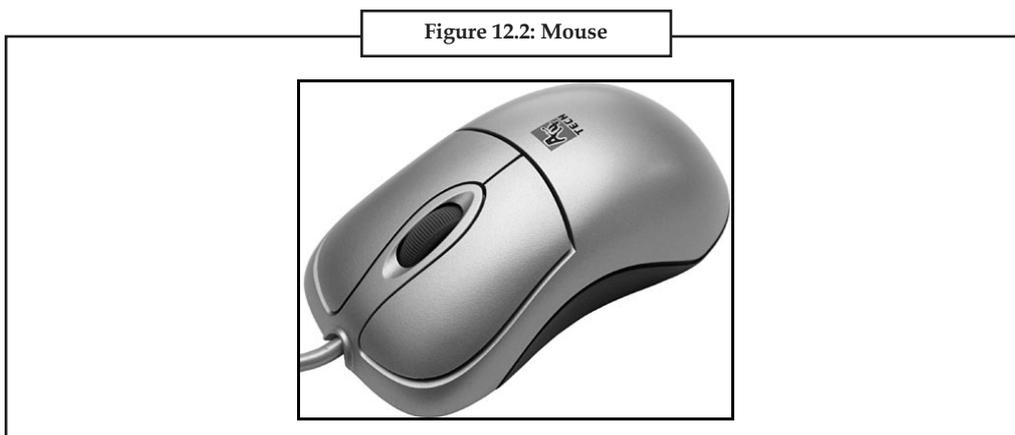
12.3.1 Input Device

A hardware device that sends information into the CPU is known as input device. Without any input devices a computer would simply be a display device and not allow users to interact with it, much like a TV. Below is a listing of different types of computer input devices.

Keyboard: One of the main input devices used on a computer, a PC's keyboard looks very similar to the keyboards of electric typewriters, with some additional keys.



Mouse: An input device that allows an individual to control a mouse pointer in a Graphical User Interface (GUI). Utilizing a mouse a user has the ability to perform various functions such as opening a program or file and does not require the user to memorize commands, like those used in a text-based environment such as MS-DOS. To the right is a picture of a Microsoft IntelliMouse and is an example of what a mouse may look like.



Scanner: Hardware input device that allows a user to take an image and/or text and convert it into a digital file, allowing the computer to read and/or display the scanned object. A scanner is commonly connected to a computer USB, Firewire, Parallel or SCSI port.

Notes

Figure 12.3: Scanner



Microphone: Sometimes abbreviated as mic, a microphone is a hardware peripheral that allows computer users to input audio into their computers.

Figure 12.4: Microphone



Web Cam: A camera connected to a computer or server that allows anyone connected to the Internet to view still pictures or motion video of a user. The majority of webcam web sites are still pictures that are frequently refreshed every few seconds, minutes, hours, or days. However, there are some sites and personal pages that can supply streaming video for users with broadband.

Figure 12.5: Web Cam



Digital camera: A type of camera that stores the pictures or video it takes in electronic format instead of to film. There are several features that make digital cameras a popular choice when compared to film cameras. First, the feature often enjoyed the most is the LCD display on the

digital camera. This display allows users to view photos or video after the picture or video has been taken, which means if you take a picture and don't like the results, you can delete it; or if you do like the picture, you can easily show it to other people. Another nice feature with digital cameras is the ability to take dozens, sometimes hundreds of different pictures.



Task Analyze practically the advantages of microphones.

Figure 12.6: Digital Camera



Joystick: A computer joystick allows an individual to easily navigate an object in a game such as navigating a plane in a flight simulator.

Figure 12.7: Joystick



12.3.2 Output Device

Any peripheral that receives and/or displays output from a computer is known as output device. Below are some examples of different types of output devices commonly found on a computer.

Monitor: Also called a Video Display Terminal (VDT) a monitor is a video display screen and the hard shell that holds it.



Notes In its most common usage, monitor refers only to devices that contain no electronic equipment other than what is essentially needed to display and adjust the characteristics of an image.

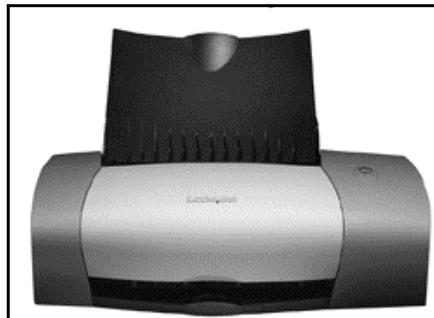
Notes

Figure 12.8: Monitor



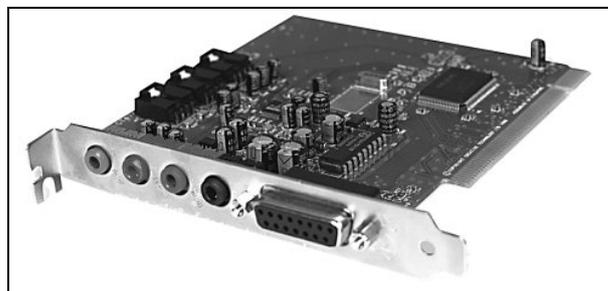
Printer: An external hardware device responsible for taking computer data and generating a hard copy of that data. Printers are one of the most used peripherals on computers and are commonly used to print text, images, and/or photos.

Figure 12.9: Printer



Sound card: Also known as a sound board or an audio card, a sound card is an expansion card or integrated circuit that provides a computer with the ability to produce sound that can be heard by the user.

Figure 12.10: Sound Card



Speakers: A hardware device connected to a computer's sound card that outputs sounds generated by the card.

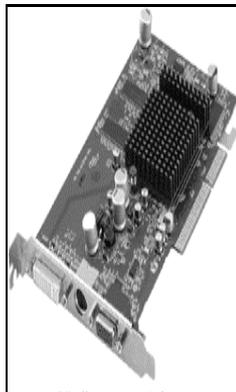
Notes

Figure 12.11: Speaker



Video card: Also known as a graphics card, video card, video board, or a video controller, a video adapter is an internal circuit board that allows a display device, such as a monitor, to display images from the computer.

Figure 12.12: Video Card



Self Assessment

Fill in the blanks:

10. A computer joystick allows an individual to easily navigate an object in a.....
11. A scanner is commonly connected to a computer..... , Firewire, Parallel or SCSI port.

Notes



Caselet

Split in Peace

Large, unwieldy files can often leave you splitting hairs with the sender, but help is at hand. This week's collection of software helps you split files into manageable chunks.

Dariolius

Dariolius allows you to split any file into smaller parts, so that they can be fitted on multiple disks or you can send them by e-mail. You can choose to later recombine the files using either a DOS batch file or a merger program, depending on your needs.

Dariolius offers you several options when splitting files. You can either specify the exact size of each part or simply tell the program how many parts you want the file to be split in. If you intend to send the files over the Internet, the program can also calculate the best size, based on the transfer speed. The interface is also easy to use.

Filesize: 1473 KB

Platform: Windows 9x/NT/ME/2K

Download location: www.kanastacorp.com

Slice-n-Save

Slice-n-Save is an easy-to-use tool that allows you to split large files into smaller chunks, which can be stored on floppies or sent by e-mail. The program automatically creates a .bat file that can be used to re-join the split file by simply clicking on it.

You can choose to delete the original file after the splitting process finishes, automatically clean up split files after a joint operation and more. Slice-n-Save also supports slicing of a file into variable sizes as well as easy drag- and-drop and Windows Explorer shell integration.

Filesize: 475 KB

Platform: Windows (All)

Download location: www.rtsoftware.org

JASP

JASP is an easy-to-use file splitting program that offers Windows Shell Integration and drag-and-drop. You can select a file to be split into smaller parts from within the interface or simply use the right-click option available in Windows Explorer.

The program automatically creates a .bat file that can be used to re-join the split parts. Split sizes are available in several pre-defined units and can also be customised.

Filesize: 389 KB

Platform: Windows (All)

Download location: www.codebytes.net

Banana Builder XP

Banana Builder XP is a multi-part file joiner and splitter that can be used to join or split files downloaded from newsgroups or the Internet. Additional features include sequence checking, update checking, custom colours and more.

Filesize: 1283 KB

Platform: Windows (All)

Contd...

Download location: www.manorsoft.com

Splitter & Merger

Splitter & Merger is a tool to split large files into smaller and easy-to-handle pieces. It offers additional file compression, disk spanning, path creation, automatic part deletion and more.

Filesize: 26 KB

Platform: Windows (All)

Download location: www.splitter.vze.com

GSplit

GSplit is a powerful file splitter that lets you split large files into a set of smaller files called pieces. These pieces are easier to copy to floppies, distribute over the Internet or through e-mail, share with friends or colleagues, and archive to zip disks or CDs. The pieces can be easily combined, using the generated Self-Uniting Executable that automatically restores the original file for you without requiring GUnite. GSplit also includes advanced features such as different splitting methods (blocked or spanned disks pieces), CRC32 checks (detects file corruption), splitting logs, and keeping file information.

Filesize: 1176 KB

Platform: Windows (All)

Download location: www.gdgsoft.com

Splitter

Using Splitter, you can split large files and directories (including subdirectories) into parts of 1.44 MB or custom sizes. It is useful to either fit the data on a floppy disk or transfer it by e-mail. To rejoin the data, you need to run Splitter again and select the files to be reunited, for which it should be included on the floppy disk. Splitter will leave sufficient room calculating its size to make it fit on the floppy.

Filesize: 35 KB

Platform: Windows (All)

Download location:

<http://martinstoeckli.gmxhome.de/splitter/splitter.html>

File Shredder

File Shredder is a program to split a large file into smaller files. You can specify either how many pieces to split the file, or how big (in KB) to split the file. There is also the option of filling the entire disks, which is convenient if you would like to split a file into multiple floppy disks. When File Shredder splits a file, it automatically generates a DOS batch file that will be used to combine smaller files into one big file.

Filesize: 654 KB

Platform: Windows 9x/NT/ME/2K

Download location: www.zdev.pair.com

Notes

12.4 Summary

- A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc.
- Sometimes you may import a file from elsewhere, for example from another computer. If you want to enter your own text or data, you will start by creating a file.
- A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. Disks provide the bulk of secondary storage on which a file system is maintained.
- To improve I/O efficiency, I/O transfer between memory and disk are performed in units of blocks.
- Each block is one or more sectors. Depending on the disk drive, sectors vary from 32 bytes to 4096 bytes; usually, they are 512 bytes.
- The file system provides the mechanism for online storage and access to both data and programs.
- The file system resides permanently on secondary storage, which has the main requirement that it must be able to hold a large amount of data, permanently.
- In a multi-user environment, a file is required to be shared among more than one users. There are several techniques and approaches to affect this operation.
- In almost every case, many files will be stored on the same disk. Security policies specify what is desired in terms of protection and security.
- Security mechanisms specify how to affect the security policies and enforce them in a given system.

12.5 Keywords

Access Hierarchy: A simple form of access hierarchy is provided by the dual, user/supervisor, mode of operation found in many computer systems.

Access Lists: Access lists are one way of recording access rights in a computer system.

Complete Mediation: Every access request for every object should be checked for authorization. The checking mechanism should be efficient because it has a profound influence on system performance.

Disk File Systems: A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.

Fail-safe Default: Access rights should be acquired by explicit permission only, and the default should be lack of access.

Flash File Systems: A flash file system is a file system designed for storing files on flash memory devices.

12.6 Review Questions

1. A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc. Substantiate.

2. File systems may use a data storage device such as a hard disk or CD-ROM. Explain.
3. A disk file system is a file system designed for the storage of files on a data storage device. Do you agree with statement? Why or why not? Give reasons to support your answer.
4. Demonstrate how disk file systems are optimized to avoid disk seeks whenever possible, due to the high cost of seeking?
5. Flash memory devices impose no seek latency. Explain with reasons.
6. The OpenFileDialog control will return a DialogResult value. Explain.
7. The word import means we import the namespace System.IO into the program. Explain with an example.
8. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Discuss.
9. Examine how an acyclic-graph structure is much easier to work with?
10. Explain how the deletion of a link does not need to affect the original file?

Notes

Answers: Self Assessment

- | | |
|-----------------------------|-------------------|
| 1. blocks | 2. subdirectories |
| 3. Traditional file systems | 4. access control |
| 5. secure | 6. abstract |
| 7. flash | 8. file-centric |
| 9. protocol | 10. game |
| 11. USB | |

12.7 Further Readings



Books

Andrew M. Lister, *Fundamentals of Operating Systems*, Published By WileyAndrew S. Tanenbaum, *Modern Operating System*, Published By Prentice HallColin Ritchie, *Operating Systems*, Published By BPB PublicationsSilberschatz Galvin, *Operating System Concepts*, Published By Addison Wesley

Online links

http://en.wikipedia.org/wiki/File_system<http://www.freebsd.org/doc/handbook/dirstructure.html>

Unit 13: ADO.Net

CONTENTS

Objectives

Introduction

13.1 Why ADO.NET?

13.1.1 A Brief History

13.1.2 ADO.NET isn't a New Version of ADO

13.2 ADO.NET and the .NET Framework

13.3 Accessing Database with ADO.Net

13.4 Binding Controls to Recordsets

13.4.1 Adding a Record to a Recordset

13.4.2 Updating a Record in a Recordset

13.4.3 Deleting a Record in a Recordset

13.4.4 Adding a Record to a Recordset using SQL Statements

13.4.5 Updating a Record in a Recordset using SQL Statements

13.4.6 Refreshing Data in a Recordset

13.4.7 Executing

13.5 Select Command with Database

13.6 Summary

13.7 Keywords

13.8 Review Questions

13.9 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan the accessing database with ADO.Net
- Describe E Executing Insertion, deletion, updating and select command with databases.

Introduction

VB.NET introduces new tools for connecting to databases. Some of these tools are controls that act as wrappers around the database access objects, called ADO.NET. These controls simplify the code you write, and allow you to bind graphical controls to DataSets you generate from the database. These controls require a new way of working with databound controls in your application.

The ADO.NET functionality is different from what you are used to with ADO. Although similar in concept, the ADO.NET objects follow a disconnected paradigm. One of the major differences is that you are not necessarily retrieving the equivalent of a recordset, but an entire schema structure.

13.1 Why ADO.NET?

Almost all applications require some data access. Before the .NET Framework, developers used data access technologies such as ODBC, OLE DB, and ADO in their applications; with the introduction of .NET, Microsoft created a new way to work with data, called ADO.NET. This is the only technology you need to use in .NET to access data sources, but in reality several of its predecessors are still lurking under the hood. We'll start this unit by telling you what they are and how they fit into the grand scheme of things.

13.1.1 A Brief History

Back at the end of the 1980s, several big database servers and a few small ones were available to buy and use. Each used SQL (with one or two of their own proprietary extensions thrown in for good measure) to access and manipulate the data they contained, and each had their own proprietary set of programming interfaces for developers to tie their own applications into the servers. This forced a great deal of extra work onto developers, as in order to support more than one database, they would have to rewrite all their application's data access code for each database. The database vendors saw the problem and collaborated to create a common set of (really) low-level interfaces that all their servers would support. They called this API set Open DataBase Connectivity (ODBC).

However, writing ODBC code directly is tricky. Microsoft saw an opportunity to make things easier and wrote a set of Component Object Model (COM) components that was much easier to develop with and also let applications access data that wasn't stored in a database something you couldn't do with ODBC. Data stored in text files, spreadsheets, and other sources could now be used as well. This new technology, called OLE DB, used ODBC to access old databases where necessary and its own code to access databases and other data sources. Based on its rapid uptake, Microsoft decided to use it as the cornerstone for its Universal Data Access (UDA) strategy.

One of the aims of UDA was to provide a more object-oriented interface to data access than the somewhat procedural one that ODBC and OLE DB presented. Thus, Data Access Objects (DAO) and Remote Data Objects (RDO) were born in 1997, and a year later ActiveX Data Objects (ADO 2.0) gave classic ASP applications access to data sources in a constantly connected environment. ADO was built on top of OLE DB, which means that an application can use ADO, which talks to OLE DB, which talks to ODBC, which talks to a database, if you're using older databases such as Paradox or dBASE. ADO was even simpler than OLE DB to use, however.



Notes With the multilayered data access model and the connected nature of ADO, you could easily end up sapping server resources and creating a performance bottleneck. ADO served well, but ADO.NET is much, much better.

13.1.2 ADO.NET isn't a New Version of ADO

ADO.NET is a completely new data access technology, with a new design that was built entirely from scratch. Let's first get this cleared up: ADO.NET doesn't stand for ActiveX Data Objects .NET. Why? It's because of many reasons, but the following are the two most important ones:

Notes

- ADO.NET is an integral part of the .NET Framework, not an external entity.
- ADO.NET isn't a collection of ActiveX components.

The name ADO.NET is analogous to ADO because Microsoft wanted developers to feel at home using ADO.NET and didn't want them to think they'd need to "learn it all over again," so it purposely named and designed ADO.NET to offer similar features implemented in a different way.

During the .NET Framework design, Microsoft realized that ADO wasn't going to fit in. ADO was available as an external package based on COM objects, requiring .NET applications to explicitly include a reference to it. In contrast, .NET applications are designed to share a single model, where all libraries are integrated into a single framework, organized into logical namespaces, and declared public to any application that wants to use them. It was wisely decided that the .NET data access technology should comply with the .NET architectural model. Hence, ADO.NET was born.

ADO.NET is designed to accommodate both connected and disconnected environments.

Also, ADO.NET embraces the fundamentally important Extensible Markup Language (XML) standard, much more than ADO did, since the explosion in XML use came about after ADO was developed. With



Caution ADO.NET you cannot only use XML to transfer data between applications, but you can also export data from your application into an XML file, store it locally on your system, and retrieve it later when you need to do so.

Performance usually comes with a price, but in the case of ADO.NET, the price is definitely reasonable. Unlike ADO, ADO.NET doesn't transparently wrap OLE DB providers; instead, it uses managed data providers that are designed specifically for each type of data source, thus leveraging their true power and adding to overall application speed and performance. It's only if the data source doesn't have its own native data provider that you must fall back to using a generic OLE DB or ODBC data provider.

ADO.NET also works in both connected and disconnected environments. You can connect to a database, remain connected while simply reading data, and then close your connection, which is a process similar to ADO. Where ADO.NET really begins to shine is in the disconnected world. If you need to edit database data, maintaining a continuous connection would be costly on the server. ADO.NET gets around this by providing a sophisticated disconnected model. Data is sent from the server and cached locally on the client. When you're ready to update the database, you can send the changed data back to the server, where updates and conflicts are managed for you.

In ADO.NET, when you retrieve data, you use an object known as a data reader. When you work with disconnected data, the data is cached locally in a relational data structure called a dataset.

Self Assessment

Fill in the blanks:

1. The functionality is different from what you are used to with ADO.
2. One of the aims of UDA was to provide a more object-oriented interface to data access than the somewhat procedural one that and OLE DB presented.
3. In ADO.NET, when you data, you use an object known as a data reader.
4. ADO.NET gets around this by providing a sophisticated model.
5. ADO.NET is designed to accommodate both and disconnected environments.

13.2 ADO.NET and the .NET Framework

A dataset can hold large amounts of data in the form of tables (as `DataTable` objects), their relationships (as `DataRelation` objects), and constraints (as `Constraint` objects) in an in-memory cache, which can then be exported to an external file or to another dataset. Table 13.1 describes the namespaces in which ADO.NET components are grouped.

Table 13.1: ADO.NET Namespaces

Namespace	Description
<code>System.Data</code>	Classes, interfaces, delegates, and enumerations that define and partially implement the ADO.NET architecture
<code>System.Data.Common</code>	Classes shared by .NET Framework data providers
<code>System.Data.Odbc</code>	The .NET Framework data provider for ODBC
<code>System.Data.OleDb</code>	The .NET Framework data provider for OLE DB
<code>System.Data.OracleClient</code>	The .NET Framework data provider for Oracle
<code>System.Data.SqlClient</code>	The .NET Framework data provider for SQL Server
<code>System.Data.SqlServerCe</code>	The .NET Compact Framework data provider for SQL Server CE
<code>System.Data.SqlTypes</code>	Classes for native SQL Server data types

Since XML support has been closely integrated into ADO.NET, some ADO.NET components in the `System.Data` namespace rely on components in the `System.Xml` namespace. So, you sometimes need to include both namespaces as references in Solution Explorer.



Did u know? What is `System.Data` namespace?

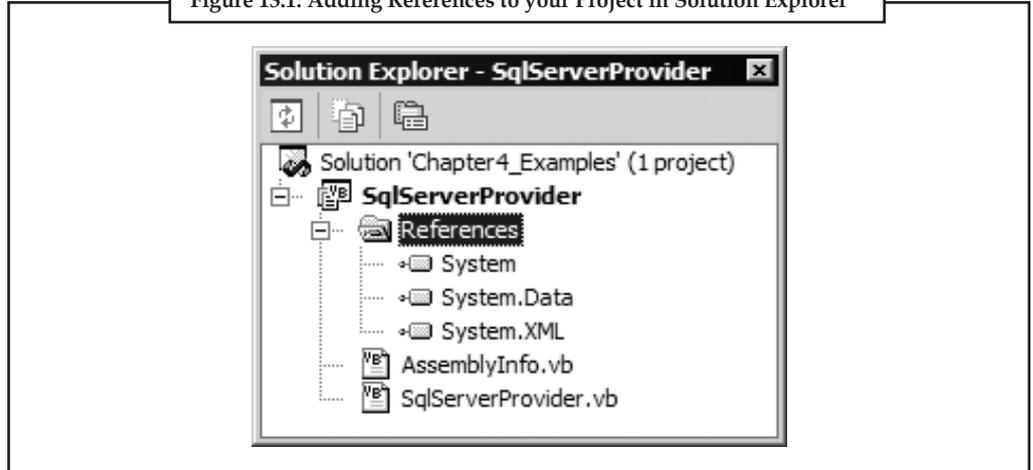
The `System.Data` namespace consists mostly of the classes that constitute the ADO.NET architecture. The ADO.NET architecture enables you to build components that efficiently manage data from multiple data sources. In a disconnected scenario (such as the Internet), ADO.NET provides the tools to request, update, and reconcile data in multiple tier systems.

These namespaces are physically implemented as assembly files, and if you create a new console application project in Visual Studio, references to the assemblies should automatically be created, along with the reference to the `System` assembly. However, if they're not present, simply perform the following steps to add the namespaces to your project:

1. Right-click the References item in Solution Explorer, and then click Add Reference....
2. A dialog box with a list of available references displays. Select `System.Data.dll`, `System.Xml.dll`, and `System.dll` (if not already present) one by one (hold down the `Ctrl` key for multiple selections), and then click the Select button.
3. Click OK, and the references will be added to the project. Solution Explorer should look similar to Figure 13.1.

Notes

Figure 13.1: Adding References to your Project in Solution Explorer



For those of you working from the command line, you can use the following compiler options to include the assemblies:

```
/r:System.dll /r:System.Data.dll /r:System.XML.dll
```

As you can see from Table 13.1, ADO.NET can work with older technologies such as OLE DB and ODBC. However, the SQL Server data provider communicates directly with SQL Server (versions 7.0 and newer), the most efficient form of connection, so it accesses SQL Server faster than OLE DB or ODBC. Likewise, the Oracle data provider accesses Oracle directly.

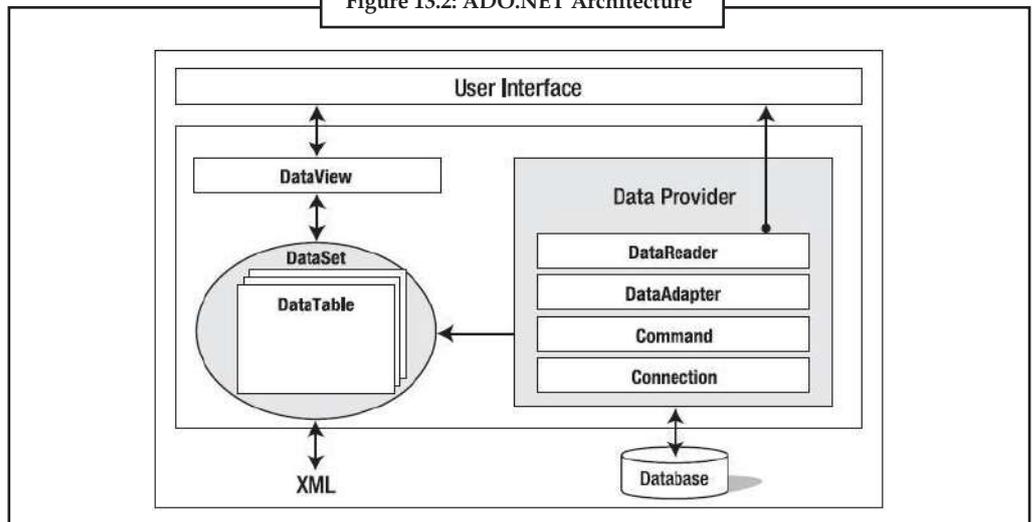


Task ADO.NET is a data access technology that's likely to be the primary data access technology of .NET for a long time, so learning how to use it is a worthwhile investment. Analyze.

13.3 Accessing Database with ADO.Net

Figure 13.2 presents the most important architectural features of ADO.NET.

Figure 13.2: ADO.NET Architecture



ADO.NET has two central components: data providers and datasets.

A data provider connects to a data source and supports data access and manipulation.

A dataset supports disconnected, independent caching of data in a relational fashion, updating the data source as required. A dataset contains one or more data tables. A data table is a row-and-column representation that provides much the same logical view as a SQL table. For example, you can store the data from the Northwind database's Employees table in a data table and manipulate the data as needed.

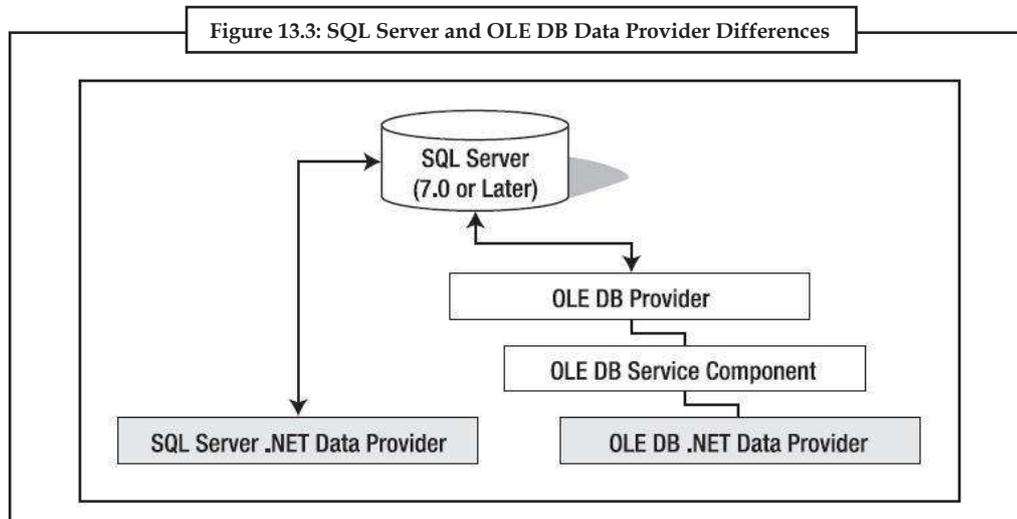
In Figure 13.2, notice the DataView class (in the System.Data namespace). This isn't a data provider component. Data views are used primarily to bind data to Windows and Web Forms.

As you saw in Table 13.1, each data provider has its own namespace. In fact, each data provider is essentially an implementation of interfaces in the System.Data namespace, specialized for a specific type of data source.

For example, if you use SQL Server version 7 or newer (SQL Server 2000 is actually version 8) as your database manager, you should use the SQL Server data provider (System.Data.SqlClient) because it's the most efficient way to work with these versions. This data provider communicates natively with SQL Server, bypassing the layers that OLE DB and ODBC connections have to use.

The OLE DB data provider supports access to older versions of SQL Server as well as to other databases, such as Access, DB2, MySQL, and Oracle. However, native data providers (such as System.Data.OracleClient) are preferable for performance, since the OLE DB data provider works through two other layers (the OLE DB service component and the OLE DB provider) before reaching the data source.

Figure 13.3 illustrates the difference between using the SQL Server and OLE DB data providers to access a SQL Server (version 7.0 or higher) database.



If your application connects to an older version of SQL Server (6.5 or newer) or to more than one kind of database server at the same time (for example, an Access and an Oracle database connected simultaneously), only then should you choose to use the OLE DB data provider.

No hard-and-fast rules exist; you can use both the OLE DB data provider for SQL Server and the Oracle data provider (System.Data.OracleClient) if you want, but it's important you choose the best provider for your purpose. Given the performance benefits of the server-specific data providers, if you use SQL Server or MSDE, 99 percent of the time you should be using the System.Data.SqlClient classes.

Notes

Before you look at what each kind of data provider does and how it's used, you need to be clear on their core functionality. Each .NET data provider is designed to do the following two things very well:

Provide access to data with an active connection to the data source. Provide data transmission to and from the independent datasets through data adapters. Database connections are established by using the data provider's connection class.



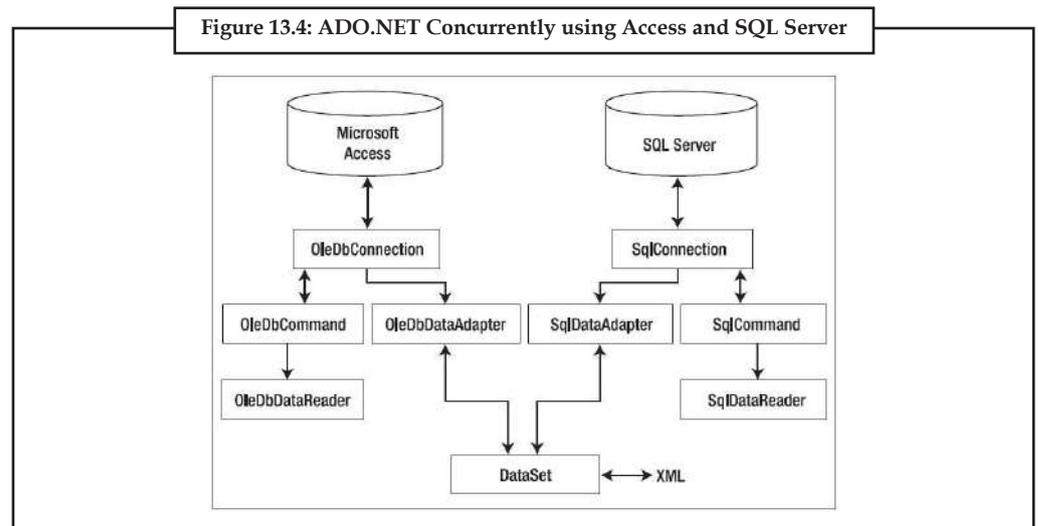
Example: System.Data.SqlClient.SqlConnection

Other components such as data readers, commands, and data adapters support retrieving data, executing SQL statements, and reading or writing to datasets, respectively.

As you've seen, each data provider is prefixed with the type of data source it connects to (for instance, the SQL Server data provider is prefixed with Sql), so its connection class is named SqlConnection. The OLE DB data provider's connection class is named OleDbConnection.

Look now at the object model of these two providers in Figure 13.4. You'll work simultaneously with different databases (Access and SQL Server) so you can see how easy it is to switch between data providers and what, if any, their main visible differences are. Note that the two data providers support the same application and share a single dataset.

The OLE DB data provider belongs to the System.Data.OleDb namespace; the SQL Server data provider belongs to System.Data.SqlClient. Both data providers seem to have a similar architecture, though they're actually very different internally.



Using the SQL Server Data Provider

The .NET data provider for SQL Server (7.0 or newer) is located in the System.Data.SqlClient namespace. This data provider communicates directly with the server using its native network protocol instead of through a number of layers (the way OLE DB does).



Task Give the fill forms

1. ODBC stands for
2. SQL stands for

Table 13.2 describes some important classes in the SqlClient namespace.

Notes

Table 13.2: Commonly used SqlClient Classes

Classes	Description
SqlCommand	Executes SQL queries, statements, or stored procedures
SqlConnection	Represents a connection to a SQL Server database
SqlDataAdapter	Represents a bridge between a dataset and a data source
SqlDataReader	Provides a forward-only, read-only data stream of the results
SqlError	Holds information on SQL Server errors and warnings
SqlParameter	Represents a command parameter
SqlTransaction	Represents a SQL Server transaction

Another namespace, System.Data.SqlTypes, maps SQL Server data types to .NET types, both enhancing performance and making developers' lives a lot easier.

Let's look at an example that uses the SQL Server data provider.

Try It Out: Creating a Simple Console Application Using the SQL Server Data Provider

You'll build a simple console application, which opens a connection and runs a query, using the SqlClient namespace against the MSDE Northwind database. You'll display the retrieved data in the console prompt window.

1. Open Visual Studio .NET, and create a new solution named 4_Examples.
2. Right-click the solution in Solution Explorer, select Add > New Project..., create a VB .NET console application, and name the application SqlServerProvider. Rename Module1.vb to SqlServerProvider.vb.
3. Since you'll be creating this example from scratch, select all the code in Code view and delete it. Enter the code in Listing 13.1.

Listing 13.1: SqlServerProvider Application Source

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Module SqlServerProvider
    Sub Main()
        'Set up connection string
        Dim ConnString As String = "server=(local)\netsdk;" & _
            "integrated security=true;database=northwind"
        'Set up query string
        Dim CmdString As String = "SELECT * FROM employees"
        'Declare Connection and DataReader variables
        Dim Conn As SqlConnection
        Dim Reader As SqlDataReader
        Try
```

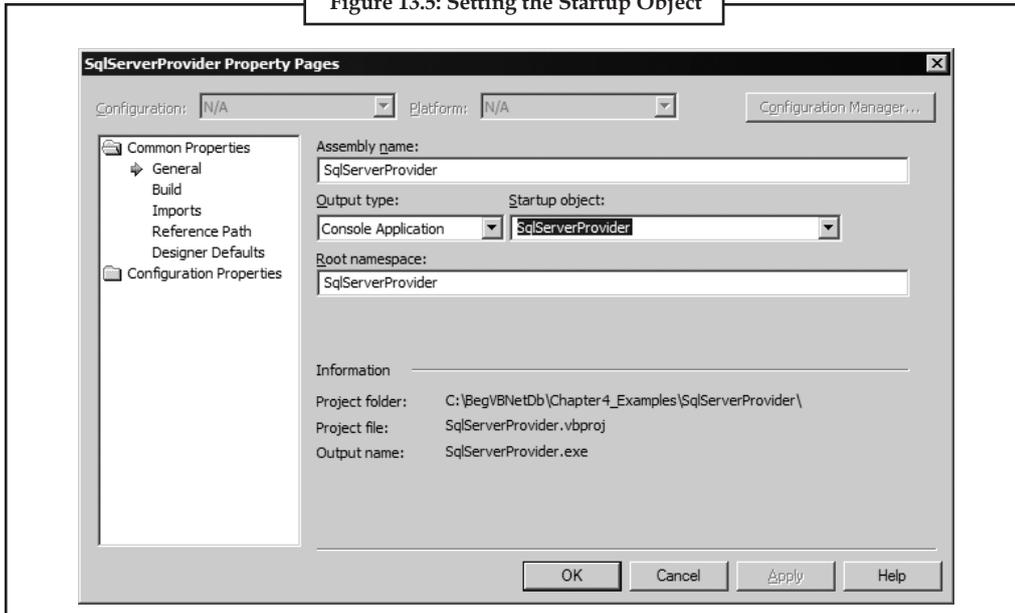
Notes

```
`Open Connection
Conn = New SqlConnection(ConnString)
Conn.Open()
`Execute Query
Dim Cmd As New SqlCommand(CmdString, Conn)
Reader = Cmd.ExecuteReader()
`Display output header
Console.WriteLine("This program demonstrates the use " & _
"of the SQL Server Data Provider." & ControlChars.NewLine)
Console.WriteLine("Querying database {0} with query {1}" & _
ControlChars.NewLine, Conn.Database, Cmd.CommandText)
Console.WriteLine("FirstName" & ControlChars.Tab & "LastName")
`Process The Result Set
While (Reader.Read())
    Console.WriteLine(Reader("FirstName").PadLeft(9) & _
ControlChars.Tab & Reader(1))
End While
Catch ex As Exception
    Console.WriteLine("Error: {0}", ex)
Finally
    `Close Connection
    Reader.Close()
    Conn.Close()
End Try
End Sub
End Module
```

4. Save the project, and then right-click the project in the Solution Explorer window. Select Properties, and set the startup object for the project to SqlServerProvider, as shown in Figure 13.5. Click OK.

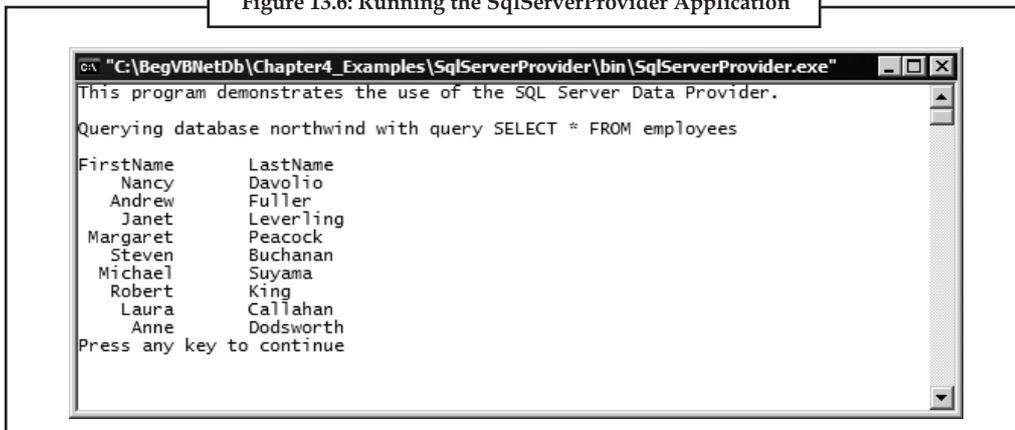
Notes

Figure 13.5: Setting the Startup Object



5. Save the project, and press Ctrl+F5 to run it. A console prompt window should appear, as shown in Figure 13.6.

Figure 13.6: Running the SqlServerProvider Application



How it Works

Let's take a look at how the code works, starting with the Imports directives.

```
Imports System
```

```
Imports System.Data
```

```
Imports System.Data.SqlClient
```

The reference to System.Data is actually not needed in this small program, since you don't explicitly use any of its members, but it's a good habit to always include it.



Caution The reference to System.Data.SqlClient is, of course, necessary, since you want to use the simple names of its members.

Notes

You specify the connection string with parameters (key-value pairs) suitable for an MSDE session.

```
`Set up connection string  
Dim ConnString As String = "server=(local)\netsdk;" & _  
"integrated security=true;database=northwind"
```

The connection string contains the following parameter, which specifies Windows Authentication:

```
integrated security=true;
```

The true value is equivalent to SSPI, which stands for Security Support Provider Interface. You then code the query, like so:

```
`Set up query string  
Dim CmdString As String = "SELECT * FROM employees"
```



Notes The connection string is actually parsed before it's assigned to the connection's `ConnectionString` property, so if you want to add newlines and extra spaces for readability, that's fine. Likewise, SQL can contain extraneous newlines and spaces, so you can format it to make it more readable and maintainable.

You next declare variables for the connection and data reader, so they're available to the rest of your code. You then create the connection, like so:

```
`Declare Connection and DataReader variables  
Dim Conn As SqlConnection  
Dim Reader As SqlDataReader  
Try  
`Open Connection  
Conn = New SqlConnection(ConnString)
```

You do this (and the rest of your database work) in a Try block to handle exceptions (in this particular case, to handle exceptions thrown by ADO.NET in response to database errors); however, in this simple example you're not interested in distinguishing them from other exceptions. Here, ADO.NET will throw an exception if the connection string parameters aren't syntactically correct, so you may as well be prepared. If you had waited until you entered the Try block to declare the connection (and data reader) variable, you wouldn't have it available in the Finally block to close the connection.

Creating the connection doesn't actually connect to the database. You need to call the Open method on the connection.

```
Conn.Open()
```

To execute the query, you first create a command object, passing its constructor the SQL statement to run and the connection on which to run it. Next, you create a data reader by calling `ExecuteReader` on the command object. This not only executes the query but also sets up the data reader. Note that unlike most objects, you have no way to create a data reader with a New expression.

```
`Execute Query  
Dim Cmd As New SqlCommand(CmdString, Conn)  
Reader = Cmd.ExecuteReader()
```

You then produce a header for your output, using connection and command properties (Database and CommandText, respectively) to get the database name and query text.

Notes

```
'Display output header
Console.WriteLine("This program demonstrates the use " & _
"of the SQL Server Data Provider." & ControlChars.NewLine)
Console.WriteLine("Querying database {0} with query {1}" & _
ControlChars.NewLine, Conn.Database, Cmd.CommandText)
Console.WriteLine("FirstName" & ControlChars.Tab & "LastName")
```

You retrieve all the rows in the result set by calling the data reader's Read method, which returns true if there are more rows and false otherwise. Note that the data reader is positioned immediately before the first row prior to the first call to Read.

```
'Process The Result Set
While (Reader.Read())
Console.WriteLine(Reader("FirstName").PadLeft(9) & _
ControlChars.Tab & Reader(1))
End While
```

You access each row's columns with the data reader's indexer (here, the SqlDataReader.Item property), which is overloaded to accept either a column name or a zero-based integer index. We're using both to demonstrate the indexer's use, but using column numbers is more efficient than using column names.

Next you handle any exceptions, quite simplistically, but at least you're developing a good habit.

```
Catch ex As Exception
Console.WriteLine("Error: {0}", ex)
```

Finally, in a Finally block, you close the data reader and the connection by calling their Close methods. As a general rule, you should close things in a Finally block to be sure they get closed no matter what happens within the Try block.

```
Finally
'Close Connection
Reader.Close()
Conn.Close()
End Try
End Sub
End Module
```

Technically, closing the connection also closes the data reader, but closing both (in the previous order) is another good habit. A connection with an open data reader can't be used for any other purpose until the data reader has been closed.

Self Assessment

Fill in the blanks:

6. The OLE DB data provider belongs to the namespace.
7. The OLE DB data provider's connection class is named.....

Notes

8. ADO.NET has two central components: data providers and.....
9. A is a row-and-column representation that provides much the same logical view as a SQL table.
10. The .NET data provider for SQL Server (7.0 or newer) is located in the namespace.
11. Finally block, you close the data reader and the connection by calling their methods.

13.4 Binding Controls to Recordsets

To bind a control to an ADO Recordset object, you just set that control's DataSource property to that object, and then set whatever other data properties those control needs to have set.

For example, to bind textboxes to an ADO Recordset:

```
Private Sub Form_Load()  
    'Declare the connection object variable  
    Dim conStudentDB As Connection  
    'Declare the recordset object variable  
    Dim rsStudent as Recordset  
    'Instantiate the connection object variable  
    Set conStudentDB = New Connection  
    'Instantiate the recordset object variable  
    Set rsStudent = New Recordset  
    'Open the database connection  
    conStudentDB.Open "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=" & App.Path &  
    "\StudDB.mdb;Persist Security Info=False"  
    'Open the Recordset for transaction  
    rsStudent.Open "SELECT * FROM Student", conStudentDB, adOpenStatic,  
    adLockOptimistic  
    'Set the DataSource and DataField property  
    Set txtLName.DataSource = rsStudent  
    txtLName.DataField = "LastName"  
    'Set the DataSource and DataField property  
    Set txtFName.DataSource = rsStudent  
    txtFName.DataField = "FirstName"  
End Sub  
  
//We Can Remove It
```

We can also bind more complex controls to a recordset. The following illustrates how to bind complex controls to a recordset.

```
Dim conStudentDB As ADODB.Connection  
Dim rsStudent As ADODB.Recordset  
'Instantiate a Connection object
```

Notes

```

Set conStudentDB = New Connection
`Open a new connection
conStudentDB.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & App.Path &
"\StudDB.mdb;Persist Security Info=False"
`Instantiate the Recordset object
Set rsStudent = New Recordset
`Open a new Recordset using the previous connection and return
`the appropriate records
rsStudent.Open "SELECT * FROM Student", conStudentDB
`Fill the DataGrid control with data from the Student table
Set dtaGridStudent.DataSource = rsStudent

```



Task Analyze how the binding controls with recordset can help us in practical life?

13.4.1 Adding a Record to a Recordset

To add a new record to an ADO record set, we use the AddNew method. After you've updated the fields of the current record, you save that record to the database with the Update method.

The following example uses the AddNew method to add a new record:

```

rsStudent.AddNew
rsStudent!LastName = txtLName.text
rsStudent!FirstName = txtFName.text
rsStudent.Update

```

13.4.2 Updating a Record in a Recordset

After changing the data in a record's fields or adding a new record, we update the data source to record the changes, using the Update method:

```
recordset.Update
```

For example, to update record in a Recordset (using data supplied by the user in the textbox):

```

Private Sub cmdUpdate_Click()
rsStudent!LName = txtLastName.text
rsStudent!FName = txtFirstName.text
rsStudent.Update
End Sub

```

13.4.3 Deleting a Record in a Recordset

To delete a record from the recordset, we must first navigate to the appropriate record. Use the Delete method to permanently remove the record from the recordset and the data source.

Notes

The following example deletes the current record in the rsStudent recordset:

```
Private Sub cmdDelete_Click()  
rsStudent.Delete  
End Sub
```



Task Explain with example for inserting the data into record set.

13.4.4 Adding a Record to a Recordset using SQL Statements

Once a connection has been established, you can begin any transaction to the database. By using the Connection object's Execute method, you can send SQL commands to the database without having to return records to the client.

For example, to add record to a Recordset using SQL statements:

```
Private Sub cmdAdd_Click()  
Dim conStudentDB As ADODB.Connection  
Dim rsStudent As ADODB.Recordset  
'Instantiate a Connection object  
Set conStudentDB = New Connection  
'Open a new connection  
conStudentDB.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & App.Path &  
"\StudDB.mdb;Persist Security Info=False"  
'Instantiate the Recordset object  
Set rsStudent = New Recordset  
'Open a new Recordset using the previous connection and return  
'the appropriate records  
rsStudent.Open "SELECT * FROM Student", conStudentDB  
conStudentDB.BeginTrans  
conStudentDB.Execute "INSERT INTO Student (LName, FName) VALUES ('Smith', 'Jane')"  
conStudentDB.CommitTrans  
conStudentDB.Close  
Set conStudentDB = Nothing  
End Sub
```

We can also insert a record to a Recordset using the values supplied by the user in a textbox or any other control.



Example: We have two textboxes named txtLastName and txtFirstName, we can insert record to a Recordset using the following code:

```
conStudentDB.Execute "INSERT INTO Student (LName, FName) VALUES ('" & txtLastName.  
text & "', '" & txtFirstName.text & "')
```

13.4.5 Updating a Record in a Recordset using SQL Statements

Notes

To update a record in a Recordset (when you know the values to update):

```
conStudentDB.Execute "UPDATE Student SET FName = 'Juan' WHERE LName = 'dela Cruz'"
```

OR

```
conStudentDB.Execute "UPDATE Student SET FName = '" & txtFirstName.text & "' WHERE LName = '" & txtLastName.text & "'"
```

If you are using the data supplied by the user in the txtLastName and txtFirstName textboxes.

Deleting a Record in a Recordset using SQL Statements

To delete a record in a Recordset when you know the values to delete:

```
conStudentDB.Execute "DELETE FROM Student WHERE LName = 'Smith'"
```

OR

```
conStudentDB.Execute "DELETE FROM Student WHERE LName = '" & txtLastName.text & "' OR FName = '" & txtFirstName.text & "'"
```

If you want to use the data supplied in the txtLastName or txtFirstName textbox.

13.4.6 Refreshing Data in a Recordset

To refresh data in a Recordset, use the following code:

```
RecordsetObject.Requery
```



Example: rsStudent.Requery

13.4.7 Executing

Each .NET Framework data provider included with the .NET Framework has its own command object that inherits from DbCommand. The .NET Framework Data Provider for OLE DB includes an OleDbCommand object, the .NET Framework Data Provider for SQL Server includes a SqlCommand object, the .NET Framework Data Provider for ODBC includes an OdbcCommand object, and the .NET Framework Data Provider for Oracle includes an OracleCommand object.



Notes Each of these objects exposes methods for executing commands based on the type of command and desired return value, as described in the following table.

Command	Return Value
ExecuteReader	Returns a DataReader object.
ExecuteScalar	Returns a single scalar value.
ExecuteNonQuery	Executes a command that does not return any rows.
ExecuteXMLReader	Returns an XmlReader. Available for a SqlCommand object only.

Each strongly typed command object also supports a CommandType enumeration that specifies how a command string is interpreted, as described in the following table.

Notes

CommandType	Description
Text	An SQL command defining the statements to be executed at the data source.
StoredProcedure	The name of the stored procedure. You can use the Parameters property of a command to access input and output parameters and return values, regardless of which Execute method is called. When using ExecuteReader, return values and output parameters will not be accessible until the DataReader is closed.
TableDirect	The name of a table.

13.5 Select Command with Database

```

ID      FirstName  LastName
1       Ham       Yin
111     F           L
11      S           S
12      J           J
13      D           D
2       Edna      Everage
114     F           L
115     F           L
2       Edna      Everage
1       Ham       Yin
    
```

```

Imports System
Imports System.Collections
Imports System.Data
Imports System.IO
Imports System.Xml.Serialization
Imports System.Windows.Forms
Imports System.Data.SqlClient

Public Class MainClass
    Shared Dim WithEvents con As SqlConnection
    Shared Sub Main()
        con = New SqlConnection("Server=(local)\SQLEXPRESS;Initial
        Catalog=MyDatabase;Integrated Security=SSPI")
        ` Create the command object
        Dim str As String = "SELECT ID, FirstName, LastName FROM Employee"
        Dim cmd As New SqlCommand(str, con)
        Dim da As New SqlDataAdapter(cmd)
        Dim ds As New DataSet()
        da.Fill(ds, "Employee")
        Dim dc As DataColumn
        For Each dc In ds.Tables(0).Columns
    
```

```

        Console.WriteLine("{0,15}", dc.ColumnName)
    Next
    Console.WriteLine(vbCrLf)
    Dim dr As DataRow
    For Each dr In ds.Tables(0).Rows
        Dim i As Integer
        For i = 1 To ds.Tables(0).Columns.Count
            Console.WriteLine("{0,15}", dr(i - 1))
        Next i
        Console.WriteLine(vbCrLf)
    Next
End Sub
End Class

```

Self Assessment

Fill in the blanks:

12. By using the Connection object's method, you can send SQL commands to the database without having to return records to the client.
13. We can also insert a record to a Recordset using the values supplied by the user in a or any other control.
14. Each .NET Framework data provider included with the .NET Framework has its own that inherits from DbCommand.
15. You can use the..... property of a command to access input and output parameters and return values, regardless of which Execute method is called.



Caselet

Google runs 'a little bit like a university'

IF Google has underwear, who will buy it? This was one of the questions that Sergey Brin posed to students of an Israeli high school in 2003. "Hands shot up," writes David A. Vise in *The Google Story*, from Macmillan (www.panmacmillan.com).

To Brin and Larry Page, the Google guys, named recently 'Men of the Year' by Financial Times, the project was 'one of the less technical' ones.

To check, you may use Google to briefly search for the brief, and see a March 2005 posting on <http://blog.searchenginewatch.com> confirming thus: "UnderGoos (naturally in beta) offers a full-line of Google-branded undergarments." Price, however, is £0.00 on www.google-store.com. "A project that is closer to our hearts is translation," Brin told the students.

The book takes one 'inside the hottest business, media and technology success of our time,' to know how Google, a start-up investment of \$1 million, became a market value of \$40 billion, all by word of mouth.

Contd...

Notes

"With its colourful, childlike logo set against a background of pure white, Google's magical ability to produce speedy, relevant responses to queries hundreds of millions of times daily has changed the way people find information and stay abreast of news," writes Vise in the introduction.

Ten years ago, search was 'not pretty,' is how Rajeev Motwani, "a 30-year old professor who had been Sergey's advisor since his arrival at Stanford in 1993", found.

Vise writes about how Motwani had tested a search engine called Inktomi after it was developed at Berkeley, where he had received his Ph.D.

"He typed in 'Inktomi' to see what would happen. Sure enough, said Motwani, 'It wasn't there. It couldn't find itself.'" Not so with Google, which shows 800,000,000 results for itself in 0.17 seconds.

"When Google went looking for someone to ramp up its computer network, Larry and Sergey hired a brain surgeon, Dr Jim Reese," informs the book. "Named Google's operations chief, Reese managed the company's burgeoning collection of computer hardware."

Vise narrates the story of how the company "cobbled together a virtual supercomputer from cheap, commodity PCs." Back at the Googleplex, the garden-variety PCs got ripped apart, and all 'unnecessary' parts that would eat up computing power and resources' disposed of, to build streamlined computers, strung together with "software, wiring, and the special sauce that made Google lightning fast."

More than one lakh inexpensive PCs, stacked in refrigerator-size racks, remain 'strictly off-limits to outsiders', notes Vise. "PCs burn out and are not replaced. Instead other PCs take over."

Learn about the 20 per cent rule in Google from Krishna Bharat, a software engineer in the company's research group.

The rule stipulates that engineers spend at least 20 per cent of their time, or one day a week, working on whatever projects interested them.

"The 20 per cent rule was a way of encouraging innovation, and both Brin and Page saw this as essential to establishing and maintaining the right culture and creating a place where bright technologists would want to work and be motivated to come up with breakthrough ideas." 3M had something similar - the 15 per cent rule - many years earlier, notes Vise. A success product that had emerged from such a pursuit was Post-it Notes.

"Rather than having employees moonlighting as inventors at home - with the risk that an idea will either fail from lack of resources or succeed to the point that they quit to pursue it full-time - Google gives them both freedom and resources," observes Vise.

When speaking at the Israeli school, Brin had said, "We run Google a little bit like a university. We have lots of projects, about 100 of them. We like to have small groups of people, three or so people, working on projects. Some of them, for example, are related to molecular biology. Others involve building hardware... The only way you are going to have success is to have lots of failures first."

And Brin toys with the idea of plugging into brain 'a little version of Google', as you'd know from the final chapter, titled 'Googling your genes.'

Dr Craig Venter, who had decoded the human genome, is of the view that genetic information is going to be the leading edge. "Working with Google, we are trying to generate a gene catalogue to characterise all the genes on the planet and understand their evolutionary development. The massive computing power can be used "to analyse vast

Contd...

quantities of data with billions of parts” says Dr Alan E. Guttmacher, deputy director of the National Human Genome Research Institute, cited in the book.

“We are beginning to have incredible tools to understand the biology of human diseases in ways we never have before, and to come up with novel ways to prevent and treat them.”

Fabulous read.

All about ADO.NET

SAHIL Malik, who has been working as a consultant in Microsoft technology for about a decade, and also leading the office of Emerging Technologies at the National Cancer Institute, has written Pro ADO.NET 2.0 from Dreamtech Press (www.wileydreamtech.com) . “ActiveX Data Objects (ADO) was the premier data access technology under the Microsoft umbrella before ADO.NET was introduced as an integral part of the .NET Framework,” chronicles Malik.

“What sets ADO.NET apart from previous data access technologies is that it allows you to interact with your database in a completely disconnected data cache to work with data offline.”

Disconnected data access is crucial for today’s high-demand applications, notes the author.

The book has chapters on connecting to a data source, retrieving data in a connected fashion, DataSets, sorting and searching, updating data, and so on. Of value is the chapter on ‘best practices’ where Malik discusses the right tools. For instance, he reminds that data reader consumes less memory than a DataSet.

“A data reader is an object that allows you to access information on only a single row of data at a given time.

What this means is that, regardless of the size of a result set, traversing this result set with a data reader will only ever have a single record loaded in memory at a given time.”

There are many flavours of transactions to choose from, writes Malik, listing out the same ‘in an increasing order of management overhead and decreasing order of performance’.

The list begins with implicit transactions, which are automatically associated with any single SQL statement and ensure “the sanctity of the data during that statement’s execution time period,” and ends with “storing a snapshot of previous data, which acts as your ‘recovery contract’ and a flag on the ‘in doubt’ rows.”

Helpful notes are strewn all over the book. One such reads, “Retrieving a large volume of data within the context of a single connection will always be faster than retrieving small portions and opening and closing the connection each time, because large-block retrieval causes less network roundtrips and incurs less latency.”

Useful for the ADO techie.

13.6 Summary

In this unit, you saw why ADO.NET was developed and how it supersedes other data access technologies in the .NET Framework. We gave an overview of its architecture and then focused on one of its core components, the data provider. You built three simple examples to practice basic data provider use and experience the uniform way data access code is written, regardless of the data provider. Finally, we offered the opinion that conceptual clarity is the key to understanding and using both data providers and the rest of the ADO.NET API.

Notes

13.7 Keywords

ADO.NET: ADO.NET is a completely new data access technology, with a new design that was built entirely from scratch.

Database: Collection of related files stored in a particular format and accessed through database software.

DataSet: The DataSet is the object you have been dealing with for some time. The DataSet actually contains one or more DataTable objects.

oleDbConnection: The oleDbConnection object uses OLE DB to connect to a data source.

VB.NET Development Environment: The Visual Basic .NET development environment comes with a bewildering assortment of tools, wizards, and windows for building, examining, and modifying Visual Basic projects.

13.8 Review Questions

1. Explain how to access database in windows application.
2. Describe oleDbConnection and SqlConnection in detail.
3. Distinguish between oleDbCommand and SqlCommand.
4. How do different components of ADO.Net interact with each other in disconnected architecture?
5. How do we read data (or records) from database using data reader?
6. Can we connect two datareader to same data source using single connection at same time?
7. How do you get records number from 5 to 15 in a dataset of 100 records? Write code.
8. What is the difference between ADO and ADO.NET? Explain in detail.
9. What are good ADO.NET object(s) to replace the ADO Recordset object?
10. What is the architecture of connected environment of data access in ADO.Net?

Answers: Self Assessment

- | | |
|--------------------|---------------------------|
| 1. ADO.NET | 2. ODBC |
| 3. retrieve | 4. disconnected |
| 5. connected | 6. System.Data.OleDb |
| 7. OleDbConnection | 8. Datasets |
| 9. data table | 10. System.Data.SqlClient |
| 11. Close | 12. Execute |
| 13. Textbox | 14. command object |
| 15. Parameters | |

13.9 Further Readings

Notes



Books

- A. Chakraborti et al., *Microsoft .NET Framework*, PHI, 2002.
- Andrew Troelsen, *C# and the .NET Platform*, a! Press, 2002.
- Bill Evjen, Jason Beres et al., *Visual Basic.Net Programming Bible*, Wiley India.
- Jeffery Richter, *Applied Microsoft .NET Framework Programming*, Microsoft, 2002.
- M. Reynolds et al., *.NET Enterprise*, Wrox/SPD, 2002.
- Paul Dicinson and Fabio Claudio ferracchiati, *Professional ADO.NET with VB.NET*, a! Press, 2002.
- Richard Lienecker, *Using ASP.NET*, Pearson Education, 2002.
- Vikas Gupta, *.Net Programming*, Dreamtech Publication.



Online links

- <http://en.wikipedia.org/wiki/ADO.NET>
- [http://msdn.microsoft.com/en-us/library/system.data\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/system.data(v=vs.71).aspx)

Unit 14: XML Basics

CONTENTS

Objectives

Introduction

14.1 What is XML?

14.2 Structure of an XML Document

14.2.1 Tree Structure of XML

14.3 Data Representation through XML

14.3.1 Document Type Definition (DTD)

14.3.2 Integrating CSS in our XML Document

14.4 XML Parsers

14.4.1 Classification of XML Parsers

14.5 XML Reader Class

14.6 XML Writer Class

14.7 Summary

14.8 Keywords

14.9 Review Questions

14.10 Further Readings

Objectives

After studying this unit, you will be able to:

- Scan what is XML?
- Describe data representation through XML
- Demonstrate the working with XMLReader and XML Writer Class

Introduction

XML stands for EXTensible Markup Language. XML is a markup language much like HTML. XML was designed to carry data, not to display data. XML tags are not predefined. You must define your own tags. XML is designed to be self-descriptive. XML is a W3C Recommendation.

14.1 What is XML?

XML, or Extensible Markup Language, is a markup language that you can use to create your own tags. It was created by the World Wide Web Consortium (W3C) to overcome the limitations of HTML, the Hypertext Markup Language that is the basis for all Web pages. Like HTML, XML is based on SGML – Standard Generalized Markup Language. Although SGML has been used in the publishing industry for decades, its perceived complexity intimidated many people that

otherwise might have used it (SGML also stands for “Sounds great, maybe later”). XML was designed with the Web in mind.

Advantages of XML

- It is as easy as HTML.
- XML is fully compatible with applications like JAVA, and it can be combined with any application which is capable of processing XML irrespective of the platform it is being used on.
- XML is an extremely portable language to the extent that it can be used on large networks with multiple platforms like the internet, and it can be used on handhelds or palmtops or PDAs.
- XML is an extendable language, meaning that you can create your own tags, or use the tags which have already been created.

There are other advantages of using XML.

- It is a platform independent language.
- It can be deployed on any network if it is amicable for usage with the application in use.
- If the application can work along with XML, then XML can work on any platform and has no boundaries.
- It is also vendor independent and system independent. While data is being exchanged using XML, there will be no loss of data even between systems that use totally different formats.

From a programmer’s point of view, there are a lot of parsers available like the API, C and many more. If your data is very rich, then using XML to capture the data makes a lot of sense mainly because it is in plain text and in a language that humans can read.



Notes XML also gives the freedom to define your own tags that fit your application needs. XML can also be stored in databases in XML format and human readable format.

The advantages of XML include that it can be used as an instrument to share data and application models in wide networks like internet.

Disadvantages of XML

However, awesome XML is, there are some drawbacks which have hindered it from gaining widespread use since its inception:

- **The lack of adequate processing applications.**

For one, XML requires a processing application. That is, the nice thing about HTML was that you knew that if you wrote an HTML document, anyone, anywhere in the world, could read your document using Netscape. Well, with XML documents, that is not yet the case. There are no XML browsers on the market yet (although the latest version of IE does a pretty good job of incorporating XSL and XML documents provided HTML is the output).

Thus, XML documents must either be converted into HTML before distribution or converting it to HTML on-the-fly by middleware. Barring translation, developers must code their own processing applications.

Notes

- **Browser support is limited.**

IE 5 and Netscape 5 are expected to fully support XML. Also, W3C's Amaya browser supports it today, as does the JUMBO browser that was created for the Chemical Markup Language.

- **XML isn't about display – it's about structure.**

This has implications that make the browser question secondary. So the whole issue of what is to be displayed and by what means is intentionally left to other applications. You can target the same XML (with different XSL) for different devices (standard web browser, palm pilot, printer, etc.). However, this takes some magic and the amount of work necessary even to print "hello world" are sometimes enough to dissuade developers from adopting the technology.

Nevertheless, parsing algorithms and tools continue to improve over time as more and more people see the long-term benefits of migrating their data to XML. The backend part of XML will continue to become simpler and simpler. Already Internet Explorer and Netscape provide a decent amount of built in XML parsing tools.

14.2 Structure of an XML Document

XML documents use a self-describing and simple syntax:



Example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>Dilkeshwar</to>
<from>Upendra</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The first line is the XML declaration. It defines the XML version (1.0) and the encoding used (ISO-8859-1 = Latin-1/West European character set).

The next line describes the root element of the document (like saying: "this document is a note"):

```
<note>
```

The next 4 lines describe 4 child elements of the root (to, from, heading, and body):

```
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
```

And finally the last line defines the end of the root element:

```
</note>
```

You can assume, from this example, that the XML document contains a note to Dilkeshwar from Upendra.

XML Naming Rules

Notes

XML elements must follow these naming rules:

- Names can contain letters, numbers, and other characters
- Names cannot start with a number or punctuation character
- Names cannot start with the letters xml (or XML, or Xml, etc.)
- Names cannot contain spaces

14.2.1 Tree Structure of XML

XML documents must contain a root element. This element is “the parent” of all other elements.

The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree.

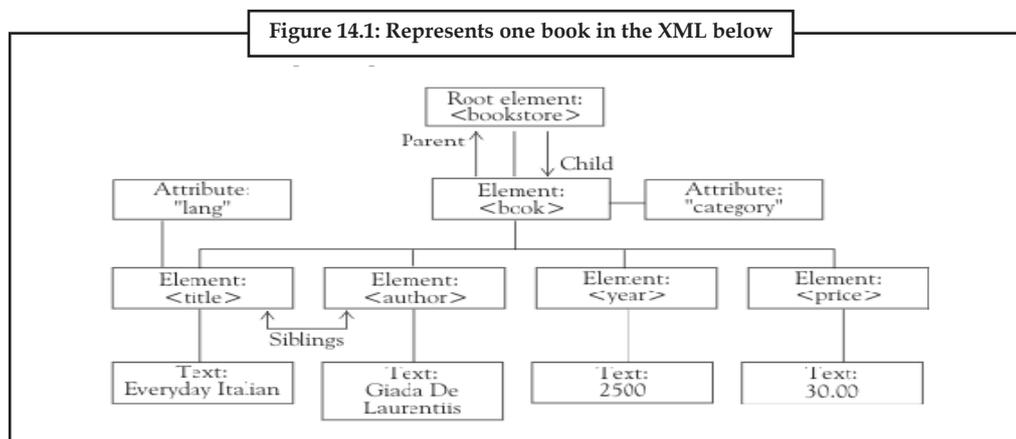
All elements can have sub elements (child elements):

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

The terms parent, child, and sibling are used to describe the relationships between elements. Parent elements have children. Children on the same level are called siblings (brothers or sisters).

All elements can have text content and attributes (just like in HTML).

Consider the following Example:



Example: XML code for the above structure.

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
```

Notes

```
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
</book>
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```

The root element in the example is <bookstore>. All <book> elements in the document are contained within <bookstore>.

The <book> element has 4 children: <title>,< author>, <year>, <price>.

Self Assessment

Fill in the blanks:

1. XML was designed with the in mind.
2. XML documents must contain a element. This element is "the parent" of all other elements.
3. The advantages of XML include that it can be used as an instrument to share data and models in wide networks like internet.
4. XML is an extremely portable language to the extent that it can be used on large with multiple platforms

14.3 Data Representation through XML

An XML entity can play several roles, such as a placeholder for repeatable characters (a type of short-hand), a section of external data (e.g., XML or other), or as a part of a declaration for elements. Schemas have their own mechanism for handling entities.

Entities have been used in most published DTDs and schemas to organize repeating values, represent special symbols (e.g., formatting characters and copyright symbols), or to define parameters for elements. You may not even realize that you've worked with XML entities, because certain entities are incorporated into HTML (e.g., using * * for a non breaking space).



Did u know? **What is the difference between XML and HTML?**

XML is not a replacement for HTML.

XML and HTML were designed with different goals:

1. XML was designed to transport and store data, with focus on what data is
2. HTML was designed to display data, with focus on how data looks

HTML is about displaying information, while XML is about carrying information.

The XML specification contains only five entities but gives developers and architects the ability to create their own entities to use throughout the current XML document. In XML, entities follow the same pattern used in HTML or SGML: an ampersand (&) followed by the name of the entity, and then a semicolon (;).

Entities Used as Placeholders

The sample below uses an internal DTD. In this scenario, an online mail system requires stores to submit this document, which will be processed into a larger system. A store has a name, phone number, some promotional text, and a logo.

Using entities, if the store's name changes, the XML editor or application doesn't need to parse through the entire document to find each time the name is referenced, because only the storeName entity needs to be changed. After you create the XML file, it can be uploaded to a server to be processed. The store owner can also submit a much simpler file containing text for the store's promotion or tag line. If this information changes, you need only submit a new text file to the system like this:

```
<?xml version="1.0"?>
<!DOCTYPE store [
<!ELEMENT mall (store)>
<!ELEMENT store (name,phone,promo,image)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT promo (#PCDATA)>
<!ELEMENT image (#PCDATA)>
<!ELEMENT text (#PCDATA)>
<!ENTITY storeName "Sample Store">
<!ENTITY storePromo SYSTEM "./StorePromo.txt">
<!ATTLIST image width CDATA #IMPLIED>
]>
<store>
<name>&storeName;</name>
<phone>222-333-4444</phone>
<promo>&storePromo;</promo>
<image width="200">logo.gif</image>
</store>
```

Notes

Notes

Entities to Retrieve External Data

When working with XML documents, there comes a time when one XML document needs to include another. There are many techniques you can use to do this (e.g., XSL and XPointers), but entities are probably the most straightforward and most supported way to set it up. Note that the text should be encoded to be valid in an XML document (don't use & or < characters). In the above example, the store's promotional text can be in any format, including an ASCII text file (the code in the previous section works only with a text file), a .gif image, or included XML as in the example below:



Example:

```
<storePromo>
<tagline>The best prices in town</tagline>
<description>Check out our prices during this week's sale</description>
</storePromo>
```

The above elements should also be defined in the main XML document's DTD or schema. When processing the document, the tag line can be accessed from the DOM tree /mall/store/promo/storePromo/tagline.

Entities that Help Define the Structure of a Document

Parameter entities are a different mechanism altogether; they allow portions of the XML document to be defined as shortcuts to an element's parameter list. As DTDs grow, element definitions can become quite complicated. Parameter entities cannot be defined and used in an internal DTD as in the examples above.

Unlike the previous example, to work with parameter entities, a separate DTD file needs to be used. The original store declaration is simple:

```
<!ELEMENT store (name,phone,promo,image)>
```

However, when you add a type to the store element, the declaration becomes more complicated. Add a few more parameters, and deciphering the element would become difficult. There is a trade-off when using parameter entities: Either the element declarations become increasingly complicated or the document structure becomes more complicated because it must manage entities and elements.



Notes Most people choose to use entities because they make it easier to manage larger DTDs with complicated elements, and the parameter entities can be reused for similar elements in the document.

At first, the type parameter added to the store element might look like this:

```
<!ELEMENT store ((retail | food | concourse),name,phone,promo,image)>
```

Using parameter entities (prefixed with a %), a *type* entity can be created. Then this entity can replace the parameter list. The *%type;* entity needs to be declared as follows:

```
<!ENTITY %type "(retail | food)">
```

The store element declaration now returns to a somewhat more readable list like this:

```
<!ELEMENT store (%type;, name,phone,promo,image)>
```

The final document is listed below. This time, I also used an XML document for the promotional tag line instead of text, like so:

```
<?xml version="1.0"?>
<!ENTITY % type "(retail|food)">
<!ENTITY % boolean "(true|false)">
<!ENTITY phonePrefix "ph:">
<!ENTITY storePromo SYSTEM "./StorePromo.xml">
<!ELEMENT store ((%type;), name, phone, promo, image)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT promo (text)>
<!ELEMENT image (#PCDATA)>
<!ELEMENT retail (#PCDATA)>
<!ELEMENT food (#PCDATA)>
<!ELEMENT text (#PCDATA)>
<!ATTLIST image
nosave %boolean; "true"
width CDATA #REQUIRED>
```

Using Attributes

Similar to using entities, attributes can supply values that are missing from the document. Attributes are a different mechanism and can cause confusion. You can define a set of valid values for an attribute and store them as a parameter entity. In the example above, the %boolean entity handles any attribute that needs a true/false pair. Again, this works only when using a separate DTD file. Attribute defaults can be useful with pictures or prices, such as shipping charges (i.e., if the shipping is empty, you default it to a standard charge).

It takes some time to get used to implementing entities with XML, but keep them in mind when defining XML documents and projects. Also, if you use them, it's a good idea to keep a list for reference or use a DTD parser to keep track of the DTD design.

14.3.1 Document Type Definition (DTD)

A DTD defines the structure of the content of an XML document, thereby allowing us to store the data in a consistent format. It specifies the element that can be presented in the xml document, attribute of these elements and their arrangement with relation to each other. It also allows to specify whether an element is optional or mandatory.

Creating a DTD is similar to creating a table in a database. In DTDs, we specify the structure of data by declaring elements to denote the data which is similar to creating columns in a table. We can also specify whether providing a value for the element is mandatory or optional and we can then store the data in an xml document that confirms to the DTD for that application. This is similar to adding records in a table.

XML allows us to create our own DTDs for applications this gives us complete control over the process of checking the content and structure of xml documents created for an application. This checking process is called validation. XML document that confirms to a DTD are considered as valid documents.

Declaring Elements and Attributes in an XML DTD

The examples below often redefine the same element. This is for simplicity only; it is an error to define an element more than once in an actual DTD.

XML Markup Languages

An XML document primarily consists of a strictly nested hierarchy of elements with a single root. Elements can contain character data, child elements, or a mixture of both. In addition, they can have attributes. Child character data and child elements are strictly ordered; attributes are not.



Example: Declaring attributes.

```
<?xml version="1.0" ?>
<Book Author="Anonymous">
  <Title>Sample Book</Title>
  <Chapter id="1">
    This is chapter 1. It is not very long or interesting.
  </Chapter>
  <Chapter id="2">
    This is chapter 2. Although it is longer than chapter 1, it is not anymore
    interesting.
  </Chapter>
</Book>
```

The names of the elements and attributes and their order in the hierarchy (among other things) form the XML markup language used by the document. This language can be defined by the document author or it can be inferred from the document's structure. In the example shown above, the language contains three elements: Book, Title, and Chapter. The Book element contains a single Title element and one or more Chapter elements.



Caution The Book element has an Author attribute and the Chapter element has an id attribute.

The main reason to explicitly define the language is so that documents can be checked to conform to it. For example, if we defined a grammar for the Book language, authors using this grammar could use a validating parser to ensure that their documents conformed to the language.

An XML markup language is defined in a Document Type Definition (DTD). The DTD is either contained in a <!DOCTYPE> tag, contained in an external file and referenced from a <!DOCTYPE> tag, or both. For example, the document shown above could contain the following <!DOCTYPE> tag:

```
<!DOCTYPE Book [
  <!ELEMENT Book (Title, Chapter+)>
  <!ATTLIST Book Author CDATA #REQUIRED>
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Chapter (#PCDATA)>
  <!ATTLIST Chapter id ID #REQUIRED>
]>
```

Elements**Notes**

1. An element is defined as a group of one or more sub elements/subgroups, character data, EMPTY, or ANY. For example:

Group:

```
<!ELEMENT A (B, C)>
```

Character data:

```
<!ELEMENT A (#PCDATA)>
```

EMPTY:

```
<!ELEMENT A EMPTY>
```

ANY:

```
<!ELEMENT A ANY>
```

2. Elements defined as groups of sub elements/subgroups constitute non-terminals in the language. Elements defined as character data, EMPTY, or ANY constitute terminals. For example:

```
<!-- Element A is a non-terminal. -->
```

```
<!ELEMENT A (B)>
```

```
<!-- Element B is a terminal. -->
```

```
<!ELEMENT B (#PCDATA)>
```

Although it is legal to define a language containing non-terminals that never resolve to terminals, such as one with purely circular definitions, it is generally impossible and/or useless to create any valid documents for such languages.

3. Groups can be either a sequence or choice of sub elements and/or subgroups. For example:

Sequence:

```
<!-- Element A consists of a single element B. -->
```

```
<!ELEMENT A (B)>
```

```
<!-- Element A consists of element B followed by element C. -->
```

```
<!ELEMENT A (B, C)>
```

```
<!-- Element A consists of a sequence, including a choice subgroup. -->
```

```
<!ELEMENT A (B, (C | D), E)>
```

Choice:

```
<!-- Element A consists of either element B or element C. -->
```

```
<!ELEMENT A (B | C)>
```

```
<!-- Element A consists of a choice, including a sequence subgroup. -->
```

```
<!ELEMENT A (B | C | (D, E))>
```

4. Optional (?), one-or-more (+), and zero-or-more (*) operators can be applied to groups, subgroups, and sub elements. For example:

Optional:

```
<!-- Subelement B is optional. -->
```

```
<!ELEMENT A (B?, C)>
```

Notes

One or more:

```
<!-- Subgroup (C | D) occurs one or more times. -->
<!ELEMENT A (B, (C | D)+, E)>
```

Zero or more:

```
<!-- Group (B, C) occurs zero or more times, i.e. A can be empty. -->
<!ELEMENT A (B, C)*>
```

5. Elements containing character data can be declared as containing only character data:

```
<!ELEMENT A (#PCDATA)>
```

or as containing a mixture of character data and elements in any order:

```
<!ELEMENT A (#PCDATA | B | C)*>
```

In the latter case, the declaration must place #PCDATA first in the group, the group must be a choice, and the group must appear zero or more times. Such groups are generally referred to as “mixed content” (as opposed to element-only groups or “element content”). Technically, mixed content refers to any element containing character data. However, in common usage it refers only to the latter case.



Notes “PCDATA” in the declarations is short for “Parsed Character DATA”. The term is inherited from SGML and comes from the fact that the text in the XML document following the element tag is parsed looking for more markup tags. Although it is possible to include unparsed character data through the use of CDATA sections, these can occur only where PCDATA occurs. While this is of interest to parser writers, it does not affect the syntax of DTDs, nor does it affect the resulting elements – they still contain character data.

6. EMPTY means that the element has no child elements or character data. Empty elements often have attributes – see below.
7. ANY means that the element can contain zero or more child elements of any declared type, as well as character data. It is therefore a shorthand for mixed content containing all declared elements.

Attributes

1. Elements can have zero or more attributes. For example:

```
<!ELEMENT A (#PCDATA)>
<!-- Declare an attribute a for element A -->
<!ATTLIST A a CDATA #IMPLIED>
```

2. A single ATTLIST statement can declare multiple attributes for the same element. Multiple ATTLIST statements can declare attributes for the same element. That is, the following are equivalent:

Single ATTLIST statement declaring multiple attributes for an element:

```
<!-- Element A has attributes a and b -->
<!ATTLIST A
    (a)          CDATA #IMPLIED
    (b)          CDATA #IMPLIED>
```

Multiple ATTLIST statements declaring attributes for the same element:

Notes

```
<!-- Element A has attributes a and b -->
<!ATTLIST A a CDATA #IMPLIED>
<!ATTLIST A b CDATA #IMPLIED>
```

3. Attributes can be optional, required, or have a fixed value. Optional attributes can have a default; fixed attributes must have a default. For example:

Optional without a default:

```
<!-- Element A has an attribute a. #IMPLIED = "optional, no default" -->
<!ATTLIST A a CDATA #IMPLIED>
```

Optional with a default:

```
<!-- If attribute a is not provided, a default of "aaa" will be used. -->
<!ATTLIST A a CDATA "aaa">
```

Required:

```
<!ATTLIST A a CDATA #REQUIRED>
```

Fixed:

```
<!-- The value of attribute a is always "aaa" -->
<!ATTLIST A a CDATA #FIXED "aaa">
```

4. Each attribute has a type:

Character data:

```
<!ATTLIST A a CDATA #IMPLIED>
```

A user-defined enumerated type:

```
<!-- Attribute a uses a simple enumeration. -->
<!ATTLIST A a (yes | no) #IMPLIED>
<!-- Attribute a uses an enumeration of notation types.
```

See the XML specification for complete details. -->

```
<!ATTLIST A a NOTATION (ps | pdf) #IMPLIED>
```

ID, IDREF: These attributes point from one element to another. The value of the IDREF attribute on the pointing element is the same as the value of the ID attribute on the pointed-to element.

```
<!-- Attribute id gives the ID of element A -->
<!ATTLIST A id ID #REQUIRED>
<!-- Attribute ref points to the ID of another element -->
<!ATTLIST A ref IDREF #IMPLIED>
```

ENTITY, ENTITIES. These attributes point to external data in the form of unparsed entities. For complete details, see the XML specification.

```
<!-- Attribute a points to a single unparsed entity -->
<!ATTLIST A a ENTITY #IMPLIED>
<!-- Attribute b points to multiple unparsed entities -->
<!ATTLIST A b ENTITIES #IMPLIED>
```

Notes

NMTOKEN, NMTOKENS. These attributes have single/multiple tokens as values.

```
<!ATTLIST A a NMTOKEN #IMPLIED>
```

```
<!ATTLIST A b NMTOKENS #IMPLIED>
```

Comments

DTDs can contain comments. Comments are delimited by `<!--` and `-->`.

For example:

```
<!-- This is a comment in an XML file. -->
```

14.3.2 Integrating CSS in our XML Document

CSS is cascading style sheet language of HTML which is used for setting/getting common tag used in HTML document. By using CSS we can develop static HTML page with some standard.

By integrating a CSS in xml, we can present the XML document. A file should be save with extension of .css and it can be integrated in xml document by using the following syntax:

```
<?xml-stylesheet type="text/css" href="name of the CSS file with extension">
```



Example: Illustrating CSS.

Vsa.css

```
id {
    color:blue;
    margin-left:20pt;
    top-margin:20pt;
    font-size:20pt;
    bgcolor:pink;
}
name {
color:blue;
    margin-left:20pt;
    top-margin:20pt;
    font-size:50pt;
    background-color:yellow;
}
type {
    margin-left:20pt;
        top-margin:20pt;
            font-size:20pt;
                background-color: red;
}
weight{ margin-left:20pt;
        top-margin:20pt;
```

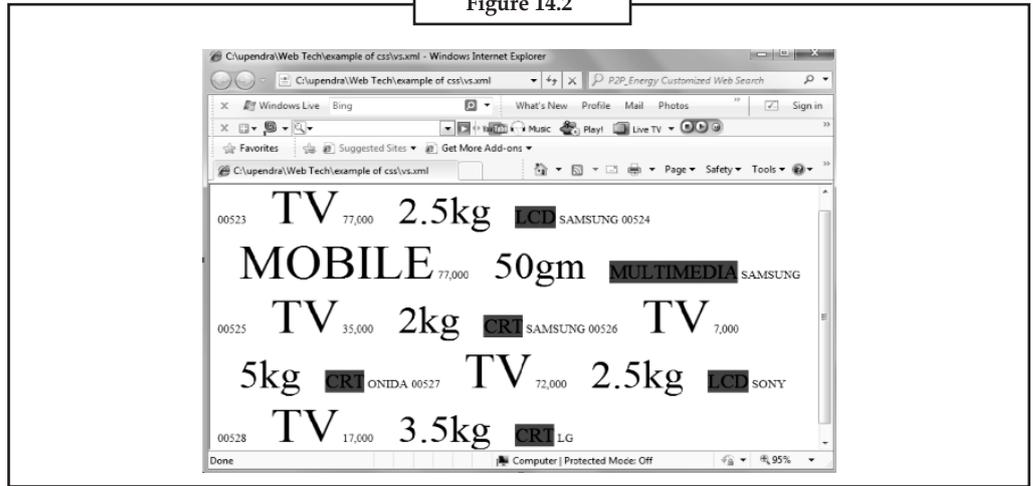
```
font-size:40pt;
background-color:yellow;
}
Vs.xml
<?xml version='1.0'?>
<?xml-stylesheet type="text/css" href="vsa.css"?>
<MyShop>
<Item>
<Identification>00523</Identification>
<name>TV</name>
<cost>77,000</cost>
<weight>2.5kg</weight>
<type>LCD</type>
<company>SAMSUNG</company>
<Identification>00524</Identification>
<name>MOBILE</name>
<cost>77,000</cost>
<weight>50gm</weight>
<type>MULTIMEDIA</type>
<company>SAMSUNG</company>
<Identification>00525</Identification>
<name>TV</name>
<cost>35,000</cost>
<weight>2kg</weight>
<type>CRT</type>
<company>SAMSUNG</company>
<Identification>00526</Identification>
<name>TV</name>
<cost>7,000</cost>
<weight>5kg</weight>
<type>CRT</type>
<company>ONIDA</company>
<Identification>00527</Identification>
<name>TV</name>
<cost>72,000</cost>
<weight>2.5kg</weight>
<type>LCD</type>
<company>SONY</company>
<Identification>00528</Identification>
```

```

Notes
      <name>TV</name>
      <cost>17,000</cost>
      <weight>3.5kg</weight>
      <type>CRT</type>
      <company>LG</company>
    </Item>
  </MyShop>
    
```

After running the program the output will be as shown in Figure 14.2.

Figure 14.2



Self Assessment

Fill in the blanks:

5. Schemas have their own mechanism for handling.....
6. Parameter entities are a different mechanism altogether; they allow portions of the XML document to be defined as to an element’s parameter list.
7. Attributes can supply values that are from the document.
8. CSS is cascading style sheet language of HTML which is used for setting/getting tag used in HTML document.
9. An XML document primarily consists of a strictly nested hierarchy of elements with a.....
10. Creating a is similar to creating a table in a database.

14.4 XML Parsers

Parser is responsible to make the XML data available to the applications through high level API.

Steps involved in xml parsing:

- Parser reads the XML document.
- Locates schema definition (ie DTD, XML Schema) provided in the XML document and read the schema definition and validate the declaration.

- Check for well firmness of XML document.
- If the parser is validating parser then validate the parser according to the schema, if not skip the step.
- Make the XML data available to the application.



Notes The complete set of API used for parsing an xml document is JAXP (Java API for XML Parsing).

14.4.1 Classification of XML Parsers

XML Parsers are classified into two part:

- DOM Parser [Document Object Model Parser]
- SAX Parser [Simple API for XML Parser]

The two parsers differ in their approach to parsing, with each parser having its strengths and weaknesses.

DOM Parser

DOM stands for the Document Object Model and it was specified by W3C. It is platform and language-neutral interface that allows programs to access and update the content, structure, and style of XML documents dynamically. DOM reads an XML document into memory and represents it as a tree; each node of the tree represents a particular piece of data from the original XML document. It is a tree based (Object based) DTD validator. The new version of DOM supports XML Schema validation also. DOM parser reads the entire XML document and loads the entire XML document into the active memory (in the form of object) and represents the objects into the application. It is a set of interfaces and classes used to model XML documents as a tree of objects called nodes. The DOM classes and interfaces represent XML documents, elements, attributes, text values etc. When an implementation of DOM parses an XML document, it reads the XML text value from some source, then builds an object graph, called a tree that mirrors the structure of the XML document. Every element in XML document is represented by a corresponding instance of the DOM Element type, attributes map to Attributes, comments to Comment, text to Text, CDATA sections to CDATASection and so on. When a DOM parser processes an XML document successfully, it returns an `org.w3c.dom.Document` object, which represents an XML document instance. The Document object provides access to the root Element, which in turn provides access to all the Elements, Text, Comments and other parts of the XML document.]



Task Analyze in a group of four that how the new version of DOM supports XML Schema validation?

- `javax.xml.parsers.DocumentBuilderFactory`
Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.
- `javax.xml.parsers.DocumentBuilder`

Defines the API to obtain DOM Document instances from an XML document. Using this class, an application programmer can obtain a Document from XML.

Notes

Interfaces in org.w3c.dom package

- Node
- NodeList
- Element
- Document

Limitation

- The memory required is more since the entire document has to be loaded into the memory
- Navigating through the elements requires a huge logic (i.e., as the path increases the number of conditions and loops increases to locate the required element)
- Searching for an element is complex



Example: Java application to check whether an XML document is a valid XML document or not?

DomValidator.java

```
import com.ibm.xml.parser.*;
import java.io.*;
class DOMValidator
{
public static void main(String args[]) throws Exception
    {
        Parser p=new Parser("My Errors");
        fis=new FileInputStream("Test.xml");
        p.readStream(fis);
        System.out.println("Test.xml can be parsed");
    }
}
```

Test.xml

```
?xml version="1.0"?>
<message>
<to server="hotmail.com" place="US"> you@hotmail.com</to>
<from>me@hotmail.com</from>
<subject>Hai</subject>
</message>
```

Before compilation of this program, set the classpath for xml4j_1_1_16.jar and after running the program get the output as shown in Figure 14.3.

Figure 14.3

Notes

```

C:\Windows\system32\cmd.exe
C:\upendra\webttech\xml programs>set classpath=c:\upendra\webttech\xml programs\xml4j_1_1_16.jar;
C:\upendra\webttech\xml programs>javac DOMValidator.java
C:\upendra\webttech\xml programs>java DOMValidator
Test.xml can be parsed
C:\upendra\webttech\xml programs>_

```



Example: Program to parse an XML Document which extracts the element names with data.

```

Dom1.java
import javax.xml.parsers.*;
import org.w3c.dom.*;
public class Dom1
{
public static void main(String args[]) throws Exception
{
// create an instance of DOM Parser
DocumentBuilderFactory factory=DocumentBuilderFactory.newInstance();
DocumentBuilder parser=factory.newDocumentBuilder();
// Provide the parser with the file path and parse into a Document
Document doc=parser.parse("Shop.xml");
// obtain the root element of the document and output its name
Node rootElement = doc.getDocumentElement();
System.out.println(rootElement.getNodeName());
Thread.sleep(30000);
// Output the elements contained by the root element
NodeList nodeList=rootElement.getChildNodes();
for(int i=0;i<nodeList.getLength();i++)
{
Node node=nodeList.item(i);
if(node.getNodeType() == Node.ELEMENT_NODE)
{
String tagName= node.getNodeName();
System.out.println(tagName+" ->" +node.getTextContent());
Thread.sleep(3000);
NodeList nodeList2=node.getChildNodes();
for(int j=0;j<nodeList2.getLength();j++)
{

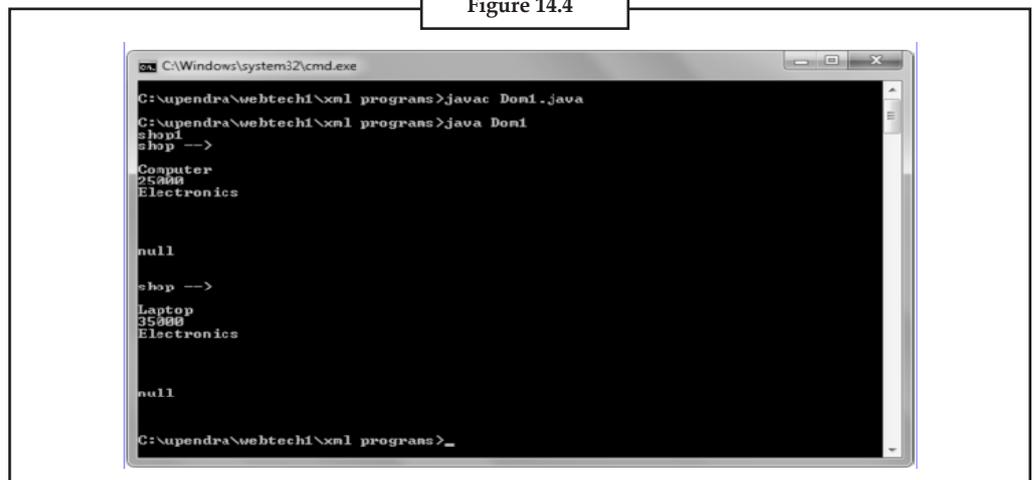
```

Notes

```
Node text=nodeList2.item(j);
System.out.println(text.getNodeValue());
Thread.sleep(3000);
}
}
}
}
}
Shop.xml
<shop1>
    <shop>
        <item itemid="1001">
            <name>Computer</name>
            <price>25000</price>
            <type>Electronics</type>
        </item>
    </shop>
    <shop>
        <item itemid="1001">
            <name>Laptop</name>
            <price>35000</price>
            <type>Electronics</type>
        </item>
    </shop>
</shop1>
```

Before compilation of this program, set the classpath for xml4j_1_1_16.jar and after running the program get the output as shown in Figure 14.4.

Figure 14.4





Example: Program to read XML data from the document using DOM Parser and creating a new XML document by adding new elements.

```
DomInsert.java

import org.w3c.dom.*;
import com.ibm.xml.parser.*;
import java.io.*;

class DOMInsertData
{
public static void main(String args[]) throws Exception
{
Parser p=new Parser("MyErrors");
TXDocument td=p.readStream(new FileInputStream("Shop.xml"));
//To create an Element Node
Element item= td.createElement("item");
// It will create an Element with Empty content and with a given name
Element name= td.createElement("name");
Element price= td.createElement("price");
Element type= td.createElement("type");
// To create a text node
Node name_text = td.createTextNode("TV");
Node price_text = td.createTextNode("45630");
Node type_text = td.createTextNode("Electronics");
// To add text to elements
name.appendChild(name_text);
price.appendChild(price_text);
type.appendChild(type_text);
//To add name, price,type in item
item.appendChild(name);
item.appendChild(price);
item.appendChild(type);
// To add Attribute
item.setAttribute("itemid", "1002");
// To add item into shop, the root element
Element shop= td.getDocumentElement();
shop.appendChild(item);
//To save the changes into a new file
td.printWithFormat(new FileWriter("shop2.xml"));
}
```

Notes

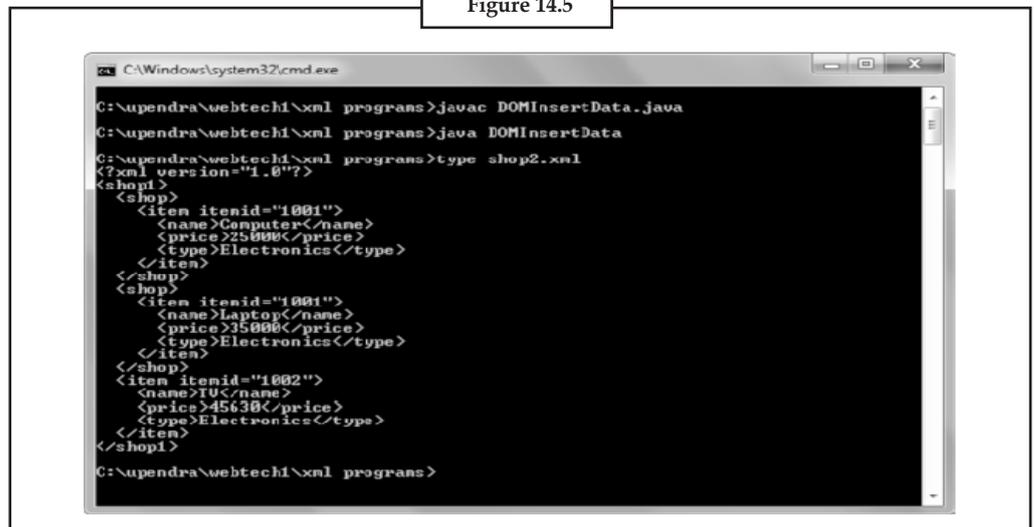
```

}
Shop.xml
<?xml version="1.0"?>
<shop1>
  <shop>
    <item itemid="1001">
      <name>Computer</name>
      <price>25000</price>
      <type>Electronics</type>
    </item>
  </shop>
</shop1>
  <shop>
    <item itemid="1001">
      <name>Laptop</name>
      <price>35000</price>
      <type>Electronics</type>
    </item>
  </shop>
</shop1>

```

To compile the above program set the classpath for xml4j_1_1_16.jar and after running the program you will get the output as shown in Figure 14.5.

Figure 14.5



14.5 XML Reader Class

Represents a reader that provides fast, non-cached, forward-only access to XML data.

Inheritance Hierarchy

System.Object

System.Xml.XmlReader

Notes

System.Xml.XmlDictionaryReader

Namespace: System.Xml

Assembly: System.Xml (in System.Xml.dll)

Syntax

C#

VB

[DefaultMemberAttribute("Item")]

```
public abstract class XmlReader : IDisposable
```

14.6 XML Writer Class

Represents a writer that provides a fast, non-cached, forward-only means of generating streams or files containing XML data.

Inheritance Hierarchy

System.Object

System.Xml.XmlWriter

System.Xml.XmlDictionaryWriter

Namespace: System.Xml

Assembly: System.Xml (in System.Xml.dll)

Syntax

C#

VB

```
public abstract class XmlWriter : IDisposable
```

Self Assessment

Fill in the blanks:

11. XML is the most common tool for between all sorts of applications.
12. The DOM classes and represent XML documents, elements, attributes, text values etc.
13. reads an XML document into memory and represents it as a tree; each node of the tree represents a particular piece of data from the original XML document.
14. The Document object provides access to the root Element, which in turn provides to all the Elements, Text, Comments and other parts of the XML document.
15. DOM parser reads the entire XML document and loads the entire XML document into the (in the form of object) and represents the objects into the application.

Notes



Caselet

All about 'X' in the Markup Language

LIKE the legendary doubles of Saddam, XML looks very similar to HTML, the traditional markup language of Web pages. Though both consist of hierarchically-nested fields, are just as easy to read and are portable, there are differences.

While "HTML contains titles, headings and italics, XML can contain customers, order numbers, prices or any data element you need". XML is fully extensible so you can add new tags and new elements to support your application. A core reference book by R. Allen Wyke is "XML Programming" from Microsoft. A few snatches:

HTML comes bound with a set of semantics and does not provide arbitrary structure. SGML (Standard Generalised Markup Language) provides arbitrary structure, but implementing SGML is too difficult for a Web browser to do on its own.

XML specifies neither semantics nor a tag set. It is a metalanguage for describing markup languages and provides a facility for defining tags and the structural relationship between them.

XML becomes the glue that binds what a piece of data actually is to what it is supposed to accomplish. It is used to describe all aspects of the data, ranging from near-physical properties to usage instructions, and its relationship to other data.

This information can be used for human and machine-readable purposes, one of the true advantages of XML.

XML is made of entities and parsed or unparsed data. Entities are a single character construct or a collection of named constructs that are referenced in the document.

Parsed data is made up of character data or markup and is processed by an XML processor. Unparsed data, on the other hand, is raw text not processed as XML.

Web Services are fantastic, but when do you not want to use them? Unfortunately, Web Services do have a few limitations. All Web Service objects are passed by value through SOAP. Pass-by-reference is not supported.

A certain amount of overhead is associated with calls to Web Services. Objects need to be marshalled into (potentially large) XML SOAP documents and return results need to be unmarshalled.

To retrieve and store data-centric XML documents, most RDBMSs provide a mapping mechanism to allow the transformation of relational data to and from XML data.

Broadly speaking, the different mapping mechanisms provided can be categorised as two types: result-based mapping and schema-based mapping. These two allow one to dynamically discover and represent the database structures from and into XML documents.

Explore the extensible language.

From VB to VB

WHAT are the differences between VB 6 and Visual Basic .NET? How to upgrade and what are the common upgrade problems? These and other questions are answered by Ed Robinson in the book "Upgrading Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET". A few picks:

Contd...

Visual Basic .NET, the latest version of Visual Basic, is not merely Visual Basic 6 with a few new features added on. Instead, VB has been thoroughly redesigned and restructured.

The language has been modernised, with new, richer object models for data, forms, transactions, and almost everything else. The file formats have also changed.

In Visual Basic 6, the memory associated with a variable or object is freed as soon as you set the variable to Nothing or the variable falls out of scope.

This is not true in the .NET. The .NET runtime marks the variable or object as needing deletion and relegates the object to the Garbage Collector (GC). The GC then deletes the object at some arbitrary time in the future.

A complete upgrade is a purebred. It entails not only upgrading code to Visual Basic .NET but also upgrading all of the technologies to their .NET equivalents.

An application that qualifies as a complete (or fully managed) upgrade does not use any COM objects; it uses only the .NET framework. However, interoperability is a good middle-of-the-road option. It usually means that your application has been upgraded to Visual Basic .NET, while core technologies remain in COM components.

Debugging in Visual Basic .NET is not exactly what traditional VB developers are used to. Probably the most surprising missing feature is Edit And Continue.

However, Visual Basic .NET offers a host of new features that greatly enhance the debugging experience, including service debugging, cross language debugging, XCopy deployment, structured exception handling, and a more sophisticated debugging API.

One characteristic that the Visual Basic .NET Object type lacks compared to the VB 6 Variant type is a value that means missing. Other than being set to an actual value, the only other state a Visual Basic .NET Object type can be set to is Nothing.

Cling on to the .NET.

I, me and 'My'

You already know My Computer, My Documents and so on. Now comes My Services from Microsoft .NET for a set of XML Message Interfaces (XMI) delivered as part of the .NET initiative. The book titled "Microsoft .NET My Services Specification" is about the details of the messaging interface model, data manipulation language, security authorisation and so on. Read on:

.NET My Services gives the end user a "digital safe-deposit box" for placing their personal information. By default, only end users have access to their information, which they can access via mobile devices while online and via mobile-device synchronisation for access while offline. The end users can choose to grant access to pieces of their personal data to various entities, such as friends, family, groups they are affiliated with, and businesses.

Content queries are relatively simple. The content query message selects a node-set and those nodes are returned in the query response message. Change queries are a little more complex.

Because nothing is shared unless .NET My Services is told to share, .NET My Services must be informed that sharing is desired. .NET My Services must be told the following items: Who to share information with? What information to share? How to share that information?

An overly complex set of inputs will result in users mis-configuring the system and in effect sharing more information with the wrong people, sharing the intended information with the wrong applications, sharing the correct information in the wrong way.

Contd...

Notes

The .NET categories service manages a list of category definitions. Examples of category definitions include child, anniversary, employee and so on. Each category definition has a human readable name and a description, which contains hints about the meaning of that category.

The .NET lists service breaks down a list into two major components. The list is defined by the list element, and an item defined by the item attributes. One interesting aspect of an item is that an item can be part of multiple lists. This allows applications to build and manage real-world lists. Suppose for instance you have a list of things to buy at the mall, and a list of things to do for an upcoming trip. It is very reasonable to have an item like "buy new shoes" appear on both of these lists.

14.7 Summary

- Extensible Markup Language, is a markup language that you can use to create your own tags.
- A DTD defines the structure of the content of an XML document, thereby allowing us to store the data in a consistent format.
- By integrating a CSS in xml, we can present the XML document.
- The Extensible Style sheet Language (XSL) is a W3C consortium standard for describing presentation rules that applies to XML document.
- Parser is responsible to make the XML data available to the applications through high level API.
- XML Schema is used to declare and define attributes for elements.

14.8 Keywords

DOM: Document Object Model

HTML: Hypertext Markup Language

SGML: Standard Generalized Markup Language

XML: Extensible Markup Language

14.9 Review Questions

1. What is XML? What are the advantages of XML?
2. Differentiate between the following:
 - (a) HTML and XML
 - (b) CSS and XSL
 - (c) DTD and XML Schema
 - (d) DOM and SAX
3. What do you mean by XML schema? Briefly explain.
4. Give a few examples of types of applications that can benefit from XML.
5. Why is XML such an important development?
6. How do we read XML data into the server using ASP?

7. The XML we are loading is being cached by my browser, how can we stop this?
8. How do we remove the whitespace XML nodes?
9. How to generate xml file with only some elements of another xml file using xsl file?
10. Is it simple to set up and use XML Socket Servers?
11. How do we debug connectivity problems between flash and the XML script?

Notes

Answers: Self Assessment

- | | |
|------------------------|----------------|
| 1. Web | 2. root |
| 3. application | 4. networks |
| 5. entities | 6. shortcuts |
| 7. missing | 8. common |
| 9. single root | 10. DTD |
| 11. data transmissions | 12. interfaces |
| 13. DOM | 14. Access |
| 15. active memory | |

14.10 Further Readings

*Online links*<http://www.w3.org/XML/><http://en.wikipedia.org/wiki/XML>

LOVELY PROFESSIONAL UNIVERSITY

Jalandhar-Delhi G.T. Road (NH-1)

Phagwara, Punjab (India)-144411

For Enquiry: +91-1824-521360

Fax.: +91-1824-506111

Email: odl@lpu.co.in