

Building Applications Using C#

Are you registered with
Onlinevarsity.com?

Yes



No



Did you download this book
from **Onlinevarsity.com**?

Yes



No



Scores

- If Yes 50 marks each. If No 0 marks each.
- If you have **scored 100 marks**, you have a **legal copy of this ebook**.
- If you have **scored <100**, you have a **pirated copy of this ebook**.

Building Applications Using C#

Learner's Guide

© 2013 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2013



Dear Learner,

We congratulate you on your decision to pursue an Aptech Worldwide course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc.* These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

➤ Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of Web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

➤ Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

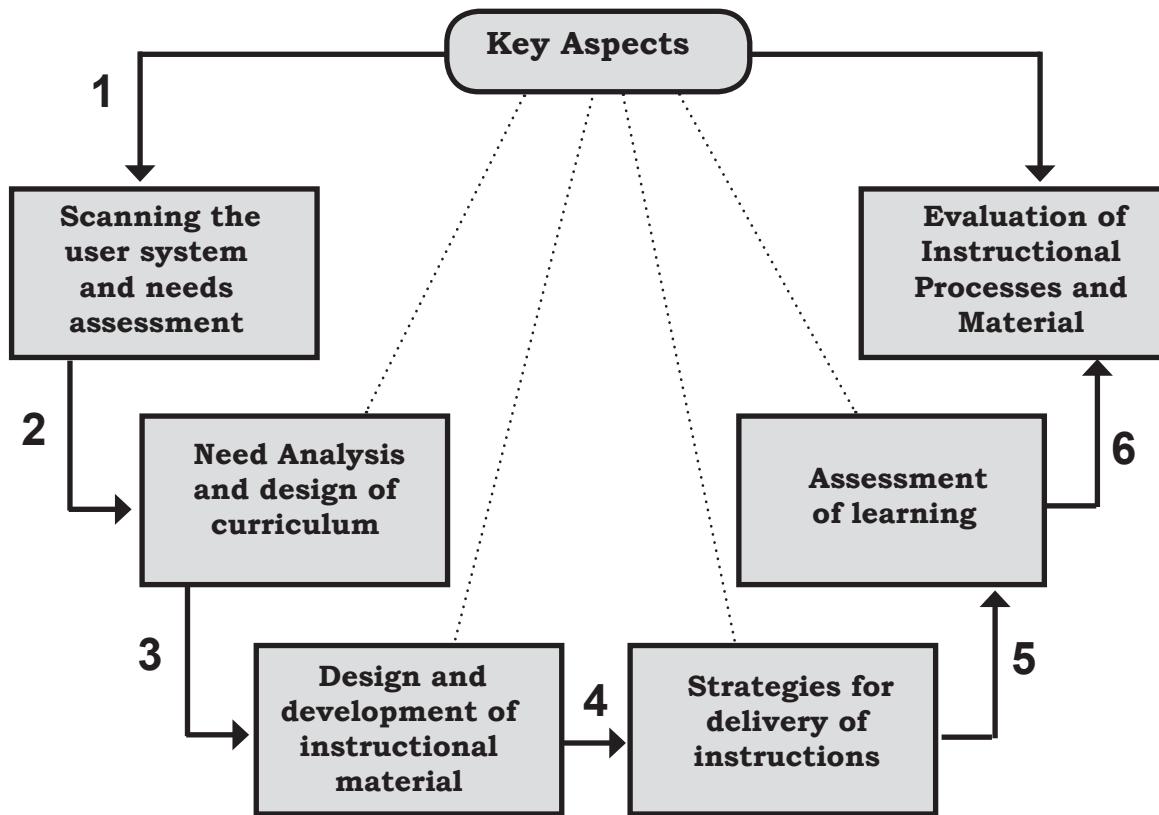
➤ Evaluation of instructional process and instructional materials

The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model



Tutor Contributed Value-ads For Students



For further reading, logon to

Preface

The book, **Building Applications Using C#**, covers the features of the C# language. C# is a language that enables programmers in quickly building solutions for the Microsoft .NET platform. The book begins by introducing .NET Framework 4.5, describing the basic features of C#, and then, explaining the object-oriented capabilities of C#. The book also describes the Visual Studio 2012 Integrated Development Environment (IDE).

The book explains various advanced features of C# such as delegates, query expressions, advanced types such as partial types, nullable types, and so on. The book also describes parallel programming and enforcing data security through encryption.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

Onlinevarsity

your e-way to learning

BI

g

Balanced Learner-Oriented Guide

for enriched learning available



Table of Contents

Sessions

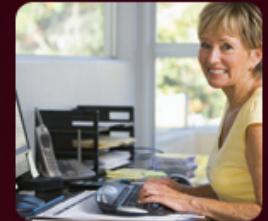
1. Getting Started with C#
2. Variables and Data Types
3. Statements and Operators
4. C# Programming Constructs
5. Arrays
6. Classes and Methods
7. Inheritance and Polymorphism
8. Abstract Classes and Interfaces
9. Properties and Indexers
10. Namespaces
11. Exception Handling
12. Events, Delegates, and Collections
13. Generics and Iterators
14. Advanced Methods and Types
15. Advanced Concepts in C#
16. Encrypting and Decrypting Data

ASK to Learn

What
Why
Where
Questions
When
How

The EXPERTS

are here to **HELP**



Session -1

Getting Started with C#

Welcome to the Session, **Getting Started with C#**.

This session provides an overview of the C# language. C# was created while building the Microsoft .NET Framework which is a software framework designed to run on Windows operating system. The Microsoft .NET Framework contains a large set of utilities that manage the execution of programs written specifically for the framework. The session describes the Microsoft .NET Framework, various language features of C#, and also covers the features and functionality of Visual Studio 2012, which is an Integrated Development Environment (IDE).

In this session, you will learn to:

- ➔ Define and describe the .NET Framework
- ➔ Explain the C# language features
- ➔ Define and describe the Visual Studio 2012 environment
- ➔ Explain the elements of Microsoft Visual Studio 2012 IDE

1.1 Introduction to .NET Framework

The .NET Framework is an infrastructure that enables building, deploying, and running different types of applications and services using .NET technologies. You can use the .NET Framework to minimize software development, deployment, and versioning conflicts.

1.1.1 The .NET Framework Architecture

With improvements in networking technology, distributed computing has provided the most effective use of processing power of both client and server processors. Also, with the emergence of Internet, applications became platform-independent, which ensured that they could be run on PCs with different hardware and software combination. Similarly, with the transformation in application development, it became possible for the clients and servers to communicate with each other in a vendor-independent manner.

Figure 1.1 shows the different features accompanying the transformation in computing, Internet, and application development.

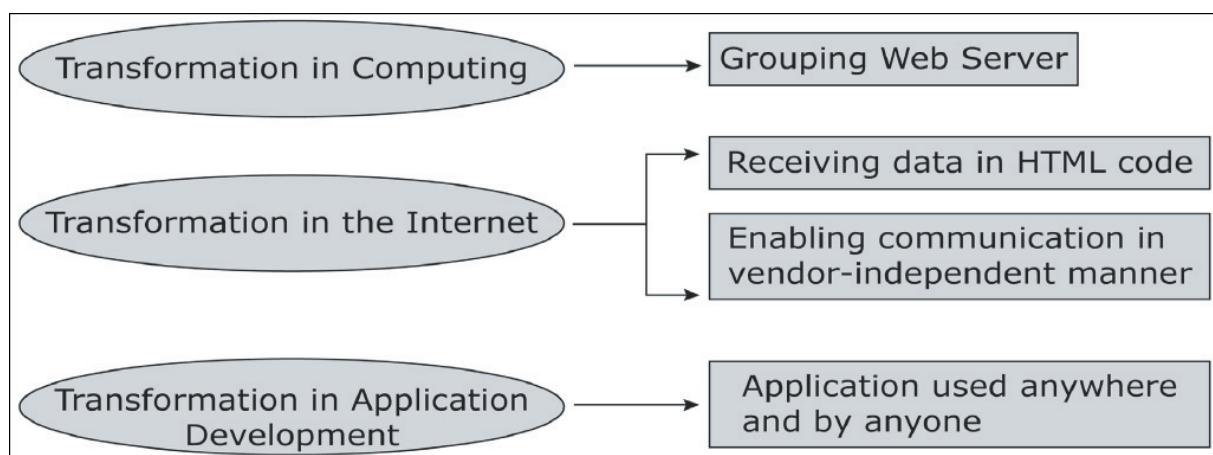


Figure 1.1: Transformations in Computing, Internet, and Application Development

All these transformations are supported by the technology platform introduced by Microsoft called as .NET Framework. Data stored using the .NET Framework is accessible to a user from any place, at any time, through any .NET compatible device.

The .NET Framework is a programming platform that is used for developing Windows, Web-based, and mobile software. It has a number of pre-coded solutions that manage the execution of programs written specifically for the framework.

The .NET Framework platform is based on two basic technologies for communication of data:

- eXtensible Markup Language (XML)
- The suite of Internet protocols

The key features of XML are as follows:

- It separates actual data from presentation.
- It unlocks information that can be organized, programmed, and edited.
- It allows Web sites to collaborate and provide groups of Web services. Thus, they can interact with each other.
- It provides a way for data to be distributed to a variety of devices.

Apart from XML, the .NET platform is also built on Internet protocols such as Hypertext Transfer Protocol (HTTP), Open Data Protocol (OData), and Simple Object Access Protocol (SOAP).

In traditional Windows applications, codes were directly compiled into the executable native code of the operating system. However, using the .NET Framework, the code of a program is compiled into CIL (formerly called MSIL) and stored in a file called assembly. This assembly is then compiled by the Common Language Runtime (CLR) to the native code at run-time.

Figure 1.2 represents the process of conversion of CIL code to the native code.

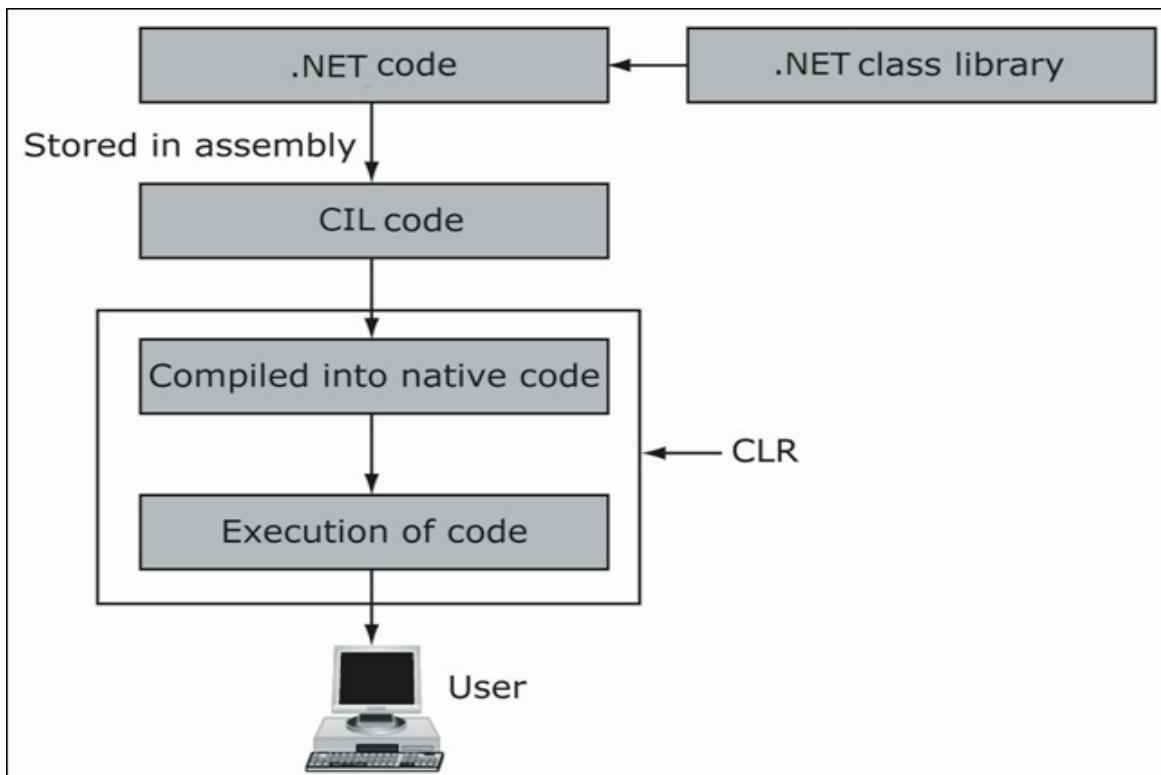


Figure 1.2: Process of Conversion of CIL Code to the Native Code

The CLR provides many features such as memory management, code execution, error handling, code safety verification, and garbage collection. Thus, the applications that run under the CLR are called managed code.

Microsoft has released different versions of the .NET Framework to include additional capabilities and functionalities with every newer version. These versions of the .NET Framework are as follows:

- **.NET Framework 1.0:** This is the first version released with Microsoft Visual Studio .NET 2002. It includes CLR, class libraries of .NET Framework and ASP. NET a development platform used to build Web pages.
- **.NET Framework 1.1:** This is first upgraded version released with Microsoft Visual Studio .NET 2003. It was incorporated with Microsoft Windows Server 2003 and included the following features:
 - Supports components used to create applications for mobiles as a part of the framework
 - Supports Oracle databases as a repository to store information in tables
 - Supports IPv6 protocol and Code Access Security (CAS) for Web-based applications
 - Enables running assemblies of Windows Forms from a Web site
 - Introduces .NET Compact Framework which provides components to create applications to be used in mobile phones and PDAs
- **.NET Framework 2.0:** This is the successor to .NET Framework 1.1 and next upgraded version included with Microsoft Visual Studio .NET 2005 and Microsoft SQL Server 2005. The version includes the following new features:
 - Support for 64-bit hardware platforms
 - Support for Generic data structures
 - Support for new Web controls used to design Web applications
 - Exposure to .NET Micro Framework which allows developers to create graphical devices in C#
- **.NET Framework 3.0:** This is built on .NET Framework 2.0 and is included with Visual Studio 2005 with .NET Framework 3.0 support. This version introduced many new technologies such as Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF), and Windows CardSpace.
- **.NET Framework 3.5:** This is the next upgraded version and is included with Visual Studio .NET 2008. The primary features of this release are support to develop AJAX-enabled Web sites and a new technology named Language Integrated Query (LINQ). The .NET Framework 3.5 Service Pack 1 was the next intermediate release in which the ADO.NET Entity Framework and ADO.NET Data Services technologies were introduced.
- **.NET Framework 4.0:** This version included with Visual Studio .NET 2010 introduced several new features, the key feature being the Dynamic Language Runtime (DLR). The DLR is a run-time

environment that enables .NET programmers to create applications using dynamic languages like Python and Ruby. Also, .NET Framework 4.0 introduced support for parallel computing that utilizes multi-core capabilities of computers. In addition, this version provides improvement in ADO.NET, WCF, and WPF, and introduces new language features, such as dynamic dispatch, named parameters, and optional parameters.

- **.NET Framework 4.5:** This version included with Visual Studio .NET 2012 provides enhancements to .NET Framework 4.0, such as enhancement in asynchronous programming through the `async` and `await` keywords, support for Zip compression, support for `regex` timeout, and more efficient garbage collection.

Table 1.1 summarizes the evolution of .NET versions.

Year	.NET Framework	Distributed with OS	IDE Name
2002	1.0		Visual Studio .NET (2002)
2003	1.1	Windows Server 2003	Visual Studio .NET 2003
2005	2.0		Visual Studio 2005
2006	3.0	Windows Vista, Windows Server 2008	Visual Studio 2005 with .NET Framework 3.0 support
2007	3.5	Windows 7, Windows Server 2008 R2	Visual Studio 2008
2010	4		Visual Studio 2010
2012	4.5	Windows 8, Windows Server 2012	Visual Studio 2012

Table 1.1: Versions of .NET Framework and Visual Studio

Note - CAS is a security mechanism provided by Microsoft to ensure that only the code trusted by .NET Framework is allowed to perform critical actions such as requesting memory allocation and accessing the database.

IPv6 stands for Internet Protocol version 6. It is a protocol that overcomes the shortage of IP addresses by supporting 5 X 1028 addresses.

1.1.2 .NET Framework Fundamentals

The .NET Framework is an essential Windows component for building and running the next generation of software applications and XML Web services.

The .NET Framework is designed to:

- Provide consistent object-oriented programming environment
- Minimize software deployment and versioning conflicts by providing a code-execution environment

- Promote safe execution of code by providing a code-execution environment
- Provide a consistent developer experience across varying types of applications such as Windows-based applications and Web-based applications

Note - The .NET Framework is a software component that can be added to the Microsoft Windows operating system. It has a number of pre-coded solutions and it manages the execution of programs written specifically for the framework.

1.1.3 .NET Framework Components

The .NET Framework is made up of several components. The two core components of the .NET Framework which are integral to any application or service development are the Common Language Runtime (CLR) and the .NET Framework class library.

→ The CLR

The CLR is the backbone of .NET Framework. It performs various functions such as:

- Memory management
- Code execution
- Error handling
- Code safety verification
- Garbage collection

→ The .NET Framework Class Library (FCL)

The class library is a comprehensive object-oriented collection of reusable types. It is used to develop applications ranging from traditional command-line to Graphical User Interface (GUI) applications that can be used on the Web.

Note - One of the major goals of the .NET Framework is to promote and facilitate code reusability.

1.1.4 Using .NET Framework

A programmer can develop applications using one of the languages supported by .NET. These applications make use of the base class libraries provided by the .NET Framework. For example, to display a text message on the screen, the following command can be used.

```
System.Console.WriteLine(".NET Architecture");
```

The same `WriteLine()` method will be used across all .NET languages. This has been made possible by making the Framework Class Library as a common class library for all .NET languages.

1.1.5 Other Components of .NET Framework

CLR and FCL are major components of the .NET Framework. Apart from these, some of the other important components are defined as follows:

→ **Common Language Specification (CLS)**

These are a set of rules that any .NET language should follow to create applications that are interoperable with other languages.

→ **Common Type System (CTS)**

Describes how data types are declared, used, and managed in the run-time and facilitates the use of types across various languages.

→ **Base Framework Classes**

These classes provide basic functionality such as input/output, string manipulation, security management, network communication and so on.

→ **ASP.NET**

Provides a set of classes to build Web applications. ASP.NET Web applications can be built using Web Forms, which is a set of classes to design forms for the Web pages similar to HyperText Markup Language (HTML). ASP.NET also supports Web services that can be accessed using a standard set of protocols.

→ **ADO.NET**

Provides classes to interact with databases.

→ **WPF**

This is a UI framework based on XML and vector graphics. WPF uses 3D computer graphics hardware and Direct 3D technologies to create desktop applications with rich UI on the Windows platform.

→ **WCF**

This is a service-oriented messaging framework. WCF allows creating service endpoints and allows programs to asynchronously send and receive data from the service endpoint.

→ **LINQ**

This is a component that provides data querying capabilities to a .NET application.

→ **ADO.NET Entity Framework**

This is a set of technologies built upon ADO.NET that enables creating data-centric applications in

object-oriented manner.

→ **Parallel LINQ**

This is a set of classes to support parallel programming using LINQ.

→ **Task Parallel Library**

This is a library that simplifies parallel and concurrent programming in a .NET application.

Figure 1.3 displays the various components of .NET framework.

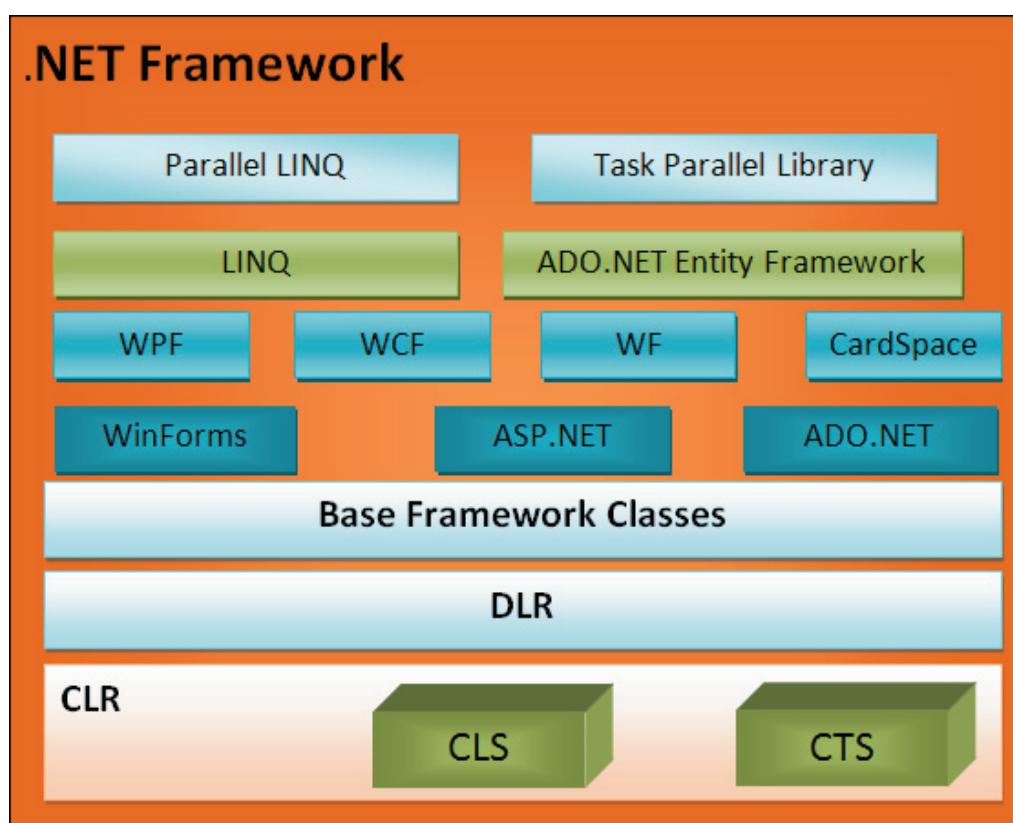


Figure 1.3: Components of .NET Framework

1.1.6 Common Intermediate Language (CIL)

Every .NET programming language generally has a compiler and a run-time environment of its own. The compiler converts the source code into executable code that can be run by the users.

One of the primary goals of .NET Framework is to combine the run-time environments so that the developers can work with a single set of run-time services.

When the code written in a .NET compatible language such as C# or VB is compiled, the output code is in the form of MSIL code. MSIL is composed of a specific set of instructions that indicate how the code should be executed.

Figure 1.4 depicts the concept of Microsoft Intermediate Language. MSIL is now called as CIL.

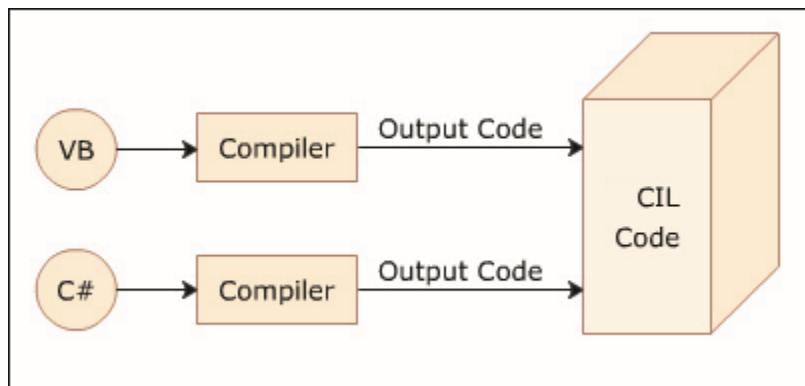


Figure 1.4: Common Intermediate Language

1.1.7 Common Language Runtime (CLR)

The CLR is the foundation of the .NET Framework. The run-time manages code at execution time and performs operations such as memory management, thread management, and remoting. In simple terms, the CLR acts as an execution engine for the .NET Framework. It manages the execution of programs and provides a suitable environment for programs to run. The .NET Framework supports a number of development tools and language compilers in its Software Development Kit (SDK). Hence, the CLR provides a multi-language execution environment.

When a code is executed for the first time, the CIL code is converted to a code native to the operating system. This is done at run-time by the **Just-In-Time (JIT)** compiler present in the **CLR**. The CLR converts the CIL code to the machine language code. Once this is done, the code can be directly executed by the CPU. Figure 1.5 depicts the working of the CLR.

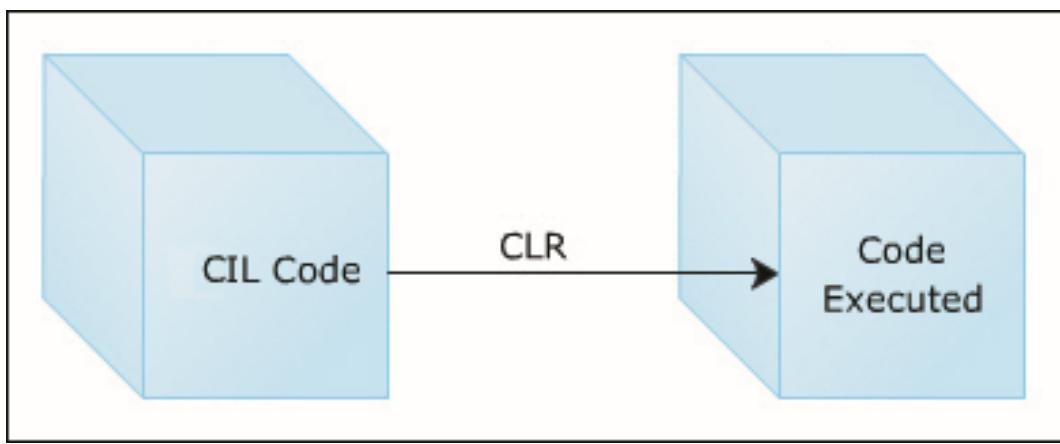


Figure 1.5: Common Language Runtime

Figure 1.6 shows a more detailed look at the working of the CLR.

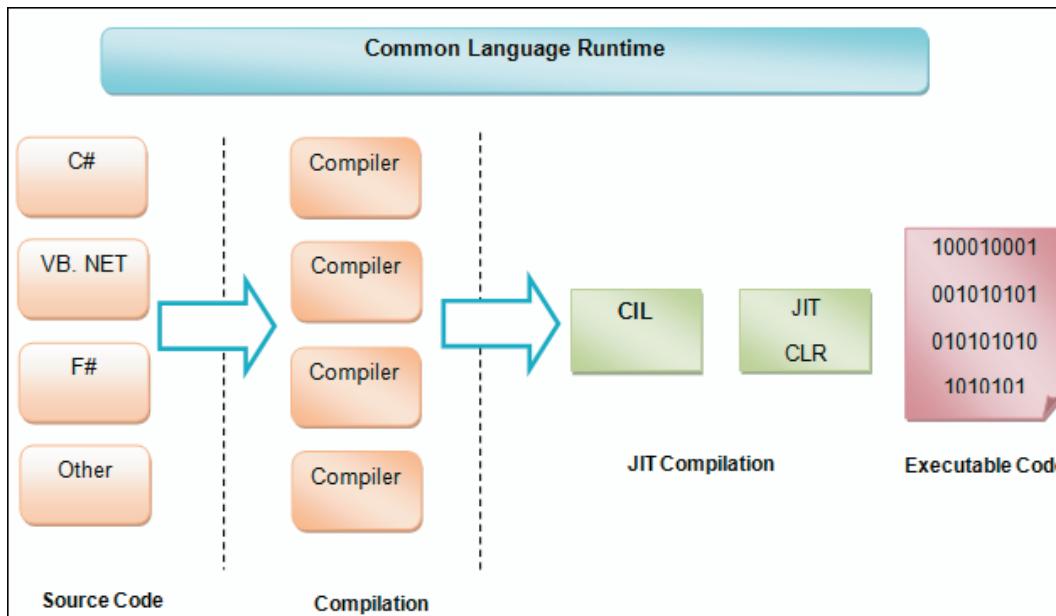


Figure 1.6: Working of the CLR

Note - All the code in .NET is managed by the CLR and is therefore, referred to as managed code. In managed code, developers allocate memory wherever and whenever required by declaring variables and the run-time garbage collector determines when the memory is no longer needed and cleans it all up. The garbage collector may also move memory around to improve efficiency. The run-time manages it all for you and hence, the term managed code is used for such programs. On the other hand, code that runs without the CLR, such as C programs, are called unmanaged code.

1.1.8 Dynamic Language Runtime (DLR)

DLR is a run-time environment built on top of the CLR to enable interoperability of dynamic languages such as Ruby and Python with the .NET Framework. The .NET Framework languages, such as C#, VB, and J# are statically typed languages, which means that the programmer needs to specify object types while developing a program. On the other hand, in dynamic languages, programmers are not required to specify object types in the development phase. DLR allows creating and porting dynamic languages to the .NET Framework. In addition, DLR provides dynamic features to the existing statically typed languages. For example, C# relies on the DLR to perform dynamic binding.

1.1.9 Need for a New Language

Microsoft introduced C# as a new programming language to address the problems posed by traditional languages.

C# was developed to provide the following benefits:

- ➔ Create a very simple and yet powerful tool for building interoperable, scalable, and robust applications.

- Create a complete object-oriented architecture.
- Support powerful component-oriented development.
- Allow access to many features previously available only in C++ while retaining the ease-of-use of a rapid application development tool such as Visual Basic.
- Provide familiarity to programmers coming from C or C++ background.
- Allow to write applications that target both desktop and mobile devices.

1.1.10 Purpose of C# Language

Microsoft .NET was formerly known as Next Generation Windows Services (NGWS). It is a completely new platform for developing the next generation of Windows/Web applications. These applications transcend device boundaries and fully harness the power of the Internet. However, building the new platform required a language that could take full advantage. This is one of the factors that led to the development of C#. C# is an object-oriented language derived from C and C++. The goal of C# is to provide a simple, efficient, productive, and object-oriented language that is familiar and yet at the same time revolutionary.

Note - C# has evolved from C/C++. Hence, it retains its family name. The # (hash symbol) in musical notations is used to refer to a sharp note and is called Sharp; hence, the name is pronounced as C Sharp.

1.2 Language Features

C# has features common to most object-oriented languages and in addition, it has language-specific features, such as type safety checking, generics, and indexers that make it the preferred language to create a wide variety of applications.

1.2.1 Basic Features of C#

C# is a programming language designed for building a wide range of applications that run on the .NET Framework. Some of its key features are as follows:

- **Object-oriented Programming:** C# application programming focuses on objects so that code written once can be reused. This helps reduce time and effort on the part of developers.
- **Type-safety Checking:** Uninitialized variables cannot be used in C#. Overflow of types can be checked. C# is a case-sensitive language.
- **Garbage Collection:** Performs automatic memory management from time to time and spares the programmer the task.

- **Standardization by European Computer Manufacturers Association (ECMA):** This standard specifies the syntax and constraints used to create standard C# programs.
- **Generic Types and Methods:** Generics are a type of data structure that contains code that remains the same throughout but the data type of the parameters can change with each use.
- **Iterators:** Enable looping (or iterations) on user-defined data types with the `foreach` loop.
- **Static Classes:** Contain only static members and do not require instantiation.
- **Partial Classes:** Allow the user to split a single class into multiple source code (.cs) files.
- **Anonymous Methods:** Enable the user to specify a small block of code within the delegate declaration.
- **Methods with Named Arguments:** Enable the user to associate a method argument with a name rather than its position in the argument list.
- **Methods with Optional Arguments:** Allow the user to define a method with an optional argument with a default value. The caller of the method may or may not pass the optional argument value during the method invocation.
- **Nullable Types:** Allow a variable to contain a value that is undefined.
- **Accessor Accessibility:** Allows the user to specify the accessibility levels of the `get` and `set` accessors.
- **Auto-implemented Properties:** Allow the user to create a property without explicitly providing the methods to get and set the value of the property.
- **Parallel Computing:** In .NET Framework and C#, there is strong support for parallel programming using which develop efficient, fine-grained, and scalable parallel code without working directly with threads or the thread pool.

1.2.2 Applications of C#

C# is an object-oriented language that can be used in a number of applications. Some of the applications are as follows:

- Web applications
- Web services

- Gaming applications
- Large-scale enterprise applications
- Mobile applications for pocket PCs, PDAs, and cell phones
- Simple standalone desktop applications such as Library Management Systems, Student Mark Sheet generation, and so on
- Complex distributed applications that can spread over a number of cities or countries
- Cloud applications

Note - The security features in-built into C# make it possible to provide safe and secure solutions for enterprises.

1.2.3 Advantages of C#

C# has become a preferred programming language over C++ because of its simplicity and user friendliness.

The advantages of C# are as follows:

→ Cross Language Support

The code written in any other .NET language can be easily used and integrated with C# applications.

→ Common Internet Protocols

.NET offers extensive support for XML, which is the preferred choice for formatting information over the Internet. Additionally, support for transfer via SOAP is also integrated.

→ Simple Deployment

Deployment of C# applications is made simple by the concept of assemblies. An assembly is a self-describing collection of code and resources. It specifies exactly the location and version of any other code it needs.

→ XML Documentation

Comments can be placed in XML format and can then be used as needed to document your code. This documentation can include example code, parameters, and references to other topics. It makes sense for a developer to document his or her code because those comments can actually become documentation independent of the source code.

Note - SOAP is a light weight protocol for information exchange.

1.2.4 Memory Management

In programming languages like C and C++, the allocation and de-allocation of memory is done manually. Performing these tasks manually is both, time-consuming and difficult.

The C# language provides the feature of allocating and releasing memory using automatic memory management. This means that there is no need to write code to allocate memory when objects are created or to release memory when objects are not required in the application. Automatic memory management increases the code quality and enhances the performance and the productivity.

1.2.5 Garbage Collection

The process of allocating and de-allocating memory using automatic memory management is done with the help of a garbage collector. Thus, garbage collection is the automatic reclaiming of memory from objects that are no longer in scope. This means that when the object is out of scope, the memory will be free for use so that other objects can be allotted the memory.

Figure 1.7 illustrates concept of garbage collection.

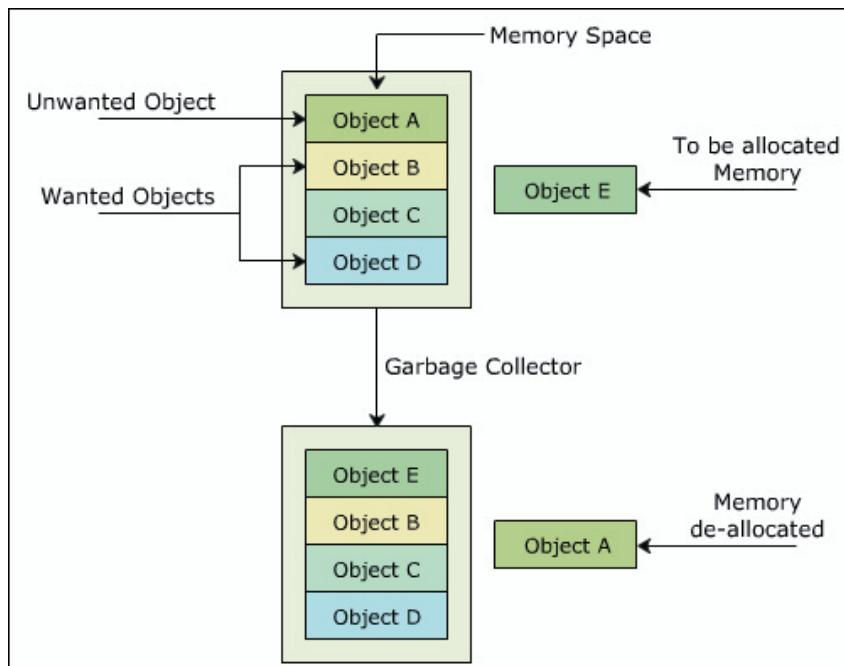


Figure 1.7: Garbage Collection

Note - Garbage collection process involves two steps:

1. Determine which objects in a program will not be accessed in the future
2. Reclaim the storage used by those objects

1.3 Visual Studio 2012 Environment

Visual Studio 2012 provides the environment to create, deploy, and run applications developed using the .NET framework. The Visual Studio 2012 environment comprises the Visual Studio Integrated Development Environment (IDE), which is a comprehensive set of tools, templates, and libraries required to create .NET framework applications.

1.3.1 Introduction to Visual Studio 2012

Visual Studio 2012 is a complete set of development tools for building high performance desktop applications, XML Web Services, mobile applications, and ASP Web applications. In addition, it is also used to simplify team-based design, development, and deployment of enterprise solutions.

Visual Studio 2012 is an IDE used to ease the development process of .NET applications such as Visual C# 2012 and Visual Basic 2012. The advantage of using Visual Studio is that for all the .NET compatible languages, the same IDE, debugger, Solution Explorer, Properties tab, Toolbox, standard menus, and toolbars are used. The following features of IDE make it useful for an easier development process:

- A single environment is provided for developing the .NET applications.
- There are several programming languages to choose from for developing applications.
- The IDE can be customized, based on the user's preferences.
- There is a built-in browser in the IDE that is used for browsing the Internet without launching another application.
- A program can be executed with or without a debugger.
- The application can be published over the Internet or onto a disk.
- The application provides Dynamic Help on a number of topics using the MSDN library.
- The syntax of the code is checked as the user is typing it and the appropriate error notification is provided in case an error is encountered.
- The IDE provides a standard code editor to write the .NET applications. When a keyword is used or a dot (.) is typed after objects or enumerations, the text editor has the ability to suggest options (methods or properties) that automatically completes the required text.
- The IDE has a set of visual designers that simplifies application developments. Commonly used visual designers are as follows:
 - **Windows Form Designer:** Allows programmers to design the layout of Windows forms and

drag and drop controls to it.

- **Web Designer:** Allows programmers to design the layout and visual elements of ASP.NET Web pages.
- **WPF Designer:** Allows programmers to create user interfaces targeting WPF.
- **Class Designer:** Allows programmers to use UML modeling to design classes, its members, and relationships between classes.
- **Data Designer:** Allows programmer to edit database schemas graphically.

→ The IDE has an integrated compiler to compile and execute the application. The user can either compile a single source file or the complete project.

Visual Studio 2012 provides multiple advantages in the development process. The primary advantages are as follows:

- Improved developer productivity
- Development of applications for Microsoft .NET Framework 4.5
- Development of plug-ins to extend the IDE's capabilities

1.3.2 Visual Studio 2012 Editions

The IDE of Microsoft Visual Studio is a result of extensive research by the Microsoft team.

The different editions of Visual Studio 2012 are as follows:

→ **Visual Studio Professional 2012**

This is the entry-level edition that provides support for developing and debugging applications, such as Web, desktop, cloud-based, and mobile applications.

→ **Visual Studio Professional 2012 with MSDN**

This edition provides all the features of the Visual Studio Professional 2012 edition along with an MSDN subscription. In addition, this edition includes Team Foundation Server and provides access to cloud, Windows Store, and Windows Phone Market place.

→ **Visual Studio Test Professional 2012 with MSDN**

This edition targets testers and Quality Assurance (QA) professionals by providing project management tools, testing tools, and virtual environment to perform application testing.

→ **Visual Studio Premium 2012 with MSDN**

This edition provides all the features of the combined Visual Studio Professional 2012 and Visual Studio Test Professional 2012 with MSDN editions. In addition, this edition supports peer code

review, User Interface (UI) validation through automated tests, and code coverage analysis to determine the amount of code being tested.

→ **Visual Studio Ultimate 2012 with MSDN**

This edition has all the features of the other editions. In addition, this edition supports designing architectural layer diagrams, performing Web performance and load testing, and analyzing diagnostic data collected from run-time systems.

Note - In addition to the commercial editions of Visual Studio 2012, Microsoft has released Microsoft Visual Studio Express 2012, which is a freeware primarily aimed at students and non-professionals. Visual Studio Express is a light weight version of the Visual Studio that provides an easy-to-learn IDE before starting professional development in Visual Studio.

1.3.3 Languages in Visual Studio 2012

Visual Studio 2012 supports multiple programming languages such as Visual Basic .NET, Visual C++, Visual C#, and Visual J#.

The classes and libraries used in the Visual Studio 2012 IDE are common for all the languages in Visual Studio 2012. This makes Visual Studio 2012 more flexible.

1.3.4 Features of Visual Studio 2012

Visual Studio 2012 provides a number of new and improved features. Some of these are as follows:

→ **Comprehensive Tools Platform**

In Visual Studio 2012, developers of all knowledge levels can make use of developer tools, which offer a development experience tailored for their unique needs.

→ **Reduced Development Complexity**

Visual Studio 2012 enables customers to deliver more easily a broad range of .NET Framework-based solutions including Windows, Office, Web, and mobile applications.

→ **Edit Marks**

Visual Studio 2012 provides a visual indication of the changes that are made and not saved and changes that are made during the current session that have been saved to the disk.

→ **Code Snippets**

Code Snippets are small units of C# source code that the developer can use quickly with the help of certain keystrokes.

→ **AutoRecover**

Visual Studio 2012 automatically saves the work on a regular basis and thus, minimizes loss of information due to unexpected closing of unsaved files. In case of an IDE crash, Visual Studio 2012 will also prompt you to recover your work after you restart.

→ **IntelliSense**

Visual Studio 2012 has the IntelliSense feature in which syntax tips, lists of methods, variables and classes pop up continually when entering code in the Code Editor, making the process of entering code more efficient.

→ **Refactoring**

Refactoring enables developers to automate common tasks when restructuring code. It changes the internal structure of the code, specifically the design of its objects, to make it more comprehensible, maintainable, and efficient without changing its behavior.

Note - Refactoring can save a great deal of time and reduce errors with editing and changing code. The results are not always perfect, but there are warnings displayed when the result will leave the code in an inconsistent state or when the refactoring operation is not possible.

1.4 *Elements of Microsoft Visual Studio 2012 IDE*

Visual Studio 2012 contains an extensive set of elements, comprising of editors, toolbox, and different windows to assist developers in creating .NET applications.

1.4.1 *Key Elements*

The key elements in Visual Studio 2012 IDE are as follows:

→ **Solution Explorer**

The Solution Explorer provides you with an organized view of your projects and their files. It also gives you ready access to the commands that pertain to these projects. Using the Solution Explorer, you can use the Visual Studio editors to work on files outside the context of a solution or project.

Figure 1.8 displays the snapshot of Solution Explorer.

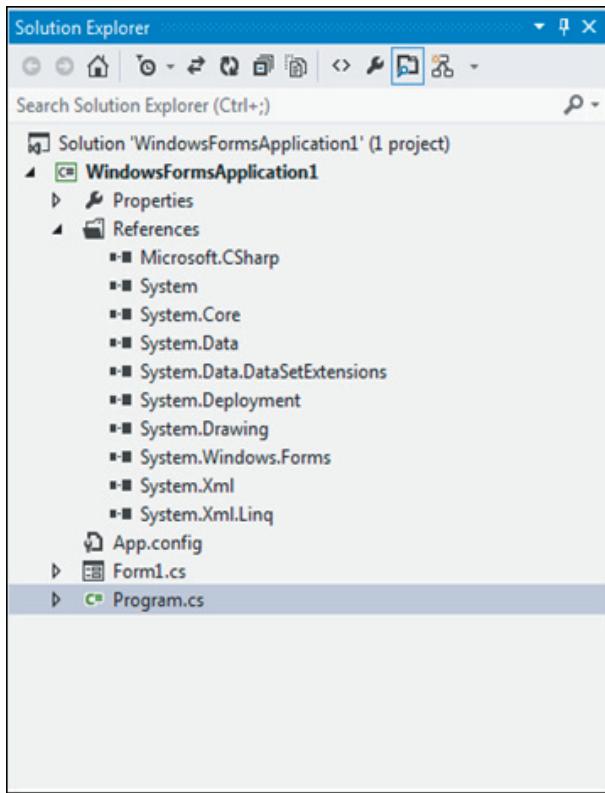


Figure 1.8: Solution Explorer

The reference node consists of the assemblies referenced in the current project. `Form1.cs` is the name of the source file.

When a code is written and saved, the .NET solution is saved in a `.sln` file, the C# project is saved in a `.csproj` file and the source code is saved in a `.cs` file.

→ Code Editor

The Code Editor is used to write, display and edit form, event and method code. You can open as many code windows as you want and easily copy and paste codes from one window to another.

Figure 1.9 displays the code editor.

The screenshot shows the Visual Studio Code Editor with the file `Form1.cs` open. The title bar indicates it is in Design mode. The code editor displays the following C# code:

```
Form1.cs*  X  Form1.cs [Design]*  
WindowsFormsApplication1.Form1  
Form1_Load(object sender, EventArgs e)  
using System;  
using System.Collections.Generic;  
using System.ComponentModel;  
using System.Data;  
using System.Drawing;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.Windows.Forms;  
  
namespace WindowsFormsApplication1  
{  
    public partial class Form1 : Form  
    {  
        public Form1()  
        {  
            InitializeComponent();  
        }  
  
        private void Form1_Load(object sender, EventArgs e)  
        {  
        }  
    }  
}
```

The `Form1_Load` event handler is currently selected, indicated by a yellow vertical bar on the left margin. The status bar at the bottom shows "100 %".

Figure 1.9: Code Editor

→ Properties Window

The Properties window is used to view and change the design-time properties and events of selected objects that are located in editors and designers. It displays different types of editing fields depending on the needs of a particular property. These fields include edit boxes, drop-down lists, and links to custom editor dialog box.

Figure 1.10 displays the properties window.

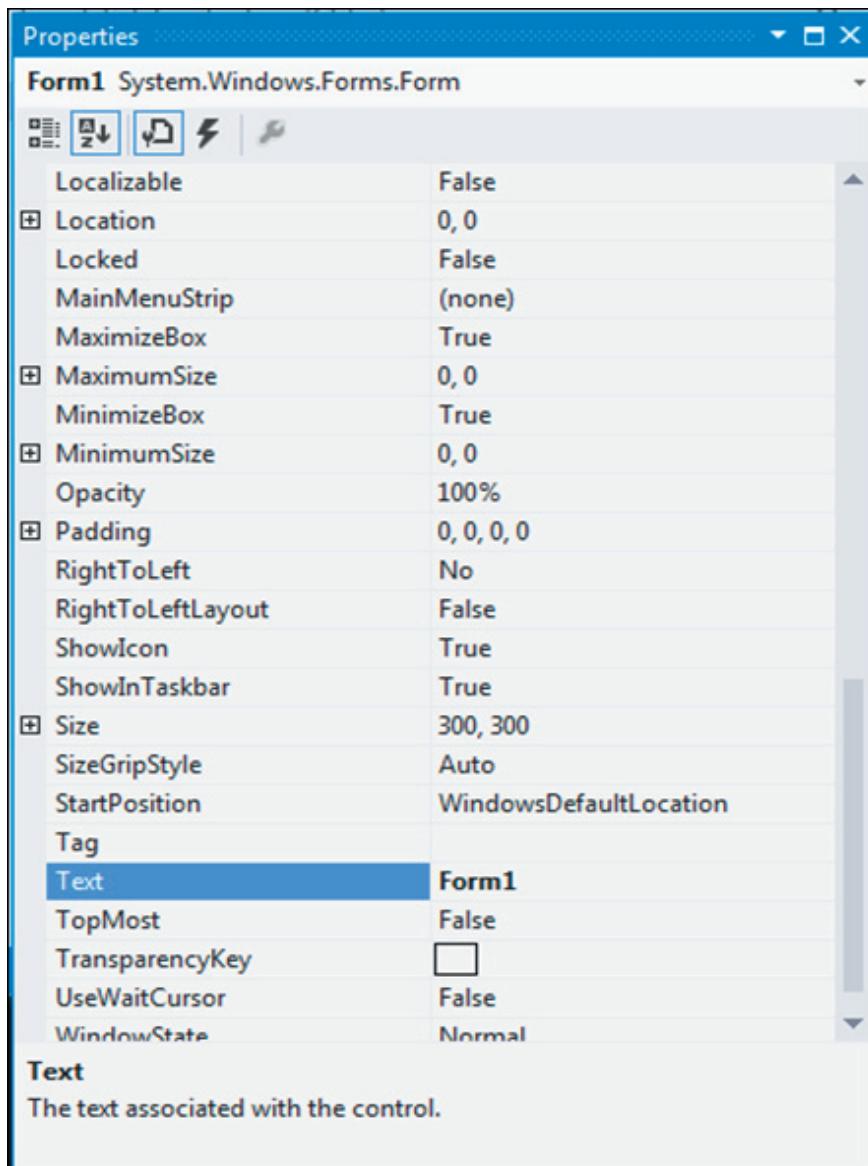


Figure 1.10: Properties Window

→ Toolbox

The Toolbox window displays the controls and components that can be added to the Design mode of the form. However, the contents of the Toolbox window change according to the type of form the user is creating or editing. For example, if the user is adding tools onto a Web form, the Toolbox displays the server controls, HTML controls, data controls, and other components that the Web form may require.

Figure 1.11 displays the Toolbox.

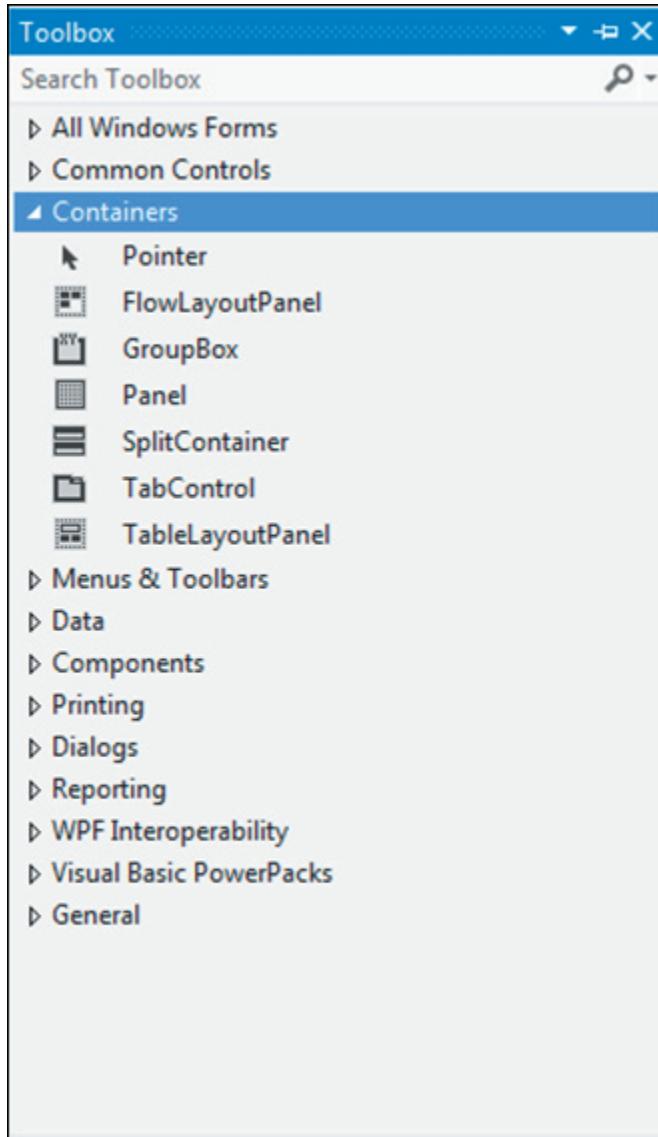


Figure 1.11: Toolbox

To use the controls or components from the Toolbox, the user can drag and drop the required control or component onto a form. However, if the user is creating or editing codes in the code editor, the Toolbox contains only a Clipboard Ring. This Clipboard Ring contains the last 20 items that have been cut or copied so that they can be pasted into the document, if necessary. To paste the text from the Clipboard Ring, click the text and drag it to the place where it is to be inserted.

→ Server Explorer

The Server Explorer is the server management console used for opening data connections, logging on to servers, exploring databases, and system services.

Using the Server Explorer, the user can perform the following activities:

- View and retrieve information from all of the databases that the user is connected to
- List database tables, views, stored procedures, and functions
- Expand individual tables to list their columns and triggers
- Open data connections to SQL and other databases
- Log on to servers and display their databases and system services such as event logs, message queries, and so on
- View information about available Web Services

Figure 1.12 displays the Server Explorer.

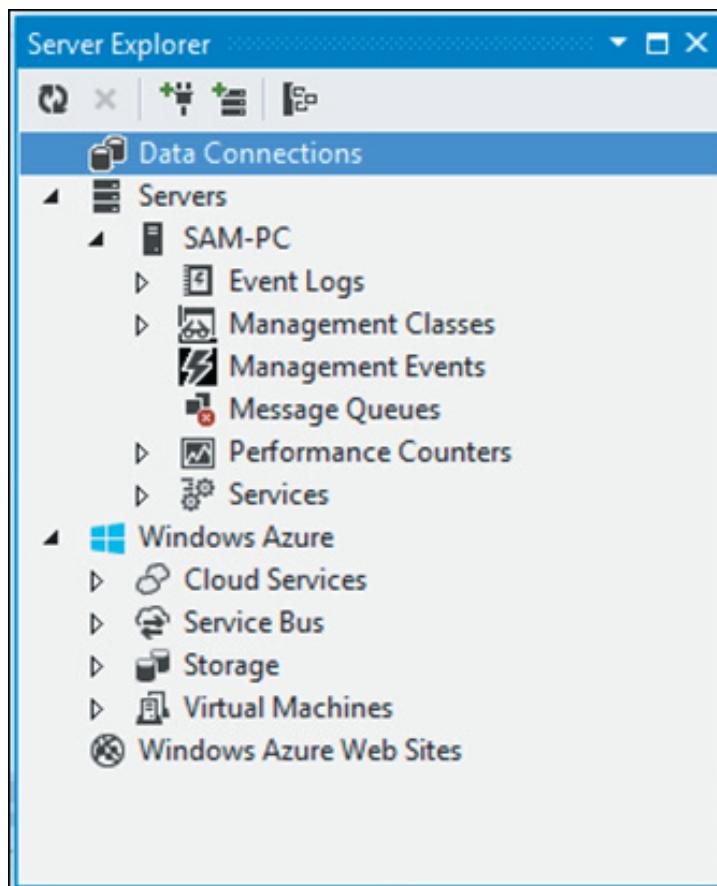


Figure 1.12: Server Explorer

→ Output Window

The Output window displays the status messages for the various functionalities in the IDE.

Figure 1.13 displays the Output window.

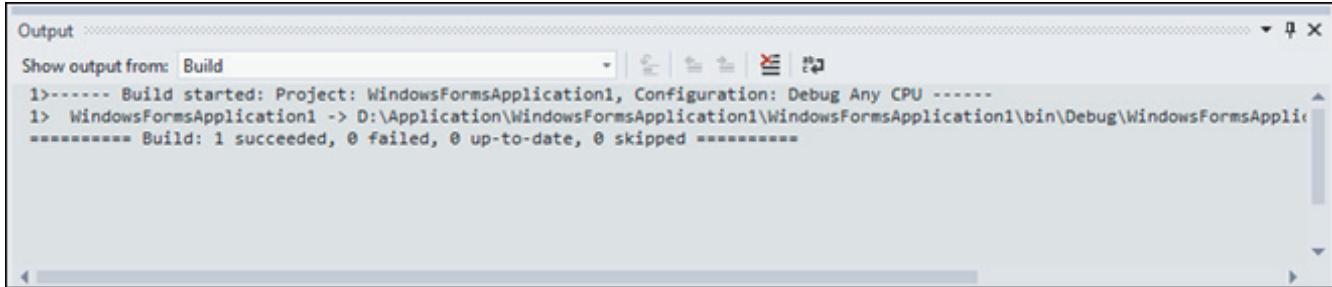


Figure 1.13: Output Window

→ Error List

With the help of the Error List, the speed of application development increases. The Error List window does the following:

- Displays the errors, warnings, and messages produced when the code is edited and compiled.
- Finds the syntactical errors noted by IntelliSense.
- Finds the deployment errors, certain static analysis errors, and errors detected while applying Enterprise Template policies.
- Filters which entries are displayed and which columns of information appear for each entry.
- When the error occurs, the user can find out the location of the error by double-clicking the error message in the Error List window.

Figure 1.14 displays the Error List window.

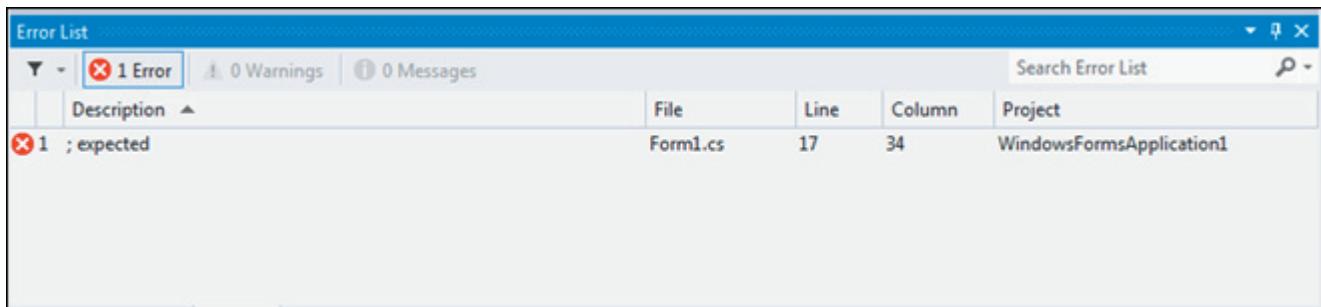


Figure 1.14: Error List Window

→ Dynamic Help

The Dynamic Help window provides a list of topics specific to the area of the IDE you are working in or the task you are working on.

1.4.2 csc Command

Console applications that are created in C# run in a console window. This window provides simple text-based output.

The **csc (C Sharp Compiler)** command can be used to compile a C# program. The steps to compile and execute a program are as follows:

- ➔ Create a New Project
1. Start Visual Studio 2012.
 2. Select **New → Project** from the **File** menu.
 3. Expand the Templates → Visual C# nodes in the left pane and select Console Application in the right pane of the **New Project** dialog box.
 4. Specify the name and location for the project and click **OK**. Visual Studio 2012 opens the Code Editor with the skeleton code of a class, as shown in Code Snippet 1:

Code Snippet 1:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace SampleProgram
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

5. Add the following code snippet after the opening curly brace of the `Main(string[] args)` method definition:

```
Console.WriteLine("This is a sample C# program");
```

→ Compile a C# Program

A C# program can be compiled using the following syntax:

```
csc <file.cs>
```

Example:

```
csc SampleProgram.cs
```

where,

SampleProgram: Specifies the name of the program to be compiled.

This command generates an executable file SampleProgram.exe.

→ Execute the Program

Open the Developer Command Prompt for VS2012, and browse to the directory that contains the .exe file. Then, type the file name at the command prompt.

Figure 1.15 displays the developer command prompt for VS2012 window.

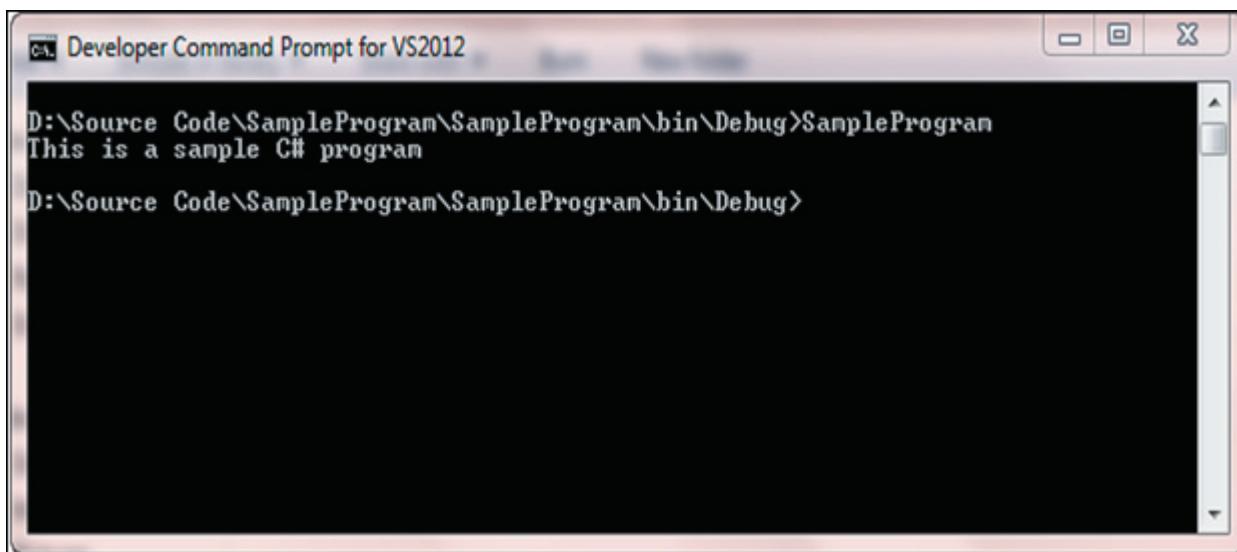


Figure 1.15: The Developer Command Prompt for VS 2012 Window

Note - The .exe file is known as **portable EXE** as it contains machine-independent instructions. The portable EXE works on any operating system that supports the .NET platform.

1.4.3 Build and Execute

In Visual Studio 2012, apart from the use of the `csc` command, the Integrated Development Environment provides the necessary support to compile and execute C# programs.

The steps involved are as follows:

→ **Compiling the C# Program**

Select **Build <application name>** from the **Build** menu. This action will create an executable file (.exe).

→ **Executing the Program**

From the **Debug** menu, select **Start Without Debugging**.

Figure 1.16 displays the output of the program.

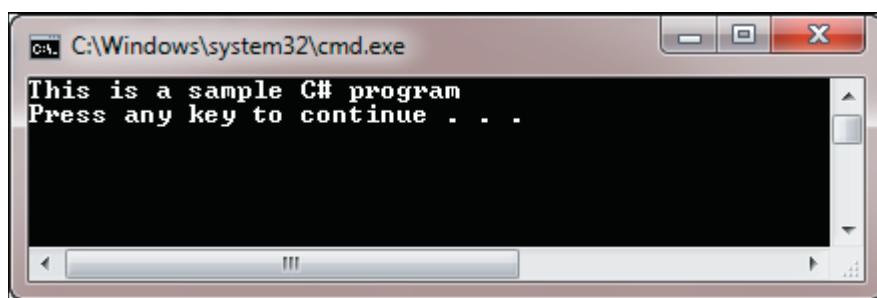


Figure 1.16: Output

1.5 Check Your Progress

1. Which of these statements about the components of the C# platform are true?

(A)	The .NET framework enables the use of C and C++ programming languages.
(B)	C# provides a complete object-oriented architecture.
(C)	C# allows you to write applications that target mobile devices.
(D)	Microsoft .NET provides a platform for developing the next generation of Windows/Web applications.
(E)	C# provides a very simple and yet powerful tool for building interoperable, scalable, and robust applications.

(A)	B, C, D, and E	(C)	C, D
(B)	B	(D)	D, E

2. Match the components of C# .NET Framework against their corresponding descriptions.

Description		Framework Component	
(A)	Performs functions such as memory management, error handling, and garbage collection.	(1)	WPF
(B)	Provides a set of classes to design forms for the Web pages similar to the HTML forms.	(2)	ASP.NET
(C)	Provides a UI framework based on XML and vector graphics to create desktop applications with rich UI on the Windows platform.	(3)	Common Language Runtime (CLR)
(D)	Provides classes to interact with databases.	(4)	Web Forms
(E)	Provides a set of classes to build Web applications.	(5)	ADO.NET

(A)	A-3, B-1, C-4, D-5, E-2	(C)	A-3, B-4, C-1, D-5, E-2
(B)	A-1, B-3, C-4, D-5, E-2	(D)	A-5, B-2, C-4, D-1, E-3

3. Which of these statements about the features of the C# platform and the .NET framework are true?

(A)	The .NET Framework minimizes software deployment and versioning conflicts by providing a code-execution environment.
(B)	The .NET Framework supports only VB and C# languages.
(C)	The Just-In-Time (JIT) compiler converts the operating system code to the MSIL code when the code is executed for the first time.
(D)	The .NET Framework provides consistent object-oriented programming environment.
(E)	The .NET Framework builds, deploys, and runs applications.

(A)	A	(C)	A, C
(B)	A, D, and E	(D)	D

4. Which of these statements about the language features of C# are true?

(A)	The garbage collector allocates and de-allocates memory using automatic memory management.
(B)	C# applications integrate and use the code written in any other .NET language.
(C)	Automatic memory management decreases the quality of the code and reduces the performance and the productivity.
(D)	Developers create mobile applications for pocket PCs, PDAs, and cell phones using the C# language.
(E)	C# application programming uses objects so that code written once can be reused.

(A)	A	(C)	C, E
(B)	B, C, D	(D)	A, B, D, E

5. Match the features of Visual Studio 2012 against their corresponding descriptions.

Description		Feature	
		(1)	Code Snippets
(A)	Saves the work on a regular basis and thus, minimizes accidental loss of information.	(2)	Edit Marks
(C)	Changes the structure and content of the code in many ways.	(3)	AutoRecover
(D)	Offers productive developer tools for novice as well as experienced programmers.	(4)	Refactoring
(E)	Provides a visual indication of what has changed during the editing session.	(5)	Comprehensive Tools Platform

(A)	A-3, B-1, C-4, D-5, E-2	(C)	A-5, B-1, C-4, D-3, E-2
(B)	A-1, B-3, C-4, D-5, E-2	(D)	A-5, B-2, C-4, D-1, E-3

1.5.1 Answers

1.	A
2.	C
3.	B
4.	C
5.	A



Summary

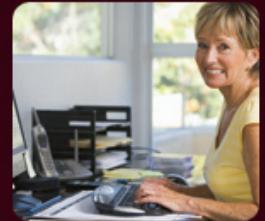
- The .NET Framework is an infrastructure that enables building, deploying, and running different types of applications and services using .NET technologies.
- The two core components of the .NET Framework which are integral to any application or service development are the CLR and the .NET Framework class library.
- The CLR is a virtual machine component of .NET that is used to convert the CIL code to the machine language code.
- C# is an object-oriented language derived from C and C++.
- The C# language provides the feature of allocating and releasing memory using automatic memory management.
- Visual Studio 2012 provides the environment to create, deploy, and run applications developed using the .NET framework.
- Some of the languages supported by Visual Studio 2012 include Visual Basic.NET, Visual C++, Visual C#, Visual J#, and Visual F#.



Need
HELP
on a topic? = **FAQs**



www.onlinevarsity.com



Session -2

Variables and Data Types in C#

Welcome to the Session, **Variables and Data Types in C#**.

This session describes variables, data types, XML commenting, and accepting and displaying data. Variables allow you to store values and reuse them later in the program. C# provides a number of data types that can be used to declare variables and store data. Visual Studio 2012 produces XML comments by taking specially marked and structured comments from within the code and building them into an XML file. C# provides the `ReadLine()` and `WriteLine()` methods to accept and display data.

In this session, you will learn to:

- Define and describe variables and data types in C#
- Explain comments and XML documentation
- Define and describe constants and literals
- List the keywords and escape sequences
- Explain input and output

2.1 Variables and Data Types in C#

A variable is used to store data in a program and is declared with an associated data type. A variable has a name and may contain a value. A data type defines the type of data that can be stored in a variable.

2.1.1 Definition

A variable is an entity whose value can keep changing during the course of a program. For example, the age of a student, the address of a faculty member, and the salary of an employee are all examples of entities that can be represented by variables.

In C#, similar to other programming languages, a variable is a location in the computer's memory that is identified by a unique name and is used to store a value. The name of the variable is used to access and read the value stored in it. For example, you can create a variable called `empName` to store the name of an employee.

Different types of data such as a character, an integer, or a string can be stored in variables. Based on the type of data that needs to be stored in a variable, variables can be assigned different data types.

2.1.2 Using Variables

In C#, memory is allocated to a variable at the time of its creation. During creation, a variable is given a name that uniquely identifies the variable within its scope. For example, you can create a variable called `empName` to store the name of an employee.

You can initialize a variable at the time of creating the variable or at a later time. Once initialized, the value of a variable can be changed as required.

In C#, variables enable you to keep track of data being used in a program. When referring to a variable, you are actually referring to the value stored in that variable.

Figure 2.1 illustrates the concept of a variable.

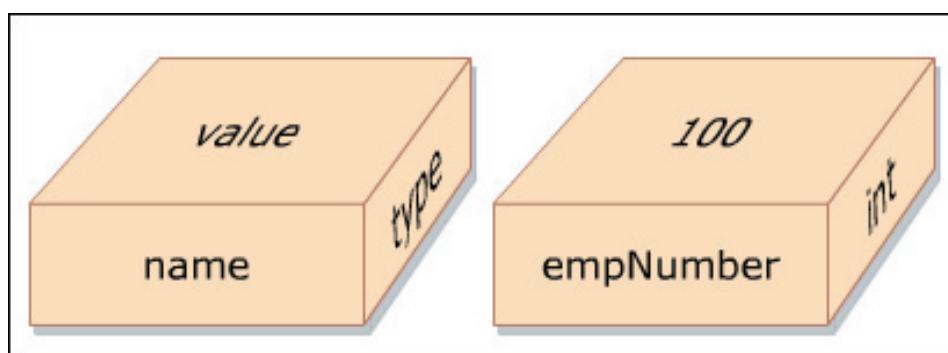


Figure 2.1: Variable

The following syntax is used to declare variables in C#.

Syntax:

```
<datatype><variableName>;
```

where,

datatype: Is a valid data type in C#.

variableName: Is a valid variable name.

The following syntax is used to initialize variables in C#.

Syntax:

```
<variableName> = <value>;
```

where,

=: Is the assignment operator used to assign values.

value: Is the data that is stored in the variable.

The following code snippet declares two variables, namely `empNumber` and `empName`.

Code Snippet 1:

```
int empNumber;
string empName;
```

Code Snippet 1 declares an integer variable, `empNumber`, and a string variable, `empName`. Memory is allocated to hold data in each variable.

Values can be assigned to variables by using the assignment operator (=), as follows:

```
empNumber = 100;
empName = "David Blake";
```

You can also assign a value to a variable upon creation, as follows:

```
int empNumber = 100;
```

2.1.3 Data Types

You can store different types of values such as numbers, characters, or strings in different variables. The compiler must know what kind of data a particular variable is expected to store. To identify the type of data that can be stored in a variable, C# provides different data types.

When a variable is declared, a data type is assigned to the variable. This allows the variable to store values of the assigned data type.

In C# programming language, data types are divided into two categories:

→ Value Types

Variables of value types store actual values. These values are stored in a stack. Stack storage results in faster memory allocation to variables of value types. The values can be either of a built-in data type or a user-defined data type. Most of the built-in data types are value types. The value type built-in data types are `int`, `float`, `double`, `char`, and `bool`. User-defined value types are created using the `struct` and `enum` keywords. Programmers use the `struct` keyword to create custom value types that holds a small set of related variables. The `enum` keyword is used to create custom value types that define a set of named integral constants.

→ Reference Types

Variables of reference type store the memory address of other variables in a heap. These values can either belong to a built-in data type or a user-defined data type. For example, `string` is a built-in data type which is a reference type. Most of the user-defined data types such as `class` are reference types.

Figure 2.2 displays the data types.

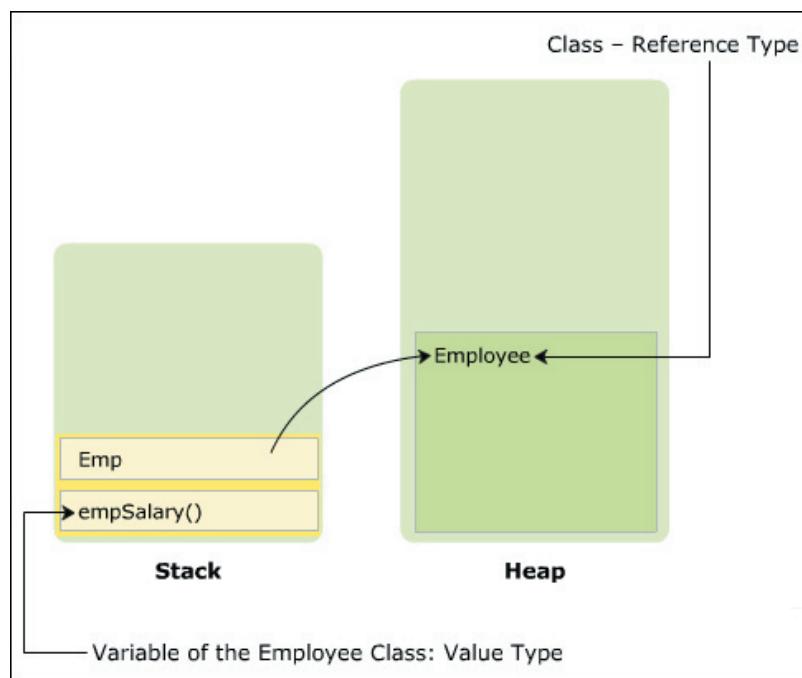


Figure 2.2: Data Types

2.1.4 Pre-defined Data Types

The pre-defined data types are referred to as basic data types in C#. These data types have a pre-defined range and size. The size of the data type helps the compiler to allocate memory space and the range helps the compiler to ensure that the value assigned is within the range of the variable's data type.

Table 2.1 summarizes the pre-defined data types in C#.

Data Type	Size	Range
byte	Unsigned 8-bit integer	0 to 255
sbyte	Signed 8-bit integer	-128 to 127
short	Signed 16-bit integer	-32,768 to 32,767
ushort	Unsigned 16-bit integer	0 to 65,535
int	Signed 32-bit integer	-2,147,483,648 to 2,147,483,647
uint	Unsigned 32-bit integer	0 to 4,294,967,295
long	Signed 64-bit integer	-9,223,372,036,854,775, 808 to 9,223,372,036,854,775,807
ulong	Unsigned 64-bit integer	0 to 18,446,744,073,709,551,615
float	32-bit floating point with 7 digits precision	$\pm 1.5e-45$ to $\pm 3.4e38$
double	64-bit floating point with 15-16 digits precision	$\pm 5.0e-324$ to $\pm 1.7e308$
decimal	128-bit floating point with 28-29 digits precision	$\pm 1.0 \times 10e-28$ to $\pm 7.9 \times 10e28$
char	Unicode 16-bit character	U+0000 to U+ffff
bool	Stores either true or false	true or false

Table 2.1: Pre-defined Data Types

Note -

→ **Unicode Characters**

Unicode is a 16-bit character set that contains all of the characters commonly used in information processing. It is an attempt to consolidate the alphabets and ideographs of the world's languages into a single, international character set.

Unicode characters are represented as 16-bit characters and are used to denote multiple languages spoken around the world. The `char` data type uses Unicode characters and these are prefixed by the letter 'U'.

→ **Signed Integers**

Signed integers can represent both positive and negative numbers.

→ **Float and Char Data Type Representation**

A value of `float` type variable must always end with the letter F or f. A value of `char` type must always be enclosed in single quotes.

2.1.5 Classification

Reference data types store the memory reference of other variables. These other variables hold the actual values. Reference types can be classified into the following types:

→ Object

`Object` is a built-in reference data type. It is a base class for all pre-defined and user-defined data types. A class is a logical structure that represents a real world entity. This means that the pre-defined and user-defined data types are created based on the `Object` class.

→ String

`String` is a built-in reference type. `String` type signifies Unicode character string values. It allows you to assign and manipulate string values. Once strings are created, they cannot be modified.

→ Class

A class is a user-defined structure that contains variables and methods. For example, the `Employee` class can be a user-defined structure that can contain variables such as `empSalary`, `empName`, and `empAddress`. In addition, it can contain methods such as `CalculateSalary()`, which returns the net salary of an employee.

→ Delegate

A delegate is a user-defined reference type that stores the reference of one or more methods.

→ Interface

An interface is a user-defined structure that groups related functionalities which may belong to any class or struct.

→ Array

An array is a user-defined data structure that contains values of the same data type, such as marks of students.

2.1.6 Rules

A variable needs to be declared before it can be referenced. You need to follow certain rules while declaring a variable:

- A variable name can begin with an uppercase or a lowercase letter. The name can contain letters, digits, and the underscore character (_).
- The first character of the variable name must be a letter and not a digit.

The underscore is also a legal first character, but it is not recommended at the beginning of a name.

- C# is a case-sensitive language; hence, variable names count and Count refer to two different variables.
- C# keywords cannot be used as variable names. If you still need to use a C# keyword, prefix it with the '@' symbol.

It is always advisable to give meaningful names to variables such that the name gives an idea about the content that is stored in the variable.

Note - Microsoft recommends camel case notation for C# variable names. You should not use underscores and must ensure that the first letter of the identifier is in lowercase. In addition, you must capitalize the first letter of each subsequent word of the identifier. For example, consider the following variable declarations:

```
int totMonths = 12;
string empName = "John Fernandes";
bool statusInfo = true;
```

2.1.7 Validity

A variable's type and identifier (name) need to be mentioned at the time of declaring a variable. This tells the compiler, the name of the variable and the type of data the variable will store. If you attempt to use an undeclared variable, the compiler will generate an error message.

Table 2.2 displays a list of valid and invalid variable names in C#.

Variable Name	Valid/Invalid
Employee	Valid
student	Valid
_Name	Valid
Emp_Name	Valid
@goto	Valid
static	Invalid as it is a keyword
4myclass	Invalid as a variable cannot start with a digit
Student&Faculty	Invalid as a variable cannot have the special character &

Table 2.2: Valid and Invalid Variable Names

Note - When you declare a variable, the computer allocates memory for it. Hence, to avoid wasting computer memory, it is recommended to declare variables only when required.

2.1.8 Declaration

In C#, you can declare multiple variables at the same time in the same way you declare a single variable. After declaring variables, you need to assign values to them. Assigning a value to a variable is called initialization. You can assign a value to a variable while declaring it or at a later time.

The following is the syntax to declare and initialize a single variable.

Syntax:

```
<data type><variable name> = <value>;
```

where,

data type: Is a valid variable type.

variable name: Is a valid variable name or identifier.

value: Is the value assigned to the variable.

The following is the syntax to declare multiple variables.

Syntax:

```
<data type><variable name1>, <variable name2>, ..., <variable nameN>;
```

where,

data type: Is a valid variable type.

variable name1, variable name2, variable nameN: Are valid variable names or identifiers.

The following is the syntax to declare and initialize multiple variables.

Syntax:

```
<data type><variable name1> = <value1>, <variable name2> = <value2>;
```

Code Snippet 2 demonstrates how to declare and initialize variables in C#.

Code Snippet 2:

```
bool boolTest = true;
short byteTest = 19;
int intTest;
string stringTest = "David";
float floatTest;
int Test = 140000;
```

```

floatTest = 14.5f;

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

Console.WriteLine("byteTest = " + byteTest);

Console.WriteLine("intTest = " + intTest);

Console.WriteLine("stringTest = " + stringTest);

Console.WriteLine("floatTest = " + floatTest);

```

In the code snippet, variables of type `bool`, `byte`, `int`, `string`, and `float` are declared. Values are assigned to each of these variables and are displayed using the `WriteLine()` method of the `Console` class.

Figure 2.3 displays the output.

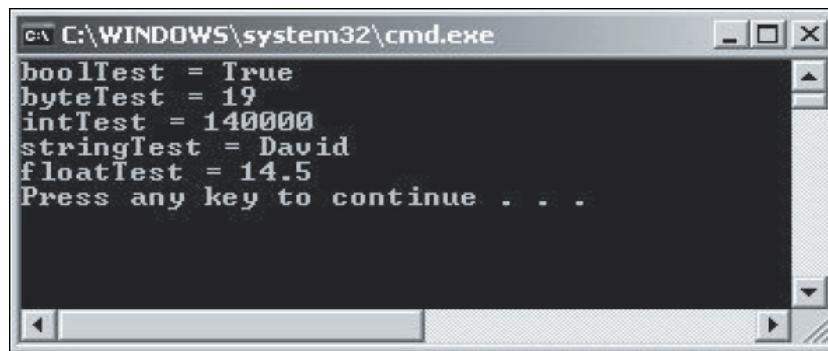


Figure 2.3: Output of Code Snippet 2

2.1.9 Implicitly Typed Variables

When you declare and initialize a variable in a single step, you can use the `var` keyword in place of the type declaration. Variables declared using the `var` keyword are called implicitly typed variables. For implicitly typed variables, the compiler infers the type of the variable from the initialization expression.

Code Snippet 3 demonstrates how to declare and initialize implicitly typed variables in C#.

Code Snippet 3:

```

var boolTest = true;
var byteTest = 19;
var intTest = 140000;
var stringTest = "David";
var floatTest = 14.5f;

Console.WriteLine("boolTest = {0}", boolTest);
Console.WriteLine("byteTest = " + byteTest);
Console.WriteLine("intTest = " + intTest);

```

```
Console.WriteLine("stringTest = " + stringTest);
Console.WriteLine("floatTest = " + floatTest);
```

In Code Snippet 3, four implicitly typed variables are declared and initialized with values. The values of each variable are displayed using the `WriteLine()` method of the `Console` class.

Figure 2.4 displays the output.

```
boolTest = True
byteTest = 19
intTest = 140000
stringTest = David
floatTest = 14.5
Press any key to continue . . .
```

Figure 2.4: Output of Code Snippet 3

Note - You must declare and initialize implicitly typed variables at the same time. Not doing so will result in the compiler reporting an error.

2.2 Comments and XML Documentation

Comments help in reading the code of a program to understand the functionality of the program. C# supports three types of comments: single-line comments, multi-line comments, and XML comments.

2.2.1 Definition

In C#, comments are given by the programmer to provide information about a piece of code. Comments make the program more readable. They help the programmer to explain the purpose of using a particular variable or method. While the program is executed, the compiler can identify comments as they are marked with special characters. Comments are ignored by the compiler during the execution of the program.

C# supports three types of comments. These are as follows:

→ Single-line Comments

Single-line comments begin with two forward slashes (//). You can insert the single-line comment as shown in Code Snippet 4.

Code Snippet 4:

```
// This block of code will add two numbers
int doSum = 4 + 3;
```

To write more than one line as comment, begin each line with the double slashes // characters as shown in the following code.

```
// This block of code will add two numbers and then put
// the result in the variable, doSum
int doSum = 4 + 3;
```

You can also write the single-line comment in the same line as shown in the following code.

```
int doSum = 4 + 3; // Adding two numbers
```

→ Multi-line Comments

Multi-line comments begin with a forward slash followed by an asterisk /*) and end with an asterisk followed by a forward slash (*). Between these starting and ending characters, you can insert multiple lines of comments.

You can insert multi-line comments as shown in Code Snippet 5.

Code Snippet 5:

```
/* This is a block of code that will multiply two numbers,
divide the resultant value by 2 and display the quotient */
int doMult = 5 * 20;
int doDiv = doMult / 2;
Console.WriteLine("Quotient is :" + doDiv)
```

→ XML Comments

XML comments begin with three forward slashes (///). Unlike single-line and multi-line comments, the XML comment must be enclosed in an XML tag. You need to create XML tags to insert XML comments. Both the XML tags and XML comments must be prefixed with three forward slashes.

You can insert an XML comment as shown in Code Snippet 6.

Code Snippet 6:

```
///<summary>
/// You are in the XML tag called summary.
///</summary>
```

Figure 2.5 displays a complete example of using XML comments.

```
class Comments
{
    //The following is an XML Comment
    /// <summary>
    /// This is the XML tag
    /// and can be extracted to an XML file
    /// </summary>

    //The following is a single line comment
    //Main method begins here

    static void Main(string[] args)
    {
        //This is a multiline comment
        /* The WriteLine method is used to
           print the specified value.
           The following statement
           uses the WriteLine method.*/
        Console.WriteLine("C#");
    }
}// End of class
```

Figure 2.5: Example of Using XML Comments

Note - When you press the <Enter> key after the opening characters ‘/*’ in a multi-line comment, an ‘*’ character appears at the beginning of the new line. This character indicates that a new line is inserted in the comment. When the multi-line comment is closed, Visual Studio 2012 will change all text contained in the comment as BOLD text. This will highlight the text that is written as a comment.

In C#, comments are also known as remarks.

2.2.2 XML Documentation

In C#, you can create an XML document that will contain all the XML comments. This document is useful when multiple programmers want to view information of the program. For example, consider a scenario where one of the programmers wants to understand the technical details of the code and another programmer wants see the total variables used in the code. In this case, you can create an XML document that will contain all the required information. To create an XML document, you must use the Visual Studio 2012 Command Prompt window.

Figure 2.6 displays the XML comments that can be extracted to an XML file.

```
class XMLComments
{
    /// <summary>
    /// This is the XML comment and can be extracted to a XML file.
    /// </summary>
    static void Main(string[] args)
    {
        Console.WriteLine("This program illustrates XML Comments");
    }
}
```

Figure 2.6: XML Comments

The following syntax is used to create an XML document from the C# source file.

Syntax:

```
csc /doc: <XMLfilename.xml><CSharpfilename.cs>
```

where,

XMLfilename.xml: Is the name of the XML file that is to be created.

CSharpfilename.cs: Is the name of the C# file from where the XML comments will be extracted.

2.2.3 Pre-defined XML Tags

XML comments are inserted in XML tags. These tags can either be pre-defined or user-defined. Table 2.3 lists the widely used pre-defined XML tags and states their conventional use.

Pre-defined Tags	Descriptions
<c>	Sets text in a code-like font.
<code>	Sets one or more lines of source code or program output.
<example>	Indicates an example.
<param>	Describes a parameter for a method or a constructor.
<returns>	Specifies the return value of a method.
<summary>	Summarizes the general information of the code.
<exception>	Documents an exception class.
<include>	Refers to comments in another file using the XPath syntax, which describes the types and members in the source code.
<list>	Inserts a list into the documentation file.
<para>	Inserts a paragraph into the documentation file.
<paramref>	Indicates that a word is a parameter.
<permission>	Documents access permissions.
<remarks>	Specifies overview information about the type.
<see>	Specifies a link.
<seealso>	Specifies the text that might be required to appear in a See Also section.
<value>	Describes a property.

Table 2.3: Pre-defined XML Tags

Figure 2.7 displays an example of pre-defined XML tags.

```
<?xml version="1.0" ?>
- <doc>
- <assembly>
  <name>XMLComments</name>
  </assembly>
- <members>
  - <member name="T:Project.XMLComments">
    <summary>This program demonstrates the use of XML comments</summary>
  </member>
  - <member name="M:Project.XMLComments.Main(System.String[])>
    <summary>
      The execution of your program begins with the Main method.
      <param name="args">Command Line Arguments</param>
      <returns>The return type of this method is void</returns>
    </summary>
    <remarks>The Main method can be declared with or without parameters.</remarks>
  </member>
</members>
</doc>
```

Figure 2.7: Example of Pre-defined XML Tags

Code Snippet 7 demonstrates the use of XML comments.

Code Snippet 7:

```
using System;

/// <summary>
/// The program demonstrates the use of XML comments .
/// 
/// Employee class uses constructors to initialize the ID and
/// name of the employee and displays them.
/// </summary>
/// <remarks>
/// This program uses both parameterized and
/// non-parameterized constructors .
/// </remarks>

class Employee
{
  /// <summary>
  /// Integer field to store employee ID.
  /// </summary>
```

```
private int _id;  
/// <summary>  
/// String field to store employee name.  
/// </summary>  
  
private string _name;  
/// <summary>  
/// This constructor initializes the id and name to -1 and null.  
/// </summary>  
/// <remarks>  
/// <seealso ref="Employee(int, string)">Employee(int, string)</seealso>  
/// </remarks>  
  
public Employee()  
{  
    _id=-1;  
    _name=null;  
}  
  
/// <summary>  
/// This constructor initializes the id and name.  
/// (<paramref name="id"/>, <paramref name="name"/>).  
/// </summary>  
/// <param name="id">Employee ID</param>  
/// <param name="name">Employee Name</param>  
  
public Employee(int id, string name)  
{  
    this._id=id;  
    this._name=name; }  
  
/// <summary>  
/// The entry point for the application.  
/// <param name="args">A list of command line arguments</param>  
/// </summary>  
  
static void Main(string[] args)
```

```
{
// Creating an object of Employee class and displaying the
// id and name of the employee
Employee objEmp = new Employee(101, "David Smith");
Console.WriteLine("Employee ID : {0} \nEmployee Name : {1}",
objEmp._name);
}
```

Figure 2.8 displays the XML document.

```
<?xml version="1.0" ?>
<doc>
- <assembly>
  - <name>Payroll</name>
  </assembly>
- <members>
  - <member name="T:Payroll.Employee">
    <summary>The program demonstrates the use of XML comments. Employee class uses constructors to initialise the ID and name of the employee and displays them.</summary>
    <remarks>This program uses both parameterised and non-parameterised constructors.</remarks>
  </member>
  - <member name="F:Payroll.Employee._id">
    <summary>Integer field to store employee ID.</summary>
  </member>
  - <member name="F:Payroll.Employee._name">
    <summary>String field to store employee name.</summary>
  </member>
  - <member name="M:Payroll.Employee.#ctor">
    <summary>This constructor initializes the id and name to -1 and null.</summary>
    - <remarks>
      <seealso cref="M:Payroll.Employee.#ctor(System.Int32,System.String)">Employee(int, string)</seealso>
    </remarks>
  </member>
  - <member name="M:Payroll.Employee.#ctor(System.Int32,System.String)">
    - <summary>
      This constructor initializes the id and name to (
      <paramref name="id" />
      .
      <paramref name="name" />
      ).
    </summary>
    <param name="id">Employee ID</param>
    <param name="name">Employee Name</param>
  </member>
  - <member name="M:Payroll.Employee.Main(System.String[])">
    - <summary>
      The entry point for the application.
      <param name="args">A list of command line arguments</param>
    </summary>
  </member>
</members>
</doc>
```

Figure 2.8: XML Document

In this example, the `<remarks>`, `<seealso>`, and `<paramref>` XML documentation tags are used. The `<remarks>` tag is used to provide information about a specific class. The `<seealso>` tag is used to specify the text that should appear in the See Also section. The `<paramref>` tag is used to indicate that the specified word is a parameter.

2.3 Constants and Literals

A constant has a fixed value that remains unchanged throughout the program while a literal provides a mean of expressing specific values in a program.

2.3.1 Need for Constants

Consider a code that calculates the area of the circle. To calculate the area of the circle, the value of pi and radius must be provided in the formula. The value of pi is a constant value. This value will remain unchanged irrespective of the value of the radius provided.

Similarly, constants in C# are fixed values assigned to identifiers that are not modified throughout the execution of the code. They are defined when you want to preserve values to reuse them later or to prevent any modification to the values.

2.3.2 Constants

In C#, you can declare constants for all data types. You have to initialize a constant at the time of its declaration. Constants are declared for value types rather than for reference types. To declare an identifier as a constant, the `const` keyword is used in the identifier declaration. The compiler can identify constants at the time of compilation because of the `const` keyword.

The following syntax is used to initialize a constant.

Syntax:

```
const<data type><identifier name> = <value>;
```

where,

`const`: Keyword denoting that the identifier is declared as constant.

`data type`: Data type of constant.

`identifier name`: Name of the identifier that will hold the constant.

`value`: Fixed value that remains unchanged throughout the execution of the code.

Code Snippet 8 declares a constant, `_pi`, and a variable, `radius`, to calculate the area of the circle.

Code Snippet 8:

```
const float _pi = 3.14F;
float radius = 5;
float area = _pi * radius * radius;
Console.WriteLine("Area of the circle is " + area);
```

In the code, a constant called `_pi` is assigned the value 3.14, which is a fixed value. The variable, `radius`, stores the radius of the circle. The code calculates the area of the circle and displays it as the output.

2.3.3 Using Literals

A literal is a static value assigned to variables and constants. You can define literals for any data type of C#. Numeric literals might suffix with a letter of the alphabet to indicate the data type of the literal. This letter can be either in upper or lowercase. For example, in the following declaration, `string bookName = "Csharp"`, `Csharp` is a literal assigned to the variable `bookName` of type `string`.

In C#, there are six types of literals. These are as follows:

→ **Boolean Literal**

Boolean literals have two values, true or false. For example,

```
boolval = true;
```

where,

`true`: Is a Boolean literal assigned to the variable `val`.

→ **Integer Literal**

An integer literal can be assigned to `int`, `uint`, `long`, or `ulong` data types. Suffixes for integer literals include `U`, `L`, `UL`, or `LU`. `U` denotes `uint` or `ulong`, `L` denotes `long`. `UL` and `LU` denote `ulong`. For example,

```
longval = 53L;
```

where,

`53L`: Is an integer literal assigned to the variable `val`.

→ **Real Literal**

A real literal is assigned to `float`, `double` (default), and `decimal` data types. This is indicated by the suffix letter appearing after the assigned value. A real literal can be suffixed by `F`, `D`, or `M`. `F` denotes `float`, `D` denotes `double`, and `M` denotes `decimal`. For example,

```
floatval = 1.66F;
```

where,

`1.66F`: Is a real literal assigned to the variable `val`.

→ **Character Literal**

A character literal is assigned to a `char` data type. A character literal is always enclosed in single quotes.

For example,

```
charval = 'A';
```

where,

A: Is a character literal assigned to the variable val.

→ String Literal

There are two types of string literals in C#, regular and verbatim. A regular string literal is a standard string. A verbatim string literal is similar to a regular string literal but is prefixed by the '@' character. A string literal is always enclosed in double quotes.

For example,

```
stringmailDomain = "@gmail.com";
```

where,

@gmail.com: Is a verbatim string literal.

→ Null Literal

The null literal has only one value, null. For example,

```
string email = null;
```

where,

null: Specifies that e-mail does not refer to any objects (reference).

Note - If you assign numeric literals with suffixes to indicate their type, these suffixes do not form part of the literal value and are not displayed in code output.

2.4 Keywords and Escape Sequences

A keyword is one of the reserved words that has a pre-defined meaning in the language. Escape sequence characters in C# are characters preceded by a back slash (\) and denote a special meaning to the compiler.

2.4.1 Keywords

Keywords are reserved words and are separately compiled by the compiler. They convey a pre-defined meaning to the compiler and hence, cannot be created or modified. For example, int is a keyword that specifies that the variable is of data type integer. You cannot use keywords as variable names, method names, or class names, unless you prefix the keywords with the '@' character.

Table 2.4 lists the keywords used in C#.

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

Table 2.4: Keywords

C# provides contextual keywords that have special meaning in the context of the code where they are used. The contextual keywords are not reserved and can be used as identifiers outside the context of the code. When new keywords are added to C#, they are added as contextual keywords.

Table 2.5 lists the contextual keywords used in C#.

add	alias	ascending	async	await	descending	dynamic
					yield	from
from	get	global	group	into	join	let
orderby	partial	remove	select	set	value	var
where						

Table 2.5: Contextual Keywords

2.4.2 Need of Escape Sequence Characters

Consider a payroll system of an organization. One of its functions is to display the monthly salary as output with the salary displayed on the next line. The programmer wants to write the code in such a way that the salary is always printed on the next line irrespective of the length of string to be displayed with the salary amount. This is done using escape sequences.

2.4.3 Definition

An escape sequence character is a special character that is prefixed by a backslash (\). Escape sequence characters are used to implement special non-printing characters such as a new line, a single space, or a backspace. These non-printing characters are used while displaying formatted output to the user to maximize readability.

The backslash character tells the compiler that the following character denotes a non-printing character. For example, \n is used to insert a new line similar to the Enter key of the keyboard.

In C#, the escape sequence characters must always be enclosed in double quotes.

2.4.4 Escape Sequence Characters in C#

There are multiple escape sequence characters in C# that are used for various kinds of formatting.

Table 2.6 displays the escape sequence characters and their corresponding non-printing characters in C#.

Escape Sequence Characters	Non-Printing Characters
\'	Single quote, needed for character literals.
\"	Double quote, needed for string literals.
\\\	Backslash, needed for string literals.
\0	Unicode character 0.
\a	Alert.
\b	Backspace.
\f	Form feed.
\n	New line.
\r	Carriage return.
\t	Horizontal tab.
\v	Vertical tab.
\?	Literal question mark.
\ooo	Matches an ASCII character using a three-digit octal character code.
\xhh	Matches an ASCII character using hexadecimal representation (exactly two digits). For example, \x61 represents the character 'a'.
\uhhhh	Matches a Unicode character using hexadecimal representation (exactly four digits). For example, the character \u0020 represents a space.

Table 2.6: Escape Sequence Characters

Code Snippet 9 demonstrates the use of Unicode characters.

Code Snippet 9:

```
string str = "\u0048\u0065\u006C\u006C\u006F";
Console.WriteLine("\t" + str + "!\\n");
Console.WriteLine("David\u0020\"2007\" ");
```

In the code, the variable **str** is declared as type **string** and stores Unicode characters for the letters H, e, l, l, and o. The method uses the horizontal tab escape sequence character to display the output leaving one tab space. The new line escape sequence character used in the string of the method displays the output of the next statement in the next line. The next statement uses the **WriteLine()** method to display David "2007". The string in the method specifies the Unicode character to display a space between David and 2007.

Output:

Hello!

David "2007"

Code Snippet 10 demonstrates the use of some of the commonly used escape sequences.

Code Snippet 10:

```
using System;
class FileDemo
{
    static void Main(string[] args)
    {
        string path = "C:\\Windows\\MyFile.txt";
        bool found = true;
        if (found)
        {
            Console.WriteLine("Filepath : \' " + path + " \' ");
        }
        else
        {
            Console.WriteLine("File Not Found! \\a");
        }
    }
}
```

In this code, the \\, \', and \a escape sequences are used. The \\ escape sequence is used for printing a backslash. The \' escape sequence is used for printing a single quote. The \a escape sequence is used for producing a beep.

Figure 2.9 displays the output of Code Snippet 10.



Figure 2.9: Output of Code Snippet 10

2.5 Input and Output

Programmers often need to display the output of a C# program to users. The programmer can use the command line interface to display the output. The programmer can similarly accept inputs from a user through the command line interface. Such input and output operations are also known as console operations.

2.5.1 Console Operations

Console operations are tasks performed on the command line interface using executable commands. The console operations are used in software applications because these operations are easily controlled by the operating system. This is because console operations are dependent on the input and output devices of the computer system.

A console application is one that performs operations at the command prompt. All console applications consist of three streams, which are a series of bytes. These streams are attached to the input and output devices of the computer system and they handle the input and output operations. The three streams are as follows:

→ Standard in

The standard `in` stream takes the input and passes it to the console application for processing.

→ Standard out

The standard `out` stream displays the output on the monitor.

→ Standard err

The standard `err` stream displays error messages on the monitor.

2.5.2 Output Methods

In C#, all console operations are handled by the `Console` class of the `System` namespace. A namespace is a collection of classes having similar functionalities.

To write data on the console, you need the standard output stream. This stream is provided by the output methods of `Console` class. There are two output methods that write to the standard output stream. They are as follows:

→ **`Console.WriteLine()`**

Writes any type of data.

→ **`Console.WriteLine()`**

Writes any type of data and this data ends with a new line character in the standard output stream. This means any data after this line will appear on the new line.

The following syntax is used for the `Console.WriteLine()` method, which allows you to display the information on the console window.

Syntax:

```
Console.WriteLine("<data>" + variables);
```

where,

data: Specifies strings or escape sequence characters enclosed in double quotes.

variables: Specify variable names whose value should be displayed on the console.

The following syntax is used for the `Console.WriteLine()` method, which allows you to display the information on a new line in the console window.

Syntax:

```
Console.WriteLine("<data>" + variables);
```

Code Snippet 11 shows the difference between the `Console.WriteLine()` method and `Console.Write()` method.

Code Snippet 11:

```
Console.WriteLine("C# is a powerful programming language");
Console.WriteLine("C# is a powerful");
Console.WriteLine("programming language");
Console.Write("C# is a powerful");
Console.WriteLine(" programming language");
```

Output:

C# is a powerful programming language

C# is a powerful

programming language

C# is a powerful programming language

2.5.3 Placeholders

The `WriteLine()` and `Write()` methods accept a list of parameters to format text before displaying the output. The first parameter is a string containing markers in braces to indicate the position, where the values of the variables will be substituted.

Each marker indicates a zero-based index based on the number of variables in the list. For example, to indicate the first parameter position, you write `{0}`, second you write `{1}`, and so on. The numbers in the curly brackets are called placeholders.

Code Snippet 12 uses placeholders in the `Console.WriteLine()` method to display the result of the multiplication operation.

Code Snippet 12:

```
int number, result;
number = 5;
result = 100 * number;
Console.WriteLine("Result is {0} when 100 is multiplied by {1}", result, number);
result = 150 / number;
Console.WriteLine("Result is {0} when 150 is divided by {1}", +result, number);
```

Output:

Result is 500 when 100 is multiplied by 5

Result is 30 when 150 is divided by 5

Here, `{0}` is replaced with the value in `result` and `{1}` is replaced with the value in `number`.

2.5.4 Input Methods

In C#, to read data, you need the standard input stream. This stream is provided by the input methods of the `Console` class. There are two input methods that enable the software to take in the input from the standard input stream.

These methods are as follows:

→ **Console.Read()**

Reads a single character.

→ **Console.ReadLine()**

Reads a line of strings.

Code Snippet 13 reads the name using the `ReadLine()` method and displays the name on the console window.

Code Snippet 13:

```
string name;
Console.Write("Enter your name: ");
name = Console.ReadLine();
Console.WriteLine("You are {0}", name);
```

In Code Snippet 13, the `ReadLine()` method reads the name as a string. The string that is given is displayed as output using placeholders.

Output:

Enter your name: David Blake

You are David Blake

Code Snippet 14 demonstrates the use of placeholders in the `Console.WriteLine()` method.

Code Snippet 14:

```
using System;
class Loan
{
    static void Main(string[] args)
    {
        string custName;
        double loanAmount;
        float interest = 0.09F;
        double interestAmount = 0;
        double totalAmount = 0;
```

```

double totalAmount = 0;

Console.Write("Enter the name of the customer : ");
custName = Console.ReadLine();

Console.Write("Enter loan amount : ");
loanAmount = Convert.ToDouble(Console.ReadLine());

interestAmount = loanAmount * interest;

totalAmount = loanAmount + interestAmount;

Console.WriteLine("\nCustomer Name : {0}", custName);

Console.WriteLine("Loan amount : ${0:#,###.#0} \nInterest rate
: {1:0%} \nInterest Amount : ${2:#,###.#0}",
loanAmount, interest, interestAmount);

Console.WriteLine("Total amount to be paid : ${0:#,###.#0} ",
totalAmount);

}
}

```

In Code Snippet 14, the name and loan amount are accepted from the user using the `Console.ReadLine()` method. The details are displayed on the console using the `Console.WriteLine()` method. The placeholders `{0}`, `{1}`, and `{2}` indicate the position of the first, second, and third parameters respectively. The `0` specified before `#` pads the single digit value with a `0`. The `#` option specifies the digit position. The `%` option multiplies the value by `100` and displays the value along with the percentage sign.

Figure 2.10 displays the output of Code Snippet 14.

The screenshot shows a Windows Command Prompt window titled "C:\WINDOWS\system32\cmd.exe". The window contains the following text:

```

Enter the name of the customer : David George
Enter loan amount : 15430

Customer Name : David George
Loan amount : $15,430.00
Interest rate : 09%
Interest Amount : $1,388.70
Total amount to be paid : $16,818.70
Press any key to continue . . .

```

Figure 2.10: Output of Code Snippet 14

2.5.5 Convert Methods

The `ReadLine()` method can also be used to accept integer values from the user. The data is accepted as a string and then converted into the `int` data type. C# provides a `Convert` class in the `System` namespace to convert one base data type to another base data type.

Note - The `Convert.ToInt32()` method converts a specified value to an equivalent 32-bit signed integer. `Convert.ToDecimal()` method converts a specified value to an equivalent decimal number.

Code Snippet 15 reads the name, age, and salary using the `Console.ReadLine()` method and converts the age and salary into `int` and `double` using the appropriate conversion methods of the `Convert` class.

Code Snippet 15:

```
string userName;
int age;
double salary;
Console.Write("Enter your name: ");
userName = Console.ReadLine();
Console.Write("Enter your age: ");
age = Convert.ToInt32(Console.ReadLine());
Console.Write("Enter the salary: ");
salary = Convert.ToDouble(Console.ReadLine());
Console.WriteLine("Name: {0}, Age: {1}, Salary: {2}", userName, age, salary);
```

Output:

```
Enter your name: David Blake
Enter your age: 34
Enter the salary: 3450.50
Name: David Blake, Age: 34, Salary: 3450.50
```

2.5.6 Numeric Format Specifiers

Format specifiers are special characters that are used to display values of variables in a particular format. For example, you can display an octal value as decimal using format specifiers.

In C#, you can convert numeric values in different formats. For example, you can display a big number in an exponential form. To convert numeric values using numeric format specifiers, you should enclose the specifier in curly braces. These curly braces must be enclosed in double quotes. This is done in the output methods of the `Console` class.

The following is the syntax for the numeric format specifier.

Syntax:

```
Console.WriteLine("{format specifier}", <variable name>);
```

where,

format specifier: Is the numeric format specifier.

variable name: Is the name of the integer variable.

2.5.7 Using Numeric Format Specifiers

Numeric format specifiers work only with numeric data. A numeric format specifier can be suffixed with digits. The digits specify the number of zeros to be inserted after the decimal location. For example, if you use a specifier such as C3, three zeros will be suffixed after the decimal location of the given number. Table 2.7 lists some of the numeric format specifiers in C#.

Format Specifier	Name	Description
C or c	Currency	The number is converted to a string that represents a currency amount.
D or d	Decimal	The number is converted to a string of decimal digits (0-9), prefixed by a minus sign in case the number is negative. The precision specifier indicates the minimum number of digits desired in the resulting string. This format is supported for fundamental types only.
E or e	Scientific (Exponential)	The number is converted to a string of the form '-d.ddd...E+ddd' or '-d.ddd...e+ddd', where each 'd' indicates a digit (0-9).

Table 2.7: Numeric Format Specifiers

2.5.8 Custom Numeric Format Strings

Custom numeric format strings contain more than one custom numeric format specifiers and define how data is formatted. A custom numeric format string is defined as any string that is not a standard numeric format string.

Table 2.8 lists the custom numeric format specifiers and their description.

Format Specifier	Description
0	If the value being formatted contains a digit where '0' appears, then it is copied to the result string
#	If the value being formatted contains a digit where '#' appears, then it is copied to the result string
.	The first '.' character verifies the location of the decimal separator
,	The ',' character serves as a thousand separator specifier and a number scaling specifier
%	The '%' character in a format string multiplies a number with 100 before it is formatted
E0, E+0, E-0, e0, e+0, e-0	If any of the given strings are present in the format string and they are followed by at least one '0' character, then the number is formatted using scientific notation
\	The backslash character causes the next character in the format string to be interpreted as an escape sequence
'ABC'	The characters that are enclosed within single or double quotes are copied to the result string
"ABC"	
;	The ';' character separates a section into positive, negative and zero numbers
Other	Any of the other characters are copied to the result string

Table 2.8: Custom Numeric Format Specifiers

Code Snippet 16 demonstrates the conversion of a numeric value using C, D, and E format specifiers.

Code Snippet 16:

```
int num = 456;

Console.WriteLine("{0:C}", num);
Console.WriteLine("{0:D}", num);
Console.WriteLine("{0:E}", num);
```

Output:

\$456.00
456
4.560000E+002

Code Snippet 17 demonstrates the use of custom numeric format specifiers.

Code Snippet 17:

```
using System;
class Banking
{
    static void Main(string[] args)
    {
        double loanAmount = 15590;
        float interest = 0.09F;
        double interestAmount = 0;
        double totalAmount = 0;
        interestAmount = loanAmount * interest;
        totalAmount = loanAmount + interestAmount;
        Console.WriteLine("Loan amount : ${0:#,###.#0}", loanAmount);
        Console.WriteLine("Interest rate : {0:0##%}", interest);
        Console.WriteLine("Total amount to be paid : ${0:#,###.#0}", totalAmount);
    }
}
```

In this code, the #, %, ., and 0 custom numeric format specifiers are used to display the loan details of the customer in the desired format.

Figure 2.11 displays the output of Code Snippet 17.

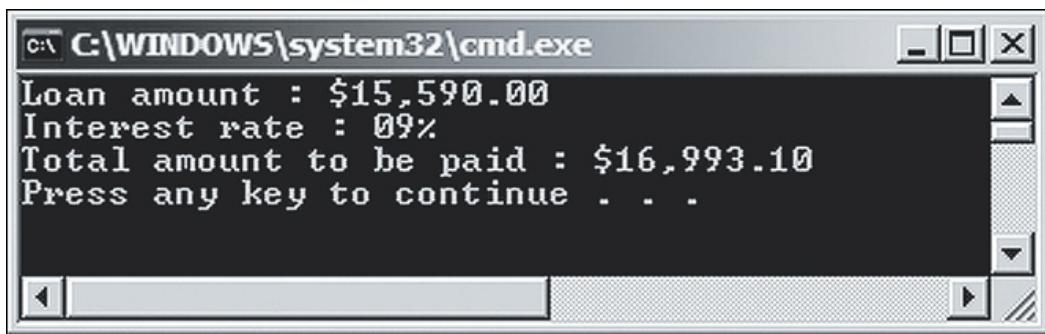


Figure 2.11: Output of Code Snippet 17

2.5.9 More Number Format Specifiers

There are some additional number format specifiers that are described in table 2.9.

Format Specifier	Name	Description
F or f	Fixed-point	The number is converted to a string of the form ‘-ddd.ddd...’ where each ‘d’ indicates a digit (0-9). If the number is negative, the string starts with a minus sign.
N or n	Number	The number is converted to a string of the form ‘-d,ddd,ddd.ddd...’, where each ‘d’ indicates a digit (0-9). If the number is negative, the string starts with a minus sign.
X or x	Hexadecimal	The number is converted to a string of hexadecimal digits. Uses “X” to produce “ABCDEF”, and “x” to produce “abcdef”.

Table 2.9: Additional Numeric Format Specifiers

Code Snippet 18 demonstrates the conversion of a numeric value using F, N, and X format specifiers.

Code Snippet 18:

```
int num = 456;
Console.WriteLine("{0:F}", num);
Console.WriteLine("{0:N}", num);
Console.WriteLine("{0:X}", num);
```

Output:

456.00

456.00

1C8

2.5.10 Standard Date and Time Format Specifiers

A date and time format specifier is a special character that enables you to display the date and time values in different formats. For example, you can display a date in mm-dd-yyyy format and time in hh:mm format. If you are displaying GMT time as the output, you can display the GMT time along with the abbreviation GMT using date and time format specifiers.

The date and time format specifiers allow you to display the date and time in 12-hour and 24-hour formats.

Note - GMT stands for Greenwich Mean Time, which is the standard accepted worldwide to display date and time.

The following is the syntax for date and time format specifiers.

Syntax:

```
Console.WriteLine("{format specifier}", <datetime object>);
```

where,

format specifier: Is the date and time format specifier.

datetime object: Is the object of the `DateTime` class.

2.5.11 Using Standard Date and Time Format Specifiers

Standard date and time format specifiers are used in the `Console.WriteLine()` method with the `datetime` object. To create the `datetime` object, you must create an object of the `DateTime` class and initialize it. The formatted date and time are always displayed as strings in the console window.

Table 2.10 displays some of the standard date and time format specifiers in C#.

Format Specifier	Name	Description
d	Short date	Displays date in short date pattern. The default format is 'mm/dd/yyyy'.
D	Long date	Displays date in long date pattern. The default format is 'dddd*, MMMM*, dd, yyyy'.
f	Full date/time (short time)	Displays date in long date and short time patterns, separated by a space. The default format is 'ddddd*, MMMMM* dd, yyyy HH*:mm*'.
F	Full date/time (long time)	Displays date in long date and long time patterns, separated by a space. The default format is 'ddddd*, MMMMM* dd, yyyy HH*: mm*: ss*'.
g	General date/time (short time)	Displays date in short date and short time patterns, separated by a space. The default format is 'MM/dd/yyyy HH*: mm*'.

Table 2.10: Standard Date and Time Format Specifiers

Code Snippet 19 demonstrates the conversion of a specified date and time using the d, D, f, F, and g date and time format specifiers.

Code Snippet 19:

```
DateTime dt = DateTime.Now;

// Returns short date (MM/DD/YYYY)
Console.WriteLine("Short date format (d) : {0:d}", dt);

// Returns long date (Day, Month Date, Year)
Console.WriteLine("Long date format (D) : {0:D}", dt);

// Returns full date with time without seconds
Console.WriteLine("Full date with time without seconds (f) :{0:f}", dt);

// Returns full date with time with seconds
Console.WriteLine("Full date with time with seconds (F) :{0:F}", dt);

// Returns short date and short time without seconds
Console.WriteLine("Short date and short time without seconds (g) :{0:g}", dt);
```

Output:

```
Short date format (d) : 23/04/2007
Long date format (D) : Monday, April 23, 2007
Full date with time without seconds (f) : Monday, April 23, 2007 12:58 PM
Full date with time with seconds (F) : Monday, April 23, 2007 12:58:43 PM
Short date and short time without seconds (g) :23/04/2007 12:58 PM
```

2.5.12 Additional Standard Date and Time Format Specifiers

Table 2.11 displays additional date and time format specifiers in C#.

Format Specifier	Name	Description
G	General date/time (long time)	Displays date in short date and long time patterns, separated by a space. The default format is 'MM/dd/yyyy HH*:mm*:ss*'.
m or M	Month day	Displays only month and day of the date. The default format is 'MMMM* dd'.
T	Short time	Displays time in short time pattern. The default format is 'HH*: mm*'.
T	Long time	Displays time in long time pattern. The default format is 'HH*:mm*:ss*'.

Format Specifier	Name	Description
y or Y	Year month pattern	Displays only month and year from the date. The default format is 'YYYY MMMM*'.

Table 2.11: Additional Standard Date and Time Formats

Code Snippet 20 demonstrates the conversion of a specified date and time using the G, m, t, T, and y date and time format specifiers.

Code Snippet 20:

```
DateTime dt = DateTime.Now;
// Returns short date and short time with seconds
Console.WriteLine("Short date and short time with seconds (G) :{0:G}", dt);
// Returns month and day - M can also be used
Console.WriteLine("Month and day (m) :{0:m}", dt);
// Returns short time
Console.WriteLine("Short time (t) :{0:t}", dt);
// Returns short time with seconds
Console.WriteLine("Short time with seconds (T) :{0:T}", dt);
// Returns year and month - Y also can be used
Console.WriteLine("Year and Month (y) :{0:y}", dt);
```

Output:

```
Short date and short time with seconds (G) :23/04/2007 12:58:43 PM
Month and day (m) :April 23
Short time (t) :12:58 PM
Short time with seconds (T) :12:58:43 PM
Year and Month (y) :April, 2007
```

2.5.13 Custom DateTime Format Strings

Any non-standard DateTime format string is referred to as a custom DateTime format string. Custom DateTime format strings consist of more than one custom DateTime format specifiers.

Table 2.12 lists some of the custom DateTime format specifiers.

Format Specifier	Description
ddd	Represents the abbreviated name of the day of the week
dddd	Represents the full name of the day of the week
FF	Represents the two digits of the seconds fraction
H	Represents the hour from 0 to 23
HH	Represents the hour from 00 to 23
MM	Represents the month as a number from 01 to 12
MMM	Represents the abbreviated name of the month
S	Represents the seconds as a number from 0 to 59

Table 2.12: Custom DateTime Format Specifiers

Code Snippet 21 demonstrates the use of custom DateTime format specifiers.

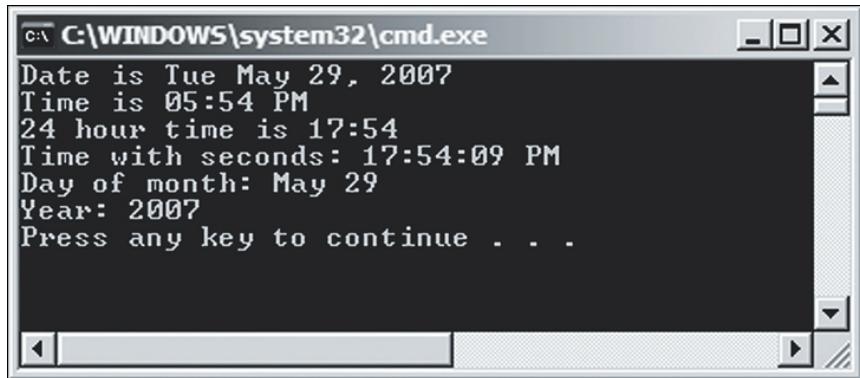
Code Snippet 21:

```
using System;

class DateTimeFormat
{
    public static void Main(string[] args)
    {
        DateTime date = DateTime.Now;
        Console.WriteLine("Date is {0:ddd MMM dd, yyyy}", date);
        Console.WriteLine("Time is {0:hh:mm tt}", date);
        Console.WriteLine("24 hour time is {0:HH:mm}", date);
        Console.WriteLine("Time with seconds: {0:HH:mm:ss tt}", date);
        Console.WriteLine("Day of month: {0:m}", date);
        Console.WriteLine("Year: {0:yyyy}", date);
    }
}
```

In this code, the date and time is displayed using the different DateTime format specifiers.

Figure 2.12 displays the output of using custom DateTime format specifiers.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text output:

```
Date is Tue May 29, 2007
Time is 05:54 PM
24 hour time is 17:54
Time with seconds: 17:54:09 PM
Day of month: May 29
Year: 2007
Press any key to continue . . .
```

Figure 2.12: Output of using Custom DateTime Format Specifiers

2.6 Check Your Progress

1. Which of these statements about the rules for naming variables in C# are true?

(A)	The name of the variable can contain letters and digits.
(B)	The first character in the variable name can be a digit.
(C)	The name of the variable can contain an underscore.
(D)	The name of the variable can contain keywords.
(E)	The variable names <code>RectLeng</code> and <code>rectleng</code> are the same.

(A)	A	(C)	C
(B)	B, C, D	(D)	A, C

2. Match the data types with the optimal values they can hold from the list.

Data Type		Value	
(A)	<code>byte</code>	(1)	125
(B)	<code>sbyte</code>	(2)	30000
(C)	<code>short</code>	(3)	244
(D)	<code>ushort</code>	(4)	800000
(E)	<code>int</code>	(5)	5000

(A)	A-3, B-1, C-5, D-2, E-4	(C)	A-5, B-1, C-3, D-2, E-4
(B)	A-1, B-3, C-5, D-2, E-4	(D)	A-4, B-2, C-5, D-1, E-3

3. Which of these statements about comments and XML comments in C# are true?

(A)	You can insert multi-line comments by starting the comment with double slash (//).
(B)	You cannot insert infinite number of comments.
(C)	You can insert XML comments by starting the comment with a double slash (//).
(D)	You can specify the parameters of a method using the <code><param></code> tag.
(E)	You can extract XML comments to an XML file.

(A)	A	(C)	C
(B)	C, D	(D)	A, D

4. Which of these statements about constants and literals are true?

(A)	Constants can be initialized after you declare identifiers.
(B)	Constants cannot be identified by the compiler at the time of compilation.
(C)	Literals can be of any data type. Literals can be of any data type.
(D)	Integer literals can be of float type.
(E)	Verbatim string literals can be prefixed by the '@' character.

(A)	C, E	(C)	C
(B)	B, C, D	(D)	A, D

5. Which of these statements about the keywords and escape sequence characters used in C# are true?

(A)	Keywords are used to avoid any conflicts during compile time.
(B)	Keywords cannot be modified in C#.
(C)	Escape sequence characters are prefixed with the '/' character.
(D)	Escape sequence characters are enclosed in single quotes.
(E)	The backslash character can be displayed using the respective escape sequence character.

(A)	A	(C)	C
(B)	B, C, D	(D)	A, B, E

6. Match the escape sequence characters against their corresponding non-printing characters.

Non-Printing Character		Escape Sequence Character	
(A)	Unicode character for hexadecimal values	(1)	\r
(B)	Carriage return	(2)	\t
(C)	Hexadecimal notation	(3)	\xhh
(D)	Horizontal tab	(4)	\0
(E)	Unicode character	(5)	\uhhhh

(A)	A-5, B-1, C-3, D-2, E-4	(C)	A-2, B-1, C-3, D-5, E-4
(B)	A-1, B-2, C-3, D-5, E-4	(D)	A-5, B-1, C-3, D-2, E-4

7. Which of these statements about input methods and format specifiers are true?

(A)	The <code>Read()</code> method always inserts the new line character at the end of the read character.
(B)	The data type conversion methods exist in the <code>System</code> class.
(C)	Format specifiers in C# enable you to display customized output.
(D)	Format specifiers require input methods of C#.
(E)	Format specifiers allow you to change a numeric value to a date-time value.

(A)	A	(C)	C
(B)	B, C, D	(D)	A, B, E

8. Can you match the date and time format specifiers with their corresponding formats?

Formats		Date and Time Format Specifiers	
(A)	MM/dd/yyyyHH:mm:ss	(1)	F
(B)	MMMM,yyyy	(2)	M
(C)	MMMM,dd	(3)	Y
(D)	MM/dd/yyyy	(4)	G
(E)	dddd,MMMMdd,yyyyHH:mm	(5)	D

(A)	A-4, B-3, C-2, D-5, E-1	(C)	A-1, B-5, C-2, D-3, E-4
(B)	A-3, B-4, C-2, D-5, E-1	(D)	A-1, B-4, C-2, D-5, E-4

2.6.1 Answers

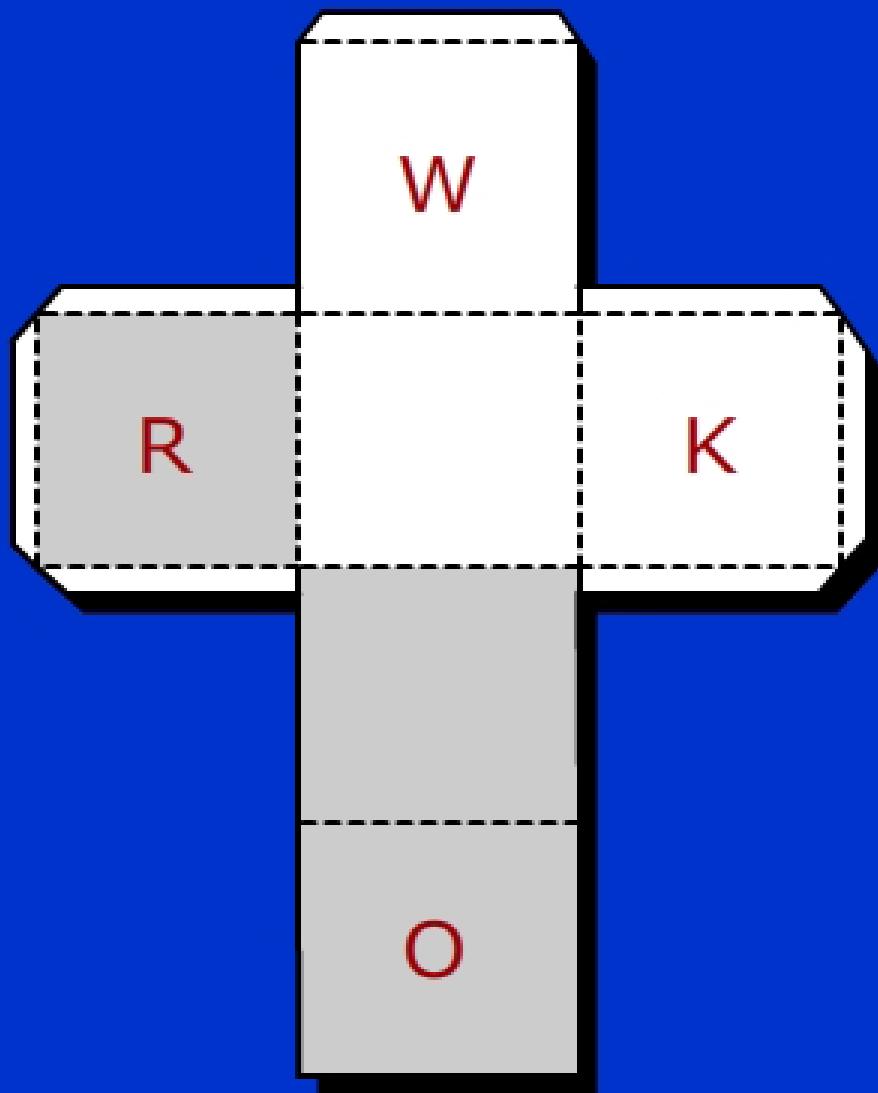
1.	D
2.	A
3.	B
4.	A
5.	D
6.	A
7.	C
8.	A



Summary

- ➔ A variable is a named location in the computer's memory and stores values.
- ➔ Comments are used to provide detailed explanation about the various lines in a code.
- ➔ Constants are static values that you cannot change throughout the program execution.
- ➔ Keywords are special words pre-defined in C# and they cannot be used as variable names, method names, or class names.
- ➔ Escape sequences are special characters prefixed by a backslash that allow you to display non-printing characters.
- ➔ Console operations are tasks performed on the command line interface using executable commands.
- ➔ Format specifiers allow you to display customized output in the console window.

WORK



ASSIGNMENT:

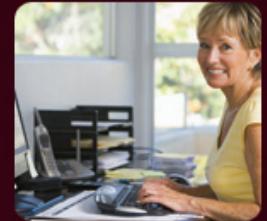
Form the cube to read '**WORK**'.

"Practice does not make perfect. Only perfect practice makes perfect."

- Vince Lombardi

Practice the Practicals for Perfection @

www.onlinevarsity.com



Session - 3

Statements and Operators

Welcome to the Session, **Statements and Operators**.

Statements help you in building a program and operators enable to perform calculations and computations. A statement can include operators, variables of different data types, and expressions. In C#, variables of low-precision data types can be converted to high-precision data types and vice-versa.

In this session, you will learn to:

- ➔ Define and describe statements and expressions
- ➔ Explain the types of operators
- ➔ Explain the process of performing data conversions in C#

3.1 Statements and Expressions

A C# program is a set of tasks that perform to achieve the overall functionality of the program. To perform the tasks, programmers provide instructions. These instructions are called statements. A C# statement can contain expressions that evaluates to a value.

3.1.1 Statements - Definition

Statements are referred to as logical grouping of variables, operators, and C# keywords that perform a specific task. For example, the line which initializes a variable by assigning it a value is a statement.

In C#, a statement ends with a semicolon. A C# program contains multiple statements grouped in blocks. A block is a code segment enclosed in curly braces. For example, the set of statements included in the `Main ()` method of a C# code is a block.

Figure 3.1 displays the definition.

```
class Circle
{
    static void Main(string[] args)
    {
        const float _pi = 3.14F;
        float radius = 5;
        float area = _pi * radius * radius;
        Console.WriteLine("Area of the circle is " + area);
    }
}
```



Figure 3.1: Definition

Note - A method in C# is equivalent to a function in earlier programming languages such as C and C++.

3.1.2 Statements - Uses

Statements are used to specify the input, the process, and the output tasks of a program. Statements can consist of:

- ➔ Data types
- ➔ Variables
- ➔ Operators
- ➔ Constants
- ➔ Literals
- ➔ Keywords
- ➔ Escape sequence characters

Statements help you build a logical flow in the program. With the help of statements, you can:

- Initialize variables and objects
- Take the input
- Call a method of a class
- Perform calculations
- Display the output

Code Snippet 1 shows an example of a statement in C#.

Code Snippet 1:

```
double area = 3.1452 * radius * radius;
```

This line of code is an example of a C# statement. The statement calculates the area of the circle and stores the value in the variable `area`.

Code Snippet 2 shows an example of a block of statements in C#.

Code Snippet 2:

```
...
{
    int side = 10;
    int height = 5;
    double area = 0.5 * side * height;
    Console.WriteLine("Area: ", area);
}
```

These lines of code show a block of code enclosed within curly braces. The first statement from the top will be executed first followed by the next statement and so on.

Code Snippet 3 shows an example of nested blocks in C#.

Code Snippet 3:

```
...
{
    int side = 5;
    int height = 10;
```

```
double area;  
...  
{  
    area = 0.5 * side * height;  
}  
Console.WriteLine(area);  
}
```

These lines of code show another block of code nested within a block of statements. The first three statements from the top will be executed in sequence. Then, the line of code within the inside braces will be executed to calculate the area. The execution is terminated at the last statement in the block of the code displaying the area.

3.1.3 Types of Statements

C# statements are similar to statements in C and C++. C# statements are classified into seven categories according to the function they perform. These categories are as follows:

→ Selection Statements

A selection statement is a decision-making statement that checks whether a particular condition is true or false. The keywords associated with this statement are `if`, `else`, `switch`, and `case`.

→ Iteration Statements

An iteration statement helps you to repeatedly execute a block of code. The keywords associated with this statement are: `do`, `for`, `foreach`, and `while`.

→ Jump Statements

A jump statement helps you transfer the flow from one block to another block in the program. The keywords associated with this statement are: `break`, `continue`, `default`, `goto`, `return`, and `yield`.

→ Exception Handling Statements

An exception handling statement manages unexpected situations that hinder the normal execution of the program. For example, if the code is dividing a number by zero, the program will not execute correctly. To avoid this situation, you can use exception handling statements. The keywords associated with this statement are: `throw`, `try-catch`, `try-finally`, and `try-catch-finally`.

→ Checked and Unchecked Statements

The checked and unchecked statements manage arithmetic overflows.

An arithmetic overflow occurs if the resultant value is greater than the range of the target variable's data type. The checked statement halts the execution of the program whereas the unchecked statement assigns junk data to the target variable. The keywords associated with these statements are `checked` and `unchecked`.

→ Fixed Statement

The fixed statement is required to tell the garbage collector not to move that object during execution. The keywords associated with this statement are `fixed` and `unsafe`.

→ Lock Statement

A lock statement helps in locking the critical code blocks. This ensures that no other process or threads running in the computer memory can interfere with the code. These statements ensure security and only work with reference types. The keyword associated with this statement is `lock`.

3.1.4 Checked and Unchecked Statements

The checked statement checks for an arithmetic overflow in arithmetic expressions. On the contrary, the unchecked statement does not check for an arithmetic overflow. An arithmetic overflow occurs if the result of an expression or a block of code is greater than the range of the target variable's data type. This causes the program to throw an exception that is caught by the `OverflowException` class. Exceptions are run-time errors that disrupt the normal flow of the program. The `System.Exception` class is used to derive several exception classes that handle the different types of exceptions.

The checked statement is associated with the `checked` keyword. When an arithmetic overflow occurs, the checked statement halts the execution of the program.

A checked statement creates a checked context for a block of statements and has the following form:

`checked-statement:`

`checked block`

Code Snippet 4 creates a class `Addition` with the checked statement. This statement throws an overflow exception.

Code Snippet 4:

```
using System;
class Addition
{
    public static void Main()
    {
        byte numOne = 255;
```

```
byte numTwo = 1;  
byte result = 0;  
  
try  
{  
    //This code throws an overflow exception  
    checked  
    {  
        result = (byte) (numOne + numTwo);  
    }  
    Console.WriteLine("Result: " + result);  
}  
  
catch (OverflowException ex)  
{  
    Console.WriteLine(ex);  
}  
}  
}
```

In Code Snippet 4, two numbers, `numOne` and `numTwo`, are added. The variables `numOne` and `numTwo` are declared as `byte` and are assigned values 255 and 1 respectively. When the statement within the checked block is executed, it gives an error because the result of the addition of the two numbers results in 256, which is too large to be stored in the `byte` variable. This causes an arithmetic overflow to occur.

Figure 3.2 shows the error message that is displayed after executing Code Snippet 4.

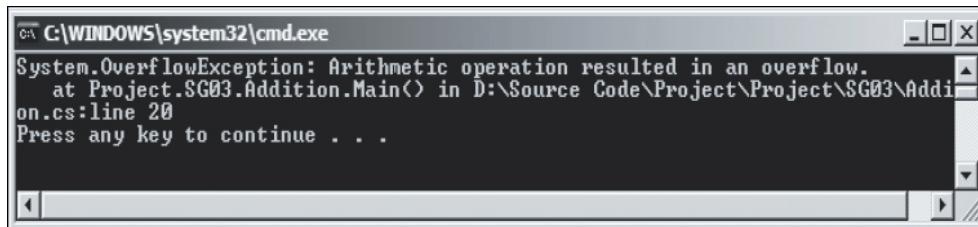


Figure 3.2: Error Message

The `unchecked` statement is associated with the `unchecked` keyword. The `unchecked` statement ignores the arithmetic overflow and assigns junk data to the target variable.

An unchecked statement creates an unchecked context for a block of statements and has the following form:

unchecked-statement:

unchecked block

Code Snippet 5 creates a class **Addition** that uses the unchecked statement.

Code Snippet 5:

```
using System;
class Addition
{
    public static void Main()
    {
        byte numOne = 255;
        byte numTwo = 1;
        byte result = 0;
        try
        {
            unchecked
            {
                result = (byte) (numOne + numTwo);
            }
            Console.WriteLine("Result: " + result);
        }
        catch (OverflowException ex)
        {
            Console.WriteLine(ex);
        }
    }
}
```

In Code Snippet 5, when the statement within the unchecked block is executed, the overflow that is generated is ignored by the unchecked statement and it returns an unexpected value.

The checked and unchecked statements are similar to the checked and unchecked operators. The only difference is that the statements operate on blocks instead of expressions.

3.1.5 Expressions - Definition

Expressions are used to manipulate data. Like in mathematics, expressions in programming languages, including C#, are constructed from the operands and operators. An expression statement in C# ends with a semicolon (;).

Expressions are used to:

- Produce values.
- Produce a result from an evaluation.
- Form part of another expression or a statement.

Code Snippet 6 demonstrates an example for expressions.

Code Snippet 6:

```
simpleInterest = principal * time * rate / 100;
eval = 25 + 6 - 78 * 5;
num++;
```

In the first two lines of code, the results of the statements are stored in the variables `SimpleInterest` and `eval`. The last statement increments the value of the variable `num`.

3.1.6 Difference between Statements and Expressions

There are some fundamental differences between statements and expressions. Table 3.1 displays these differences.

Statements	Expressions
Do not necessarily return values. For example, consider the following statement: int oddNum = 5; The statement only stores the value 5 in the <code>oddNum</code> variable.	Always evaluates to a values. For example, consider the following expression: $100 * (25 * 10)$ The expression evaluates to the value 2500.
Statements are executed by the compiler.	Expressions are part of statements and are evaluated by the compiler.

Table 3.1: Difference between Statements and Expressions

3.2 Operators

Expressions in C# comprise of one or more operators that performs some operations on variables. An operation is an action performed on single or multiple values stored in variables in order to modify them

or to generate a new value. Therefore, an operation takes place with the help of minimum one symbol and a value. The symbol is called an operator and it determines the type of action to be performed on the value. The value on which the operation is to be performed is called an operand.

An operand might be a complex expression. For example, $(X * Y) + (X - Y)$ is a complex expression, where the $+$ operator is used to join two operands.

3.2.1 Types of Operators

Operators are used to simplify expressions. In C#, there is a predefined set of operators used to perform various types of operations. C# operators are classified into six categories based on the action they perform on values.

These six categories of operators are as follows:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Conditional Operators
- Increment and Decrement Operators
- Assignment Operators

3.2.2 Arithmetic Operators - Types

Arithmetic operators are binary operators because they work with two operands, with the operator being placed in between the operands. These operators allow you to perform computations on numeric or string data.

Table 3.2 lists the arithmetic operators along with their descriptions and an example of each type.

Operators	Description	Examples
+ (Addition)	Performs addition. If the two operands are strings, then it functions as a string concatenation operator and adds one string to the end of the other.	40 + 20
- (Subtraction)	Performs subtraction. If a greater value is subtracted from a lower value, the resultant output is a negative value.	100 - 47

Operators	Description	Examples
* (Multiplication)	Performs multiplication.	67 * 46
/ (Division)	Performs division. The operator divides the first operand by the second operand and gives the quotient as the output.	12000 / 10
% (Modulo)	Performs modulo operation. The operator divides the two operands and gives the remainder of the division operation as the output.	100 % 33

Table 3.2: Arithmetic Operator Types

Code Snippet 7 demonstrates how to use the arithmetic operators.

Code Snippet 7:

```
int valueOne = 10;
int valueTwo = 2;
int add = valueOne + valueTwo;
int sub = valueOne - valueTwo;
int mult = valueOne * valueTwo;
int div = valueOne / valueTwo;
int modu = valueOne % valueTwo;
Console.WriteLine("Addition " + add);
Console.WriteLine("Subtraction " + sub);
Console.WriteLine("Multiplication " + mult);
Console.WriteLine("Division " + div);
Console.WriteLine("Remainder " + modu);
```

Output:

Addition 12

Subtraction 8

Multiplication 20

Division 5

Remainder 0

3.2.3 Relational Operators

Relational operators make a comparison between two operands and return a boolean value, true or false.

Table 3.3 lists the relational operators along with their descriptions and an example of each type.

Relational Operators	Description	Examples
==	Checks whether the two operands are identical.	85 == 95
!=	Checks for inequality between two operands.	35 != 40
>	Checks whether the first value is greater than the second value.	50 > 30
<	Checks whether the first value is lesser than the second value.	20 < 30
>=	Checks whether the first value is greater than or equal to the second value.	100 >= 30
<=	Checks whether the first value is lesser than or equal to the second value.	75 <= 80

Table 3.3: Relational Operator Types

Code Snippet 8 demonstrates how to use the relational operators.

Code Snippet 8:

```
int leftVal = 50;
int rightVal = 100;
Console.WriteLine("Equal: " + (leftVal == rightVal));
Console.WriteLine("Not Equal: " + (leftVal != rightVal));
Console.WriteLine("Greater: " + (leftVal > rightVal));
Console.WriteLine("Lesser: " + (leftVal < rightVal));
Console.WriteLine("Greater or Equal: " + (leftVal >= rightVal));
Console.WriteLine("Lesser or Equal: " + (leftVal <= rightVal));
```

Output:

```
Equal: False
Not Equal: True
Greater: False
Lesser: True
Greater or Equal: False
Lesser or Equal: True
```

3.2.4 Logical Operators

Logical operators are binary operators that perform logical operations on two operands and return a boolean value. C# supports two types of logical operators, boolean and bitwise.

→ Boolean Logical Operators

The boolean logical operators perform boolean logical operations on both the operands. They return a boolean value based on the logical operator used.

Table 3.4 lists the boolean logical operators along with their descriptions and an example of each type.

Logical Operator	Description	Example
& (Boolean AND)	Returns true if both the expressions are evaluated to true.	(percent >= 75) & (percent <= 100)
(Boolean Inclusive OR)	Returns true if at least one of the expressions is evaluated to true.	(choice == 'Y') (choice == 'y')
^ (Boolean Exclusive OR)	Returns true if only one of the expressions is evaluated to true. If both the expressions evaluate to true, the operator returns false.	(choice == 'Q') ^ (choice == 'q')

Table 3.4: Logical Operator Types

Code Snippet 9 explains the use of the boolean inclusive OR operator.

Code Snippet 9:

```
if ((quantity>2000) | (price<10.5))
{
    Console.WriteLine ("You can buy more goods at a lower price");
}
```

In the code, the boolean inclusive OR operator checks both the expressions. If either one of them evaluates to true, the complete expression returns true and the statement within the block is executed.

Code Snippet 10 explains the use of the boolean AND operator.

Code Snippet 10:

```
if ((quantity==2000) & (price==10.5))
{
    Console.WriteLine ("The goods are correctly priced");
}
```

In the code, the boolean AND operator checks both the expressions. If both the expressions evaluate to true, the complete expression returns true and the statement within the block is executed.

Code Snippet 11 explains the use of the boolean exclusive OR operator.

Code Snippet 11:

```
if ((quantity == 2000) ^ (price == 10.5))
{
    Console.WriteLine ("You have to compromise between quantity and
    price");
}
```

In the code, the boolean exclusive OR operator checks both the expressions. If only one of them evaluates to true, the complete expression returns true and the statement within the block is executed. If both of them are true, the expression returns false.

→ Bitwise Logical Operators

The bitwise logical operators perform logical operations on the corresponding individual bits of two operands. Table 3.5 lists the bitwise logical operators along with their descriptions and an example of each type.

Logical Operator	Description	Example
& (Bitwise AND)	Compares two bits and returns 1 if both bits are 1, else returns 0.	00111000 & 00011100
(Bitwise Inclusive OR)	Compares two bits and returns 1 if either of the bits is 1.	00010101 00011110
^ (Bitwise Exclusive OR)	Compares two bits and returns 1 if only one of the bits is 1.	00001011 ^ 00011110

Table 3.5: Bitwise Logical Operators

Code Snippet 12 explains the working of the bitwise AND operator.

Code Snippet 12:

```
result = 56 & 28; // (56 = 00111000 and 28 = 00011100)
Console.WriteLine(result);
```

In Code Snippet 12, the bitwise AND operator compares the corresponding bits of the two operands. It returns 1 if both the bits in that position are 1 or else returns 0. This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number. This number is automatically converted to integer, which is displayed as the output. The resultant output for the code is “24”.

Figure 3.3 displays the bitwise logical operators.

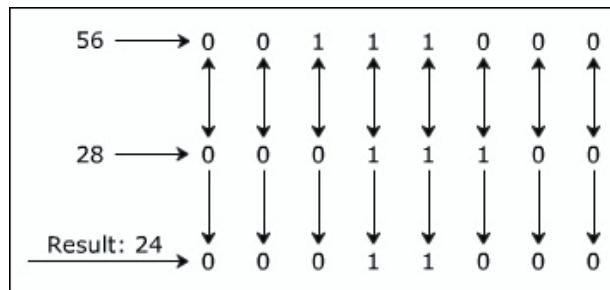


Figure 3.3: Bitwise Logical Operators

Code Snippet 13 explains the working of the bitwise inclusive OR operator.

Code Snippet 13:

```
result = 56 | 28;
Console.WriteLine(result);
```

The bitwise inclusive OR operator compares the corresponding bits of the two operands. It returns 1 if either of the bits in that position is 1, else it returns 0. This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number. This number is automatically converted to integer, which is displayed as the output. The resultant output for the code is 60.

Code Snippet 14 explains the working of the bitwise exclusive OR operator.

Code Snippet 14:

```
result = 56 ^ 28;
Console.WriteLine(result);
```

The bitwise exclusive OR operator compares the corresponding bits of the two operands. It returns 1 if only 1 of the bits in that position is 1, else it returns 0. This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number. This number is automatically converted to integer, which is displayed as the output. The resultant output for the code is "36".

3.2.5 Conditional Operators

There are two types of conditional operators, conditional AND (`&&`) and conditional OR (`||`). Conditional operators are similar to the boolean logical operators but has the following differences.

- The conditional AND operator evaluates the second expression only if the first expression returns true. This is because this operator returns true only if both expressions are true. Thus, if the first expression itself evaluates to false, the result of the second expression is immaterial.
- The conditional OR operator evaluates the second expression only if the first expression returns false. This is because this operator returns true if either of the expressions is true. Thus, if the first

expression itself evaluates to true, the result of the second expression is immaterial. Figure 3.4 displays the conditional operators.

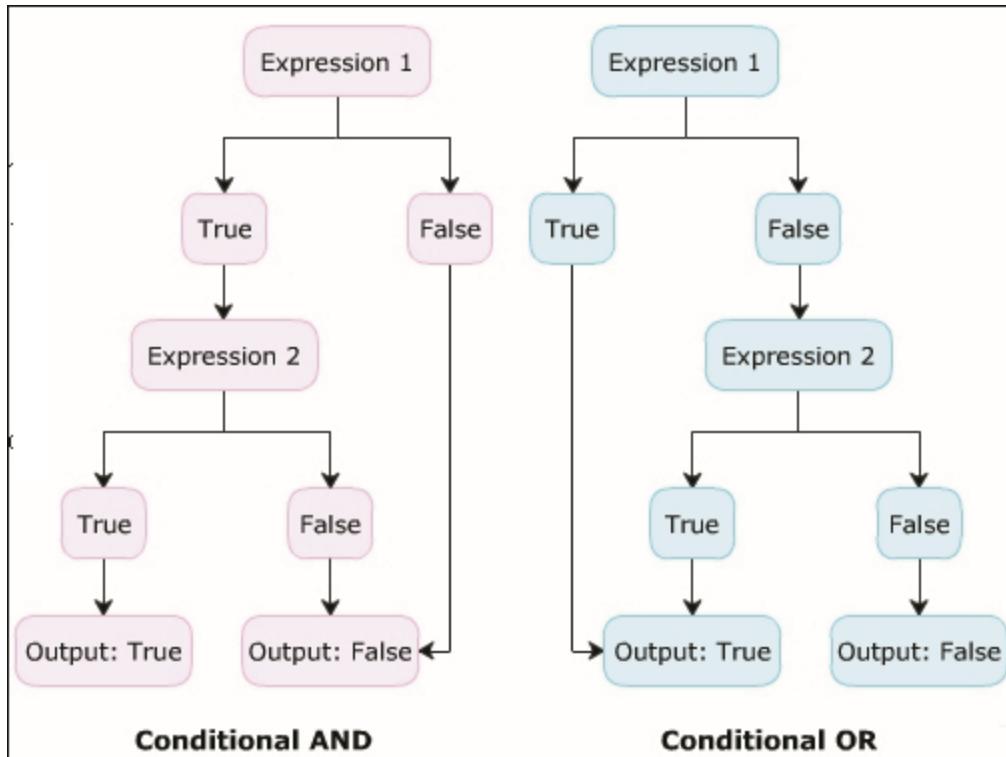


Figure 3.4: Conditional Operators

Code Snippet 15 displays the use of the conditional AND (`&&`) operator.

Code Snippet 15:

```

int num=0;
if (num>=1 && num<=10)
{
    Console.WriteLine("The number exists between 1 and 10");
}
else
{
    Console.WriteLine("The number does not exist between 1 and 10");
}

```

In the code, the conditional AND operator checks the first expression. The first expression returns false, Therefore, the operator does not check the second expression. The whole expression evaluates to false, the statement in the else block is executed.

Output:

The number does not exist between 1 and 10

Code Snippet 16 displays the use of the conditional OR (||) operator.

Code Snippet 16:

```
int num = -5;

if (num < 0 || num > 10)

{
    Console.WriteLine ("The number does not exist between 1 and 10");
}

else

{
    Console.WriteLine ("The number exists between 1 and 10");
}
```

In the code, the conditional OR operator checks the first expression. The first expression returns true, Therefore, the operator does not check the second expression. The whole expression evaluates to true, the statement in the `if` block is executed.

Output:

The number does not exist between 1 and 10

3.2.6 Increment and Decrement Operators

Two of the most common calculations performed in programming are increasing and decreasing the value of the variable by 1. In C#, the increment operator (++) is used to increase the value by 1 while the decrement operator (--) is used to decrease the value by 1.

If the operator is placed before the operand, the expression is called pre-increment or pre-decrement. If the operator is placed after the operand, the expression is called post-increment or post-decrement.

Table 3.6 depicts the use of increment and decrement operators assuming the value of the variable `valueOne` is 5.

Expression	Type	Result
<code>valueTwo = ++valueOne;</code>	Pre-Increment	<code>valueTwo = 6</code>
<code>valueTwo = valueOne++;</code>	Post-Increment	<code>valueTwo = 5</code>
<code>valueTwo = --valueOne;</code>	Pre-Decrement	<code>valueTwo = 4</code>
<code>valueTwo = valueOne--;</code>	Post-Decrement	<code>valueTwo = 5</code>

Table 3.6: Increment and Decrement Operators

3.2.7 Assignment Operators - Types

Assignment operators are used to assign the value of the right side operand to the operand on the left side using the equal to operator (`=`). The assignment operators are divided into two categories in C#. These are as follows:

- **Simple assignment operators:** The simple assignment operator is `=`, which is used to assign a value or result of an expression to a variable.
- **Compound assignment operators:** The compound assignment operators are formed by combining the simple assignment operator with the arithmetic operators.

Table 3.7 shows the use of assignment operators assuming the value of the variable `valueOne` is 10.

Expression	Description	Result
<code>valueOne += 5;</code>	<code>valueOne = valueOne + 5</code>	<code>valueOne = 15</code>
<code>valueOne -= 5;</code>	<code>valueOne = valueOne - 5</code>	<code>valueOne = 5</code>
<code>valueOne *= 5;</code>	<code>valueOne = valueOne * 5</code>	<code>valueOne = 50</code>
<code>valueOne %= 5;</code>	<code>valueOne = valueOne % 5</code>	<code>valueOne = 0</code>
<code>valueOne /= 5;</code>	<code>valueOne = valueOne / 5</code>	<code>valueOne = 2</code>

Table 3.7: Use of Assignment Operators

Code Snippet 17 demonstrates how to use assignment operators.

Code Snippet 17:

```
int valueOne=5;
int valueTwo=10;
Console.WriteLine("Value1=" + valueOne);
valueOne += 4;
Console.WriteLine("Value1 += 4=" + valueOne);
valueOne -= 8;
Console.WriteLine("Value1 -= 8=" + valueOne);
valueOne *= 7;
Console.WriteLine("Value1 *= 7=" + valueOne);
valueOne /= 2;
Console.WriteLine("Value1 /= 2=" + valueOne);
Console.WriteLine("Value1 == Value2: {0}", (valueOne==valueTwo));
```

Output:

Value1 =5

```

Value1 += 4= 9
Value1 -= 8= 1
Value1 *= 7= 7
Value1 /= 2= 3
Value1 == Value2: False

```

Note - The assignment operator is different from the equality operator (==). This is because the equality operator returns true if the values of both the operands are equal, else returns false.

3.2.8 Precedence and Associativity

Operators in C# have certain associated priority levels. The C# compiler executes operators in the sequence defined by the priority level of the operators. For example, the multiplication operator (*) has higher priority over the addition (+) operator. Thus, if an expression involves both the operators, the multiplication operation is carried out before the addition operation. In addition, the execution of the expression (associativity) is either from left to right or vice-versa depending upon the operators used.

Table 3.8 lists the precedence of the operators, from the highest to the lowest precedence, their description and their associativity.

Precedence (where 1 is the highest)	Operator	Description	Associativity
1	()	Parentheses	Left to Right
2	++ or --	Increment or Decrement	Right to Left
3	*, /, %	Multiplication, Division, Modulus	Left to Right
4	+, -	Addition, Subtraction	Left to Right
5	<, <=, >, >=	Less than, Less than or equal to, Greater than, Greater than or equal to	Left to Right
6	=, !=	Equal to, Not Equal to	Left to Right
7	&&	Conditional AND	Left to Right
8		Conditional OR	Left to Right
9	=, +=, -=, *=, /=, %=	Assignment Operators	Right to Left

Table 3.8: Precedence of Operators

Code Snippet 18 demonstrates how to use logical operators.

Code Snippet 18:

```
int valueOne = 10;
Console.WriteLine( (4 * 5 - 3) / 6 + 7 - 8 % 5 );
Console.WriteLine( (32 < 4) || (8 == 8) );
Console.WriteLine( ((valueOne *= 6) > (valueOne += 5)) &&
((valueOne /= 2) != (valueOne -= 5)) );
```

In the code, the variable **valueOne** is initialized to the value 10. The next three statements display the results of the expressions. The expression given in the parentheses is solved first.

Output:

```
6
True
False
```

3.2.9 Shift Operators

The shift operators allow the programmer to perform shifting operations. The two shifting operators are the left shift (`<<`) and the right shift (`>>`) operators. The left shift operator allows shifting the bit positions towards the left side where the last bit is truncated and zero is added on the right. The right shift operator allows shifting the bit positions towards the right side and the zero is added on the left.

Code Snippet 19 demonstrates the use of shift operators.

Code Snippet 19:

```
using System;
class ShiftOperator
{
    static void Main(string[] args)
    {
        uint num = 100; // 01100100 = 100
        uint result = num << 1; // 11001000 = 200
        Console.WriteLine("Value before left shift : " + num);
        Console.WriteLine("Value after left shift " + result);
        num = 80; // 10100000 =
        result = num >> 1; // 01010000 = 40
```

```

        Console.WriteLine("\nValue before right shift : " + num);
        Console.WriteLine("Value after right shift : " + result);
    }
}

```

The output of Code Snippet 19 is shown in figure 3.5.

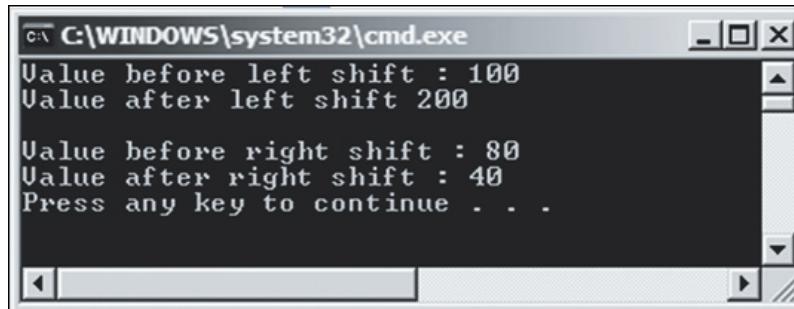


Figure 3.5: Output of Code Snippet 19

In Code Snippet 19, the `Main()` method of the class `ShiftOperator` performs the shifting operations.

The `Main()` method initializes the variable `num` to 100. After shifting towards left, the value of `num` is doubled. Now, the variable `num` is initialized to 80. After shifting towards right, the value of `num` is halved.

3.2.10 String Concatenation Operator

The arithmetic operator (+) allows the programmer to add numerical values. However, if one or more operands are characters or binary strings, columns, or a combination of strings and column names into one expression, then the string concatenation operator concatenates them. In other words, the arithmetic operator (+) is overloaded for string values.

Code Snippet 20 demonstrates the use of string concatenation operator.

Code Snippet 20:

```

using System;
class Concatenation
{
    static void Main(string[] args)
    {
        int num = 6;
        string msg = "";

```

```

if (num<0)
{
    msg = "The number " + num + " is negative";
}

else if ((num % 2) == 0)
{
    msg = "The number " + num + " is even";
}

else
{
    msg = "The number " + num + " is odd";
}

if (msg != "")
    Console.WriteLine(msg);
}
}

```

In Code Snippet 20, the `Main()` method of the class `Concatenation` uses the construct to check whether a number is even, odd or negative. Depending upon the condition, the code displays the output by using the string concatenation operator (+) to concatenate the strings with numbers.

The output of Code Snippet 20 shows the use of string concatenation operator, as seen in figure 3.6.

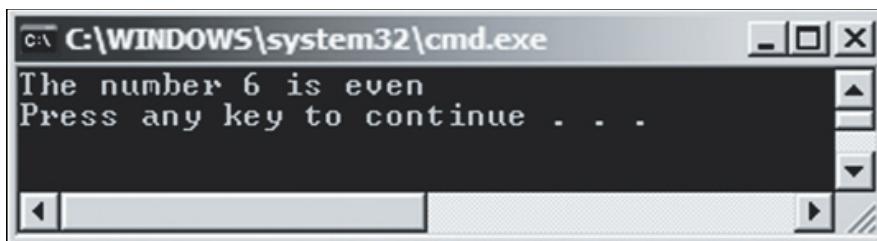


Figure 3.6: Output of Code Snippet 20

3.2.11 Ternary or Conditional Operator

The ?: is referred to as the conditional operator. It is generally used to replace the `if-else` constructs. Since it requires three operands, it is also referred to as the ternary operator. The first expression returns a `bool` value, and depending on the value returned by the first expression, the second or third expression is evaluated. If the first expression returns a true value, the second expression is evaluated, whereas if the first expression returns a false value, the third expression is evaluated.

Syntax:

```
<Expression 1> ? <Expression 2>: <Expression 3>;
```

where,

Expression 1: Is a bool expression.

Expression 2: Is evaluated if expression 1 returns a true value.

Expression 3: Is evaluated if expression 1 returns a false value.

Code Snippet 21 demonstrates the use of the ternary operator.

Code Snippet 21:

```
using System;

class LargestNumber
{
    public static void Main()
    {
        int numOne = 5;
        int numTwo = 25;
        int numThree = 15;
        int result = 0;
        if (numOne > numTwo)
        {
            result = (numOne > numThree) ? result = numOne : result =
            numThree;
        }
        else
        {
            result = (numTwo > numThree) ? result = numTwo : result =
            numThree;
        }
        if (result != 0)
            Console.WriteLine("{0} is the largest number", result);
    }
}
```

In Code Snippet 21, the `Main()` method of the class `LargestNumber` checks and displays the largest of three numbers, `numOne`, `numTwo`, and `numThree`. This largest number is stored in the variable `result`. If `numOne` is greater than `numTwo`, then the ternary operator within the `if` loop is executed.

The ternary operator (`?:`) checks whether `numOne` is greater than `numThree`. If this condition is true, then the second expression of the ternary operator is executed, which will assign `numOne` to `result`. Otherwise, if `numOne` is not greater than `numThree`, then the third expression of the ternary operator is executed, which will assign `numThree` to `result`.

Similar operation is done for comparison of `numOne` and `numTwo` and the `result` variable will contain the larger value out of these two.

Figure 3.7 shows the output of the example using ternary operator.

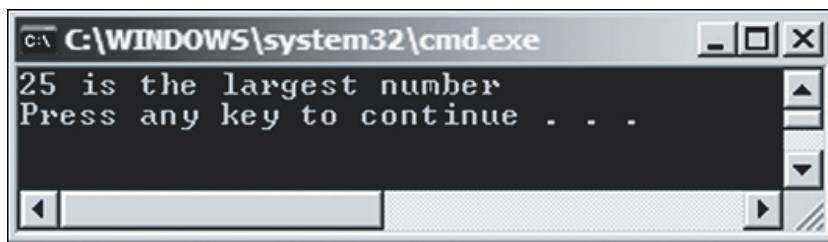


Figure 3.7: Output of Code Snippet 21

3.3 Data Conversions in C#

Data conversions is performed in C# through casting, a mechanism to convert one data type to another.

3.3.1 Typecasting and its Benefits

Consider the payroll system of an organization. The gross salary of an employee is calculated and stored in a variable of `float` type. Currently, the output is shown as `float` values. The payroll department wants the salary amount as a whole number and thus, wants any digits after the decimal point of the calculated salary to be ignored. The programmer is able to achieve this using the typecasting feature of C#. Typecasting allows you to change the data type of a variable.

C# supports two types of casting, namely Implicit and Explicit. Typecasting is mainly used to:

- Convert a data type to another data type that belongs to the same or a different hierarchy. For example, the numeric hierarchy includes `int`, `float`, and `double`. You can convert the `char` type into `int` type to display the ASCII value.
- Display the exact numeric output. For example, you can display exact quotients during mathematical divisions.
- Prevent loss of numeric data if the resultant value exceeds the range of its variable's data type.

3.3.2 Implicit Conversions for C# Data Types - Definition

Implicit typecasting refers to an automatic conversion of data types. This is done by the C# compiler. Implicit typecasting is done only when the destination and source data types belong to the same hierarchy. In addition, the destination data type must hold a larger range of values than the source data type. Implicit conversion prevents the loss of data as the destination data type is always larger than the source data type. For example, if you have a value of `int` type, you can assign that value to the variable of `long` type.

Code Snippet 22 shows an example of implicit conversion.

Code Snippet 22:

```
int valueOne = 34;  
float valueTwo;  
valueTwo = valueOne;
```

In Code Snippet 22, the compiler generates code that automatically converts the value in `valueOne` into a floating-point value before storing the result in `valueTwo`. Converting an integer value to a floating point value is safe.

Note - Typecasting returns a value as per the destination data type without changing the value of the variable, which is of the source data type. Implicit type conversion is also known as coercion.

3.3.3 Implicit Conversions for C# Data Types - Rules

Implicit typecasting is carried out automatically by the compiler. The C# compiler automatically converts a lower precision data type into a higher precision data type when the target variable is of a higher precision than the source variable.

Figure 3.8 illustrates the data types of higher precision to which they can be converted.

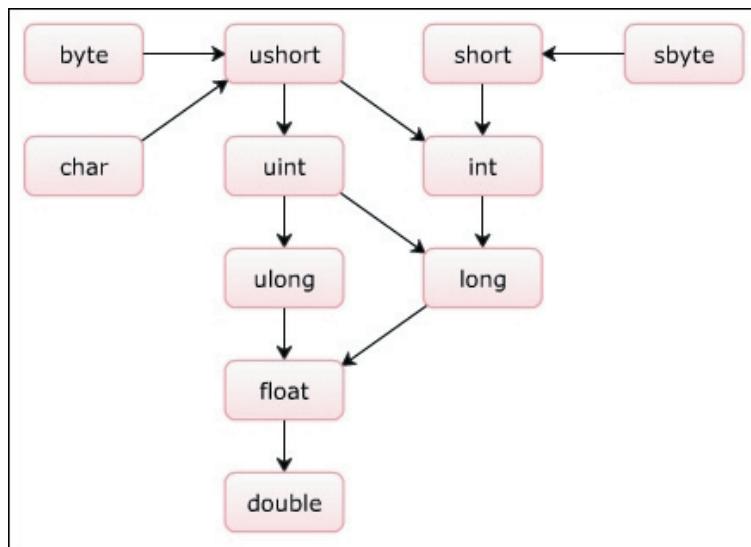


Figure 3.8: Data Types of Higher Precision

3.3.4 Explicit Type Conversion - Definition

Explicit typecasting refers to changing a data type of higher precision into a data type of lower precision. For example, using explicit typecasting, you can manually convert the value of `float` type into `int` type. This typecasting might result in loss of data. This is because when you convert the `float` data type into the `int` data type, the digits after the decimal point are lost.

The following is the syntax for explicit conversion.

Syntax:

`<target data type> <variable name> = (target data type)<source data type>;`

where,

`target data type`: Is the resultant data type.

`variable name`: Is the name of the variable, which is of the target data type.

`target data type`: Is the target data type in parentheses.

`source data type`: Is the data type which is to be converted.

Code Snippet 23 displays the use of explicit conversion for calculating the area of a square.

Code Snippet 23:

```
double side = 10.5;
int area;
area = (int) (side * side);
Console.WriteLine("Area of the square = {0}", area);
```

Output:

Area of the square = 110

3.3.5 Explicit Type Conversion - Implementation

There are two ways to implement explicit typecasting in C# using the built-in methods. These are using:

- **System.Convert class:** This class provides useful methods to convert any built-in data type to another built-in data type.

For example, `Convert.ToString(float)` method converts a float value into a `char` value.

- **ToString() method:** This method belongs to the `Object` class and converts any data type value into string.

Code Snippet 24 displays a float value as string using the `ToString()` method.

Code Snippet 24:

```
float flotNum = 500.25F;
string stNum = flotNum.ToString();
Console.WriteLine(stNum);
```

In the code, the value of `float` type is converted to string type using the `ToString()` method. The string is displayed as output in the console window.

Output:

500.25

Note - The `Object` class is the base class for all value types created in C#.

3.3.6 Boxing and Unboxing

Boxing is a process for converting a value type, like integers, to its reference type, like objects. This conversion is useful to reduce the overhead on the system during execution. This is because all value types are implicitly of `object` type.

To implement boxing, you need to assign the value type to an object. While boxing, the variable of type `object` holds the value of the value type variable. This means that the `object` type has the copy of the value type instead of its reference.

Boxing is done implicitly when a value type is provided instead of the expected reference type.

Figure 3.9 illustrates with an analogy the concept of boxing.

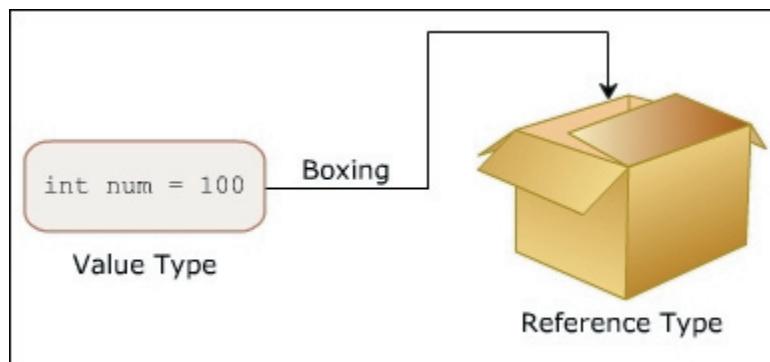


Figure 3.9: Boxing

The syntax for boxing is as follows:

Syntax:

`object <instance of the object class> = <variable of value type>;`

where,

`object`: Is the base class for all value types.

`instance of the object class`: Is the name referencing the `Object` class.

`variable of value type`: Is the identifier whose data type is of value type.

Code Snippet 25 demonstrates the use of implicit boxing.

Code Snippet 25:

```
int radius=10;
double area;
area=3.14 * radius * radius;
object boxed=area;
Console.WriteLine("Area of the circle = {0}",boxed);
```

Output:

Area of the circle = 314

In Code Snippet 25, implicit boxing occurs when the value of `double` variable, `area`, is assigned to an `object`, `boxed`.

Code Snippet 26 demonstrates the use of explicit boxing.

Code Snippet 26:

```

float radius = 4.5F;
double circumference;
circumference = 2 * 3.14 * radius;
object boxed = (object)circumference;
Console.WriteLine("Circumference of the circle = {0}", circumference);

```

In Code Snippet 26, explicit boxing occurs by casting the variable, `circumference`, which is assigned to object, `boxed`.

Output:

Circumference of the circle = 28.26

Unboxing refers to converting a reference type to a value type. The stored value inside an object is unboxed to the value type. The object type must be of the destination value type. This is explicit conversion, without which the program will give an error.

Figure 3.10 illustrates with an analogy the concept of boxing.

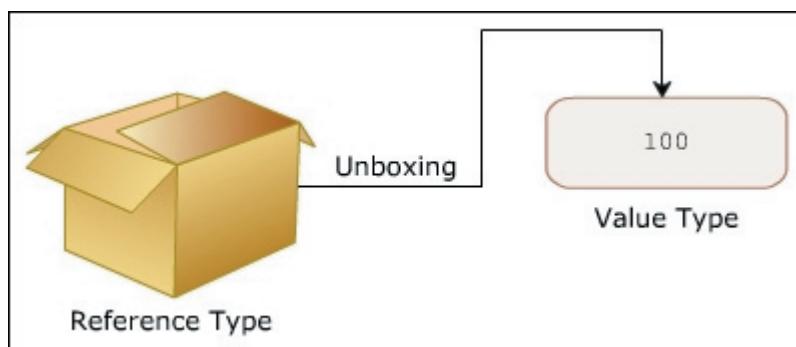


Figure 3.10: Unboxing

The syntax for unboxing is as follows:

Syntax:

<target value type> <variable name> = (target value type) <object type>;

where,

target value type: Is the resultant data type.

variable name: Is the name of the variable of value type.

target value type: Is the resultant value type in parentheses.

object type: Is the reference name of the Object class.

Code Snippet 27 demonstrates the use of unboxing while calculating the area of the rectangle.

Code Snippet 27:

```
int length=10;
int breadth=20;
int area;
area=length * breadth;
object boxed=area;
int num= (int)boxed;
Console.WriteLine("Area of the rectangle= {0}", num);
```

In Code Snippet 27, boxing is done when the value of the variable `area` is assigned to an object, `boxed`. However, when the value of the object, `boxed`, is to be assigned to `num`, unboxing is done explicitly because `boxed` is a reference type and `num` is a value type.

Code Snippet 28 demonstrates how boxing and unboxing occur.

Code Snippet 28:

```
using System;
class Number
{
    static void Main(string[] args)
    {
        int num=8;
        int result;
        result=Square(num);
        Console.WriteLine("Square of {0} = {1}", num, result);
    }
    static int Square(object inum)
    {
        return (int)inum * (int)inum;
    }
}
```

In Code Snippet 28, the `Main()` method calculates the square of the specified value. Boxing occurs when the variable `num`, which is of value type, is passed to the `Square()` method whose input parameter, `inum`, is of reference type.

Unboxing occurs when this reference type variable, `inum`, is converted to value type, `int`, before its square is returned to variable `result`.

Figure 3.11 shows the output of the code demonstrating boxing and unboxing.

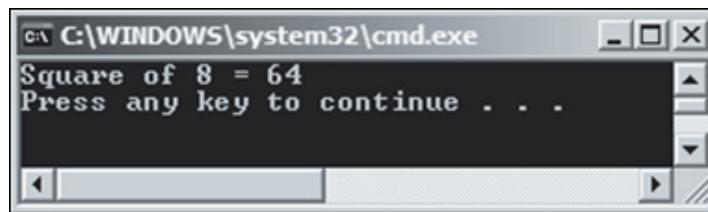


Figure 3.11: Example of Boxing and Unboxing

3.4 Check Your Progress

1. Which of these statements about expressions and statements are true?

(A)	Statements end with a comma.
(B)	Functions refer to related statements in a program.
(C)	Expressions are part of statements.
(D)	Jump statements help in executing another block of code repeatedly.
(E)	Expressions might not always return values.

(A)	A	(C)	C
(B)	B	(D)	D

2. Which of these following options are types of statements?

(A)	Block Statements
(B)	Selection Statements
(C)	Decision-making Statements
(D)	Logical Statements
(E)	Exception Handling Statements
(F)	Loop Statements
(G)	Iteration Statements
(H)	Error Handling Statements

(A)	B, E, G	(C)	C, D
(B)	B, C	(D)	D

3. Match the keywords in C# against their corresponding types of statements.

Types of Statements		Keywords	
(A)	Selection	(1)	throw
(B)	Exception handling	(2)	foreach
(C)	Iteration	(3)	return
(D)	Jump	(4)	checked
(E)	Checked	(5)	default

(A)	A-1, B-3, C-2, D-4, E-5	(C)	A-5, B-4, C-3, D-2, E-1
(B)	A-5, B-1, C-2, D-3, E-4	(D)	A-2, B-1, C-5, D-3, E-4

4. Which of the following statements on arithmetic and logical operators and operator precedence are true?

(A)	The conditional operator works with two operands.
(B)	The increment operator works with a single operand.
(C)	The bitwise exclusive OR operator compares bits and returns 1 if only one of the bits is 1.
(D)	The conditional && operator does not check the second expression if the first expression is false.
(E)	The compound assignment operators execute from right to left.

(A)	A, B	(C)	A, C, D
(B)	B, C, D, E	(D)	D

5. You are trying to manipulate the values in the variables `valueOne` and `valueTwo` using the increment and decrement operators such that the final value is 33 and is stored in the result variable. Which of the following codes will help you to achieve this?

(A)

```
int valueOne=10;
int valueTwo=20;
int result;
result=valueOne+valueTwo;
valueTwo++;
++valueOne;
result=valueOne+++valueTwo--;
Console.WriteLine(result);
```

(B)

```
int valueOne=10;
int valueTwo=20;
int result;
result=valueOne+valueTwo;
valueTwo++;
valueTwo=valueOne++;
result=++valueOne+--valueTwo;
Console.WriteLine(result);
```

(C)

```
int valueOne=10;
int valueTwo=20;
int result;
result=valueOne+valueTwo;
++valueTwo;
++valueOne;
result=++valueOne+++valueTwo;
Console.WriteLine(result);
```

<pre>(D) int valueOne=10; int valueTwo=20; int result; result=valueOne+valueTwo; valueTwo++; ++valueOne; result=++valueOne + valueTwo--; Console.WriteLine(result);</pre>

(A)	A	(C)	C
(B)	B	(D)	D

6. You are trying to convert and display the input from the console window to a 32-bit signed integer. Which of the following code will help you to achieve this?

<pre>(A) int input; Console.WriteLine("Enter a number : "); input=Convert.toInt32(Console.ReadLine()); Console.WriteLine(input);</pre>
<pre>(B) int input; Console.WriteLine("Enter a number : "); Input=System.ToInt(Console.ReadLine()); Console.WriteLine(input);</pre>
<pre>(C) int input; Console.WriteLine("Enter a number : "); input=Convert.ToInt32(Console.ReadLine()); Console.WriteLine(input);</pre>
<pre>(D) int input; Console.WriteLine("Enter a number : "); input=(toInt32)(Console.ReadLine()); Console.WriteLine(input);</pre>

(A)	A	(C)	C
(B)	B	(D)	D

7. You are trying to convert and display a reference type to the float type. Which of the following code will help you to achieve this?

(A)	object obj = 100.50; float num = float (obj); Console.WriteLine (num);
(B)	object obj = 100.50F; Float num = (Float) obj; Console.WriteLine (num);
(C)	object obj = 100.50; float num = obj (float); Console.WriteLine (num);
(D)	object obj = 100.50F; float num = (float) obj; Console.WriteLine (num);

(A)	A	(C)	C
(B)	B	(D)	D

3.4.1 Answers

1.	C
2.	A
3.	B
4.	B
5.	D
6.	C
7.	D

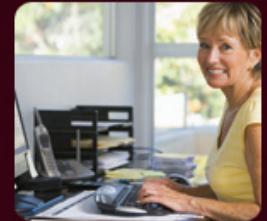


Summary

- Statements are executable lines of code that build up a program.
- Expressions are a part of statements that always result in generating a value as the output.
- Operators are symbols used to perform mathematical and logical calculations.
- Each operator in C# is associated with a priority level in comparison with other operators.
- You can convert a data type into another data type implicitly or explicitly in C#.
- A value type can be converted to a reference type using the boxing technique.
- A reference type can be converted to a value type using the unboxing technique.

GROWTH
RESEARCH
OBSERVATION
UPDATES
PARTICIPATION





Session - 4

C# Programming Constructs

Welcome to the Session, **C# Programming Constructs**.

Constructs help in building the flow of a program. Conditional and loop constructs are used for performing specific actions depending on whether a certain condition is satisfied or not. In C#, using constructs, you can control the flow of a program.

In this session, you will learn to:

- ➔ Explain selection constructs
- ➔ Describe loop constructs
- ➔ Explain jump statements in C#

4.1 Selection Constructs

A selection construct is a programming construct supported by C# that controls the flow of a program. It executes a particular block of statements based on a boolean condition, which is an expression returning true or false. The selection constructs are referred to as decision-making constructs. Therefore, selection constructs allow you to take logical decisions about executing different blocks of a program to achieve the required logical output. C# supports the following decision-making constructs:

- if... else
- if...else...if
- Nested if
- switch...case

4.1.1 The *if* Statement

The *if* statement allows you to execute a block of statements after evaluating the specified logical condition. The *if* statement starts with the *if* keyword and is followed by the condition. If the condition evaluates to true, the block of statements following the *if* statement is executed. If the condition evaluates to false, the block of statements following the *if* statement is ignored and the statement after the block is executed.

The following is the syntax for the *if* statement.

Syntax:

```
if (condition)
{
    // one or more statements;
}
```

where,

condition: Is the boolean expression.

statements: Are set of executable instructions executed when the boolean expression returns true.

Figure 4.1 displays an example of the `if` construct.

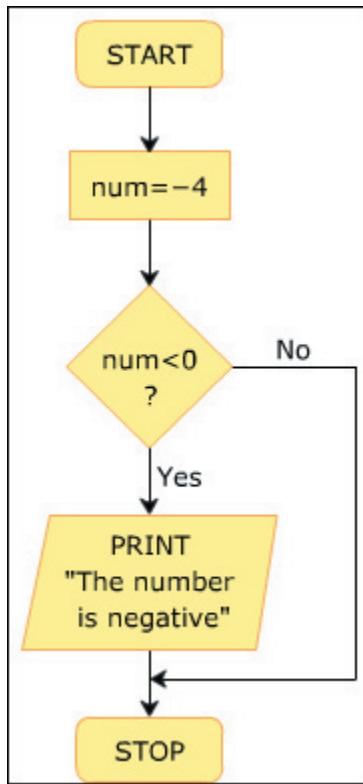


Figure 4.1: The if Construct

Code Snippet 1 displays whether the number is negative using the `if` statement.

Code Snippet 1:

```

int num = -4;
if (num < 0)
{
    Console.WriteLine("The number is negative");
}
  
```

In Code Snippet 1, `num` is declared as an integer variable and is initialized to value -4. The `if` statement is executed and the value of `num` is checked to see if it is less than 0. The condition evaluates to true and the output "The number is negative" is displayed in the console window.

Output:

The number is negative.

Note - If the `if` statement is followed by only one statement, it is not required to include the statement in curly braces.

4.1.2 The if...else Construct

The `if` statement executes a block of statements only if the specified condition is true. However, in some situations, it is required to define an action for a false condition. This is done using an `if...else` construct.

The `if...else` construct starts with the `if` block followed by an `else` block. The `else` block starts with the `else` keyword followed by a block of statements. If the condition specified in the `if` statement evaluates to false, the statements in the `else` block are executed.

The following is the syntax for the `if...else` statement.

Syntax:

```
if (condition)
{
    // one or more statements;
}

else
{
    //one or more statements;
}
```

Figure 4.2 demonstrates the `if-else` construct with an example.

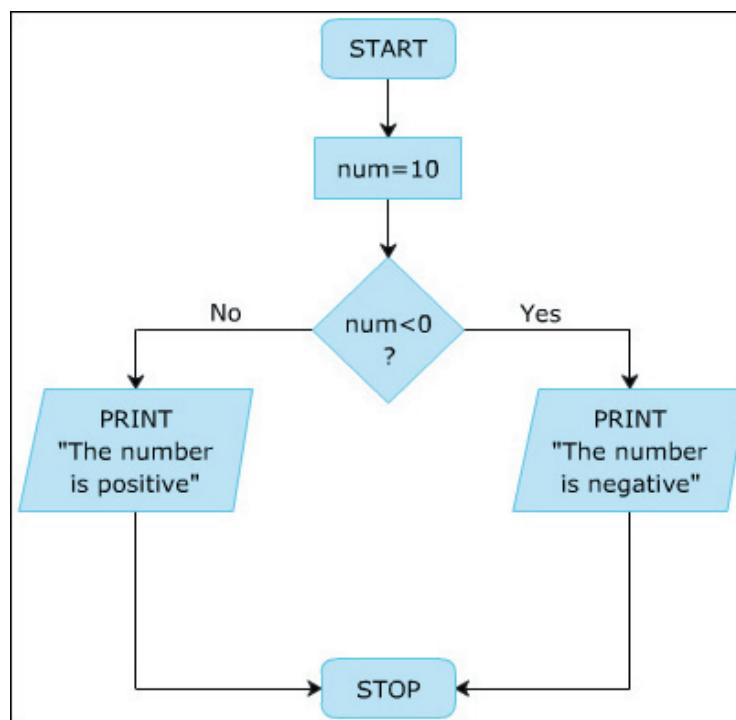


Figure 4.2: The if-else Construct

Code Snippet 2 displays whether a number is positive or negative using the `if...else` construct.

Code Snippet 2:

```
int num=10;
if (num< 0)
{
    Console.WriteLine("The number is negative");
}
else
{
    Console.WriteLine("The number is positive");
}
```

In Code Snippet 2, `num` is declared as an integer variable and is initialized to value 10. The `if` statement is executed and the value of `num` is checked to see if it is less than 0. The condition evaluates to `false` and the program control is passed to the `else` block and the output “The number is positive” is displayed in the console window.

Output:

The number is positive.

4.1.3 The `if...else if` Construct

The `if...else if` construct allows you to check multiple conditions and it executes a different block of code for each condition. This construct is also referred to as `if-else-if` ladder. The construct starts with the `if` statement followed by multiple `else if` statements followed by an optional `else` block.

The conditions specified in the `if..else if` construct are evaluated sequentially. The execution starts from the `if` statement. If a condition evaluates to `false`, the condition specified in the following `else...if` statement is evaluated.

The following is the syntax for the `if..else if` construct.

Syntax:

```
if (condition)
{
    // one or more statements;
}
else if (condition)
{
```

```
// one or more statements;  
}  
  
else  
{  
// one or more statements;  
}
```

Figure 4.3 displays an example of the if-else-if construct.

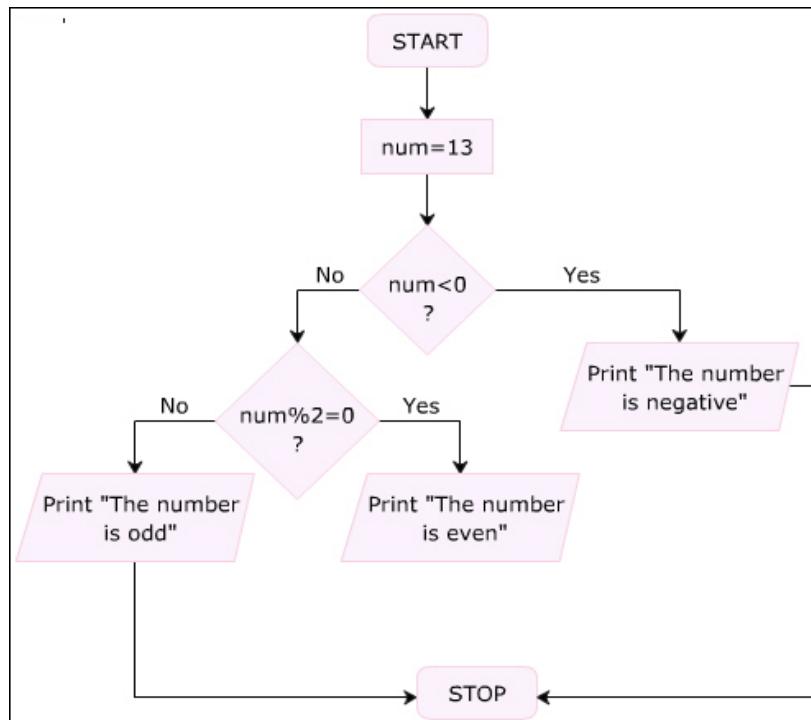


Figure 4.3: The if-else-if Construct

Code Snippet 3 displays whether a number is negative, even or odd using the `if...else...if` construct.

Code Snippet 3:

```
int num=13;
if (num< 0)
{
    Console.WriteLine("The number is negative");
}
else if ((num % 2) == 0)
{
    Console.WriteLine("The number is even");
}
```

In Code Snippet 3, `num` is declared as an integer variable and is initialized to value 13. The `if` statement is executed and the value of `num` is checked to see if it is less than 0. The condition evaluates to `false` and the program control is passed to the `else if` block. The value of `num` is divided by 2 and the remainder is checked to see if it is 0. This condition evaluates to `false` and the control passes to the `else` block. Finally, the output “The number is odd” is displayed in the console window.

Output:

The number is odd.

4.1.4 Nested `if` Construct

The nested `if` construct consists of multiple `if` statements. The nested `if` construct starts with the `if` statement, which is called the outer `if` statement, and contains multiple `if` statements, which are called inner `if` statements.

In the nested `if` construct, the outer `if` condition controls the execution of the inner `if` statements. The compiler executes the inner `if` statements only if the condition in the outer `if` statement is true. In addition, each inner `if` statement is executed only if the condition in its previous inner `if` statement is true.

The following is the syntax for the nested `if` construct.

Syntax:

```
if (condition)
{
    // one or more statements;
    if (condition)
```

```

{
    // one or more statements;

    if (condition)
    {
        // one or more statements;
    }
}

```

Figure 4.4 displays the nested `if` construct.

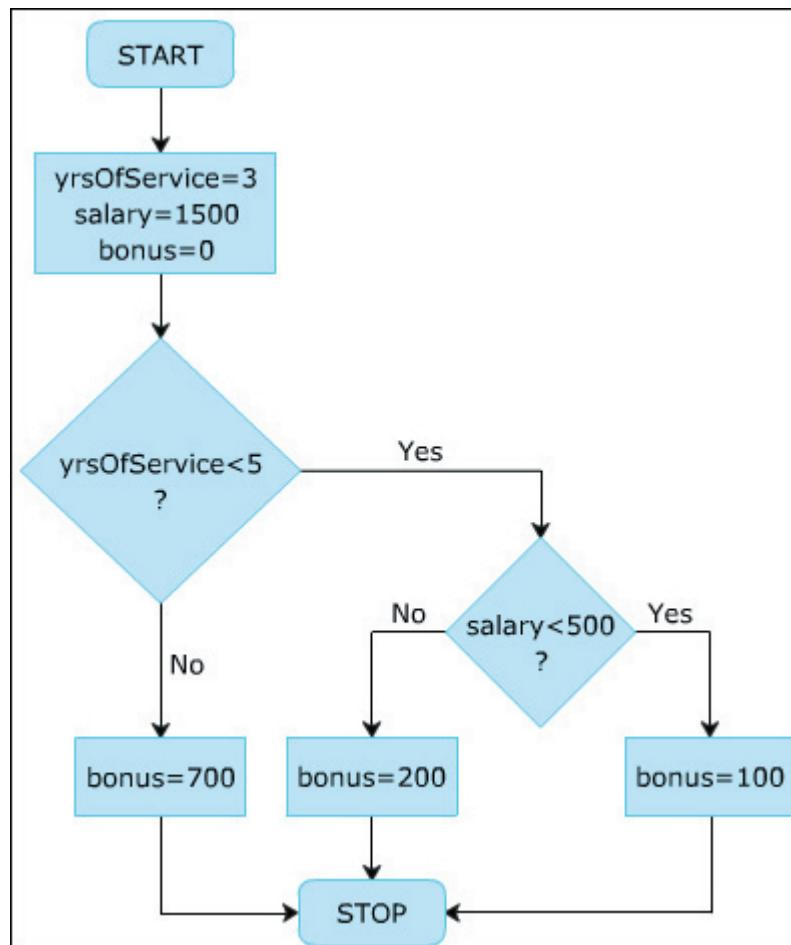


Figure 4.4: Nested if Construct

Code Snippet 4 displays the bonus amount using the nested `if` construct.

Code Snippet 4:

```
int yrsOfService = 3;
double salary = 1500;
int bonus = 0;
if (yrsOfService < 5)
{
    if (salary < 500)
    {
        bonus = 100;
    }
    else
    {
        bonus = 200;
    }
}
else
{
    bonus = 700;
}
Console.WriteLine("Bonus amount: " + bonus);
```

In Code Snippet 4, `yrsOfService` and `bonus` are declared as integer variables and initialized to values 3 and 0 respectively. In addition, `salary` is declared as a double and is initialized to value 1500. The first `if` statement is executed and the value of `yrsOfService` is checked to see if it is less than 5. This condition is found to be true. Next, the value of `salary` is checked to see if it is less than 500. This condition is found to be false. Hence, the control passes to the `else` block of the inner `if` statement. Finally, the bonus amount is displayed as 200.

200.

Output:

Bonus amount: 200

4.1.5 `switch...case` Construct

A program is difficult to comprehend when there are too many `if` statements representing multiple

selection constructs. To avoid using multiple `if` statements, in certain cases the `switch...case` approach can be used as an alternative.

The `switch...case` statement is used when a variable needs to be compared against different values.

→ **`switch`**

The `switch` keyword is followed by an integer expression enclosed in parentheses. The expression must be of type `int`, `char`, `byte`, or `short`. The `switch` statement executes the `case` corresponding to the value of the expression.

→ **`case`**

The `case` keyword is followed by a unique integer constant and a colon. Thus, the `case` statement cannot contain a variable. The block following a particular `case` statement is executed when the `switch` expression and the `case` value match. Each `case` block must end with the `break` keyword that passes the control out of the `switch` construct.

→ **`default`**

If no `case` value matches the `switch` expression value, the program control is transferred to the `default` block. This is the equivalent of the `else` block of the `if...else...if` construct.

→ **`break`**

The `break` statement is optional and is used inside the `switch...case` statement to terminate the execution of the statement sequence. The control is transferred to the statement after the end of `switch`. If there is no `break`, execution flows sequentially into the next `case` statement. Sometimes, multiple cases can be present without `break` statements between them.

Code Snippet 5 displays the day of the week using the `switch...case` construct.

Code Snippet 5:

```
int day = 5;

switch (day)
{
    case 1:
        Console.WriteLine("Sunday");
        break;
    case 2:
        Console.WriteLine("Monday");
        break;
```

```

case 3:
    Console.WriteLine("Tuesday");
    break;

case 4:
    Console.WriteLine("Wednesday");
    break;

case 5:
    Console.WriteLine("Thursday");
    break;

case 6:
    Console.WriteLine("Friday");
    break;

case 7:
    Console.WriteLine("Saturday");
    break;

default:
    Console.WriteLine("Enter a number between 1 to 7");
    break;
}

```

In Code Snippet 5, **day** is declared as an integer variable and is initialized to value 5. The block of code following the **case 5** statement is executed because the value of **day** is 5 and the day is displayed as Thursday. When the **break** statement is encountered, the control passes out of the **switch...case** construct.

Output:

Thursday

4.1.6 Nested switch...case Construct

C# allows the **switch...case** construct to be nested. That is, a **case** block of a **switch...case** construct can contain another **switch...case** construct. Also, the **case constants** of the **inner** **switch...case** construct can have values that are identical to the **case constants** of the **outer** construct.

Code Snippet 6 demonstrates the use of nested switch.

Code Snippet 6:

```
using System;
class Math
{
    static void Main(string[] args)
    {
        int numOne;
        int numTwo;
        int result = 0;
        Console.WriteLine("(1) Addition");
        Console.WriteLine("(2) Subtraction");
        Console.WriteLine("(3) Multiplication");
        Console.WriteLine("(4) Division");
        int input = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter value one");
        numOne = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter value two");
        numTwo = Convert.ToInt32(Console.ReadLine());
        switch (input)
        {
            case 1:
                result = numOne + numTwo;
                break;
            case 2:
                result = numOne - numTwo;
                break;
            case 3:
                result = numOne * numTwo;
                break;
```

```

case 4:

    Console.WriteLine("Do you want to calculate
        the quotient or remainder?");
    Console.WriteLine("(1) Quotient");
    Console.WriteLine("(2) Remainder");
    int choice = Convert.ToInt32(Console.ReadLine());
    switch (choice)

    {

case 1:

        result = numOne / numTwo;
        break;

case 2:

        result = numOne % numTwo;
        break;

default:

        Console.WriteLine("Incorrect Choice");
        break;
    }

break;

default:

        Console.WriteLine("Incorrect Choice");
        break;
    }

Console.WriteLine("Result: " + result);
}
}

```

In Code Snippet 6, the user is asked to choose the desired arithmetic operation. The user then enters the numbers on which the operation is to be performed. Using the switch...case construct, based on the input from the user, the appropriate operation is performed.

The `case` block for the division option uses an inner `switch...case` construct to either calculate the quotient or the remainder of the division as per the choice selected by the user.

Figure 4.5 displays the output of nested switch.

```
C:\WINDOWS\system32\cmd.exe
<1> Addition
<2> Subtraction
<3> Multiplication
<4> Division
4
Enter value one
22
Enter value two
5
Do you want to calculate the quotient or remainder?
<1> Quotient
<2> Remainder
1
Result: 4
Press any key to continue . . .
```

Figure 4.5: Output of Using Nested switch

4.1.7 No-Fall-Through Rule

Languages like C, C++, and Java allow statement execution associated with one `case` to continue into the next `case`. This continuation of execution into the next `case` is referred to as falling through. However, in C#, the flow of execution from one `case` statement is not allowed to continue to the next `case` statement. This is referred to as the ‘no-fall-through’ rule of C#. Thus, the list of statements inside a `case` block generally ends with a `break` or a `goto` statement, which causes the control of the program to exit the `switch...case` construct and go to the statement following the construct. The last `case` block (or the `default` block) also needs to have a statement like `break` or `goto` to explicitly take the control outside the `switch...case` construct.

C# introduced the no fall-through rule to allow the compiler to rearrange the order of the case blocks for performance optimization. Also, this rule prevents accidental continuation of statements from one case into another due to error by the programmer.

Although C# does not permit the statement sequence of one case block to fall through to the next, it does allow empty case blocks (case blocks without any statements) to fall through. As a result, multiple case statements can be made to execute the same code sequence, as shown in Code Snippet 7.

Code Snippet 7:

```
using System;
class Months
{
    static void Main(string[] args)
```

```
{  
    string input;  
  
    Console.WriteLine("Enter the month");  
  
    input = Console.ReadLine().ToUpper();  
  
    switch (input)  
    {  
  
        case "JANUARY":  
  
        case "MARCH":  
  
        case "MAY":  
  
        case "JULY":  
  
        case "AUGUST":  
  
        case "OCTOBER":  
  
        case "DECEMBER":  
  
            Console.WriteLine ("This month has 31 days");  
  
            break;  
  
        case "APRIL":  
  
        case "JUNE":  
  
        case "SEPTEMBER":  
  
        case "NOVEMBER":  
  
            Console.WriteLine ("This month has 30 days");  
  
            break;  
  
        case "FEBRUARY":  
  
            Console.WriteLine ("This month has 28 days in  
a non-leap year and 29 days in a leap year");  
  
            break;  
  
        default:  
  
            Console.WriteLine ("Incorrect choice");  
            break;  
    }  
}  
}  
}
```

In Code Snippet 7, the `switch...case` construct is used to display the number of days in a particular month. Here, all months having 31 days have been stacked together to execute the same statement.

Similarly, all months having 30 days have been stacked together to execute the same statement. Here, the no-fall-through rule is not violated as the stacked `case` statements execute the same code. By stacking the `case` statements, unnecessary repetition of code is avoided.

Figure 4.6 shows the output of multiple `case` statements.

```
C:\WINDOWS\system32\cmd.exe
Enter the month
February
This month has 28 days in a non-leap year and 29 days in a leap year
Press any key to continue . . . -
```

Figure 4.6: Multiple case Statements

4.2 Loop Constructs

Loops allow you to execute a single statement or a block of statements repetitively. The most common uses of loops include displaying a series of numbers and taking repetitive input. In software programming, a loop construct contains a condition that helps the compiler identify the number of times a specific block will be executed. If the condition is not specified, the loop continues infinitely and is termed as an infinite loop. The loop constructs are also referred to as iteration statements.

C# supports four types of loop constructs. These are as follows:

- The `while` loop
- The `do...while` loop
- The `for` loop
- The `foreach` loop

4.2.1 The `while` Loop

The `while` loop is used to execute a block of code repetitively as long as the condition of the loop remains true. The `while` loop consists of the `while` statement, which begins with the `while` keyword followed by a boolean condition. If the condition evaluates to true, the block of statements after the `while` statement is executed.

After each iteration, the control is transferred back to the `while` statement and the condition is checked again for another round of execution. When the condition is evaluated to false, the block of statements following the `while` statement is ignored and the statement appearing after the block is executed by the compiler.

The following is the syntax of the while loop.

Syntax:

```
while (condition)
{
    // one or more statements;
}
```

where,

condition: Specifies the boolean expression.

Figure 4.7 displays the while loop.

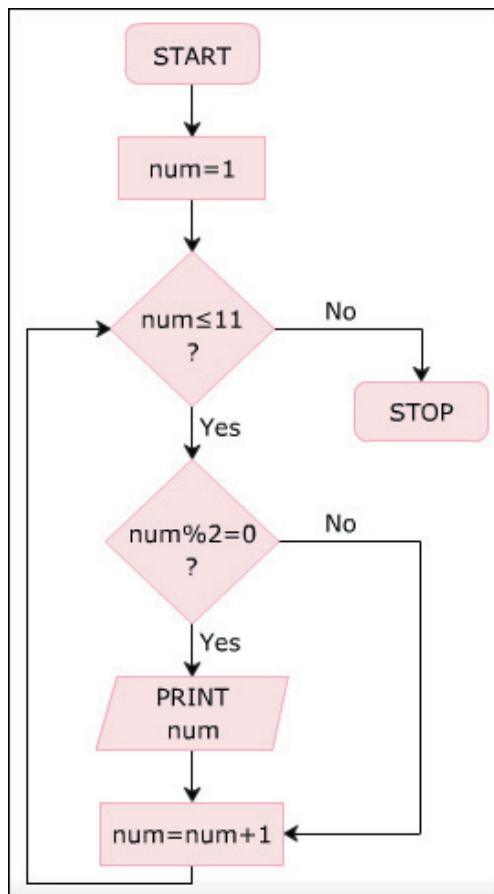


Figure 4.7: The while Loop

Code Snippet 8 displays even numbers from 1 to 10 using the `while` loop.

Code Snippet 8:

```
public int num=1;  
Console.WriteLine("Even Numbers");  
while (num<=11)  
{  
    if ((num%2) == 0)  
    {  
        Console.WriteLine(num);  
    }  
    num = num + 1;  
}
```

In Code Snippet 8, `num` is declared as an integer variable and initialized to value 1. The condition in the `while` loop is checked, which specifies that the value of `num` variable should be less than or equal to 11. If this condition is true, the value of the `num` variable is divided by 2 and the remainder is checked to see if it is 0. If the remainder is 0, the value of the variable `num` is displayed in the console window and the variable `num` is incremented by 1. Then, the program control is passed to the `while` statement to check the condition again. When the value of `num` becomes 12, the `while` loop terminates as the loop condition becomes false.

Output:

```
Even Numbers  
2  
4  
6  
8  
10
```

Note - The condition for the `while` loop is always checked before executing the loop. Therefore, the `while` loop is also referred to as the pre-test loop.

4.2.2 Nested while Loop

A `while` loop can be created within another `while` loop to create a nested `while` loop structure.

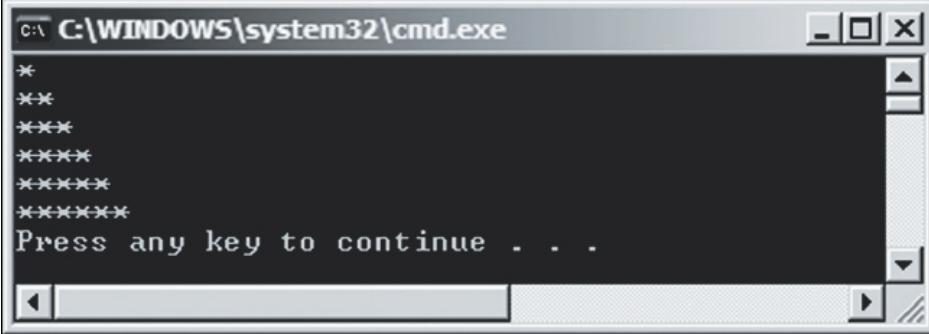
Code Snippet 9 demonstrates the use of nested `while` loops to create a geometric pattern.

Code Snippet 9:

```
using System;
class Pattern
{
    static void Main(string[] args)
    {
        int i = 0;
        int j;
        while (i <= 5)
        {
            j = 0;
            while (j <= i)
            {
                Console.Write("*");
                j++;
            }
            Console.WriteLine();
            i++;
        }
    }
}
```

In Code Snippet 9, a pattern of a right-angled triangle is created using the asterisk (*) symbol. This is done using the nested `while` loop.

Figure 4.8 shows the output of using nested `while` loop.



```
C:\WINDOWS\system32\cmd.exe
*
**
***
****
*****
******
Press any key to continue . . .
```

Figure 4.8: Output of Using Nested `while` Loop

4.2.3 The `do-while` Loop

The `do-while` loop is similar to the `while` loop; however, it is always executed at least once without the condition being checked. The loop starts with the `do` keyword and is followed by a block of executable statements. The `while` statement along with the condition appears at the end of this block.

The statements in the `do-while` loop are executed as long as the specified condition remains true. When the condition evaluates to false, the block of statements after the `do` keyword are ignored and the immediate statement after the `while` statement is executed.

The following is the syntax of the `do-while` loop.

Syntax:

```
do
{
    // one or more statements;
} while (condition);
```

Figure 4.9 displays the do-while loop.

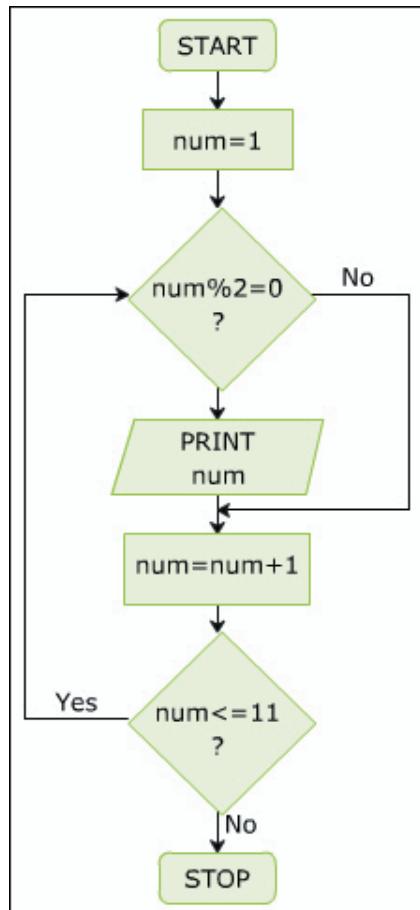


Figure 4.9: The do-while Loop

Code Snippet 10 displays even numbers from 1 to 10 using the do-while loop.

Code Snippet 10:

```

int num=1;

Console.WriteLine("Even Numbers");

do
{
    if ((num % 2) == 0)
    {
        Console.WriteLine(num);
    }
    num = num + 1;
} while (num <= 11);
  
```

In Code Snippet 10, `num` is declared as an integer variable and is initialized to value 1. In the `do` block, without checking any condition, the value of `num` is first divided by 2 and the remainder is checked to see if it is 0. If the remainder is 0, the value of `num` is displayed and it is then incremented by 1. Then, the condition in the `while` statement is checked to see if the value of `num` is less than or equal to 11. If this condition is true, the `do-while` loop executes again. When the value of `num` becomes 12, the `do-while` loop terminates.

Output:

Even Numbers

2
4
6
8
10

Note - The statements defined in the `do-while` loop are executed for the first time and then the specified condition is checked. Therefore, the `do-while` loop is referred to as the post-test loop.

4.2.4 The `for` Loop

The `for` statement is similar to the `while` statement in its function. The statements within the body of the loop are executed as long as the condition is true. Here too, the condition is checked before the statements are executed.

The following is the syntax of the `for` loop.

Syntax:

```
for (initialisation; condition; increment/decrement)
{
    // one or more statements;
}
```

where,

`initialisation`: Initialises the variable(s) that will be used in the condition.

`condition`: Comprises the condition that is tested before the statements in the loop are executed.

`increment/decrement`: Comprises the statement that changes the value of the variable(s) to ensure that the condition specified in the condition section is reached. Typically, increment and decrement operators like `++`, `--` and shortcut operators like `+=` or `-=` are used in this section. Note that there is no semicolon at the end of the increment/decrement expressions.

Figure 4.10 displays the `for` loop.

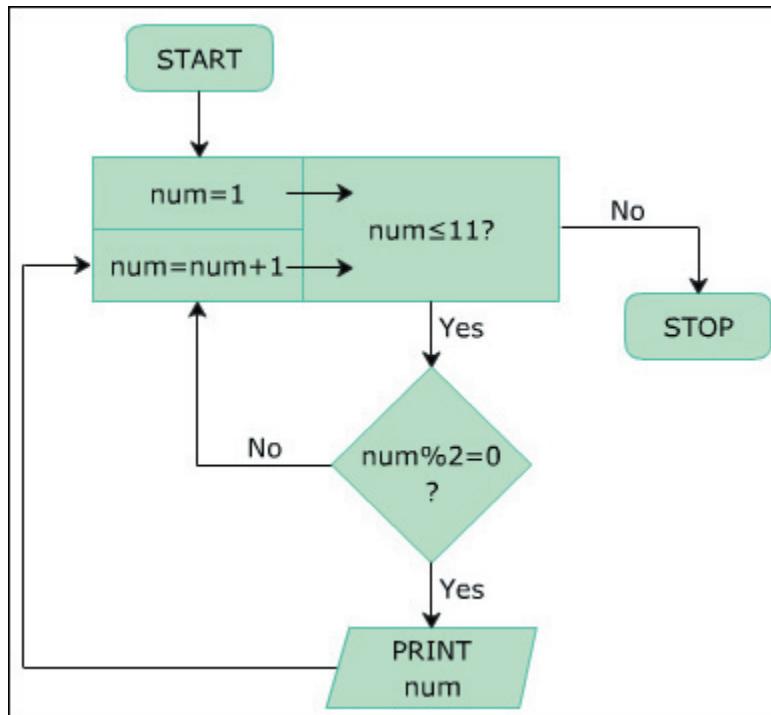


Figure 4.10: The for Loop

Code Snippet 11 displays even numbers from 1 to 10 using the `for` loop.

Code Snippet 11:

```

int num;

Console.WriteLine("Even Numbers");

for (num=1; num<=11; num++)
{
    if ((num % 2) == 0)
    {
        Console.WriteLine(num);
    }
}
  
```

In Code Snippet 11, `num` is declared as an integer variable and it is initialized to value 1 in the `for` statement. The condition specified in the `for` statement is checked for value of `num` to be less than or equal to 11. If this condition is true, value of `num` is divided by 2 and the remainder is checked to see if it is 0. If this condition is true, the control is passed to the `for` statement again. Here, the value of `num` is incremented and the condition is checked again. When the value of `num` becomes 12, the condition of

the `for` loop becomes false and the loop terminates.

Output:

Even Numbers

2
4
6
8
10

4.2.5 Nested `for` Loops

The nested `for` loop consists of multiple `for` statements. When one `for` loop is enclosed inside another `for` loop, the loops are said to be nested. The `for` loop that encloses the other `for` loop is referred to as the outer `for` loop whereas the enclosed `for` loop is referred to as the inner `for` loop.

The outer `for` loop determines the number of times the inner `for` loop will be invoked. For each iteration of the outer `for` loop, the inner `for` loop executes all its iterations.

Code Snippet 12 demonstrates a 2X2 matrix using nested `for` loops.

Code Snippet 12:

```
int rows = 2;
int columns = 2;
for (int i = 0; i < rows; i++)
{
    for (int j = 0; j < columns; j++)
    {
        Console.Write("{0} ", i * j);
    }
    Console.WriteLine();
}
```

In Code Snippet 12, `rows` and `columns` are declared as integer variables and are initialized to value 2. In addition, `i` and `j` are declared as integer variables in the outer and inner `for` loops respectively and both are initialized to 0. When the outer `for` loop is executed, the value of `i` is checked to see if it is less than 2. Here, the condition is true; hence, the inner `for` loop is executed. In this loop, the value of `j` is checked to see if it is less than 2. As long as it is less than 2, the product of the values of `i` and `j` is displayed in the console window. This inner `for` loop executes until `j` becomes greater than or equal to 2, at which time, the control passes to the outer `for` loop.

Output:

0 0

0 1

4.2.6 The for Loop with Multiple Loop Control Variables

The `for` loop allows the use of multiple variables to control the loop.

Code Snippet 13 demonstrates the use of the `for` loop with two variables.

Code Snippet 13:

```
using System;
class Numbers
{
    static void Main(string[] args)
    {
        Console.WriteLine("Square \t\tCube");
        for (int i = 1, j = 0; i < 11; i++, j++)
        {
            if ((i % 2) == 0)
            {
                Console.Write("{0} = {1} \t", i, (i * i));
                Console.Write("{0} = {1} \n", j, (j * j * j));
            }
        }
    }
}
```

In Code Snippet 13, the initialization portion as well as the increment/decrement portion of the `for` loop definition contain two variables, `i` and `j`. These variables are used in the body of the `for` loop, to display the square of all even numbers, and the cube of all odd numbers between 1 and 10.

Figure 4.11 shows the use of `for` loop.

```
C:\WINDOWS\system32\cmd.exe
Square           Cube
2   =   4           1   =   1
4   =   16          3   =   27
6   =   36          5   =   125
8   =   64          7   =   343
10  =   100         9   =   729
Press any key to continue . . .
```

Figure 4.11: Use of a for Loop

4.2.7 The `for` Loop with Missing Portions

The `for` loop definition can be divided into three portions, the initialization, the conditional expression, and the increment/decrement portion. C# allows the creation of the `for` loop even if one or more portions of the loop definition are omitted. In fact, the `for` loop can be created with all the three portions omitted.

Code Snippet 14 demonstrates a `for` loop that leaves out or omits the increment/decrement portion of the loop definition.

Code Snippet 14:

```
using System;

class Investment
{
    static void Main(string[] args)
    {
        int investment;
        int returns;
        int expenses;
        int profit;
        int counter=0;
        for (investment=1000, returns=0; returns<investment;)
        {
            Console.WriteLine("Enter the monthly expenditure");
            expenses=Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("Enter the monthly profit");
```

```

    profit = Convert.ToInt32(Console.ReadLine());
    investment += expenses;
    returns += profit;
    counter++;
}
Console.WriteLine("Number of months to break even: " + counter);
}
}

```

In Code Snippet 14, the `for` loop is used to calculate the number of months it has taken for a business venture to recover its investment and break even. The expenditure and the profit for every month is accepted from the user and stored in variables `expenses` and `profit` respectively. The total investment is calculated as the initial investment plus the monthly expenses. The total returns are calculated as the sum of the profits for each month. The number of iterations of the `for` loop is stored in the variable `counter`. The conditional expression of the loop terminates the loop once the total returns of the business becomes greater than or equal to the total investment. The code then prints the total number of months it has taken the business to break even. The number of months equals the number of iterations of the `for` loop as each iteration represents a new month.

Figure 4.12 shows the `for` loop with missing operations.

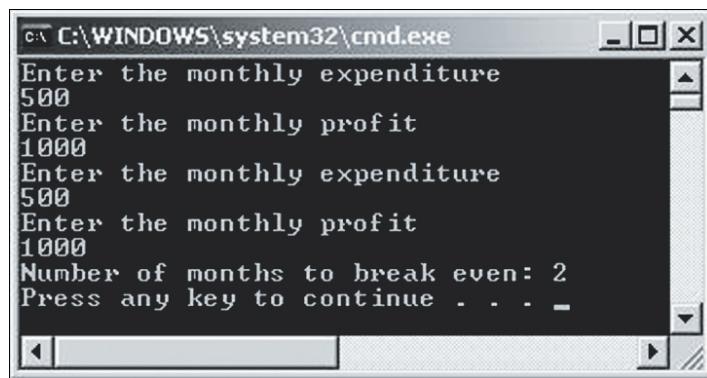


Figure 4.12: The for Loop with Missing Operations

Code Snippet 15 demonstrates a `for` loop that omits the initialization portion as well as the increment/decrement portion of the loop definition.

Code Snippet 15:

```

int investment = 1000;
int returns = 0;
for (; returns < investment; )

```

```
{  
// for loop statements  
}
```

In Code Snippet 15, the initialization of the variables **investment** and **returns** has been done prior to the **for** loop definition. Hence, the loop has been defined with the initialization portion left empty.

If the conditional expression portion of the **for** loop is omitted, the loop becomes an infinite loop. Such loops can then be terminated using jump statements such as **break** or **goto**, to exit the loop.

Code Snippet 16 demonstrates the use of an infinite **for** loop.

Code Snippet 16:

```
using System;  
  
class Summation  
{  
    static void Main(string[] args)  
    {  
        char c;  
        int numOne;  
        int numTwo;  
        int result;  
        for (; ; )  
        {  
            Console.WriteLine("Enter number one");  
            numOne = Convert.ToInt32(Console.ReadLine());  
            Console.WriteLine("Enter number two");  
            numTwo = Convert.ToInt32(Console.ReadLine());  
            result = numOne + numTwo;  
            Console.WriteLine("Result of Addition: " + result);  
            Console.WriteLine("Do you wish to continue [Y / N]");  
            c = Convert.ToChar(Console.ReadLine());  
            if (c == 'Y' || c == 'y')  
            {
```

```
        continue;  
    }  
  
    else  
  
    {  
  
        break;  
  
    }  
  
}  
  
}
```

In Code Snippet 16, an infinite `for` loop is used to repetitively accept two numbers from the user. These numbers are added and their result printed on the console. After each iteration of the loop, the `if` construct is used to check whether the user wishes to continue or not.

If the answer is Y or y (denoting yes), the loop is executed again, else the loop is exited using the break statement.

Figure 4.13 shows the use of an infinite `for` loop.

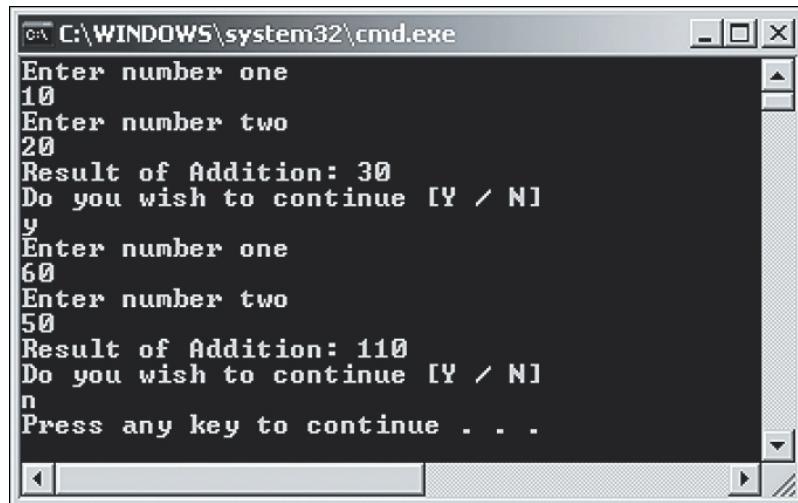


Figure 4.13: Infinite for Loop

4.2.8 The for Loop without a Body

In C#, a `for` loop can be created without a body. Such a loop is created when the operations performed within the body of the loop can be accommodated within the loop definition itself.

Code Snippet 17 demonstrates the use of a `for` loop without a body.

Code Snippet 17:

```
using System;
class Factorial
{
    static void Main(string[] args)
    {
        int fact = 1;
        int num, i;
        Console.WriteLine("Enter the number whose factorial you
wish to calculate");
        num = Convert.ToInt32(Console.ReadLine());
        for (i = 1; i <= num; fact *= i++);
        Console.WriteLine("Factorial: " + fact);
    }
}
```

In Code Snippet 17, the process of calculating the factorial of a user-given number is done entirely in the `for` loop definition; the loop has no body.

Figure 4.14 shows the output of using a `for` loop without a body.

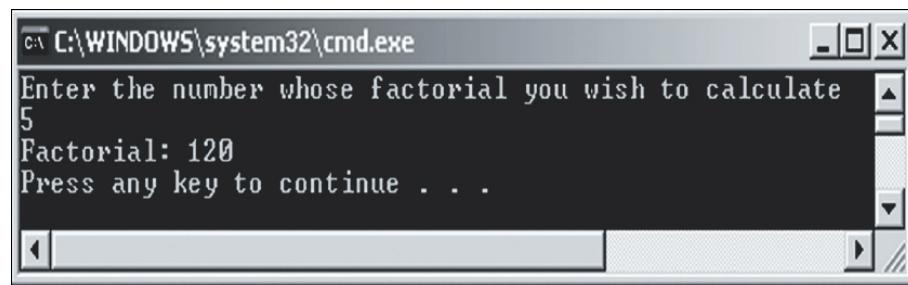


Figure 4.14: Output of Using for Loop Without Body

4.2.9 Declaring Loop Control Variables within the `for` Loop Definition

The loop control variables are often created for loops such as the `for` loop. Once the loop is terminated, there is no further use of these variables. In such cases, these variables can be created within the initialization portion of the `for` loop definition.

The following is the syntax for declaring the loop control variable within the loop definition.

Syntax:

```
for (int i = 1; i <= num; fact *= i++);
foreach (<datatype><identifier> in <list>
{
    // one or more statements;
}
```

where,

datatype: Specifies the data type of the elements in the list.

identifier: Is an appropriate name for the collection of elements.

list: Specifies the name of the list.

Code Snippet 18 displays the employee names using the `foreach` loop.

Code Snippet 18:

```
string[] employeeNames = { "Maria", "Wilson", "Elton", "Garry" };
Console.WriteLine("Employee Names");
foreach (string names in employeeNames)
{
    Console.WriteLine("{0}", names);
}
```

In Code Snippet 18, the list of employee names is declared in an array of `string` variables called `employeeNames`. In the `foreach` statement, the data type is declared as `string` and the identifier is specified as `names`. This variable refers to all values from the `employeeNames` array. The `foreach` loop displays names of the employees in the order in which they are stored.

Output:

```
Employee Names
Maria
Wilson
Elton
Garry
```

Figure 4.15 displays the `foreach` loop.

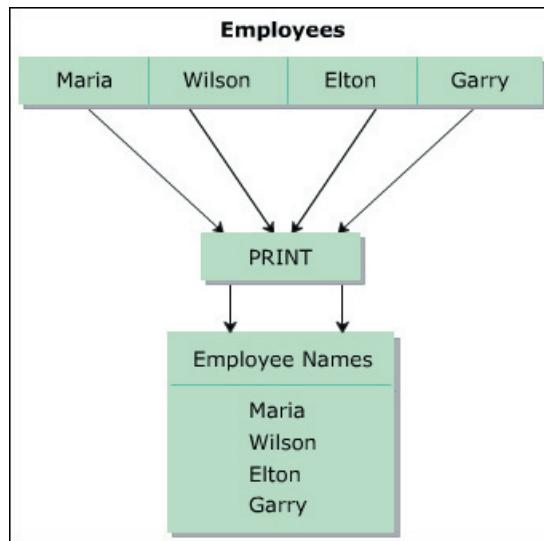


Figure 4.15: The `foreach` Loop

4.3 Jump Statements in C#

Jump statements are used to transfer control from one point in a program to another. There will be situations where you need to exit out of a loop prematurely and continue with the program.

In such cases, jump statements are used. Jump statement unconditionally transfer control of a program to a different location. The location to which a jump statement transfers control is called the target of the jump statement.

C# supports four types of `jump` statements. These are as follows:

- ➔ `break`
- ➔ `continue`
- ➔ `goto`
- ➔ `return`

4.3.1 The `break` Statement

The `break` statement is used in the selection and loop constructs. It is most widely used in the `switch...case` construct and in the `for` and `while` loops. The `break` statement is denoted by the `break` keyword. In the `switch...case` construct, it is used to terminate the execution of the construct. In loops, it is used to exit the loop without testing the loop condition. In this case, the control passes to the next statement following the loop.

Figure 4.16 displays the break statement.

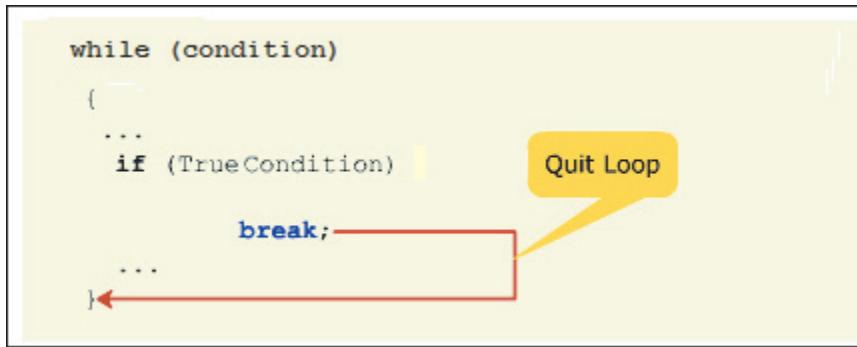


Figure 4.16: The break Statement

Code Snippet 19 displays a prime number using the `while` loop and the `break` statement.

Code Snippet 19:

```

int numOne=17;
int numTwo = 2;

while(numTwo <= numOne-1)

{
    if (numOne % numTwo == 0)

    {
        Console.WriteLine ("Not a Prime Number");
        break;
    }

    numTwo++;
}

if (numTwo == numOne)

{
    Console.WriteLine ("Prime Number");
}

```

In Code Snippet 19, `numOne` and `numTwo` are declared as integer variables and are initialized to values 17 and 2 respectively. In the `while` statement, if the condition is true, the inner `if` condition is checked. If this condition evaluates to true, the program control passes to the `if` statement outside the `while` loop. If the condition is false, the value of `numTwo` is incremented and the control passes to the `while` statement again.

Output:

Prime Number

4.3.2 The continue Statement

The `continue` statement is most widely used in the loop constructs. This statement is denoted by the `continue` keyword. The `continue` statement is used to end the current iteration of the loop and transfer the program control back to the beginning of the loop. The statements of the loop following the `continue` statement are ignored in the current iteration.

Figure 4.17 displays the `continue` statement.

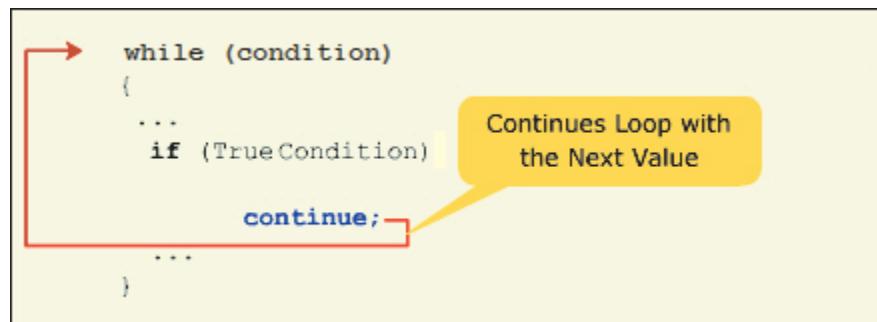


Figure 4.17: The `continue` Statement

Code Snippet 20 displays the even numbers in the range of 1 to 10 using the `for` loop and the `continue` statement.

Code Snippet 20:

```

Console.WriteLine("Even numbers in the range of 1-10");
for (int i=1; i<=10; i++)
{
    if (i % 2 != 0)
    {
        continue;
    }
    Console.Write(i + " ");
}

```

In Code Snippet 20, `i` is declared as an integer and is initialized to value 1 in the `for` loop definition. In the body of the loop, the value of `i` is divided by 2 and the remainder is checked to see if it is equal to 0. If the remainder is zero, the value of `i` is displayed as the value is an even number. If the remainder is not equal to 0, the `continue` statement is executed and the program control is transferred to the beginning of the `for` loop.

Output:

Even numbers in the range of 1-10.

2 4 6 8 10

4.3.3 The goto Statement

The `goto` statement allows you to directly execute a labeled statement or a labeled block of statements. A labeled block or a labeled statement starts with a label. A label is an identifier ending with a colon. A single labeled block can be referred by more than one `goto` statements.

The `goto` statement is denoted by the `goto` keyword. Figure 4.18 displays the `goto` statement.

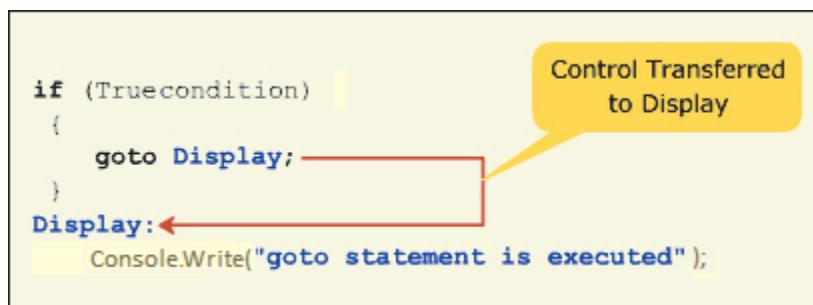


Figure 4.18: The goto Statement

Code Snippet 21 displays the output “Hello World” five times using the `goto` statement.

Code Snippet 21:

```

int i = 0;
display:
    Console.WriteLine("HelloWorld");
    i++;
    if (i < 5)
    {
        goto display;
    }
}

```

In Code Snippet 21, `i` is declared as an integer and is initialized to value 0. The program control transfers to the `display` label and the message “Hello World” is displayed.

Then, the value of `i` is incremented by 1 and is checked to see if it is less than 5. If this condition evaluates to true, the `goto` statement is executed and the program control is transferred to the `display` label. If the condition evaluates to false, the program ends.

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

Note - You cannot use the `goto` statement for moving inside a block under the `for`, `while` or `do-while` loops.

Code Snippet 22 demonstrates the use of `goto` statement to break out of a nested loop.

Code Snippet 22:

```
using System;
class Factorial
{
    static void Main(string[] args)
    {
        byte num = 0;
        while (true)
        {
            byte fact = 1;
            Console.Write("Please enter a number : ");
            num = Convert.ToByte(Console.ReadLine());
            if (num < 0)
            {
                goto stop;
            }
            for (byte j = num; j > 0; j--)
            {
                if (j > 4)
                {
                    goto stop;
                }
                fact *= j;
            }
        }
    }
}
```

```

        Console.WriteLine("Factorial of {0} is {1}", num, fact);
    }

    stop:
        Console.WriteLine("Exiting the program");
    }
}

```

In Code Snippet 22, if numbers less than or equal to 4 are entered, the program keeps displaying their factorials. However, if a number greater than 4 is entered, the loop is exited.

Figure 4.19 shows the use of `goto` statement.

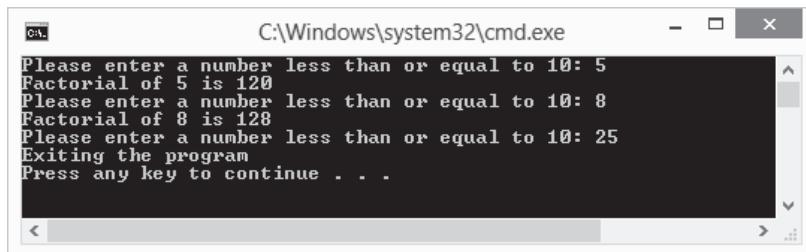


Figure 4.19: Use of `goto` Statement

4.3.4 The `return` Statement

The `return` statement is used to return a value of an expression or is used to transfer the control to the method from which the currently executing method was invoked. The `return` statement is denoted by the `return` keyword. The `return` statement must be the last statement in the method block.

Figure 4.20 displays the `return` statement.

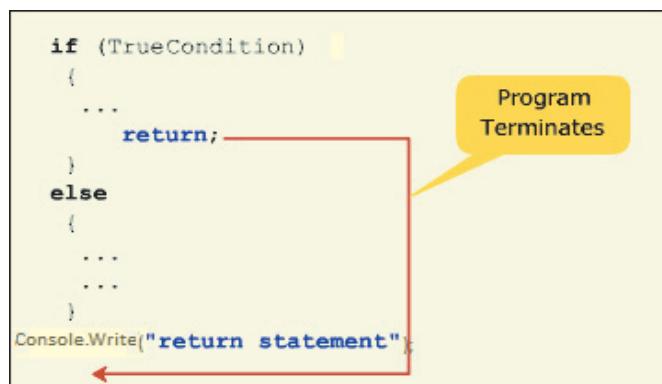


Figure 4.20: The `return` Statement

Code Snippet 23 displays the cube of a number using the `return` statement.

Code Snippet 23:

```
static void Main(string[] args)
{
    int num = 23;
    Console.WriteLine("Cube of {0} = {1}", num, Cube(num));
}

static int Cube(int n)
{
    return (n * n * n);
}
```

In Code Snippet 23, the variable `num` is declared as an integer and is initialized to value 23. The `Cube()` method is invoked by the `Console.WriteLine()` method. At this point, the program control passes to the `Cube()` method, which returns the cube of the specified value. The `return` statement returns the calculated cube value back to the `Console.WriteLine()` method, which displays the calculated cube of 23.

Output:

Cube of 23 = 12167

Code Snippet 24 demonstrates the use of the `return` statement to terminate the program.

Code Snippet 24:

```
class Factorial
{
    static void Main(string[] args)
    {
        int yrsOfService = 5;
        double salary = 1250;
        double bonus = 0;
        if (yrsOfService <= 5)
        {
            bonus = 50;
            return;
        }
        else
        {
            bonus = salary * 0.2;
        }
    }
}
```

```
    }  
    Console.WriteLine("Salary amount: " + salary);  
    Console.WriteLine("Bonus amount: " + bonus);  
}  
}
```

In Code Snippet 24, **yrsOfService** is declared as an integer variable and is initialized to value 5. In addition, **salary** and **bonus** are declared as **double** and are initialized to values 1250 and 0 respectively. The value of the **yrsOfService** variable is checked to see if it is less than or equal to 5.

This condition is true and the bonus variable is assigned the value 50. Then, the return statement is executed and the program terminates without executing the remaining statements of the program. Therefore, no output is displayed from this program.

4.4 Check Your Progress

1. Which of these statements about the selection constructs of C# are true?

(A)	A selection construct executes a block of code for the number of times specified in the condition.
(B)	The block of code following the <code>if</code> statement is skipped if the condition in the <code>if</code> statement returns false.
(C)	The <code>if..else if</code> construct checks multiple conditions and executes an appropriate block of code for each condition.
(D)	The inner <code>if</code> statements of the nested if construct control the execution of the outer <code>if</code> statement.
(E)	The <code>switch</code> statement can contain an expression of type <code>string</code> .

(A)	A, B	(C)	A, C
(B)	B, C	(D)	D

2. You are trying to display the output “311”. Which of the following codes will help you to achieve this?

(A)	<pre>public int num=3; while (num>0) { if (num>2) Console.Write(num+""); --num; if (num-- == 2) { Console.WriteLine("{0}", num); } if (--num== 1) { Console.WriteLine("{0}", num); } --num; }</pre>
-----	--

(B)

```

public int num=3;
while (num>0)
{
if (num-->2)
    Console.WriteLine(num+"");
if (num---==2)
{
    Console.Write("{0}", num);
}
if (--num==1)
{
    Console.WriteLine("{0}", ++num);
}
}

```

(C)

```

public int num=3;
while (num>0)
{
if (num>2)
{
    Console.Write(++num+"");
}
num++;
if (num---==2)
{
    Console.Write("{0}", num);
}
if (num==1)
{
    Console.WriteLine("{0}", num);
}
num--;
}

```

(D)

```
public int num=3;  
while (num>0)  
{  
    if (num>2)  
    {  
        Console.Write(num+"");  
    }  
    -- num;  
    if (num--==2)  
    {  
        Console.WriteLine("{0}", num);  
    }  
    if (num==1)  
    {  
        Console.WriteLine("{0}", num);  
    }  
    num--;  
}
```

(A)	A	(C)	C
(B)	B	(D)	D

3. You are trying to display the output "Welcome to guest". Which of the following codes will help you to achieve this?

(A)	<pre>public string login="guest"; if (login=="admin") { Console.WriteLine("Welcome to admin"); } else if(login=="guest") { Console.WriteLine("Welcome to guest"); }</pre>
(B)	<pre>public string login="guest"; if (login="admin") { Console.WriteLine("Welcome to admin"); } else if (login="guest") { Console.WriteLine("Welcome to guest"); }</pre>
(C)	<pre>public string login="guest"; if (login="admin") { Console.WriteLine("Welcome to admin"); } else if (login=="guest") { Console.WriteLine("Welcome to guest"); }</pre>

```

public string login="guest";
if (login=="admin")
Console.WriteLine("Welcome to admin");
(D) elseif (login=="guest");
{
Console.WriteLine("Welcome to guest");
}

```

(A)	A	(C)	C
(B)	B	(D)	D

4. You are trying to display the output "0 1 1 2 3 5 8 13". Which of the following codes will help you to achieve this?

```

public int firstNum=0;
public int secondNum=1;
public int result;
Console.Write("{0} ", firstNum);
Console.Write("{0} ", secondNum);
for(result=0; result<10;)
{
result=firstNum+secondNum;
Console.Write("{0} ", result);
firstNum=secondNum;
secondNum=result;
}

```

(B)

```
public int firstNum= 0;
public int secondNum= 1;
public int result;
for(result= 0; result< 10;)
{
    result= firstNum + secondNum;
    firstNum= secondNum;
    secondNum= result;
    Console.Write("{0} ", result);
}
```

(C)

```
public int firstNum= 0;
public int secondNum= 1;
public int result;
Console.Write("{0} ", firstNum);
Console.Write("{0} ", secondNum);
for(result= 0; result<= 10;)
{
    Console.Write("{0} ", result);
    result= firstNum + secondNum;
    firstNum= secondNum;
    secondNum= result;
}
```

(D)

```

public int firstNum=0
public int secondNum=1;
public int result;
Console.WriteLine("{0}", firstNum);
Console.WriteLine("{0}", secondNum);
for(result=0; result<=13;)
{
    result=firstNum+secondNum;
    Console.WriteLine("{0}", result);
    firstNum=secondNum;
    secondNum=result;
}

```

(A)	A	(C)	C
(B)	B	(D)	D

5. You want the output to be displayed as “10 9 8 7 6 5”. Can you arrange the steps in sequence to achieve the same?

(A)	while (num-- > 0);
(B)	if (num >= 5)
(C)	public int num=10;
(D)	Console.WriteLine("{0}", num);
(E)	Do

(A)	C, E, B, D,A	(C)	A, B, C, E, D
(B)	B, E, C, A, D	(D)	B, C, D, E, A

6. You are trying to display the output "Welcome to C#". Which of the following codes will help you to achieve this?

(A)	<pre>public int num=5; while (++num> 5) { if (num>= 6) { continue; Console.WriteLine("Welcome to C#"); } break; }</pre>
(B)	<pre>public int num=5; while (num++ > 5) { Console.WriteLine("Welcome to C#"); break; }</pre>
(C)	<pre>public int num=2; while (++num<= 5) { if (num== 5) Console.WriteLine("Welcome to C#"); break; }</pre>
(D)	<pre>public int num=2; while (++num<= 4) { if (num== 4) continue; Console.WriteLine("Welcome to C#"); }</pre>

(A)	A	(C)	C
(B)	B	(D)	D

7. You are trying to display the output “01 23 45”. Which of the following codes will help you to achieve this?

(A)	<pre>for (int i = 0; i <= 10 / 2; i++) { Console.Write(i + " "); if (i % 2 == 0) continue; Console.WriteLine("\t"); }</pre>
(B)	<pre>for (int i = 0; i <= 10 / 2; i++) { Console.Write(i + " "); if (i % 2 == 0) Console.WriteLine("\t"); else continue; }</pre>
(C)	<pre>for (int i = 0; i < 10 / 2; i++) { Console.Write(i + " "); if (i % 2 == 0); continue; }</pre>
(D)	<pre>for (int i = 0; i <= 10 / 2; i++) { Console.Write(i + " "); if (i % 2 == 0) continue; Console.WriteLine("\t"); }</pre>

(A)	A	(C)	C
(B)	B	(D)	D

4.4.1 Answers

1.	B
2.	D
3.	A
4.	A
5.	C
6.	D
7.	D



Summary

- Selection constructs are decision-making blocks that execute a group of statements based on the boolean value of a condition.
- C# supports if..else, if..else if, nested if, and switch...case selection constructs.
- Loop constructs execute a block of statements repeatedly for a particular condition.
- C# supports while, do-while, for, and foreach loop constructs.
- The loop control variables are often created for loops such as the for loop.
- Jump statements transfer the control to any labeled statement or block within a program.
- C# supports jump statements such as break, continue, goto, and return.



To add on to your knowledge of the subject,
visit the **REFERENCES** page





Session -5

Arrays

Welcome to the Session, **Arrays**.

An array is a set of values of a single data type. These values are accessed using a common identifier. This session provides an overview of arrays in C#, covers different types of arrays, and briefly describes the functionality of the `Array` class.

In this session, you will learn to:

- ➔ Define and describe arrays
- ➔ List and explain the types of arrays
- ➔ Explain the `Array` class

5.1 Introduction to Arrays

An array is a collection of elements of a single data type stored in adjacent memory locations. For example, in a program an array can be defined to contain 30 elements to store the scores of 30 students.

5.1.1 Purpose

Consider a program that stores the names of 100 students. To store the names, the programmer would create 100 variables of type string.

Creating and managing these 100 variables is a tedious task as it results in inefficient memory utilization. In such situations, the programmer can create an array for storing the 100 names.

An array is a collection of related values placed in contiguous memory locations and these values are referenced using a common array name. This simplifies the task of maintaining these values.

Figure 5.1 shows an example to demonstrate the purpose of array.

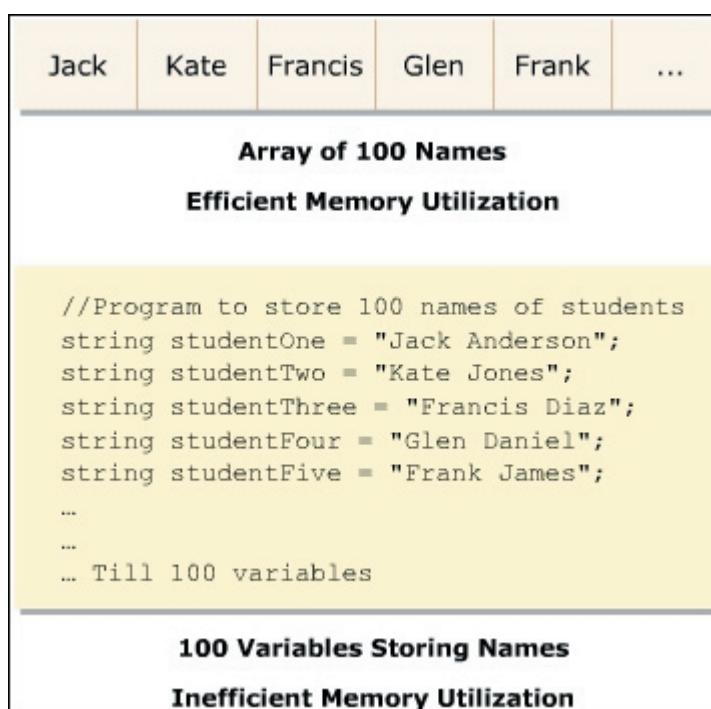


Figure 5.1: Purpose of Array

5.1.2 Definition

An array always stores values of a single data type. Each value is referred to as an element. These elements are accessed using subscripts or index numbers that determine the position of the element in the array list.

C# supports zero-based index values in an array. This means that the first array element has index number zero while the last element has index number $n-1$, where n stands for the total number of elements in the array.

This arrangement of storing values helps in efficient storage of data, easy sorting of data, and easy tracking of the data length.

Figure 5.2 displays an example of the subscripts and elements in an array.

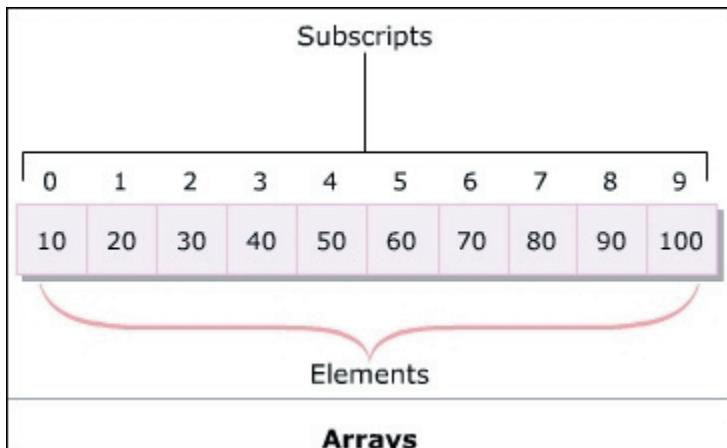


Figure 5.2: Subscripts and Elements in an Array

5.1.3 Declaring Arrays

Arrays are reference type variables whose creation involves two steps: declaration and memory allocation.

An array declaration specifies the type of data that it can hold as well as an identifier. This identifier is basically an array name and is used with a subscript to retrieve or set the data value at that location. Declaring an array does not allocate memory to the array.

The following is the syntax for declaring an array.

Syntax:

```
type[] arrayName;
```

where,

type: Specifies the data type of the array elements (for example, `int` and `char`).

arrayName: Specifies the name of the array.

5.1.4 Initializing Arrays

An array can be created using the `new` keyword and then initialized. Alternatively, an array can be initialized at the time of declaration itself, in which case the `new` keyword is not used. Creating and initializing an array with the `new` keyword involves specifying the size of an array. The number of elements stored in an array depends upon the specified size. The `new` keyword allocates memory to the array and values can then be assigned to the array.

If the elements are not explicitly assigned, default values are stored in the array.

Table 5.1 lists the default values for some of the widely used data types.

Data Types	Default Values
int	0
float	0.0
double	0.0
char	'\0'
string	Null

Table 5.1: Default Values

The following syntax is used to create an array.

Syntax:

```
arrayName = new type[size-value];
```

The following syntax is used to declare and create an array in the same statement using the `new` keyword.

Syntax:

```
type[] arrayName = new type[size-value];
```

where,

`size-value`: Specifies the number of elements in the array. You can specify a variable of type `int` that stores the size of the array instead of directly specifying a value.

Once an array has been created using the syntax, its elements can be assigned values using either a subscript or using an iteration construct such as a `for` loop.

The following syntax is used to create and initialize an array without using the `new` keyword.

Syntax:

```
type[ ] arrayIdentifier = {val1, val2, val3, ..., valN};
```

where,

`val1`: is the value of the first element.

`valN`: is the value of the nth element.

Code Snippet 1 creates an integer array which can have a maximum of five elements in it.

Code Snippet 1:

```
public int[] number = new int[5];
```

Code Snippet 2 initializes an array of type `string` that assigns names at appropriate index locations.

Code Snippet 2:

```
public string[] studNames = new string{"Allan", "Wilson", "James", "Arnold"};
```

In Code Snippet 2, the string “Allan” is stored at subscript 0, “Wilson” at subscript 1, “James” at subscript 2 and “Arnold” at subscript 3.

Code Snippet 3 stores the string “Jack” as the name of the fifth enrolled student.

Code Snippet 3:

```
studNames[4] = "Jack";
```

Code Snippet 4 demonstrates another approach for creating and initializing an array. An array called `count` is created and is assigned `int` values.

Code Snippet 4:

```
using System;
class Numbers
{
    static void Main(string[] args)
    {
        int[] count = new int[10]; //array is created
        int counter = 0;
        for (int i = 0; i < 10; i++)
        {
            count[i] = counter++; //values are assigned to the elements
            Console.WriteLine("The count value is: " + count[i]);
            //element values are printed
        }
    }
}
```

In Code Snippet 4, the class `Numbers` declares an array variable `count` of size 10. An `int` variable `counter` is declared and is assigned the value 0. Using the `for` loop, every element of the array `count` is assigned the incremented value of the variable `counter`.

Output:

The count value is: 0

The count value is: 1

The count value is: 2

The count value is: 3

The count value is: 4

The count value is: 5

The count value is: 6

The count value is: 7

The count value is: 8

The count value is: 9

5.2 Types of Arrays

Based on how arrays store elements, arrays can be categorized into single-dimension and multi-dimension arrays.

5.2.1 Single-dimensional Arrays

The elements of a single-dimensional array are stored in a single row in the allocated memory. The declaration and initialization of single-dimensional arrays are the same as the standard declaration and initialization of arrays.

In a single-dimensional array, the elements are indexed from 0 to (n-1), where n is the total number of elements in the array. For example, an array of 5 elements will have the elements indexed from 0 to 4 such that the first element is indexed 0 and the last element is indexed 4.

The following syntax is used for declaring and initializing a single-dimensional array.

Syntax:

```
type[] arrayName; //declaration  
arrayName = new type[length]; // creation
```

where,

type: Is a variable type and is followed by square brackets ([]).

arrayName: Is the name of the variable.

length: Specifies the number of elements to be declared in the array.

new: Instantiates the array.

Code Snippet 5 initializes a single-dimensional array to store the name of students.

Code Snippet 5:

```
using System;
class SingleDimensionArray
{
    static void Main(string[] args)
    {
        string[] students = new string[3] {"James", "Alex", "Fernando"};
        for (int i=0; i<students.Length; i++)
        {
            Console.WriteLine(students[i]);
        }
    }
}
```

In Code Snippet 5, the class **SingleDimensionArray** stores the names of the students in the **students** array.

An integer variable **i** is declared in the **for** loop that indicates the total number of students to be displayed. Using the **for** loop, the names of the students are displayed as the output.

Output:

James

Alex

Fernando

5.2.2 Multi-dimensional Arrays

Consider a scenario where you need to store the roll numbers of 50 students and their marks in three exams. Using a single-dimensional array, you require two separate arrays for storing roll numbers and marks respectively. However, using a multi-dimensional array, you just need one array to store both roll numbers as well as marks.

A multi-dimensional array allows you to store combination of values of a single type in two or more dimensions. The dimensions of the array are represented as rows and columns similar to the rows and columns of a Microsoft Excel sheet.

There are two types of multi-dimensional arrays. These are as follows:

→ **Rectangular Array**

A rectangular array is a multi-dimensional array where all the specified dimensions have constant

values. A rectangular array will always have the same number of columns for each row.

→ Jagged Array

A jagged array is a multi-dimensional array where one of the specified dimensions can have varying sizes. Jagged arrays can have unequal number of columns for each row.

The following is the syntax for creating a rectangular array.

Syntax:

```
type [,] <arrayName>; //declaration  
arrayName = new type [value1, value2]; //initialization
```

where,

type: Is the data type and is followed by [].

arrayName: Is the name of the array.

value1: Specifies the number of rows.

value2: Specifies the number of columns.

Code Snippet 6 demonstrates the use of rectangular arrays.

Code Snippet 6:

```
using System;  
  
class RectangularArray  
{  
    static void Main (string [] args)  
    {  
        int [,] dimension = new int [4, 5];  
        int numOne = 0;  
        for (int i=0; i<4; i++)  
        {  
            for (int j=0; j<5; j++)  
            {  
                dimension [i, j] = numOne;  
                numOne++;  
            }  
        }  
    }  
}
```

```

for (int i=0; i<4; i++)
{
    for (int j=0; j<5; j++)
    {
        Console.Write(dimension [i, j] + " ");
    }
    Console.WriteLine();
}
}
}

```

In Code Snippet 6, a rectangular array called `dimension` is created that will have four rows and five columns. The `int` variable `numOne` is initialized to zero.

The code uses nested `for` loops to store each incremented value of `numOne` in the `dimension` array. These values are then displayed in the matrix format using again the nested `for` loops.

Output:

```

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

```

Note - A multi-dimensional array can have a maximum of eight dimensions.

5.2.3 Fixed and Dynamic Arrays

Arrays can be either fixed-length arrays or dynamic arrays. In a fixed-length array, the number of elements is defined at the time of declaration. For example, an array declared for storing days of the week will have exactly seven elements. Here, the number of elements is known and hence, can be defined at the time of declaration. Therefore, a fixed-length array can be used. On the other hand, in a dynamic array, the size of the array is not fixed at the time of the array declaration and can dynamically increase at run-time or whenever required. For example, an array declared to store the e-mail addresses of all users who access a particular Web site cannot have a predefined length. In such a case, the length of the array cannot be specified at the time of declaration. Here, a dynamic array has to be used. A dynamic array can add more elements to the array as and when required. Dynamic arrays are created using built-in classes of the .NET Framework.

Code Snippet 7 demonstrates the use of fixed arrays.

Code Snippet 7:

```
using System;
class DaysofWeek
{
    static void Main(string[] args)
    {
        string[] days = new string[7];
        days[0] = "Sunday";
        days[1] = "Monday";
        days[2] = "Tuesday";
        days[3] = "Wednesday";
        days[4] = "Thursday";
        days[5] = "Friday";
        days[6] = "Saturday";
        for(int i=0; i<days.Length; i++)
        {
            Console.WriteLine(days[i]);
        }
    }
}
```

In this example, a fixed-length array variable, **days**, of data type **string**, is declared to store the seven days of the week. The days from Sunday to Saturday are stored in the index positions 0 to 6 of the array and are displayed on the console using the `Console.WriteLine()` method.

Figure 5.3 displays the use of fixed arrays.

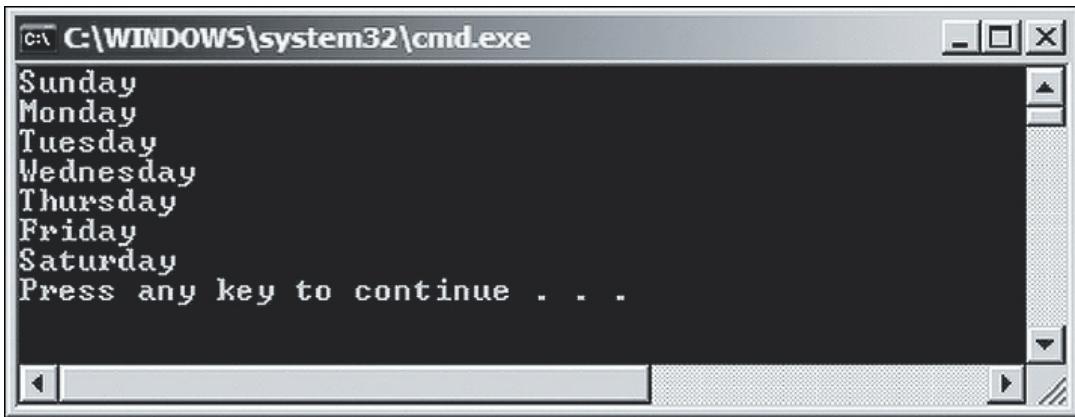


Figure 5.3: Use of Fixed Arrays

5.2.4 Array References

An array variable can be referenced by another array variable (referring variable). While referring, the referring array variable refers to the values of the referenced array variable.

Code Snippet 8 demonstrates the use of array references.

Code Snippet 8:

```
using System;

class StudentReferences
{
    public static void Main()
    {
        string[] classOne = { "Allan", "Chris", "Monica" };
        string[] classTwo = { "Katie", "Niel", "Mark" };
        Console.WriteLine("Students of Class I:\tStudents of Class II");
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(classOne[i] + "\t\t\t" + classTwo[i]);
        }
        classTwo = classOne;
        Console.WriteLine("\nStudents of Class II after referencing Class I:");
    }
}
```

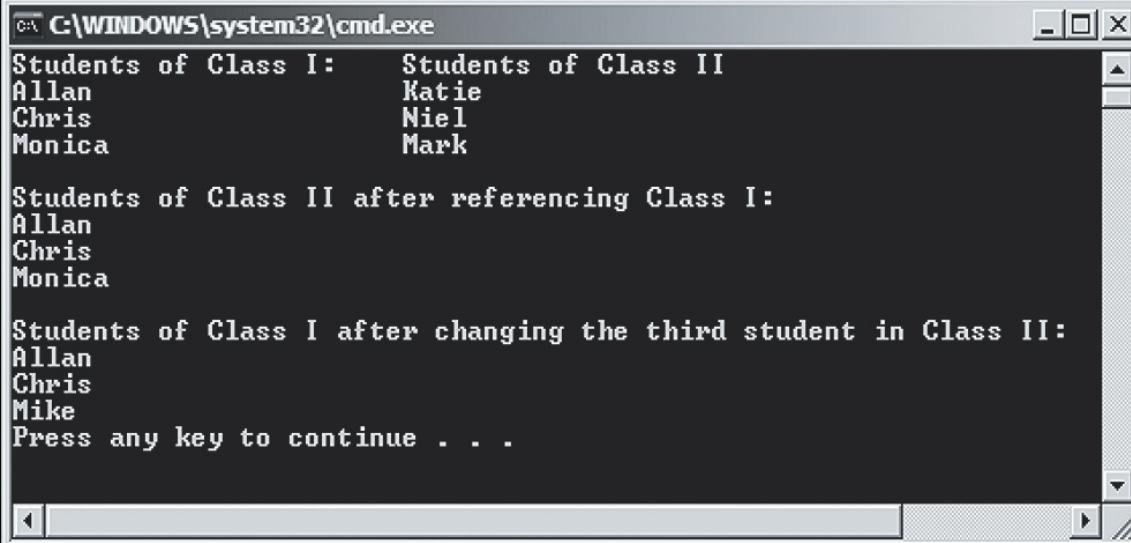
```

for (int i=0; i<3; i++)
{
    Console.WriteLine(classTwo[i] + " ");
}
Console.WriteLine();
classTwo[2] = "Mike";
Console.WriteLine("Students of Class I after changing the third
student in Class II:");
for (int i=0; i<3; i++)
{
    Console.WriteLine(classOne[i] + " ");
}
}
}

```

In Code Snippet 8, **classOne** is assigned to **classTwo**; Therefore, both the arrays reference the same set of values. Consequently, when the third array element of **classTwo** is changed from “Monica” to “Mike”, an identical change is seen in the third element of **classOne**.

Figure 5.4 displays the use of array references.



The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The output of the program is as follows:

```

Students of Class I:      Students of Class II
Allan                      Katie
Chris                      Niel
Monica                     Mark

Students of Class II after referencing Class I:
Allan
Chris
Monica

Students of Class I after changing the third student in Class II:
Allan
Chris
Mike
Press any key to continue . . .

```

Figure 5.4: Use of Array References

5.2.5 Rectangular Arrays

A rectangular array is a two-dimensional array where each row has an equal number of columns. The following syntax displays the marks stored in a rectangular array.

Syntax:

```
type [, ]<variableName>;
variableName = new type [value1 , value2];
```

where,

type: Specifies the data type of the array elements.

[,]: Specifies that the array is a two-dimensional array.

variableName: Specifies the name of the two-dimensional array.

new: Is the operator used to instantiate the array.

value1: Specifies the number of rows in the two-dimensional array.

value2: Specifies the number of columns in the two-dimensional array.

Code Snippet 9 allows the user to specify the number of students, their names, the number of exams, and the marks scored by each student in each exam. All these marks are stored in a rectangular array.

Code Snippet 9:

```
using System;
class StudentsScore
{
    void StudentDetails()
    {
        Console.Write("Enter the number of Students: ");
        int noOfStds = Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter the number of Exams: ");
        int exams = Convert.ToInt32(Console.ReadLine());
        string[] stdName = new string[noOfStds];
        string[,] details = new string[noOfStds, exams];
        for (int i = 0; i < noOfStds; i++)
        {
            Console.WriteLine();
            Console.Write("Enter the Student Name: ");
```

```
stdName[i] = Convert.ToString(Console.ReadLine());  
for (int y = 0; y < exams; y++)  
{  
    Console.Write("Enter Score in Exam " + (y + 1) + ": ");  
    details[i, y] = Convert.ToString(Console.ReadLine());  
}  
}  
Console.WriteLine();  
Console.WriteLine("Student Exam Details");  
Console.WriteLine("-----");  
Console.WriteLine();  
Console.WriteLine("Student\t\tMarks");  
Console.WriteLine("----\t\t----")  
for (int i = 0; i < stdName.Length; i++)  
{  
    Console.WriteLine(stdName[i]);  
    for (int j = 0; j < exams; j++)  
    {  
        Console.WriteLine("\t\t" + details[i, j]);  
    }  
    Console.WriteLine();  
}  
}  
static void Main()  
{  
    StudentsScore objStudentsScore = new StudentsScore();  
    objStudentsScore.StudentDetails();  
}
```

In Code Snippet 9, the **StudentsScore** class allows the user to enter the number of students in the class, the names of the students, the number of exams conducted, and the marks scored by each student in each exam.

The class declares a method **StudentDetails**, which accepts the student and the exam details. The variable **noOfStds** stores the number of students whose details are to be stored.

The variable **exams** stores the number of exams the students have appeared in. The array **stdName** stores the names of the students. The dimensions of the rectangular array **details** are defined by the variables **noOfStds** and **exams**. This array stores the marks scored by students in the various exams. A nested **for** loop is used for displaying the student details. In the **Main** method, an object is created of the class **StudentsScore** and the method **StudentDetails** is called through this object.

Figure 5.5 displays the use of a rectangular array.

```

C:\> dir
Volume in drive C has no label
Volume serial number is 0000-0000
  Directory of C:\

10/23/2023  10:23 AM    <DIR>          .
10/23/2023  10:23 AM    <DIR>          ..
10/23/2023  10:23 AM           14 array.txt
               1 File(s)      14 bytes
               2 Dir(s)   1,048,576 bytes free

C:\> type array.txt
Enter the number of Students: 2
Enter the number of Exams: 2

Enter the Student Name: Mike
Enter Score in Exam 1: 50
Enter Score in Exam 2: 60

Enter the Student Name: Susan
Enter Score in Exam 1: 70
Enter Score in Exam 2: 80

Student Exam Details
-----
Student      Marks
-----
Mike          50
              60
Susan         70
              80

Press any key to continue . . .

```

Figure 5.5: Use of Rectangular Arrays

5.2.6 Jagged Arrays

A jagged array is a multi-dimensional array and is referred to as an array of arrays. It consists of multiple arrays where the number of elements within each array can be different. Thus, rows of jagged arrays can have different number of columns.

A jagged array optimizes the memory utilization and performance because navigating and accessing elements in a jagged array is quicker as compared to other multi-dimensional arrays.

For example, consider a class of 500 students where each student has opted for a different number of subjects. Here, you can create a jagged array because the number of subjects for each student varies.

Figure 5.6 displays the representation of jagged arrays.

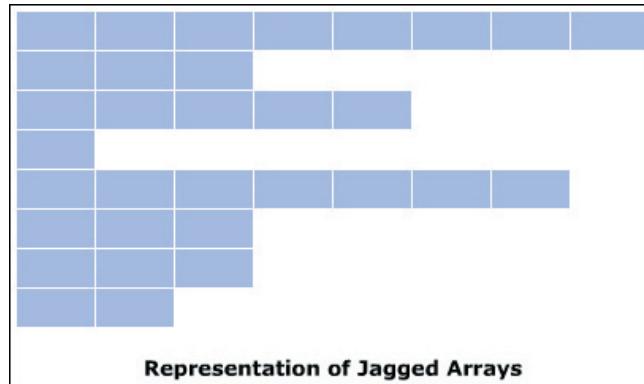


Figure 5.6: Jagged Arrays

Code Snippet 10 demonstrates the use of jagged arrays to store the names of companies.

Code Snippet 10:

```
using System;
class JaggedArray
{
    static void Main (string[] args)
    {
        string[][] companies = new string[3][];
        companies[0] = new string[] {"Intel", "AMD"};
        companies[1] = new string[] {"IBM", "Microsoft", "Sun"};
        companies[2] = new string[] {"HP", "Canon", "Lexmark", "Epson"};
        for (int i=0; i<companies.GetLength (0); i++)
        {
            Console.Write("List of companies in group " + (i+1) + ":\t");
            for (int j=0; j<companies[i].GetLength (0); j++)
            {
                Console.Write(companies [i] [j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

In Code Snippet 10, a jagged array called companies is created that has three rows. The values “Intel” and “AMD” are stored in two separate columns of the first row. Similarly, the values “IBM”, “Microsoft” and “Sun” are stored in three separate columns of the second row. Finally, the values “HP”, “Canon”, “Lexmark” and “Epson” are stored in four separate columns of the third row.

Output:

```
List of companies in group 1: Intel AMD
List of companies in group 2: IBM Microsoft Sun
List of companies in group 3: HP Canon Lexmark Epson
```

5.2.7 Using the foreach Loop for Arrays

The `foreach` loop in C# is an extension of the `for` loop. This loop is used to perform specific actions on large data collections and can even be used on arrays. The loop reads every element in the specified array and allows you to execute a block of code for each element in the array. This is particularly useful for reference types, such as strings.

The following is the syntax for the `foreach` loop.

Syntax:

```
foreach(type<identifier> in <list>
{
    // statements
}
```

where,

`type`: Is the variable type.

`identifier`: Is the variable name.

`list`: Is the array variable name.

Code Snippet 11 displays the name and the leave grant status of each student using the `foreach` loop.

Code Snippet 11:

```
using System;
class Students
{
    static void Main(string[] args)
    {
        string[] studentNames = new string[3] { "Ashley", "Joe", "Mikel" };
        foreach (string studName in studentNames)
        {
            // code
        }
    }
}
```

```
{
    Console.WriteLine("Congratulations!! " + studName + " you have been
granted an extra leave");
}
}
}
```

In Code Snippet 11, the **Students** class initializes an array variable called **studentNames**. The array variable **studentNames** stores the names of the students. In the **foreach** loop, a string variable **studName** refers to every element stored in the array variable **studentNames**. For each element stored in the **studentNames** array, the **foreach** loop displays the name of the student and grants a day's leave extra for each student.

Output:

Congratulations!! Ashley you have been granted an extra leave
 Congratulations!! Joe you have been granted an extra leave
 Congratulations!! Mikel you have been granted an extra leave

Note - The **foreach** loop allows you to navigate through an array without re-loading the array in the memory. During iteration, the elements in the list are in the read-only format. To change the values in the array, you need to use the **for** loop within the **foreach** loop. The **foreach** loop executes once for each element of the array.

5.3 Array Class

Consider a code that stores the marks of a particular subject for 100 students. The programmer wants to sort the marks, and to do this, he has to manually write the code to perform sorting. This can be tedious and result in increased lines of code. However, if the array is declared as an object of the **Array** class, the built-in methods of the **Array** class can be used to sort the array.

The **Array** class is a built-in class in the **System** namespace and is the base class for all arrays in C#.

The **Array** class provides methods for various tasks such as creating, searching, copying and sorting arrays.

Note - A class is a reference data type that is used to initialize variables and define methods. A class can be a built-in class defined in the system library of the .NET Framework or it can be user-defined.

5.3.1 Properties and Methods

The **Array** class consists of system-defined properties and methods that are used to create and manipulate arrays in C#. The properties are also referred to as system array class properties.

→ Properties

The properties of the `Array` class allow you to modify the elements declared in the array. Table 5.2 displays the properties of the `Array` class.

Properties	Descriptions
<code>IsFixedSize</code>	Returns a boolean value, which indicates whether the array has a fixed size or not. The default value is true.
<code>IsReadOnly</code>	Returns a boolean value, which indicates whether an array is read-only or not. The default value is false.
<code>IsSynchronized</code>	Returns a boolean value, which indicates whether an array can function well while being executed by multiple threads together. The default value is false.
<code>Length</code>	Returns a 32-bit integer value that denotes the total number of elements in an array.
<code>LongLength</code>	Returns a 64-bit integer value that denotes the total number of elements in an array.
<code>Rank</code>	Returns an integer value that denotes the rank, which is the number of dimensions in an array.
<code>SyncRoot</code>	Returns an object which is used to synchronize access to the array.

Table 5.2: Properties of Array Class

→ Methods

The `Array` class allows you to clear, copy, search, and sort the elements declared in the array.

Table 5.3 displays the most commonly used methods in the `Array` class.

Methods	Descriptions
<code>Clear</code>	Deletes all elements within the array and sets the size of the array to 0.
<code>CopyTo</code>	Copies all elements of the current single-dimensional array to another single-dimensional array starting from the specified index position.
<code>GetLength</code>	Returns number of elements in an array.
<code>GetLowerBound</code>	Returns the lower bound of an array.
<code>GetUpperBound</code>	Returns the upper bound of an array.
<code>Initialize</code>	Initializes each element of the array by calling the default constructor of the <code>Array</code> class.
<code>Sort</code>	Sorts the elements in the single-dimensional array.
<code>SetValue</code>	Sets the specified value at the specified index position in the array.

Methods	Descriptions
GetValue	Gets the specified value from the specified index position in the array.

Table 5.3: Methods in Array Class

5.3.2 Using the Array Class

The `Array` class allows you to create arrays using the `CreateInstance()` method. This method can be used with different parameters to create single-dimensional and multi-dimensional arrays. For creating an array using this class, you need to invoke the `CreateInstance()` method that is accessed by specifying the class name because the method is declared as static.

The following is the syntax for signature of the `CreateInstance()` method used for creating a single-dimensional array.

Syntax:

```
public static Array CreateInstance(Type elementType, int length)
```

where,

`Array`: Returns a reference to the created array.

`Type`: Uses the `typeof` operator for explicit casting.

`elementType`: Is the resultant data type in casting.

`Length`: Specifies the length of the array.

The following is the syntax for signature of the `CreateInstance()` method used for creating a multi-dimensional array.

Syntax:

```
public static Array CreateInstance(Type elementType, int length1, int length2)
```

where,

`length1`: Specifies the row length.

`length2`: Specifies the column length.

These syntax determine how the method is declared in the `Array` class. To create single-dimensional and multi-dimensional arrays, you must explicitly invoke the method with the appropriate parameters.

Code Snippet 12 creates an array of length 5 using the `Array` class and stores the different subject names.

Code Snippet 12:

```
using System;
class Subjects
{
    static void Main(string [] args)
    {
        Array objArray=Array.CreateInstance(typeof (string) , 5);
        objArray.SetValue("Marketing", 0);
        objArray.SetValue("Finance", 1);
        objArray.SetValue("Human Resources", 2);
        objArray.SetValue("Information Technology", 3);
        objArray.SetValue("Business Administration", 4);
        for (int i = 0; i <= objArray.GetUpperBound(0); i++)
        {
            Console.WriteLine(objArray.GetValue(i));
        }
    }
}
```

In Code Snippet 12, the `Subjects` class creates an object of the `Array` class called `objArray`.

The `CreateInstance()` method creates a single-dimensional array and returns a reference of the `Array` class. Here, the parameter of the method specifies the data type of the array. The `SetValue()` method assigns the names of subjects in the `objArray`. Using the `GetValue()` method, the names of subjects are displayed in the console window.

Note - More than one `CreateInstance()` method is declared in the `Array` class, which takes in different parameters. The parameters can accept 64-bit integers to accommodate large-capacity arrays.

For manipulating an array, the `Array` class uses four interfaces. These are as follows:

- **ICloneable:** The `ICloneable` interface belongs to the `System` namespace and contains the `Clone()` method that allows you to create an exact copy of the current object of the class.
- **ICollection:** The `ICollection` interface belongs to the `System.Collections` namespace and contains properties that allow you to count the number of elements, check whether the

elements are synchronized and if they are not, then synchronize the elements in the collection.

- **IList:** The `IList` interface belongs to the `System.Collections` namespace and allows you to modify the elements defined in the array. The interface defines three properties, `IsFixedSize`, `IsReadOnly` and `Item`.
- **IEnumerable:** The `IEnumerable` interface belongs to the `System.Collections` namespace. This interface returns an enumerator that can be used with the `foreach` loop to iterate through a collection of elements such as an array.

5.3.3 Rank of an Array

`Rank` is a read-only property that specifies the number of dimensions of an array. For example, a three-dimensional array has rank three.

Code Snippet 13 demonstrates the use of the `Rank` property.

Code Snippet 13:

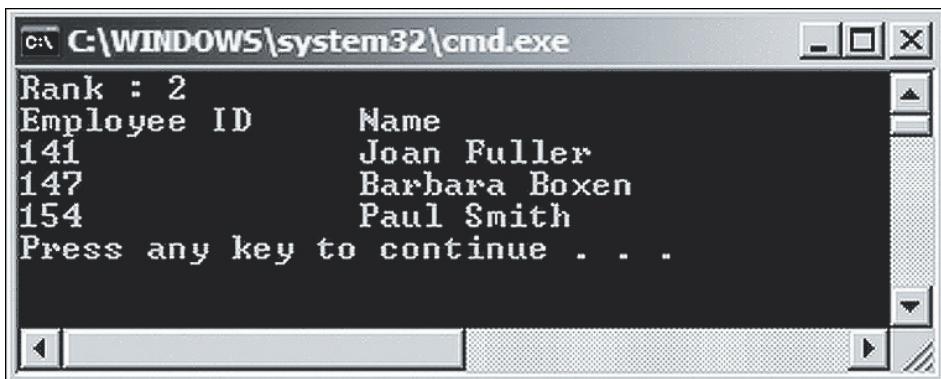
```
using System;
class Employee
{
    public static void Main()
    {
        Array objEmployeeDetails = Array.CreateInstance(typeof(string), 2, 3);
        objEmployeeDetails.SetValue("141", 0, 0);
        objEmployeeDetails.SetValue("147", 0, 1);
        objEmployeeDetails.SetValue("154", 0, 2);
        objEmployeeDetails.SetValue("Joan Fuller", 1, 0);
        objEmployeeDetails.SetValue("Barbara Boxen", 1, 1);
        objEmployeeDetails.SetValue("Paul Smith", 1, 2);
        Console.WriteLine("Rank : " + objEmployeeDetails.Rank);
        Console.WriteLine("Employee ID \tName");
        for (int i = 0; i < 1; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                Console.Write(objEmployeeDetails.GetValue(i, j) + "\t\t");
            }
            Console.WriteLine(objEmployeeDetails.GetValue(i+1, j));
        }
    }
}
```

```
    }  
}  
}  
}
```

In Code Snippet 13, the `CreateInstance()` method creates a two-dimensional array of the specified type and dimension lengths. Since this array has two dimensions, its rank will be 2.

An instance of this class `objEmployeeDetails` is created and two sets of values are then inserted in the object `objEmployeeDetails` using the method `SetValue()`. The values stored in the array are employee ID and the name of the employee. The `Rank` property retrieves the rank of the array which is displayed by the `WriteLine()` method.

Figure 5.7 displays the use of `Rank` property.



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window contains the following text:

```
Rank : 2  
Employee ID      Name  
141              Joan Fuller  
147              Barbara Boxen  
154              Paul Smith  
Press any key to continue . . .
```

Figure 5.7: Use of Rank Property

5.4 Check Your Progress

1. Which of these statements about arrays are true?

(A)	The elements of an array are typically not of the same data type.
(B)	The first element of an array is always indexed zero.
(C)	The last element of an array is always indexed at $(n - 2)$ position.
(D)	An array is used to achieve contiguous memory collection of values of a particular data type.
(E)	The size of an array is declared using an <code>int</code> variable.

(A)	A, E	(C)	C, E
(B)	B, D, E	(D)	D

2. You are trying to create an array of type string that will store seven values. Which of the following codes helps you to achieve this?

(A)	<code>public string[] cities = new string[7];</code>
(B)	<code>public string[7] cities = new[] string;</code>
(C)	<code>public string cities[] = new string[7];</code>
(D)	<code>public string cities[7] = new string{};</code>

(A)	A	(C)	C
(B)	B	(D)	D

3. Match the terms used in arrays against their corresponding descriptions.

Description		Term	
(A)	Has equal number of rows and columns.	(1)	Single-dimensional Array
(B)	Has improved performance.	(2)	Rectangular Array
(C)	Have many values in a single row.	(3)	Multi-dimensional Array
(D)	Has unequal number of columns.	(4)	Jagged Array
(E)	Have many rows and columns.		

(A)	A-1, B-3, C-2, D-4, E-3	(C)	A-1, B-4, C-3, D-2, E-1
(B)	A-4, B-1, C-2, D-3, E-4	(D)	A-2, B-4, C-1, D-4, E-3

4. Which of these statements about types of arrays are true?

(A)	The <code>foreach</code> loop is used to prevent any changes to the elements specified in the collection.
(B)	A single-dimensional array is referred to as an n -dimensional array where n is the number of elements.
(C)	A jagged array is a set of multiple arrays.
(D)	The <code>new</code> keyword is used to declare a single-dimensional array.
(E)	The single-dimensional array is a type of array that stores the first element in the index position 1.

(A)	A, C	(C)	C
(B)	B, D	(D)	D

5. Which of these statements about the `Array` class are true?

(A)	The <code>Array</code> class exists in the <code>System</code> namespace.
(B)	The <code>Array</code> class contains the <code>CreateInstance()</code> method that allows you to change the elements in an array.
(C)	The <code>Array</code> class declares methods to create and manipulate only multi-dimensional arrays.
(D)	The <code>Array</code> class defines the <code>Copy()</code> method to create a copy of an array.
(E)	The <code>Array</code> class contains the <code>Dimensions</code> property that determines the dimensions of an array.

(A)	A	(C)	C
(B)	B	(D)	D

6. Match the properties and methods of the `Array` class against their corresponding descriptions.

Description		Properties and Methods	
(A)	Places elements in another array from the specified index position.	(1)	<code>GetLength()</code>
(B)	Returns a 32-bit integer indicating the total number of elements in an array.	(2)	<code>CreateInstance()</code>
(C)	Returns an integer value indicating the number of dimensions in an array.	(3)	<code>CopyTo()</code>
(D)	Returns number of elements in an array.	(4)	<code>Length</code>
(E)	Creates an object of the <code>Array</code> class.	(5)	<code>Rank</code>

(A)	A-1, B-3, C-2, D-4, E-5	(C)	A-5, B-4, C-3, D-2, E-1
(B)	A-3, B-4, C-5, D-1, E-2	(D)	A-2, B-1, C-5, D-3, E-4

5.4.1 Answers

1.	B,
2.	A
3.	D
4.	A
5.	A
6.	B



Summary

- Arrays are a collection of values of the same data type.
- C# supports zero-based index feature.
- There are two types of arrays in C#: single-dimensional and multi-dimensional arrays.
- A single-dimensional array stores values in a single row whereas a multi-dimensional array stores values in a combination of rows and columns.
- Multi-dimensional arrays can be further classified into rectangular and jagged arrays.
- The `Array` class defined in the `System` namespace enables to create arrays easily.
- The `Array` class contains the `CreateInstance()` method, which allows you to create single and multi-dimensional arrays.

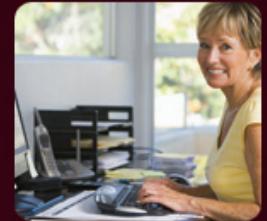
TechnoWise



Are you a
TECHIE GEEK
looking for updates?

Logon to

www.onlinevarsity.com



Session -6

Classes and Methods

Welcome to the Session, **Classes and Methods**.

This session provides an overview of C# as an object-oriented programming language. It describes the use of classes, objects, and methods in C#. The session introduces different access modifiers and explains the concept of method overloading wherein you can have multiple methods with the same name. Finally, the session explains the use of constructors and destructors.

In this session, you will learn to:

- Explain classes and objects
- Define and describe methods
- List the access modifiers
- Explain method overloading
- Define and describe constructors and destructors

6.1 Object-Oriented Programming

Programming languages have always been designed based on two fundamental concepts, data and ways to manipulate data. Traditional languages such as Pascal and C used the procedural approach which focused more on ways to manipulate data rather than on the data itself. This approach had several drawbacks such as lack of re-use and lack of maintainability.

To overcome these difficulties, OOP was introduced, which focused on data rather than the ways to manipulate data. The object-oriented approach defines objects as entities having a defined set of values and a defined set of operations that can be performed on these values.

Note - Some of the object-oriented programming languages are C++, C#, Java, and VB.NET.

6.1.1 Features

Object-oriented programming provides a number of features that distinguish it from the traditional programming approach. These features are as follows:

→ Abstraction

Abstraction is the feature of extracting only the required information from objects. For example, consider a television as an object. It has a manual stating how to use the television. However, this manual does not show all the technical details of the television, thus, giving only an abstraction to the user.

→ Encapsulation

Details of what a class contains need not be visible to other classes and objects that use it. Instead, only specific information can be made visible and the others can be hidden. This is achieved through encapsulation, also called data hiding. Both abstraction and encapsulation are complementary to each other.

→ Inheritance

Inheritance is the process of creating a new class based on the attributes and methods of an existing class. The existing class is called the base class whereas the new class created is called the derived class. This is a very important concept of object-oriented programming as it helps to reuse the inherited attributes and methods.

→ Polymorphism

Polymorphism is the ability to behave differently in different situations. It is basically seen in programs where you have multiple methods declared with the same name but with different parameters and different behavior.

6.2 Classes and Objects

C# programs are composed of classes that represent the entities of the program. The programs also include code to instantiate the classes as objects. When the program runs, objects are created for the classes and they may interact with each other to provide the functionalities of the program.

6.2.1 Objects

An object is a tangible entity such as a car, a table or a briefcase. Every object has some characteristics and is capable of performing certain actions.

The concept of objects in the real world can also be extended to the programming world. Like its real-world counterpart, an object in a programming language has a unique identity, state, and behavior. The identity of the object distinguishes it from the other objects of the same type. The state of the object refers to its characteristics or attributes whereas the behavior of the object comprises its actions. In simple terms, an object has various features that can describe it. These features could be the company name, model, price, mileage, and so on.

Figure 6.1 shows an example of an object with identity, state, and behavior.

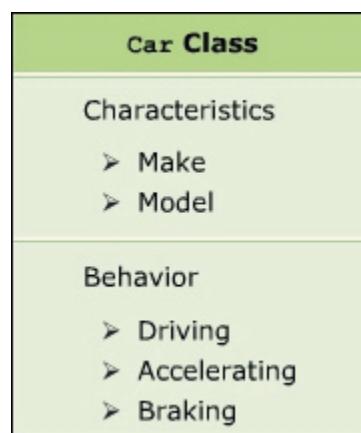


Figure 6.1: Object Identity, State, and Behavior

An object stores its identity and state in fields (also called variables) and exposes its behavior through methods.

6.2.2 Classes

Several objects have a common state and behavior and thus, can be grouped under a single class. For example, a Ford Mustang, a Volkswagen Beetle, and a Toyota Camry can be grouped together under the class Car. Here, Car is the class whereas Ford Mustang, Volkswagen Beetle, and Toyota Camry are objects of the class Car.

Figure 6.2 displays the class Car.

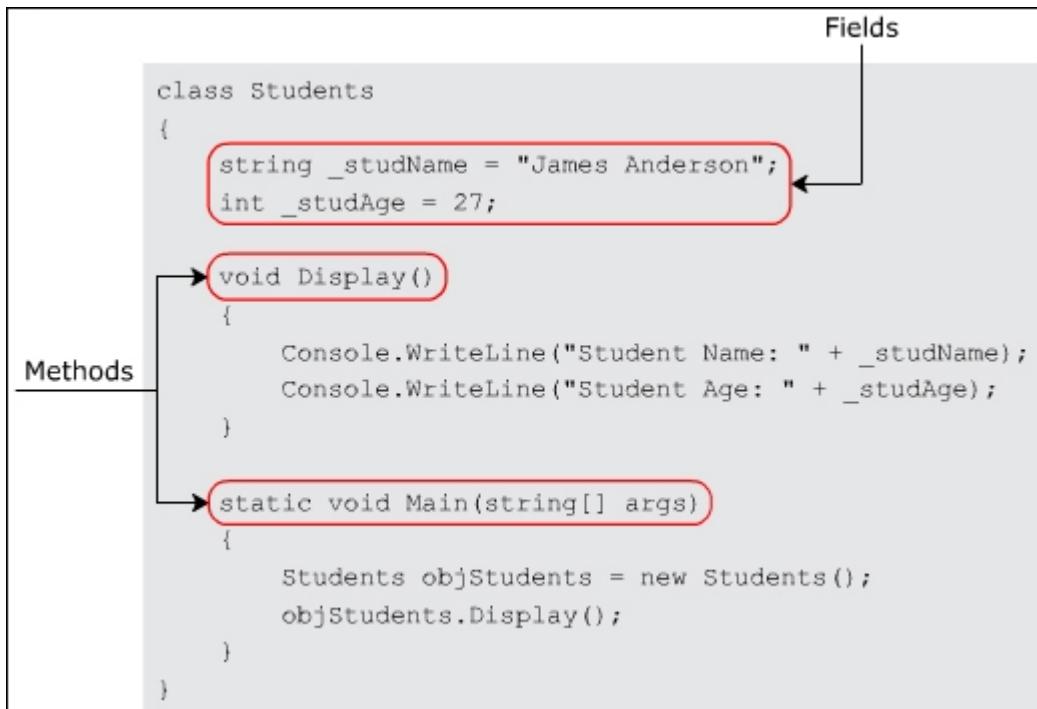


Figure 6.2: Class Car

6.2.3 Creating Classes

The concept of classes in the real world can be extended to the programming world, similar to the concept of objects. In object-oriented programming languages like C#, a class is a template or blueprint which defines the state and behavior of all objects belonging to that class.

A class comprises fields, properties, methods, and so on, collectively called data members of the class. In C#, the class declaration starts with the `class` keyword followed by the name of the class.

The following syntax is used to declare a class.

Syntax:

```

class <ClassName>
{
    // class members
}

```

where,

`ClassName`: Specifies the name of the class.

Figure 6.3 displays a sample class.

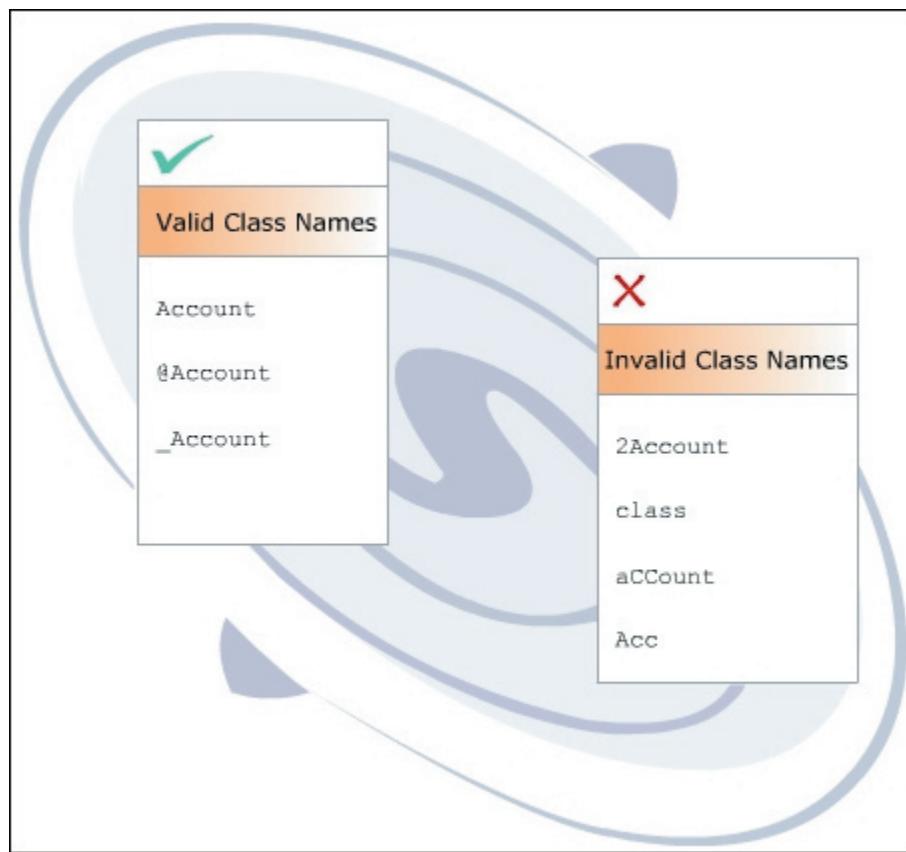


Figure 6.3: A Sample Class

6.2.4 Guidelines for Naming Classes

There are certain conventions to be followed for class names while creating a class. These conventions help you to follow a standard for naming classes. These conventions state that a class name:

- Should be a noun.
- Cannot be in mixed case and should have the first letter of each word capitalized.
- Should be simple, descriptive, and meaningful.
- Cannot be a C# keyword.
- Cannot begin with a digit. However, they can begin with the '@' character or an underscore (_), though this is not usually a recommended practice.

Some examples of valid class names are: **Account**, **@Account**, and **_Account**. Examples of invalid class names are: **2account**, **class**, **Acc count**, and **Account123**.

6.2.5 Main () Method

The `Main()` method indicates to the CLR that this is the first method of the program. This method is declared within a class and specifies where the program execution begins. Every C# program that is to be executed must have a `Main()` method as it is the entry point to the program. The return type of the `Main()` in C# can be `int` or `void`.

6.2.6 Instantiating Objects

When you create a class, it is necessary to create an object of the class to access the variables and methods defined within it. In C#, an object is instantiated using the `new` keyword. On encountering the `new` keyword, the JIT compiler allocates memory for the object and returns a reference of that allocated memory.

The following syntax is used to instantiate an object.

Syntax:

```
<ClassName> <objectName> = new <ClassName>();
```

where,

`ClassName`: Specifies the name of the class.

`objectName`: Specifies the name of the object.

Figure 6.4 displays an example of object instantiation.

```
class StudentDetails
{
    string _studName = "James" ;
    int rollNumber = 20;

    static void Main (string[] args)
    {
        StudentDetails objStudents = new StudentDetails();
        Console.WriteLine ("Student Name: "+
        objStudents._studName);
        Console.WriteLine ("Roll Number: "+
        objStudents._rollNumber);
    }
}
```

Figure 6.4: Object Instantiation

Note - When the code written in a .NET-compatible language such as C# is compiled, the output of the code is in the MSIL. To run this code on the computer, the MSIL code must be converted to a code native to the operating system. This is done by the JIT compiler.

6.3 Methods

Methods are functions declared in a class and may be used to perform operations on class variables. They are blocks of code that can take parameters and may or may not return a value. A method implements the behavior of an object, which can be accessed by instantiating the object of the class in which it is defined and then invoking the method. For example, the class `Car` can have a method `Brake()` that represents the 'Apply Brake' action. To perform the action, the method `Brake()` will have to be invoked by an object of class `Car`.

6.3.1 Creating Methods

Methods specify the manner in which a particular operation is to be carried out on the required data members of the class. They are declared within a class by specifying the return type of the method, method name, and an optional list of parameters. There are certain conventions that should be followed for naming methods. These conventions state that a method name:

- Cannot be a C# keyword
- Cannot contain spaces
- Cannot begin with a digit
- Can begin with a letter, underscore or the "@" character

Some examples of valid method names are: `Add()`, `Sum_Add()`, and `@Add()`.

Examples of invalid method names include `5Add`, `Add Sum()`, and `int()`.

The following syntax is used to create a method.

Syntax:

```
<access_modifier> <return_type> <MethodName> ([list of parameters])
{
    // body of the method
}
```

where,

`access_modifier`: Specifies the scope of access for the method. If no access modifier is specified, then, by default, the method will be considered as `private`.

`return_type`: Specifies the data type of the value that is returned by the method and it is optional. If the method does not return anything, the `void` keyword is mentioned here.

`MethodName`: Specifies the name of the method.

`list of parameters`: Specifies the arguments to be passed to the method.

Code Snippet 1 shows the definition of a method named `Add()` that adds two integer numbers.

Code Snippet 1:

```
class Sum
{
    int Add(int numOne, int numTwo)
    {
        int addResult = numOne + numTwo;
        Console.WriteLine("Addition=" + addResult);
    }
}
```

In Code Snippet 1, the `Add()` method takes two parameters of type `int` and performs addition of those two values. Finally, it displays the result of the addition operation.

Note - The `Main()` method of a class is mandatory if the program is to be executed. If an application has two or more classes, one of them must contain a `Main()`, failing which the application cannot be executed.

6.3.2 Invoking Methods

You can invoke a method in a class by creating an object of the class. To invoke a method, the object name is followed by a period (.) and the name of the method followed by parentheses.

In C#, a method is always invoked from another method. The method in which a method is invoked is referred to as the **calling** method. The invoked method is referred to as the **called** method. Most of the methods are invoked from the `Main()` method of the class, which is the entry point of the program execution.

Figure 6.5 displays how a method invocation or call is stored in the stack in memory and how a method body is defined.

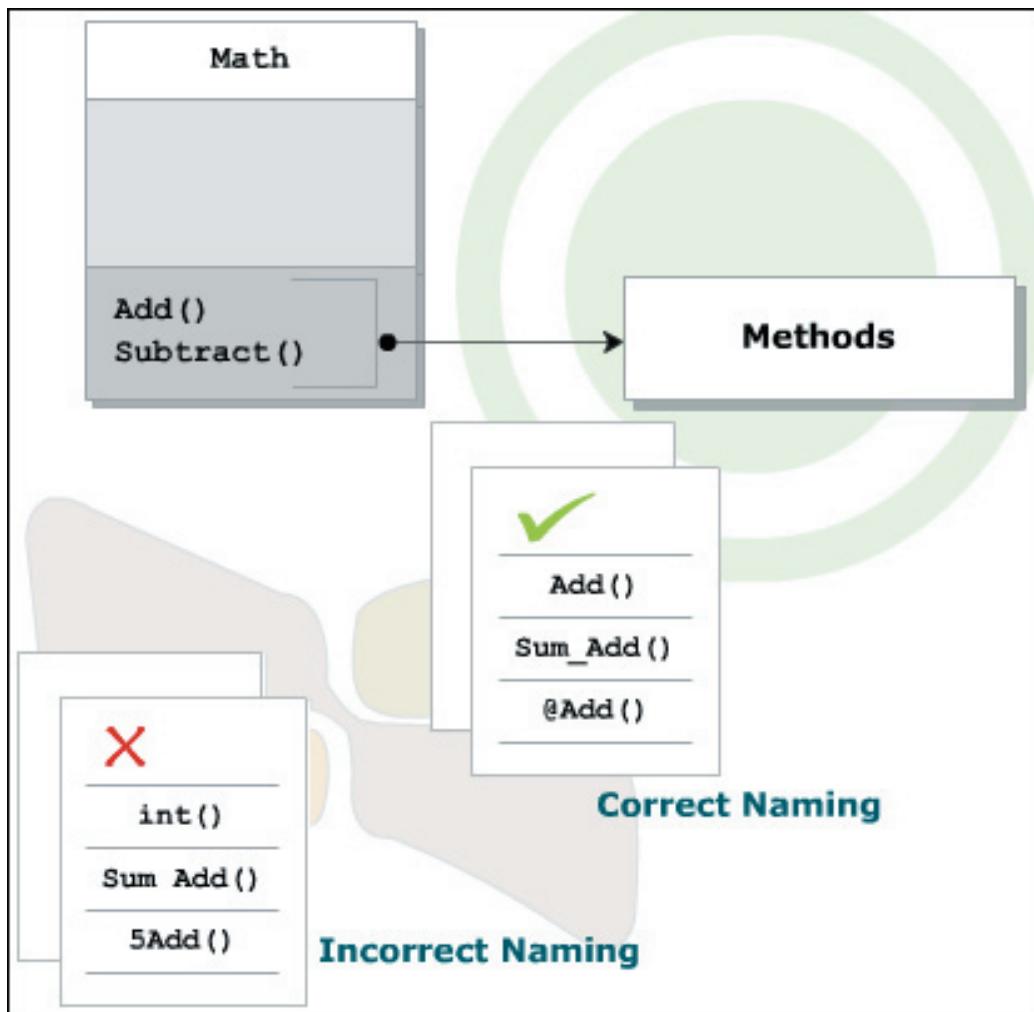


Figure 6.5: Method Invocation

Code Snippet 2 is used to define methods **Print()** and **Input()** in the **Book** class and then invoke them through an object **objBook** in the **Main()** method.

Code Snippet 2:

```
class Book
{
    string _bookName;
    public string Print()
    {
        return _bookName;
    }
}
```

```
public void Input(string bkName)
{
    _bookName = bkName;
}

static void Main(string[] args)
{
    Book objBook = new Book();
    objBook.Input("C#-The Complete Reference");
    Console.WriteLine(objBook.Print());
}
```

In Code Snippet 2, the `Main()` method is the calling method and the `Print()` and `Input()` methods are the called methods. The `Input()` method takes in the book name as a parameter and assigns the name of the book to the `_bookName` variable. Finally, the `Print()` method is called from the `Main()` method and it displays the name of the book as the output.

Output:

C#-The Complete Reference

6.3.3 Method Parameters and Arguments

The variables included in a method definition are called parameters. When the method is called, the data that you send into the method's parameters are called arguments.

A method may have zero or more parameters, enclosed in parentheses and separated by commas. If the method takes no parameters, it is indicated by empty parentheses.

Figure 6.6 shows an example of parameters and arguments.

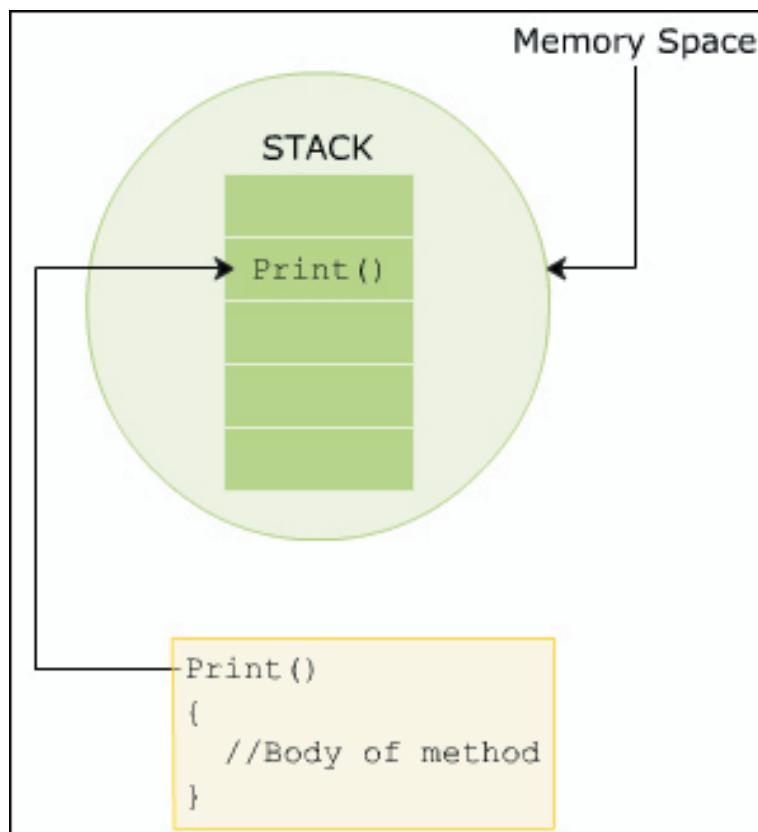


Figure 6.6: Parameters and Arguments

6.3.4 Named and Optional Arguments

A method in a C# program can accept multiple arguments. By default, the arguments are passed based on the position of the parameters in the method signature. However, a method caller can explicitly name one or more arguments being passed to the method instead of passing the arguments based on their position. Such an argument passed by its name instead of its position is called a named argument.

While passing named arguments, the order of the arguments declared in the method does not matter. The second argument of the method can be passed ahead of the first argument. Also, a named argument can follow positional arguments.

Named arguments are beneficial because you do not have to remember the exact order of parameters in the parameter list of methods.

Code Snippet 3 demonstrates how to use named arguments.

Code Snippet 3:

```
class Student
{
    void printName(String firstName, String lastName)
    {
        Console.WriteLine("First Name = {0}, Last Name = {1}", firstName, lastName);
    }

    static void Main(string[] args)
    {
        Student student = new Student();

        /* Passing argument by position */
        student.printName("Henry", "Parker");

        /* Passing named argument */
        student.printName(firstName: "Henry", lastName: "Parker");
        student.printName(lastName: "Parker", firstName: "Henry");

        /* Passing named argument after positional argument */
        student.printName("Henry", lastName: "Parker");
    }
}
```

In Code Snippet 3, the first call to the `printNamed()` method passes positional arguments. The second and third call passes named arguments in different orders. The fourth call passes a positional argument followed by a named argument.

Output:

```
First Name = Henry, Last Name = Parker
```

Note - In a method call, you cannot pass a named argument followed by a positional argument.

Code Snippet 4 shows another example of using named arguments.

Code Snippet 4:

```
class TestProgram
{
    voidCount(int boys, int girls)
    {
        Console.WriteLine(boys + girls);
    }

    static void Main(string[] args)
    {
        TestProgram objTest = new TestProgram();
        objTest.Count(boys: 16, girls: 24);
    }
}
```

C# also supports optional arguments in methods. An optional argument, as its name indicates is optional and can be emitted by the method caller. Each optional argument has a default value that is used if the caller does not provide the argument value.

Code Snippet 5 demonstrates how to use optional arguments.

Code Snippet 5:

```
class OptionalParameterExample
{
    void printMessage(String message="Hello user!")
    {
        Console.WriteLine("{0}", message);
    }

    static void Main(string[] args)
    {
        OptionalParameterExample opExample = new OptionalParameterExample();
        opExample.printMessage("Welcome User!");
        opExample.printMessage();
    }
}
```

In Code Snippet 5, the `printMessage()` method declares an optional argument message with a default value `Hello user!`. The first call to the `printMessage()` method passes an argument value that is printed on the console. The second call does not pass any value and therefore, the default value gets printed on the console.

Output:

Welcome User!

Hello user!

6.4 Static Classes

Classes that cannot be instantiated or inherited are known as static classes. To create a static class, during the declaration of the class, the `static` keyword is used before the class name. A static class can consist of static data members and static methods.

It is not possible to create an instance of a static class using the `new` keyword. The main features of static classes are as follows:

- They can only contain static members.
- They cannot be instantiated.
- They cannot be inherited.
- They cannot contain instance constructors. However, the developer can create static constructors to initialize the static members.

Since there is no need to create objects of the static class to call the required methods, the implementation of the program is simpler and faster than programs containing instance classes.

Code Snippet 6 creates a static class `Product` having static variables `_productId` and `price` and a static method called `Display()`. It also defines a constructor `Product()` which initializes the class variables to 10 and 156.32 respectively.

Code Snippet 6:

```
using System;

static class Product
{
    static int _productId;
    static double _price;
    static Product()
    {
        _productId = 10;
        _price = 156.32;
    }
}
```

```
_productId=10;  
_price=156.32;  
}  
  
public static void Display()  
{  
    Console.WriteLine("Product ID: " + _productId);  
    Console.WriteLine("Product price: " + _price);  
}  
}  
  
class Medicine  
{  
    static void Main(string[] args)  
    {  
        Product.Display();  
    }  
}
```

In Code Snippet 6, since the class **Product** is a static class, it is not instantiated. So, the method **Display()** is called by mentioning the class name followed by a period (.) and the name of the method.

Output:

```
Product ID: 10  
Product price: 156.32
```

6.5 Static Methods

A method, by default, is called using an object of the class. However, it is possible for a method to be called without creating any objects of the class. This can be done by declaring a method as static. A static method is declared using the **static** keyword. For example, the **Main()** method is a static method and it does not require any instance of the class for it to be invoked. A static method can directly refer only to static variables and other static methods of the class. However, static methods can refer to non-static methods and variables by using an instance of the class.

The following syntax is used to create a static method.

Syntax:

```
static<return_type><MethodName>()
{
    // body of the method
}
```

Code Snippet 7 creates a static method **Addition()** in the class **Calculate** and then invokes it in another class named **StaticMethods**.

Code Snippet 7:

```
class Calculate
{
    public static void Addition(int val1, int val2)
    {
        Console.WriteLine(val1 + val2);
    }

    public void Multiply(int val1, int val2)
    {
        Console.WriteLine(val1 * val2);
    }
}

class StaticMethods
{
    static void Main(string [] args)
    {
        Calculate.Addition(10, 50);

        Calculate objCal = new Calculate();
        objCal.Multiply(10, 20);
    }
}
```

In Code Snippet 7, the static method **Addition()** is invoked using the class name whereas the **Multiply()** method is invoked using the instance of the class. Finally, the results of the addition and multiplication operation are displayed in the console window.

Output:

60

200

Figure 6.7 displays invoking a static method.

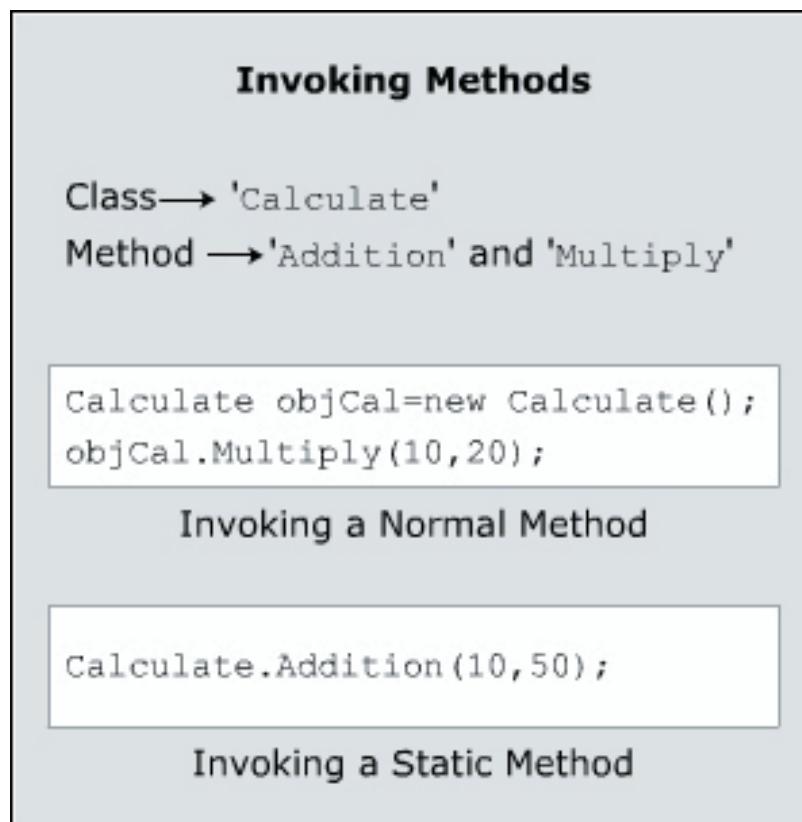


Figure 6.7: Invoking a Static Method

6.6 Static Variables

In addition to static methods, you can also have static variables in C#. A static variable is a special variable that is accessed without using an object of a class. A variable is declared as static using the `static` keyword. When a static variable is created, it is automatically initialized before it is accessed.

Only one copy of a static variable is shared by all the objects of the class. Therefore, a change in the value of such a variable is reflected by all the objects of the class. An instance of a class cannot access static variables.

Figure 6.8 displays the static variables.

```
class Employee
{
    public static int EmpId = 20 ;
    public static string EmpName = "James";

    static void Main (string[] args)
    {

        Console.WriteLine ("Employee ID: " + EmpId);
        Console.WriteLine ("Employee Name: " + EmpName);
    }
}
```

Figure 6.8: Static Variables

6.7 Access Modifiers

Object-oriented programming enables you to restrict access of data members defined in a class so that only specific classes can access them. To specify these restrictions, C# provides access modifiers that allow you to specify which classes can access the data members of a particular class. Access modifiers are specified using C# keywords.

In C#, there are four commonly used access modifiers. These are as follows:

→ **public**

The `public` access modifier provides the most permissive access level. The members declared as `public` can be accessed anywhere in the class as well as from other classes.

Code Snippet 8 declares a public string variable called `Name` to store the name of the person. This means it can be publicly accessed by any other class.

Code Snippet 8:

```
class Employee
{
    // No access restrictions.
    public string Name = "Wilson";
}
```

→ **private**

The `private` access modifier provides the least permissive access level. Private members are accessible only within the class in which they are declared.

Code Snippet 9 declares a variable called `_salary` as `private`, which means it cannot be accessed by any other class except for the `Employee` class.

Code Snippet 9:

```
class Employee8
{
    // Accessible only within the class
    private float _salary;
}
```

→ **protected**

The `protected` access modifier allows the class members to be accessible within the class as well as within the derived classes.

Code Snippet 10 declares a variable called `Salary` as `protected`, which means it can be accessed only by the `Employee` class and its derived classes.

Code Snippet 10:

```
class Employee
{
    // Protected access
    protected float Salary;
}
```

→ **internal**

The `internal` access modifier allows the class members to be accessible only within the classes of the same assembly. An assembly is a file that is automatically generated by the compiler upon successful compilation of a .NET application.

Code Snippet 11 declares a variable called `NumOne` as `internal`, which means it has only assembly-level access.

Code Snippet 11:

```
public class Sample
{
    // Only accessible within the same assembly
    internal static int NumOne = 3;
}
```

Figure 6.9 displays the various accessibility levels.

		Accessibility Levels		
		Applicable to the Application	Applicable to the Current Class	Applicable to the Derived Class
Access Modifiers	public	✓	✓	✓
	private	✗	✓	✗
	protected	✗	✓	✓
	internal	✗	✓	✓

Figure 6.9: Accessibility Levels

Note - C# also supports other data structure types such as interfaces, enumerations and structures. The four accessibility levels - public, private, protected, and internal – can be applied to these types too.

6.8 *ref* and *out* Keywords

The *ref* keyword causes arguments to be passed in a method by reference. In call by reference, the called method changes the value of the parameters passed to it from the calling method. Any changes made to the parameters in the called method will be reflected in the parameters passed from the calling method when control passes back to the calling method.

It is necessary that both the called method and the calling method must explicitly specify the *ref* keyword before the required parameters. The variables passed by reference from the calling method must be first initialized.

The following syntax is used to pass values by reference using the *ref* keyword.

Syntax:

```
<access_modifier> <return_type> <MethodName> (ref parameter1, ref parameter2,
parameter3, parameter4, ... parameterN)

{
// actions to be performed
}
```

where,

`parameter1...parameterN`: Specifies that there can be any number of parameters and it is not necessary for all the parameters to be `ref` parameters.

Code Snippet 12 uses the `ref` keyword to pass the arguments by reference.

Code Snippet 12:

```
class RefParameters
{
    static void Calculate(ref int numValueOne, ref int numValueTwo)
    {
        numValueOne = numValueOne * 2;
        numValueTwo = numValueTwo / 2;
    }

    static void Main(string[] args)
    {
        int numOne = 10;
        int numTwo = 20;
        Console.WriteLine("Value of Num1 and Num2 before calling method "
            + numOne + ", " + numTwo);
        Calculate(ref numOne, ref numTwo);
        Console.WriteLine("Value of Num1 and Num2 after calling method "
            + numOne + ", " + numTwo);
    }
}
```

In Code Snippet 12, the `Calculate()` method is called from the `Main()` method, which takes the parameters prefixed with the `ref` keyword. The same keyword is also used in the `Calculate()` method before the variables `numValueOne` and `numValueTwo`. In the `Calculate()` method, the multiplication and division operations are performed on the values passed as parameters and the results are stored in the `numValueOne` and `numValueTwo` variables respectively. The resultant values stored in these variables are also reflected in the `numOne` and `numTwo` variables respectively as the values are passed by reference to the method `Calculate()`.

Figure 6.10 displays the use of `ref` keyword.

```
c:\> C:\WINDOWS\system32\cmd.exe
Value of Num1 and Num2 before calling method 10, 2
Value of Num1 and Num2 after calling method 20, 10
Press any key to continue . . .
```

Figure 6.10: Use of `ref` Keyword

Note - It is not necessary for all the method parameters to be `ref` parameters.

The `out` keyword is similar to the `ref` keyword and causes arguments to be passed by reference. The only difference between the two is that the `out` keyword does not require the variables that are passed by reference to be initialized. Both the called method and the calling method must explicitly use the `out` keyword.

The following syntax is used to pass values by reference using the `out` keyword.

Syntax:

```
<access_modifier> <return_type> <MethodName> (out parameter1, out parameter2,
... parameterN)

{
    // actions to be performed
}
```

where,

`parameter 1...parameterN`: Specifies that there can be any number of parameters and it is not necessary for all the parameters to be `out` parameters.

Code Snippet 13 uses the `out` keyword to pass the parameters by reference.

Code Snippet 13:

```
class OutParameters
{
    static void Depreciation(out int val)
    {
        val=20000;
        int dep=val * 5/100;
        int amt=val - dep;
        Console.WriteLine("Depreciation Amount: " + dep);
    }
}
```

```

        Console.WriteLine ("Reduced value after depreciation: " + amt);
    }

    static void Main(string[] args)
    {
        int value;
        Depreciation(out value);
    }
}

```

In Code Snippet 13, the `Depreciation()` method is invoked from the `Main()` method passing the `val` parameter using the `out` keyword. In the `Depreciation()` method, the depreciation is calculated and the resultant depreciated amount is deducted from the `val` variable. The final value in the `amt` variable is displayed as the output.

Figure 6.11 displays the use of `out` keyword.

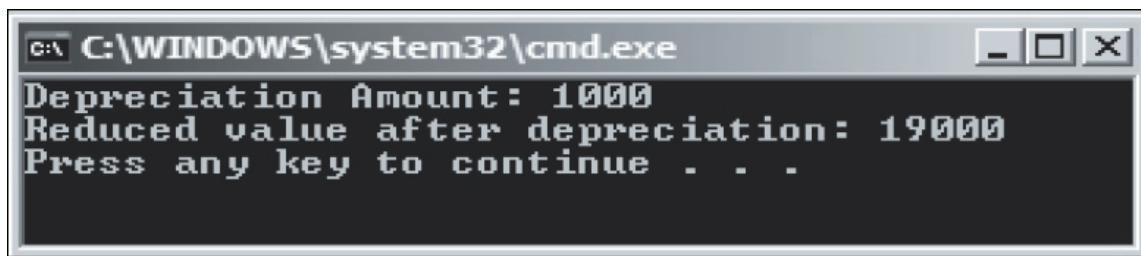


Figure 6.11: Use of out Keyword

6.9 Method Overloading

Consider a holiday resort having two employees with the same name, Peter. One is the chef while the other is in charge of maintenance. The delivery man knows which Peter does what, so if there is a delivery of fresh vegetables for a Mr. Peter, it is forwarded to the chef. Similarly, if there is a delivery of electric bulbs and wires for a Mr. Peter, it is forwarded to the maintenance person.

Though there are two Peters, the items delivered give a clear indication to which Peter the delivery has to be made.

Similarly, in C#, two methods can have the same name as they can be distinguished by their signatures. This feature is called method overloading.

6.9.1 Method Overloading in C#

In object-oriented programming, every method has a signature. This comprises the number of parameters passed to the method, the data types of parameters and the order in which the parameters are written. While declaring a method, the signature of the method is written in parentheses next to the method name.

No class is allowed to contain two methods with the same name and same signature. However, it is possible for a class to have two methods having the same name but different signatures. The concept of declaring more than one method with the same method name but different signatures is called method overloading.

Figure 6.12 displays the concept of method overloading using an example.

```

int Addition (int valOne, int valTwo)
{
    return valOne + valTwo;
}

int Addition (int valOne, int valTwo)
{
    int result = valOne + valTwo;
    return result;
} X  

Not Allowed in C#
```



```

int Addition (int valOne, int valTwo)
{
    return valOne + valTwo;
}

int Addition (int valOne, int valTwo, int valThree)
{
    return valOne + valTwo + valThree; ✓  

Allowed in C#
```

Figure 6.12: Method Overloading

Code Snippet 14 overloads the **Square()** method to calculate the square of the given **int** and **float** values.

Code Snippet 14:

```

class MethodOverloadExample
{
    static void Main(string[] args)
    {
        Console.WriteLine("Square of integer value " + Square(5));
        Console.WriteLine("Square of float value " + Square(2.5F));
```

```

    }

    static int Square(int num)

    {

        return num * num;

    }

    static float Square(float num)

    {

        return num * num;

    }

}

```

In Code Snippet 14, two methods with the same name but with different parameters are declared in the class. The two **Square()** methods take in parameters of **int** type and **float** type respectively. Within the **Main()** method, depending on the type of value passed, the appropriate method is invoked and the square of the specified number is displayed in the console window.

Output:

Square of integer value 25

Square of float value 6.25

Note - The signatures of two methods are said to be the same if all the three conditions - the number of parameters passed to the method, the parameter types and the order in which the parameters are written - are the same. The return type of a method is not a part of its signature.

6.9.2 Guidelines and Restrictions

While overloading methods in a program, you must follow certain guidelines to ensure that the overloaded methods function accurately. These guidelines are as follows:

- ➔ The methods to be overloaded should perform the same task. Though a program will not raise any compiler error if this is violated, for best practices it is recommended that they perform the same task.
- ➔ The signatures of the overloaded methods must be unique.
- ➔ When overloading methods, the return type of the methods can be the same as it is not a part of the signature.
- ➔ The **ref** and **out** parameters can be included as a part of the signature in overloaded methods.

6.9.3 The `this` Keyword

The `this` keyword is used to refer to the current object of the class. It is used to resolve conflicts between variables having same names and to pass the current object as a parameter. You cannot use the `this` keyword with static variables and methods.

Code Snippet 15 uses the `this` keyword to refer to the `_length` and `_breadth` fields of the current instance of the class `Dimension`.

Code Snippet 15:

```
class Dimension
{
    double _length;
    double _breadth;
    public double Area(double _length, double _breadth)
    {
        this._length=_length;
        this._breadth=_breadth;
        return _length * _breadth;
    }
    static void Main(string[] args)
    {
        Dimension objDimension = new Dimension();
        Console.WriteLine("Area of rectangle = " + objDimension.Area(10.5, 12.5));
    }
}
```

In Code Snippet 15, the `Area()` method has two parameters `_length` and `_breadth` as instance variables. The values of these variables are assigned to the class variables using the `this` keyword. The method is invoked in the `Main()` method. Finally, the area is calculated and is displayed as output in the console window.

Figure 6.13 displays the use of `this` keyword.

```
C:\WINDOWS\system32\cmd.exe
Area of rectangle = 131.25
Press any key to continue . . .
```

Figure 6.13: Use of this Keyword

6.10 Constructors and Destructors

A C# class can contain one or more special member functions having the same name as the class, called constructors. Constructors are executed when an object of the class is created in order to initialize the object with data. A C# class can also have a destructor (only one is allowed per class), which is a special method and also has the same name as the class but prefixed with a special symbol ~. A destructor of an object is executed when the object is no longer required in order to de-allocate memory of the object.

6.10.1 Constructors

A class can contain multiple variables whose declaration and initialization becomes difficult to track if they are done within different blocks. Likewise, there may be other startup operations that need to be performed in an application like opening a file and so forth. To simplify these tasks, a constructor is used. A constructor is a method having the same name as that of the class. Constructors can initialize the variables of a class or perform startup operations only once when the object of the class is instantiated. They are automatically executed whenever an instance of a class is created.

Figure 6.14 shows the constructor declaration.

```
C:\WINDOWS\system32\cmd.exe
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

CircleDetails.cs(11,28): error CS0122: 'Circle.Circle()' is inaccessible due
    to its protection level
CircleDetails.cs(3,13): <Location of symbol related to previous error>
Press any key to continue . . .
```

Figure 6.14: Constructor Declaration

It is possible to specify the accessibility level of constructors within an application. This is done by the use of access modifiers such as:

- **public:** Specifies that the constructor will be called whenever a class is instantiated. This instantiation can be done from anywhere and from any assembly.

- **private**: Specifies that this constructor cannot be invoked by an instance of a class.
- **protected**: Specifies that the base class will initialize on its own whenever its derived classes are created. Here, the class object can only be created in the derived classes.
- **internal**: Specifies that the constructor has its access limited to the current assembly. It cannot be accessed outside the assembly.

Code Snippet 16 creates a class **Circle** with a **private** constructor.

Code Snippet 16:

```
public class Circle
{
    private Circle()
    {
    }
}

class CircleDetails
{
    public static void Main(string[] args)
    {
        Circle objCircle = new Circle();
    }
}
```

In Code Snippet 16, the program will generate a compile-time error because an instance of the **Circle** class attempts to invoke the constructor which is declared as private. This is an illegal attempt. Private constructors are used to prevent class instantiation. If a class has defined only private constructors, the **new** keyword cannot be used to instantiate the object of the class. This means no other class can use the data members of the class that has only private constructors. Therefore, private constructors are only used if a class contains only static data members. This is because static members are invoked using the class name.

Figure 6.15 shows the output for creating a class **Circle** with a private constructor.

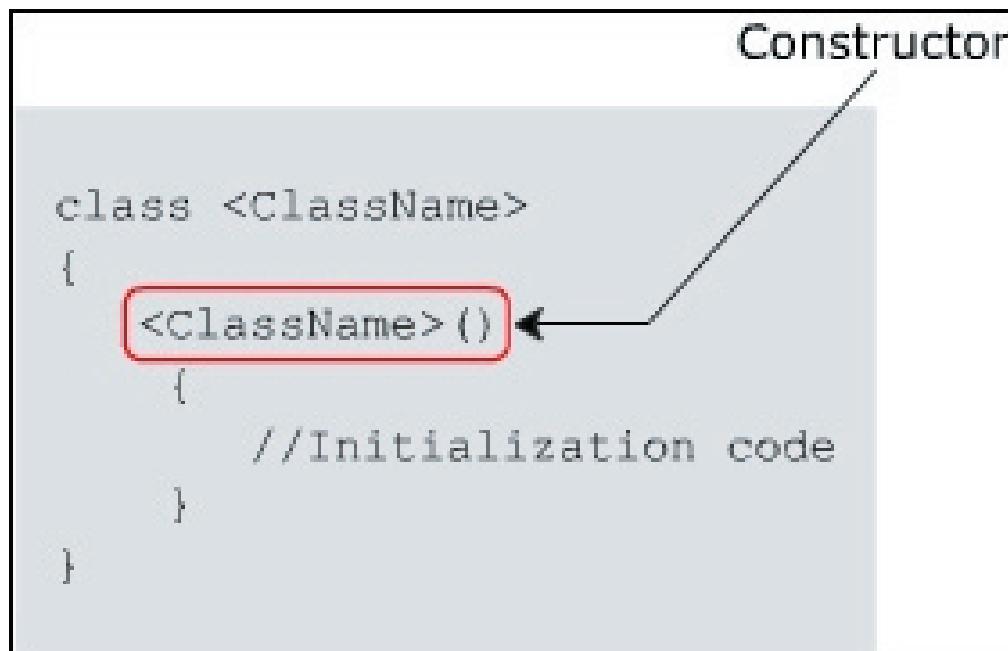


Figure 6.15: Class Circle with Private Constructor

Code Snippet 17 is used to initialize the values of `_empName`, `_empAge`, and `_deptName` with the help of a constructor.

Code Snippet 17:

```

class Employees
{
    string _empName;
    int _empAge;
    string _deptName;
    Employees(string name, int num)
    {
        _empName = name;
        _empAge = num;
        _deptName = "Research & Development";
    }
    static void Main(string[] args)
    {
    }
}

```

```
Employees objEmp = new Employees ("John", 10);  
Console.WriteLine(objEmp._deptName);  
}  
}
```

In Code Snippet 17, a constructor is created for the class `Employees`. When the class is instantiated, the constructor is invoked with the parameters `John` and `10`. These values are stored in the class variables `empName` and `empAge` respectively. The department of the employee is then displayed in the console window.

Note - Constructors have no return type. This is because the implicit return type of a constructor is the class itself. It is possible to have overloaded constructors in C#.

6.10.2 Default Constructors

C# creates a default constructor for a class if no constructor is specified within the class. The default constructor automatically initializes all the numeric data type instance variables of the class to zero. If you define a constructor in the class, the default constructor is no longer used.

6.10.3 Static Constructors

A static constructor is used to initialize static variables of the class and to perform a particular action only once. It is invoked before any static member of the class is accessed. You can have only one static constructor in the class. The `static` keyword is used to declare a constructor as static.

A static constructor does not take any parameters and does not use any access modifiers because it is invoked directly by the CLR instead of the object. In addition, it cannot access any non-static data member of the class.

Figure 6.16 illustrates the syntax for a static constructor.

```
class <ClassName>
{
    static <ClassName>() ← Static Constructor
    {
        //Initialization code
    }
}
```

Figure 6.16: Static Constructor

Code Snippet 18 shows how static constructors are created and invoked.

Code Snippet 18:

```
class Multiplication
{
    static int _valueOne = 10;
    static int _product;
    static Multiplication()
    {
        Console.WriteLine("Static Constructor initialized");
        _product = _valueOne * _valueOne;
    }
    public static void Method()
    {
        Console.WriteLine("Value of product = " + _product);
    }
    static void Main(string[] args)
    {
        Multiplication.Method();
    }
}
```

In Code Snippet 18, the static constructor `Multiplication()` is used to initialize the static variable `_product`. Here, the static constructor is invoked before the static method `Method()` is called from the `Main()` method.

Output:

Static Constructor initialized

Value of product = 100

6.10.4 Constructor Overloading

The concept of declaring more than one constructor in a class is called constructor overloading. The process of overloading constructors is similar to overloading methods. Every constructor has a signature similar to that of a method. You can declare multiple constructors in a class wherein each constructor will have different signatures. Constructor overloading is used when different objects of the class might want to use different initialized values. Overloaded constructors reduce the task of assigning different values to member variables each time when needed by different objects of the class.

Code Snippet 19 demonstrates the use of constructor overloading.

Code Snippet 19:

```
public class Rectangle
{
    double _length;
    double _breadth;
    public Rectangle()
    {
        _length=13.5;
        _breadth=20.5;
    }
    public Rectangle(double len, double wide)
    {
        _length=len;
        _breadth=wide;
    }
    public double Area()
    {
        return _length*_breadth;
    }
}
```

```

    }

    static void Main(string[] args)
    {

        Rectangle objRect1 = new Rectangle();

        Console.WriteLine("Area of rectangle = " + objRect1.Area());

        Rectangle objRect2 = new Rectangle(2.5, 6.9);

        Console.WriteLine("Area of rectangle = " + objRect2.Area());
    }
}

```

In Code Snippet 19, two constructors are created having the same name, `Rectangle`. However, the signatures of these constructors are different. Hence, while calling the method `Area()` from the `Main()` method, the parameters passed to the calling method are identified. Then, the corresponding constructor is used to initialize the variables `_length` and `_breadth`. Finally, the multiplication operation is performed on these variables and the area values are displayed as the output.

Output:

Area of rectangle1 = 276.75

Area of rectangle2 = 17.25

6.10.5 Destructors

A destructor is a special method which has the same name as the class but starts with the character ~ before the class name. Destructors immediately de-allocate memory of objects that are no longer required. They are invoked automatically when the objects are not in use. You can define only one destructor in a class. Apart from this, destructors have some more features. These features are as follows:

- Destructors cannot be overloaded or inherited.
- Destructors cannot be explicitly invoked.
- Destructors cannot specify access modifiers and cannot take parameters.

Code Snippet 20 demonstrates the use of destructors.

Code Snippet 20:

```

class Employee
{
    private int _empId;
}

```

```

private string _empName;
private int _age;
private double _salary;
Employee(int id, string name, int age, double sal)
{
    Console.WriteLine("Constructor for Employee called");
    _empId = id;
    _empName = name;
    _age = age;
    _salary = sal;
}
~Employee()
{
    Console.WriteLine("Destructor for Employee called");
}
static void Main(string[] args)
{
    Employee objEmp = new Employee(1, "John", 45, 35000);
    Console.WriteLine("Employee ID: " + objEmp._empId);
    Console.WriteLine("Employee Name: " + objEmp._empName);
    Console.WriteLine("Age: " + objEmp._age);
    Console.WriteLine("Salary: " + objEmp._salary);
}
}

```

In Code Snippet 20, the destructor `~Employee` is created having the same name as that of the class and the constructor. The destructor is automatically called when the object `objEmp` is no longer needed to be used. However, when this will happen cannot be determined and hence, you have no control on when the destructor is going to be executed.

6.11 Check Your Progress

1. Which of these statements about object-oriented programming and objects are true?

(A)	Object-oriented programming focuses on the data-based approach.
(B)	An object stores its state in variables and implements its behavior through methods.
(C)	Classes contain objects that may or may not have a unique identity.
(D)	Declaration of classes involves the use of the class keyword.
(E)	Creating objects involves the use of the new keyword.

(A)	A, B, D, E	(C)	B, C, D
(B)	A, C	(D)	A, B, C

2. Match the object-oriented programming terms against their corresponding descriptions.

Description		OOP Term	
(A)	Represents the behavior of an object.	1.	Object
(B)	Represents an instance of a class.	2.	Class
(C)	Represents the state of an object.	3.	Method
(D)	Defines the state and behavior for all objects belonging to a particular entity.	4.	Field
(E)	Describes an entity.		

(A)	(A)-(4), (B)-(3), (C)-(1), (D)-(2), (E)-(1)	(C)	(A)-(2), (B)-(3), (C)-(2), (D)-(3), (E)-(1)
(B)	(A)-(3), (B)-(1), (C)-(4), (D)-(2), (E)-(2)	(D)	(A)-(1), (B)-(2), (C)-(2), (D)-(4), (E)-(3)

3. Which of these statements about methods are true?

(A)	A static method can directly refer only to static variables of the class.
(B)	A method definition includes the return type as null if the method does not return any value.
(C)	A method name begins with the & symbol or an underscore.
(D)	A static method uses the static keyword in its declaration.
(E)	A method declaration can specify multiple parameters to be used in the method.

(A)	A, B, D	(C)	B, C, D
(B)	A, C	(D)	A, D, E

4. You are trying to display the area of a rectangle. Which of the following codes will help you to achieve this assuming that the System namespace is included in the program?

```
(A) class Dimensions
{
    static double _length;
    static double _breadth;
    public static double Area()
    {
        return _length * _breadth;
    }
    public static void SetDimension(double numOne, double numTwo)
    {
        _length = numOne;
        _breadth = numTwo;
    }
}
class StaticMethods
{
    static void Main(string [] args)
    {
```

(A)	<pre>Dimension.SetDimension(20.1, 18.3); Console.WriteLine("Area of Rectangle: " + Dimensions.Area()); } }</pre>
(B)	<pre>class Dimensions { static double _length; static double _breadth; public static double Area() { return _length * _breadth; } public void SetDimension(double numOne, double numTwo) { _length = numOne; _breadth = numTwo; } } class StaticMethods { static void Main(string[] args) { Dimensions.SetDimension(20.1, 18.3); Console.WriteLine("Area of Rectangle: " + Dimensions.Area()); } }</pre>
(C)	<pre>class Dimensions { static double _length; static double _breadth; public static double Area() { return _length * _breadth; }</pre>

(C)	<pre> } public static void SetDimension(double numOne, double numTwo) { _length = numOne; _breadth = numTwo; } } class StaticMethods { static void Main(string[] args) { Dimensions.SetDimension(20.1, 18.3); Console.WriteLine("Area of Rectangle: " + Dimensions.Area()); } } </pre>
(D)	<pre> class Dimensions { static double _length; static double _breadth; public static double Area() { return _length * _breadth; } public static void SetDimension(double numOne, double numTwo) { _length = numOne; _breadth = numTwo; } } class StaticMethods </pre>

(D)	{ static void Main(string[] args) { Dimensions objDimensions = new Dimensions(); objDimensions.SetDimension(20.1, 18.3); Console.WriteLine("Area of Rectangle: " + objDimensions.Area()); } }
-----	---

(A)	A	(C)	C
(B)	B	(D)	D

5. Match the keywords used for access modifiers against their corresponding descriptions.

Description		Access Modifier	
(A)	Allows access to the member by all classes	1.	private
(B)	Allows access to the members only in the same assembly	2.	public
(C)	Allows access to the members only within the class	3.	protected
(D)	Provides the most permissive access level	4.	internal
(E)	Allows access to the base class members in the derived classes		

(A)	(A)-(4), (B)-(3), (C)-(1), (D)-(2), (E)-(1)	(C)	(A)-(2), (B)-(4), (C)-(1), (D)-(2), (E)-(3)
(B)	(A)-(3), (B)-(1), (C)-(4), (D)-(2), (E)-(2)	(D)	(A)-(1), (B)-(2), (C)-(2), (D)-(4), (E)-(3)

6. Which of these statements about method overloading and the this keyword are true?

(A)	The return types of the overloaded methods should be the same.
(B)	Overloaded methods should contain parameters of the same type.
(C)	The ref and out keywords should not be part of the signature of overloaded methods.
(D)	Overloaded methods should have the same method name but with different casing.
(E)	The this keyword is used to pass the current object as a parameter to a method.

(A)	A, B, D	(C)	B, C, D
(B)	A, C	(D)	E

7. Which of these statements about constructors and destructors in C# are true?

(A)	A constructor can have return types such as int and void.
(B)	A class can have more than one constructor.
(C)	A static constructor can contain the public access modifier.
(D)	A destructor can contain the private access modifier.
(E)	A static constructor can refer to static variables only.

(A)	A, B, D	(C)	B, C, D
(B)	A, C	(D)	B, E

6.11.1 Answers

1.	A
2.	B
3.	D
4.	C
5.	C
6.	D
7.	D



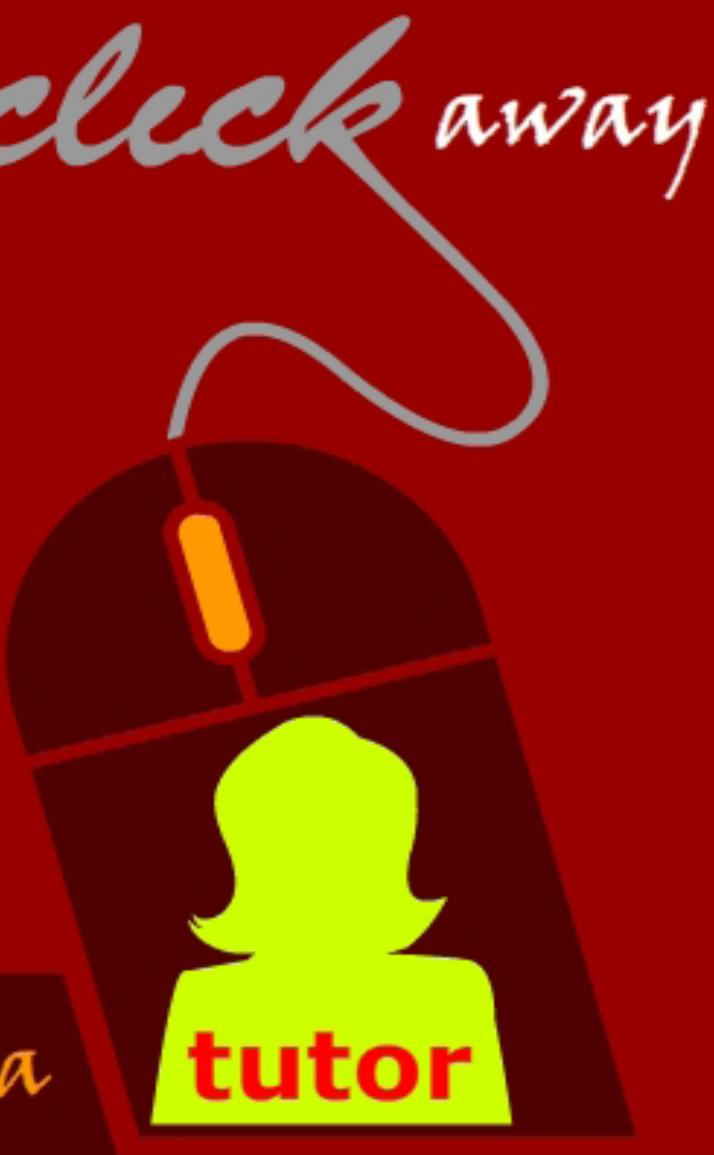
Summary

- ➔ The programming model that uses objects to design a software application is termed as OOP.
- ➔ A method is defined as the actual implementation of an action on an object and can be declared by specifying the return type, the name and the parameters passed to the method.
- ➔ It is possible to call a method without creating instances by declaring the method as static.
- ➔ Access modifiers determine the scope of access for classes and their members.
- ➔ The four types of access modifiers in C# are public, private, protected, and internal.
- ➔ Methods with same name but different signatures are referred to as overloaded methods.
- ➔ In C#, a constructor is typically used to initialize the variables of a class.

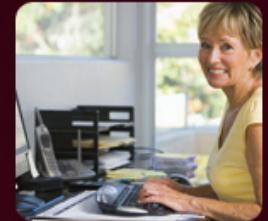
Are you looking for online **HELP?**

We are just a *click* away

To chat with a



Login to www.onlinevarsity.com



Session -7

Inheritance and Polymorphism

Welcome to the Session, **Inheritance and Polymorphism**.

Inheritance is a process of creating a new class from an existing class. Inheritance allows you to inherit attributes and methods of the base class in the newly created class. Polymorphism is a feature of object-oriented programming that allows the data members of a class to behave differently based on their parameters and data types.

In this session, you will learn to:

- Define and describe inheritance
- Explain method overriding
- Define and describe sealed classes
- Explain polymorphism

7.1 Inheritance

A programmer does not always need to create a class in a C# application from scratch. At times, the programmer can create a new class by extending the features of an existing class. The process of creating a new class by extending some features of an existing class is known as inheritance.

7.1.1 Definition of Inheritance

The similarity in physical features of a child to that of its parents is due to the child having inherited these features from its parents. Similarly, in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.

The class from which the new class is created is known as the base class and the created class is known as the derived class.

For example, consider a class called **Vehicle** that consists of a variable called **color** and a method called **Speed()**. These data members of the **Vehicle** class can be inherited by the **TwoWheelerVehicle** and **FourWheelerVehicle** classes. Figure 7.1 illustrates this example.

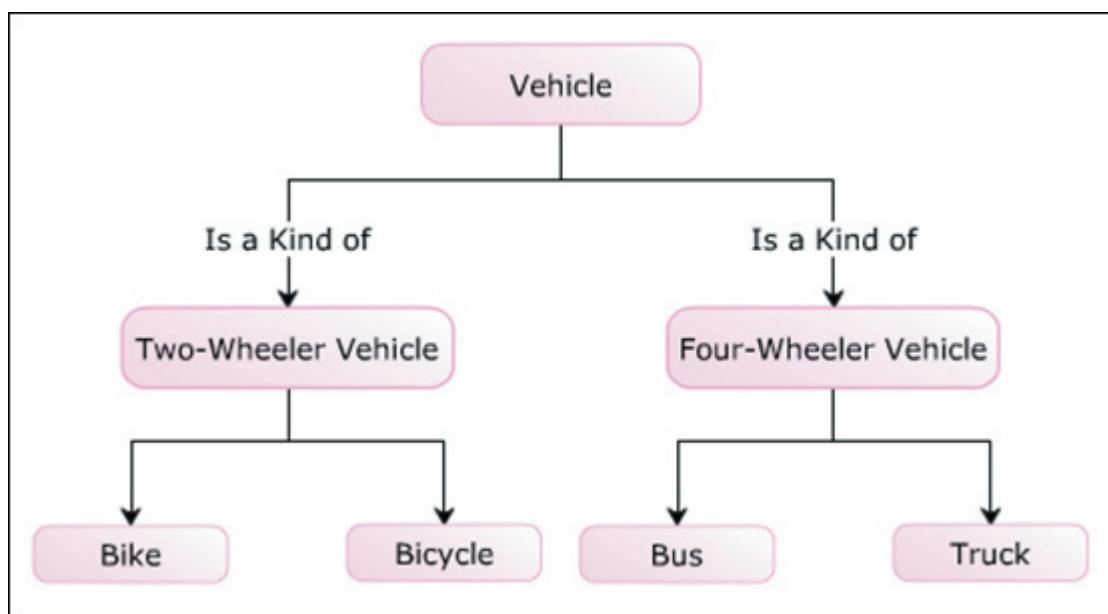


Figure 7.1: Example of Inheritance

7.1.2 Purpose

The purpose of inheritance is to reuse common methods and attributes among classes without recreating them. Reusability of a code enables you to use the same code in different applications with little or no changes. Consider a class named **Animal** which defines attributes and behavior for animals. If a new class named **Cat** has to be created, it can be done based on **Animal** because cat is also an animal. Thus, you can reuse the code from the previously-defined class.

Apart from reusability, inheritance is widely used for:

→ Generalization

Inheritance allows you to implement generalization by creating base classes. For example, consider the class **Vehicle**, which is the base class for its derived classes **Truck** and **Bike**. The class **Vehicle** consists of general attributes and methods that are implemented more specifically in the respective derived classes.

→ Specialization

Inheritance allows you to implement specialization by creating derived classes.

For example, the derived classes such as **Bike**, **Bicycle**, **Bus**, and **Truck** are specialized by implementing only specific methods from its generalized base class **Vehicle**.

→ Extension

Inheritance allows you to extend the functionalities of a derived class by creating more methods and attributes that are not present in the base class. It allows you to provide additional features to the existing derived class without modifying the existing code.

Figure 7.2 displays a real-world example demonstrating the purpose of inheritance.

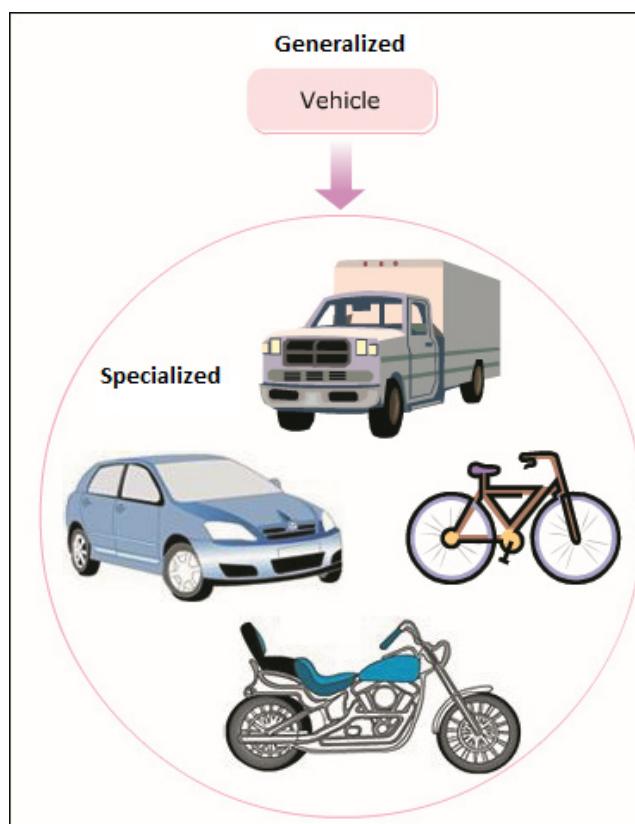


Figure 7.2: Purpose of Inheritance

7.1.3 Multi-level Hierarchy

Inheritance allows the programmer to build hierarchies that can contain multiple levels of inheritance. For example, consider three classes **Mammal**, **Animal**, and **Dog**. The class **Mammal** is inherited from the base class **Animal**, which inherits all the attributes of the **Animal** class. The class **Dog** is inherited from the class **Mammal** and inherits all the attributes of both the **Animal** and **Mammal** classes.

Figure 7.3 depicts multi-level hierarchy of related classes.

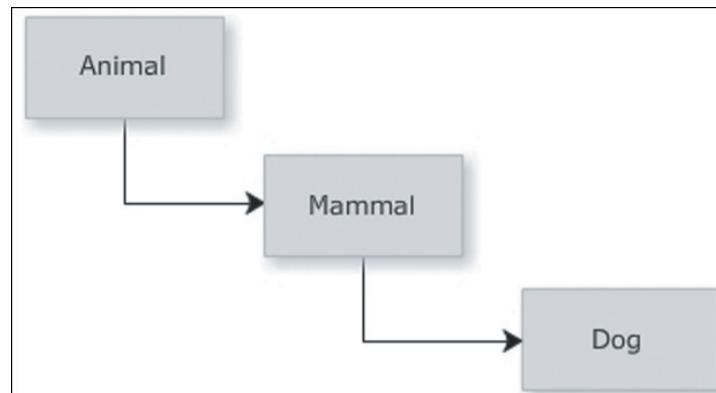


Figure 7.3: Multi-level Hierarchy

Code Snippet 1 demonstrates multiple levels of inheritance.

Code Snippet 1:

```

using System;

class Animal
{
    public void Eat()
    {
        Console.WriteLine("Every animal eats something.");
    }
}

class Mammal : Animal
{
    public void Feature()
    {
        Console.WriteLine("Mammals give birth to young ones.");
    }
}
  
```

```

class Dog : Mammal
{
    public void Noise()
    {
        Console.WriteLine("Dog Barks.");
    }

    static void Main(string[] args)
    {
        Dog objDog = new Dog();
        objDog.Eat();
        objDog.Feature();
        objDog.Noise();
    }
}

```

In Code Snippet 1, the `Main()` method of the class `Dog` invokes the methods of the class `Animal`, `Mammal`, and `Dog`.

Output:

Every animal eats something.

Mammals give birth to young ones.

Dog Barks.

7.1.4 Implementing Inheritance

The syntax to derive a class from another class is quite simple in C#. You just need to insert a colon after the name of the derived class followed by the name of the base class. The derived class can now inherit all non-private methods and attributes of the base class.

The following syntax is used to inherit a class in C#.

Syntax:

<DerivedClassName> : <BaseClassName>

where,

DerivedClassName: Is the name of the newly created child class.

BaseClassName: Is the name of the parent class from which the current class is inherited.

The following syntax is used to invoke a method of the base class.

Syntax:

<objectName>. <MethodName>;

where,

objectName: Is the object of the base class.

MethodName: Is the name of the method of the base class.

Code Snippet 2 demonstrates how to derive a class from another existing class and inherit methods from the base class.

Code Snippet 2:

```
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Every animal eats something.");
    }

    public void DoSomething()
    {
        Console.WriteLine("Every animal does something.");
    }
}

class Cat : Animal
{
    static void Main(String[] args)
    {
        Cat objCat = new Cat();
        objCat.Eat();
        objCat.DoSomething();
    }
}
```

In Code Snippet 2, the class **Animal** consists of two methods, **Eat()** and **DoSomething()**. The class **Cat** is inherited from the class **Animal**. The instance of the class **Cat** is created and it invokes the two methods defined in the class **Animal**. Even though an instance of the derived class is created, it is the

methods of the base class that are invoked because these methods are not implemented again in the derived class. When the instance of the class `Cat` invokes the `Eat()` and `DoSomething()` methods, the statements in the `Eat()` and `DoSomething()` methods of the base class `Animal` are executed.

Output:

Every animal eats something.

Every animal does something.

7.1.5 protected Access Modifier

The `protected` access modifier protects the data members that are declared using this modifier. The `protected` access modifier is specified using the `protected` keyword. Variables or methods that are declared as `protected` are accessed only by the class in which they are declared or by a class that is derived from this class. Figure 7.4 displays an example of using the `protected` access modifier.

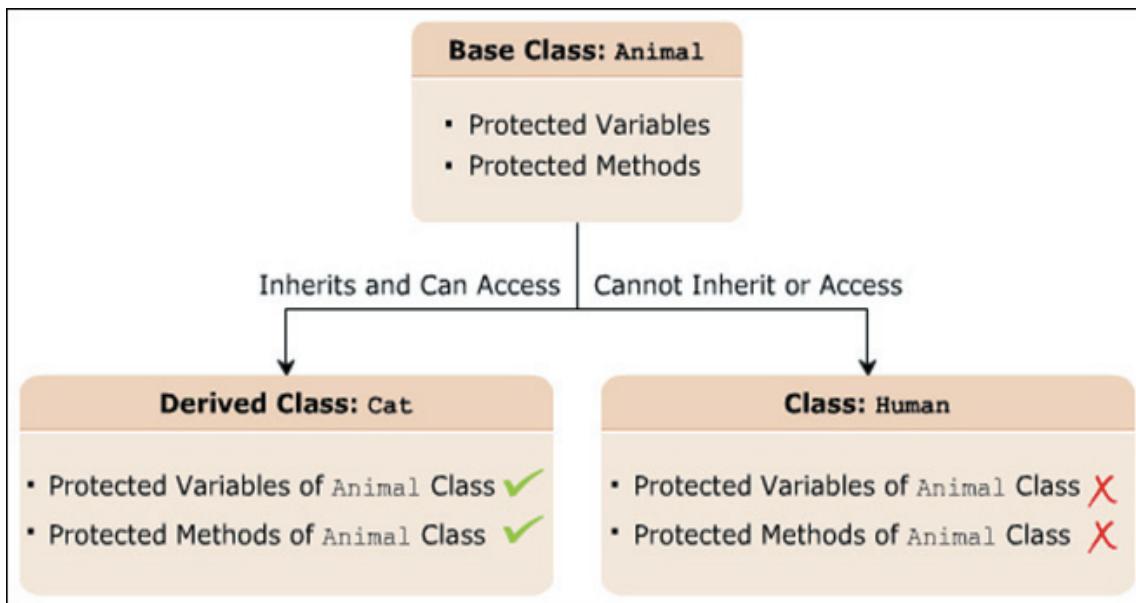


Figure 7.4: protected Access Modifier

The following syntax declares a `protected` variable.

Syntax:

```
protected <data_type> <VariableName>;
```

where,

`data_type`: Is the data type of the data member.

`VariableName`: Is the name of the variable.

The following syntax declares a protected method.

Syntax:

```
protected<return_type><MethodName>(argument_list);
```

where,

return_type: Is the type of value a method will return.

MethodName: Is the name of the method.

argument_list: Is the list of parameters.

Code Snippet 3 demonstrates the use of the `protected` access modifier.

Code Snippet 3:

```
class Animal
{
    protected string Food;
    protected string Activity;
}

class Cat : Animal
{
    static void Main(String[] args)
    {
        Cat objCat = new Cat();
        objCat.Food = "Mouse";
        objCat.Activity = "laze around";
        Console.WriteLine("The Cat loves to eat " + objCat.Food + ".");
        Console.WriteLine("The Cat loves to " + objCat.Activity + ".");
    }
}
```

In Code Snippet 3, two variables are created in the class `Animal` with the `protected` keyword. The class `Cat` is inherited from the class `Animal`. The instance of the class `Cat` is created that is referring the two variables defined in the class `Animal` using the dot (.) operator. The `protected` access modifier allows the variables declared in the class `Animal` to be accessed by the derived class `Cat`.

Output:

The Cat loves to eat Mouse.

The Cat loves to laze around.

7.1.6 base Keyword

The `base` keyword allows you to access the variables and methods of the base class from the derived class. When you inherit a class, the methods and variables defined in the base class can be re-declared in the derived class. Now, when you invoke methods or access variables, the derived class data members are invoked and not data members of the base class. In such situations, you can access the base class members using the `base` keyword.

You cannot use the `base` keyword for invoking the static methods of the base class.

The following syntax is used to specify the `base` keyword.

Syntax:

```
class <ClassName>
{
<access modifier><returntype><BaseMethod> { }
}

class <ClassName1> : <ClassName>
{
base.<BaseMethod>;
}
```

where,

`<ClassName>`: Is the name of the base class.

`<access modifier>`: Specifies the scope of the class or method.

`<returntype>`: Specifies the type of data the method will return.

`<BaseMethod>`: Is the base class method.

`<ClassName1>`: Is the name of the derived class.

`base`: Is a keyword used to access the base class members.

Figure 7.5 displays an example of using the `base` keyword.

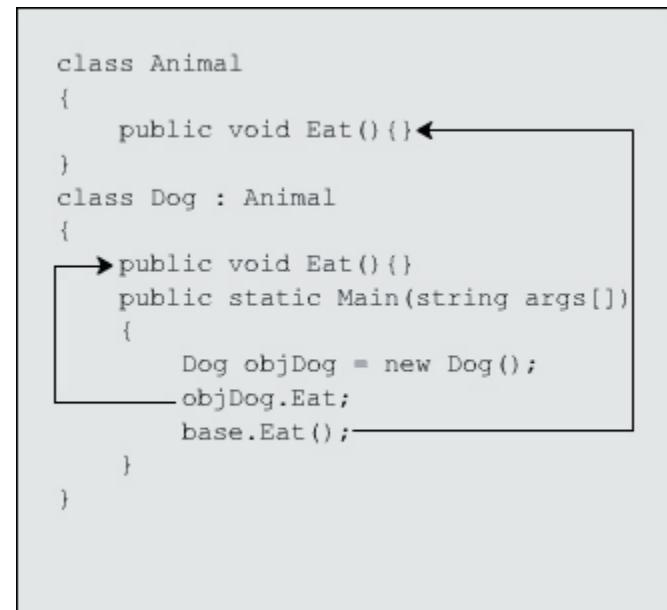


Figure 7.5: base Keyword

7.1.7 new Keyword

The `new` keyword can either be used as an operator or as a modifier in C#. The `new` operator is used to instantiate a class by creating its object. This instantiation finally invokes the constructor of the class. As a modifier, the `new` keyword is used to hide the methods or variables of the base class that are inherited in the derived class. This allows you to redefine the inherited methods or variables in the derived class. Since redefining the base class members in the derived class results in base class members being hidden, the only way you can access these is by using the `base` keyword.

The following syntax shows the use of the `new` modifier.

Syntax:

```

<access modifier> class <ClassName>
{
    <access modifier> <returntype> <BaseMethod> { }

    <access modifier> class <ClassName1> : <ClassName>
    {
        new <access modifier> void <BaseMethod> { }
    }
}

```

where,

- <access modifier>: Specifies the scope of the class or method.
- <return type>: Specifies the type of data the method will return.
- <ClassName>: Is the name of the base class.
- <ClassName1>: Is the name of the derived class.
- new**: Is a keyword used to hide the base class method.

Code Snippet 4 creates an object using the new operator.

Code Snippet 4:

```
Employees objEmp = new Employees();
```

Here, the code creates an instance called **objEmp** of the class **Employees** and invokes its constructor.

Code Snippet 5 demonstrates the use of the new modifier to redefine the inherited methods in the base class.

Code Snippet 5:

```
class Employees
{
    int _empId=1;
    string _empName = "James Anderson";
    int _age=25;
    public void Display()
    {
        Console.WriteLine("Employee ID: " + _empId);
        Console.WriteLine("Employee Name: " + _empName);
    }
}

class Department : Employees
{
    int _deptId=501;
    string _deptName = "Sales";
    new void Display()
    {
        base.Display();
    }
}
```

```

Console.WriteLine("Department ID: " + _deptId);
Console.WriteLine("Department Name: " + _deptName);
}

static void Main(string [] args)
{
    Department objDepartment = new Department ();
    objDepartment.Display();
}
}

```

In Code Snippet 5, the class **Employees** declares a method called **Display()**. This method is inherited in the derived class **Department** and is preceded by the **new** keyword. The new keyword hides the inherited method **Display()** that was defined in the base class, thereby executing the **Display()** method of the derived class when a call is made to it. However, the base keyword allows you to access the base class members. Therefore, the statements in the **Display()** method of the derived class and the base class are executed, and, finally, the employee ID, employee name, department ID and department name are displayed in the console window.

Output:

```

Employee ID: 1
Employee Name: James Anderson
Department ID: 501
Department Name: Sales

```

7.1.8 Constructor Inheritance

In C#, you cannot inherit constructors similar to how you inherit methods. However, you can invoke the base class constructor by either instantiating the derived class or the base class. The instance of the derived class will always first invoke the constructor of the base class followed by the constructor of the derived class. In addition, you can explicitly invoke the base class constructor by using the **base** keyword in the derived class constructor declaration. The **base** keyword allows you to pass parameters to the constructor.

Figure 7.6 displays an example of constructor inheritance.

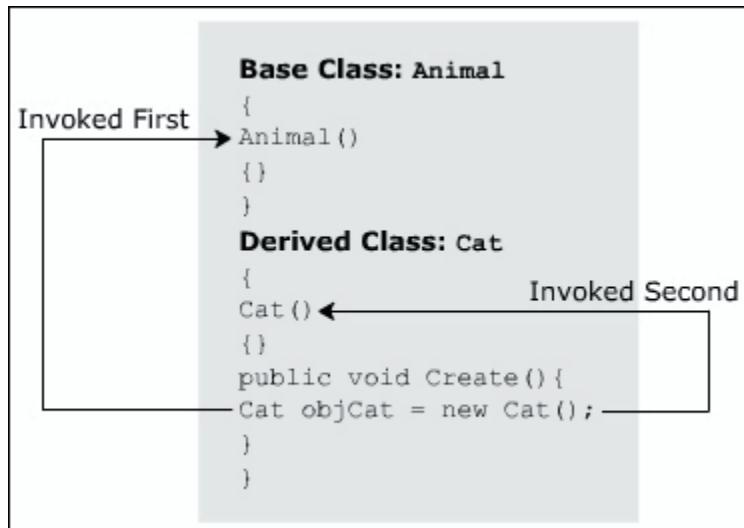


Figure 7.6: Constructor Inheritance

Code Snippet 6 explicitly invokes the base class constructor using the `base` keyword.

Code Snippet 6:

```

class Animal
{
    public Animal()
    {
        Console.WriteLine("Animal constructor without parameters");
    }

    public Animal(String name)
    {
        Console.WriteLine("Animal constructor with a string parameter");
    }
}

class Canine : Animal
{
    //base() takes a string value called "Lion"
    public Canine() : base("Lion")
    {

        Console.WriteLine("Derived Canine");
    }
}
  
```

```

    }
}

class Details
{
    static void Main(String[] args)
    {
        Canine objCanine = new Canine();
    }
}

```

In Code Snippet 6, the class **Animal** consists of two constructors, one without a parameter and the other with a **string** parameter. The class **Canine** is inherited from the class **Animal**. The derived class **Canine** consists of a constructor that invokes the constructor of the base class **Animal** by using the **base** keyword. If the **base** keyword does not take a string in the parenthesis, the constructor of the class **Animal** that does not contain parameters is invoked. In the class **Details**, when the derived class constructor is invoked, it will in turn invoke the parameterized constructor of the base class.

Output:

Animal constructor with a string parameter

Derived Canine

7.1.9 Invoking Parameterized Base Class Constructors

The derived class constructor can explicitly invoke the base class constructor by using the **base** keyword. If a base class constructor has a parameter, the **base** keyword is followed by the value of the type specified in the constructor declaration. If there are no parameters, the **base** keyword is followed by a pair of parentheses. Code Snippet 7 demonstrates how parameterized constructors are invoked in a multi-level hierarchy.

Code Snippet 7:

```

using System;
class Metals
{
    string _metalType;
    public Metals(string type)
    {
        _metalType = type;
    }
}

```

```

        Console.WriteLine("Metal: \t\t" + _metalType);
    }

}

class SteelCompany : Metals
{
    string _grade;

    public SteelCompany(string grade) : base("Steel")
    {
        _grade = grade;
        Console.WriteLine("Grade: \t\t" + _grade);
    }
}

class Automobiles : SteelCompany
{
    string _part;

    public Automobiles(string part) : base("Cast Iron")
    {
        _part = part;
        Console.WriteLine("Part: \t\t" + _part);
    }

    static void Main(string[] args)
    {
        Automobiles objAutomobiles = new Automobiles("Chassies");
    }
}

```

In Code Snippet 7, the **Automobiles** class inherits the **SteelCompany** class. The **SteelCompany** class inherits the **Metals** class. In the **Main()** method, when an instance of the **Automobiles** class is created, it invokes the constructor of the **Metals** class, followed by the constructor of the **SteelCompany** class. Finally, the constructor of the **Automobiles** class is invoked.

Output:

Metal: Steel

Grade: Cast Iron

Part: Classes

7.2 Method Overriding

Method overriding is a feature that allows the derived class to override or redefine the methods of the base class. Overriding a method in the derived class can change the body of the method that was declared in the base class. Thus, the same method with the same name and signature declared in the base class can be reused in the derived class to define a new behavior. This is how reusability is ensured while inheriting classes.

The method implemented in the derived class from the base class is known as the **Overridden Base Method**. Figure 7.7 displays the method overriding.

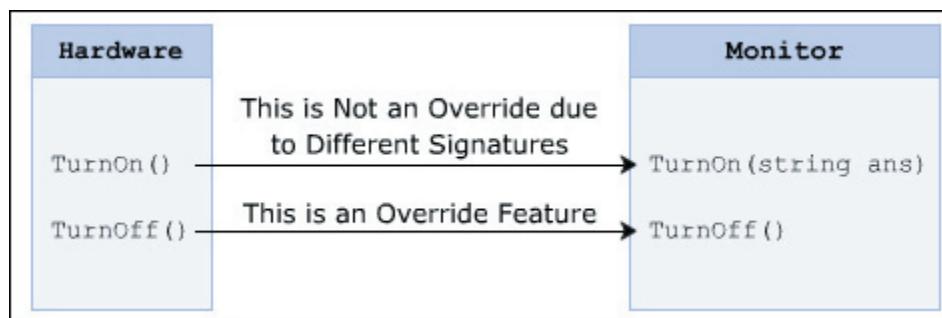


Figure 7.7: Method Overriding

Note - While overriding a base class method, you should consider the accessibility scope of the method. This means that the base class method with less accessibility scope cannot be overridden in the derived class. For example, a private method in the base class cannot be overridden as public in the derived class.

7.2.1 virtual and override Keywords

You can override a base class method in the derived class using appropriate C# keywords such as `virtual` and `override`. If you want to override a particular method of the base class in the derived class, you need to declare the method in the base class using the `virtual` keyword. A method declared using the `virtual` keyword is referred to as a `virtual` method.

In the derived class, you need to declare the inherited `virtual` method using the `override` keyword. This is mandatory for any `virtual` method that is inherited in the derived class. The `override` keyword overrides the base class method in the derived class.

The following is the syntax for declaring a `virtual` method using the `virtual` keyword.

Syntax:

```
<access_modifier> virtual <return_type> <MethodName>(<parameter-list>);
```

where,

`access_modifier`: Is the access modifier of the method, which can be `private`, `public`,

`protected` or `internal`.

`virtual`: Is a keyword used to declare a method in the base class that can be overridden by the derived class.

`return_type`: Is the type of value the method will return.

`MethodName`: Is the name of the virtual method.

`parameter-list`: Is the parameter list of the method; it is optional.

The following is the syntax for overriding a method using the `override` keyword.

Syntax:

```
<access modifier> override <return type> <MethodName> (<parameters-list>)
```

where,

`override`: Is the keyword used to override a method in the derived class.

Code Snippet 8 demonstrates the application of the `virtual` and `override` keywords in the base and derived classes respectively.

Code Snippet 8:

```
class Animal
{
    public virtual void Eat()
    {
        Console.WriteLine("Every animal eats something");
    }

    protected void DoSomething()
    {
        Console.WriteLine("Every animal does something");
    }
}

class Cat : Animal
{
    //Class Cat overrides Eat() method of class Animal
    public override void Eat()
    {
        Console.WriteLine("Cat loves to eat the mouse");
    }
}
```

```

    }

    static void Main(String[] args)
    {
        Cat objCat = new Cat();
        objCat.Eat();
    }
}

```

In Code Snippet 8, the class **Animal** consists of two methods, the **Eat()** method with the **virtual** keyword and the **DoSomething()** method with the **protected** keyword. The class **Cat** is inherited from the class **Animal**. An instance of the class **Cat** is created and the dot (.) operator is used to invoke the **Eat()** and the **DoSomething()** methods. The virtual method **Eat()** is overridden in the derived class using the **override** keyword. This enables the C# compiler to execute the code within the **Eat()** method of the derived class.

Output:

Cat loves to eat the mouse

Note - If the derived class tries to override a non-virtual method, the C# compiler generates an error. If you fail to override the virtual methods, the C# compiler generates a compile-time warning. However, in this case, the code will run successfully.

You cannot use the keywords **new**, **static** and **virtual** with the **override** keyword.

7.2.2 Calling the Base Class Method

Method overriding allows the derived class to redefine the methods of the base class. Redefining the base class methods allows you to access the new method but not the original base class method. Sometimes, you might want both, the base class method as well as the derived class method, to be executed. In this case, you can create an instance of the base class, which allows you to access the base class method, and an instance of the derived class, to access the derived class method.

Code Snippet 9 demonstrates how to access a base class method.

Code Snippet 9:

```

class Student
{
    string _studentName = "James";
    string _address = "California";
    public virtual void PrintDetails()
    {
    }
}

```

```

Console.WriteLine("Student Name: " + _studentName);
Console.WriteLine("Address: " + _address);
}

}

class Grade : Student
{
    string _class = "Four";
    float _percent = 71.25F;
    public override void PrintDetails()
    {
        Console.WriteLine("Class: " + _class);
        Console.WriteLine("Percentage: " + _percent);
    }
    static void Main(string[] args)
    {
        Student objStudent = new Student();
        Grade objGrade = new Grade();
        objStudent.PrintDetails();
        objGrade.PrintDetails();
    }
}

```

In Code Snippet 9, the class **Student** consists of a virtual method called **PrintDetails()**. The class **Grade** inherits the class **Student** and overrides the base class method **PrintDetails()**. The **Main()** method creates an instance of the base class **Student** and the derived class **Grade**. The instance of the base class **Student** uses the dot (.) operator to invoke the base class method **PrintDetails()**. The instance of the derived class **Grade** uses the dot (.) operator to invoke the derived class method **PrintDetails()**.

Output:

Student Name: James

Address: California

Class: Four

Percentage: 71.25

7.3 Sealed Classes

A sealed class is a class that prevents inheritance. You can declare a sealed class by preceding the class keyword with the sealed keyword. The sealed keyword prevents a class from being inherited by any other class. Therefore, the sealed class cannot be a base class as it cannot be inherited by any other class. If a class tries to derive a sealed class, the C# compiler generates an error.

The following syntax is used to declare a sealed class.

Syntax:

```
sealed class <ClassName>
{
    //body of the class
}
```

where,

sealed: Is a keyword used to prevent a class from being inherited.

ClassName: Is the name of the class that needs to be sealed.

Code Snippet 10 demonstrates the use of a sealed class in C#. This code will generate a compiler error.

Code Snippet 10:

```
sealed class Product
{
    public int Quantity;
    public int Cost;
}

class Goods
{
    static void Main(string [] args)
    {
        Product objProduct = new Product();
        objProduct.Quantity = 50;
        objProduct.Cost = 75;
        Console.WriteLine("Quantity of the Product: " + objProduct.Quantity);
    }
}
```

```

        Console.WriteLine("Cost of the Product: " + objProduct.Cost);
    }

}

class Pen : Product
{
}

```

In Code Snippet 10, the class **Product** is declared as `sealed` and it consists of two variables. The class **Goods** contains the code to create an instance of **Product** and uses the dot (.) operator to invoke variables declared in **Product**. However, when the class **Pen** tries to inherit the sealed class **Product**, the C# compiler generates an error, as shown in figure 7.8.

The screenshot shows a code editor with the following C# code:

```

sealed class Product
{
    public int Quantity;
    public int Cost;
}
class Goods
{
    static void Main(string[] args)
    {
        Product objProduct = new Product();
        objProduct.Quantity = 50;
        objProduct.Cost = 75;
        Console.WriteLine("Quantity of the Product: " + objProduct.
        Quantity);
        Console.WriteLine("Cost of the Product: " + objProduct.Cost);
    }
}
class Pen : Product
{
}

```

Below the code editor is the Error List window, which displays one error:

Description	File	Line	Column	Project
ConsoleApplication1.Pen': cannot derive from sealed type 'ConsoleApplication1.Product'	Goods.cs	26	11	ConsoleApplication1

Figure 7.8: Compiler Error

Note - A sealed class cannot have any protected members. If you attempt to declare protected members in a sealed class, the C# compiler generates a warning because protected members are accessible only by the derived classes. A sealed class does not have derived classes as it cannot be inherited.

7.3.1 Purpose of Sealed Classes

Consider a class named **SystemInformation** that consists of critical methods that affect the working of the operating system. You might not want any third party to inherit the class **SystemInformation** and override its methods, thus, causing security and copyright issues. Here, you can declare the **SystemInformation** class as sealed to prevent any change in its variables and methods.

7.3.2 Guidelines

Sealed classes are restricted classes that cannot be inherited. The list depicts the conditions in which a class can be marked as sealed.

- If overriding the methods of a class might result in unexpected functioning of the class
- When you want to prevent any third party from modifying your class

7.3.3 Sealed Methods

A sealed class cannot be inherited by any other class. In C#, a method cannot be declared as sealed. However, when the derived class overrides a base class method, variable, property or event, then the new method, variable, property, or event can be declared as sealed. Sealing the new method prevents the method from further overriding. An overridden method can be sealed by preceding the `override` keyword with the `sealed` keyword.

The following syntax is used to declare an overridden method as sealed.

Syntax:

```
sealed override <return_type> <MethodName>{ }
```

where,

`return_type`: Specifies the data type of value returned by the method.

`MethodName`: Specifies the name of the overridden method.

Code Snippet 11 declares an overridden method `Print()` as sealed.

Code Snippet 11:

```
using System;
class ITSystem
{
    public virtual void Print()
    {
        Console.WriteLine ("The system should be handled carefully");
    }
}
```

```

}

}

class CompanySystem : ITSystem
{
    public override sealed void Print()
    {
        Console.WriteLine ("The system information is
        confidential");
        Console.WriteLine ("This information should not be
        overridden");
    }
}

class SealedSystem : CompanySystem
{
    public override void Print()
    {
        Console.WriteLine ("This statement won't get
        executed");
    }

    static void Main (string [] args)
    {
        SealedSystem objSealed= new SealedSystem();
        objSealed.Print ();
    }
}

```

In Code Snippet 11, the class **ITSystem** consists of a virtual function **Print()**.

The class **CompanySystem** is inherited from the class **ITSystem**. It overrides the base class method **Print()**. The overridden method **Print()** is sealed by using the sealed keyword, which prevents further overriding of that method. The class **SealedSystem** is inherited from the class **CompanySystem**.

When the class **SealedSystem** overrides the sealed method **Print()**, the C# compiler generates an error as shown in figure 7.9.

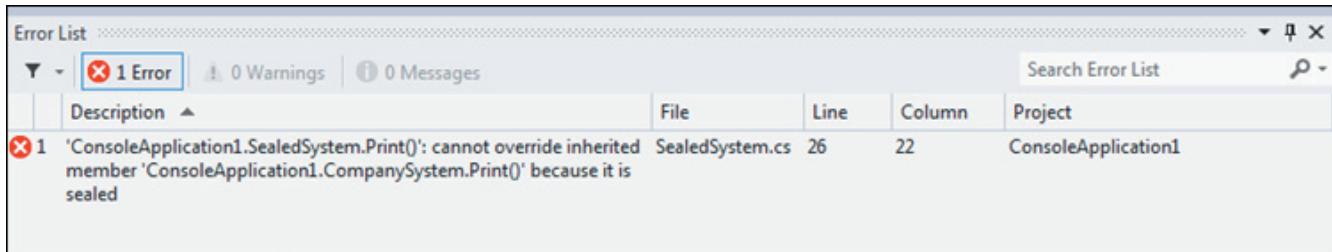


Figure 7.9: Error

7.4 Polymorphism

Polymorphism is derived from two Greek words, namely **Poly** and **Morphos**. Poly means many and Morphos means forms. Polymorphism means existing in multiple forms. Polymorphism is the ability of an entity to behave differently in different situations. Consider the following two methods in a class having the same name but different signatures performing the same basic operation but in different ways:

- **Area(float radius)**
- **Area(float base, float height)**

The two methods calculate the area of the circle and triangle taking different parameters and using different formulae. This is an example of polymorphism in C#.

Polymorphism allows methods to function differently based on the parameters and their data types.

Figure 7.10 displays the polymorphism.

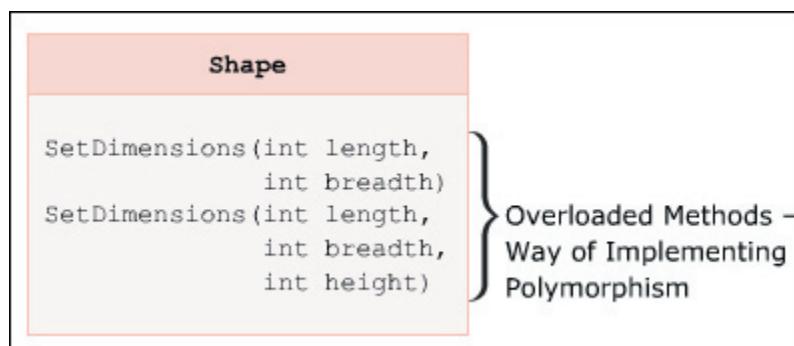


Figure 7.10: Polymorphism

7.4.1 Implementation

You can implement polymorphism in C# through method overloading and method overriding. You can create multiple methods with the same name in a class or in different classes having different method body or different signatures. Methods having the same name but different signatures in a class are

referred to as overloaded methods. Here, the same method performs the same function on different values.

Methods inherited from the base class in the derived class and modified within the derived class are referred to as overridden methods. Here, only the body of the method changes in order to function according to the required output.

Figure 7.11 displays the implementation.

- Method Overriding

```
class Hardware
{
    public virtual bool TurnOn() { return true; }
}
class Monitor:Hardware
{
    public override bool TurnOn() { return true; }
}
```

- Method Overloading

```
class Hardware
{
    public bool TurnOn() { return true; }
    public bool TurnOn(string ans) { return true; }
}
```

Figure 7.11: Implementation

Code Snippet 12 demonstrates the use of method overloading feature.

Code Snippet 12:

```
class Area
{
    static int CalculateArea(int len, int wide)
    {
        return len * wide;
    }
    static double CalculateArea(double valOne, double valTwo)
    {
        return 0.5 * valOne * valTwo;
    }
}
```

```

static void Main(string[] args)
{
    int length = 10;
    int breadth = 22;
    double tbase = 2.5;
    double theight = 1.5;
    Console.WriteLine("Area of Rectangle: " + CalculateArea(length, breadth));
    Console.WriteLine("Area of triangle: " + CalculateArea(tbase, theight));
}
}

```

In Code Snippet 12, the class **Area** consists of two static methods of the same name, **CalculateArea**. However, both these methods have different return types and take different parameters.

Output:

Area of Rectangle: 220

Area of triangle: 1.875

7.4.2 Compile-time and Run-time Polymorphism

Polymorphism can be broadly classified into two categories, compile-time polymorphism and run-time polymorphism. Table 7.1 differentiates between compile-time and run-time polymorphisms.

Compile-time Polymorphism	Run-time Polymorphism
Is implemented through method overloading .	Is implemented through method overriding .
Is executed at the compile-time since the compiler knows which method to execute depending on the number of parameters and their data types.	Is executed at run-time since the compiler does not know the method to be executed, whether it is the base class method that will be called or the derived class method.
Is referred to as static polymorphism.	Is referred to as dynamic polymorphism.

Table 7.1: Difference between Compile-time and Run-time Polymorphism

Code Snippet 13 demonstrates the implementation of run-time polymorphism.

Code Snippet 13:

```
using System;
class Circle
{
    protected const double PI = 3.14;
    protected double Radius = 14.9;
    public virtual double Area()
    {
        return PI * Radius * Radius;
    }
}
class Cone : Circle
{
    protected double Side = 10.2;
    public override double Area()
    {
        return PI * Radius * Side;
    }
    static void Main(string[] args)
    {
        Circle objRunOne = new Circle();
        Console.WriteLine("Area is: " + objRunOne.Area());
        Circle objRunTwo = new Cone();
        Console.WriteLine("Area is: " + objRunTwo.Area());
    }
}
```

In Code Snippet 13, the class **Circle** initializes protected variables and contains a virtual method **Area()** that returns the area of the circle. The class **Cone** is derived from the class **Circle**, which overrides the method **Area()**. The **Area()** method returns the area of the cone by considering the length of the cone, which is initialized to the value 10.2. The **Main()** method demonstrates how polymorphism can take place by first creating an object of type **Circle** and invoking its **Area()** method and later creating a reference of type **Circle** but instantiating it to **Cone** at run-time and then calling the **Area()** method.

In this case, the **Area()** method of **Cone** will be called even though the reference created was that of **Circle**.

Output:

```
Area is: 697.1114
```

```
Area is: 477.2172
```

7.5 Check Your Progress

1. Can you match the terms, keywords and modifiers related to inheritance against their corresponding descriptions?

Descriptions		Terms	
(A)	Is used to invoke methods.	(1)	new operator
(B)	Is used to create new objects	(2)	new modifier
(C)	Is used to access base class members only by the derived class.	(3)	dot operator
(D)	Is used to execute the inherited methods of the derived class.	(4)	base keyword
(E)	Is used to access the constructor of the base class.	(5)	protected keyword

(A)	(A)-(4), (B)-(3), (C)-(1), (D)-(2), (E)-(1)	(C)	(A)-(2), (B)-(4), (C)-(1), (D)-(2), (E)-(3)
(B)	(A)-(3), (B)-(1), (C)-(5), (D)-(2), (E)-(4)	(D)	(A)-(1), (B)-(2), (C)-(2), (D)-(4), (E)-(3)

2. You are trying to display the name, id, address, and age of a student as “John”, “10”, “Los Angeles, California”, and “12”. Which of the following codes will help you achieve this?

(A)	<pre> class Student { string _studentName = "John"; int _studentId = 10; void Display() { Console.WriteLine("Student Name: " + _studentName); Console.WriteLine("Student ID: " + _studentId); } } class Details : Student { string _address = "Los Angeles, California"; int _age = 12; } </pre>
-----	---

(A)	<pre>newpublic void Display() { base.Display(); Console.WriteLine("Address: " + _address); Console.WriteLine("Age: " + _age); } static void Main(string[] args) { Details objDetails = new Details(); objDetails.Display(); }</pre>
(B)	<pre>class Student { string _studentName = "John"; int _studentId = 10; public void Display() { Console.WriteLine("Student Name: " + _studentName); Console.WriteLine("Student ID: " + _studentId); } } class Details : Student { string _address = "Los Angeles, California"; int _age = 12; newpublic void Display() { base.Display(); Console.WriteLine("Address: " + _address); Console.WriteLine("Age: " + _age); } }</pre>

(B)	<pre> } static void Main(string[] args) { Details objDetails = new Details(); objDetails.Display(); } } </pre>
(C)	<pre> class Student { string _studentName = "John"; int _studentId = 10; public void Print() { Console.WriteLine("Student Name: " + _studentName); Console.WriteLine("Student ID: " + _studentId); } } class Details : Student { string _address = "Los Angeles, California"; int _age = 12; new public void Display() { base.Display(); Console.WriteLine("Address: " + _address); Console.WriteLine("Age: " + _age); } } static void Main(string[] args) { Details objDetails = new Details(); objDetails.Display(); } </pre>

(C)	}
(D)	<pre> class Student { static string _studentName = "John"; static int _studentId = 10; static void Display() { Console.WriteLine("Student Name: " + _studentName); Console.WriteLine("Student ID: " + _studentId); } } class Details : Student { string _address = "Los Angeles, California"; int _age = 12; new public static void Display() { base.Display(); Console.WriteLine("Address: " + _address); Console.WriteLine("Age: " + _age); } static void Main(string[] args) { Details obj = new Details(); obj.Display(); } } </pre>

(A)	A	(C)	C
(B)	B	(D)	D

3. Which of these statements about method overriding are true?

(A)	A method is overridden with the same name and signature as declared in the base class.
(B)	A method that is overriding the method defined in the base class is preceded by the virtual keyword.
(C)	A base class method is preceded with the override keyword in order to override it in the derived class.
(D)	A method is known as overridden derived method when it is overridden in the derived class.
(E)	A method is overridden by invoking it in the derived class.

(A)	A	(C)	B, C, D
(B)	A, C	(D)	B, E

4. You are trying to display the name, ID, designation, and salary of an employee. Which of the following codes will help you to achieve this?

(A)	<pre>class Employee { string _empName = "James Ambrose"; int _empId = 101; protected virtual void Print() { Console.WriteLine("Employee Name: " + _empName); Console.WriteLine("Employee ID: " + _empId); } } class Salary : Employee { double _salary = 1005.60; string _designation = "Marketing"; protected override void Print() {</pre>
-----	---

(A)	<pre> base.Print(); Console.WriteLine("Designation: " + _designation); Console.WriteLine("Salary Details: " + _salary); } static void Main (string [] args) { Salary objSalary = new Salary (); objSalary.Print(); } </pre>
(B)	<pre> class Employee { string _empName = "James Ambrose"; int _empId = 101; private virtual void Print() { Console.WriteLine("Employee Name: " + _empName); Console.WriteLine("Employee ID: " + _empId); } } class Salary : Employee { double _salary = 1005.60; string _designation = "Marketing"; protected override void Print() { base.Print(); Console.WriteLine("Designation: " + _designation); Console.WriteLine("Salary Details: " + _salary); } static void Main (string [] args) </pre>

(B)	<pre>{ Salary objSalary = new Salary(); objSalary.Print(); }</pre>
(C)	<pre>class Employee { string _empName = "James Ambrose"; int _empId = 101; protected virtual void Print() { Console.WriteLine("Employee Name: " + _empName); Console.WriteLine("Employee ID: " + _empId); } } class Salary : Employee { double _salary = 1005.60; string _designation = "Marketing"; protected override void Print() { base.Print(); Console.WriteLine("Designation: " + _designation); Console.WriteLine("Salary Details: " + _salary); } } static void Main(string[] args) { Salary objSalary = new Salary(); objSalary.Print(); }</pre>

(D)

```

class Employee
{
    string _empName = "James Ambrose";
    int _empId = 101;
    protected virtual void Print()
    {
        Console.WriteLine("Employee Name: " + _empName);
        Console.WriteLine("Employee ID: " + _empId);
    }
}

class Salary : Employee
{
    double _salary = 1005.60;
    string _designation = "Marketing";
    protected override void Print()
    {
        Print();
        Console.WriteLine("Designation: " + _designation);
        Console.WriteLine("Salary Details: " + _salary);
    }
    static void Main(string[] args)
    {
        Salary objSalary = new Salary();
        objSalary.Print();
    }
}

```

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of these statements about sealed classes are true?

(A)	The seal keyword is used to declare a class as sealed.
(B)	A sealed class is used to ensure security by preventing any modification to its existing methods.
(C)	A sealed class is also referred to as final class in C#.
(D)	A sealed class is a class whose data members cannot be modified.
(E)	A sealed class is used to support the functioning of the virtual keyword.

(A)	A, B, D	(C)	B, D
(B)	A, C	(D)	B, E

6. Which of these statements about polymorphism are true?

(A)	Polymorphism supports the existence of a single class in multiple forms.
(B)	Polymorphism implements multiple methods with different names but with the same signature.
(C)	Compile-time polymorphism allows the compiler to identify the methods to be executed.
(D)	Run-time polymorphism allows you to execute overridden methods.
(E)	Run-time polymorphism supports methods with different signatures.

(A)	A	(C)	C
(B)	B	(D)	D

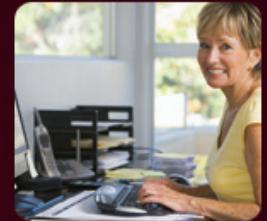
7.5.1 Answers

1.	B
2.	B
3.	A
4.	A
5.	C
6.	D



Summary

- Inheritance allows you to create a new class from another class, thereby inheriting its common properties and methods.
- Inheritance can be implemented by writing the derived class name followed by a colon and the name of the base class.
- Method overriding is a process of redefining the base class methods in the derived class.
- Methods can be overridden by using a combination of virtual and override keywords within the base and derived classes respectively.
- Sealed classes are classes that cannot be inherited by other classes.
- You can declare a sealed class in C# by using the sealed keyword.
- Polymorphism is the ability of an entity to exist in two forms that are compile-time polymorphism and run-time polymorphism.



Session -8

Abstract Classes and Interfaces

Welcome to the Session, **Abstract Classes and Interfaces**.

A class that is defined using the `abstract` keyword and contains at least one method without a body is referred to as an abstract class. However, it can contain other methods that are implemented in the class. An interface, like an abstract class, can declare abstract methods. Unlike an abstract class, an interface cannot implement any methods.

In this session, you will learn to:

- Define and describe abstract classes
- Explain interfaces
- Compare abstract classes and interfaces

8.1 Abstract Classes

C# allows designing a class specifically to be used as a base class by declaring it an abstract class. Such class can be referred to as an incomplete base class, as it cannot be instantiated, but it can only be implemented or derived.

An abstract class is a class declared using the `abstract` keyword and which may or may not contain one or more of the following: normal data member(s), normal method(s), and abstract method(s).

8.1.1 Purpose

Consider the base class, `Animal`, that defines methods such as `Eat()`, `Habitat()`, and `AnimalSound()`. The `Animal` class is inherited by different subclasses such as `Dog`, `Cat`, `Lion`, and `Tiger`. The dogs, cats, lions, and tigers neither share the same food nor habitat and nor do they make similar sounds. Hence, the `Eat()`, `Habitat()`, and `AnimalSound()` methods need to be different for different animals even though they inherit the same base class. These differences can be incorporated using abstract classes.

Figure 8.1 displays an example of abstract class and subclasses.

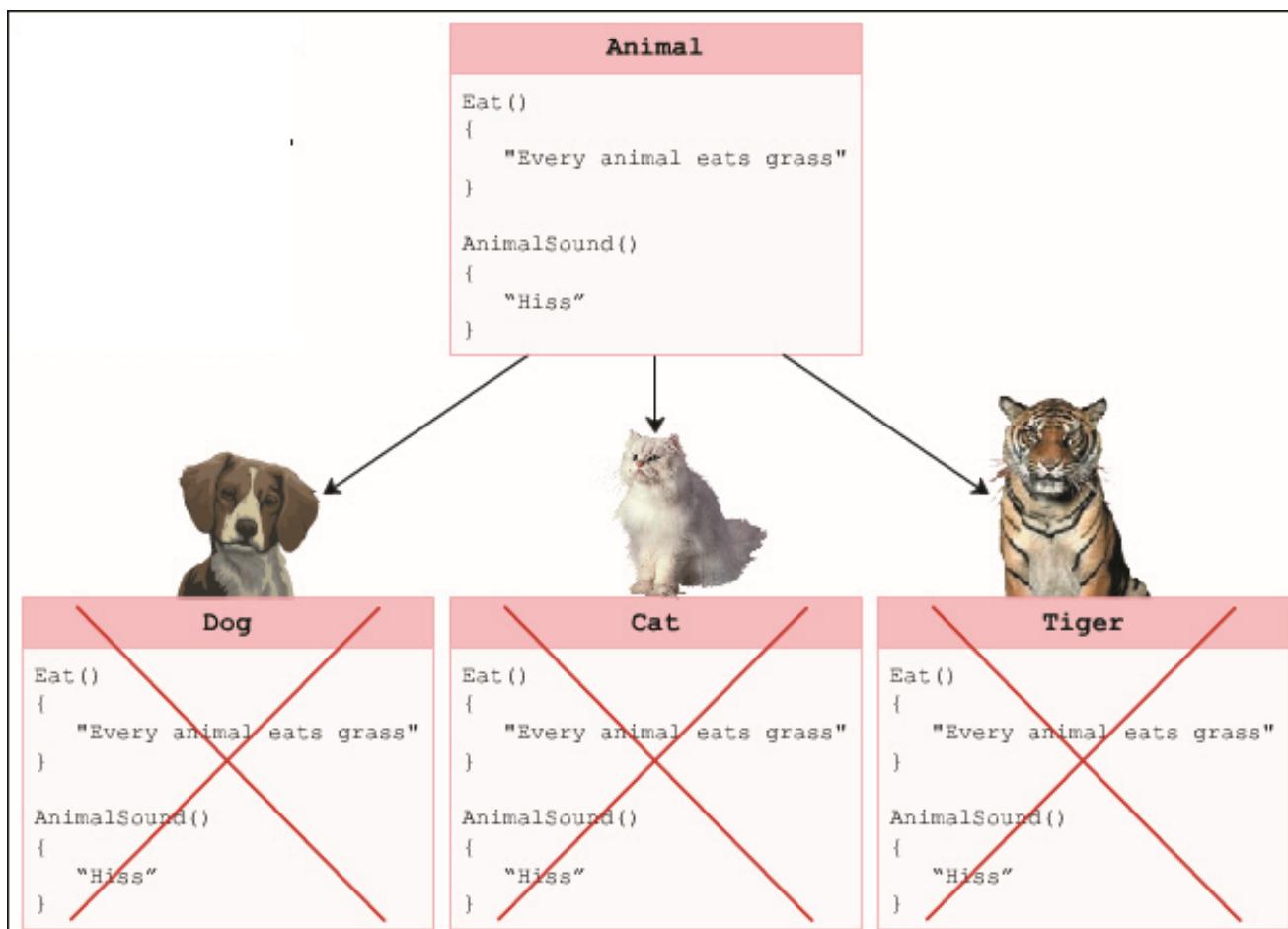


Figure 8.1: Abstract and SubClasses

8.1.2 Definition

An abstract class can implement methods that are similar for all the subclasses. Additionally, it can declare methods that are different for different subclasses without implementing them. Such methods are referred to as abstract methods.

A class that is defined using the `abstract` keyword and that contains at least one method which is not implemented in the class itself is referred to as an `abstract class`. In the absence of the `abstract` keyword, the class cannot be compiled. Since the abstract class contains at least one method without a body, the class cannot be instantiated using the `new` keyword.

Figure 8.2 displays the contents of an abstract class.

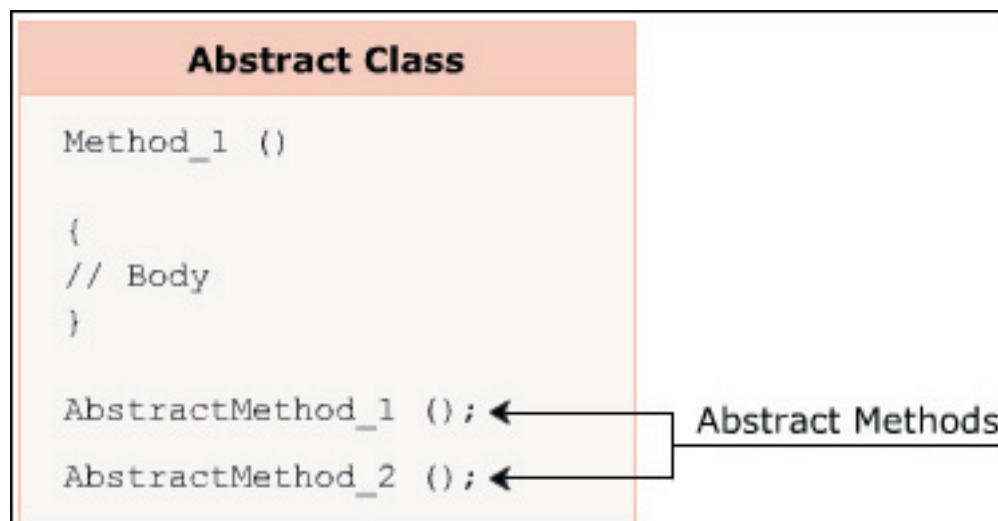


Figure 8.2: Abstract Class

The following syntax is used for declaring an abstract class.

Syntax:

```

public abstract class <ClassName>
{
<access_modifier> abstract <return_type> <MethodName>(<argument_list>);
}

```

where,

`abstract`: Specifies that the declared class is abstract.

`ClassName`: Specifies the name of the class.

Code Snippet 1 declares an abstract class **Animal**.

Code Snippet 1:

```
public abstract class Animal
{
    //Non-abstract method implementation public void Eat()
    {
        Console.WriteLine("Every animal eats food in order to survive");
    }

    //Abstract method declaration
    public abstract void AnimalSound();
    public abstract void Habitat();
}
```

In Code Snippet 1, the abstract class **Animal** is created using the `abstract` keyword. The **Animal** class implements the non-abstract method, `Eat()`, as well as declares two abstract methods, `AnimalSound()` and `Habitat()`.

Note - It is not mandatory for the abstract class to contain only abstract methods. It can contain non-abstract methods too. An abstract class cannot be sealed.

8.1.3 Implementation

An abstract class can be implemented in a way similar to implementing a normal base class. The subclass inheriting the abstract class has to override and implement the abstract methods. In addition, the subclass can implement the methods implemented in the abstract class with the same name and arguments.

If the subclass fails to implement the abstract methods, the subclass cannot be instantiated as the C# compiler considers it as abstract.

Figure 8.3 displays an example of inheriting an abstract class.

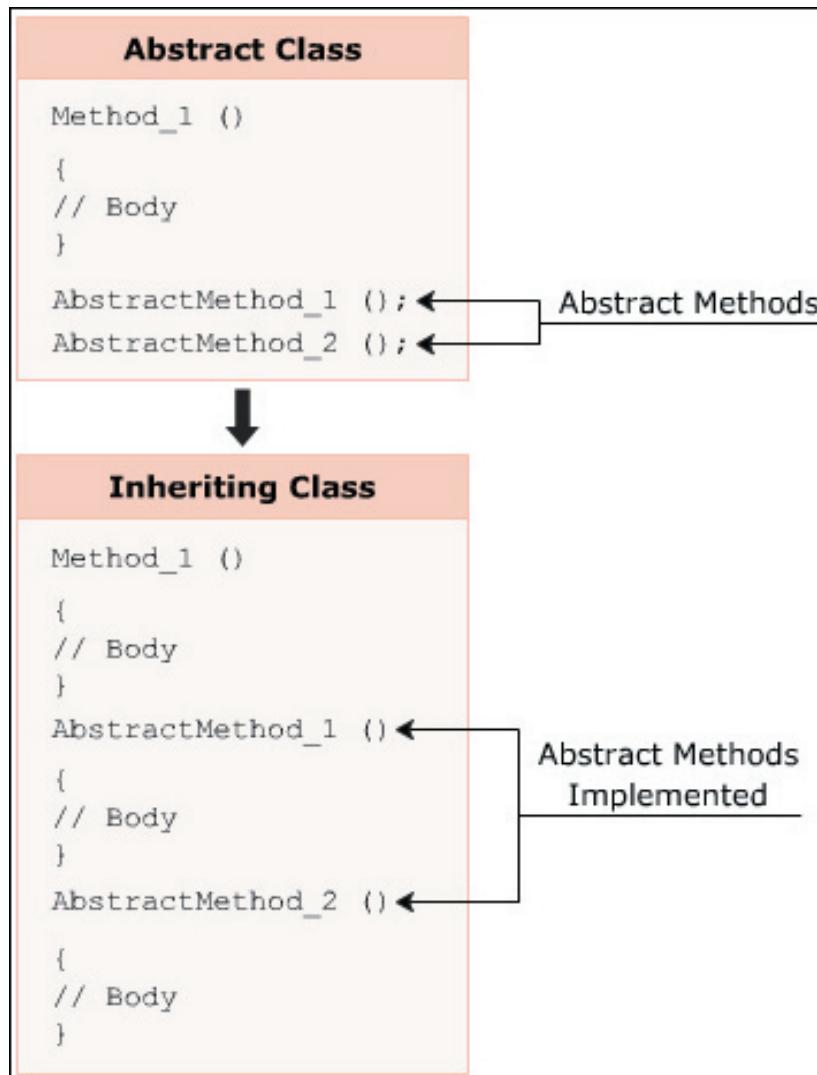


Figure 8.3: Inheriting an Abstract Class

The following syntax is used to implement an abstract class.

Syntax:

```

class <ClassName> : <AbstractClassName>
{
    // class members;
}
  
```

where,

`AbstractClassName`: Specifies the name of the inherited abstract class.

Code Snippet 2 declares and implements an abstract class.

Code Snippet 2:

```
abstract class Animal
{
    public void Eat()
    {
        Console.WriteLine("Every animal eats food in order to survive");
    }

    public abstract void AnimalSound();
}

class Lion : Animal
{
    public override void AnimalSound()
    {
        Console.WriteLine("Lion roars");
    }

    static void Main(string[] args)
    {
        Lion objLion = new Lion();
        objLion.AnimalSound();
        objLion.Eat();
    }
}
```

In Code Snippet 2, the abstract class **Animal** is declared, and the class **Lion** inherits the abstract class **Animal**. Since the **Animal** class declares an abstract method called **AnimalSound()**, the **Lion** class overrides the method **AnimalSound()** using the **override** keyword and implements it. The **Main()** method of the **Lion** class then invokes the methods **AnimalSound()** and **Eat()** using the **dot(.)** operator.

Output:

Lion roars

Every animal eats food in order to survive

8.1.4 Implement Abstract Base Class Using IntelliSense

IntelliSense provides access to member variables, functions and methods of an object or a class. Thus, it helps the programmer to easily develop the software by reducing the amount of input typed in, since IntelliSense performs the required typing. IntelliSense can be used to implement system-defined abstract classes.

The steps performed to implement an abstract class using IntelliSense are as follows:

1. Place the cursor after the `class IntelliSenseDemo` statement.
2. Type `: TimeZone`. Now the class declaration becomes `class IntelliSenseDemo : TimeZone`. (The `TimeZone` class is a system-defined class that represents the time zone where the standard time is being used.)
3. Click the smart tag that appears below the `TimeZone` class.
4. Click `Implement abstract class System.TimeZone`. IntelliSense provides four override methods from the system-defined `TimeZone` class to the user-defined `IntelliSenseDemo` class.

Code Snippet 3 demonstrates the way the methods of the abstract class `TimeZone` are invoked automatically by IntelliSense.

Code Snippet 3:

```
using System;

class IntelliSenseDemo : TimeZone
{
    public override string DaylightName
    {
        get { throw new Exception("The method or operation is not implemented."); }
    }

    public override System.Globalization.DaylightTime GetDaylightChanges
        (int year)
    {
        throw new Exception("The method or operation is not implemented.");
    }

    public override TimeSpan GetUtcOffset(DateTime time)
    {
        throw new Exception("The method or operation is not implemented.");
    }
}
```

```

public override string StandardName
{
    get { throw new Exception("The method or operation is not implemented."); }
}
}

```

8.1.5 Abstract Methods

The methods in the abstract class that are declared without a body are termed as abstract methods. These methods are implemented in the inheriting class.

Similar to a regular method, an abstract method is declared with an access modifier, a return type and a signature. However, an abstract method does not have a body and the method declaration ends with a semicolon.

Abstract methods provide a common functionality for the classes inheriting the abstract class. The subclasses of the abstract class can override and implement the abstract methods.

Figure 8.4 displays an example of abstract methods.

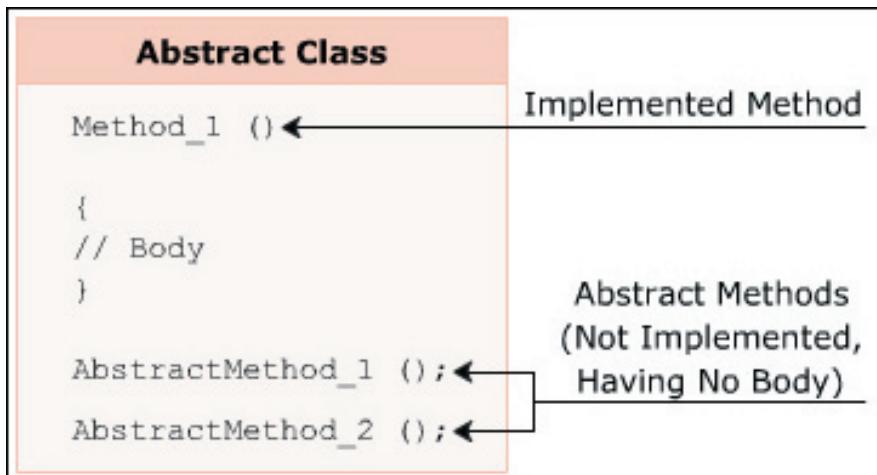


Figure 8.4: Abstract Methods

8.2 Multiple Inheritance through Interfaces

A subclass in C# cannot inherit two or more base classes. This is because C# does not support multiple inheritance. To overcome this drawback, interfaces were introduced. A class in C# can implement multiple interfaces.

8.2.1 Purpose

Consider a class **Dog** that needs to inherit features of **Canine** and **Animal** classes. The **Dog** class cannot inherit methods of both these classes as C# does not support multiple inheritance. However, if **Canine** and **Animal** are declared as interfaces, the class **Dog** can implement methods from both the interfaces.

Figure 8.5 displays an example of the subclasses in C#.

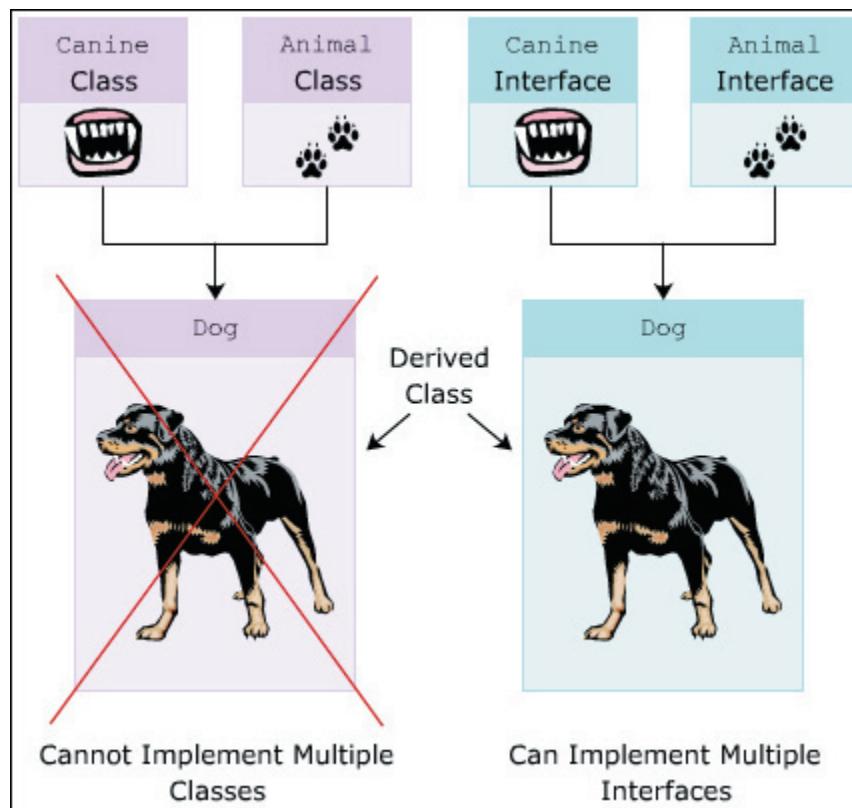


Figure 8.5: Subclasses in C#

8.2.2 Interfaces

An interface contains only abstract members. Unlike an abstract class, an interface cannot implement any method. Similar to an abstract class, an interface cannot be instantiated. An interface can only be inherited by classes or other interfaces.

An interface is declared using the keyword `interface`. In C#, by default, all members declared in an interface have `public` as the access modifier.

Figure 8.6 displays an example of an interface.

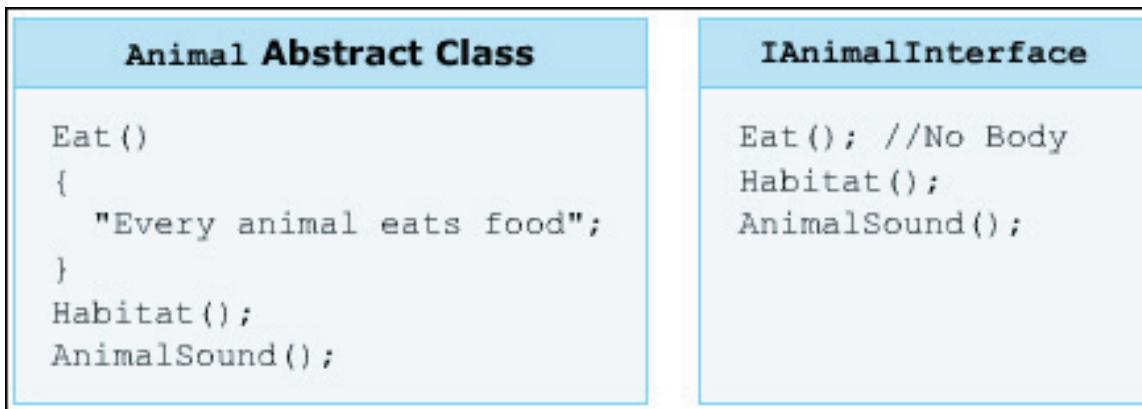


Figure 8.6: Interfaces

The following syntax is used to declare an interface.

Syntax:

```
interface <InterfaceName>
{
    //interface members
}
```

where,

interface: Declares an interface.

InterfaceName: Is the name of the interface.

Code Snippet 4 declares an interface IAnimal.

Code Snippet 4:

```
interface IAnimal
{
    void AnimalType();
}
```

In Code Snippet 4, the interface **IAnimal** is declared and the interface declares an abstract method **AnimalType()**.

Note - Interfaces cannot contain constants, data fields, constructors, destructors, and static members.

8.2.3 Implementing an Interface

An interface is implemented by a class in a way similar to inheriting a class. When implementing an interface in a class, you need to implement all the abstract methods declared in the interface. If all the methods are not implemented, the class cannot be compiled. The methods implemented in the class should be declared with the same name and signature as defined in the interface.

Figure 8.7 displays the implementation of an interface.

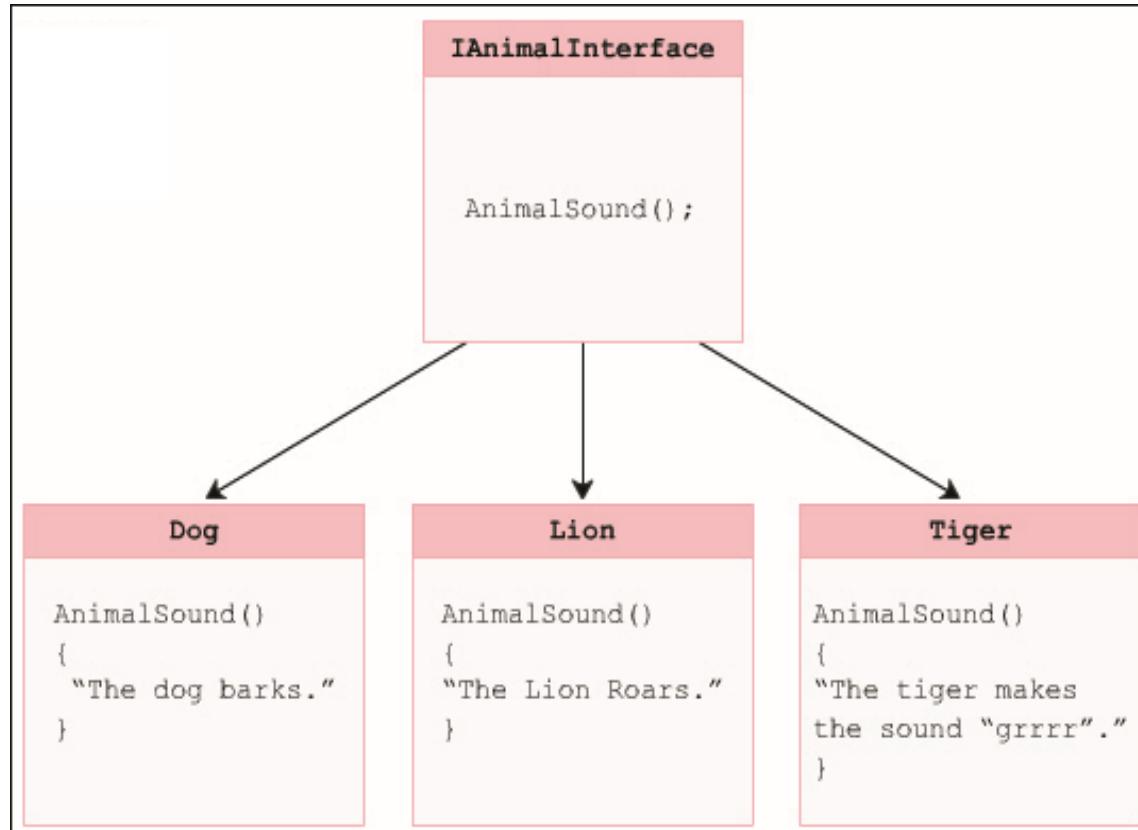


Figure 8.7: Implementation of Interface

The following syntax is used to implement an interface.

Syntax:

```

class <ClassName> : <InterfaceName>
{
    //Implement the interface methods.
    //class members
}
  
```

where,

`<InterfaceName>`: Specifies the name of the interface.

Code Snippet 5 declares an interface **IAnimal** and implements it in the class **Dog**.

Code Snippet 5:

```
interface IAnimal
{
    void Habitat();
}

class Dog : IAnimal
{
    public void Habitat()
    {
        Console.WriteLine("Can be housed with human beings");
    }

    static void Main(string[] args)
    {
        Dog objDog = new Dog();
        Console.WriteLine(objDog.GetType().Name);
        objDog.Habitat();
    }
}
```

Code Snippet 5 creates an interface **IAnimal** that declares the method **Habitat()**. The class **Dog** implements the interface **IAnimal** and its method **Habitat()**. In the **Main()** method of the **Dog** class, the class name is displayed using the object and then, the method **Habitat()** is invoked using the instance of the **Dog** class.

Output:

Dog

Can be housed with human beings

8.2.4 Interfaces and Multiple Inheritance

Multiple interfaces can be implemented in a single class. This implementation provides the functionality of multiple inheritance. You can implement multiple interfaces by placing commas between the interface names while implementing them in a class.

A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces. The **override** keyword is not used while implementing abstract methods of an interface.

Figure 8.8 displays the concept of multiple inheritance using interfaces.

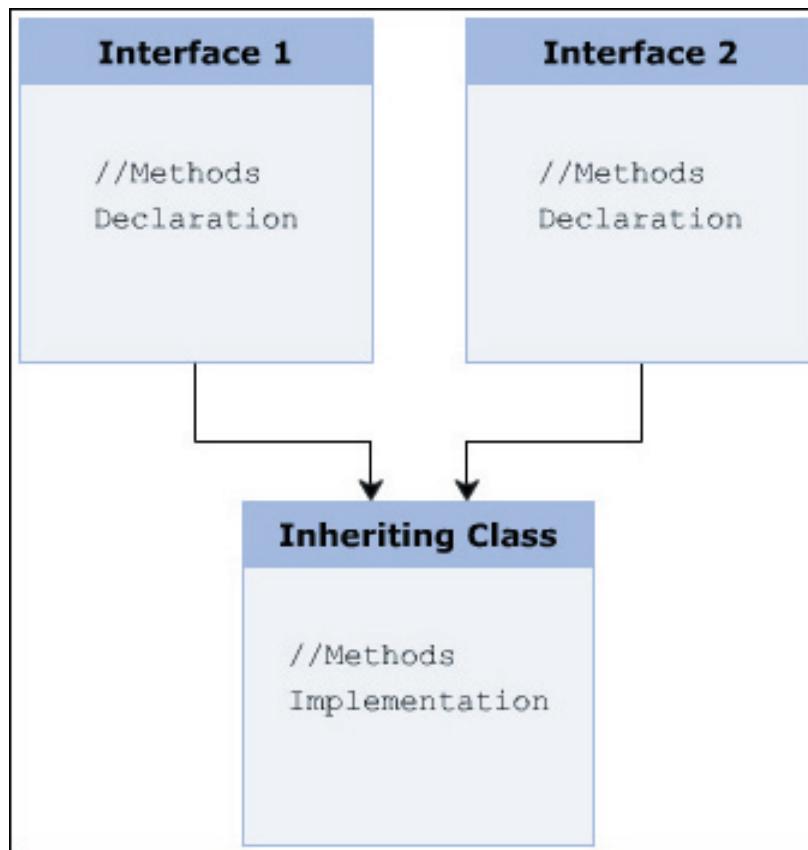


Figure 8.8: Multiple Inheritance Using Interfaces

The following syntax is used to implement multiple interfaces.

Syntax:

```
class <ClassName> : <Interface1>, <Interface2>
{
    //Implement the interface methods
}
```

where,

Interface1: Specifies the name of the first interface.

Interface2: Specifies the name of the second interface.

Code Snippet 6 declares and implements multiple interfaces.

Code Snippet 6:

```
interface ITerrestrialAnimal
{
    void Eat();
}

interface IMarineAnimal
{
    void Swim();
}

class Crocodile : ITerrestrialAnimal, IMarineAnimal
{
    public void Eat()
    {
        Console.WriteLine("The Crocodile eats flesh");
    }

    public void Swim()
    {
        Console.WriteLine("The Crocodile can swim four times faster than an
Olympic swimmer");
    }

    static void Main(string[] args)
    {
        Crocodile objCrocodile = new Crocodile();
        objCrocodile.Eat();
        objCrocodile.Swim();
    }
}
```

In Code Snippet 6, the interfaces **ITerrestrialAnimal** and **IMarineAnimal** are declared. The two interfaces declare methods **Eat()** and **Swim()**.

Output:

The Crocodile eats flesh

The Crocodile can swim four times faster than an Olympic swimmer

Note - C# allows you to inherit a base class and implement more than one interface at the same time.

8.2.5 Explicit Interface Implementation

A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names. In addition, if an interface has a method name identical to the name of a method declared in the inheriting class, this interface has to be explicitly implemented.

Consider the interfaces `ITerrestrialAnimal` and `IMarineAnimal`. The interface `ITerrestrialAnimal` declares methods `Eat()` and `Habitat()`. The interface `IMarineAnimal` declares methods `Eat()` and `Swim()`. The class `Crocodile` implementing the two interfaces has to explicitly implement the method `Eat()` from both interfaces by specifying the interface name before the method name.

While explicitly implementing an interface, you cannot mention modifiers such as `abstract`, `virtual`, `override`, or `new`.

Figure 8.9 displays the explicit implementation of interface.

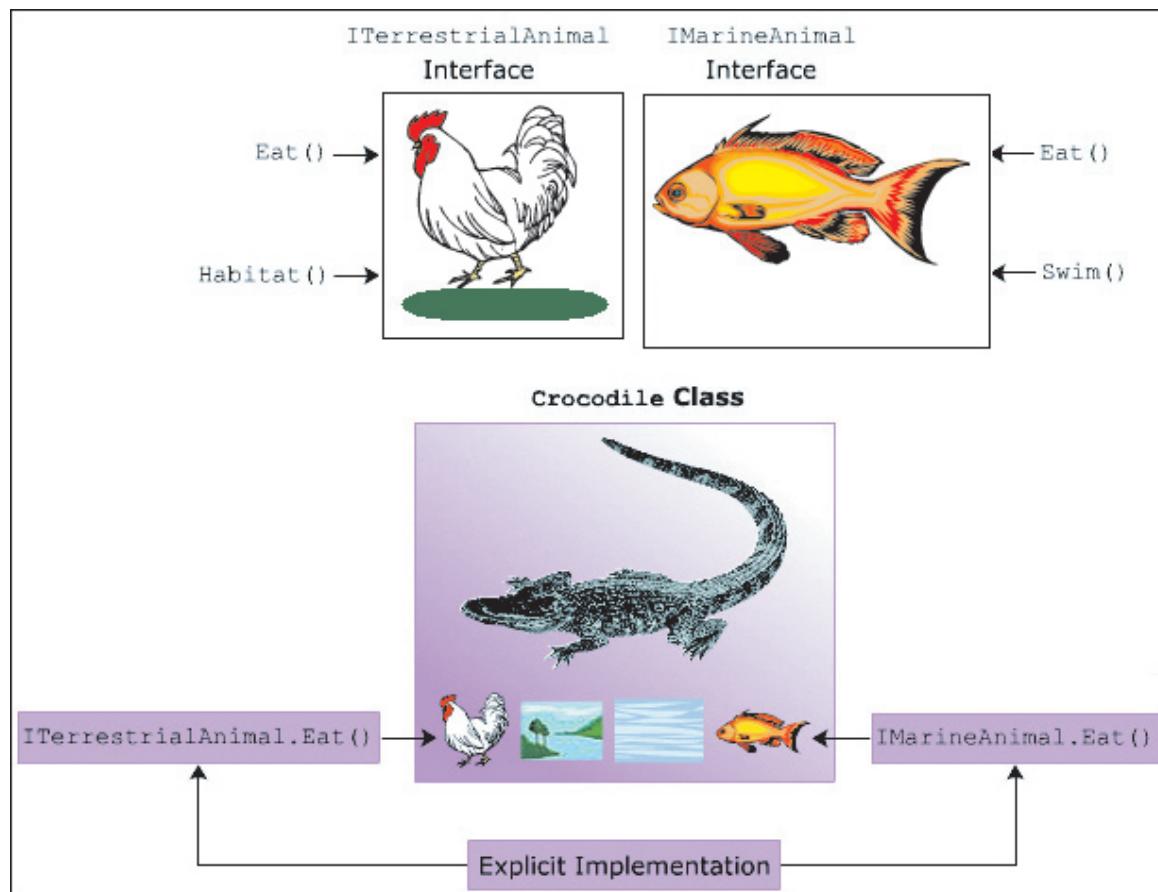


Figure 8.9: Explicit Implementation of Interface

The following syntax is used to explicitly implement interfaces.

Syntax:

```
class <ClassName> : <Interface1>, <Interface2>
{
<access modifier> Interface1.Method();
{
//statements;
}

<access modifier> Interface2.Method();
{
//statements;
}
}
```

where,

Interface1: Specifies the first interface implemented.

Interface2: Specifies the second interface implemented.

Method(): Specifies the same method name declared in the two interfaces.

Code Snippet 7 demonstrates the use of implementing interfaces explicitly.

Code Snippet 7:

```
interface ITerrestrialAnimal
{
    string Eat();
}

interface IMarineAnimal
{
    string Eat();
}

class Crocodile : ITerrestrialAnimal, IMarineAnimal
{
    string ITerrestrialAnimal.Eat()
    {
```

```
        string terCroc = "Crocodile eats other animals";
        return terCroc;
    }

    string IMarineAnimal.Eat()
    {
        string marCroc = "Crocodile eats fish and marine animals";
        return marCroc;
    }

    public string EatTerrestrial()
    {
        ITerrestrialAnimal objTerAnimal;
        objTerAnimal = this;
        return objTerAnimal.Eat();
    }

    public string EatMarine()
    {
        IMarineAnimal objMarAnimal;
        objMarAnimal = this;
        return objMarAnimal.Eat();
    }

    public static void Main(string[] args)
    {
        Crocodile objCrocodile = new Crocodile();
        string terCroc = objCrocodile.EatTerrestrial();
        Console.WriteLine(terCroc);
        string marCroc = objCrocodile.EatMarine();
        Console.WriteLine(marCroc);
    }
}
```

In Code Snippet 7, the class **Crocodile** explicitly implements the method **Eat()** of the two interfaces, **ITerrestrialAnimal** and **IMarineAnimal**. The method **Eat()** is called by creating a reference of the two interfaces and then calling the method.

Output:

Crocodile eats other animals

Crocodile eats fish and marine animals

8.2.6 Interface Inheritance

An interface can inherit multiple interfaces but cannot implement them. The implementation has to be done by a class.

Consider three interfaces, **IAnimal**, **ICarnivorous** and **IReptile**. The interface **IAnimal** declares methods defining general behavior of all animals. The interface **ICarnivorous** declares methods defining the general eating habits of carnivorous animals. The interface **IReptile** inherits interfaces **IAnimal** and **ICarnivorous**. However, these interfaces cannot be implemented by the interface **IReptile** as interfaces cannot implement methods. The class implementing the **IReptile** interface must implement the methods declared in the **IReptile** interface as well as the methods declared in the **IAnimal** and **ICarnivorous** interfaces.

Figure 8.10 displays an example of interface inheritance.

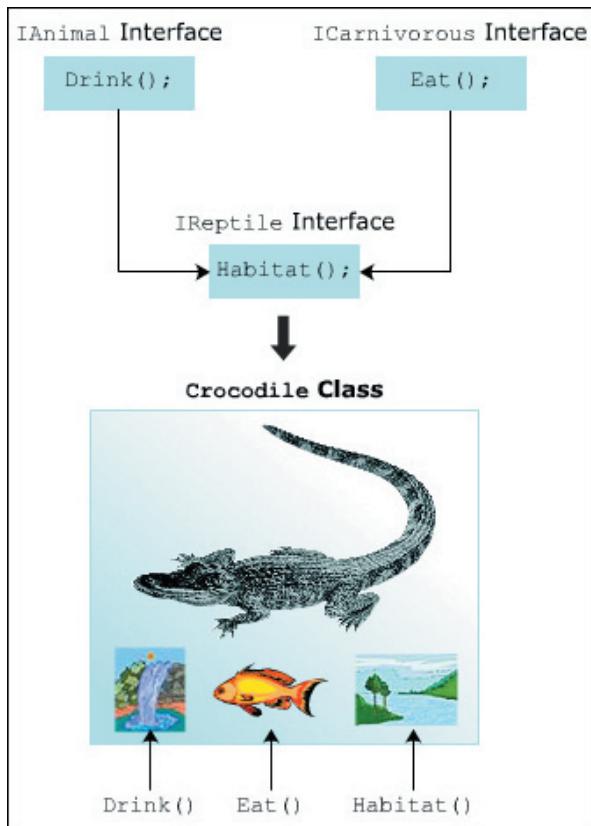


Figure 8.10: Interface Inheritance

The following syntax is used to inherit an interface.

Syntax:

```
interface<InterfaceName> : <Inherited_InterfaceName>
{
    // method declaration;
}
```

where,

InterfaceName: Specifies the name of the interface that inherits another interface.

Inherited_InterfaceName: Specifies the name of the inherited interface.

Code Snippet 8 declares interfaces that are inherited by other interfaces.

Code Snippet 8:

```
interface IAnimal
{
    void Drink();
}

interface ICarnivorous
{
    void Eat();
}

interface IReptile : IAnimal, ICarnivorous
{
    void Habitat();
}

class Crocodile : IReptile
{
    public void Drink()
    {
        Console.WriteLine("Drinks freshwater");
    }

    public void Habitat()
    {
```

```

Console.WriteLine("Can stay in Water and Land");
}

public void Eat()
{
    Console.WriteLine("Eats Flesh");
}

static void Main(string[] args)
{
    Crocodile objCrocodile = new Crocodile();

    Console.WriteLine(objCrocodile.GetType().Name);

    objCrocodile.Habitat();
    objCrocodile.Eat();
    objCrocodile.Drink();
}
}

```

In Code Snippet 8, three interfaces, **IAnimal**, **ICarnivorous**, and **IReptile**, are declared. The three interfaces declare methods **Drink()**, **Eat()** and **Habitat()** respectively. The **IReptile** interface inherits the **IAnimal** and **ICarnivorous** interfaces. The class **Crocodile** implements the interface **IReptile**, its declared method **Habitat()** and the inherited methods **Eat()** and **Drink()** of the **ICarnivorous** and **IAnimal** interfaces.

Output:

```

Crocodile
Can stay in Water and Land
Eats Flesh
Drinks freshwater

```

8.2.7 Interface Re-implementation

A class can re-implement an interface. Re-implementation occurs when the method declared in the interface is implemented in a class using the **virtual** keyword and this virtual method is then overridden in the derived class.

Code Snippet 9 demonstrates the purpose of re-implementation of an interface.

Code Snippet 9:

```
using System;

interface IMath

{

    voidArea();

}

class Circle : IMath

{

    public const float PI = 3.14F;

    protected float Radius;

    protected double AreaOfCircle;

    public virtual voidArea()

    {

        AreaOfCircle = (PI * Radius * Radius);

    }

}

class Sphere : Circle

{

    double _areaOfSphere;

    public override voidArea()

    {

        base.Area();

        _areaOfSphere = (AreaOfCircle * 4);

    }

    static void Main(string[] args)

    {

        Sphere objSphere = new Sphere();

        objSphere.Radius = 7;

        objSphere.Area();

    }

}
```

```

        Console.WriteLine("Area of Sphere: {0:F2}" ,
objSphere._areaOfSphere);
}
}

```

In Code Snippet 9, the interface **IMath** declares the method **Area()**. The class **Circle** implements the interface **IMath**. The class **Circle** declares a virtual method **Area()** that calculates the area of a circle. The class **Sphere** inherits the class **Circle** and overrides the base class method **Area()** to calculate the area of the sphere. The **base** keyword calls the base class method **Area()**, thereby allowing the use of base class methods in the derived class.

Figure 8.11 displays the output of re-implementation of an interface.

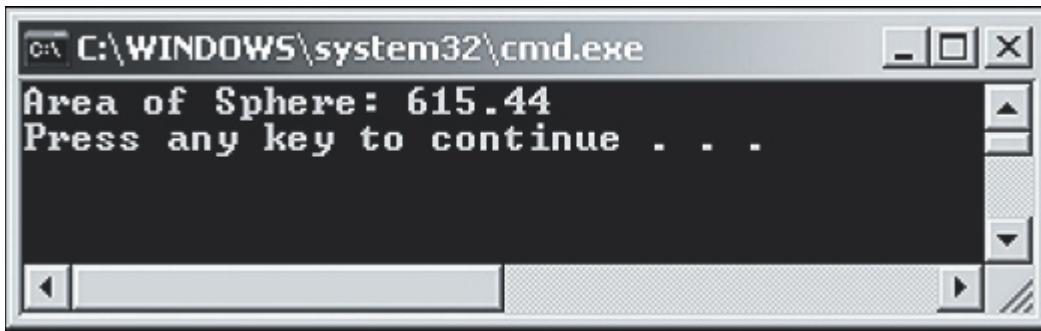


Figure 8.11: Re-implementation of an Interface

8.2.8 The *is* and *as* Operators in Interfaces

The **is** and **as** operators in C# verify whether the specified interface is implemented or not. The **is** operator is used to check the compatibility between two types or classes. It returns a boolean value based on the check operation performed. On the other hand, the **as** operator returns **null** if the two types or classes are not compatible with each other.

Code Snippet 10 demonstrates an interface with the **is** operator.

Code Snippet 10:

```

using System;

interface ICalculate
{
    double Area();
}

class Rectangle : ICalculate
{
    float _length;
}

```

```
float _breadth;
public Rectangle(float valOne, float valTwo)
{
    _length=valOne;
    _breadth=valTwo;
}
public double Area()
{
    return _length * _breadth;
}
static void Main(string[] args)
{
    Rectangle objRectangle = new Rectangle(10.2F, 20.3F);
    if (objRectangle is ICalculate)
    {
        Console.WriteLine("Area of rectangle: {0:F2}", objRectangle.Area());
    }
    else
    {
        Console.WriteLine("Interface method not implemented");
    }
}
```

In Code Snippet 10, an interface **ICalculate** declares a method **Area()**. The class **Rectangle** implements the interface **ICalculate** and it consists of a parameterized constructor that assigns the dimension values of the rectangle. The **Area()** method calculates the area of the rectangle. The **Main()** method creates an instance of the class **Rectangle**. The **is** operator is used within the **if-else** construct to check whether the class **Rectangle** implements the methods declared in the interface **ICalculate**.

Figure 8.12 displays the output of the example using `is` operator.

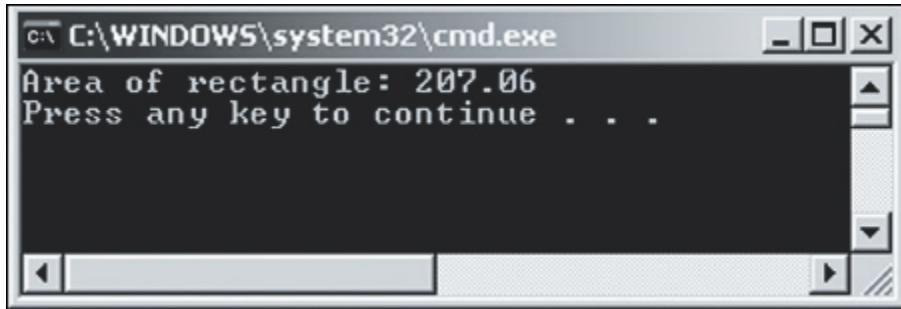


Figure 8.12: Example of `is` Operator

Code Snippet 11 demonstrates an interface with the `as` operator.

Code Snippet 11:

```
using System;

interface ISet
{
    void AcceptDetails(int valOne, string valTwo);
}

interface IGet
{
    void Display();
}

class Employee : ISet
{
    int _empID;
    string _empName;
    public void AcceptDetails(int valOne, string valTwo)
    {
        _empID = valOne;
        _empName = valTwo;
    }
    static void Main(string[] args)
    {
```

```
Employee objEmployee = new Employee();
objEmployee.AcceptDetails(10, "Jack");
IGet objGet = objEmployee as IGet;
if (objGet != null)
{
    objGet.Display();
}
else
{
    Console.WriteLine("Invalid casting occurred");
}
```

In Code Snippet 11, the interface **ISet** declares a method **AcceptDetails** with two parameters and the interface **IGet** declares a method **Display()**. The class **Employee** implements the interface **ISet** and implements the method declared within **ISet**. The **Main()** method creates an instance of the class **Employee**. An attempt is made to retrieve an instance of **IGet** interface checks whether the class **Employee** implements the methods defined in the interface. Since the **as** operator returns null, the code displays the specified error message.

Figure 8.13 displays the output of the example that uses the as operator.



Figure 8.13: Example of as Operator

8.3 Abstract Classes and Interfaces

Abstract classes and interfaces both declare methods without implementing them. Although both abstract classes and interfaces share similar characteristics, they serve different purposes in a C# application.

8.3.1 Similarities

The similarities between abstract classes and interfaces are as follows:

- Neither an abstract class nor an interface can be instantiated.
- Both, abstract classes as well as interfaces, contain abstract methods.
- Abstract methods of both, the abstract class as well as the interface are implemented by the inheriting subclass.
- Both, abstract classes as well as interfaces, can inherit multiple interfaces.

Figure 8.14 displays the similarities between an abstract class and an interface.

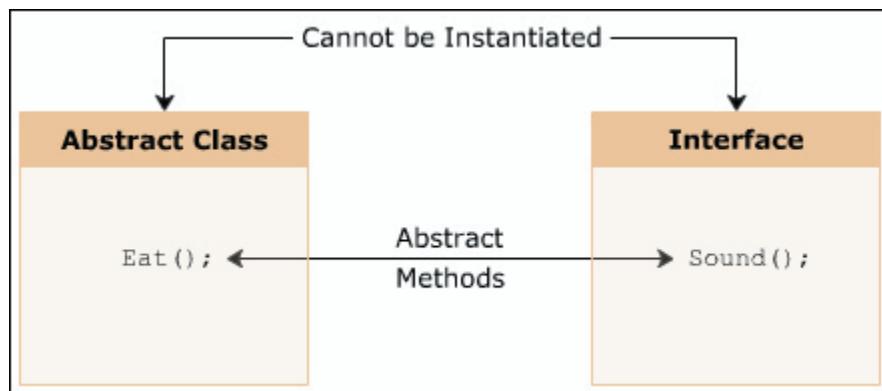


Figure 8.14: Similarities between an Abstract Class and an Interface

8.3.2 Differences

Abstract classes and interfaces are similar because both contain abstract methods that are implemented by the inheriting class. However, there are certain differences between an abstract class and an interface. Table 8.1 lists the differences between the two.

Abstract Classes	Interfaces
An abstract class can inherit a class and multiple interfaces.	An interface can inherit multiple interfaces but cannot inherit a class.
An abstract class can have methods with a body.	An interface cannot have methods with a body.
An abstract class method is implemented using the <code>override</code> keyword.	An interface method is implemented without using the <code>override</code> keyword.
An abstract class is a better option when you need to implement common methods and declare common abstract methods.	An interface is a better option when you need to declare only abstract methods.

Abstract Classes	Interfaces
An abstract class can declare constructors and destructors.	An interface cannot declare constructors or destructors.

Table 8.1: Differences between Abstract Classes and Interfaces

8.3.3 Recommendations for Using Abstract Classes and Interfaces

Abstract classes and interfaces both contain abstract methods that are implemented by the inheriting class. However, an abstract class can inherit another class whereas an interface cannot inherit a class. Therefore, abstract classes and interfaces have certain similarities as well as certain differences.

There are some guidelines to decide when to use an interface and when to use an abstract class. These are as follows:

- If a programmer wants to create reusable programs and maintain multiple versions of these programs, it is recommended to create an abstract class. Abstract classes help to maintain the version of the programs in a simple manner. This is because by updating the base class, all inheriting classes are automatically updated with the required change. Unlike abstract classes, interfaces cannot be changed once they are created. A new interface needs to be created to create a new version of the existing interface.
- If a programmer wants to create different methods that are useful for multiple different types of objects, it is recommended to create an interface. This is because abstract classes are widely created only for related objects. There must exist a relationship between the abstract class and the classes that inherit the abstract class. On the other hand, interfaces are suitable for implementing similar functionalities in dissimilar classes.

8.4 Check Your Progress

1. Which of these statements about abstract classes and abstract methods are true?

(A)	An abstract class can be declared using the <code>declare</code> keyword.
(B)	An abstract class cannot be instantiated using the <code>new</code> keyword.
(C)	An abstract class can be created by declaring and defining methods.
(D)	An abstract method can be declared without an access modifier.
(E)	An abstract method can be implemented in the inheriting class using the <code>override</code> keyword.

(A)	A, B, D	(C)	B, C, D
(B)	A, C	(D)	B, E

2. You are trying to inherit the abstract class `Vehicle` and implement its methods in the subclass `Ferrari`. Which of the following codes will help you to achieve this?

(A)	<pre>class Vehicle { public void Wheels() { Console.WriteLine("Every Car has four wheels"); } public abstract void Speed(); } class Ferrari : Vehicle { public override void Speed() { Console.WriteLine("The Speed of Ferrari exceeds 200mph"); } }</pre>
-----	--

(A)	<pre>static void Main(String[] args) { Ferrari objCar = new Ferrari(); objCar.Speed(); }</pre>
(B)	<pre>Abstract class Vehicle { public void Wheels() { Console.WriteLine("Every Car has four wheels"); } public abstract void Speed(); } class Ferrari : Vehicle { public override void Speed() { Console.WriteLine("The Speed of Ferrari exceeds 200 mph"); } static void Main(String args[]) { Ferrari objCar = new Ferrari(); objCar.Speed(); } }</pre>
(C)	<pre>Class Vehicle { public void Wheels() { Console.WriteLine("Every Car has four wheels"); } }</pre>

```

        }

        public abstract void Speed();

    }

    class Ferrari extends Vehicle
    {

        public override void Speed()
        {

            Console.WriteLine("The Speed of Ferrari exceeds 200 mph");
        }

        static void Main(String[] args)
        {

            Ferrari objCar = new Ferrari();
            objCar.Speed();
        }
    }
}

```

```

abstract class Vehicle
{
    public void Wheels()
    {
        Console.WriteLine("Every Car has four wheels");
    }

    public abstract void Speed();
}

class Ferrari : Vehicle
{
    public override void Speed()
    {
        Console.WriteLine("The Speed of Ferrari exceeds 200 mph");
    }

    static void Main(String[] args)
}

```

(D)	<pre>{ Ferrari objCar = new Ferrari(); objCar.Speed(); }</pre>
-----	--

(A)	A	(C)	C
(B)	B	(D)	D

3. Which of these statements about interfaces are true?

(A)	An interface can contain abstract as well as implemented methods.
(B)	An inheriting class can override the implemented methods of an interface.
(C)	A class can implement abstract methods from multiple interfaces.
(D)	A class can explicitly implement multiple interfaces when the interfaces have methods with identical names.
(E)	An interface can implement multiple interfaces but only a single abstract class.

(A)	A, B, D	(C)	C, D
(B)	A, C	(D)	B, E

4. You are trying to inherit and implement the interface ICar and its methods in the subclass Zen. Which of the following codes will help you to achieve this?

(A)	<pre>interface ICar { public void Wheels(); public void Speed(); } class Zen : ICar { public void Wheels() {</pre>
-----	---

```

        Console.WriteLine("Zen has four wheels");

    }

    public void Speed()
    {

        Console.WriteLine("Zen crosses 200 mph");
    }

    static void Main(String[] args)
    {

        Zen objZen = new Zen();

        objZen.Wheels();

        objZen.Speed();
    }

}

```

```

interface ICar
{
    void Wheels();
    void Speed();
}

class Zen : ICar
{
    public void Wheels()
    {

        Console.WriteLine("Zen has four wheels");
    }

    public void Speed()
    {

        Console.WriteLine("Zen crosses 200 mph");
    }

    static void Main(String[] args)
    {

        Zen objZen = new Zen();
    }
}

```

(B)	<pre> objZen.Wheels(); objZen.Speed(); } } </pre>
(C)	<pre> interface ICar { void Wheels(); void Speed(); } class Zen :: ICar { public void Wheels() { Console.WriteLine("Zen has four wheels"); } public void Speed() { Console.WriteLine("Zen crosses 200 mph"); } static void Main(String[] args) { Zen objZen = new Zen(); objZen.Wheels(); objZen.Speed(); } } </pre>
(D)	<pre> interface ICar { void Wheels(); void Speed(); } </pre>

(D)

```

        }

class Zen implements ICar
{
    public void Wheels()
    {
        Console.WriteLine("Zen has four wheels");
    }

    public void Speed()
    {
        Console.WriteLine("Zen crosses 200 mph");
    }

    static void Main(String[] args)
    {
        Zen objZen = new Zen();
        objZen.Wheels();
        objZen.Speed();
    }
}

```

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of these statements about abstract classes and interfaces are true?

(A)	An abstract class can inherit multiple classes.
(B)	An interface can inherit multiple classes and interfaces.
(C)	An interface cannot declare constructors or destructors.
(D)	An abstract class method can be implemented using the override keyword.
(E)	An interface cannot be instantiated.

(A)	A, B	(C)	C, D, E
(B)	B	(D)	A, D

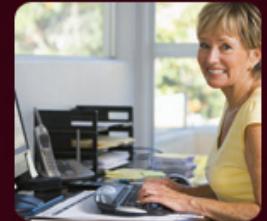
8.4.1 Answers

1.	D
2.	D
3.	C
4.	B
5.	C



Summary

- An abstract class can be referred to as an incomplete base class and can implement methods that are similar for all the subclasses.
- IntelliSense provides access to member variables, functions, and methods of an object or a class.
- When implementing an interface in a class, you need to implement all the abstract methods declared in the interface.
- A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- Re-implementation occurs when the method declared in the interface is implemented in a class using the virtual keyword and this virtual method is then overridden in the derived class.
- The is operator is used to check the compatibility between two types or classes and as operator returns null if the two types or classes are not compatible with each other.



Session -9

Properties and Indexers

Welcome to the Session, **Properties and Indexers**.

Properties are data members that allow you to protect the private fields of a class. Property accessors known as methods allow you to read and assign values to fields. Apart from properties, C# also supports indexers, which allow you to access objects like arrays.

In this session, you will learn to:

- ➔ Define properties in C#
- ➔ Explain properties, fields, and methods
- ➔ Explain indexers

9.1 Properties in C#

Access modifiers like **public**, **private**, **protected**, and **internal** are used to control the accessibility of fields and methods in C#. The **public** fields are accessible by other classes, but **private** fields are accessible only by the class in which they are declared. C# uses a feature called properties that allows you to set and retrieve values of fields declared with any access modifier in a secured manner. This is because properties allow you to validate values before assigning them to fields.

For example, consider fields that store names and IDs of employees. You can create properties for these fields to ensure accuracy and validity of values stored in them.

9.1.1 Applications of Properties

Properties allow you to protect a field in the class by reading and writing to the field through a property declaration. Additionally, properties allow you to access private fields, which would otherwise be inaccessible.

Properties can validate values before allowing you to change them and also perform specified actions on those changes. Therefore, properties ensure security of private data.

Properties support abstraction and encapsulation by exposing only necessary actions and hiding their implementation.

9.1.2 Declaring Properties

The following syntax is used to declare a property in C#.

Syntax:

```
<access_modifier><return_type><PropertyName>
{
    //body of the property
}
```

where,

access_modifier: Defines the scope of access for the property, which can be **private**, **public**, **protected** or **internal**.

return_type: Determines the type of data the property will return.

PropertyName: Is the name of the property.

Note - A property declaration contains special methods to read and set private values. However, properties are accessed in a way that is similar to accessing a field. Therefore, properties are also known as **smart fields**.

9.1.3 get and set Accessors

Property accessors allow you to read and assign a value to a field by implementing two special methods. These methods are referred to as the `get` and `set` accessors.

The `get` accessor is used to read a value and is executed when the property name is referred. It does not take any parameter and returns a value that is of the return type of the property.

The `set` accessor is used to assign a value and is executed when the property is assigned a new value using the equal to (=) operator. This value is stored in the private field by an implicit parameter called `value` (keyword in C#) used in the `set` accessor.

The following syntax is used to declare the accessors of a property.

Syntax:

```
<access_modifier><return_type>PropertyName>
{
    get
    {
        // return value
    }
    set
    {
        // assign value
    }
}
```

Code Snippet 1 demonstrates the use of the `get` and `set` accessors.

Code Snippet 1:

```
using System;
class SalaryDetails
{
    private string _empName;
    public string EmployeeName
    {
        get
        {
            return _empName;
        }
    }
}
```

```

    }
    set
    {
        _empName = value;
    }
}

static void Main (string [] args)
{
    SalaryDetails objSal = new SalaryDetails ();
    objSal.EmployeeName = "Patrick Johnson";
    Console.WriteLine ("Employee Name: " + objSal.EmployeeName);
}
}

```

In Code Snippet 1, the class **SalaryDetails** creates a private variable **_empName** and declares a property called **EmployeeName**. The instance of the **SalaryDetails** class, **objSal**, invokes the property **EmployeeName** using the dot (.) operator to initialize the value of employee name. This invokes the **set** accessor, where the **value** keyword assigns the value to **_empName**.

The code displays the employee name by invoking the property name. This invokes the **get** accessor, which returns the assigned employee name.

Output:

Employee Name: Patrick Johnson

Note - It is mandatory to always end the get accessor with a return statement.

9.1.4 Categories of Properties

Properties are broadly divided into three categories, read-only, write-only, and read-write properties.

→ **Read-Only Property**

The read-only property allows you to retrieve the value of a private field. To create a read-only property, you should define the **get** accessor.

The following syntax creates a read-only property.

Syntax:

```
<access_modifer> <return_type> <PropertyName> {
```

```
get
{
    // return value
}
```

Code Snippet 2 demonstrates how to create a read-only property.

Code Snippet 2:

```
using System;
class Books
{
    string _bookName;
    long _bookID;
    public Books(string name, int value)
    {
        _bookName = name;
        _bookID = value;
    }
    public string BookName
    {
        get
        {
            return _bookName;
        }
    }
    public long BookID
    {
        get
        {
            return _bookID;
        }
    }
}
```

```

}

class BookStore
{
    static void Main(string[] args)
    {
        Books objBook = new Books("Learn C# in 21 Days", 10015);
        Console.WriteLine("Book Name: " + objBook.BookName);
        Console.WriteLine("Book ID: " + objBook.BookID);
    }
}

```

In Code Snippet 2, the **Books** class creates two read-only properties that returns the name and ID of the book. The class **BookStore** defines a `Main()` method that creates an instance of the class **Books** by passing the parameter values that refer to the name and ID of the book. The output displays the name and ID of the book by invoking the get accessor of the appropriate read-only properties.

Output:

Book Name: Learn C# in 21 Days

Book ID: 10015

→ **Write-Only Property**

The write-only property allows you to change the value of a `private` field. To create a write-only property, you should define the `set` accessor.

The following syntax creates a write-only property:

Syntax:

```

<access_modifier> <return_type> <PropertyName>{
    set
    {
        // assign value
    }
}

```

Code Snippet 3 demonstrates how to create a write-only property.

Code Snippet 3:

```
using System;
class Department
{
    string _deptName;
    int _deptID;
    public string DeptName
    {
        set
        {
            _deptName = value;
        }
    }
    public int DeptID
    {
        set
        {
            _deptID = value;
        }
    }
    public void Display()
    {
        Console.WriteLine("Department Name: " + _deptName);
        Console.WriteLine("Department ID: " + _deptID);
    }
}
class Company
{
    static void Main(string[] args)
```

```
{
    Department objDepartment = new Department();
    objDepartment.DeptID = 201;
    objDepartment.DeptName = "Sales";
    objDepartment.Display();
}
```

In Code Snippet 3, the **Department** class consists of two write-only properties. The **Main()** method of the class **Company** instantiates the class **Department** and this instance invokes the set accessor of the appropriate write-only properties to assign the department name and its ID. The **Display()** method of the class **Department** displays the name and ID of the department.

Output:

```
Department Name: Sales
Department ID: 201
```

→ **Read-Write Property**

The read-write property allows you to set and retrieve the value of a private field. To create a read-write property, you should define the set and get accessors.

The following syntax creates a read-write property:

Syntax:

```
<access_modifier> <return type> <PropertyName>{
    get
    {
        // return value
    }
    set
    {
        // assign value
    }
}
```

Code Snippet 4 demonstrates how to create a read-write property.

Code Snippet 4:

```
using System;
class Product
{
    string _productName;
    int _productID;
    float _price;
    public Product(string name, int val)
    {
        _productName = name;
        _productID = val;
    }
    public float Price
    {
        get
        {
            return _price;
        }
        set
        {
            if (value < 0)
            {
                _price = 0;
            }
            else
            {
                _price = value;
            }
        }
    }
}
```

```

public void Display()
{
    Console.WriteLine("Product Name: " + _productName);
    Console.WriteLine("Product ID: " + _productID);
    Console.WriteLine("Price: " + _price + "$");
}

class Goods
{
    static void Main(string[] args)
    {
        Product objProduct = new Product("Hard Disk", 101);
        objProduct.Price = 345.25F;
        objProduct.Display();
    }
}

```

In Code Snippet 4, the class **Product** creates a read-write property **Price** that assigns and retrieves the price of the product based on the **if** statement. The **Goods** class defines the **Main()** method that creates an instance of the class **Product** by passing the values as parameters that are name and ID of the product. The **Display()** method of the class **Product** is invoked that displays the name, ID, and price of the product.

Output:

```

Product Name: Hard Disk
Product ID: 101
Price: 345.25$

```

Properties can be further classified as static, abstract, and boolean properties.

9.1.5 Static Properties

The static property is declared by using the **static** keyword. It is accessed using the class name and thus, belongs to the class rather than just an instance of the class. Thus, a programmer can use a static property without creating an instance of the class. A static property is used to access and manipulate static fields of a class in a safe manner.

Code Snippet 5 demonstrates a class with a static property.

Code Snippet 5:

```
using System;
class University
{
    private static string _department;
    private static string _universityName;
    public static string Department
    {
        get
        {
            return _department;
        }
        set
        {
            _department = value;
        }
    }
    public static string UniversityName
    {
        get
        {
            return _universityName;
        }
        set
        {
            _universityName = value;
        }
    }
}
```

```

class Physics
{
    static void Main(string[] args)
    {
        University.UniversityName = "University of Maryland";
        University.Department = "Physics";
        Console.WriteLine("University Name: " + University.UniversityName);
        Console.WriteLine("Department name: " + University.Department);
    }
}

```

In Code Snippet 5, the class **University** defines two static properties **UniversityName** and **DepartmentName**. The **Main()** method of the class **Physics** invokes the static properties **UniversityName** and **DepartmentName** of the class **University** by using the dot (.) operator. This initializes the static fields of the class by invoking the set accessor of the appropriate properties. The code displays the name of the university and the department by invoking the get accessor of the appropriate properties.

Output:

University Name: University of Maryland

Department name: Physics

9.1.6 Abstract Properties

The abstract property is declared by using the **abstract** keyword. The abstract property in a class just contains the declaration of the property without the body of the get and set accessors. The get and set accessors does not contain any statements. These accessors can be implemented in the derived class. An abstract property declaration is only allowed in an abstract class. An abstract property is used when it is required to secure data within multiple fields of the derived class of the abstract class. Further, it is used to avoid redefining properties by reusing the existing properties.

Code Snippet 6 demonstrates a class that uses an abstract property.

Code Snippet 6:

```

using System;
public abstract class Figure
{
    public abstract float DimensionOne
    {

```

```
    set;  
}  
  
    public abstract float DimensionTwo  
    {  
        set;  
    }  
}  
  
class Rectangle : Figure  
{  
    float _dimensionOne;  
    float _dimensionTwo;  
    public override float DimensionOne  
    {  
        set  
        {  
            if (value <= 0)  
            {  
                _dimensionOne = 0;  
            }  
            else  
            {  
                _dimensionOne = value;  
            }  
        }  
    }  
  
    public override float DimensionTwo  
    {  
        set  
        {  
            if (value <= 0)  
            {  
                _dimensionTwo = 0;  
            }  
            else  
            {  
                _dimensionTwo = value;  
            }  
        }  
    }  
}
```

```

        _dimensionTwo = 0;
    }

    else
    {
        _dimensionTwo = value;
    }
}

float Area()
{
    return _dimensionOne * _dimensionTwo;
}

static void Main(string[] args)
{
    Rectangle objRectangle = new Rectangle();
    objRectangle.DimensionOne = 20;
    objRectangle.DimensionTwo = 4.233F;
    Console.WriteLine("Area of Rectangle: " + objRectangle.Area());
}
}

```

In Code Snippet 6, the abstract class **Figure** declares two write-only abstract properties **DimensionOne** and **DimensionTwo**. The class **Rectangle** inherits the abstract class **Figure** and overrides the two abstract properties **DimensionOne** and **DimensionTwo** by setting appropriate dimension values for the rectangle. The **Area()** method calculates the area of the rectangle. The **Main()** method creates an instance of the derived class **Rectangle**. This instance invokes the properties **DimensionOne** and **DimensionTwo**, which, in turn, invokes the set accessor of appropriate properties to assign appropriate dimension values. The code displays the area of the rectangle by invoking the **Area()** method of the **Rectangle** class.

Output:

Area of Rectangle: 84.66

9.1.7 Boolean Properties

A boolean property is declared by specifying the data type of the property as **bool**. Unlike other properties, the boolean property produces only true or false values.

While working with a boolean property, a programmer needs to be sure that the `get` accessor returns the boolean value.

Note - A property can be declared as static by using the `static` keyword. A static property is accessed using the class name and is available to the entire class rather than just an instance of the class. The `set` and the `get` accessors of the static property can access only the static members of the class.

9.1.8 Implementing Inheritance

Properties can be inherited just like other members of the class. This means the base class properties are inherited by the derived class.

Code Snippet 7 demonstrates how properties can be inherited.

Code Snippet 7:

```
using System;

class Employee
{
    string _empName;
    int _empID;
    float _salary;
    public string EmpName
    {
        get
        {
            return _empName;
        }
        set
        {
            _empName = value;
        }
    }
    public int EmpID
    {
        get
        {
```

```
    return _empID;
}

set
{
    _empID = value;
}

}

public float Salary
{
    get
    {
        return _salary;
    }
    set
    {
        if (value < 0)
        {
            _salary = 0;
        }
        else
        {
            _salary = value;
        }
    }
}

class SalaryDetails : Employee
{
    static void Main(string[] args)
    {
        SalaryDetails objSalary = new SalaryDetails();
```

```

    objSalary.EmpName = "Frank";
    objSalary.EmpID = 10;
    objSalary.Salary = 1000.25F;
    Console.WriteLine("Name: " + objSalary.EmpName);
    Console.WriteLine("ID: " + objSalary.EmpID);
    Console.WriteLine("Salary: " + objSalary.Salary + "$");
}
}

```

In Code Snippet 7, the class **Employee** creates three properties to set and retrieve the employee name, ID, and salary respectively. The class **SalaryDetails** is derived from the **Employee** class and inherits its public members. The instance of the **SalaryDetails** class initializes the value of the **_empName**, **_empID**, and **_salary** using the respective properties **EmpName**, **EmpID**, and **Salary** of the base class **Employee**. This invokes the **set** accessors of the respective properties. The code displays the name, ID, and salary of an employee by invoking the **get** accessor of the respective properties. Thus, by implementing inheritance, the code implemented in the base class for defining property can be reused in the derived class.

Output:

```

Name: Frank
ID: 10
Salary: 1000.25$

```

9.1.9 Auto-Implemented Properties

C# provides an alternative syntax to declare properties where a programmer can specify a property in a class without explicitly providing the **get** and **set** accessors. Such properties are called auto-implemented properties and results in more concise and easy-to-understand programs. For an auto-implemented property, the compiler automatically creates a private field to store the property variable. In addition, the compiler automatically creates the corresponding **get** and **set** accessors.

The following syntax creates an auto-implemented property.

Syntax:

```
public string Name { get; set; }
```

Code Snippet 8 uses auto-implemented properties.

Code Snippet 8:

```
class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Designation { get; set; }

    static void Main (string [] args)
    {
        Employee emp = new Employee ();
        emp.Name = "John Doe";
        emp.Age = 24;
        emp.Designation = "Sales Person";
        Console.WriteLine ("Name: {0}, Age: {1}, Designation: {2}", emp.Name,
                           emp.Age, emp.Designation);
    }
}
```

Code Snippet 8 declares three auto-implemented properties: **Name**, **Age**, and **Designation**. The **Main()** method first sets the values of the properties, and then retrieves the values and writes to the console.

Output:

Name : John Doe, Age : 24, Designation : Sales Person

Like normal properties, auto-implemented properties can be declared to be read-only and write-only.

Code Snippet 9 declares read-only and write-only properties.

Code Snippet 9:

```
public float Age { get; private set; }
public int Salary { private get; set; }
```

In Code Snippet 9, the **private** keyword before the **set** keyword declares the **Age** property as read-only. In the second property declaration, the **private** keyword before the **get** keyword declares the **Salary** property as write-only.

Note - Unlike normal properties, auto-implemented properties cannot be assigned a default value at the time of declarations. Any assignment of value to an auto implemented property must be done in the constructor. In addition, auto-implemented properties cannot provide additional functionalities, such as data validations in either of the accessors. Such properties are only meant for simple storage of values.

9.1.10 Object Initializers

In C#, programmer can use object initializers to initialize an object with values without explicitly calling the constructor. The declarative form of object initializers makes the initialization of objects more readable in a program. When object initializers are used in a program, the compiler first accesses the default instance constructor of the class to create the object and then performs the initialization.

Code Snippet 10 uses object initializers to initialize an **Employee** object.

Code Snippet 10 :

```
class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Designation { get; set; }

    static void Main (string [] args)
    {
        Employee emp1 = new Employee
        {
            Name = "John Doe",
            Age = 24,
            Designation = "Sales Person"
        };
        Console.WriteLine ("Name: {0}, Age: {1}, Designation: {2}",
            emp1.Name, emp1.Age, emp1.Designation);
    }
}
```

Code Snippet 10 creates three auto-implemented properties in an **Employee** class. The **Main()** method uses an object initializer to create an **Employee** object initialized with values of its properties.

Output:

Name: John Doe, Age: 24, Designation: Sales Person

9.1.11 Implementing Polymorphism

Properties can implement polymorphism by overriding the base class properties in the derived class. However, properties cannot be overloaded.

Code Snippet 11 demonstrates the implementation of polymorphism by overriding the base class properties.

Code Snippet 11:

```
using System;  
  
class Car  
  
{  
    string _carType;  
    public virtual string CarType  
    {  
        get  
        {  
            return _carType;  
        }  
        set  
        {  
            _carType = value;  
        }  
    }  
}  
  
class Ferrari : Car  
  
{  
    string _carType;  
    public override string CarType  
    {  
        get  
        {  
            return base.CarType;  
        }  
        set  
    }
```

```

{
    base.CarType = value;
    _carType = value;
}
}

static void Main(string[] args)
{
    Car objCar = new Car();
    objCar.CarType = "Utility Vehicle";
    Ferrari objFerrari = new Ferrari();
    objFerrari.CarType = "Sports Car";
    Console.WriteLine("Car Type: " + objCar.CarType);
    Console.WriteLine("Ferrari Car Type: " +
        objFerrari.CarType);
}
}
}

```

In Code Snippet 11, the class **Car** declares a virtual property **CarType**. The class **Ferrari** inherits the base class **Car** and overrides the property **CarType**. The **Main()** method of the class **Ferrari** declares an instance of the base class **Car**.

When the **Main()** method creates an instance of the derived class **Ferrari** and invokes the derived class property **CarType**, the **virtual** property is overridden. However, since the **set** accessor of the derived class invokes the base class property **CarType** using the **base** keyword, the output displays both the **Car** type and the **Ferrari** car type. Thus, the code shows that the properties can be overridden in the derived classes and can be useful in giving customized output.

Output:

```

Car Type: Utility Vehicle
Ferrari Car Type: Sports Car

```

9.2 Properties, Fields, and Methods

A class in a C# program can contain a mix of properties, fields, and methods, each serving a different purpose in the class. It is important to understand the differences between them in order to use them effectively in the class.

9.2.1 Difference between Properties and Fields

Properties are similar to fields as both contain values that can be accessed. However, there are certain differences between them.

Table 9.1 lists the differences between properties and fields.

Properties	Fields
Properties are data members that can assign and retrieve values.	Fields are data members that store values.
Properties cannot be classified as variables and therefore, cannot use the <code>ref</code> and <code>out</code> keywords.	Fields are variables that can use the <code>ref</code> and <code>out</code> keywords.
Properties are defined as a series of executable statements.	Fields can be defined in a single statement.
Properties are defined with two accessors or methods, the <code>get</code> and <code>set</code> accessors.	Fields are not defined with accessors.
Properties can perform custom actions on change of the field's value.	Fields are not capable of performing any customized actions.

Table 9.1: Differences between Properties and Fields

9.2.2 Properties versus Methods

The implementation of properties covers both, the implementation of fields and the implementation of methods. This is because properties contain two special methods and are invoked in a similar manner as fields. There are a few differences between properties and methods as listed in table 9.2.

Properties	Methods
Properties represent characteristics of an object.	Methods represent the behavior of an object.
Properties contain two methods which are automatically invoked without specifying their names.	Methods are invoked by specifying method names along with the object of the class.
Properties cannot have any parameters.	Methods can include a list of parameters.
Properties can be overridden but cannot be overloaded.	Methods can be overridden as well as overloaded.

Table 9.2: Differences between Properties and Methods

9.3 Indexers

In a C# program, indexers allow instances of a class or struct to be indexed like arrays. Indexers are syntactically similar to properties, but unlike properties, the accessors of indexers accept one or more parameters.

9.3.1 Purpose of Indexers

Consider a high school teacher who wants to go through the records of a particular student to check the student's progress. If the teacher calls the appropriate methods every time to set and get a particular record, the task becomes a little tedious. On the other hand, if the teacher creates an indexer for student ID, it makes the task of accessing the record much easier. This is because indexers use index position of the student ID to locate the student record.

Figure 9.1 displays the index position of indexers for this example.

Indexer		Student_Details	
	StudentID	StudentID	StudentName
	S001	S003	John
	S002	S002	Smith
	S003	S004	Albert
	S004	S001	Rosa

Figure 9.1: Index Position of Indexers

9.3.2 Definition of Indexers

Indexers are data members that allow you to access data within objects in a way that is similar to accessing arrays. Indexers provide faster access to the data within an object as they help in indexing the data. In arrays, you use the index position of an object to access its value. Similarly, an indexer allows you to use the index of an object to access the values within the object.

The implementation of indexers is similar to properties, except that the declaration of an indexer can contain parameters. In C#, indexers are also known as **smart arrays**.

9.3.3 Declaration of Indexers

Indexers allow you to index a class, struct, or an interface. An indexer can be defined by specifying the following:

- An access modifier, which decides the scope of the indexer.
- The return type of the indexer, which specifies the type of value an indexer, will return.
- The `this` keyword, which refers to the current instance of the current class.
- The bracket notation (`[]`), which consists of the data type and identifier of the index.
- The open and close curly braces, which contain the declaration of the `set` and `get` accessors.

The following syntax creates an indexer.

Syntax:

```
<access_modifier><return_type>this [<parameter>]
{
get
{
// return value
}
set
{
// assign value
}
}
```

where,

access_modifier: Determines the scope of the indexer, which can be **private**, **public**, **protected**, or **internal**.

return_type: Determines the type of value an indexer will return.

parameter: Is the parameter of the indexer.

Code Snippet 12 demonstrates the use of indexers.

Code Snippet 12:

```
class EmployeeDetails
{
    public string[] empName = new string[2];
    public string this[int index]
    {
        get
        {
            return empName[index];
        }
        set
        {
```

```

        empName[index] = value;
    }
}

static void Main(string[] args)
{
    EmployeeDetails objEmp = new EmployeeDetails();
    objEmp[0] = "Jack Anderson";
    objEmp[1] = "Kate Jones";
    Console.WriteLine("Employee Names : ");
    for (int i=0; i<2; i++)
    {
        Console.Write(objEmp[i] + "\t");
    }
}
}

```

In Code Snippet 12, the class **EmployeeDetails** creates an indexer that takes a parameter of type **int**. The instance of the class, **objEmp**, is assigned values at each index position. The **set** accessor is invoked for each index position. The **for** loop iterates for two times and displays values assigned at each index position using the **get** accessor.

Output:

Employee Names : Jack Anderson Kate Jones

9.3.4 Parameters

Indexers must have at least one parameter. The parameter denotes the index position, using which the stored value at that position is set or accessed. This is similar to setting or accessing a value in a single-dimensional array. However, indexers can also have multiple parameters. Such indexers can be accessed like a multi-dimensional array.

When accessing arrays, you need to mention the object name followed by the array name. Then, the value can be accessed by specifying the index position. However, indexers can be accessed directly by specifying the index number along with the instance of the class.

9.3.5 Implementing Inheritance

Indexers can be inherited like other members of the class. This means that the base class indexers can be inherited by the derived class.

Code Snippet 13 demonstrates the implementation of inheritance with indexers.

Code Snippet 13:

```
using System;

class Numbers
{
    private int[] num = new int[3];
    public int this[int index]
    {
        get
        {
            return num[index];
        }
        set
        {
            num[index] = value;
        }
    }
}

class EvenNumbers : Numbers
{
    public static void Main()
    {
        EvenNumbers objEven = new EvenNumbers();
        objEven[0] = 0;
        objEven[1] = 2;
        objEven[2] = 4;
        for (int i = 0; i < 3; i++)
        {
    }
```

```
        Console.WriteLine(objEven[i]);  
    }  
}  
}
```

In Code Snippet 13, the class **Numbers** creates an indexer that takes a parameter of type `int`. The class **EvenNumbers** inherits the class **Numbers**. The `Main()` method creates an instance of the derived class **EvenNumbers**. When this instance is assigned values at each index position, the `set` accessor of the indexer defined in the base class **Numbers** is invoked for each index position. The `for` loop iterates three times and displays values assigned at each index position using the accessor. Thus, by inheriting, an indexer in the base class can be reused in the derived class.

Output:

0
2
4

9.3.6 Implementing Polymorphism

Indexers can implement polymorphism by overriding the base class indexers or by overloading indexers. By implementing polymorphism, a programmer allows the derived class indexers to override the base class indexers. In addition, a particular class can include more than one indexer having different signatures. This feature of polymorphism is called as **overloading**. Thus, polymorphism allows the indexer to function with different data types of C# and generate customized output.

Code Snippet 14 demonstrates the implementation of polymorphism with indexers by overriding the base class indexers.

Code Snippet 14:

```
using System;  
  
class Student  
{  
  
    string[] studName = new string[2];  
  
    public virtual string this[int index]  
    {  
  
        get  
        {  
  
            return studName[index];  
        }  
    }  
}
```

```
set
{
    studName[index] = value;
}

}

class Result : Student
{
    string[] result = new string[2];
    public override string this[int index]
    {
        get
        {
            return base[index];
        }
        set
        {
            base[index] = value;
        }
    }
    static void Main(string[] args)
    {
        Result objResult = new Result();
        objResult[0] = "First";
        objResult[1] = "Pass";
        Student objStudent = new Student();
        objStudent[0] = "Peter";
        objStudent[1] = "Patrick";
        for (int i = 0; i < 2; i++)
        {
```

```
        Console.WriteLine(objStudent[i] + "\t\t" + objResult[i] + " class");
    }
}
}
```

In Code Snippet 14, the class **Student** declares an array variable and a virtual indexer. The class **Result** inherits the class **Student** and overrides the virtual indexer. The **Main()** method declares an instance of the base class **Student** and the derived class **Result**. When the instance of the class **Student** is assigned values at each index position, the **set** accessor of the class **Student** is invoked. When the instance of the class **Result** is assigned values at each index position, the **set** accessor of the class **Result** is invoked. This overrides the base class indexer.

The set accessor of the derived class **Result** invokes the base class indexer by using the `base` keyword. The `for` loop displays values at each index position by invoking the `get` accessors of the appropriate classes.

Output:

Peter First class

Patrick Pass class

9.3.7 Multiple Parameters in Indexers

Indexers must be declared with at least one parameter within the square bracket notation ([]). However, indexers can include multiple parameters. An indexer with multiple parameters can be accessed like a multi-dimensional array. A parameterized indexer can be used to hold a set of related values. For example, it can be used to store and change values in multi-dimensional arrays.

Code Snippet 15 demonstrates how multiple parameters can be passed to an indexer.

Code Snippet 15:

```
using System;

class Account

{
    string[,] accountDetails = new string[4, 2];

    public string this[int pos, int column]
    {
        get
        {
            return (accountDetails[pos, column]);
        }
    }
}
```

```
set
{
    accountDetails[pos, column] = value;
}
}

static void Main(string[] args)
{
    Account objAccount = new Account();
    string[] id = new string[3] { "1001", "1002", "1003" };
    string[] name = new string[3] { "John", "Peter", "Patrick" };
    int counter = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 1; j++)
        {
            objAccount[i, j] = id[counter];
            objAccount[i, j + 1] = name[counter++];
        }
    }
    Console.WriteLine("ID Name");
    Console.WriteLine();
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            Console.Write(objAccount[i, j] + " ");
        }
        Console.WriteLine();
    }
}
```

In Code Snippet 15, the class **Account** creates an array variable **accountDetails** having 4 rows and 2 columns. A parameterized indexer is defined to enter values in the array **accountDetails**. The indexer takes two parameters, which defines the positions of the values that will be stored in an array. The **Main()** method creates an instance of the **Account** class. This instance is used to enter values in the **accountDetails** array using a **for** loop. This invokes the **set** accessor of the indexer which assigns value in the array. A **for** loop displays the customer ID and name that is stored in an array which invokes the **get** accessor.

Output:

```
ID Name
1001 John
1002 Peter
1003 Patrick
```

9.3.8 Indexers in Interfaces

Indexers can also be declared in interfaces. However, the accessors of indexers declared in interfaces differ from the indexers declared within a class. The set and get accessors declared within an interface do not use access modifiers and do not contain a body. An indexer declared in the interface must be implemented in the class implementing the interface. This enforces reusability and provides the flexibility to customize indexers.

Code Snippet 16 demonstrates the implementation of interface indexers.

Code Snippet 16:

```
using System;
public interface IDetails
{
    string this[int index]
    {
        get;
        set;
    }
}
class Students : IDetails
{
    string [] studentName = new string[3];
    int [] studentID = new int[3];
```

```

public string this[int index]
{
    get
    {
        return studentName[index];
    }
    set
    {
        studentName[index] = value;
    }
}
static void Main(string[] args)
{
    Students objStudent = new Students();
    objStudent[0] = "James";
    objStudent[1] = "Wilson";
    objStudent[2] = "Patrick";
    Console.WriteLine("Student Names");
    Console.WriteLine();
    for (int i = 0; i < 3; i++)
    {
        Console.WriteLine(objStudent[i]);
    }
}
}

```

In Code Snippet 16, the interface **IDetails** declares a read-write indexer. The **Students** class implements the **IDetails** interface and implements the indexer defined in the interface. The **Main()** method creates an instance of the **Students** class and assigns values at different index positions. This invokes the **set** accessor. The **for** loop displays the output by invoking the **get** accessor of the indexer.

9.3.9 Difference between Properties and Indexers

Indexers are syntactically similar to properties. However, there are certain differences between them.

Table 9.3 lists the differences between properties and indexers.

Properties	Indexers
Properties are assigned a unique name in their declaration.	Indexers cannot be assigned a name and use the <code>this</code> keyword in their declaration.
Properties are invoked using the specified name.	Indexers are invoked through an index of the created instance.
Properties can be declared as <code>static</code> .	Indexers can never be declared as <code>static</code> .
Properties are always declared without parameters.	Indexers are declared with at least one parameter.
Properties cannot be overloaded.	Indexers can be overloaded.
Overridden properties are accessed using the syntax <code>base.Prop</code> , where <code>Prop</code> is the name of the property.	Overridden indexers are accessed using the syntax <code>base[indExp]</code> , where <code>indExp</code> is the list of parameters separated by commas.

Table 9:3: Difference between Properties and Indexers

9.4 Check Your Progress

1. You are trying to display the output of the balance amount as “1005.50” using properties. Which of the following codes will help you to achieve this?

	<pre>class Balance { private double _balanceAmount; public double BalanceAmount { get { return value; } set { _balanceAmount = value; } } static void Main(string [] args) { Balance objBal = new Balance(); objBal.BalanceAmount = 1005.50; Console.WriteLine(objBal.BalanceAmount); } }</pre>
(B)	<pre>class Balance { private double _balanceAmount; public double BalanceAmount(double value) { get</pre>

	<pre>{ return _balanceAmount; } set { _balanceAmount = value; } (B) } static void Main(string [] args) { Balance objBal = new Balance(); objBal.BalanceAmount(1005.50); Console.WriteLine(objBal.BalanceAmount); } }</pre>
(C)	<pre>class Balance { private double _balanceAmount; public double BalanceAmount { get { return _balanceAmount; } set { _balanceAmount = value; } } static void Main(string [] args) {</pre>

(C)	<pre>Balance objBal = new Balance(); objBal.BalanceAmount = 1005.50; Console.WriteLine(objBal.BalanceAmount); }</pre>
(D)	<pre>class Balance { private double _balanceAmount; public BalanceAmount { get { return _balanceAmount; } set { _balanceAmount = value; } } static void Main(string [] args) { Balance objBal = new Balance(); objBal.BalanceAmount = 1005.50; Console.WriteLine(objBal.BalanceAmount); } }</pre>

(A)	A	(C)	C
(B)	B	(D)	D

2. Which of these statements about the property accessors and the types of properties are true?

(A)	The read-only property can be defined using the <code>set</code> accessor.
(B)	The write-only property can be defined using the <code>get</code> accessor.
(C)	The <code>get</code> accessor can be executed by referring to the name of the property.
(D)	The <code>set</code> accessor can be executed when the property is assigned a new value.
(E)	The <code>get</code> accessor can be declared using a parameter called <code>value</code> .

(A)	A, B	(C)	C, D
(B)	B	(D)	D

3. Match the terms in C# against their corresponding descriptions.

Description		Term	
(A)	Can be overridden and overloaded.	(1)	Fields
(B)	Can be defined as a single executable statement.	(2)	Properties
(C)	Can be defined with accessors used to assign and retrieve values.	(3)	Methods
(D)	Can be overridden but cannot be overloaded.		
(E)	Cannot be defined using the <code>ref</code> and <code>out</code> keywords.		

(A)	A-1, B-1, C-2, D-3, E-3	(C)	A-3, B-1, C-2, D-1, E-1
(B)	A-3, B-1, C-2, D-2, E-2	(D)	A-1, B-2, C-3, D-3, E-3

4. You are trying to display the output of employee name and employee ID as “James” and “10” using properties, fields and methods. Which of the following codes will help you to achieve this?

(A)

```
class EmployeeDetails
{
    private string _empName;
    private int _empID;
    public string EmpName
    {
        get
        {
            return _empName;
        }
        set
        {
            _empName = value;
        }
    }
    public static void SetId(int val)
    {
        _empId = val;
    }
    static void Main(string[] args)
    {
        EmployeeDetails objDetails = new EmployeeDetails();
        objDetails.EmpName = "James";
        objDetails.SetId(10);
        Console.WriteLine("Employee Name: " +
        objDetails.EmpName);
        Console.WriteLine("Employee ID: " + objDetails._empId);
    }
}
```

(B)

```
class EmployeeDetails
{
    private string _empName;
    private int _empId;
    public string EmpName
    {
        get
        {
            return value;
        }
        set
        {
            _empName = value;
        }
    }
    public void SetId(int val)
    {
        _empId = val;
    }
    static void Main(string[] args)
    {
        EmployeeDetails objDetails = new EmployeeDetails();
        objDetails.EmpName = "James";
        objDetails.SetId(10);
        Console.WriteLine("Employee Name: " +
            objDetails.EmpName);
        Console.WriteLine("Employee ID: " + objDetails._empId);
    }
}
```

```
class EmployeeDetails
{
    private string _empName;
    private int _empId;
    public string EmpName
    {
        get()
        {
            return _empName;
        }
        set()
        {
            _empName = value;
        }
    }
    public void SetId(int val)
    {
        _empId = val;
    }
    static void Main(string[] args)
    {
        EmployeeDetails objDetails = new EmployeeDetails();
        objDetails.EmpName = "James";
        objDetails.SetId(10);
        Console.WriteLine("Employee Name: " +
            objDetails.EmpName);
        Console.WriteLine("Employee ID: " + objDetails._empId);
    }
}
```

```
(D) class EmployeeDetails
{
    private string _empName;
    private int _empId;
    public string EmpName
    {
        get()
        {
            return _empName;
        }
        set()
        {
            _empName = value;
        }
    }
    public void SetId(int val)
    {
        _empId = val;
    }
    static void Main(string[] args)
    {
        EmployeeDetails objDetails = new EmployeeDetails();
        objDetails.EmpName = "James";
        objDetails.SetId(10);
        Console.WriteLine("Employee Name: " +
            objDetails.EmpName);
        Console.WriteLine("Employee ID: " + objDetails._empId);
    }
}
```

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of these statements about indexers are true?

(A)	Indexers cannot be overloaded.
(B)	Indexers can be declared as static.
(C)	Indexers can be overridden.
(D)	Indexers may or may not contain parameters.

(A)	A	(C)	C
(B)	B	(D)	D

6. You are trying to display the output of the different products as 'Mouse', 'Keyboard', and 'Speakers' using indexers. Which of the following codes will help you to achieve this?

(A)

```
class Products
{
    private string[] productName = new string[3];
    public string this[int index]
    {
        get
        {
            return productName[index];
        }
        set
        {
            productName[index] = value;
        }
    }
    static void Main(string[] args)
    {
        Products objProduct = new Products();
        objProduct[0] = "Mouse";
        objProduct[1] = "KeyBoard";
        objProduct[2] = "Speakers";
        Console.WriteLine("Product Name");
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(objProduct[i]);
        }
    }
}
```

```
class Products
{
    private string[] productName = new string[3];
    public string Products(int index)
    {
        get
        {
            return productName[index];
        }
        set
        {
            productName[index] = value;
        }
    }
}

static void Main(string[] args)
{
    Products objProduct = new Products();
    objProduct[0] = "Mouse";
    objProduct[1] = "KeyBoard";
    objProduct[2] = "Speakers";
    Console.WriteLine("Product Name");
    for (int i = 0; i < 3; i++)
    {
        Console.WriteLine(objProduct[i]);
    }
}
```

```
class Products
{
    private string[] _productName = new string[3];
    public string this[int index]
    {
        get
        {
            return value[index];
        }
        set
        {
            _productName[index] = value;
        }
    }
    static void Main(string[] args)
    {
        Products objProduct = new Products();
        objProduct[0] = "Mouse";
        objProduct[1] = "KeyBoard";
        objProduct[2] = "Speakers";
        Console.WriteLine("Product Name");
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(objProduct[i]);
        }
    }
}
```

```

class Products
{
    private string[] _productName = new string[3];
    public this[int index]
    {
        get
        {
            return _productName(index);
        }
        set
        {
            _productName[index] = value;
        }
    }
    static void Main(string[] args)
    {
        Products objProduct = new Products();
        objProduct[0] = "Mouse";
        objProduct[1] = "KeyBoard";
        objProduct[2] = "Speakers";
        Console.WriteLine("Product Name");
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(objProduct[i]);
        }
    }
}

```

(A)	A	(C)	C
(B)	B	(D)	D

9.4.1 Answers

1.	C
2.	C
3.	B
4.	D
5.	C
6.	A



Summary

- ➔ Properties protect the fields of the class while accessing them.
- ➔ Property accessors enable you to read and assign values to fields.
- ➔ A field is a data member that stores some information.
- ➔ Properties enable you to access the private fields of the class.
- ➔ Methods are data members that define a behavior performed by an object.
- ➔ Indexers treat an object like an array, thereby providing faster access to data within the object.
- ➔ Indexers are syntactically similar to properties, except that they are defined using the this keyword along with the bracket notation ([]).



Session -10

Namespaces

Welcome to the Session, **Namespaces**.

A namespace is an element of the C# program that helps in organizing classes, interfaces, and structures. Namespaces avoid name clashes between the C# classes, interfaces and structures. You can create multiple namespaces within a namespace.

In this session, you will learn to:

- ➔ Define and describe namespaces
- ➔ Explain nested namespaces

10.1 Namespaces

A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names. Namespaces reduce any complexities when the same program is required in another application.

10.1.1 Introduction to Namespaces - Purpose

Consider Venice, which is a city in the U.S. as well as in Italy. You can easily distinguish between the two cities by associating them with their respective countries.

Similarly, when working on a huge project, there may be situations where classes have identical names. This may result in name conflicts. This problem can be solved by having the individual modules of the project use separate namespaces to store their respective classes. By doing this, classes can have identical names without any resultant name clashes.

Code Snippet 1 renames identical classes by inserting a descriptive prefix.

Code Snippet 1:

```
class SamsungTelevision
{
    ...
}

class SamsungWalkMan
{
    ...
}

class SonyTelevision
{
    ...
}

class SonyWalkMan
{
    ...
}
```

In Code Snippet 1, the identical classes **Television** and **WalkMan** are prefixed with their respective company names to avoid any conflicts. This is because there cannot be two classes with the same name. However, when this is done, it is observed that the names of the classes get long and become difficult to maintain.

Code Snippet 2 demonstrates a solution to overcome this, by using namespaces.

Code Snippet 2:

```
namespace Samsung
{
    class Television
    {
        ...
    }

    class WalkMan
    {
        ...
    }
}

namespace Sony
{
    class Television
    {
        ...
    }

    class Walkman
    {
        ...
    }
}
```

In Code Snippet 2, each of the identical classes is placed in their respective namespaces, which denote respective company names. It can be observed that this is a neater, better organized, and more structured way to handle naming conflicts.

10.1.2 Using Namespaces

C# allows you to specify a unique identifier for each namespace. This identifier helps you to access the classes within the namespace.

Apart from classes, the following data structures can be declared in a namespace.

→ **Interface**

An interface is a reference type that contains declarations of the events, indexers, methods, and properties. Interfaces are inherited by classes and structures and all the declarations are implemented in these classes and structures.

→ **Structure**

A structure is a value type that can hold values of different data types. It can include fields, methods, constants, constructors, properties, indexers, operators, and other structures.

→ **Enumeration**

An enumeration is a value type that consists of a list of named constants. This list of named constants is known as the enumerator list.

→ **Delegate**

A delegate is a user-defined reference type that refers to one or more methods. It can be used to pass data as parameters to methods.

10.1.3 Characteristics and Benefits

A namespace groups common and related classes, structures or interfaces, which support OOP concepts of encapsulation and abstraction. A namespace has the following characteristics:

- It provides a hierarchical structure that helps to identify the logic for grouping the classes.
- It allows you to add more classes, structures, enumerations, delegates and interfaces once the namespace is declared.
- It includes classes with names that are unique within the namespace.

A namespace provides the following benefits:

- It allows you to use multiple classes with same names by creating them in different namespaces.
- It makes the system modular.

10.1.4 Built-in Namespaces

The .NET Framework comprises several built-in namespaces that contain classes, interfaces, structures, delegates, and enumerations. These namespaces are referred to as system-defined namespaces. The most commonly used built-in namespace of the .NET Framework is System.

The System namespace contains classes that define value and reference data types, interfaces, and

other namespaces. In addition, it contains classes that allow you to interact with the system, including the standard input and output devices. Some of the most widely used namespaces within the `System` namespace are as follows:

→ **System.Collections**

The `System.Collections` namespace contains classes and interfaces that define complex data structures such as lists, queues, bit arrays, hash tables, and dictionaries.

→ **System.Data**

The `System.Data` namespace contains classes that make up the ADO.NET architecture.

The ADO.NET architecture allows you to build components that can be used to insert, modify and delete data from multiple data sources.

→ **System.Diagnostics**

The `System.Diagnostics` namespace contains classes that are used to interact with the system processes. This namespace also provides classes that are used to debug applications and trace the execution of the code.

→ **System.IO**

The `System.IO` namespace contains classes that enable you to read from and write to data streams and files.

→ **System.Net**

The `System.Net` namespace contains classes that allow you to create Web-based applications.

→ **System.Web**

The `System.Web` namespace provides classes and interfaces that allow communication between the browser and the server.

Figure 10.1 displays some built-in namespaces.

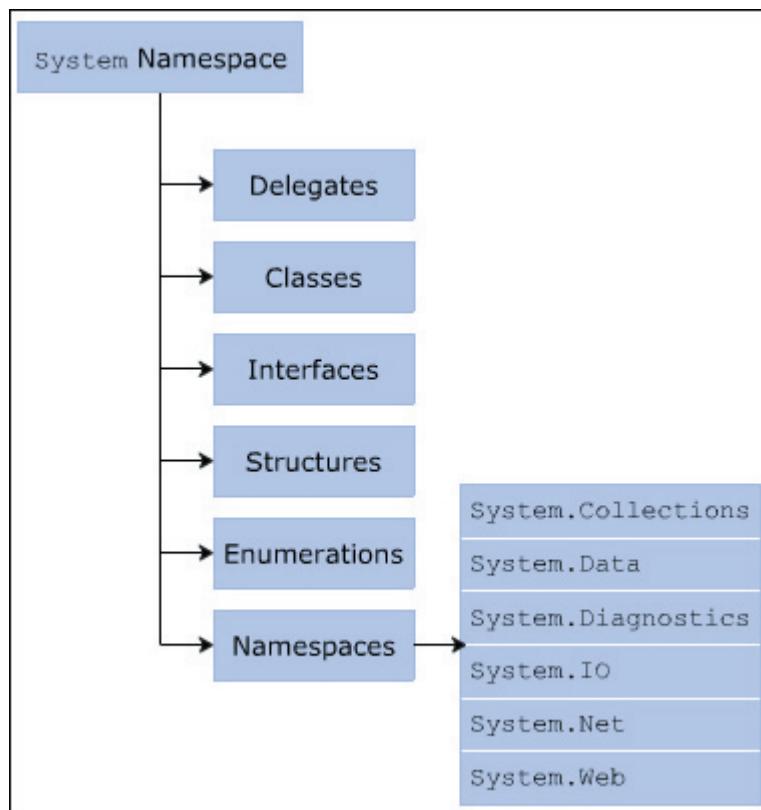


Figure 10.1: System-defined Namespaces

10.1.5 Using the System Namespace

The `System` namespace is imported by default in the .NET Framework. It appears as the first line of the program along with the `using` keyword. For referring to classes within a built-in namespace, you need to explicitly refer to the required classes. This is done by specifying the namespace and the class name separated by the dot (.) operator after the `using` keyword at the beginning of the program.

Alternatively, you can refer to classes within the namespaces in the same manner without the `using` keyword. However, this results in redundancy because you need to mention the whole declaration every time you refer to the class in the code.

Figure 10.2 displays the two approaches of using the `System` namespace.

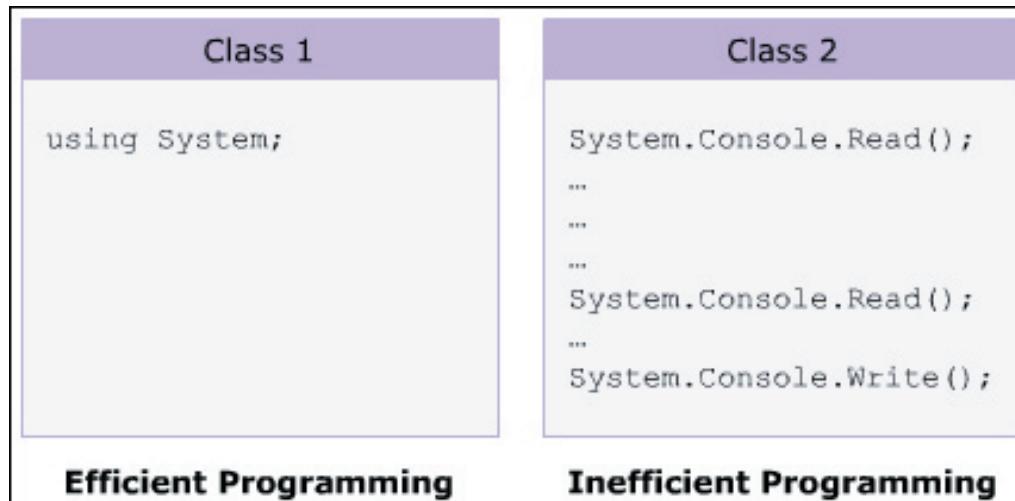


Figure 10.2: Using the System Namespace

The following syntax is used to access a method in a system-defined namespace.

Syntax:

`<NamespaceName>.<ClassName>.<MethodName>;`

where,

`NamespaceName`: Is the name of the namespace.

`ClassName`: Is the name of the class that you want to access.

`MethodName`: Is the name of the method within the class that is to be invoked.

The following syntax is used to access the system-defined namespaces with the `using` keyword.

Syntax:

`using <NamespaceName>;`

`using <NamespaceName>.<ClassName>;`

where,

`NamespaceName`: Is the name of the namespace and it will refer to all classes, interfaces, structures and enumerations.

`ClassName`: Is the name of the specific class defined in the namespace that you want to access.

Code Snippet 3 demonstrates the use of the `using` keyword while using namespaces.

Code Snippet 3:

```
using System;
class World
```

```
{
    static void Main(string[] args)
    {
        Console.WriteLine("HelloWorld");
    }
}
```

In Code Snippet 3, the `System` namespace is imported within the program with the `using` keyword. If this were not done, the program would not even compile as the `Console` class exists in the `System` namespace.

Output:

Hello World

Code Snippet 4 refers to the `Console` class of the `System` namespace multiple times. Here, the class is not imported but the `System` namespace members are used along with the statements.

Code Snippet 4:

```
class World
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("HelloWorld");
        System.Console.WriteLine("This is C# Programming");
        System.Console.WriteLine("You have executed a simple program of C#");
    }
}
```

Output:

Hello World

This is C# Programming

You have executed a simple program of C#

Note - The `using` keyword is used to globally declare the namespace to be used in the C# file. This feature allows you to directly use the classes within the namespace in the program.

10.1.6 Custom Namespaces

C# allows you to create namespaces with appropriate names to organize structures, classes, interfaces, delegates and enumerations that can be used across different C# applications. These user-defined namespaces are referred to as custom namespaces. When using a custom namespace, you need not worry about name clashes with classes, interfaces, and so on in other namespaces.

Custom namespaces enable you to control the scope of a class by deciding the appropriate namespace for the class. A custom namespace is declared using the `namespace` keyword and is accessed with the `using` keyword similar to any built-in namespace. Figure 10.3 displays a general example of using custom namespaces.

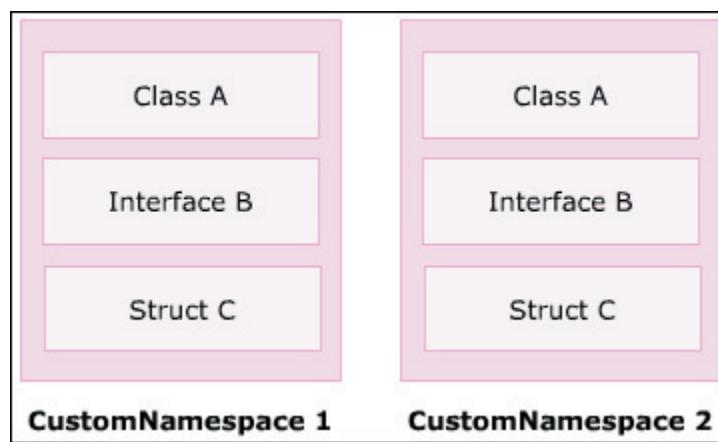


Figure 10.3: Custom Namespaces

The following syntax is used to declare a custom namespace.

Syntax:

```
namespace <NamespaceName>
{
    //type-declarations;
}
```

where,

`NamespaceName`: Is the name of the custom namespace.

`type-declarations`: Are the different types that can be declared. It can be a class, interface, struct, enum, delegate or another namespace.

Code Snippet 5 creates a custom namespace named **Department**.

Code Snippet 5:

```
namespace Department
{
```

```

class Sales
{
    static void Main(string [] args)
    {
        System.Console.WriteLine ("You have created a custom namespace named
        Department");
    }
}

```

In Code Snippet 5, **Department** is declared as the custom namespace. The class **Sales** is declared within this namespace.

Output:

You have created a custom namespace named Department

Note - If a namespace is not declared in a C# source file, the compiler creates a default namespace in every file. This unnamed namespace is referred to as the global namespace.

Once a namespace is created, C# allows additional classes to be included later in that namespace. Hence, namespaces are additive. C# also allows a namespace to be declared more than once. These namespaces can be split and saved in separate files or in the same file. At the time of compilation, these namespaces are added together.

The namespace split over multiple files is illustrated in the following Code Snippets 6, 7, and 8.

Code Snippet 6:

```

// The Automotive namespace contains the class SpareParts and this namespace is
partly stored in the SpareParts.cs file.

using System;
namespace Automotive
{
    public class SpareParts
    {
        string _spareName;
        public SpareParts()
        {
            _spareName = "Gear Box";
        }
    }
}

```

```
    }

    public void Display()
    {
        Console.WriteLine("Spare Part name: " + _spareName);
    }
}
```

Code Snippet 7:

```
//The Automotive namespace contains the class Category and this namespace is  
partly stored in the Category.cs file.
```

```
using System;  
  
namespace Automotive  
{  
  
    public class Category  
    {  
  
        string _category;  
  
        public Category()  
        {  
  
            _category = "Multi Utility Vehicle";  
        }  
  
        public void Display()  
        {  
  
            Console.WriteLine("Category: " + _category);  
        }  
    }  
}
```

Code Snippet 8:

```
//The Automotive namespace contains the class Toyota and this namespace is //  
partly stored in the Toyota.cs file.  
  
namespace Automotive  
{
```

```

class Toyota
{
    static void Main(string[] args)
    {
        Category objCategory = new Category();
        SpareParts objSpare = new SpareParts();
        objCategory.Display();
        objSpare.Display();
    }
}

```

The three classes, **SpareParts**, **Category**, and **Toyota** are stored in three different files, **SpareParts.cs**, **Category.cs**, and **Toyota.cs** respectively. Even though the classes are stored in different files, they are still in the same namespace, namely **Automotive**. Hence, a reference is not required here.

In these examples, a single namespace **Automotive** is used to enclose three different classes. The code for each class is saved as a separate file. When the three files are compiled, the resultant namespace is still **Automotive**.

Figure 10.4 shows the output of the application containing the three files.

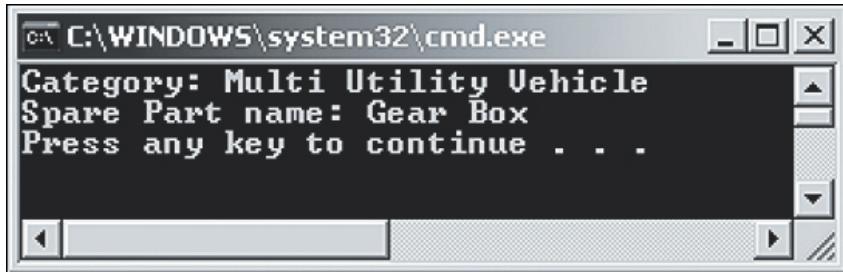


Figure 10.4: Output

10.1.7 Guidelines for Creating Custom Namespaces

While designing a large framework for a project, it is required to create namespaces to group the types into the appropriate namespaces such that the identical types do not collide. Therefore, the following guidelines must be considered for creating custom namespaces:

- All similar elements such as classes and interfaces must be created into a single namespace. This will form a logical grouping of similar types and any programmer will be easily able to search for similar classes.

- Creating deep hierarchies that are difficult to browse must be avoided.
- Creating too many namespaces must be avoided for simplicity.

Nested namespaces must contain types that depend on the namespace within which it is declared. For example, the classes in the **Country**.**States**.**Cities** namespace will depend on the classes in the namespace **Country**.**States**. For example, if a user has created a class called **us** in the **States** namespace, only the cities residing in U.S. can appear as classes in the **Cities** namespace.

10.1.8 Access Modifiers for Namespaces

Namespaces are always public. You cannot apply access modifiers such as **public**, **protected**, **private**, or **internal** to namespaces. This is because a namespace is accessible by any other namespace or class that exists outside the accessed namespace. Therefore, the access scope of a namespace cannot be restricted.

If any of the access modifiers is specified in the namespace declaration, the compiler generates an error.

Code Snippet 9 declares the namespace as **public**.

Code Snippet 9:

```
using System;
public namespace Products
{
    class Computers
    {
        static void Main(string [] args)
        {
            Console.WriteLine ("This class provides information about
Computers");
        }
    }
}
```

Code Snippet 9 generates the error, 'A namespace declaration cannot have modifiers or attributes'.

10.1.9 Unqualified Naming

A class defined within a namespace is accessed only by its name. To access this class, you just have to specify the name of that class. This form of specifying a class is known as **Unqualified naming**. The use of

unqualified naming results in short names and can be implemented by the `using` keyword. This makes the program simple and readable.

Code Snippet 10 displays the student's name, ID, subject, and marks scored using an unqualified name.

Code Snippet 10:

```
using System;
using Students;
namespace Students
{
    class StudentDetails
    {
        string _studName = "Alexander";
        int _studID = 30;
        public StudentDetails()
        {
            Console.WriteLine("Student Name: " + _studName);
            Console.WriteLine("Student ID: " + _studID);
        }
    }
}

namespace Examination
{
    class ScoreReport
    {
        public string Subject = "Science";
        public int Marks = 60;
        static void Main(string[] args)
        {
            StudentDetails objStudents = new
            StudentDetails();
            ScoreReport objReport = new ScoreReport();
            Console.WriteLine("Subject: " + objReport.Subject);
        }
    }
}
```

```
        Console.WriteLine("Marks: " + objReport.Marks);  
    }  
}  
}
```

In Code Snippet 10, the class **ScoreReport** uses the class **StudentDetails** defined in the namespace **Examination**. The class is accessed by its name.

10.1.10 Qualified Naming

C# allows you to use a class outside its namespace. A class outside its namespace can be accessed by specifying its namespace followed by the `.` operator and the class name. This form of specifying the class is known as **Fully Qualified naming**.

The use of fully qualified names results in long names and repetition throughout the code. Instead, you can access classes outside their namespaces with the `using` keyword. This makes the names short and meaningful.

Code Snippet 11 displays the student's name, ID, subject, and marks scored using a fully qualified name.

Code Snippet 11:

```
using System;  
  
namespace Students  
{  
  
    class StudentDetails  
    {  
  
        string _studName = "Alexander";  
  
        int _studId = 30;  
  
        public StudentDetails()  
        {  
  
            Console.WriteLine("Student Name: " + _studName);  
            Console.WriteLine("Student ID: " + _studId);  
  
        }  
  
    }  
  
}  
  
namespace Examination  
{
```

```

class ScoreReport
{
    public string Subject = "Science";
    public int Marks = 60;
    static void Main(string[] args)
    {
        Students.StudentDetails objStudents = new Students.StudentDetails();
        ScoreReport objReport = new ScoreReport();
        Console.WriteLine("Subject: " + objReport.Subject);
        Console.WriteLine("Marks: " + objReport.Marks);
    }
}
}

```

In Code Snippet 11, the class **ScoreReport** uses the class **StudentDetails** defined in the namespace **Examination**. The class is accessed by its fully qualified name.

10.1.11 Naming Conventions for Namespaces

When namespaces are being created to handle projects of an organization, it is recommended that namespaces are prefixed with the company name followed by the technology name, the feature, and the design of the brand. Namespaces for projects of an organization can be created as follows:

CompanyName.TechnologyName [.Feature] [.Design]

There are certain naming conventions which must be followed for creating namespaces. The conventions are:

- Use Pascal case for naming the namespaces.
- Use periods to separate the logical components.
- Use plural names for namespaces wherever applicable.
- Ensure that a namespace and a class do not have same names.
- Ensure that the name of a namespace is not identical to the name of the assembly.

10.2 Nested Namespaces

C# allows you to define namespaces within a namespace. This arrangement of namespaces is referred to as nested namespaces.

For an organization running multiple projects, nested namespaces is useful. The root namespace can be given the name of the organization and nested namespaces can be given the names of individual projects or modules. This allows the developers to store commonly used classes in appropriate namespaces and use them for all other similar programs.

Figure 10.5 illustrates the concept of nested namespaces.

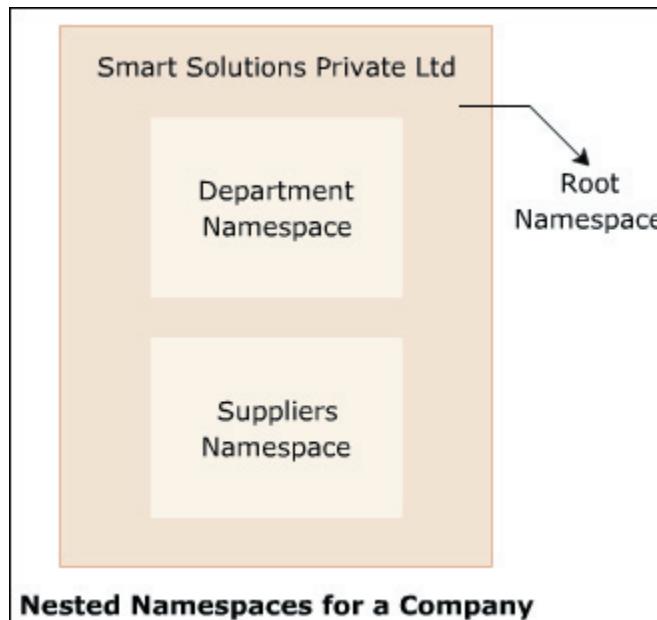


Figure 10.5: Nested Namespaces

10.2.1 Implementing Nested Namespaces

C# allows you to create a hierarchy of namespaces by creating namespaces within namespaces. Such nesting of namespaces is done by enclosing one namespace declaration inside the declaration of the other namespace.

The following syntax can be used to create nested namespaces.

Syntax:

```
namespace <NamespaceName>
{
    namespace <NamespaceName>
    {
    }
}
```

```
namespace <NamespaceName>
{
}
}
```

Code Snippet 12 creates nested namespaces.

Code Snippet 12:

```
namespace Contact
{
    public class Employees
    {
        public int EmpID;
    }

    namespace Salary
    {
        public class SalaryDetails
        {
            public double EmpSalary;
        }
    }
}
```

In Code Snippet 12, **Contact** is declared as a custom namespace that contains the class **Employees** and another namespace **Salary**. The **Salary** namespace contains the class **SalaryDetails**.

Code Snippet 13 displays the salary of an employee using the nested namespace that was created in Code Snippet 12.

Code Snippet 13:

```
using System;
class EmployeeDetails
{
    static void Main(string [] args)
    {
        Contact.Salary.SalaryDetails objSal = new Contact.Salary.SalaryDetails();
```

```

    objSal.EmpSalary=1000.50;
    Console.WriteLine("Salary: "+objSal.EmpSalary);
}
}

```

In the **EmployeeDetails** class of Code Snippet 13, the object of the **SalaryDetails** class is created using the namespaces in which the classes are declared. The value of the variable **EmpSalary** of the **SalaryDetails** class is initialized to 1000.5 and the salary amount is displayed as the output.

Output:

Salary: 1000.5

10.2.2 Namespace Aliases

Aliases are temporary alternate names that refer to the same entity. The namespace referred to with the **using** keyword refers to all the classes within the namespace. However, sometimes you might want to access only one class from a particular namespace. You can use an alias name to refer to the required class and to prevent the use of fully qualified names.

Aliases are also useful when a program contains many nested namespace declarations and you would like to distinguish as to which class belongs to which namespace. The alias would make the code more readable for other programmers and would make it easier to maintain.

Figure 10.6 displays an example of using namespace aliases.

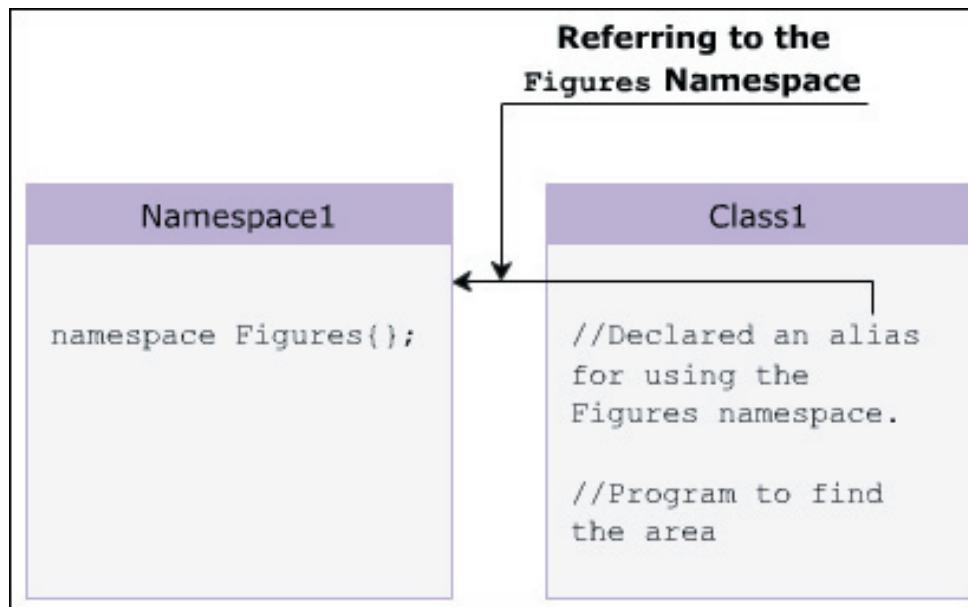


Figure 10.6: Namespace Aliases

The following syntax is used for creating a namespace alias.

Syntax:

```
using <aliasName> = <NamespaceName>;
```

where,

aliasName: Is the user-defined name assigned to the namespace.

Code Snippet 14 creates a custom namespace called **Bank.Accounts.EmployeeDetails**.

Code Snippet 14:

```
namespace Bank.Accounts.EmployeeDetails
{
    public class Employees
    {
        public string EmpName;
    }
}
```

Code Snippet 15 displays the name of an employee using the aliases of the **System.Console** and **Bank.Accounts.EmployeeDetails** namespaces.

Code Snippet 15:

```
using IO = System.Console;
using Emp = Bank.Accounts.EmployeeDetails;
class AliasExample
{
    static void Main (string[] args)
    {
        Emp.Employees objEmp = new Emp.Employees ();
        objEmp.EmpName = "Peter";
        IO.WriteLine ("Employee Name : " + objEmp.EmpName);
    }
}
```

In Code Snippet 15, the **Bank.Accounts.EmployeeDetails** is aliased as **Emp** and **System.Console** is aliased as **IO**.

These alias names are used in the **AliasExample** class to access the **Employees** and **Console** classes defined in the respective namespaces.

Output:

```
Employee Name: Peter
```

10.2.3 External Aliasing

Consider a large organization working on multiple projects. The programmers working on two different projects may use the same class name belonging to the same namespace and store them in different assemblies. The assemblies created can also contain similar method names. When these assemblies are used in a single program and a common method from these assemblies is invoked, compilation error is generated. The compilation error occurs since the same method resides in both the assemblies. This is called an ambiguous reference and it can be resolved by implementing external aliasing.

External aliasing in C# allows the users to define assembly qualified namespace aliases. It can be implemented using the `extern` keyword. This is illustrated in Code Snippet 16.

Code Snippet 16:

```
extern alias LibraryOne;
extern alias LibraryTwo;
using System;
class Companies
{
    static void Main (string [] args)
    {
        LibraryOne::Employees.Display();
        LibraryTwo::Employees.Display();
    }
}
```

In Code Snippet 16, the `Companies` class references to two different assemblies in which the `Employees` class is created with the `Display()` method. The external aliases for the two assemblies are defined as `LibraryOne` and `LibraryTwo`. During the compilation of this code, the aliases must be mapped to the path of the respective assemblies through the compiler options.

For example, you may write:

```
/reference: LibraryOne =One.dll
/reference: LibraryTwo =two.dll
```

10.2.4 Namespace Alias Qualifier

There are some situations wherein the alias provided to a namespace matches with the name of an existing namespace. Then, the compiler generates an error while executing the program that references to that namespace.

This is illustrated in Code Snippet 17.

Code Snippet 17:

```
//The following program is saved in the Automobile.cs file under the // Automotive project.

using System;
using System.Collections.Generic;
using System.Text;
using Utility_Vehicle.Car;
using Utility_Vehicle=Automotive.Vehicle.Jeep;

namespace Automotive
{
    namespace Vehicle
    {
        namespace Jeep
        {
            class Category
            {
                string _category;
                public Category()
                {
                    _category="Multi Utility Vehicle";
                }
                public void Display()
                {
                    Console.WriteLine("Jeep Category: " + _category);
                }
            }
        }
    }
}
```

```
    }
}
}

class Automobile
{
    static void Main(string[] args)
    {
        Category objCat = new Category();
        objCat.Display();
        Utility_Vehicle.Category objCategory = new Utility_Vehicle.Category();
        objCategory.Display();
    }
}
}
```

Another project is created under the **Automotive** project to store the program shown in Code Snippet 18.

Code Snippet 18:

```
//The following program is saved in the Category.cs file under the
//Utility_Vehicle project created under the Automotive project.

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

namespace Utility_Vehicle
{
    namespace Car
    {
        class Category
        {
    }
```

```
string _category;  
public Category()  
{  
    _category = "Luxury Vehicle";  
}  
public void Display()  
{  
    Console.WriteLine("Car Category: " + _category);  
}  
}
```

Both these files are compiled using the following syntax:

```
csc /out:<FileName>.exe <FullPath of File1>.cs <Full Path of File2>.cs
```

where,

FileName: Is the name of the single .exe file that will be generated.

FullPath of File 1: Is the complete path where the first file is located.

FullPath of File 2: Is the complete path where the second file is located.

Figure 10.7 shows the outcome of the compiled code.

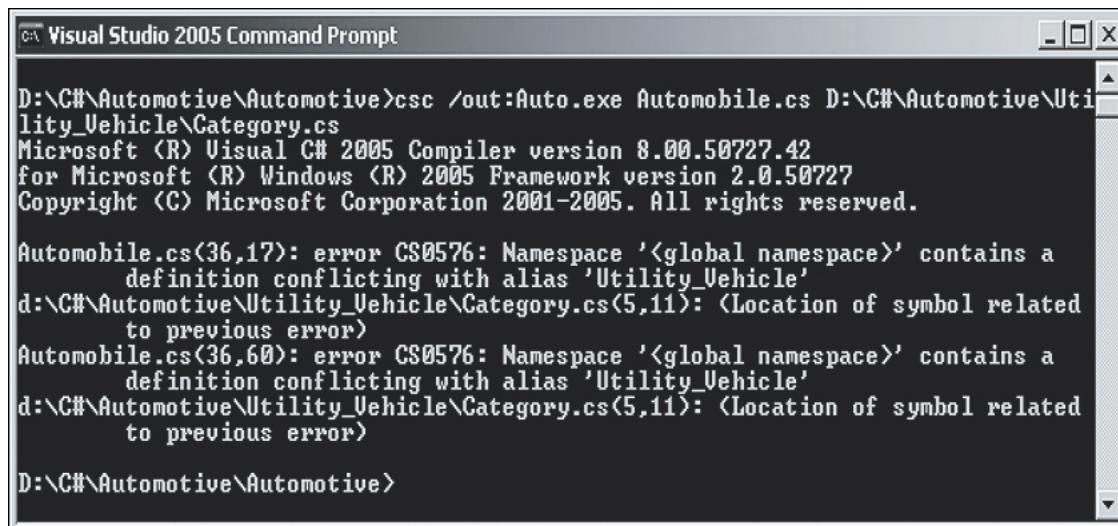


Figure 10.7: Outcome of the Compiled Code

In Code Snippets 17 and 18, the namespaces `Jeep` and `Car` include the class `Category`. An alias

Utility_Vehicle is provided to the namespace **Automotive.Jeep**. This alias matches with the name of the other namespace in which the namespace **Car** is nested. To compile both the programs, the csc command is used which uses the complete path to refer to the **Category.cs file**. In the **Automobile.cs** file, the alias name **Utility_Vehicle** is the same as the namespace which is being referred by the file. Due to this name conflict, the compiler generates an error.

This problem of ambiguous name references can be resolved by using the namespace alias qualifier.

Namespace alias qualifier is a new feature of C# and it can be used in the form of:

<LeftOperand> :: <RightOperand>

where,

LeftOperand: Is a namespace alias, an extern, or a global identifier.

RightOperand: Is the type.

The correct code for this is illustrated in Code Snippet 19.

Code Snippet 19:

```
using System;
using System.Collections.Generic;
using System.Text;
using Utility_Vehicle.Car;
using Utility_Vehicle=Automotive.Vehicle.Jeep;

namespace Automotive
{
    namespace Vehicle
    {
        namespace Jeep
        {
            class Category
            {
                string _category;
                public Category()
                {
                    _category="Multi Utility Vehicle";
                }
            }
        }
    }
}
```

```
}

public void Display()
{
    Console.WriteLine("Jeep Category: " + _category);
}

}

class Automobile
{
    static void Main(string[] args)
    {
        Category objCat = new Category();
        objCat.Display();

        Utility_Vehicle::Category objCategory = new Utility_Vehicle::
Category();
        objCategory.Display();
    }
}
```

The output of Code Snippet 19 is shown in figure 10.8.

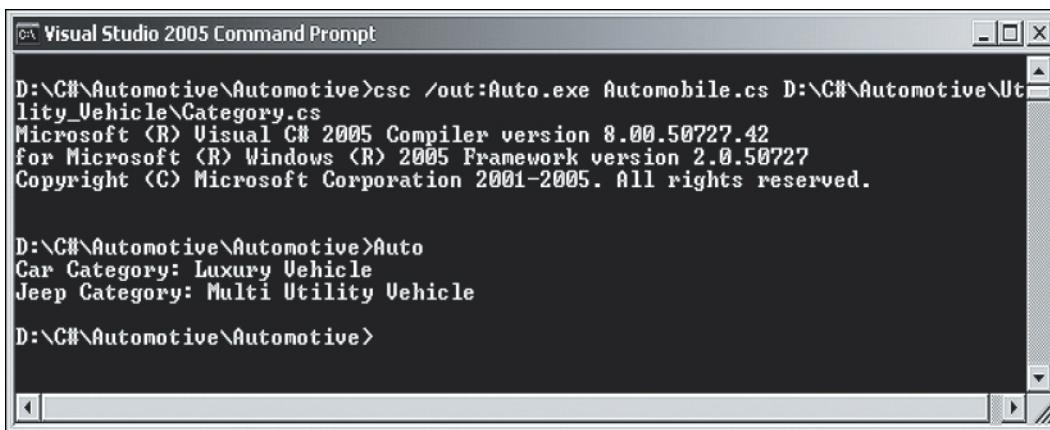


Figure 10.8: Output of Code Snippet 19

In Code Snippet 19, in the class **Automobile**, the namespace alias qualifier is used. The alias **Utility**

Vehicle is specified in the left operand and the class **Category** in the right operand.

Another situation where a namespace alias qualifier is used is when a user needs to define the class **System** in the custom namespace. C# does not allow users to create a class named **System**. If a class named **System** is created and a constant variable named **Console** is declared within this class, the compiler will generate an error on using the **System.Console** class. This is because C# includes the system defined namespace called **System**. This problem can be resolved by using namespace alias qualifier with its left operand defined as global.

This is illustrated in Code Snippet 20.

Code Snippet 20:

```
using System;
namespace ITCompany
{
    class System
    {
        const string Console = "Console";
        public static string WriteLine()
        {
            return "WriteLine method of my System class";
        }
        static void Main(string[] args)
        {
            global::System.Console.WriteLine (WriteLine());
        }
    }
}
```

In Code Snippet 20, the namespace alias qualifier is used in the class **System**. The left operand of the namespace alias qualifier is specified as global using the **global** keyword. Therefore, the search for the right operand starts at the global namespace.

Output:

WriteLine method of my System class

10.3 Check Your Progress

1. Which of these statements about namespaces in C# are true?

(A)	A namespace supports grouping of data structures.
(B)	A namespace supports OOP concepts of polymorphism and inheritance.
(C)	A namespace prevents class name conflicts.
(D)	The .NET Framework includes the namespace System.
(E)	Custom namespaces control scope of a class.

(A)	A, C, D, and E	(C)	A, C
(B)	A, D, and E	(D)	D

2. You are trying to display the output ‘Manufactures Laptops of different configurations.’ Which of the following codes will help you to achieve this?

(A)	<pre> using System; namespace Sony { class Laptops { public Laptops() { Console.WriteLine("Manufactures Laptops of different configurations"); } } class Players { public Players() { Console.WriteLine("CD as well as DVD players are manufactured"); } } } </pre>
-----	---

```
(A) }  
}  
}  
}  
  
namespace Dell  
  
{  
  
class Laptops  
  
{  
  
static void Main(string[] args)  
  
{  
  
Sony.Laptops objSonyLaptop = new Sony.Laptops();  
  
}  
}  
}  
}
```

```
public namespace Sony
{
    class Laptops
    {
        public Laptops()
        {
            Console.WriteLine("Manufactures Laptops of different
configurations");
        }
    }
}

class Players
{
    public Players()
    {
        Console.WriteLine("CD as well as DVD players are manufactured");
    }
}
```

(B)	<pre>namespace Dell { class Laptops { static void Main(string[] args) { Sony.Laptops objSonyLaptop = new Sony.Laptops(); } } }</pre>
(C)	<pre>namespace Sony { class Laptops { public Laptops() { Console.WriteLine("Manufactures Laptops of different configurations"); } } } class Players { public Players() { Console.WriteLine("CD as well as DVD players are manufactured"); } } namespace Dell {</pre>

(C)	<pre>class Laptops { static void Main(string[] args) { Laptops objSonyLaptop = new Laptops(); } }</pre>
(D)	<pre>namespace Sony { class Laptops { Laptops() { Console.WriteLine("Manufactures Laptops of different configurations"); } } class Players { Players() { Console.WriteLine("CD as well as DVD players are manufactured"); } } } namespace Dell { class Laptops</pre>

(D)	<pre> { static void Main(string[] args) { Sony.Laptops objSonyLaptop = new Sony.Laptops(); } } </pre>
-----	---

(A)	A	(C)	C
(B)	B	(D)	D

3. Match the commonly used system-defined namespaces in C# with their corresponding descriptions.

Description		Namespace	
(A)	Contains classes that provide a programming interface for network protocols.	(1)	System.Data
(B)	Contains classes that allow browser-server communication.	(2)	System.Diagnostics
(C)	Contains classes that make up the ADO.NET architecture.	(3)	System.IO
(D)	Contains classes that read from and write to data streams and files.	(4)	System.Net
(E)	Contains classes that can be used to interact with system processes.	(5)	System.Web

(A)	A-3, B-5, C-1, D-4, E-2	(C)	A-4, B-5, C-1, D-3, E-2
(B)	A-5, B-4, C-1, D-3, E-2	(D)	A-1, B-2, C-4, D-3, E-5

4. Which of these statements about nested namespaces and namespace alias in C# are true?

(A)	C# allows multiple namespaces to be declared within a single namespace.
(B)	Nested namespaces represent a hierarchical structure.
(C)	The declaration of nested namespaces involves creating multiple classes in a single namespace.

(D)	C# allows short names to be assigned to a longer namespace.
(E)	Namespace alias improves the readability of the code.

(A)	A	(C)	A, C
(B)	A, B, D, E	(D)	D

5. You are trying to display the following output:

“CD-R as well as CD-RW are available”

“DVD-R as well as DVD-RW are available”

Which of the following codes will help you to achieve this?

(A)	<pre>using System; using Play = Sony.Players.DvdPlayer; namespace Sony { namespace Players { class CdPlayer { public CdPlayer() { Console.WriteLine("CD-R as well as CD-RW are available"); } } class DvdPlayer { public DvdPlayer() { Console.WriteLine("DVD-R as well as DVD-RW are available"); } } } }</pre>
-----	--

	<pre> { public DvdPlayer() { Console.WriteLine("DVD-R as well as DVD-RW are available"); } } } </pre>
(A)	<pre> namespace Samsung { { class Players { static void Main(string[] args) { Play.DvdPlayer objPlayers = new Play.DvdPlayer(); } } } </pre>
(B)	<pre> using System; using Sony.Players = Play; namespace Sony { namespace Players { class CdPlayer } } </pre>

```
(B)   {
        public CdPlayer()
        {
            Console.WriteLine("CD-R as well as CD-RW are
available");
        }
    }

    class DvdPlayer
    {
        public DvdPlayer()
        {
            Console.WriteLine("DVD-R as well as DVD-RW are
available");
        }
    }
}

namespace Samsung
{
    class Players
    {
        static void Main(string[] args)
        {
            Play.DvdPlayer objPlayers = new Play.DvdPlayer ();
        }
    }
}
```

(C)

```
using System;
using Sony;
using Play = Sony.Players;

namespace Sony
{
    namespace Players
    {
        class CdPlayer
        {
            public CdPlayer()
            {
                Console.WriteLine("CD-R as well as CD-RW are
available");
            }
        }

        class DvdPlayer
        {
            public DvdPlayer()
            {
                Console.WriteLine("DVD-R as well as DVD-RW are
available");
            }
        }
    }
}

namespace Samsung
{
    class Players
```

(C)	<pre>static void Main(string[] args) { Play.CdPlayer objCd = new Play.CdPlayer(); Play.DvdPlayer objDvd = new Play.DvdPlayer(); }</pre>
(D)	<pre>using System; using Sony; using Play = Sony.Players; namespace Sony { namespace Players { class CdPlayer { public CdPlayer() { Console.WriteLine("CD-R as well as CD-RW are available"); } } class DvdPlayer { public DvdPlayer() { Console.WriteLine("DVD-R as well as DVD-RW are available"); } } } }</pre>

(D)

```
        }
    }

}

namespace Samsung
{
    class Players
    {
        static void Main(string[] args)
        {
            CdPlayer objCd = new CdPlayer();
            DvdPlayer objDvd = new DvdPlayer();
        }
    }
}
```

(A)	A	(C)	C
(B)	B	(D)	D

10.3.1 Answers

1.	A
2.	A
3.	C
4.	B
5.	C



Summary

- A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names.
- The System namespace is imported by default in the .NET Framework.
- Custom namespaces enable you to control the scope of a class by deciding the appropriate namespace for the class.
- You cannot apply access modifiers such as public, protected, private, or internal to namespaces.
- A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- C# supports nested namespaces that allows you to define namespaces within a namespace.
- External aliasing in C# allows the users to define assembly qualified namespace aliases.



Session -11

Exception Handling

Welcome to the Session, **Exception Handling**.

Exceptions are run-time errors that disrupt the execution flow of instructions in a program. In C#, you can handle these exceptions by using the try-catch or try-catch-finally constructs. In addition, C# allows you to define custom exceptions that allow you to customize the error-handling process.

In this session, you will learn to:

- ➔ Define and describe exceptions
- ➔ Explain the process of throwing and catching exceptions
- ➔ Explain nested try and multiple catch blocks
- ➔ Define and describe custom exceptions

11.1 Exceptions

Exceptions are run-time errors that may cause a program to be abruptly terminated. Exception handling is a process of handling these run-time errors. Handling an exception refers to the action to be taken when an error occurs in order to save the program from being prematurely terminated.

11.1.1 Purpose

Exceptions are abnormal events that prevent a certain task from being completed successfully. For example, consider a vehicle that halts abruptly due to some problem in the engine. Now, until this problem is sorted out, the vehicle may not move ahead. Similarly, in C#, exceptions disrupt the normal flow of the program.

11.1.2 Exceptions in C#

Consider a C# application that is currently being executed. Assume that at some point of time, the CLR discovers that it does not have the read permission to read a particular file. The CLR immediately stops further processing of the program and terminates its execution abruptly. To avoid this from happening, you can use the exception handling features of C#.

11.1.3 Types of Exceptions in C#

C# can handle different types of exceptions using exception handling statements. It allows you to handle basically two kinds of exceptions. These are as follows:

→ System-level Exceptions

System-level exceptions are the exceptions thrown by the system. These exceptions are thrown by the CLR. For example, exceptions thrown due to failure in database connection or network connection are system-level exceptions.

→ Application-level Exceptions

Application-level exceptions are thrown by user-created applications. For example, exceptions thrown due to arithmetic operations or referencing any null object are application-level exceptions.

Figure 11.1 displays the types of exceptions in C#.

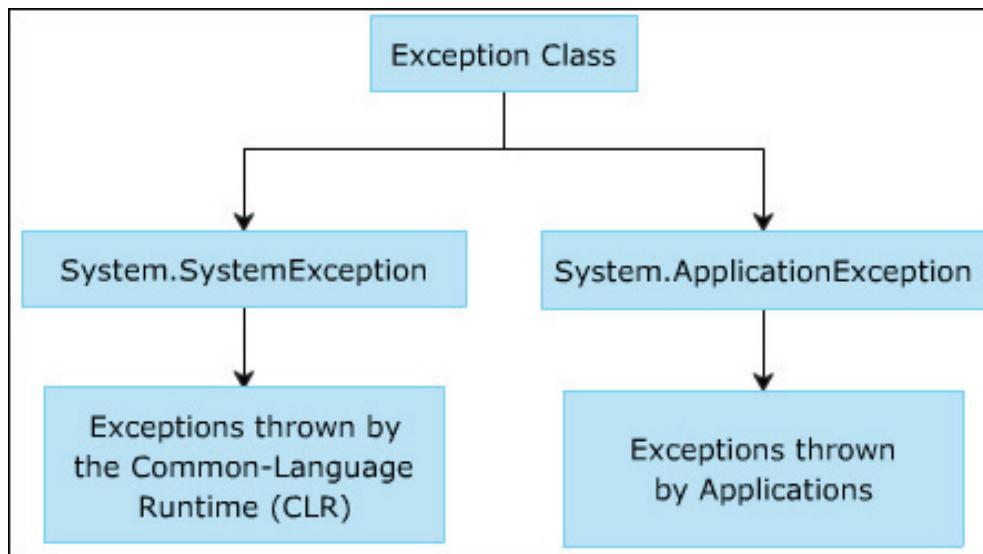


Figure 11.1: Exceptions in C#

11.1.4 The Exception Class

The `System.Exception` class is the base class that allows you to handle all exceptions in C#. This means that all exceptions in C# inherit the `System.Exception` class either directly or indirectly.

The `System.Exception` class contains public and protected methods that can be inherited by other exception classes. In addition, the `System.Exception` class contains properties that are common to all exceptions. Table 11.1 describes the properties of exception class.

Properties	Descriptions
Message	Displays a message which indicates the reason for the exception.
Source	Provides the name of the application or the object that caused the exception.
StackTrace	Provides exception details on the stack at the time the exception was thrown.
InnerException	Returns the <code>Exception</code> instance that caused the current exception.

Table 11.1: Properties of Exception

11.1.5 Commonly Used Exception Classes

The `System.Exception` class has a number of derived exception classes to handle different types of exceptions.

The hierarchy shown in figure 11.2 displays the different exception classes in the `System` namespace.

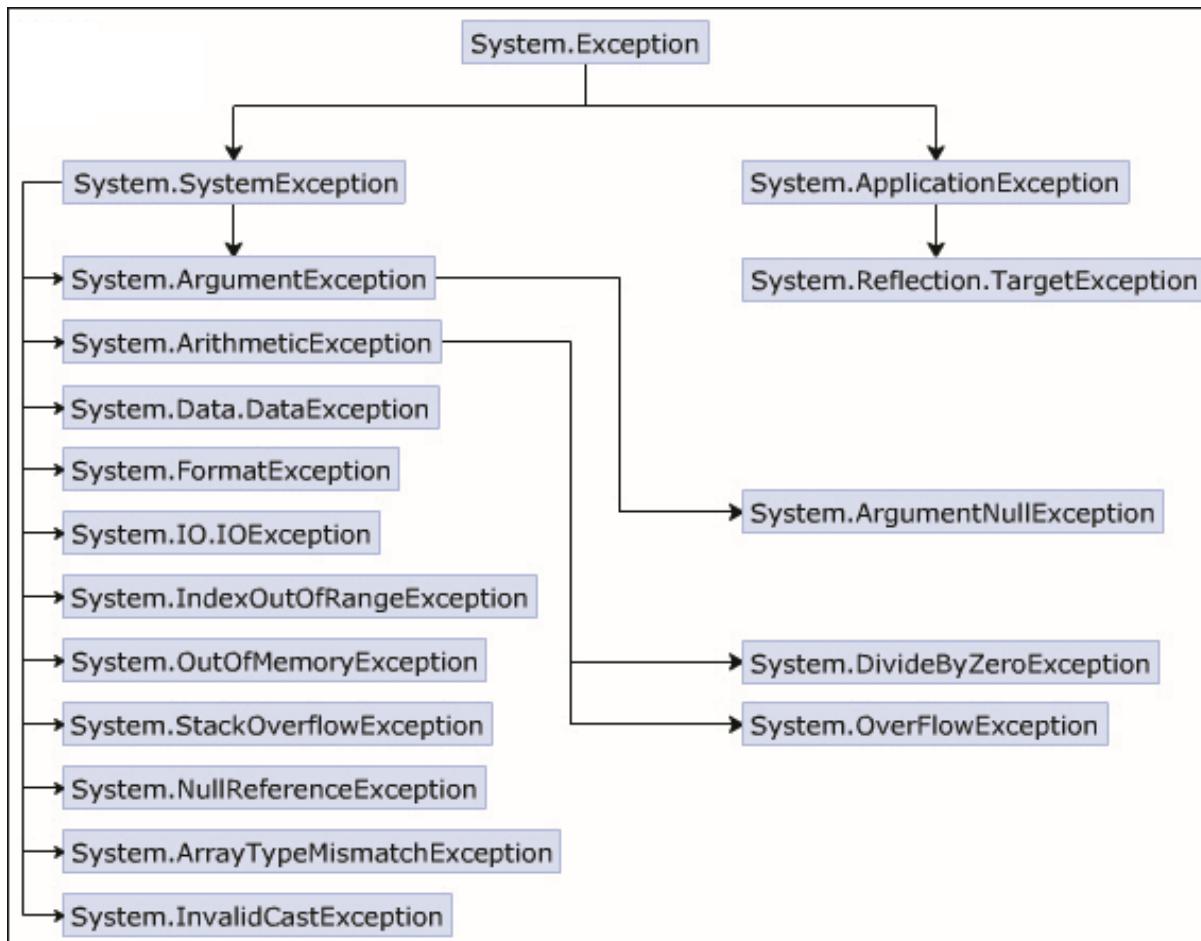


Figure 11.2: Commonly Used Exceptions

11.1.6 Exception Classes

The `System` namespace contains the different exception classes that C# provides. The type of exception to be handled depends on the specified exception class. Table 11.2 lists some of the commonly used exception classes.

Exceptions	Descriptions
<code>System.ArithmetiException</code>	This exception is thrown for problems that occur due to arithmetic or casting and conversion operations.
<code>System.ArgumentException</code>	This exception is thrown when one of the arguments does not match the parameter specifications of the invoked method.
<code>System.ArrayTypeMismatchException</code>	This exception is thrown when an attempt is made to store data in an array whose type is incompatible with the type of the array.

Exceptions	Descriptions
System.DivideByZeroException	This exception is thrown when an attempt is made to divide a numeric value by zero.
System.IndexOutOfRangeException	This exception is thrown when an attempt is made to store data in an array using an index that is less than zero or outside the upper bound of the array.
System.InvalidCastException	This exception is thrown when an explicit conversion from the base type or interface type to another type fails.
System.ArgumentNullException	This exception is thrown when a null reference is passed to an argument of a method that does not accept null values.

Table 11.2: Commonly Used Exception Classes

Table 11.3 lists the other exception classes that are most commonly used.

Exceptions	Descriptions
System.NullReferenceException	This exception is thrown when you try to assign a value to a null object.
System.OutOfMemoryException	This exception is thrown when there is not enough memory to allocate to an object.
System.OverflowException	This exception is thrown when the result of an arithmetic, casting or conversion operation is too large to be stored in the destination object or variable.
System.StackOverflowException	This exception is thrown when the stack runs out of space due to having too many pending method calls.
System.Data.DataException	This exception is thrown when errors are generated while using the ADO.NET components.
System.FormatException	This exception is thrown when the format of an argument does not match the format of the parameter data type of the invoked method.
System.IO.IOException	This exception is thrown when any I/O error occurs while accessing information using streams, files, and directories.

Table 11.3: Other Exception Classes

11.1.7 *InvalidCastException Class*

The `InvalidCastException` exception is thrown when an explicit conversion from a base type to another type fails.

Code Snippet 1 demonstrates the `InvalidCastException` exception.

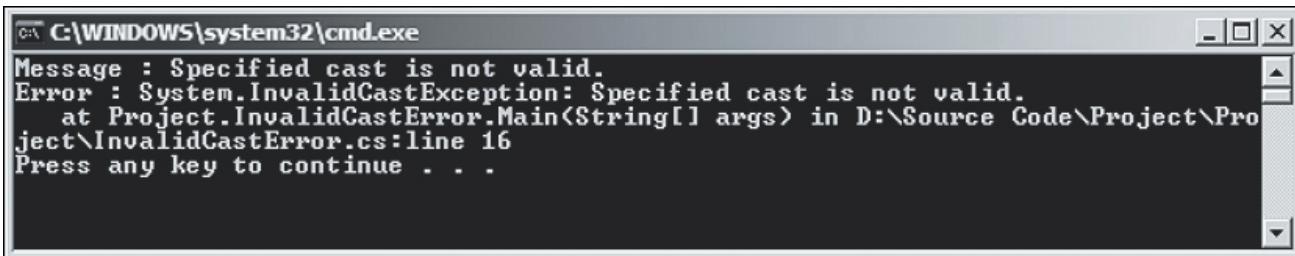
Code Snippet 1:

```
using System;

class InvalidCastError
{
    static void Main(string[] args)
    {
        try
        {
            float numOne = 3.14F;
            Object obj = numOne;
            int result = (int) obj;
            Console.WriteLine("Value of numOne = {0}", result);
        }
        catch (InvalidCastException objEx)
        {
            Console.WriteLine("Message : {0}", objEx.Message);
            Console.WriteLine("Error : {0}", objEx);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}
```

In Code Snippet 1, a variable `numOne` is defined as `float`. When this variable is boxed, it is converted to type `object`. However, when it is unboxed, it causes an `InvalidCastException` and displays a message for the same. This is because a value of type `float` is unboxed to type `int`, which is not allowed in C#.

Figure 11.3 displays the exception that is generated when the program is executed.



```
C:\WINDOWS\system32\cmd.exe
Message : Specified cast is not valid.
Error : System.InvalidCastException: Specified cast is not valid.
        at Project.InvalidCastError.Main<String[] args> in D:\Source Code\Project\Project\InvalidCastError.cs:line 16
Press any key to continue . . .
```

Figure 11.3: InvalidCastException Generated at Run-time

11.1.8 ArrayTypeMismatchException Class

The `ArrayTypeMismatchException` exception is thrown when the data type of the value being stored is incompatible with the data type of the array.

Code Snippet 2 demonstrates the `ArrayTypeMismatchException` exception.

Code Snippet 2:

```
using System;
class ArrayMisMatch
{
    static void Main(string[] args)
    {
        string[] names = { "James", "Jack", "Peter" };
        int[] id = { 10, 11, 12 };
        double[] salary = { 1000, 2000, 3000 };
        float[] bonus = new float[3];
        try
        {
            salary.CopyTo(bonus, 0);
        }
        catch (ArrayTypeMismatchException objType)
        {
            Console.WriteLine("Error: " + objType);
        }
        catch (Exception objEx)
        {
```

```
        Console.WriteLine("Error: " + objEx);  
    }  
}  
}
```

In Code Snippet 2, the **salary** array is of double data type and the **bonus** array is of float data type. When copying the values stored in **salary** array to **bonus** array using the `CopyTo()` method, data type mismatch occurs.

Figure 11.4 displays the exception that is generated when the program is executed.

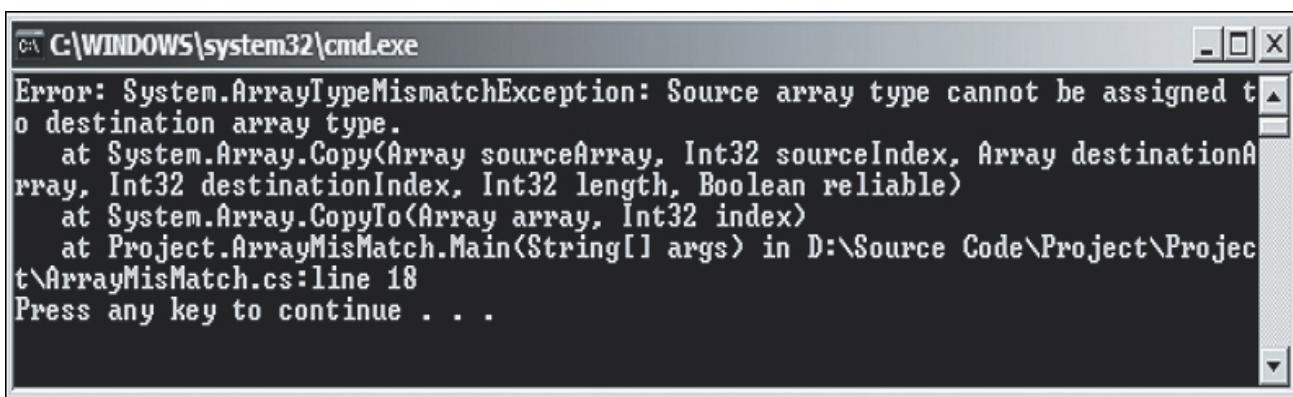


Figure 11.4: ArrayTypeMismatchException Generated at Run-time

11.1.9 NullReferenceException Class

The `NullReferenceException` exception is thrown when an attempt is made to operate on a null reference.

Code Snippet 3 demonstrates the `NullReferenceException` exception.

Code Snippet 3:

```
using System;  
  
class Employee  
{  
  
    private string _empName;  
    private int _empID;  
  
    public Employee()  
    {  
  
        _empName = "David";  
        _empID = 101;  
    }  
}
```

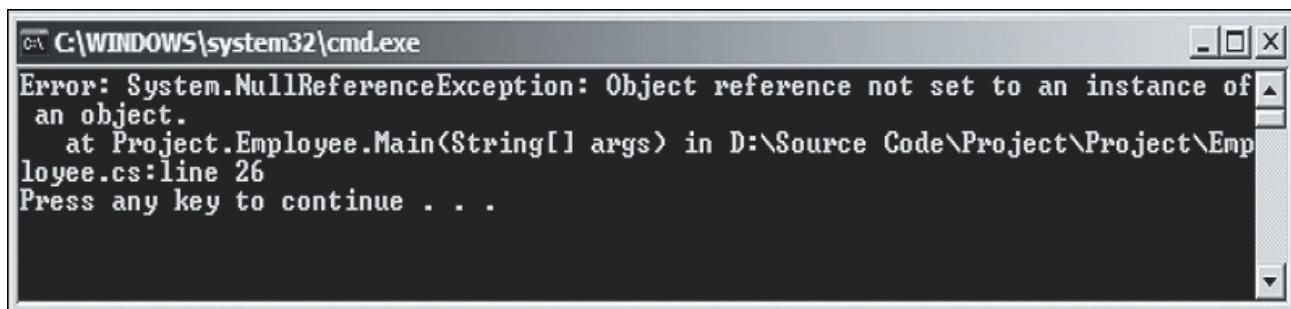
```
}

static void Main(string[] args)
{
    Employee objEmployee = new Employee();
    Employee objEmp = objEmployee;
    objEmployee = null;

    try
    {
        Console.WriteLine("Employee Name: " + objEmployee._empName);
        Console.WriteLine("Employee ID: " + objEmployee._empID);
    }
    catch (NullReferenceException objNull)
    {
        Console.WriteLine("Error: " + objNull);
    }
    catch (Exception objEx)
    {
        Console.WriteLine("Error: " + objEx);
    }
}
```

In Code Snippet 3, a class **Employee** is defined which contains the details of an employee. Two instances of class **Employee** are created with the second instance referencing the first. The first instance is de-referenced using **null**. If the first instance is used to print the values of the **Employee** class, a **NullReferenceException** exception is thrown, which is caught by the **catch** block.

Figure 11.5 displays the exception that is generated when the program is executed.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the following text:

```
Error: System.NullReferenceException: Object reference not set to an instance of
an object.
  at Project.Employee.Main(String[] args) in D:\Source Code\Project\Project\Emp
loyee.cs:line 26
Press any key to continue . . .
```

Figure 11.5: Use of NullReferenceException Class

11.1.10 System.Exception

`System.Exception` is the base class that handles all exceptions. It contains `public` and `protected` methods that can be inherited by other exception classes.

Table 11.4 lists the `public` methods of the `System.Exception` class and their corresponding description.

Method	Description
<code>Equals</code>	Determines whether objects are equal
<code>GetBaseException</code>	Returns a type of <code>Exception</code> class when overridden in a derived class
<code>GetHashCode</code>	Returns a hash function for a particular type
<code>GetObjectData</code>	Stores information about serializing or deserializing a particular object with information about the exception when overridden in the inheriting class
<code>GetType</code>	Retrieves the type of the current instance
<code>ToString</code>	Returns a string representation of the thrown exception

Table 11.4: Public Methods of `System.Exception` Class

Code Snippet 4 demonstrates some public methods of the `System.Exception` class.

Code Snippet 4:

```
using System;
class ExceptionMethods
{
    static void Main(string[] args)
    {
        byte numOne = 200;
```

```
byte numTwo = 5;

byte result = 0;

try

{

    result = checked( (byte) (numOne * numTwo));

    Console.WriteLine("Result = {0}", result);

}

catch (Exception objEx)

{

    Console.WriteLine("Error Description : {0}", objEx.ToString());

    Console.WriteLine("Exception : {0}", objEx.GetType());

}

}

}
```

In Code Snippet 4, the arithmetic overflow occurs because the result of multiplication of two byte numbers exceeds the range of the data type of the destination variable, `result`. The arithmetic overflow exception is thrown and it is caught by the `catch` block. The block uses the `ToString()` method to retrieve the string representation of the exception. The `GetType()` method of the `System.Exception` class retrieves the type of exception which is then displayed by using the `WriteLine()` method.

Figure 11.6 displays the some public methods of the System.Exception class.



Figure 11.6: Methods of System.Exception Class

Table 11.5 lists the protected methods of the `System.Exception` class and their corresponding description.

Name	Description
Finalize	Allows objects to perform cleanup operations before they are reclaimed by the garbage collector
MemberwiseClone	Creates a copy of the current object

Table 11.5: Protected Methods of `System.Exception`

Table 11.6 lists the commonly used public properties in the `System.Exception` class.

Public Property	Description
HelpLink	Retrieves or sets a link to the help file associated with the exception
InnerException	Retrieves the reference of the object of type <code>Exception</code> that caused the thrown exception
Message	Retrieves the description of the current exception
Source	Retrieves or sets the name of the application or the object that resulted in the error
StackTrace	Retrieves a string of the frames on the stack when an exception is thrown
TargetSite	Retrieves the method that throws the exception

Table 11.6: Public Properties of `System.Exception` Class

Code Snippet 5 demonstrates the use of some of the public properties of the `System.Exception` class.

Code Snippet 5:

```
using System;
class ExceptionProperties
{
    static void Main(string[] args)
    {
        byte numOne = 200;
        byte numTwo = 5;
        byte result = 0;
        try
        {
            result = numOne / numTwo;
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

```
        result = checked((byte)(numOne * numTwo));

        Console.WriteLine("Result = {0}", result);

    }

    catch (OverflowException objEx)

    {

        Console.WriteLine("Message : {0}", objEx.Message);

        Console.WriteLine("Source : {0}", objEx.Source);

        Console.WriteLine("TargetSite : {0}", objEx.TargetSite);

        Console.WriteLine("StackTrace : {0}", objEx.StackTrace);

    }

    catch (Exception objEx)

    {

        Console.WriteLine("Error : {0}", objEx);

    }

}
```

In Code Snippet 5, the arithmetic overflow occurs because the result of multiplication of two byte numbers exceeds the range of the destination variable type, `result`. The arithmetic overflow exception is thrown and it is caught by the `catch` block. The block uses various properties of the `System.Exception` class to display the source and target site of the error.

Figure 11.7 displays use of some of the public properties of the System.Exception class.

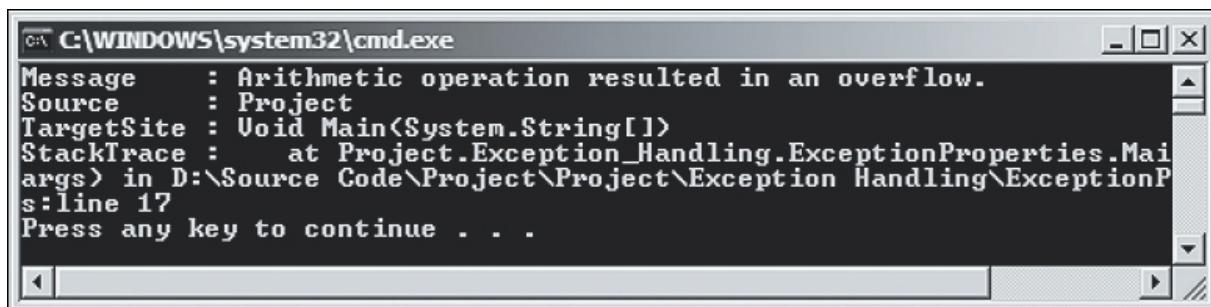


Figure 11.7: Using the Public Properties of System.Exception Class

11.2 Throwing and Catching Exceptions

An exception is an error that occurs during program execution. An exception arises when an operation cannot be completed normally. In such situations, the system throws an error. The error is handled through the process of exception handling.

11.2.1 Purpose of Exception Handlers

Consider a group of boys playing throwball wherein one boy throws a ball and the other boys catch the ball. If any of the boys fail to catch the ball, the game will be terminated. Thus, the game goes on till the boys are successful in catching the ball on time.

Similarly, in C#, exceptions that occur while working on a particular program need to be caught by exception handlers. If the program does not contain the appropriate exception handler, then the program might be terminated.

11.2.2 Catching Exceptions

Exception handling is implemented using the `try`-`catch` construct in C#. This construct consists of two blocks, the `try` block and the `catch` block. The `try` block encloses statements that might generate exceptions. When these exceptions are thrown, the required actions are performed using the `catch` block. Thus, the `catch` block consists of the appropriate error-handlers that handle exceptions.

The `catch` block follows the `try` block and may or may not contain parameters. If the `catch` block does not contain any parameter, it can catch any type of exception. If the `catch` block contains a parameter, it catches the type of exception specified by the parameter.

The following syntax is used for handling errors using the `try` and `catch` blocks.

Syntax:

```
try
{
    // program code
}

catch[ (<ExceptionClass> <objException>) ]
{
    // error handling code
}
```

where,

`try`: Specifies that the block encloses statements that may throw exceptions.

`program code`: Are statements that may generate exceptions.

catch: Specifies that the block encloses statements that catch exceptions and carry out the appropriate actions.

ExceptionClass: Is the name of exception class. It is optional.

objException: Is an instance of the particular exception class. It can be omitted if the **Exception** class is omitted.

Code Snippet 6 demonstrates the use of **try-catch** blocks.

Code Snippet 6:

```
using System;
class DivisionError
{
    static void Main(string[] args)
    {
        int numOne = 133;
        int numTwo = 0;
        int result;
        try
        {
            result = numOne / numTwo;
        }
        catch (DivideByZeroException objDivide)
        {
            Console.WriteLine("Exception caught: " + objDivide);
        }
    }
}
```

In Code Snippet 6, the **Main()** method of the class **DivisionError** declares three variables, two of which are initialized. The **try** block contains the code to divide **numOne** by **numTwo** and store the output in the **result** variable. As **numTwo** has a value of zero, the **try** block throws an exception. This exception is handled by the corresponding **catch** block, which takes in **objDivide** as the instance of the **DivideByZeroException** class. The exception is caught by this **catch** block and the appropriate message is displayed as the output.

Output:

Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at DivisionError.Main(String[] args)

Note - The `catch` block is only executed if the `try` block throws an exception.

11.2.3 General catch Block

The `catch` block can handle all types of exceptions. However, the type of exception that the `catch` block handles depends on the specified exception class. Sometimes, you might not know the type of exception the program might throw. In such a case, you can create a `catch` block with the base class `Exception`. Such `catch` blocks are referred to as **general catch blocks**.

A general `catch` block can handle all types of exceptions. However, the disadvantage of the general `catch` block is that there is no instance of the exception and thus, you cannot know what appropriate action must be performed for handling the exception.

Code Snippet 7 demonstrates the way in which a general `catch` block is declared.

Code Snippet 7:

```
using System;
class Students
{
    string[] _names = { "James", "John", "Alexander" };
    static void Main(string[] args)
    {
        Students objStudents = new Students();
        try
        {
            objStudents._names[4] = "Michael";
        }
        catch (Exception objException)
        {
            Console.WriteLine("Error: " + objException);
        }
    }
}
```

In Code Snippet 7, a `string` array called `names` is initialized to contain three values. In the `try` block, there is a statement trying to reference a fourth array element.

The array can store only three values, so this will cause an exception. The class **Students** consists of a general `catch` block that is declared with the base class `Exception` and this `catch` block can handle any type of exception.

Output:

```
Error: System.IndexOutOfRangeException: Index was outside the bounds of the
array at Project_New.Exception_Handling.Students.Main(String[] args) in D:\Exception Handling\Students.cs:line 17
```

11.2.4 The throw Statement

The `throw` statement in C# allows you to programmatically throw exceptions using the `throw` keyword. The `throw` statement takes an instance of the particular exception class as a parameter. However, if the instance does not refer to the valid exception class, the C# compiler generates an error.

When you `throw` an exception using the `throw` keyword, the exception is handled by the `catch` block. The following syntax is used to throw an exception programmatically.

Syntax:

```
throw exceptionObject;
```

where,

`throw`: Specifies that the exception is thrown programmatically.

`exceptionObject`: Is an instance of a particular exception class.

Code Snippet 8 demonstrates the use of the `throw` statement.

Code Snippet 8:

```
using System;

class Employee
{
    static void ThrowException(string name)
    {
        if (name == null)
        {
            throw new ArgumentNullException();
        }
    }

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

```
Console.WriteLine("Throw Example");

try
{
    string empName = null;
    ThrowException(empName);
}

catch (ArgumentNullException objNull)
{
    Console.WriteLine("Exception caught: " + objNull);
}

}
```

In Code Snippet 8, the class `Employee` declares a static method called `ThrowException` that takes a string parameter called `name`. If the value of `name` is `null`, the C# compiler throws the exception, which is caught by the instance of the `ArgumentNullException` class. In the `try` block, the value of `name` is `null` and the control of the program goes back to the method `ThrowException`, where the exception is thrown for referencing a `null` value. The `catch` block consists of the error handler, which is executed when the exception is thrown.

Output:

Throw Example

Exception caught: System.ArgumentNullException: Value cannot be null.

at Exception_Handling.Employee.ThrowException(String name) in D:\Exception Handling\Employee.cs:

line 13

at Exception_Handling.Employee.Main(String[] args) in D:\Exception Handling\Employee.cs:line 24

11.2.5 The *finally* Statement

In general, if any of the statements in the `try` block raises an exception, the `catch` block is executed and the rest of the statements in the `try` block are ignored. Sometimes, it becomes mandatory to execute some statements irrespective of whether an exception is raised or not. In such cases, a `finally` block is used. Examples of actions that can be placed in a `finally` block are: closing a file, assigning objects to null, closing a database connection and so forth.

The `finally` block is an optional block and it appears after the `catch` block. It encloses statements that are executed regardless of whether an exception occurs.

The following syntax demonstrates the use of the `finally` block.

Syntax:

```
finally
{
    // cleanup code;
}
```

where,

`finally`: Specifies that the statements in the block have to be executed irrespective of whether or not an exception is raised.

Code Snippet 9 demonstrates the use of the `try-catch-finally` construct.

Code Snippet 9:

```
using System;

class DivisionError
{
    static void Main(string[] args)
    {
        int numOne = 133;
        int numTwo = 0;
        int result;
        try
        {
            result = numOne / numTwo;
        }
        catch (DivideByZeroException objDivide)
        {
            Console.WriteLine("Exception caught: " + objDivide);
        }
        finally
        {
            Console.WriteLine("This finally block will always be executed");
        }
    }
}
```

```
}
```

In Code Snippet 9, the `Main()` method of the class `DivisionError` declares and initializes two variables. An attempt is made to divide one of the variables by zero and an exception is raised. This exception is caught using the `try-catch-finally` construct. The `finally` block is executed at the end, even though an exception is thrown by the `try` block.

Output:

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at DivisionError.Main(String[] args)
This finally block will always be executed
```

11.2.6 Performance

Exceptions hinder the performance of a program. There are two design patterns that help in minimizing the hindrance caused due to exceptions. These are:

→ **Tester-Doer Pattern**

A call that might throw exceptions is divided into two parts, tester and doer, using the tester-doer pattern. The tester part will perform a test on the condition that might cause the doer part to throw an exception. This test is then placed just before the code that throws an exception.

Code Snippet 10 demonstrates the use of the tester-doer pattern.

Code Snippet 10:

```
using System;
class NumberDoer
{
    public void Process(int numOne, int numTwo)
    {
        try
        {
            if (numTwo == 0)
            {
                throw new DivideByZeroException("Value of divisor is zero");
            }
        }
        else
```

```
{  
    Console.WriteLine ("Quotient: " + (numOne / numTwo));  
    Console.WriteLine ("Remainder: " + (numOne % numTwo));  
}  
}  
catch (DivideByZeroException objDivide)  
{  
    Console.WriteLine ("Error: " + objDivide);  
}  
}  
}  
}  
  
class NumberTester  
{  
    NumberDoer objDoer = new NumberDoer();  
    public void AcceptDetails()  
    {  
        int dividend = 0;  
        int divisor = 0;  
        Console.WriteLine ("Enter the value of dividend");  
        try  
        {  
            dividend = Convert.ToInt32 (Console.ReadLine());  
        }  
        catch (FormatException objForm)  
        {  
            Console.WriteLine ("Error: " + objForm);  
        }  
        Console.WriteLine ("Enter the value of divisor");  
        try  
        {  
            divisor = Convert.ToInt32 (Console.ReadLine());  
        }
```

```
}

catch (FormatException objFormat)
{
    Console.WriteLine("Error: " + objFormat);
}

if ((dividend > 0) || (divisor > 0))
{
    objDoer.Process(dividend, divisor);
}
else
{
    Console.WriteLine("Invalid input");
}

static void Main(string[] args)
{
    NumberTester objTester = new NumberTester();
    objTester.AcceptDetails();
}
```

In Code Snippet 10, two classes, **NumberDoer** and **NumberTester**, are defined. In the **NumberTester** class, if the value entered is not of `int` data type, an exception is thrown. Next, a test is performed to check if either of the values of dividend and divisor is greater than 0. If the result is true, the values are passed to the `Process()` method of the **NumberDoer** class. In the **NumberDoer** class, another test is performed to check whether the value of the divisor is equal to 0. If the condition is true, then the `DivideByZero` exception is thrown and caught by the `catch` block.

Figure 11.8 displays the use of tester-doer pattern.

```
C:\WINDOWS\system32\cmd.exe
Enter the value of dividend
5
Enter the value of divisor
0
Error: System.DivideByZeroException: Value of divisor is zero
      at Project.NumberDoer.Process<Int32 numOne, Int32 numTwo> in D:\Source Code\Project\Project\NumberTester.cs:line 15
Press any key to continue . . .
```

Figure 11.8: Use of Tester-Doer Pattern

→ TryParse Pattern

There are two different methods to implement the `TryParse` pattern. The first method is `X`, which performs the operation and throws the exception.

The second method is `TryX`, which also performs the operation but does not throw the exception.

Instead, it returns a boolean value that indicates whether the operation failed or succeeded.

Code Snippet 11 demonstrates the use of the `TryParse` pattern.

Code Snippet 11:

```
using System;

class Product
{
    private int _quantity;
    private float _price;
    private double _sales;
    string _productName;
    public Product()
    {
        _productName = "Motherboard";
    }
    public void AcceptDetails()
    {
        Console.WriteLine("Enter the number of " + _productName + " sold");
        try
        {
```

```
        _quantity=Int32.Parse(Console.ReadLine());  
    }  
  
    catch (FormatException objFormat)  
    {  
  
        Console.WriteLine("Error: " + objFormat);  
        return;  
    }  
  
    Console.WriteLine("Enter the price of the product");  
  
    if (float.TryParse((Console.ReadLine()), out _price) == true)  
    {  
  
        _sales=_price * _quantity;  
    }  
  
    else  
    {  
  
        Console.WriteLine("Invalid price inserted");  
    }  
}  
  
public void Display()  
{  
  
    Console.WriteLine("Product Name: " + _productName);  
    Console.WriteLine("Product Price: " + _price);  
    Console.WriteLine("Quantity sold: " + _quantity);  
    Console.WriteLine("Total Sale Value: " + _sales);  
}  
  
static void Main(string[] args)  
{  
  
    Product objGoods = new Product();  
    objGoods.AcceptDetails();  
    objGoods.Display();  
}  
}
```

In Code Snippet 11, a class `Product` is defined in which the total product sales is calculated. The user-defined function `AcceptDetails` accepts the number of products sold and the price of each product. The number of products sold is converted to `int` data type using the `Parse()` method of the `Int32` structure. This method throws an exception if the value entered is not of the `int` data type. Hence, this method is included in the `try...catch` block. When accepting the price of the product, the `TryParse` pattern is used to verify whether the value entered is of the correct data type. If the value returned by the `TryParse()` method is true, the sale price is calculated.

Figure 11.9 displays output of Code Snippet 11 when the user enters a wrong value for the price of the product.

```
C:\WINDOWS\system32\cmd.exe
Enter the number of Motherboard sold
10
Enter the price of the product
1000.2a
Invalid price inserted
Product Name: Motherboard
Product Price: 0
Quantity sold: 10
Total Sale Value: 0
Press any key to continue . . .
```

Figure 11.9: Output of Example for Wrong Values

Figure 11.10 displays the output of Code Snippet 11 when the user enters a value which is not of `int` data type.

```
C:\WINDOWS\system32\cmd.exe
Enter the number of Motherboard sold
abc
Error: System.FormatException: Input string was not in a correct format.
  at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer
r& number, NumberFormatInfo info, Boolean parseDecimal)
  at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo in
fo)
  at System.Int32.Parse(String s)
  at Project.Product.AcceptDetails() in D:\Source Code\Project\Project\Product.
cs:line 24
Product Name: Motherboard
Product Price: 0
Quantity sold: 0
Total Sale Value: 0
Press any key to continue . . .
```

Figure 11.10: Output of Example for Wrong Type

11.3 Nested `try` and Multiple `catch` Blocks

Exception handling code can be nested in a program. In nested exception handling, a `try` block can enclose another `try-catch` block. In addition, a single `try` block can have multiple `catch` blocks that are sequenced to catch and handle different type of exceptions raised in the `try` block.

11.3.1 Nested `try` Blocks

The nested `try` block consists of multiple `try-catch` constructs. A nested `try` block starts with a `try` block, which is called the outer `try` block. This outer `try` block contains multiple `try` blocks within it, which are called inner `try` blocks. If an exception is thrown by a nested `try` block, the control passes to its corresponding nested `catch` block.

Consider an outer `try` block containing a nested `try-catch-finally` construct. If the inner `try` block throws an exception, control is passed to the inner `catch` block. However, if the inner `catch` block does not contain the appropriate error handler, the control is passed to the outer `catch` block.

The following syntax is used to create nested `try...catch` blocks.

Syntax:

```
try
{
    // outer try block
    try
    {
        // inner try block
    }
    catch
    {
        // inner catch block
    }
    // this is optional
    finally
    {
        // inner finally block
    }
}
```

```
{
    // outer catch block
}
// this is optional
finally
{
    // outer finally block
}
```

Code Snippet 12 demonstrates the use of nested `try` blocks.

Code Snippet 12:

```
static void Main(string[] args)
{
    string[] names = {"John", "James"};
    int numOne = 0;
    int result;
    try
    {
        Console.WriteLine("This is the outer try block");
        try
        {
            result = 133 / numOne;
        }
        catch (ArithmetiException objMaths)
        {
            Console.WriteLine("Divide by 0 " + objMaths);
        }
        names[2] = "Smith";
    }
    catch (IndexOutOfRangeException objIndex)
    {
        Console.WriteLine("Wrong number of arguments supplied" + objIndex);
    }
}
```

```
}
```

```
}
```

In Code Snippet 12, the array variable called `names` of type `string` is initialized to have two values. The outer `try` block consists of another `try-catch` construct. The inner `try` block divides two numbers. As an attempt is made to divide the number by zero, the inner `try` block throws an exception, which is handled by the inner `catch` block. In addition, in the outer `try` block, there is a statement referencing a third array element whereas the array can store only two values. So, the outer `try` block also throws an exception, which is handled by the outer `catch` block.

Output:

```
This is the outer try block

Divide by 0 System.DivideByZeroException: Attempted to divide by zero.
   at Product.Main(String[] args) in c:\ConsoleApplication1\Program.cs:line 52
Wrong number of arguments supplied System.IndexOutOfRangeException: Index was
outside the bounds of the array.

   at Product.Main(String[] args) in c:\ConsoleApplication1\Program.cs:line 58
```

11.3.2 Multiple catch Blocks

A `try` block can throw multiple types of exceptions, which need to be handled by the `catch` block. C# allows you to define multiple `catch` blocks to handle the different types of exceptions that might be raised by the `try` block. Depending on the type of exception thrown by the `try` block, the appropriate `catch` block (if present) is executed.

However, if the compiler does not find the appropriate `catch` block, then the general `catch` block is executed.

Once the `catch` block is executed, the program control is passed to the `finally` block (if any) and then the control terminates the `try-catch-finally` construct.

The following syntax is used for defining multiple `catch` blocks.

Syntax:

```
try
{
// program code
}

catch (<ExceptionClass> <objException>)
{
// statements for handling the exception
}
```

```
catch (<ExceptionClass1> <objException>)
{
// statements for handling the exception
}

. . .
```

Code Snippet 13 demonstrates the use of multiple `catch` blocks.

Code Snippet 13:

```
static void Main(string[] args)
{
    string[] names = { "John", "James" };
    int numOne = 10;
    int result = 0;
    int index = 0;
    try
    {
        index = 3;
        names[index] = "Smith";
        result = 130 / numOne;
    }
    catch (DivideByZeroException objDivide)
    {
        Console.WriteLine("Divide by 0 " + objDivide);
    }
    catch (IndexOutOfRangeException objIndex)
    {
        Console.WriteLine("Wrong number of arguments supplied " + objIndex);
    }
    catch (Exception objException)
    {
        Console.WriteLine("Error: " + objException);
    }
}
```

```
Console.WriteLine(result);
}
```

In Code Snippet 13, the array, `names`, is initialized to two element values, and two integer variables are declared and initialized. As there is a reference to a third array element, an exception of type `IndexOutOfRangeException` is thrown and the second `catch` block is executed. Instead of hard-coding the index value, it could also happen that some mathematical operation results in the value of index becoming more than 2. Since the `try` block encounters an exception in the first statement, the next statement in the `try` block is not executed and the control terminates the `try-catch` construct. So, the C# compiler prints the initialized value of the variable `result` and not the value obtained by dividing the two numbers. However, if an exception occurs that cannot be caught using either of the two `catch` blocks, then the last `catch` block with the general `Exception` class will be executed.

11.3.3 System.TypeInitializationException Class

The `System.TypeInitializationException` class is used to handle exceptions that are thrown when an instance of a class attempts to invoke the static constructor of the class.

Code Snippet 14 demonstrates the use of the `System.TypeInitializationException` class.

Code Snippet 14:

```
using System;
class TypeInitError
{
    static TypeInitError()
    {
        throw new ApplicationException("This program throws
                                         TypeInitializationException error.");
    }
    static void Main(string[] args)
    {
        try
        {
            TypeInitError objType = new TypeInitError();
        }
    }
}
```

```

        catch (TypeInitializationException objEx)
        {
            Console.WriteLine("Error : {0}", objEx);
        }

        catch (Exception objEx)
        {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}

```

In Code Snippet 14, a static constructor **TypeInitError** is defined. Here, only the static constructor is defined and when the instance of the class, **objType** attempts to invoke the constructor, an exception is thrown. This invocation of the static constructor is not allowed and the instance of the TypeInitializationException class is used to display the error message in the `catch` block.

Figure 11.11 displays the `System.TypeInitializationException` generated at run-time.

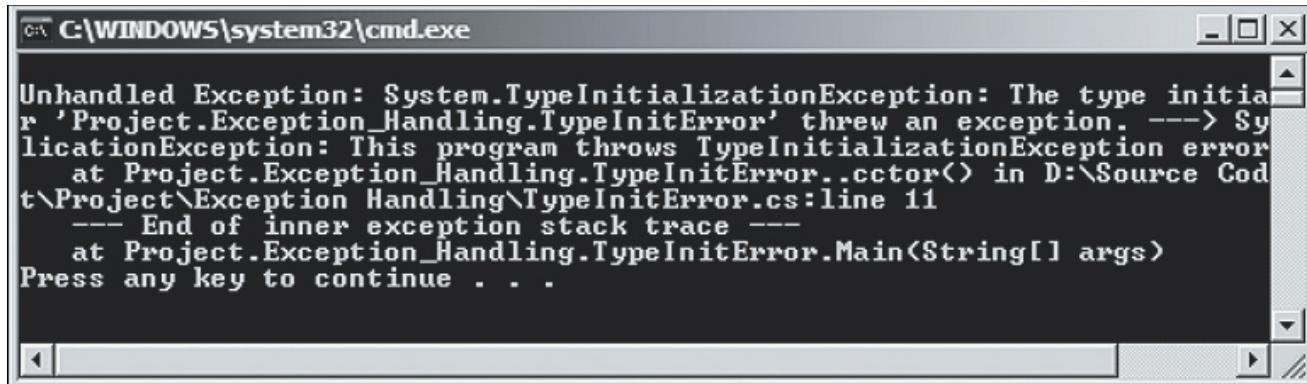


Figure 11.11: `System.TypeInitializationException`

11.4 Custom Exceptions

Custom exceptions are user-defined exceptions that allow users to recognize the cause of unexpected events in specific programs and display custom messages. Thus, custom exceptions allow you to simplify and improve the process of error-handling in a program. Even though C# provides you with a rich set of exception classes, they do not address certain application-specific and system-specific constraints. To handle such constraints, you can define custom exceptions.

Custom exceptions can be created to handle exceptions that are thrown by the CLR as well as those that are thrown by user applications.

11.4.1 Implementing Custom Exceptions

Custom exceptions can be created by deriving from the `Exception` class, the `SystemException` class, or the `ApplicationException` class. The `SystemException` class handles exceptions thrown by the CLR. The `ApplicationException` class handles exceptions thrown by the user-created applications.

However, to define a custom exception, you first need to define a new class and then derive it from the `Exception`, `SystemException` or `ApplicationException` class. Once you have created a custom exception, you can reuse it in any C# application.

Code Snippet 15 demonstrates the implementation of custom exceptions.

Code Snippet 15:

```
public class CustomMessage : Exception
{
    public CustomMessage (string message) : base (message)
    {
    }
}

public class CustomException
{
    static void Main (string[] args)
    {
        try
        {
            throw new CustomMessage ("This illustrates creation and
                catching of custom exception");
        }

        catch (CustomMessage objCustom)
        {
            Console.WriteLine (objCustom.Message);
        }
    }
}
```

In Code Snippet 15, the class `CustomMessage` is created, which is derived from the class `Exception`. The constructor of the class `CustomMessage` is created and it takes a `string` parameter. This constructor, in turn, calls the constructor of the base class, `Exception`. The class `CustomException` consists of the `Main()` method. The `try` block of the class `CustomException` throws the custom exception that passes a `string` value. The `catch` block catches the exception thrown by the `try` block and prints the message.

Output:

This illustrates creation and catching of custom exception

11.4.2 Guidelines for Designing Custom Exceptions

Custom exceptions are user-defined exceptions. Certain guidelines should be followed when designing custom exceptions:

- Do not create deep exception hierarchies.
- Derive an exception from the `System.Exception` base class and not from `ApplicationException`.
- End the exception class names with the `Exception` suffix.
- Make the exceptions serializable.

11.4.3 Exception Assistant

Exception assistant is a new feature for debugging C# applications. The Exception dialog box feature in earlier versions of C# is replaced by Exception assistant. This assistant appears whenever a run-time exception occurs. It shows the type of exception that occurred, the troubleshooting tips, and the corrective action to be taken, and can also be used to view the details of an exception object.

Exception assistant provides more information about an exception than the Exception dialog box. The assistant makes it easier to locate the cause of the exception and to solve the problem.

11.5 Check Your Progress

1. Which of these statements about exceptions and exception classes are true?

(A)	Exceptions are compile-time errors that disrupt the program flow.
(B)	Display.Exception is the base class for all exceptions.
(C)	InnerException is a property of the Exception class which returns an Exception object that caused the exception.
(D)	Message is a property of the Exception class that displays the reason for the exception.
(E)	Source is a property of the Exception class that provides the name of the application that caused the exception.

(A)	C, D, E	(C)	A, C
(B)	B	(D)	D, E

2. Match the commonly used exception classes against their corresponding descriptions.

Description		Exception Class	
(A)	Raised when you try to create a new object and there is not enough memory	(1)	ArrayTypeMismatchException
(B)	Raised when you try to store an element in an array whose data type is not compatible with the data type of the array	(2)	DataException
(C)	Raised when an explicit conversion fails	(3)	IndexOutOfRangeException
(D)	Raised when errors are generated while working with the ADO.NET components	(4)	InvalidOperationException
(E)	Raised when you try to store data in an array using an index that is not within the bounds of the array	(5)	OutOfMemoryException

(A)	A-1, B-5, C-4, D-2, E-3	(C)	A-2, B-5, C-4, D-1, E-3
(B)	A-5, B-1, C-4, D-2, E-3	(D)	A-5, B-1, C-4, D-2, E-3

3. Match the keywords used for handling exceptions against their corresponding descriptions.

Description		Keyword	
(A)	Executes at the end of the try block	(1)	try
(B)	Enables throwing of exceptions through code	(2)	catch
(C)	Consists of the exception-handling statements	(3)	finally
(D)	Contains statements that might raise exceptions	(4)	throw
(E)	Executes the code within the block irrespective of whether an exception is raised or not	(5)	

(A)	A-3, B-4, C-2, D-1, E-3	(C)	A-1, B-4, C-2, D-3, E-1
(B)	A-2, B-4, C-3, D-1, E-2	(D)	A-4, B-4, C-2, D-1, E-3

4. You are trying to write an exception handler to catch the exception which may occur when the user inputs any value other than an integer value. Which of the following codes helps you achieve this?

(A)	<pre>class Format { int _dividend=133; int _divisor; int _result=0; static void Main(string[] args) { Format objFormat = new Format(); Console.WriteLine("Enter the value of divisor:"); try { objFormat._divisor = </pre>
-----	--

```

        Convert.ToInt32(Console.ReadLine()));

        objFormat._result = objFormat._dividend /
        objFormat._divisor;

    }

    catch (FormatException objForm)
    {
        Console.WriteLine("Error: " + objForm);

    }

    Console.WriteLine("Result of division: " +
        objFormat._result);

}

}

```

```

class Format
{
    int _dividend = 133;
    int _divisor;
    int _result = 0;
    static void Main(string[] args)
    {
        Format objFormat = new Format();
        Console.WriteLine("Enter the value of divisor: ");
        try
        {
            objFormat._divisor =
            Convert.ToInt32(Console.ReadLine());
            objFormat._result = objFormat._dividend /
            objFormat._divisor;
        }
    }
}

```

(B)	<pre> catch (InvalidCastException objCast) { Console.WriteLine("Error: " + objCast); } Console.WriteLine("Result of division: " + objFormat._result); } } </pre>
(C)	<pre> class Format { int _dividend = 133; int _divisor; int _result = 0; static void Main(string[] args) { Format objFormat = new Format(); Console.WriteLine("Enter the value of divisor:"); try { objFormat._divisor = Convert.ToInt32(Console.ReadLine()); objFormat._result = objFormat._dividend / objFormat._divisor; } finally (FormatException objForm) { Console.WriteLine("Error: " + objForm); } Console.WriteLine("Result of division: " + objFormat._result); } } </pre>

(D)

```

class Format
{
    int _dividend=133;
    int _divisor;
    int _result=0;
    static void Main(string[] args)
    {
        Format objFormat = new Format();
        Console.WriteLine("Enter the value of divisor: ");
        try
        {
            objFormat._divisor =
                Convert.ToInt32(Console.ReadLine());
            objFormat._result = objFormat._dividend /
                objFormat._divisor;
        }
        catch ()
        {
            Console.WriteLine("Error: " + objFormat);
        }
        Console.WriteLine("Result of division: " +
            objFormat._result);
    }
}

```

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of these statements about nested `try` blocks and multiple catch blocks are true?

(A)	Nested <code>try</code> blocks can include multiple <code>try</code> - <code>catch</code> constructs within a <code>try</code> block.		
(B)	<code>try</code> blocks cannot throw more than one exception.		
(C)	The outer <code>catch</code> block cannot handle exceptions thrown by the inner <code>try</code> block.		
(D)	Multiple <code>catch</code> blocks can handle different types of exceptions thrown by the <code>try</code> block.		
(E)	The program control can return to the <code>try</code> block once the corresponding <code>catch</code> block is executed.		

(A)	A	(C)	C
(B)	B, C, D	(D)	A, D

6. Which of these statements about custom exceptions are true?

(A)	Custom exceptions can be used to customize the error-handling process.		
(B)	The <code>ApplicationException</code> class can be used for exceptions thrown by the system applications.		
(C)	Custom exceptions can be derived from the <code>Exception</code> class.		
(D)	Custom exceptions can be used to address the system and application-specific constraints.		
(E)	Custom exceptions can be used to change the built-in exceptions with modified messages.		

(A)	A, B	(C)	A, C, D
(B)	B	(D)	A, D

7. You are trying to create a custom exception that will be raised and handled if the user enters a zero or a negative value. Which of the following codes helps you achieve this?

(A)	using System; class Custom : Exception { int number;
-----	---

(A)

```
public CustomExample(string msg) : base(msg)
{
}

static void Main(string[] args)
{
    Console.WriteLine("Enter a positive number:");
    try
    {
        number = Convert.ToInt32(Console.ReadLine());
        if (number == 0)
        {
            throw new CustomExample("Zero not allowed here");
        }
        if (number < 0)
        {
            throw new CustomExample("Please enter a positive number");
        }
        Console.WriteLine("You entered: " + number.ToString());
    }
    catch (CustomExample objCustom)
    {
        Console.WriteLine("Error: " + objCustom.Message);
    }
}
```

(B)

```
using System;

class Custom : Exception
{
    static int number;

    public CustomExample()
    {
    }

    static void Main(string[] args)
    {
        Console.WriteLine("Enter a positive number:");

        try
        {
            number = Convert.ToInt32(Console.ReadLine());
            if (number == 0)
            {
                throw new CustomExample("Zero not allowed here");
            }
            if (number < 0)
            {
                throw new CustomExample("Please enter a positive number");
            }
            Console.WriteLine("You entered: " +
                number.ToString());
        }
        catch (CustomExample objCustom)
        {
```

(B)	Console.WriteLine("Error: " + objCustom.Message); } } }
(C)	using System; class Custom : Exception { static int number; public CustomExample(string msg) { base(msg); } static void Main(string[] args) { Console.WriteLine("Enter a positive number:"); try { number = Convert.ToInt32(Console.ReadLine()); if (number == 0) { throw new CustomExample("Zero not allowed here"); } if (number < 0) { throw new CustomExample("Please enter a positive number"); } Console.WriteLine("You entered: " + number.ToString()); } } }

(C)	<pre> } catch (CustomExample objCustom) { Console.WriteLine("Error: " + objCustom.Message); } } } </pre>
(D)	<pre> using System; class CustomExample : Exception { static int number; public CustomExample(string msg) : base(msg) { } static void Main(string[] args) { Console.WriteLine("Enter a positive number:"); try { number = Convert.ToInt32(Console.ReadLine()); if (number == 0) { throw new CustomExample("Zero not allowed here"); } if (number < 0) { throw new CustomExample("Please enter a positive number"); } } } } </pre>

```
(D)    }  
      Console.WriteLine("You entered: " +  
        number.ToString());  
    }  
  catch (CustomExample objCustom)  
  {  
    Console.WriteLine("Error: " +  
      objCustom.Message);  
  }  
}
```

(A)	A	(C)	C
(B)	B	(D)	D

11.5.1 Answers

1.	A
2.	B
3.	A
4.	A
5.	D
6.	C
7.	D



Summary

- Exceptions are errors encountered at run-time.
- Exception-handling allows you to handle methods that are expected to generate exceptions.
- The try block should enclose statements that may generate exceptions while the catch block should catch these exceptions.
- The finally block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the try block.
- Nested try blocks allow you to have a try-catch-finally construct within a try block.
- Multiple catch blocks can be implemented when a try block throws multiple types of exceptions.
- Custom exceptions are user-defined exceptions that allow users to handle system and application specific run-time errors.

Amplification **Metaphrase**
Decoding **Abbreviations**
Terms **Acronyms**
GLOSSARY **Explanation**
look-up guide Terminology
Wordbook Wordlist



Session -12

Events, Delegates, and Collections

Welcome to the Session, **Events, Delegates, and Collections**.

Delegates in C# are used to reference methods defined in a class. They provide the ability to refer to a method in the form of a parameter. Events are actions that trigger methods to execute a set of statements. Delegates can be used with events to bind them to the methods that handle the events. Collections allow you to control and manipulate a group of objects dynamically at run-time. The `System.Collections` namespace consists of collections of arrays, lists, hashtables, and dictionaries. The `System.Collections.Generic` namespace consists of generic collections, which provide better type-safety and performance.

In this session, you will learn to:

- ➔ Explain delegates
- ➔ Explain events
- ➔ Define and describe collections

12.1 Delegates

In the .NET Framework, a delegate points to one or more methods. Once you instantiate the delegate, the corresponding methods invoke.

Delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names. Using delegates, you can call any method, which is identified only at run-time. A delegate is like having a general method name that points to various methods at different times and invokes the required method at run-time. In C#, invoking a delegate will execute the referenced method at run-time.

To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.

12.1.1 Delegates in C#

Consider two methods, `Add()` and `Subtract()`. The method `Add()` takes two parameters of type integer and returns their sum as an integer value. Similarly, the method `Subtract()` takes two parameters of type integer and returns their difference as an integer value.

Since both methods have the same parameter and return types, a delegate, `Calculation`, can be created to be used to refer to `Add()` or `Subtract()`. However, when the delegate is called while pointing to `Add()`, the parameters will be added. Similarly, if the delegate is called while pointing to `Subtract()`, the parameters will be subtracted.

Delegates in C# have some features that distinguish them from normal methods. These features are as follows:

- Methods can be passed as parameters to a delegate. In addition, a delegate can accept a block of code as a parameter. Such blocks are referred to as anonymous methods because they have no method name.
- A delegate can invoke multiple methods simultaneously. This is known as multicasting.
- A delegate can encapsulate static methods.
- Delegates ensure type-safety as the return and parameter types of the delegate are the same as that of the referenced method. This ensures secured reliable data to be passed to the invoked method.

12.1.2 Declaring Delegates

Delegates in C# are declared using the `delegate` keyword followed by the return type and the parameters of the referenced method. Declaring a delegate is quite similar to declaring a method except that there is no implementation. Thus, the declaration statement must end with a semi-colon.

Figure 12.1 displays an example of declaring delegates.

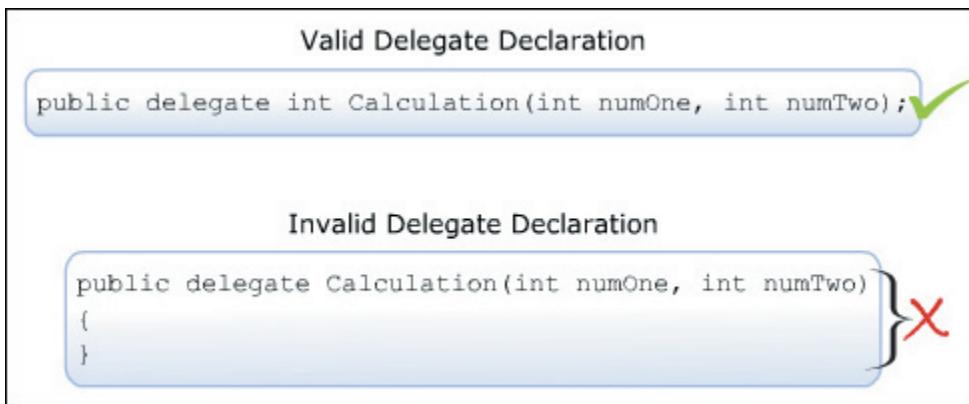


Figure 12.1: Declaring Delegates

The following syntax is used to declare a delegate.

Syntax:

<access_modifier> delegate <return_type> DelegateName([list_of_parameters]);

where,

access_modifier: Specifies the scope of access for the delegate. If declared outside the class, the scope will always be **public**.

return_type: Specifies the data type of the value that is returned by the method.

DelegateName: Specifies the name of the delegate.

list_of_parameters: Specifies the data types and names of parameters to be passed to the method.

Code Snippet 1 declares the delegate **Calculation** with the return type and the parameter types as **int**.

Code Snippet 1:

```
public delegate int Calculation(int numOne, int numTwo);
```

Note - If the delegate is declared outside the class, you cannot declare another delegate with the same name in that namespace.

12.1.3 Instantiating Delegates

The next step after declaring the delegate is to instantiate the delegate and associate it with the required method. Here, you must create an object of the delegate.

Like all other objects, an object of a delegate is created using the **new** keyword. This object takes the name of the method as a parameter and this method has a signature similar to that of the delegate. The created object is used to invoke the associated method at run-time.

Figure 12.2 displays an example of instantiating delegates.

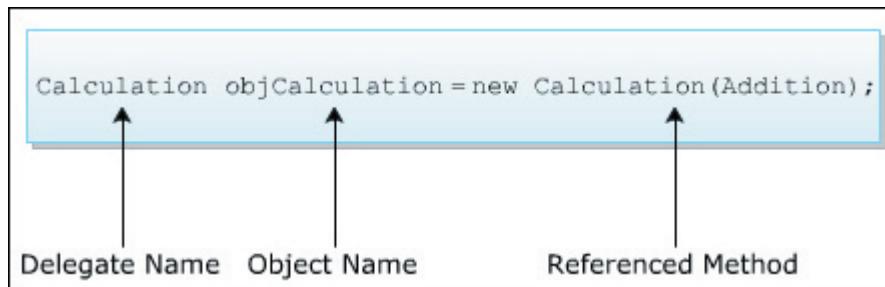


Figure 12.2: Instantiating Delegates

The following syntax is used to instantiate a delegate.

Syntax:

`<DelegateName> <objName> = new <DelegateName>(<MethodName>);`

where,

DelegateName: Specifies the name of the delegate.

objName: Specifies the name of the delegate object.

MethodName: Specifies the name of the method to be referenced by the delegate object.

Code Snippet 2 declares a delegate **Calculation** outside the class **Mathematics** and instantiates it in the class.

Code Snippet 2:

```
public delegate int Calculation (int numOne, int numTwo);

class Mathematics
{
    static int Addition(int numOne, int numTwo)
    {
        return (numOne + numTwo);
    }

    static int Subtraction(int numOne, int numTwo)
    {
        return (numOne - numTwo);
    }

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

```
int valOne = 5;  
int valTwo = 23;  
Calculation objCalculation = new Calculation(Addition);  
Console.WriteLine (valOne + " + " + valTwo + " = " +  
objCalculation (valOne, valTwo));  
}  
}
```

In Code Snippet 2, the delegate called **Calculation** is declared outside the class **Mathematics**.

In the **Main()** method, an object of the delegate is created that takes the **Addition()** method as the parameter. The parameter type of the method and that of the delegate is the same, which is type **int**.

Output:

5 + 23 = 28

Note - In recent versions of C#, anonymous functions called 'lambda' expressions are used to create delegates. This is explored in detail in next module.

12.1.4 Using Delegates

A delegate can be declared either before creating the class (having the method to be referenced) or can be defined within the class.

There are four steps to implement delegates in C#. These steps are as follows:

1. Declare a delegate.
2. Create the method to be referenced by the delegate.
3. Instantiate the delegate.
4. Call the method using the object of the delegate.

Each of these step is demonstrated with an example in figure 12.3.

```
class DelegatesDemo
{
    public delegate double Temperature(double temp);

    public static double FahrenheitToCelsius(double temp)
    {
        return ((temp-32) / 9)*5;
    }
    public static void Main()
    {
        Temperature tempConversion = new Temperature(FahrenheitToCelsius);

        double tempF = 96;

        double tempC = tempConversion(tempF);

        Console.WriteLine("Temperature in Fahrenheit = {0:F} ".tempF);

        Console.WriteLine("Temperature in Celsius = {0:F} ".tempC);
    }
}
```

Figure 12.3: Using Delegates

An anonymous method is an inline block of code that can be passed as a delegate parameter. Using anonymous methods, you can avoid creating named methods. Figure 12.4 displays an example of using anonymous methods.

```
void Action()
{
    System.Threading.Thread objThread = new
    System.Threading.Thread
    (delegate()
    {
        Console.Write("Testing... ");
        Console.WriteLine("Threads.");
    });
    objThread.Start();
}
```



Figure 12.4: Anonymous Methods

12.1.5 Delegate-Event Model

The delegate-event model is a programming model that enables a user to interact with a computer and computer-controlled devices using graphical user interfaces. This model consists of:

- An event source, which is the console window in case of console-based applications.
- Listeners that receive the events from the event source.
- A medium that gives the necessary protocol by which every event is communicated.

In this model, every listener must implement a medium for the event that it wants to listen to. Using the medium, every time the source generates an event, the event is notified to the registered listeners.

Consider a guest ringing a doorbell at the doorstep of a home. The host at home listens to the bell and responds to the ringing action by opening the door. Here, the ringing of the bell is an event that resulted in the reaction of opening the door. Similarly, in C#, an event is a generated action that triggers its reaction. For example, pressing `Ctrl+Break` on a console-based server window is an event that will cause the server to terminate.

This event results in storing the information in the database, which is the triggered reaction. Here, the listener is the object that invokes the required method to store the information in the database.

Delegates can be used to handle events. As parameters, they take methods that need to be invoked when events occur. These methods are referred to as the event handlers.

12.1.6 Multiple Delegates

In C#, a user can invoke multiple delegates within a single program. Depending on the delegate name or the type of parameters passed to the delegate, the appropriate delegate is invoked.

Code Snippet 3 demonstrates the use of multiple delegates by creating two delegates `CalculateArea` and `CalculateVolume` that have their return types and parameter types as double.

Code Snippet 3:

```
using System;

public delegate double CalculateArea(double val);

public delegate double CalculateVolume(double val);

class Cube
{
    static double Area(double val)
    {
        return val * val;
    }

    static double Volume(double val)
    {
        return val * val * val;
    }
}
```

```

        return 6 * (val * val);

    }

    static double Volume(double val)
    {
        return (val * val);
    }

    static void Main(string[] args)
    {
        CalculateArea objCalculateArea = new CalculateArea(Area);
        CalculateVolume objCalculateVolume = new CalculateVolume(Volume);
        Console.WriteLine("Surface Area of Cube: " + objCalculateArea(200.32));
        Console.WriteLine("Volume of Cube: " + objCalculateVolume(20.56));
    }
}

```

In Code Snippet 3, when the delegates **CalculateArea** and **CalculateVolume** are instantiated in the **Main()** method, the references of the methods **Area** and **Volume** are passed as parameters to the delegates **CalculateArea** and **CalculateVolume** respectively. The values are passed to the instances of appropriate delegates, which in turn invoke the respective methods.

Figure 12.5 shows the use of multiple delegates.

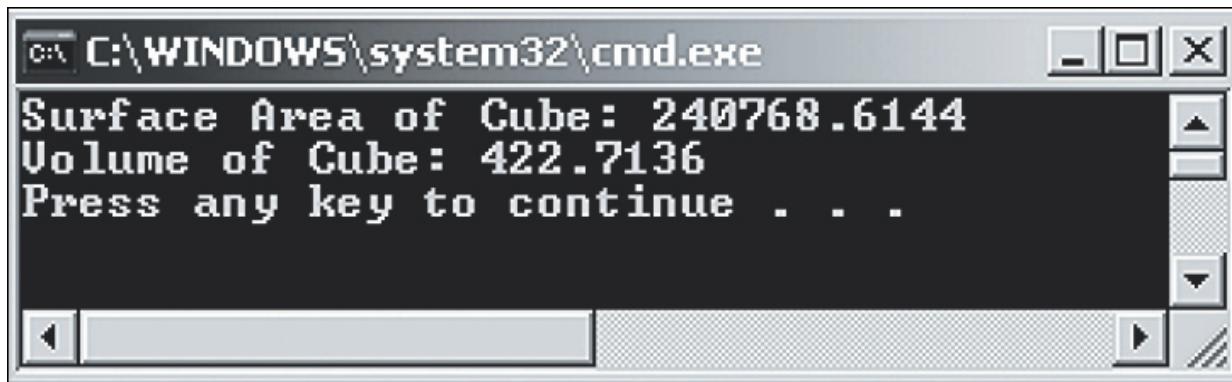


Figure 12.5: Use of Multiple Delegates

12.1.7 Multicast Delegate

A single delegate can encapsulate the references of multiple methods at a time. In other words, a delegate can hold a number of method references. Such delegates are termed as 'Multicast Delegates'. A multicast delegate maintains a list of methods (invocation list) that will be automatically called when the delegate is invoked.

Multicast delegates in C# are sub-types of the `System.MulticastDelegate` class. Multicast delegates are defined in the same way as simple delegates, however, the return type of multicast delegates can only be `void`. If any other return type is specified, a run-time exception will occur. This is because if the delegate returns a value, the return value of the last method in the invocation list of the delegate will become the return type of the delegate. This will result in inappropriate results. Hence, the return type is always `void`.

To add methods into the invocation list of a multicast delegate, the user can use the '+' or the '+=' assignment operator. Similarly, to remove a method from the delegate's invocation list, the user can use the '-' or the '-=' operator. When a multicast delegate is invoked, all the methods in the list are invoked sequentially in the same order in which they have been added.

Code Snippet 4 creates a multicast delegate `Maths`. This delegate encapsulates the reference to the methods `Addition`, `Subtraction`, `Multiplication`, and `Division`.

Code Snippet 4:

```
using System;

public delegate void Maths (int valOne, int valTwo);

class MathsDemo
{
    static void Addition (int valOne, int valTwo)
    {
        int result = valOne + valTwo;
        Console.WriteLine ("Addition: " + valOne + " + " + valTwo + " = " + result);
    }

    static void Subtraction (int valOne, int valTwo)
    {
        int result = valOne - valTwo;
        Console.WriteLine ("Subtraction: " + valOne + " - " + valTwo + " = " + result);
    }
}
```

```
}

static void Multiplication(int valOne, int valTwo)
{
    int result = valOne * valTwo;
    Console.WriteLine("Multiplication: " + valOne + " * " + valTwo + "= " +
        result);
}

static void Division(int valOne, int valTwo)
{
    int result = valOne / valTwo;
    Console.WriteLine("Division: " + valOne + " / " + valTwo + "= " + result);
}

static void Main(string[] args)
{
    Maths objMaths = new Maths(Addition);

    objMaths += new Maths(Subtraction);
    objMaths += new Maths(Multiplication);
    objMaths += new Maths(Division);

    if (objMaths != null)
    {
        objMaths(20, 10);
    }
}
}
```

In Code Snippet 4, the delegate **Maths** is instantiated in the `Main()` method. Once the object is created, methods are added to it using the '`+=`' assignment operator, which makes the delegate a multicast delegate.

Figure 12.6 shows the creation of a multicast delegate.

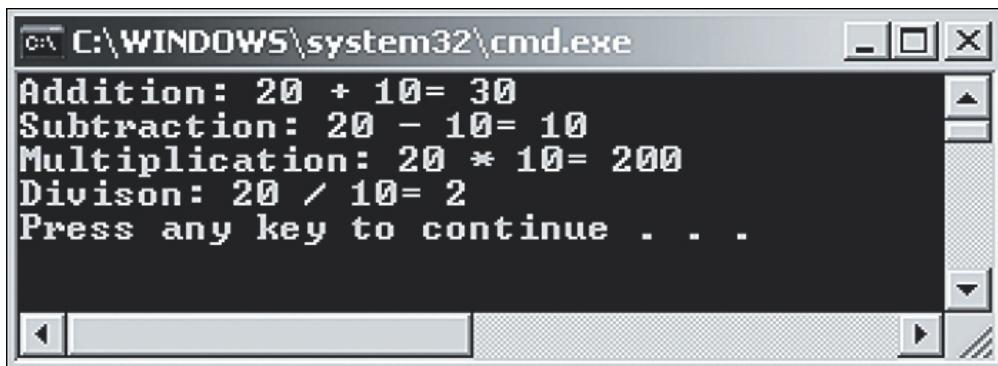


Figure 12.6: Creation of Multicast Delegate

12.1.8 System.Delegate Class

The `Delegate` class of the `System` namespace is a built-in class defined to create delegates in C#. All delegates in C# implicitly inherit from the `Delegate` class. This is because the `delegate` keyword indicates to the compiler that the defined delegate in a program is to be derived from the `Delegate` class. The `Delegate` class provides various constructors, methods, and properties to create, manipulate, and retrieve delegates defined in a program.

Table 12.1 lists the constructors defined in the `Delegate` class.

Constructor	Description
<code>Delegate(object, string)</code>	Calls a method referenced by the object of the class given as the parameter
<code>Delegate(type, string)</code>	Calls a static method of the class given as the parameter

Table 12.1: Constructors of the Delegate Class

Table 12.2 lists the properties defined in the `Delegate` class.

Property	Description
<code>Method</code>	Retrieves the referenced method
<code>Target</code>	Retrieves the object of the class in which the delegate invokes the referenced method

Table 12.2: Properties of the Delegate Class

Table 12.3 lists some of the methods defined in the `Delegate` class.

Method	Description
<code>Clone</code>	Makes a copy of the current delegate
<code>Combine</code>	Merges the invocation lists of the multicast delegates

Method	Description
CreateDelegate	Declares and initializes a delegate
DynamicInvoke	Calls the referenced method at run-time
GetInvocationList	Retrieves the invocation list of the current delegate

Table 12.3: Methods of the Delegate Class

Code Snippet 5 demonstrates the use of some of the properties and methods of the built-in `Delegate` class.

Code Snippet 5:

```
using System;

public delegate void Messenger(int value);

class CompositeDelegates
{
    static void EvenNumbers(int value)
    {
        Console.Write("Even Numbers: ");
        for (int i = 2; i <= value; i += 2)
        {
            Console.Write(i + " ");
        }
    }

    void OddNumbers(int value)
    {
        Console.WriteLine();
        Console.Write("Odd Numbers: ");
        for (int i = 1; i <= value; i += 2)
        {
            Console.Write(i + " ");
        }
    }

    static void Start(int number)
    {
```

```
CompositeDelegates objComposite = new CompositeDelegates();
Messenger objDisplayOne = new Messenger(EvenNumbers);
Messenger objDisplayTwo = new Messenger(objComposite.OddNumbers);
Messenger objDisplayComposite = (Messenger) Delegate.Combine
    (objDisplayOne, objDisplayTwo);
objDisplayComposite(number);
Console.WriteLine();
Object obj = objDisplayComposite.Method.ToString();
if (obj != null)
{
    Console.WriteLine("The delegate invokes an instance method: " + obj);
}
else
{
    Console.WriteLine("The delegate invokes only static methods");
}
static void Main(string[] args)
{
    int value = 0;
    Console.WriteLine("Enter the values till which you want to display even and
        odd numbers");
    try
    {
        value = Convert.ToInt32(Console.ReadLine());
    }
    catch (FormatException objFormat)
    {
        Console.WriteLine("Error: " + objFormat);
    }
}
```

```

    Start(value);
}
}

```

In Code Snippet 5, the delegate **Messenger** is instantiated in the **Start()** method. An instance of the delegate, **objDisplayOne**, takes the static method, **EvenNumbers()**, as a parameter, and another instance of the delegate, **objDisplayTwo**, takes the non-static method, **OddNumbers()**, as a parameter by using the instance of the class. The **Combine()** method merges the delegates provided in the list within the parentheses.

The **Method** property checks whether the program contains instance methods or static methods. If the program contains only static methods, then the **Method** property returns a null value. The **Main()** method allows the user to enter a value. The **Start()** method is called by passing this value as a parameter. This value is again passed to the instance of the class **CompositeDelegates** as a parameter, which in turn invokes both the delegates. The code displays even and odd numbers within the specified range by invoking the appropriate methods.

Figure 12.7 shows the use of some of the properties and methods of the **Delegate** class.

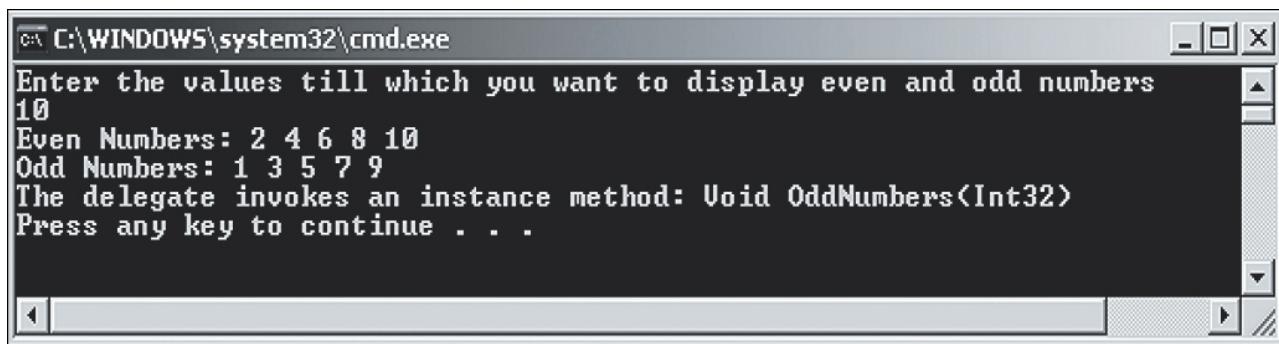


Figure 12.7: Properties and Methods of Delegate Class

12.2 Events

Consider a group of people at a party playing Bingo. When a number is called, the participants check if the number is on their cards whereas the non-participants go about their business, enjoying other activities. If this situation is analyzed from a programmer's perspective, the calling of the number corresponds to the occurrence of an event. The notification about the event is given by the announcer. Here, the people playing the game are paying attention (subscribing) to what the announcer (the source of the event) has to say (notify).

Similarly, in C#, events allow an object (source of the event) to notify other objects (subscribers) about the event (a change having occurred).

Figure 12.8 depicts the concept of events.

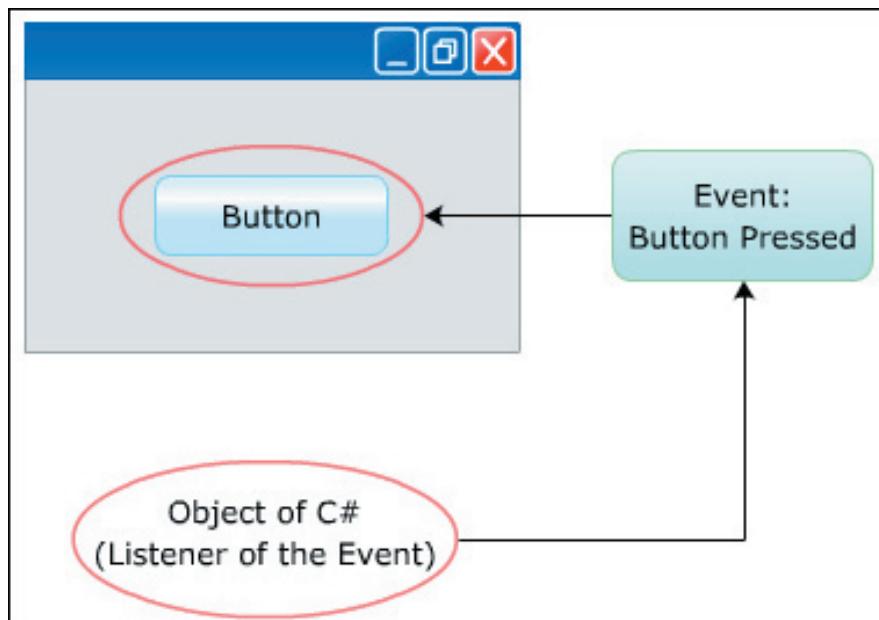


Figure 12.8: Events

12.2.1 Features

An event is a user-generated or system-generated action that enables the required objects to notify other objects or classes to handle the event. Events in C# have the following features:

- They can be declared in classes and interfaces.
- They can be declared as abstract or sealed.
- They can be declared as virtual.
- They are implemented using delegates.

Events can be used to perform customized actions that are not already supported by C#. Events are widely used in creating GUI based applications, where events such as, selecting an item from a list and closing a window are tracked.

12.2.2 Creating and Using Events

There are four steps for implementing events in C#. These are as follows:

1. Define a public delegate for the event.
2. Create the event using the delegate.
3. Subscribe to listen and handle the event.

4. Raise the event.

Events use delegates to call methods in objects that have subscribed to the event. When an event containing a number of subscribers is raised, many delegates will be invoked.

12.2.3 Declaring Events

An event declaration consists of two steps, creating a delegate and creating the event. A delegate is declared using the `delegate` keyword. The delegate passes the parameters of the appropriate method to be invoked when an event is generated. This method is known as the event handler. The event is then declared using the `event` keyword followed by the name of the delegate and the name of the event. This declaration associates the event with the delegate. Figure 12.9 displays the syntax for declaring delegates and events.

Declaring a Delegate:

```
<access_modifier> delegate <return type> <Identifier> (parameters);
```

Declaring an Event:

```
<access_modifier> event <DelegateName> <EventName>;
```

Figure 12.9: Declaring Delegates and Events

An object can subscribe to an event only if the event exists. To subscribe to the event, the object adds a delegate that calls a method when the event is raised. This is done by associating the event handler to the created event, using the **`+= addition assignment`** operator. This is known as subscribing to an event.

To unsubscribe from an event, use the **`-= subtraction assignment`** operator.

The following syntax is used to create a method in the receiver class.

Syntax:

```
<access_modifier> <return_type> <MethodName> (parameters);
```

The following syntax is used to associate the method to the event.

Syntax:

```
<objectName>.<EventName> += new <DelegateName> (MethodName);
```

where,

objectName: Is the object of the class in which the event handler is defined.

Code Snippet 6 associates the event handler to the declared event.

Code Snippet 6:

```
using System;
public delegate void PrintDetails();

class TestEvent
{
    event PrintDetails Print;

    void Show()
    {
        Console.WriteLine("This program illustrate how to subscribe objects
to an event");

        Console.WriteLine("This method will not execute since the event has
not been raised");
    }

    static void Main(string[] args)
    {
        TestEvent objTestEvent = new TestEvent();
        objTestEvent.Print += new PrintDetails(objEvents.Show);
    }
}
```

In Code Snippet 6, the delegate called `PrintDetails()` is declared without any parameters. In the class `TestEvent`, the event `Print` is created that is associated with the delegate. In the `Main()` method, object of the class `TestEvent` is used to subscribe the event handler called `Show()` to the event `Print`.

12.2.4 Raising Events

An event is raised to notify all the objects that have subscribed to the event. Events are either raised by the user or the system. Once an event is generated, all the associated event handlers are executed. The delegate calls all the handlers that have been added to the event. However, before raising an event, it is important for you to create handlers and thus, make sure that the event is associated to the appropriate event handlers. If the event is not associated to any event handler, the declared event is considered to be `null`. Figure 12.10 displays the raising events.

```

public delegate void Display();

class Events
{
    event Display Print;

    void Show()
    {
        Console.WriteLine("This is an event driven program");
    }

    static void Main(string[] args)
    {
        Events objEvents = new Events();
        objEvents.Print += new Display(objEvents.Show);
        objEvents.Print();
    }
}

Output:

This is an event driven program

```

Invoking the Event Handler through the Created Event

Figure 12.10: Raising Events

Code Snippet 7 can be used to check a particular condition before raising the event.

Code Snippet 7:

```

if(condition)
{
    eventMe();
}

```

In Code Snippet 7, if the checked condition is satisfied, the event **eventMe** is raised.

The syntax for raising an event is similar to the syntax for calling a method. When **eventMe** is raised, it will invoke all the delegates of the objects that have subscribed to it. If no objects have subscribed to the event and the event has been raised, an exception is thrown.

12.2.5 Events and Inheritance

Events in C# can only be invoked in the class in which they are declared and defined. Therefore, events cannot be directly invoked by the derived classes. However, events can be invoked indirectly in C# by creating a protected method in the base class that will, in turn, invoke the event defined in the base class.

Code Snippet 8 illustrates how an event can be indirectly invoked.

Code Snippet 8:

```
using System;

public delegate void Display(string msg);

public class Parent
{
    event Display Print;
    protected void InvokeMethod()
    {
        Print += new Display(PrintMessage);
        Check();
    }
    void Check()
    {
        if (Print != null)
        {
            PrintMessage("Welcome to C#");
        }
    }
    void PrintMessage(string msg)
    {
        Console.WriteLine(msg);
    }
}

class Child : Parent
{
    static void Main(string[] args)
    {
        Child objChild = new Child();
        objChild.InvokeMethod();
    }
}
```

```
}
```

In Code Snippet 8, the class **Child** is inherited from the class **Parent**. An event named **Print** is created in the class **Parent** and is associated with the delegate **Display**. The protected method **InvokeMethod()** associates the event with the delegate and passes the method **PrintMessage()** as a parameter to the delegate. The **Check()** method checks whether any method is subscribing to the event. Since the **PrintMessage()** method is subscribing to the **Print** event, this method is called. The **Main()** method creates an instance of the derived class **Child**. This instance invokes the **InvokeMethod()** method, which allows the derived class **Child** access to the event **Print** declared in the base class **Parent**.

Figure 12.11 shows the outcome of invoking the event.



Figure 12.11: Outcome of Invoking the Event

Note - An event can be declared as abstract though only in abstract classes. Such an event must be overridden in the derived classes of the abstract class. Abstract events are used to customize the event when implemented in the derived classes. An event can be declared as sealed in a base class to prevent its invocation from any of the derived classes. A sealed event cannot be overridden in any of the derived classes to ensure safe functioning of the event.

12.3 Collections

A collection is a set of related data that may not necessarily belong to the same data type. It can be set or modified dynamically at run-time. Accessing collections is similar to accessing arrays, where elements are accessed by their index numbers. However, there are differences between arrays and collections in C#.

Table 12.4 lists the differences between arrays and collections.

Arrays	Collections
Cannot be resized at run-time.	Can be resized at run-time.
The individual elements are of the same data type.	The individual elements can be of different data types.
Do not contain any methods for operations on elements.	Contain methods for operations on elements.

Table 12.4: Differences between the Arrays and Collections

12.3.1 System.Collections Namespace

The `System.Collections` namespace in C# allows you to construct and manipulate a collection of objects. This collection can include elements of different data types. The `System.Collections` namespace defines various collections such as dynamic arrays, lists, and dictionaries.

The `System.Collections` namespace consists of classes and interfaces that define the different collections.

Table 12.5 lists the commonly used classes and interfaces in the `System.Collections` namespace.

Class/Interface	Description
<code>ArrayList</code> Class	Provides a collection that is similar to an array except that the items can be dynamically added and retrieved from the list and it can contain values of different types
<code>Stack</code> Class	Provides a collection that follows the Last-In-First-Out (LIFO) principle, which means the last item inserted in the collection, will be removed first
<code>Hashtable</code> Class	Provides a collection of key and value pairs that are arranged, based on the hash code of the key
<code>SortedList</code> Class	Provides a collection of key and value pairs where the items are sorted, based on the keys
<code>IDictionary</code> Interface	Represents a collection consisting of key/value pairs
<code>IDictionaryEnumerator</code> Interface	Lists the dictionary elements
<code>IEnumerable</code> Interface	Defines an enumerator to perform iteration over a collection
<code>ICollection</code> Interface	Specifies the size and synchronization methods for all collections
<code>IEnumerator</code> Interface	Supports iteration over the elements of the collection
<code>IList</code> Interface	Represents a collection of items that can be accessed by their index number

Table 12.5: Classes and Interfaces of the `System.Collections` Namespace

Code Snippet 9 demonstrates the use of the commonly used classes and interfaces of the `System.Collections` namespace.

Code Snippet 9:

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

```
class Employee : DictionaryBase
{
    public void Add(int id, string name)
    {
        Dictionary.Add(id, name);
    }

    public void OnRemove(int id)
    {
        Console.WriteLine("You are going to delete record containing ID: " + id);
        Dictionary.Remove(id);
    }

    public void GetDetails()
    {
        IDictionaryEnumerator objEnumerate = Dictionary.GetEnumerable();
        while (objEnumerate.MoveNext())
        {
            Console.WriteLine(objEnumerate.Key.ToString() + "\t\t" +
                objEnumerate.Value);
        }
    }

    static void Main(string[] args)
    {
        Employee objEmployee = new Employee();
        objEmployee.Add(102, "John");
        objEmployee.Add(105, "James");
        objEmployee.Add(106, "Peter");
        Console.WriteLine("Original values stored in Dictionary");
        objEmployee.GetDetails();
        objEmployee.OnRemove(106);
    }
}
```

```

Console.WriteLine("Modified values stored in
Dictionary");
objEmployee.GetDetails();
}
}

```

In Code Snippet 9, the class **Employee** is inherited from the **DictionaryBase** class. The **DictionaryBase** class is an abstract class. The details of the employees are inserted using the methods present in the **DictionaryBase** class. The user-defined **Add()** method takes two parameters, namely **id** and **name**. These parameters are passed onto the **Add()** method of the **Dictionary** class. The **Dictionary** class stores these values as a key/value pair. The **OnRemove()** method of **DictionaryBase** is overridden. It takes a parameter specifying the key whose key/value pair is to be removed from the **Dictionary** class. This method then prints a warning statement on the console before deleting the record from the **Dictionary** class. The **Dictionary.Remove()** method is used to remove the key/value pair. The **GetEnumerator()** method returns an **IDictionaryEnumerator**, which is used to traverse through the list.

Figure 12.12 displays the output of the example.

```

C:\WINDOWS\system32\cmd.exe
Original values stored in Dictionary
106      Peter
105      James
102      John
You are going to delete record containing ID: 106
Modified values stored in Dictionary
105      James
102      John
Press any key to continue . . .

```

Figure 12.12: System.Collections Namespace Example

12.3.2 System.Collections.Generic Namespace

Consider an online application form used by students to register for an examination conducted by a university. The application form can be used to apply for examination of any course offered by the university. Similarly, in C#, generics allow you to define data structures that consist of functionalities which can be implemented for any data type. Thus, generics allow you to reuse a code for different data types.

To create generics, you should use the built-in classes of the **System.Collections.Generic** namespace. These classes ensure type-safety, which is a feature of C# that ensures a value is treated as the type with which it is declared.

Note - The System.Collections.Generic namespace is similar to the System.Collections namespace as both allow you to create collections. However, generic collections are type-safe.

12.3.3 Classes and Interfaces

The System.Collections.Generic namespace consists of classes and interfaces that define the different generic collections.

→ Classes

The System.Collections.Generic namespace consists of classes that allow you to create type-safe collections. Table 12.6 lists the commonly used classes in the System.Collections.Generic namespace.

Class	Description
List<T>	Provides a generic collection of items that can be dynamically resized
Stack<T>	Provides a generic collection that follows the LIFO principle, which means that the last item inserted in the collection will be removed first
Queue<T>	Provides a generic collection that follows the FIFO principle, which means that the first item inserted in the collection will be removed first
Dictionary<K, V>	Provides a generic collection of keys and values
SortedDictionary<K, V>	Provides a generic collection of sorted key and value pairs that consist of items sorted according to their key
LinkedList<T>	Implements the doubly linked list by storing elements in it

Table 12.6: Classes in System.Collections.Generic Namespace

→ Interfaces and Structures

The System.Collections.Generic namespace consists of interfaces and structures that can be implemented to create type-safe collections. Table 12.7 lists some of the commonly used ones.

Interface	Description
ICollection Interface	Defines methods to control the different generic collections
IEnumerable Interface	Is an interface that defines an enumerator to perform an iteration of a collection of a specific type
IComparer Interface	Is an interface that defines a method to compare two objects

Interface	Description
IDictionary Interface	Represents a generic collection consisting of the key and value pairs
IEnumerator Interface	Supports simple iteration over elements of a generic collection
IList Interface	Represents a generic collection of items that can be accessed using the index position
Dictionary.Enumerator Structure	Lists the elements of a Dictionary
Dictionary.KeyCollection.Enumerator Structure	Lists the elements of a Dictionary.KeyCollection
Dictionary.ValueCollection.Enumerator Structure	Lists the elements of a Dictionary.ValueCollection
KeyValuePair Structure	Defines a key/value pair

Table 12.7: Interfaces and Structures in the System.Collections.Generic Namespace

Code Snippet 10 demonstrates the use of the commonly used classes, interfaces, and structures of the System.Collection.Generic namespace.

Code Snippet 10:

```
using System;
using System.Collections;
using System.Collections.Generic;
class Student : IEnumerable
{
    LinkedList<string> objList = new LinkedList<string>();
    public void StudentDetails()
    {
        objList.AddFirst("James");
        objList.AddFirst("John");
        objList.AddFirst("Patrick");
        objList.AddFirst("Peter");
        objList.AddFirst("James");
        Console.WriteLine("Number of elements stored in the list: "
            + objList.Count);
    }
}
```

```
}

public void Display(string name)
{
    LinkedListNode<string> objNode;
    int count = 0;
    for (objNode = objList.First; objNode != null; objNode =
        objNode.Next)
    {
        if (objNode.Value.Equals(name))
        {
            count++;
        }
    }
    Console.WriteLine("The value " + name + " appears " + count
        + " times in the list");
}

public IEnumerator GetEnumerator()
{
    return objList.GetEnumerator();
}

static void Main(string[] args)
{
    Student objStudent = new Student();
    objStudent.StudentDetails();
    foreach (string str in objStudent)
    {
        Console.WriteLine(str);
    }
    objStudent.Display("James");
}
```

In Code Snippet 10, the `student` class implements the `IEnumerable` interface. A doubly-linked list of `string` type is created. The `StudentDetails()` method is defined to insert values in the linked list. The `AddFirst()` method of the `LinkedList` class is used to insert values in the linked list. The `Display()` method accepts a single string argument that is used to search for a particular value. A `LinkedListNode` class reference of `string` type is created inside the `Display()` method. This reference is used to traverse through the linked list. Whenever a match is found for the string argument accepted in the `Display()` method, a counter is incremented. This counter is then used to display the number of times the specified string has occurred in the linked list. The `GetEnumerator()` method is implemented, which returns an `IEnumerator`. The `IEnumerator` is used to traverse through the list and display all the values stored in the linked list.

Figure 12.13 displays the `System.Collection.Generic` namespace example.

```
C:\WINDOWS\system32\cmd.exe
Number of elements stored in the list: 5
James
Peter
Patrick
John
James
The value James appears 2 times in the list
Press any key to continue . . .
```

Figure 12.13: System.Collections.Generic Namespace Example

12.3.4 ArrayList Class

The `ArrayList` class is a variable-length array, that can dynamically increase or decrease in size. Unlike the `Array` class, this class can store elements of different data types. The `ArrayList` class allows you to specify the size of the collection, during program execution.

The `ArrayList` class allows you to define the capacity that specifies the number of elements an array list can contain. However, the default capacity of an `ArrayList` class is 16. If the number of elements in the list reaches the specified capacity, the capacity of the list gets doubled automatically. It can accept null values and can also include duplicate elements.

The `ArrayList` class allows you to add, modify, and delete any type of element in the list even at run-time. The elements in the `ArrayList` can be accessed by using the index position. While working with the `ArrayList` class, you need not bother about freeing up the memory. The `ArrayList` class consists of different methods and properties. These methods and properties are used to add and manipulate the elements of the list.

→ Methods

The methods of the `ArrayList` class allow you to perform actions such as adding, removing, and copying elements in the list. Table 12.8 displays the commonly used methods of the `ArrayList` class.

Method	Description
Add	Adds an element at the end of the list
Remove	Removes the specified element that has occurred for the first time in the list
RemoveAt	Removes the element present at the specified index position in the list
Insert	Inserts an element into the list at the specified index position
Contains	Determines the existence of a particular element in the list
IndexOf	Returns the index position of an element occurring for the first time in the list
Reverse	Reverses the values stored in the <code>ArrayList</code>
Sort	Rearranges the elements in an ascending order

Table 12.8: Methods of `ArrayList` Class

→ Properties

The properties of the `ArrayList` class allow you to count or retrieve the elements in the list. Table 12.9 displays the commonly used properties of the `ArrayList` class.

Property	Description
Capacity	Specifies the number of elements the list can contain
Count	Determines the number of elements present in the list
Item	Retrieves or sets value at the specified position

Table 12.9: Properties of `ArrayList` Class

Code Snippet 11 demonstrates the use of the methods and properties of the `ArrayList` class.

Code Snippet 11:

```
using System;
using System.Collections;
class ArrayCollection
{
    static void Main(string[] args)
    {
        ArrayList objArray = new ArrayList();
```

```
objArray.Add("John");
objArray.Add("James");
objArray.Add("Peter");
objArray.RemoveAt(2);
objArray.Insert(2, "Williams");
Console.WriteLine("Capacity: " + objArray.Capacity);
Console.WriteLine("Count: " + objArray.Count);
Console.WriteLine();
Console.WriteLine("Elements of the ArrayList");
foreach (string str in objArray)
{
    Console.WriteLine(str);
}
}
```

In Code Snippet 11, the `Add()` method inserts values into the instance of the class at different index positions. The `RemoveAt()` method removes the value James from the index position 2 and the `Insert()` method inserts the value Williams at the index position 2. The `WriteLine()` method is used to display the number of elements the list can contain and the number of elements present in the list using the `Capacity` and `Count` properties respectively.

Output:

```
Capacity: 4
Count: 3
Elements of the ArrayList
John
James
Williams
```

Note - When you try referencing an element at a position greater than the list's size, the C# compiler generates an error.

Code Snippet 12 demonstrates the use of methods of the `ArrayList` class.

Code Snippet 12:

```
using System;
using System.Collections;

class Customers
{
    static void Main(string[] args)
    {
        ArrayList objCustomers = new ArrayList();
        objCustomers.Add("Nicole Anderson");
        objCustomers.Add("Ashley Thomas");
        objCustomers.Add("Garry White");
        Console.WriteLine("FixedSize : " +
            objCustomersFixedSize);
        Console.WriteLine("Count : " + objCustomers.Count);
        Console.WriteLine("List of customers:");
        foreach (string names in objCustomers)
        {
            Console.WriteLine("{0}", names);
        }
        objCustomers.Sort();
        Console.WriteLine("\nList of customers after
            sorting:");
        foreach (string names in objCustomers)
        {
            Console.WriteLine("{0}", names);
        }
        objCustomers.Clear();
```

```

        Console.WriteLine("Count after removing all elements
: " + objCustomers.Count);
}
}
}

```

In Code Snippet 12, the `Add()` method inserts value at the end of the `ArrayList`. The values inserted in the array are displayed in the same order before the `Sort()` method is used. The `Sort()` method then displays the values in the sorted order. The `FixedSize()` property checks whether the array is of a fixed size. When the `Reverse()` method is called, it displays the values in the reverse order. The `Clear()` method deletes all the values from the `ArrayList` class.

Figure 12.14 displays the use of methods of the `ArrayList` class.

```

C:\WINDOWS\system32\cmd.exe
Fixed Size : False
Count : 3
List of customers:
Nicole Anderson
Ashley Thomas
Garry White

List of customers after sorting:
Ashley Thomas
Garry White
Nicole Anderson

List of customers after reversing:
Nicole Anderson
Garry White
Ashley Thomas
Count after removing removing all elements : 0
Press any key to continue . . .

```

Figure 12.14: Methods of `ArrayList` Class

12.3.5 *Hashtable Class*

Consider the reception area of a hotel where you find the keyholder storing a bunch of keys. Each key in the keyholder uniquely identifies a room and thus, each room is uniquely identified by its key.

Figure 12.15 demonstrates a real-world example of unique keys.

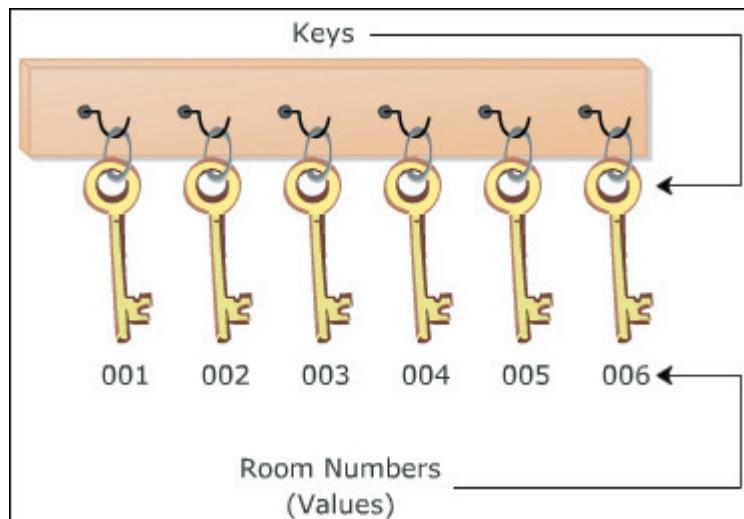


Figure 12.15: Unique Keys

Similar to the keyholder, the `Hashtable` class in C# allows you to create collections in the form of keys and values. It generates a hashtable which associates keys with their corresponding values. The `Hashtable` class uses the hashtable to retrieve values associated with their unique key.

The hashtable generated by the `Hashtable` class uses the hashing technique to retrieve the corresponding value of a key. Hashing is a process of generating the hash code for the key. The code is used to identify the corresponding value of the key.

The `Hashtable` object takes the key to search the value, performs a hashing function and generates a hash code for that key. When you search for a particular value using the key, the hash code is used as an index to locate the desired record. For example, a student name can be used as a key to retrieve the student id and the corresponding residential address.

Figure 12.16 represents the Hashtable.

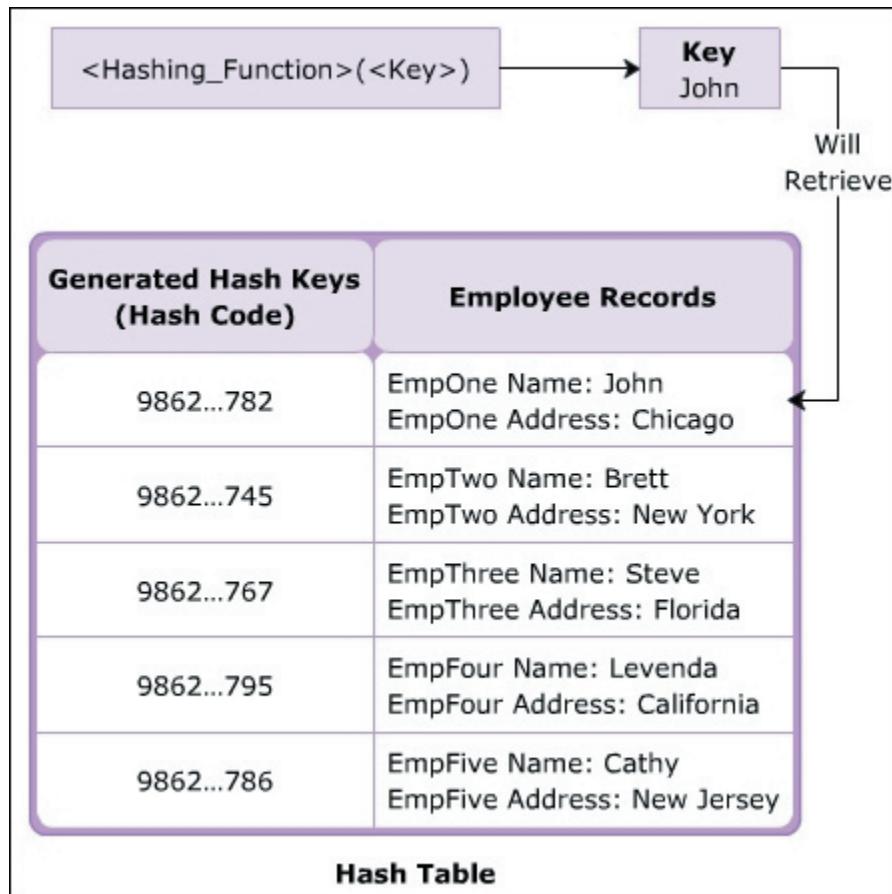


Figure 12.16: Representation of a Hashtable

The `Hashtable` class consists of different methods and properties that are used to add and manipulate the data within the hashtable.

The methods of the `Hashtable` class allow you to perform certain actions on the data in the hashtable. Table 12.10 displays the commonly used methods of the `Hashtable` class.

Method	Description
Add	Adds an element with the specified key and value
Remove	Removes the element having the specified key
CopyTo	Copies elements of the hashtable to an array at the specified index
ContainsKey	Checks whether the hashtable contains the specified key
ContainsValue	Checks whether the hashtable contains the specified value
GetEnumerator	Returns an <code>IDictionaryEnumerator</code> that traverses through the <code>Hashtable</code>

Table 12.10: Methods of Hashtable Class

→ Properties

The properties of the `Hashtable` class allow you to access and modify the data in the hashtable. Table 12.11 displays the commonly used properties of the `Hashtable` class.

Property	Description
Count	Specifies the number of key and value pairs in the hashtable
Item	Specifies the value, adds a new value or modifies the existing value for the specified key
Keys	Provides an <code>ICollection</code> consisting of keys in the hashtable
Values	Provides an <code>ICollection</code> consisting of values in the hashtable
IsReadOnly	Checks whether the <code>Hashtable</code> is read-only

Table 12.11: Properties of `Hashtable` Class

Code Snippet 13 demonstrates the use of the methods and properties of the `Hashtable` class.

Code Snippet 13:

```
using System;
using System.Collections;
class HashCollection
{
    static void Main(string[] args)
    {
        Hashtable objTable = new Hashtable();
        objTable.Add(001, "John");
        objTable.Add(002, "Peter");
        objTable.Add(003, "James");
        objTable.Add(004, "Joe");
        Console.WriteLine("Number of elements in the hashtable: " +
        objTable.Count);
        ICollection objCollection = objTable.Keys;
        Console.WriteLine("Original values stored in hashtable are:
        ");
        foreach (int i in objCollection)
        {
            Console.WriteLine(i + ":" + objTable[i]);
        }
    }
}
```

```
        }

        if (objTable.ContainsKey(002))

        {

            objTable[002] = "Patrick";

        }

        Console.WriteLine("Values stored in the hashtable after

            removing values");

        foreach (int i in objCollection)

        {

            Console.WriteLine(i + ":" + objTable[i]);

        }

    }

}
```

In Code Snippet 13, the `Add()` method inserts the keys and their corresponding values into the instance. The `Count` property displays the number of elements in the hashtable. The `Keys` property provides the number of keys to the instance of the `ICollection` interface. The `ContainsKey()` method checks whether the hashtable contains the specified key. If the hashtable contains the specified key, `002`, the default `Item` property that is invoked using the square bracket notation (`[]`) replaces the value `Peter` to the value `Patrick`. This output is in the descending order of the key. However, the output may not always be displayed in this order. It could be either in ascending or random orders depending on the hash code.

Output:

```
Number of elements in the hashtable: 4

Original values stored in hashtable are:

4 : Joe
3 : James
2 : Peter
1 : John

Values stored in the hashtable after removing values

4 : Joe
3 : James
2 : Patrick
1 : John
```

Code Snippet 14 demonstrates the use of methods and properties of the `Hashtable` class.

Code Snippet 14:

```
using System;
using System.Collections;
class Authors
{
    static void Main(string[] args)
    {
        Hashtable objAuthors = new Hashtable();
        objAuthors.Add("AU01", "John");
        objAuthors.Add("AU04", "Mary");
        objAuthors.Add("AU09", "William");
        objAuthors.Add("AU11", "Rodrick");
        Console.WriteLine("Read-only : " +
            objAuthors.IsReadOnly);
        Console.WriteLine("Count : " + objAuthors.Count);
        IDictionaryEnumerator objCollection =
            objAuthors.GetEnumerator();
        Console.WriteLine("List of authors:\n");
        Console.WriteLine("Author ID \t Name");
        while (objCollection.MoveNext())
        {
            Console.WriteLine(objCollection.Key + "\t" +
                objCollection.Value);
        }
        if (objAuthors.Contains("AU01"))
        {
            Console.WriteLine("\nList contains author with id AU01");
        }
        else
```

```
{
    Console.WriteLine("\nList does not contain author
with id AU01");
}
}
}
```

In Code Snippet 14 , the `Add()` method inserts values in the `Hashtable`. The `IsReadOnly()` method checks whether the values in the array can be modified or not. The `Contains()` method checks whether the value `AU01` is present in the list.

Figure 12.17 displays the `Hashtable` example.

```
C:\WINDOWS\system32\cmd.exe
Read-only : False
Count : 4
List of authors:
Author ID      Name
AU04           Mary
AU01           John
AU09           William
AU11           Rodrick

List contains author with id AU01
Press any key to continue . . .
```

Figure 12.17: Hashtable Example

12.3.6 SortedList Class

The `SortedList` class represents a collection of key and value pairs where elements are sorted according to the key. By default, the `SortedList` class sorts the elements in ascending order, however, this can be changed if an `IComparable` object is passed to the constructor of the `SortedList` class. These elements are either accessed using the corresponding keys or the index numbers.

If you access elements using their keys, the `SortedList` class behaves like a hashtable, whereas if you access elements based on their index number, it behaves like an array.

The `SortedList` class consists of different methods and properties that are used to add and manipulate the data in the sorted list.

→ Methods

The methods of the `SortedList` class allow you to perform certain actions on the data in the sorted list.

Table 12.12 displays the commonly used methods of the `SortedList` class.

Method	Description
Add	Adds an element to the sorted list with the specified key and value
Remove	Removes the element having the specified key from the sorted list
GetKey	Returns the key at the specified index position
GetByIndex	Returns the value at the specified index position
ContainsKey	Checks whether the instance of the <code>SortedList</code> class contains the specified key
ContainsValue	Checks whether the instance of the <code>SortedList</code> class contains the specified value
RemoveAt	Deletes the element at the specified index

Table 12.12: `SortedList` Class Methods

→ Properties

The properties of the `SortedList` class allow you to access and modify the data in the sorted list. Table 12.13 displays the commonly used properties of the `SortedList` class.

Property	Description
Capacity	Specifies the number of elements the sorted list can contain
Count	Specifies the number of elements in the sorted list
Item	Returns the value, adds a new value or modifies the existing value for the specified key
Keys	Returns the keys in the sorted list
Values	Returns the values in the sorted list

Table 12.13: `SortedList` Class Properties

Code Snippet 15 demonstrates the use of the methods and properties of the `SortedList` class.

Code Snippet 15:

```
using System;
using System.Collections;
class SortedCollection
{
    static void Main(string[] args)
    {
```

```
SortedList objSortList = new SortedList();  
objSortList.Add("John", "Administration");  
objSortList.Add("Jack", "Human Resources");  
objSortList.Add("Peter", "Finance");  
objSortList.Add("Joel", "Marketing");  
Console.WriteLine("Original values stored in the  
sortedlist");  
Console.WriteLine("Key \t\tValues");  
for (int i=0; i<objSortList.Count; i++)  
{  
    Console.WriteLine(objSortList.GetKey(i) + "\t\t" +  
        objSortList.GetByIndex(i));  
}  
if (!objSortList.ContainsKey("Jerry"))  
{  
    objSortList.Add("Jerry", "Construction");  
}  
objSortList["Peter"] = "Engineering";  
objSortList["Jerry"] = "Information Technology";  
Console.WriteLine();  
Console.WriteLine("Updated values stored in hashtable");  
Console.WriteLine("Key \t\tValues");  
for (int i=0; i<objSortList.Count; i++)  
{  
    Console.WriteLine(objSortList.GetKey(i) + "\t\t" +  
        objSortList.GetByIndex(i));  
}  
}
```

In Code Snippet 15, the `Add()` method inserts keys and their corresponding values into the instance and the `Count` property counts the number of elements in the sorted list. The `GetKey()` method returns the keys in the sorted order from the sorted list while the `GetByIndex()` method returns

the values at the specified index position. If the sorted list does not contain the specified key, Jerry, then the Add () method adds the key, Jerry with its corresponding value.

The default Item property that is invoked using the square bracket notation ([])) replaces the values associated with the specified keys, Peter and Jerry.

Output:

Original values stored in the sorted list

Key Values

Jack Human Resources

Joel Marketing

John Administration

Peter Finance

Updated values stored in hashtable

Key Values

Jack Human Resources

Jerry Information Technology

Joel Marketing

John Administration

Peter Engineering

Code Snippet 16 demonstrates the use of methods in the SortedList class.

Code Snippet 16:

```
using System;
using System.Collections;
class Countries
{
    static void Main(string[] args)
    {
        SortedList objCountries = new SortedList();
        objCountries.Add("UK", "United Kingdom");
        objCountries.Add("GER", "Germany");
        objCountries.Add("USA", "United States of America");
        objCountries.Add("AUS", "Australia");
```

```
Console.WriteLine("Read-only : " +  
objCountries.IsReadOnly);  
  
Console.WriteLine("Count : " + objCountries.Count);  
  
Console.WriteLine("List of countries:\n");  
  
Console.WriteLine("Country Code \t Name");  
  
for (int i = 0; i < objCountries.Count; i++)  
{  
  
    Console.WriteLine(objCountries.GetKey(i) + "\t\t"  
    " + objCountries.GetByIndex(i));  
}  
  
objCountries.RemoveAt(1);  
  
Console.WriteLine("\nList of countries after removing  
element at index 1:\n");  
  
Console.WriteLine("Country Code \t Name");  
  
for (int i = 0; i < objCountries.Count; i++)  
{  
  
    Console.WriteLine(objCountries.GetKey(i) + "\t\t"  
    " + objCountries.GetByIndex(i));  
}  
  
}
```

Figure 12.18 displays the `SortedList` class example.

```

C:\WINDOWS\system32\cmd.exe
Read-only : False
Count : 4
List of countries:
Country Code      Name
AUS               Australia
GER               Germany
UK                United Kingdom
USA               United States of America

List of countries after removing element at index 1:
Country Code      Name
AUS               Australia
UK                United Kingdom
USA               United States of America
Press any key to continue . .

```

Figure 12.18: `SortedList` Class Example

In Code Snippet 16, the `Add()` method inserts values in the list. The `IsReadOnly()` method checks whether the values in the list can be modified or not. The `GetByIndex()` method returns the value at the specified index. The `RemoveAt()` method removes the value at the specified index.

12.3.7 Dictionary Generic Class

The `System.Collections.Generic` namespace contains a vast number of generic collections. One of the most commonly used among these is the `Dictionary` generic class. It consists of a generic collection of elements organized in key and value pairs. It maps the keys to their corresponding values. Unlike other collections in the `System.Collections` namespace, it is used to create a collection of a single data type at a time.

Every element that you add to the dictionary consists of a value, which is associated with its key. You can retrieve a value from the dictionary by using its key.

The following syntax declares a `Dictionary` generic class.

Syntax:

`Dictionary<TKey, TValue>`

where,

TKey: Is the type parameter of the keys to be stored in the instance of the Dictionary class.

TValue: Is the type parameter of the values to be stored in the instance of the Dictionary class.

Note - The Dictionary class does not allow null values as elements. The capacity of the Dictionary class is the number of elements that it can hold. However, as the elements are added, the capacity is automatically increased.

The Dictionary generic class consists of different methods and properties that are used to add and manipulate elements in a collection.

→ Methods

The methods of the Dictionary generic class allow you to perform certain actions on the data in the collection. Table 12.14 displays the commonly used methods of the Dictionary generic class.

Method	Description
Add	Adds the specified key and value in the collection
Remove	Removes the value associated with the specified key
ContainsKey	Checks whether the collection contains the specified key
ContainsValue	Checks whether the collection contains the specified value
GetEnumerator	Returns an enumerator that traverses through the Dictionary
GetType	Retrieves the Type of the current instance

Table 12.14: Methods of Dictionary Generic Class

→ Properties

The properties of the Dictionary generic class allow you to modify the data in the collection. Table 12.15 displays the commonly used properties of the Dictionary generic class.

Property	Description
Count	Determines the number of key and value pairs in the collection
Item	Returns the value, adds a new value or modifies the existing value for the specified key
Keys	Returns the collection containing the keys
Values	Returns the collection containing the values

Table 12.15: Properties of Dictionary Generic Class

Code Snippet 17 demonstrates the use of the methods and properties of the Dictionary class.

Code Snippet 17:

```
using System;
using System.Collections;
class DictionaryCollection
{
    static void Main(string[] args)
    {
        Dictionary<int, string> objDictionary = new Dictionary<int,
        string>();
        objDictionary.Add(25, "HardDisk");
        objDictionary.Add(30, "Processor");
        objDictionary.Add(15, "MotherBoard");
        objDictionary.Add(65, "Memory");
        ICollection objCollect = objDictionary.Keys;
        Console.WriteLine("Original values stored in the collection");
        Console.WriteLine("Keys \t Values");
        Console.WriteLine("-----");
        foreach (int i in objCollect)
        {
            Console.WriteLine(i + "\t" + objDictionary[i]);
        }
        objDictionary.Remove(65);
        Console.WriteLine();
        if (objDictionary.ContainsKey("Memory"))
        {
            Console.WriteLine("Value Memory could not be deleted");
        }
        else
        {
            Console.WriteLine("Value Memory deleted successfully");
        }
    }
}
```

```

    }

    Console.WriteLine();

    Console.WriteLine("Values stored after removing element");

    Console.WriteLine("Keys \t Values");

    Console.WriteLine("-----");

    foreach (int i in objCollect)

    {

        Console.WriteLine(i + "\t" + objDictionary[i]);

    }

}

}

```

In Code Snippet 17, the `Dictionary` class is instantiated by specifying the `int` and `string` data types as the two parameters. The `int` data type indicates the keys and the `string` data type indicates values. The `Add()` method inserts keys and values into the instance of the `Dictionary` class. The `Keys` property provides the number of keys to the instance of the `ICollection` interface. The `ICollection` interface defines the size and synchronization methods to manipulate the specified generic collection. The `Remove()` method removes the value `Memory` by specifying the key associated with it, which is `65`. The `ContainsValue()` method checks whether the value `Memory` is present in the collection and displays the appropriate message.

Output:

```

Original values stored in the collection
Keys Values
-----
25 Hard Disk
30 Processor
15 MotherBoard
65 Memory
Value Memory deleted successfully
Values stored after removing element
Keys Values
-----
25 Hard Disk
30 Processor
15 MotherBoard

```

Code Snippet 18 demonstrates the use of methods in Dictionary generic class.

Code Snippet 18:

```
using System;
using System.Collections;
using System.Collections.Generic;
class Car
{
    static void Main(string[] args)
    {
        Dictionary<int, string> objDictionary = new
        Dictionary<int, string>();
        objDictionary.Add(201, "Gear Box");
        objDictionary.Add(220, "Oil Filter");
        objDictionary.Add(330, "Engine");
        objDictionary.Add(305, "Radiator");
        objDictionary.Add(303, "Steering");
        Console.WriteLine("Dictionary class contains values of type");
        Console.WriteLine(objDictionary.GetType());
        Console.WriteLine("Keys \t\tValues");
        Console.WriteLine("_____");
        IDictionaryEnumerator objDictionayEnumerator =
        objDictionary.GetEnumerator();
        while (objDictionayEnumerator.MoveNext())
        {
            Console.WriteLine(objDictionayEnumerator.Key.ToString()
            + "\t\t" + objDictionayEnumerator.Value);
        }
    }
}
```

In Code Snippet 18, the `Add()` method inserts values into the list. The `GetType()` method returns the type of the object.

Figure 12.19 displays the use of `Dictionary` generic class.

```
C:\WINDOWS\system32\cmd.exe
Dictionary class contains values of type
System.Collections.Generic.Dictionary`2[System.Int32,System.String]

Keys          Values
201           Gear Box
220           Oil Filter
330           Engine
305           Radiator
303           Steering
Press any key to continue . . .
```

Figure 12.19: Use of Dictionary Generic Class

12.3.8 Collection Initializers

Collection initializers allow adding elements to the collection classes of the `System.Collections` and `System.Collections.Generic` namespaces that implements the `IEnumerable` interface using element initializers. The element initializers can be a simple value, an expression, or an object initializer. When a programmer uses collection initializer, the programmer is not required to provide multiple `Add()` methods to add elements to the collection, making the code concise. It is the responsibility of the compiler to provide the `Add()` methods when the program is compiled.

Code Snippet 19 uses a collection initializer to initialize an `ArrayList` with integers.

Code Snippet 19:

```
using System;
using System.Collections;

class Car
{
    static void Main (string [] args)
    {
        ArrayList nums=new ArrayList{1,2,3*6,4,5};
```

```
foreach (int num in nums)
{
    Console.WriteLine("{0}", num);
}
```

In Code Snippet 19, the `Main()` method uses collection initializer to create an `ArrayList` object initialized with integer values and expressions that evaluates to integer values.

Output:

```
1
2
18
4
5
```

As collection often contains objects, collection initializers accept object initializers to initialize a collection. Code Snippet 20 shows a collection initializer that initializes a generic `Dictionary` object with an integer keys and `Employee` objects.

Code Snippet 20:

```
using System;
using System.Collections;
using System.Collections.Generic;
class Employee
{
    public String Name { get; set; }
    public String Designation { get; set; }
}
class CollectionInitializerDemo
{
    static void Main(string[] args)
{
```

```
Dictionary<int, Employee> dict = new Dictionary<int,  
Employee>() {  
    { 1, new Employee {Name="Andy Parker",  
Designation="Sales Person"} },  
    { 2, new Employee {Name="Patrick Elvis",  
Designation="Marketing Manager"} }  
};  
}  
}
```

Code Snippet 20 creates an `Employee` class with two public properties: `Name` and `Designation`. The `Main()` method creates a `Dictionary<int, Employee>` object and encloses the collection initializers within a pair of braces. For each element, added to the collection, the innermost pair of braces encloses the object initializer of the `Employee` class.

12.4 Check Your Progress

1. Which of these statements about delegates are true?

(A)	Delegates are used to hide static methods.		
(B)	Delegates are reference types used to refer to multiple methods at once.		
(C)	Delegates are declared with a return type that may or may not be the same as the return type of the referenced method.		
(D)	Delegates are declared using the <code>delegates</code> keyword.		
(E)	Delegates are used to invoke overridden methods.		

(A)	A, B	(C)	C
(B)	B, C, D	(D)	A, D

2. You are trying to convert the weight of 12.2 kilograms to pounds using delegates in a C# application. Which of the following codes will help you to achieve this?

(A)	<pre> delegate double Conversion (double temp) ; class Delegates { static double WeightConversion (double temp) { return (temp * 2.2); } static void Main (string [] args) { double weightKg = 12.2; Conversion objConvert = new Delegate (WeightConversion); Console.WriteLine ("Weight in Kilograms: " + weightKg); Console.Write ("Corresponding weight in pounds: "); Console.WriteLine (objConvert (weightKg)); } } delegate double Conversion (double temp); class Delegates </pre>
-----	---

(B)

```
{
static double WeightConversion(double temp)
{
return (temp * 2.2);
}
static void Main(string[] args)
{
double weightKg = 12.2;
Conversion objConvert = new
Conversion(WeightConversion(weightKg));
Console.WriteLine("Weight in Kilograms: " + weightKg);
Console.Write("Corresponding weight in pounds: ");
}
}
```

(C)

```
delegate double Conversion(double temp);

class Delegates
{
static double WeightConversion(double temp)
{
return (temp * 2.2);
}
static void Main(string[] args)
{
double weightKg = 12.2;
Conversion objConvert = new
Conversion(WeightConversion);
Console.WriteLine("Weight in Kilograms: " + weightKg);
Console.Write("Corresponding weight in pounds: ");
Console.WriteLine(objConvert(weightKg));
}
}
```

```

delegate double Conversion (double temp) ;

class Delegates
{
    static double WeightConversion (double temp)
    {
        return (temp * 2.2);
    }

    static void Main (string [] args)
    {
        double weightKg = 12.2;
        Conversion objConvert = new Delegate (WeightConversion);
        Console.WriteLine ("Weight in Kilograms: " + weightKg);
        Console.Write ("Corresponding weight in pounds: ");
        Console.WriteLine (objConvert (weightKg));
    }
}

```

(A)	A	(C)	C
(B)	B	(D)	D

3. Which of these statements about delegates are true?

(A)	Events can be implemented without using delegates.		
(B)	Events can be used to perform customized actions that are not already present in the language construct.		
(C)	Events can be declared as abstract or sealed.		
(D)	Events cannot be declared in interfaces.		
(E)	All the objects that have subscribed to an event can be notified by raising the event.		

(A)	A	(C)	B, C, E
(B)	B, C, D	(D)	A, D

4. You are trying to display the output “This is a demonstration of events” using events in a C# application. Which of the following codes will help you to achieve this?

(A)

```
public delegate void DisplayMessage(string msg);

class EventProgram

{
    event DisplayMessage Print;

    void Show(string msg)
    {
        Console.WriteLine(msg);
    }

    static void Main(string[] args)
    {
        EventProgram objEvent = new EventProgram();

        objEvent.Print += new DisplayMessage(objEvent.Show);

        objEvent.Print("This is a demonstration of events");
    }
}
```

(B)

```
public event DisplayMessage Print;

class EventProgram

{
    void Show(string msg)
    {
        Console.WriteLine(msg);
    }

    static void Main(string[] args)
    {
        EventProgram objEvent = new EventProgram();

        objEvent.Print += new DisplayMessage(objEvent.Show);

        objEvent.Print("This is a demonstration of events");
    }
}
```

(C)	<pre>public delegate void DisplayMessage(string msg); class EventProgram { event DisplayMessage Print; void Show(string msg) { Console.WriteLine(msg); } static void Main(string[] args) { EventProgram objEvent = new EventProgram(); objEvent.Print = new Print(objEvent.Show); objEvent.Print("This is a demonstration of events"); } }</pre>
(D)	<pre>public delegate void DisplayMessage(string msg); class EventProgram { event DisplayMessage Print(); void Show(string msg) { Console.WriteLine(msg); } static void Main(string[] args) { EventProgram objEvent = new EventProgram(); objEvent.Print += new DisplayMessage(objEvent.Show); objEvent.Print("This is a demonstration of events"); } }</pre>

(A)	A	(C)	C
(B)	B	(D)	D

5. You are trying to display output as Hard Disk, Memory, and MotherBoard. Which of the following codes will help you to achieve this?

(A)

```
class ArrayCollection
{
    static void Main(string[] args)
    {
        ArrayList objArray = new ArrayList();
        objArray.Add("HardDisk");
        objArray.Add("Memory");
        objArray.Add("Processor");
        objArray.RemoveAt(2);
        objArray.Insert(2, "MotherBoard");
        Console.WriteLine("Elements of the ArrayList");
        foreach (string str in objArray)
        {
            Console.WriteLine(str);
        }
    }
}
```

(B)

```
class ArrayCollection
{
    static void Main(string[] args)
    {
        ArrayList objArray = new ArrayList();
        objArray.AddItem("HardDisk");
        objArray.AddItem("Memory");
```

(B)	<pre> objArray.AddItem("Processor"); objArray.RemoveAt(2); objArray.Insert(2, "MotherBoard"); Console.WriteLine("Elements of the ArrayList"); foreach (string str in objArray) { Console.WriteLine(str); } } </pre>
(C)	<pre> class ArrayCollection { static void Main(string[] args) { ArrayList objArray = new ArrayList(); objArray.Add(new string("Hard Disk")); objArray.Add(new string("Memory")); objArray.Add(new string("Processor")); objArray.RemoveAt(2); objArray.Insert(2, "MotherBoard"); Console.WriteLine("Elements of the ArrayList"); foreach (string str in objArray) { Console.WriteLine(str); } } } </pre>
(D)	<pre> class ArrayCollection { static void Main(string[] args) { } </pre>

```

ArrayList objArray = new ArrayList();
objArray.Add("Hard Disk");
objArray.Add("Memory");
objArray.Add("Processor");
objArray.Remove(2);
objArray.InsertAt(2, "MotherBoard");
Console.WriteLine("Elements of the ArrayList");
foreach (string str in objArray)
{
    Console.WriteLine(str);
}
}
}
}

```

(A)	A	(C)	C
(B)	B	(D)	D

6. Which of these statements about the `SortedList` class are true?

(A)	The <code>SortedList</code> class represents a collection of key and value pairs arranged according to the key.
(B)	The <code>SortedList</code> class allows access to elements either by using the value or the index position.
(C)	The <code>SortedList</code> class behaves like the hashtable when you access elements based on their index position.
(D)	The <code>IndexOfKey()</code> method returns the index position of the specified key.
(E)	The <code>GetByIndex()</code> method returns the key at the specified index position.

(A)	A	(C)	C
(B)	B, C, D	(D)	A, D

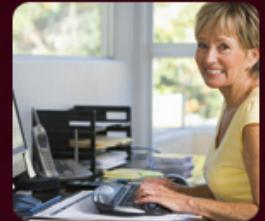
12.4.1 Answers

1.	A
2.	C
3.	C
4.	A
5.	A
6.	D



Summary

- ➔ A delegate in C# is used to refer to a method in a safe manner.
- ➔ An event is a data member that enables an object to provide notifications to other objects about a particular action.
- ➔ The System.Collections.Generic namespace consists of generic collections that allow reusability of code and provide better type-safety.
- ➔ The ArrayList class allows you to increase or decrease the size of the collection during program execution.
- ➔ The Hashtable class stores elements as key and value pairs where the data is organized based on the hash code. Each value in the hashtable is uniquely identified by its key.
- ➔ The SortedList class allows you to store elements as key and value pairs where the data is sorted based on the key.
- ➔ The Dictionary generic class represents a collection of elements organized in key and value pairs.



Session -13

Generics and Iterators

Welcome to the Session, **Generics and Iterators**.

Generics are data structures that allow you to reuse the same code for different types such as classes, interfaces, and so forth. Generics can be most useful while working with arrays and enumerator collections. Iterators are blocks of code that can iterate through the values of the collection.

In this session, you will learn to:

- ➔ Define and describe generics
- ➔ Explain creating and using generics
- ➔ Explain iterators

13.1 Generics

Generics are a kind of parameterized data structures that can work with value types as well as reference types. You can define a class, interface, structure, method, or a delegate as a generic type in C#.

Consider a C# program that uses an array variable of type `Object` to store a collection of student names. The names are read from the console as value types and are boxed to enable storing each of them as type `Object`. In this case, the compiler cannot verify the data stored against its data type as it allows you to cast any value to and from `Object`. If you enter numeric data, it will be accepted without any verification.

To ensure type-safety, C# introduces generics, which has a number of features including the ability to allow you to define generalized type templates based on which the type can be constructed later.

13.1.1 Namespaces, Classes, and Interfaces for Generics

There are several namespaces in the .NET Framework that facilitate creation and use of generics.

The `System.Collections.ObjectModel` namespace allows you to create dynamic and read-only generic collections. The `System.Collections.Generic` namespace consists of classes and interfaces that allow you to define customized generic collections.

→ Classes

The `System.Collections.Generic` namespace consists of classes that allow you to create type-safe collections. Table 13.1 lists some of the widely used classes of the `System.Collections.Generic` namespace.

Class	Descriptions
Comparer	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the <code>IComparer</code> interface
Dictionary.KeyCollection	Consists of keys present in the instance of the <code>Dictionary</code> class
Dictionary.ValueCollection	Consists of values present in the instance of the <code>Dictionary</code> class
EqualityComparer	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the <code>IEqualityComparer</code> interface

Table 13.1: Classes of `System.Collections.Generic` Namespace

→ Interfaces

The `System.Collections.Generic` namespace consists of interfaces that allow you to create type-safe collections.

Table 13.2 lists some of the widely used interfaces of the `System.Collections.Generic` namespace.

Interface	Descriptions
<code>IComparer</code>	Defines a generic method <code>Compare()</code> that compares values within a collection
<code>IEnumerable</code>	Defines a generic method <code>GetEnumerator()</code> that iterates over a collection
<code>IEqualityComparer</code>	Consists of methods which check for the equality between two objects

Table 13.2: Interfaces of `System.Collections.Generic` Namespace

13.1.2 `System.Collections.ObjectModel`

The `System.Collections.ObjectModel` namespace consists of classes that can be used to create customized generic collections.

Table 13.3 shows the classes contained in the `System.Collections.ObjectModel` namespace.

Classes	Descriptions
<code>Collection<></code>	Provides the base class for generic collections
<code>KeyedCollection<></code>	Provides an abstract class for a collection whose keys are associated with values
<code>ReadOnlyCollection<></code>	Is a read-only generic base class that prevents modification of collection

Table 13.3: Classes of `System.Collections.ObjectModel` namespace

Code Snippet 1 demonstrates the use of the `ReadOnlyCollection<>` class.

Code Snippet 1:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
class ReadOnly
{
    static void Main(string[] args)
    {
        List<string> objList = new List<string>();
        objList.Add("Francis");
        objList.Add("James");
```

```
objList.Add("Baptista");
objList.Add("Micheal");
ReadOnlyCollection<string> objReadOnly = new ReadOnlyCollection
    <string>(objList);
Console.WriteLine("Values stored in the readonly collection");
foreach (string str in objReadOnly)
{
    Console.WriteLine(str);
}
Console.WriteLine();
Console.WriteLine("Total number of elements in the readonly collection:
    " + objReadOnly.Count);
if (objList.Contains("Francis"))
{
    objList.Insert(2, "Peter");
}
Console.WriteLine("\nValues stored in the list after modification");
foreach (string str in objReadOnly)
{
    Console.WriteLine(str);
}
string[] array = new string[objReadOnly.Count * 2];
objReadOnly.CopyTo(array, 5);
Console.WriteLine("\nTotal number of values that can be stored in the new
    array: " + array.Length);
Console.WriteLine("Values in the new array");
foreach (string str in array)
{
    if (str == null)
    {
        Console.WriteLine("null");
```

```
        }

    else

    {

        Console.WriteLine(str);

    }

}

}
```

In Code Snippet 1, the Main() method of the **ReadOnly** class creates an instance of the List class. The Add() method inserts elements in the instance of the List class. An instance of the **ReadOnlyCollection** class of type **string** is created and the elements stored in the instance of the List class are copied to the instance of the **ReadOnlyCollection** class. The Contains() method checks whether the List class contains the specified element. If the List class contains the specified element, **Francis**, then the new element, **Peter**, is inserted at the specified index position, 2. The code creates an array variable that is twice the size of the **ReadOnlyCollection** class. The CopyTo() method copies the elements from the **ReadOnlyCollection** class to the array variable from the fifth position onwards.

Figure 13.1 displays the output of Code Snippet 1.

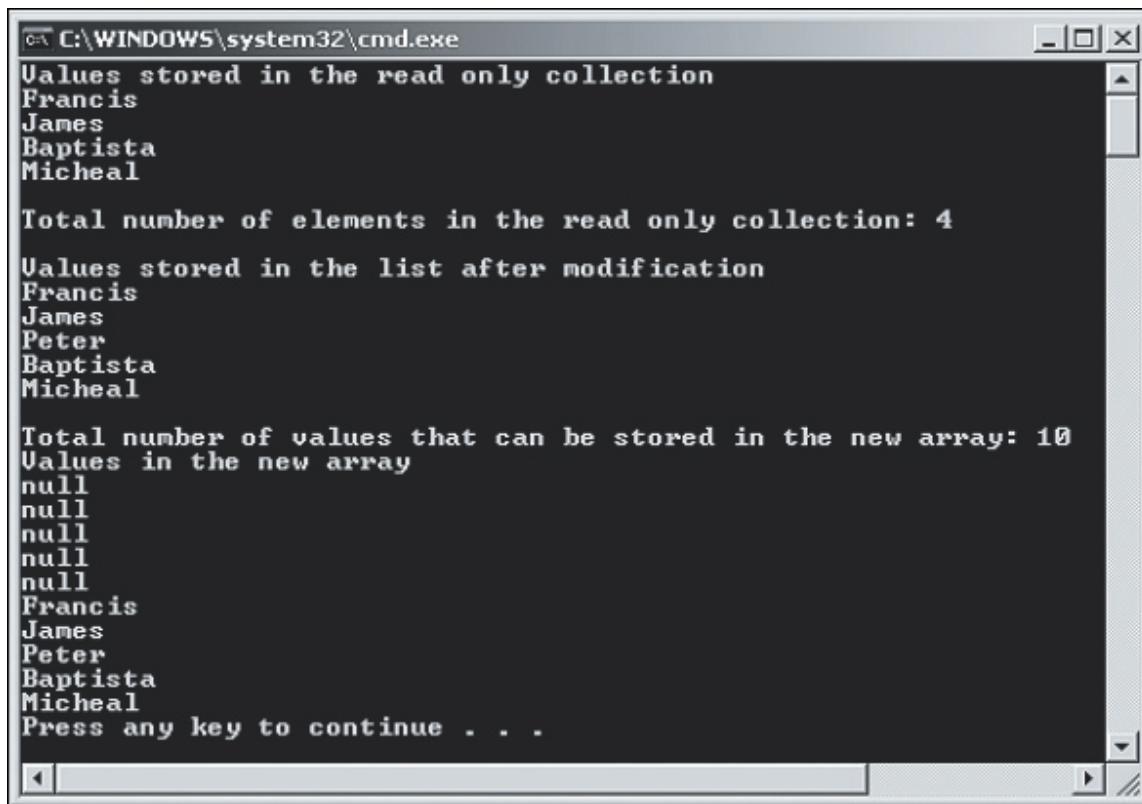


Figure 13.1: Output of Code Snippet 1

13.1.3 Creating Generic Types

A generic declaration always accepts a **type parameter**, which is a placeholder for the required data type. The type is specified only when a generic type is referred to or constructed as a type within a program.

The process of creating a generic type begins with a generic type definition containing type parameters. The definition acts like a blueprint. Later, a generic type is constructed from the definition by specifying actual types as the generic type arguments, which will substitute for the type parameters or the placeholders.

13.1.4 Benefits

Generics ensure type-safety at compile-time. Generics allow you to reuse the code in a safe manner without casting or boxing. A generic type definition is reusable with different types but can accept values of a single type at a time. Apart from reusability, there are several other benefits of using generics. These are as follows:

- Improved performance because of low memory usage as no casting or boxing operation is required to create a generic
- Ensured strongly-typed programming model
- Reduced run-time errors that may occur due to casting or boxing

13.2 Creating and Using Generics

Generic types are not confined to classes alone but can include interfaces and delegates. This section examines generic classes, methods, interfaces, delegates, and so on.

13.2.1 Generic Classes

Generic classes define functionalities that can be used for any data type. Generic classes are declared with a class declaration followed by a **type parameter** enclosed within angular brackets. While declaring a generic class, you can apply some restrictions or constraints to the type parameters by using the `where` keyword. However, applying constraints to the type parameters is optional.

Thus, while creating a generic class, you must generalize the data types into the type parameter and optionally decide the constraints to be applied on the type parameter.

The following syntax is used for creating a generic class.

Syntax:

```
<access_modifier> class <ClassName> <<type parameter list>> [where <type parameter constraint clause>]
```

where,

`access_modifier`: Specifies the scope of the generic class. It is optional.

ClassName: Is the name of the new generic class to be created.

<type parameter list>: Is used as a placeholder for the actual data type.

type parameter constraint clause: Is an optional class or an interface applied to the type parameter with the where keyword.

Code Snippet 2 creates a generic class that can be used for any specified data type.

Code Snippet 2:

```
using System;
using System.Collections.Generic;
class General<T>
{
    T[] values;
    int _counter = 0;
    public General(int max)
    {
        values = new T[max];
    }
    public void Add(T val)
    {
        if (_counter < values.Length)
        {
            values[_counter] = val;
            _counter++;
        }
    }
    public void Display()
    {
        Console.WriteLine("Constructed Class is of type: " + typeof(T));
        Console.WriteLine("Values stored in the object of constructed class are:");
        for (int i = 0; i < values.Length; i++)
        {
            Console.WriteLine(values[i]);
        }
    }
}
```

```

    }
}

}

class Students
{
    static void Main(string[] args)
    {
        General<string> objGeneral = new General<string>(3);
        objGeneral.Add("John");
        objGeneral.Add("Patrick");
        objGeneral.Display();
        General<int> objGeneral2 = new General<int>(2);
        objGeneral2.Add(200);
        objGeneral2.Add(35);
        objGeneral2.Display();
    }
}

```

In Code Snippet 2, a generic class definition for **General** is created that takes a type parameter **T**. The generic class declares a parameterized constructor with an **int** value. The **Add()** method takes a parameter of the same type as the generic class. The method **Display()** displays the value type specified by the type parameter and the values supplied by the user through the object. The **Main()** method of the class **Students** creates an instance of the generic class **General** by providing the type parameter value as **string** and total values to be stored as 3. This instance invokes the **Add()** method which takes student names as values. These student names are displayed by invoking the **Display()** method. Later, another object is created of a different data type, **int**, based on the same class definition. The class definition is generic, we need not change the code now, but can reuse the same code for an **int** data type as well. Thus, using the same generic class definition, we can create two different lists of data.

Output:

```

Constructed Class is of type: System.String
Values stored in the object of constructed class are:
John
Patrick
Constructed Class is of type: System.Int32
Values stored in the object of constructed class are:

```

200

35

Note - Generic classes can be nested within other generic or non-generic classes. However, any class nested within a generic class is itself a generic class since the type parameters of the outer class are supplied to the nested class.

13.2.2 Constraints on Type Parameters

You can apply constraints on the type parameter while declaring a generic type. A constraint is a restriction imposed on the data type of the type parameter. Constraints are specified using the `where` keyword. They are used when the programmer wants to limit the data types of the type parameter to ensure consistency and reliability of data in a collection.

Table 13.4 lists the types of constraints that can be applied to the type parameter.

Constraints	Descriptions
<code>T : struct</code>	Specifies that the type parameter must be of a value type only except the null value
<code>T : class</code>	Specifies that the type parameter must be of a reference type such as a class, interface, or a delegate
<code>T : new()</code>	Specifies that the type parameter must consist of a constructor without any parameter which can be invoked publicly
<code>T : <base class name></code>	Specifies that the type parameter must be the parent class or should inherit from a parent class
<code>T : <interface name></code>	Specifies that the type parameter must be an interface or should inherit an interface

Table 13.4: Types of Constraints

Code Snippet 3 creates a generic class that uses a class constraint.

Code Snippet 3:

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

class Employee
{
    string _empName;
    int _empID;
    public Employee(string name, int num)
    {
```

```
_empName = name;
    _empID = num;
}
public string Name
{
    get
    {
        return _empName;
    }
}
public int ID
{
    get
    {
        return _empID;
    }
}
}

class GenericList<T> where T : Employee
{
    T[] _name = new T[3];
    int _counter = 0;
    public void Add(T val)
    {
        _name[_counter] = val;
        _counter++;
    }
    public void Display()
    {
        for (int i = 0; i < _counter; i++)
        {
```

```

        Console.WriteLine(_name[i].Name + ", " + _name[i].ID);
    }
}
}

class ClassConstraintDemo
{
    static void Main(string[] args)
    {
        GenericList<Employee> objList = new
            GenericList<Employee>();
        objList.Add(new Employee("John", 100));
        objList.Add(new Employee("James", 200));
        objList.Add(new Employee("Patrich", 300));
        objList.Display();
    }
}

```

In Code Snippet 3, the class **GenericList** is created that takes a type parameter **T**. This type parameter is applied a class constraint, which means the type parameter can only include details of the **Employee** type. The generic class creates an array variable with the type parameter **T**, which means it can include values of type **Employee**. The **Add()** method consists of a parameter **val**, which will contain the values set in the **Main()** method. Since, the type parameter should be of the **Employee** type, the constructor is called while setting the values in the **Main()** method.

Output:

John, 100
James, 200
Patrich, 300

Note - When you use a certain type as a constraint, the type used as a constraint must have greater scope of accessibility than the generic type that will be using the constraint.

13.2.3 Inheriting Generic Classes

A generic class can be inherited like any other non-generic class in C#. Thus, a generic class can act both as a base class or a derived class.

While inheriting a generic class in another generic class, you can use the generic type parameter of the base class instead of passing the data type of the parameter. However, while inheriting a generic class in

a non-generic class, you must provide the data type of the parameter instead of the base class generic type parameter. The constraints imposed at the base class level must be included in the derived generic class.

Figure 13.2 displays a generic class as a base class.

```

Generic -> Generic

public class Student<T>
{
}
public class Marks<T>: Student<T>
{
}

Generic -> Non-Generic

public class Student<T>
{
}
public class Marks: Student<int>
{
}

```

Figure 13.2: Generic Class as Base Class

The following syntax is used to inherit a generic class from an existing generic class.

Syntax:

```
<access_modifier> class <BaseClass><<generic type parameter>>{ }
<access_modifier> class <DerivedClass> : <BaseClass><<generic type parameter>>{ }
```

where,

access_modifier: Specifies the scope of the generic class.

BaseClass: Is the generic base class.

<generic type parameter>: Is a placeholder for the specified data type.

DerivedClass: Is the generic derived class.

The following syntax is used to inherit a non-generic class from a generic class.

Syntax:

```
<access_modifier> class <BaseClass><<generic type parameter>>{ }
<access_modifier> class <DerivedClass> : <BaseClass><<type parameter value>>{ }
```

where,

<type parameter value>: Can be a data type such as int, string, or float.

13.2.4 Generic Methods

Generic methods process values whose data types are known only when accessing the variables that store these values. A generic method is declared with the generic type parameter list enclosed within angular brackets. Defining methods with type parameters allow you to call the method with a different type every time. You can declare a generic method within generic or non-generic class declarations. When you declare a generic method within a generic class declaration, the body of the method refers to the type parameters of both, the method and class declaration.

Generic methods can be declared with the following keywords:

→ **`virtual`**

The generic methods declared with the `virtual` keyword can be overridden in the derived class.

→ **`override`**

The generic method declared with the `override` keyword overrides the base class method. However, while overriding, the method does not specify the type parameter constraints since the constraints are overridden from the overridden method.

→ **`abstract`**

The generic method declared with the `abstract` keyword contains only the declaration of the method. Such methods are typically implemented in a derived class.

The following syntax is used for declaring a generic method.

Syntax:

`<access_modifier> <return_type> <MethodName> <<type parameter list>>`

where,

`access_modifier`: Specifies the scope of the method.

`return_type`: Determines the type of value the generic method will return.

`MethodName`: Is the name of the generic method.

`<type parameter list>`: Is used as a placeholder for the actual data type.

Code Snippet 4 creates a generic method within a non-generic class.

Code Snippet 4:

```
using System;
using System.Collections.Generic;
class SwapNumbers
{
```

```

static void Swap<T>(ref T valOne, ref T valTwo)
{
    T temp = valOne;
    valOne = valTwo;
    valTwo = temp;
}

static void Main(string[] args)
{
    int numOne = 23;
    int numTwo = 45;

    Console.WriteLine("Values before swapping: " + numOne + " & " + numTwo);

    Swap<int>(ref numOne, ref numTwo);

    Console.WriteLine("Values after swapping: " + numOne + " & " + numTwo);
}
}

```

In Code Snippet 4, the class **SwapNumbers** consists of a generic method **Swap()** that takes a type parameter **T** within angular brackets and two parameters within parenthesis of type **T**. The **Swap()** method creates a variable **temp** of type **T** that is assigned the value within the variable **valOne**. The **Main()** method displays the values initialized within variables and calls the **Swap()** method by providing the type **int** within angular brackets. This will substitute for the type parameter in the generic method definition and will display the swapped values within the variables.

Output:

Values before swapping: 23 & 45

Values after swapping: 45 & 23

13.2.5 Generic Interfaces

Generic interfaces are useful for generic collections or generic classes representing the items in the collection. You can use the generic classes with the generic interfaces to avoid boxing and unboxing operations on the value types.

Generic classes can implement the generic interfaces by passing the required parameters specified in the interface. Similar to generic classes, generic interfaces also implement inheritance.

The syntax for declaring an interface is similar to the syntax for class declaration.

The following syntax is used for creating a generic interface.

Syntax:

```
<access_modifier> interface <InterfaceName> <<type parameter list>> [where
<type parameter constraint clause>]
```

where,

access_modifier: Specifies the scope of the generic interface.

InterfaceName: Is the name of the new generic interface.

<type parameter list>: Is used as a placeholder for the actual data type.

type parameter constraint clause: Is an optional class or an interface applied to the type parameter with the **where** keyword.

Code Snippet 5 creates a generic interface that is implemented by the non-generic class.

Code Snippet 5:

```
using System;
using System.Collections.Generic;
interface IMaths<T>
{
    T Addition(T valOne, T valTwo);
    T Subtraction(T valOne, T valTwo);
}
class Numbers : IMaths<int>
{
    public int Addition(int valOne, int valTwo)
    {
        return valOne + valTwo;
    }
    public int Subtraction(int valOne, int valTwo)
    {
        if (valOne > valTwo)
        {
            return (valOne - valTwo);
        }
    }
}
```

```

else
{
    return (valTwo - valOne);
}
}

static void Main(string[] args)
{
    int numOne = 23;
    int numTwo = 45;
    Numbers objInterface = new Numbers();
    Console.Write("Addition of two integer values is: ");
    Console.WriteLine(objInterface.Addition(numOne, numTwo));
    Console.Write("Subtraction of two integer values is: ");
    Console.WriteLine(objInterface.Subtraction(numOne, numTwo));
}
}

```

In Code Snippet 5, the generic interface **IMaths** takes a type parameter **T** and declares two methods of type **T**. The class **Numbers** implements the interface **IMaths** by providing the type **int** within angular brackets and implements the two methods declared in the generic interface. The **Main()** method creates an instance of the class **Numbers** and displays the addition and subtraction of two numbers.

Output:

```

Addition of two integer values is: 68
Subtraction of two integer values is: 22

```

13.2.6 Generic Interface Constraints

You can specify an interface as a constraint on a type parameter. This enables you to use the members of the interface within the generic class. In addition, it ensures that only the types that implement the interface are used.

You can also specify multiple interfaces as constraints on a single type parameter. Code Snippet 6 creates a generic interface that is used as a constraint on a generic class.

Code Snippet 6:

```

using System;
using System.Collections.Generic;

```

```
interface IDetails
{
    void GetDetails();
}

class Student : IDetails
{
    string _studName;
    int _studID;
    public Student(string name, int num)
    {
        _studName = name;
        _studID = num;
    }
    public void GetDetails()
    {
        Console.WriteLine(_studID + "\t" + _studName);
    }
}

class GenericList<T> where T : IDetails
{
    T[] _values = new T[3];
    int _counter = 0;
    public void Add(T val)
    {
        _values[_counter] = val;
        _counter++;
    }
    public void Display()
    {
        for (int i = 0; i < 3; i++)
        {
    }
```

```

        _values[i].GetDetails();

    }

}

}

class InterfaceConstraintDemo
{
    static void Main(string[] args)
    {
        GenericList<Student> objList = new GenericList<Student>();
        objList.Add(new Student("Wilson", 120));
        objList.Add(new Student("Jack", 130));
        objList.Add(new Student("Peter", 140));
        objList.Display();
    }
}

```

In Code Snippet 6, an interface **IDetails** declares a method **GetDetails()**. The class **Student** implements the interface **IDetails**. The class **GenericList** is created that takes a type parameter **T**. This type parameter is applied an interface constraint, which means the type parameter can only include details of the **IDetails** type. The **Main()** method creates an instance of the class **GenericList** by passing the type parameter value as **Student**, since the class **Student** implements the interface **IDetails**.

Figure 13.3 creates a generic interface.

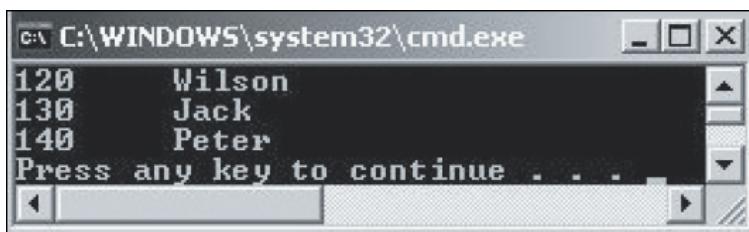


Figure 13.3: Generic Interface

13.2.7 Generic Delegates

Delegates are reference types that encapsulate a reference to a method that has a signature and a return type. Similar to classes, interfaces, and structures, user-defined methods and delegates can also be declared as generic. A generic delegate can be used to refer to multiple methods in a class with different types of parameters. However, the number of parameters of the delegate and the referenced methods must be the same. The syntax for declaring a generic delegate is similar to that of declaring a generic

method, where the type parameter list is specified after the delegate's name.

The following syntax is used to declare a generic delegate.

Syntax:

```
delegate <return_type> <DelegateName><type parameter list>(<argument_list>);
```

where,

return_type: Determines the type of value the delegate will return.

DelegateName: Is the name of the generic delegate.

type parameter list: Is used as a placeholder for the actual data type.

argument_list: Specifies the parameter within the delegate.

Code Snippet 7 declares a generic delegate.

Code Snippet 7:

```
using System;

delegate T DelMath<T>(T val);

class Numbers
{
    static int NumberType(int num)
    {
        if (num % 2 == 0)
            return num;
        else
            return (0);
    }

    static float NumberType(float num)
    {
        return num % 2.5F;
    }

    public static void Main(string[] args)
    {
        DelMath<int> objDel = NumberType;
        DelMath<float> objDel2 = NumberType;
        Console.WriteLine(objDel(10));
    }
}
```

```

        Console.WriteLine(objDel2(108.756F));
    }
}

```

In Code Snippet 7, a generic delegate is declared in the **Numbers** class. In the **Main()** method of the class, an object of the delegate is created, which is referring to the **NumberType()** method and takes the parameter of **int** type. An integer value is passed to the method, which displays the value only if it is an even number. Another object of the delegate is created in the **Main()** method, which is referring to the **NumberType()** method and takes the parameter of **float** type. A float value is passed to the method, which displays the remainder of the division operation. Therefore, generic delegates can be used for overloaded methods.

Figure 13.4 declares a generic delegate.

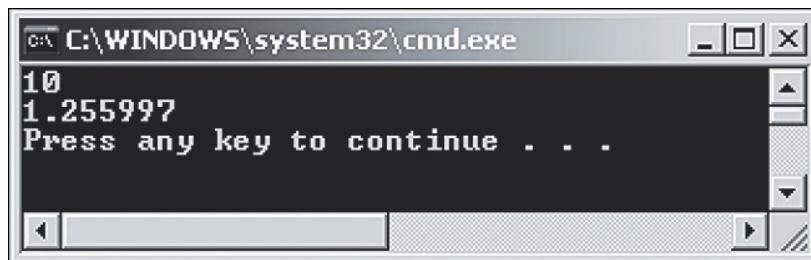


Figure 13.4: Generic Delegate

13.2.8 Overloading Methods Using Type Parameters

Methods of a generic class that take generic type parameters can be overloaded. The programmer can overload the methods that use type parameters by changing the type or the number of parameters. However, the type difference is not based on the generic type parameter, but is based on the data type of the parameter passed.

Code Snippet 8 demonstrates how to overload methods that use type parameters.

Code Snippet 8:

```

using System;
using System.Collections;
using System.Collections.Generic;
class General<T, U>
{
    T _valOne;
    U _valTwo;
    public void AcceptValues(T item)
    {

```

```

        _valOne=item;
    }

    public void AcceptValues (U item)
    {
        _valTwo=item;
    }

    public void Display()
    {
        Console.Write(_valOne + "\t" + _valTwo);
    }
}

class MethodOverload
{
    static void Main (string [] args)
    {
        General<int, string> objGenOne = new General<int, string>();
        objGenOne.AcceptValues (10);
        objGenOne.AcceptValues ("Smith");
        Console.WriteLine ("ID\tName\tDesignation\tSalary");
        objGenOne.Display();

        General<string, float> objGenTwo = new General<string, float>();
        objGenTwo.AcceptValues ("Mechanic");
        objGenTwo.AcceptValues (2500);
        Console.Write ("\t");
        objGenTwo.Display();
        Console.WriteLine ();
    }
}

```

In Code Snippet 8, the **General** class has two overloaded methods with different type parameters. In the **Main()** method, the instance of the **General** class is created. The class is initialized by specifying the data type for the generic parameters **T** and **U** as **string** and **int** respectively. The overloaded methods are invoked by specifying appropriate values. The methods store these values in the respective variables defined in the **General** class. These values indicate the ID and name of the employee.

Another instance of the **General** class is created specifying the type of data the class can contain as **string** and **float**. The overloaded methods are invoked by specifying appropriate values. The methods store these values in the respective variables defined in the **General** class. These values indicate the designation and salary of the employee.

Figure 13.5 displays the output of using overload methods with type parameters.

ID	Name	Designation	Salary
10	Smith	Mechanic	2500

Figure 13.5: Overloaded Methods Using Type Parameters

13.2.9 Overriding Virtual Methods in Generic Class

Methods in generic classes can be overridden like the method in any non-generic class. To override a method in the generic class, the method in the base class must be declared as **virtual** and this method can be overridden in the derived class, using the **override** keyword.

Code Snippet 9 demonstrates how to override virtual methods of a generic class.

Code Snippet 9:

```
using System;
using System.Collections;
using System.Collections.Generic;
class GeneralList<T>
{
    protected T ItemOne;
    public GeneralList(T valOne)
    {
        ItemOne = valOne;
    }
    public virtual T GetValue()
    {
        return ItemOne;
    }
}
```

```

class Student<T> : GeneralList<T>
{
    public TValue;
    public Student (TvalOne, TvalTwo) : base (valOne)
    {
        Value = valTwo;
    }
    public override T GetValue ()
    {
        Console.Write (base.GetValue () + "\t\t");
        return Value;
    }
}
class StudentList
{
    public static void Main ()
    {
        Student<string> objStudent = new Student<string> ("Patrick", "Male");
        Console.WriteLine ("Name\tSex");
        Console.WriteLine (objStudent.GetValue ());
    }
}

```

In Code Snippet 9, the **GeneralList** class consists of a constructor that assigns the name of the student. The **GetValue()** method of the **GeneralList** class is overridden in the **Student** class. The constructor of the **Student** class invokes the base class constructor by using the **base** keyword and assigns the gender of the specified student. The **GetValue()** method of the derived class returns the sex of the student. The name of the student is retrieved by using the **base** keyword to call the **GetValue()** method of the base class. The **StudentList** class creates an instance of the **Student** class. This instance invokes the **GetValue()** method of the derived class, which in turn invokes the **GetValue()** method of the base class by using the **base** keyword.

Figure 13.6 displays the output of using overridden virtual methods for generic class.

Figure 13.6: Overridden Virtual Methods of Generic Class

13.3 Iterators

Consider a scenario where a person is trying to memorize a book of 100 pages. To finish the task, the person has to iterate through each of these 100 pages.

Similar to this person who iterates through the pages, an iterator in C# is used to traverse through a list of values or a collection. It is a block of code that uses the `foreach` loop to refer to a collection of values in a sequential manner. For example, consider a collection of values that needs to be sorted. To implement the logic manually, a programmer can iterate through each value sequentially using iterators to compare the values.

Figure 13.7 illustrates these analogies.

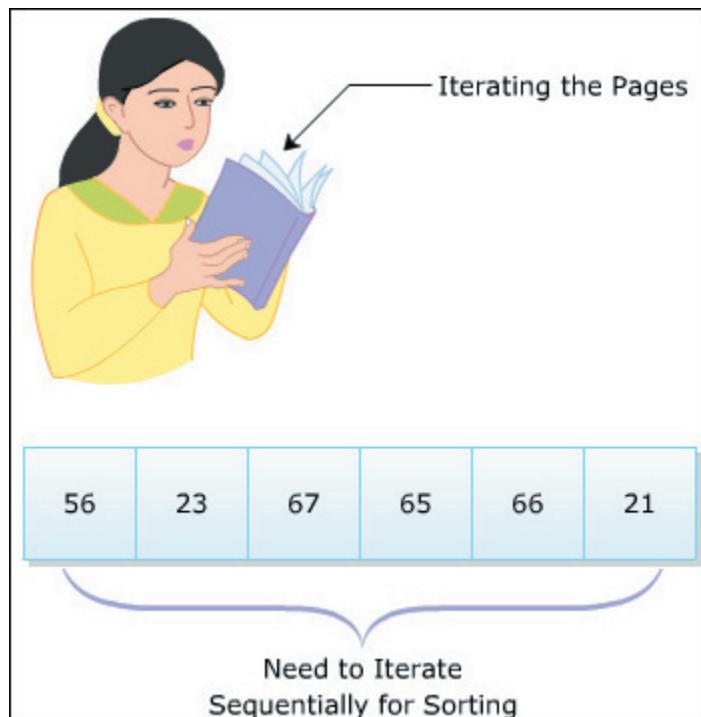


Figure 13.7: Example of Iteration

An iterator is not a data member but is a way of accessing the member. It can be a method, a get accessor, or an operator that allows you to navigate through the values in a collection. Iterators specify the way values are generated, when the `foreach` statement accesses the elements within a collection.

They keep track of the elements in the collection, so that you can retrieve these values if required.

For example, consider an array variable consisting of six elements, where the iterator can return all the elements within an array one by one.

13.3.1 Benefits

For a class that behaves like a collection, it is preferable to use iterators to iterate through the values of the collection with the `foreach` statement. By doing this, one can get the following benefits:

- Iterators provide a simplified and faster way of iterating through the values of a collection.
- Iterators reduce the complexity of providing an enumerator for a collection.
- Iterators can return large number of values.
- Iterators can be used to evaluate and return only those values that are needed.
- Iterators can return values without consuming memory by referring each value in the list.

13.3.2 Implementation

Iterators can be created by implementing the `GetEnumerator()` method that returns a reference of the `IEnumerator` interface.

The iterator block uses the `yield` keyword to provide values to the instance of the enumerator or to terminate the iteration. The `yield return` statement returns the values, while the `yield break` statement ends the iteration process. When the program control reaches the `yield return` statement, the current location is stored, and the next time the iterator is called, the execution is started from the stored location.

Code Snippet 10 demonstrates the use of iterators to iterate through the values of a collection.

Code Snippet 10:

```
using System;
using System.Collections;
class Department : IEnumerable
{
    string[] departmentNames = {"Marketing", "Finance", "Information
        Technology", "Human Resources"};
    public IEnumerator GetEnumerator()
    {
```

```

for (int i=0; i<departmentNames.Length; i++)
{
    yield return departmentNames [i];
}

}

static void Main (string [] args)
{
    Department objDepartment = new Department ();
    Console.WriteLine ("Department Names");
    Console.WriteLine ();
    foreach (string str in objDepartment)
    {
        Console.WriteLine (str);
    }
}
}

```

In Code Snippet 10, the class **Department** implements the interface **IEnumerable**. The class **Department** consists of an array variable that stores the department names and a **GetEnumerator()** method, that contains the **for** loop. The **for** loop returns the department names at each index position within the array variable. This block of code within the **GetEnumerator()** method comprises the iterator in this example. The **Main()** method creates an instance of the class **Department** and contains a **foreach** loop that displays the department names.

Figure 13.8 displays the use of iterators.

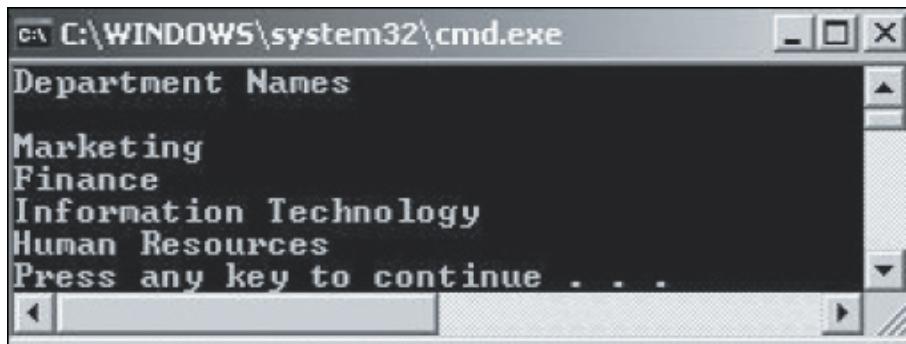


Figure 13.8: Use of Iterators

Note - When the C# compiler comes across an iterator, it invokes the Current, MoveNext, and Dispose methods of the `IEnumerable` interface by default. These methods are used to traverse through the data within the collection.

13.3.3 Generic Iterators

C# allows programmers to create generic iterators. Generic iterators are created by returning an object of the generic `IEnumerator<T>` or `IEnumerable<T>` interface. They are used to iterate through values of any value type.

Code Snippet 11 demonstrates how to create a generic iterator to iterate through values of any type.

Code Snippet 11:

```
using System;
using System.Collections.Generic;
class GenericDepartment<T>
{
    T[] item;
    public GenericDepartment(T[] val)
    {
        item=val;
    }
    public IEnumerator<T> GetEnumerator()
    {
        foreach (T value in item)
        {
            yield return value;
        }
    }
}
class GenericIterator
{
    static void Main(string[] args)
    {
        string[] departmentNames = { "Marketing", "Finance",
```

```

        "Information Technology", "Human Resources" };

GenericDepartment<string> objGeneralName = new GenericDepartment<string>(
    departmentNames);

foreach (string val in objGeneralName)
{
    Console.WriteLine(val + "\t");
}

int[] departmentID = { 101, 110, 210, 220 };

GenericDepartment<int> objGeneralID = new GenericDepartment<int>
(departmentID);

Console.WriteLine();
foreach (int val in objGeneralID)
{
    Console.WriteLine(val + "\t\t");
}
Console.WriteLine();
}
}
}

```

Figure 13.9 creates a generic iterator.

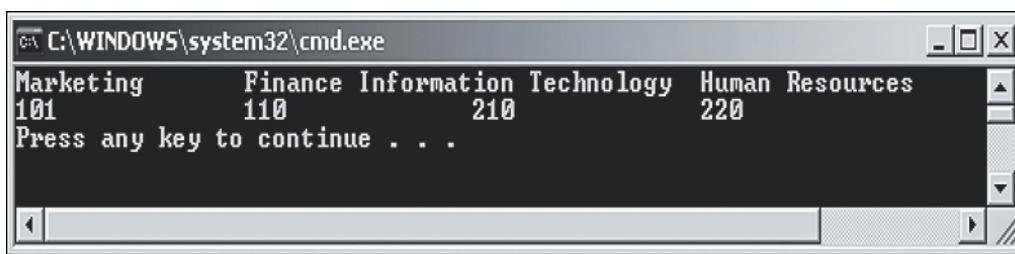


Figure 13.9: Generic Iterator

In Code Snippet 11, the generic class, **GenericDepartment**, is created with the generic type parameter **T**. The class declares an array variable and consists of a parameterized constructor that assigns values to this array variable. In the generic class, **GenericDepartment**, the **GetEnumerator()** method returns a generic type of the **IEnumerator** interface. This method returns elements stored in the array variable, using the **yield** statement. In the **GenericIterator** class, an instance of the **GenericDepartment** class is created that refers to the different department names within the array. Another object of the **GenericDepartment** class is created, that refers to the different department IDs within the array.

13.3.4 Implementing Named Iterators

Another way of creating iterators is by creating a method, whose return type is the `IEnumerable` interface. This is called a **named iterator**. Named iterators can accept parameters that can be used to manage the starting and end points of a `foreach` loop. This flexible technique allows you to fetch the required values from the collection.

The following syntax creates a named iterator.

Syntax:

```
<access_modifier> IEnumerable <IteratorName> (<parameter list>) {}
```

where,

`access_modifier`: Specifies the scope of the named iterator.

`IteratorName`: Is the name of the iterator method.

`parameter list`: Defines zero or more parameters to be passed to the iterator method.

Code Snippet 12 demonstrates how to create a named iterator.

Code Snippet 12:

```
using System;
class NamedIterators
{
    string[] cars = { "Ferrari", "Mercedes", "BMW", "Toyota", "Nissan" };
    public IEnumerable GetCarNames()
    {
        for (int i = 0; i < cars.Length; i++)
        {
            yield return cars[i];
        }
    }
    static void Main(string[] args)
    {
        NamedIterators objIterator = new NamedIterators();
        foreach (string str in objIterator.GetCarNames())
        {
            Console.WriteLine(str);
        }
    }
}
```

```
}
```

In Code Snippet 12, the **NamedIterators** class consists of an array variable and a method **GetCarNames()**, whose return type is **IEnumerable**. The **for** loop iterates through the values within the array variable. The **Main()** method creates an instance of the class **NamedIterators** and this instance is used in the **foreach** loop to display the names of the cars from the array variable.

Output:

Ferrari

Mercedes

BMW

Toyota

Nissan

13.4 Check Your Progress

1. Which of the following statements about generics are true?

(A)	Generics are used only for value types.		
(B)	Generics are used to work with multiple data types simultaneously.		
(C)	Generics can be implemented without a need for explicit or implicit casting.		
(D)	Generics are verified at run-time.		
(E)	Generics are declared with or without a type parameter.		

(A)	A	(C)	C
(B)	B, C, D	(D)	A, D

2. Match the classes and interfaces in the `System.Collections.Generic` namespace against their corresponding descriptions.

Description		Class/Interface	
(A)	Defines a method for iteration	(1)	<code>IComparer</code>
(B)	Allows using a method of the <code>IComparer</code> interface	(2)	<code>Dictionary.KeyCollection</code>
(C)	Defines methods to check whether the two objects are equal or not	(3)	<code>Comparer</code>
(D)	Contains keys present in the dictionary collection	(4)	<code>IEnumerable</code>
(E)	Allows comparison among objects of the collection	(5)	<code>IEqualityComparer</code>

(A)	A-3, B-4, C-5, D-2, E-1	(C)	A-5, B-3, C-4, D-2, E-1
(B)	A-4, B-3, C-5, D-2, E-1	(D)	A-1, B-2, C-3, D-5, E-2

3. Which of the following statements about generic classes, generic methods, and generic interfaces are true?

(A)	Non-generic classes cannot be inherited from generic classes.
(B)	Generic classes can be declared with a class declaration followed by a type parameter and optional constraints that are applied on the type parameter.
(C)	Generic methods can be declared only within the generic class declaration.
(D)	Generic interfaces can be used as constraints on a type parameter.

(E)

Generic methods can be declared with the `override` keyword that needs to explicitly specify the type parameter constraints.

(A)

B, D

(C)

C

(B)

B, C, D

(D)

A, D

4. You are trying to display the area and volume of a cube by implementing a generic interface. Which of the following code will help you to achieve this?

(A)

```

interface IArea<T>
{
    T Area(T valOne);
    T Volume(T valOne);
}

class Cube : IArea
{
    public double Area(double valOne)
    {
        return 6 * valOne * valOne;
    }

    public double Volume(double valOne)
    {
        return valOne * valOne * valOne;
    }

    static void Main(string[] args)
    {
        double side = 23.15;

        Cube objCube = new Cube();

        Console.Write("Area of cube: ");
        Console.WriteLine(objCube.Area(side));

        Console.Write("Volume of cube: ");
        Console.WriteLine(objCube.Volume(side));
    }
}

```

(B)

```
interface IArea(T)
{
    T Area(T valOne);
    T Volume(T valOne);
}

class Cube : IArea<double>
{
    public double Area(double valOne)
    {
        return 6 * valOne * valOne;
    }

    public double Volume(double valOne)
    {
        return valOne * valOne * valOne;
    }

    static void Main(string[] args)
    {
        double side = 23.15;
        Cube objCube = new Cube();
        Console.Write("Area of cube: ");
        Console.WriteLine(objCube.Area(side));
        Console.Write("Volume of cube: ");
        Console.WriteLine(objCube.Volume(side));
    }
}
```

(C)

```
interface IArea<T>
{
    T Area(T valOne);
    T Volume(T valOne);
}

class Cube : IArea<T>
{
    public double Area(double valOne)
    {
        return 6 * valOne * valOne;
    }

    public double Volume(double valOne)
    {
        return valOne * valOne * valOne;
    }

    static void Main(string[] args)
    {
        double side = 23.15;
        Cube objCube = new Cube();
        Console.Write("Area of cube: ");
        Console.WriteLine(objCube.Area(side));
        Console.Write("Volume of cube: ");
        Console.WriteLine(objCube.Volume(side));
    }
}
```

(D)

```

interface IArea<T>
{
    T Area (T valOne);
    T Volume (T valOne);
}

class Cube : IArea<double>
{
    public double Area (double valOne)
    {
        return 6 * valOne * valOne;
    }

    public double Volume (double valOne)
    {
        return valOne * valOne * valOne;
    }

    static void Main (string [] args)
    {
        double side = 23.15;
        Cube objCube = new Cube ();
        Console.Write ("Area of cube: ");
        Console.WriteLine (objCube.Area (side));
        Console.Write ("Volume of cube: ");
        Console.WriteLine (objCube.Volume (side));
    }
}

```

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of the following statements about iterators are true?

(A)	Iterators return sequentially ordered values of the same type.
(B)	Iterators return a fixed number of values that cannot be changed.
(C)	Iterators consume memory by storing the iterated values in the list.
(D)	One of the ways to create iterators is to implement the GetEnumerator() method of the IEnumerable interface.
(E)	Iterators use the yield break statement to end the iteration process.

(A)	A, D, E	(C)	C
(B)	B, C, D	(D)	A, D

6. You are trying to display the different programming languages as 'C', 'C++', 'Java', and 'C#'. Which of the following code will help you to achieve this?

(A)	<pre> using System; class Languages : IEnumerable { string[] _language = { "C", "C++", "Java", "C#" }; public IEnumerator GetEnumerator() { for (int i = 0; i < _language.Length; i++) { yield return _language[i]; } } static void Main(string[] args) { Languages objLanguage = new Languages(); Console.WriteLine("Programming languages"); Console.WriteLine(); foreach (string str in objLanguage) { </pre>
-----	--

(A)	<pre> Console.Write(str + "\t"); } Console.WriteLine(); } } </pre>
(B)	<pre> using System; class Languages : IEnumerable { string[] _language = { "C", "C++", "Java", "C#" }; public IEnumerator GetEnumerator() { for (int i = 0; i < _language.Length; i++) { yield return _language[i]; } } static void Main(string[] args) { Languages objLanguage = new Languages(); Console.WriteLine("Programming languages"); Console.WriteLine(); foreach (string str in objLanguage) { Console.Write(str + "\t"); } Console.WriteLine(); } } </pre>
(C)	<pre> using System; class Languages : IEnumerator { } </pre>

(C)

```

string[] _language = { "C", "C++", "Java", "C#" };
public IEnumerator GetEnumerator()
{
    for (int i = 0; i < _language.Length; i++)
    {
        yield return _language[i];
    }
}

static void Main(string[] args)
{
    Languages objLanguage = new Languages();
    Console.WriteLine("Programming languages");
    Console.WriteLine();
    foreach (string str in objLanguage)
    {
        Console.Write(str + "\t");
    }
    Console.WriteLine();
}
}

```

(D)

```

using System;
class Languages : IEnumerable
{
    string[] _language = { "C", "C++", "Java", "C#" };
    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < _language.Length; i++)
        {
            yield return _language[i];
        }
    }
}

```

(D)

```
static void Main(string[] args)
{
    Languages objLanguage = new Languages();
    Console.WriteLine("Programming languages");
    Console.WriteLine();
    foreach (string str in objLanguage)
    {
        Console.Write(str + "\t");
    }
    Console.WriteLine();
}
```

(A)	A	(C)	C
(B)	B, C, D	(D)	A, D

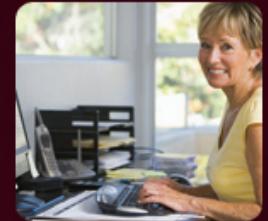
13.4.1 Answers

1.	C
2.	B
3.	A
4.	D
5.	A
6.	B



Summary

- Generics are data structures that allow you to reuse a code for different types such as classes or interfaces.
- Generics provide several benefits such as type-safety and better performance.
- Generic types can be declared by using the type parameter, which is a placeholder for a particular type.
- Generic classes can be created by the class declaration followed by the type parameter list enclosed in the angular brackets and application of constraints (optional) on the type parameters.
- An iterator is a block of code that returns sequentially ordered values of the same type.
- One of the ways to create iterators is by using the GetEnumerator() method of the IEnumerable or IEnumerator interface.
- The yield keyword provides values to the enumerator object or to signal the end of the iteration.



Session -14

Advanced Methods and Types

Welcome to the Session, **Advanced Methods and Types**.

C# supports several advanced methods and types such as anonymous methods, extension methods, partial methods, and so on. These methods and types greatly enhance the object oriented programming experience of C# developers.

In this session, you will learn to:

- Describe anonymous methods
- Define extension methods
- Explain anonymous types
- Explain partial types
- Explain nullable types

14.1 Anonymous Methods

An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.

Typically, delegates can invoke one or more named methods that are included while declaring the delegates. Prior to anonymous methods, if you wanted to pass a small block of code to a delegate, you always had to create a method and then pass it to the delegate. With the introduction of anonymous methods, you can pass an inline block of code to a delegate without actually creating a method.

Figure 14.1 displays an example of anonymous method.

```
void Action()
{
    System.Threading.Thread objThread = new
    System.Threading.Thread
    (delegate()
    {
        Console.Write("Testing... ");
        Console.WriteLine("Threads.");
    });
    objThread.Start();
}
```

} Anonymous Method

Figure 14.1: Anonymous Method

14.1.1 Features

An anonymous method is used in place of a named method if that method is to be invoked only through a delegate. An anonymous method has the following features:

- It appears as an inline code in the delegate declaration.
- It is best suited for small blocks.
- It can accept parameters of any type.
- Parameters using the `ref` and `out` keywords can be passed to it.
- It can include parameters of a generic type.
- It cannot include jump statements such as `goto` and `break` that transfer control out of the scope of the method.

14.1.2 Creating Anonymous Methods

An anonymous method is created when you instantiate or reference a delegate with a block of unnamed code. Following points need to be noted while creating anonymous methods:

- When a delegate keyword is used inside a method body, it must be followed by an anonymous method body.
- The method is defined as a set of statements within curly braces while creating an object of a delegate.
- Anonymous methods are not given any return type.
- Anonymous methods are not prefixed with access modifiers.

Figure 14.2 displays the syntax of anonymous methods.

Syntax for Using Delegates with Anonymous Methods

```
// Create a delegate instance

<access modifier> delegate <return type>
<DelegateName> (parameters);

// Instantiate the delegate using an anonymous method

<DelegateName> <objDelegate> = new <DelegateName>
(parameters)
{ /* ... */ };
```

Figure 14.2: Syntax

Code Snippet 1 creates an anonymous method.

Code Snippet 1:

```
using System;

class AnonymousMethods

{
    //This line remains same even if named methods are used
    delegate void Display();
    static void Main(string[] args)
    {
```

```
//Here is where a difference occurs when using anonymous methods
Display objDisp=delegate()
{
    Console.WriteLine("This illustrates an anonymous method");
};

objDisp();
}
```

In Code Snippet 1, a delegate named **Display** is created. The delegate **Display** is instantiated with an anonymous method. When the delegate is called, it is the anonymous block of code that will execute.

Output:

This illustrates an anonymous method

14.1.3 Referencing Multiple Anonymous Methods

C# allows you to create and instantiate a delegate that can reference multiple anonymous methods. This is done using the **+=** operator. The **+=** operator is used to add additional references to either named or anonymous methods after instantiating the delegate.

Code Snippet 2 shows how one delegate instance can reference several anonymous methods.

Code Snippet 2:

```
using System;
class MultipleAnonymousMethods
{
    delegate void Display();
    static void Main(string[] args)
    {
        //delegate instantiated with one anonymous method reference
        Display objDisp=delegate()
        {
            Console.WriteLine("This illustrates one anonymous method");
        };
        //delegate instantiated with another anonymous method reference
```

```

    objDisp += delegate()
    {
        Console.WriteLine("This illustrates another anonymous
method with the same delegate instance");
    };
    objDisp();
}
}

```

In Code Snippet 2, an anonymous method is created during the delegate instantiation and another anonymous method is created and referenced by the delegate using the `+=` operator.

Output:

This illustrates one anonymous method

This illustrates another anonymous method with the same delegate instance

14.1.4 Outer Variables in Anonymous Methods

An anonymous method can declare variables, which are called outer variables. These variables are said to be captured when they get executed. They exist in memory until the delegate is subjected to garbage collection. The scope of a local variable is only within the method in which it is declared. However, if the anonymous method uses local variables, they exist until the execution of the anonymous method ends. This is true even if the methods in which they are declared are already executed.

14.1.5 Passing Parameters

C# allows passing parameters to anonymous methods. The type of parameters that can be passed to an anonymous method is specified at the time of declaring the delegate. These parameters are specified within parenthesis. The block of code within the anonymous method can access these specified parameters just like any normal method. You can pass the parameter values to the anonymous method while invoking the delegate.

Code Snippet 3 demonstrates how parameters are passed to anonymous methods.

Code Snippet 3:

```

using System;
class Parameters
{
    delegate void Display(string msg, int num);
}

```

```

static void Main(string[] args)
{
    Display objDisp = delegate (string msg, int num)
    {
        Console.WriteLine(msg + num);
    };
    objDisp("This illustrates passing parameters to anonymous methods. The int
parameter passed is: ", 100);
}
}

```

In Code Snippet 3, a delegate **Display** is created. Two arguments are specified in the delegate declaration, a **string** and an **int**. The delegate is then instantiated with an anonymous method to which the **string** and **int** variables are passed as parameters. The anonymous method uses these parameters to display the output.

Output:

This illustrates passing parameters to anonymous methods. The int parameter passed is: 100

Note - If the anonymous method definition does not include any arguments, you can use a pair of empty parenthesis in the declaration of the delegate. The anonymous methods without a parameter list cannot be used with delegates that specify out the parameters.

14.2 Extension Methods

Extension methods allow you to extend an existing type with new functionality without directly modifying those types. Extension methods are static methods that have to be declared in a static class. You can declare an extension method by specifying the first parameter with this keyword. The first parameter in this method identifies the type of objects in which the method can be called. The object that you use to invoke the method is automatically passed as the first parameter.

Syntax:

static return-type MethodName (this type obj, param-list)

where,

return-type: the data type of the return value

MethodName: the extension method name

type: the data type of the object

param-list: the list of parameters (optional)

Code Snippet 4 creates an extension method for a string and converts the first character of the string to lowercase.

Code Snippet 4:

```
using System;  
/// <summary>  
/// Class ExtensionExample defines the extension method  
/// </summary>  
static class ExtensionExample  
{  
    // Extension Method to convert the first character to  
    // lowercase  
    public static string FirstLetterLower(this string result)  
    {  
        if (result.Length > 0)  
        {  
            char[] s = result.ToCharArray();  
            s[0] = char.ToLower(s[0]);  
            return new string(s);  
        }  
        return result;  
    }  
}  
  
class Program  
{  
    public static void Main(string[] args)  
    {  
        string country = "Great Britain";  
        // Calling the extension method  
        Console.WriteLine(country.FirstLetterLower());  
    }  
}
```

In Code Snippet 4, an extension method named **FirstLetterLower** is defined with one parameter

that is preceded with `this` keyword. This method converts the first letter of any sentence or word to lowercase. Note that the extension method is invoked by using the object, `country`. The value 'Great Britain' is automatically passed to the parameter result.

Figure 14.3 depicts the output of Code Snippet 4.

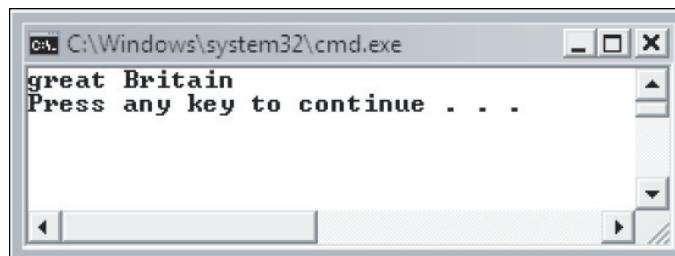


Figure 14.3: Output of Code Snippet 4

The advantages of extension methods are as follows:

- You can extend the functionality of the existing type without modification. This will avoid the problems of breaking source code in existing applications.
- You can add additional methods to standard interfaces without physically altering the existing class libraries.

Code Snippet 5 is an example for an extension method that removes all the duplicate values from a generic collection and displays the result. This program extends the generic `List` class with added functionality.

Code Snippet 5:

```
using System;
using System.Collections.Generic;
/// <summary>
/// Class ExtensionExample defines the extension method
/// </summary>
static class ExtensionExample
{
    // Extension method that accepts and returns a collection.
    public static List<T> RemoveDuplicate<T>(this List<T> allCities)
    {
        List<T> finalCities = new List<T>();
        foreach (var eachCity in allCities)
            if (!finalCities.Contains(eachCity))
                finalCities.Add(eachCity);
    }
}
```

```

finalCities.Add(eachCity);

return finalCities;
}

}

class Program
{

    public static void Main(string[] args)
    {

        List<string> cities = new List<string>();
        cities.Add("Seoul");
        cities.Add("Beijing");
        cities.Add("Berlin");
        cities.Add("Istanbul");
        cities.Add("Seoul");
        cities.Add("Istanbul");
        cities.Add("Paris");
        // Invoke the Extension method, RemoveDuplicate() .
        List<string> result = cities.RemoveDuplicate();
        foreach (string city in result)
            Console.WriteLine(city);
    }
}

```

In Code Snippet 5, the extension method **RemoveDuplicate()** is declared and returns a generic `List` when invoked.

The method accepts a generic `List<T>` as the first argument:

```
public static List<T> RemoveDuplicate<T>(this List<T> allCities)
```

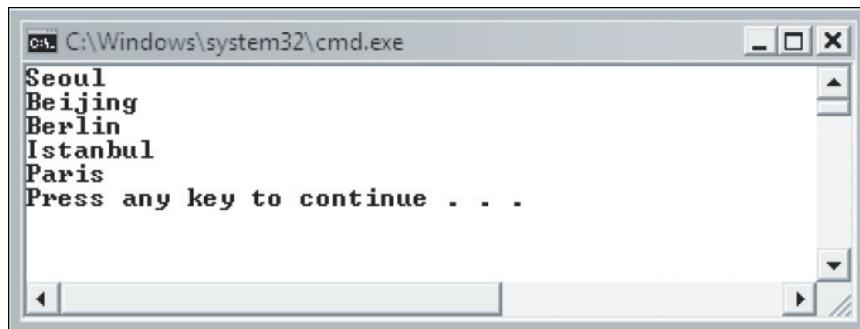
The following lines of code iterate through each value in the collection, remove the duplicate values, and store the unique values in the `List`, **finalCities**.

```

foreach (var eachCity in allCities)
if (!finalCities.Contains(eachCity))
finalCities.Add(eachCity);

```

Figure 14.4 displays the output of Code Snippet 5.



```
C:\Windows\system32\cmd.exe
Seoul
Beijing
Berlin
Istanbul
Paris
Press any key to continue . . .
```

Figure 14.4: Output of Code Snippet 5

Note - Even though the extension method is declared as static, you can still call it using an object.

14.3 Anonymous Types

Anonymous type is basically a class with no name and is not explicitly defined in code. An anonymous type uses object initializers to initialize properties and fields. Since it has no name, you need to declare an implicitly typed variable to refer to it.

Syntax:

```
new { identifierA = valueA, identifierB = valueB, ..... }
```

where,

identifierA, identifierB,...: Identifiers that will be translated into read-only properties that are initialized with values

Code Snippet 6 demonstrates the use of anonymous types.

Code Snippet 6:

```
using System;
/// <summary>
/// Class AnonymousTypeExample to demonstrate anonymous type
/// </summary>
class AnonymousTypeExample
{
    public static void Main(string[] args)
    {
        // Anonymous Type with three properties.
        var stock = new { Name = "Michigan Enterprises", Code = 1301, Price = 35056.75 };
    }
}
```

```

Console.WriteLine("Stock Name: " + stock.Name);
Console.WriteLine("Stock Code: " + stock.Code);
Console.WriteLine("Stock Price: " + stock.Price);
}
}

```

Consider the following line of code:

```
var stock = new { Name = "Michigan Enterprises", Code = 1301, Price = 35056.75 };
```

The compiler creates an anonymous type with all the properties that is inferred from object initializer. In this case, the type will have properties **Name**, **Code**, and **Price**. The compiler automatically generates the **get** and **set** methods, as well as the corresponding private variables to hold these properties. At run-time, the C# compiler creates an instance of this type and the properties are given the values Michigan Enterprises, 1301, and 35056.75 respectively.

Figure 14.5 displays output of Code Snippet 6.

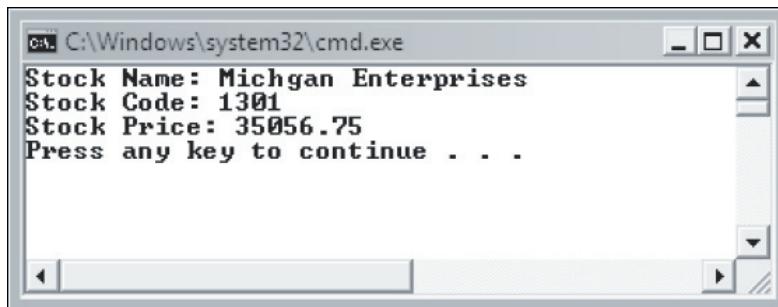


Figure 14.5: Output of Code Snippet 6

When an anonymous type is created, the C# compiler carries out the following tasks:

- Interprets the type
- Generates a new class
- Use the new class to instantiate a new object
- Assigns the object with the required parameters

The compiler internally creates a class with the respective properties when Code Snippet 6 is compiled.

In this program, the class might look like the one that is shown in Code Snippet 7.

Code Snippet 7:

```
class __NO_NAME__
{
    private string _Name;
    private int _Code;
    private double _Price;
    public string Name
    {
        get { return _Name; }
        set { _Name = value; }
    }
    public int Code
    {
        get { return _Code; }
        set { _Code = value; }
    }
    public double Price
    {
        get { return _Price; }
        set { _Price = value; }
    }
}
```

Note - The C# compiler generates the anonymous type at compile time, not at run-time.

Code Snippet 8 demonstrates passing an instance of the anonymous type to a method and displaying the details.

Code Snippet 8:

```
using System;
using System.Reflection;
/// <summary>
/// Class Employee to demonstrate anonymous type.
/// </summary>
```

```
///</summary>
public class Employee
{
    public void DisplayDetails(object emp)
    {
        String fName = "";
        String lName = "";
        int age = 0;

        PropertyInfo[] attrs = emp.GetType().GetProperties();
        foreach (PropertyInfo attr in attrs)
        {
            switch (attr.Name)
            {
                case "FirstName":
                    fName = attr.GetValue(emp, null).ToString();
                    break;
                case "LastName":
                    lName = attr.GetValue(emp, null).ToString();
                    break;
                case "Age":
                    age = (int)attr.GetValue(emp, null);
                    break;
            }
        }
        Console.WriteLine("Name: {0} {1}, Age: {2}", fName, lName, age);
    }
}

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

```
{
    Employee david = new Employee();
    // Creating the anonymous type instance and passing it to a method.
    david.DisplayDetails(new { FirstName = "David", LastName = "Blake", Age = 30 });
}
```

Code Snippet 8 creates an instance of the anonymous type with three properties, `FirstName`, `LastName`, and `Age` with values David, Blake, and 30 respectively. This instance is then passed to the method, `DisplayDetails()`.

In `DisplayDetails()` method, the instance that was passed as parameter is stored in the object, `emp`. Then, the code uses reflection to query the object's properties. The `GetType()` method retrieves the type of the current instance, `emp` and `GetProperties()` method retrieves the properties of the object, `emp`. The details are then stored in the `PropertyInfo` collection, `attr`. Finally, the details are extracted through the `GetValue()` method of `PropertyInfo` class. If this program did not make use of an anonymous type, a lot more code would have been required to produce the same output.

Figure 14.6 displays the output of Code Snippet 8.

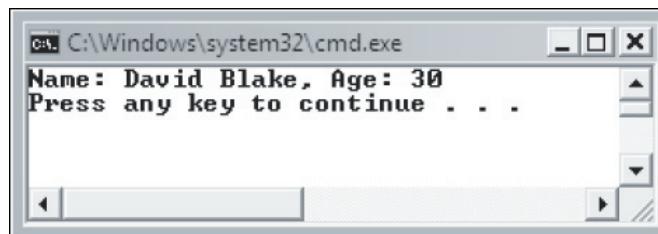


Figure 14.6: Output of Code Snippet 8

14.4 Partial Types

Assume that a large organization has its IT department spread over two locations, Melbourne and Sydney. The overall functioning takes place through consolidated data gathered from both the locations. The customer of the organization would see it as a whole entity, whereas, in reality, it would be composed of multiple units.

Now, think of a very large C# class or structure with lots of member definitions. You can split the data members of the class or structure and store them in different files. These members can be combined into a single unit while executing the program. This can be done by creating partial types.

14.4.1 Features of Partial Types

The partial types feature facilitates the definition of classes, structures, and interfaces over multiple files. Partial types provide various benefits. These are as follows:

- They separate the generator code from the application code.
- They help in easier development and maintenance of the code.
- They make the debugging process easier.
- They prevent programmers from accidentally modifying the existing code.

Figure 14.7 displays an example of a partial type.

File 1	File 2
<pre>partial struct Sample { <MethodOne>; }</pre>	<pre>partial struct Sample { <MethodTwo>; }</pre>

Figure 14.7: Partial Type

Note - C# does not support partial type definitions for enumerations. However, generic types can be partial.

14.4.2 Merged Elements during Compilation

The members of partial classes, partial structures, or partial interfaces declared and stored at different locations are combined together at the time of compilation. These members can include:

- XML comments
- Interfaces
- Generic-type parameters
- Class variables
- Local variables
- Methods

→ Properties

A partial type can be compiled at the Developer Command Prompt for VS2012. The command to compile a partial type is:

```
csc /out:<FileName>.exe <CSharpFileNameOne>.cs <CSharpFileNameTwo>.cs
```

where,

FileName: Is the user specified name of the .exe file.

CSharpFileNameOne: Is the name of the first file where a partial type is defined.

CSharpFileNameTwo: Is the name of the second file where a partial type is defined.

Finally, you can directly run the .exe file to see the required output. This is demonstrated in these two Code Snippets 9 and 10.

Code Snippet 9:

```
using System;
using System.Collections.Generic;
using System.Text;
//Stored in StudentDetails.csfile
namespace School
{
    public partial class StudentDetails
    {
        int _rollNo;
        string _studName;
        public StudentDetails(int number, string name)
        {
            _rollNo = number;
            _studName = name;
        }
    }
}
```

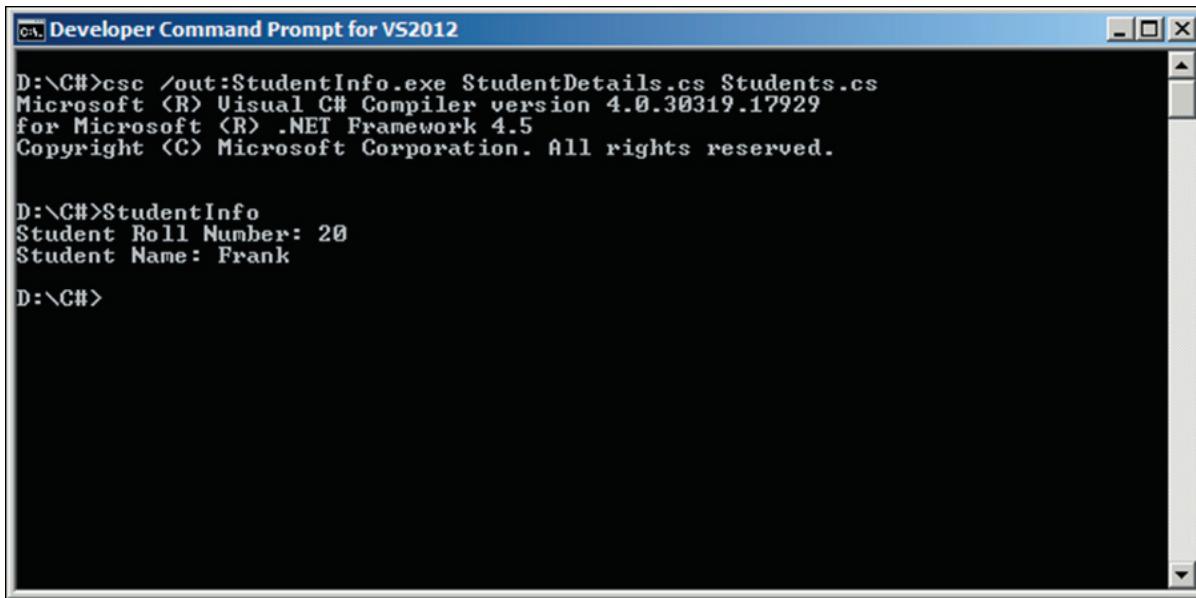
Code Snippet 10:

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

```
using System.Text;
//Stored in Students.cs file
namespace School
{
    public partial class StudentDetails
    {
        public void Display()
        {
            Console.WriteLine("Student Roll Number: " + _rollNo);
            Console.WriteLine("Student Name: " + _studName);
        }
    }
    public class Students
    {
        static void Main(string[] args)
        {
            StudentDetails objStudents = new StudentDetails(20, "Frank");
            objStudents.Display();
        }
    }
}
```

In Code Snippets 9 and 10, the partial class **StudentDetails** exists in two different files. When both these files are compiled at the Visual Studio Command Prompt, an .exe file is created which merges the **StudentDetails** class from both the files. On executing the .exe file at the command prompt, the student's roll number and name are displayed as output.

Figure 14.8 shows how to compile and execute the `StudentDetails.cs` and `Students.cs` files created in the examples using Developer Command Prompt for VS2012.



```
C:\ Developer Command Prompt for VS2012
D:\C#\>csc /out:StudentInfo.exe StudentDetails.cs Students.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

D:\C#\>StudentInfo
Student Roll Number: 20
Student Name: Frank

D:\C#\>
```

Figure 14.8: Compilation and Execution of Examples

14.4.3 Rules for Partial Types

There are certain rules for creating and working with partial types. These rules must be followed, without which a user might not be able to create partial types successfully.

The rules are as follows:

- The partial-type definitions must include the `partial` keyword in each file.
- The `partial` keyword must always follow the `class`, `struct`, or `interface` keywords.
- The partial-type definitions of the same type must be saved in the same assembly.
- Generic types can be defined as partial. Here, the type parameters and its order must be the same in all the declarations.
- The partial-type definitions can contain certain C# keywords which must exist in the declaration in different files. These keywords are as follows:
 - `public`
 - `private`
 - `protected`
 - `internal`

- abstract
- sealed
- new

14.4.4 Implementing Partial Types

Partial types are implemented using the `partial` keyword. This keyword specifies that the code is split into multiple parts and these parts are defined in different files and namespaces.

The type names of all the constituent parts of a partial code are prefixed with the `partial` keyword. For example, if the complete definition of a structure is split over three files, each file must contain a partial structure having the `partial` keyword preceding the type name. Also, each of the partial parts of the code must have the same access modifier.

The following syntax is used to split the definition of a class, a struct, or an interface.

Syntax:

`[<access_modifier>] [<keyword>] partial <type> <Identifier>`

where,

`access_modifier`: Is an optional access modifier such as `public`, `private`, and so on.

`keyword`: Is an optional keyword such as `abstract`, `sealed`, and so on.

`type`: Is a specification for a class, a structure, or an interface.

`Identifier`: Is the name of the class, structure, or interface.

Code Snippet 11 creates an interface with two partial interface definitions.

Code Snippet 11:

```
using System;
//ProgramName: MathsDemo.cs
partial interface MathsDemo
{
    int Addition(int valOne, int valTwo);
}
//ProgramName: MathsDemo2.cs
partial interface MathsDemo
{
    int Subtraction(int valOne, int valTwo);
}
```

```

class Calculation : MathsDemo
{
    public int Addition(int valOne, int valTwo)
    {
        return valOne + valTwo;
    }

    public int Subtraction(int valOne, int valTwo)
    {
        return valOne - valTwo;
    }

    static void Main(string[] args)
    {
        int numOne = 45;
        int numTwo = 10;
        Calculation objCalculate = new Calculation();
        Console.WriteLine("Addition of two numbers: " + objCalculate.Addition
            (numOne, numTwo));
        Console.WriteLine("Subtraction of two numbers: " +
            objCalculate.Subtraction(numOne, numTwo));
    }
}

```

In Code Snippet 11, a partial interface **Maths** is created that contains the **Addition** method. This file is saved as **MathsDemo.cs**. The remaining part of the same interface contains the **Subtraction** method and is saved under the filename **MathsDemo2.cs**. This file also includes the class **Calculation**, which inherits the interface **Maths** and implements the two methods, **Addition** and **Subtraction**.

Output:

Addition of two numbers: 55

Subtraction of two numbers: 35

Note - If a part of code stored in one file is declared as abstract and the other parts are declared as public, the entire code is considered abstract. This same rule applies for sealed classes.

14.4.5 Partial Classes

A class is one of the types in C# that supports partial definitions. Classes can be defined over multiple locations to store different members such as variables, methods, and so on. Although, the definition of the class is split into different parts stored under different names, all these sections of the definition are combined during compilation to create a single class.

You can create partial classes to store private members in one file and public members in another file. More importantly, multiple developers can work on separate sections of a single class simultaneously, if the class itself is spread over separate files.

Code Snippet 12 creates two partial classes that display the name and roll number of a student.

Code Snippet 12:

```
using System;

//Program: StudentDetails.cs

public partial class StudentDetails
{
    public void Display()
    {
        Console.WriteLine("Student Roll Number: " + _rollNo);
        Console.WriteLine("Student Name: " + _studName);
    }
}

//Program StudentDetails2.cs

public partial class StudentDetails
{
    int _rollNo;
    string _studName;
    public StudentDetails(int number, string name)
    {
        _rollNo = number;
        _studName = name;
    }
}

public class Students
```

```
{
    static void Main(string[] args)
    {
        StudentDetails objStudents = new StudentDetails(20, "Frank");
        objStudents.Display();
    }
}
```

In Code Snippet 12, the class `StudentDetails` has its definition spread over two files, `StudentDetails.cs` and `StudentDetails2.cs`. `StudentDetails.cs` contains the part of the class that contains the `Display()` method. `StudentDetails2.cs` contains the remaining part of the class that includes the constructor. The class `Students` creates an instance of the class `StudentDetails` and invokes the method `Display`. The output displays the roll number and the name of the student.

Output:

Student Roll Number: 20

Student Name: Frank

14.4.6 Partial Methods

Consider a partial class `Shape` whose complete definition is spread over two files. Now, consider that a method `Create()` has a signature defined in `Shape`.

The partial class `Shape` contains the definition of `Create()` in `Shape.cs`. The remaining part of partial class `Shape` is present in `RealShape.cs` and it contains the implementation of `Create()`. Hence, `Create()` is a partial method whose definition is spread over two files.

A partial method is a method whose signature is included in a partial type, such as a partial class or struct. The method may be optionally implemented in another part of the partial class or type or same part of the class or type.

Code Snippets 13 and 14 illustrate how to create and use partial methods. Code Snippet 13 contains only the signature and Code Snippet 14 contains the implementation.

Code Snippet 13:

```
using System;
namespace PartialTest
{
    /// <summary>
    /// Class Shape is a partial class and defines a partial method.
}
```

```
///</summary>
public partial class Shape
{
    partial void Create();
}
```

Code Snippet 14:

```
using System;
namespace PartialTest
{
    ///<summary>
    /// Class Shape is a partial class and contains the implementation
    /// of a partial method.
    ///</summary>
    public partial class Shape
    {
        partial void Create()
        {
            Console.WriteLine("Creating Shape");
        }
        public void Test()
        {
            Create();
        }
    }
    class Program
    {
        static void Main(String[] args)
        {
            Shape s = new Shape();
        }
    }
}
```

```
    s.Test();  
}  
}  
}
```

By separating the definition and implementation into two files, it is possible that two developers can work on them or even use a code-generator tool to create the definition of the method. Also, it is up to the developer whether to implement the partial method or not. It is also valid to have both the signature and implementation of **Create()** in the same part of **Shape**.

Code Snippet 15 demonstrates how you can define and implement a method in a single file.

Code Snippet 15:

```
namespace PartialTest  
{  
    /// <summary>  
    /// Class Shape is a partial class and contains the definition and  
    /// implementation of a partial method.  
    /// </summary>  
    public partial class Shape  
    {  
        partial void Create();  
        ...  
        ...  
        partial void Create()  
        {  
            Console.WriteLine("Creating Shape");  
        }  
        public void Test()  
        {  
            Create();  
        }  
    }  
    class Program  
    {
```

```
static void Main(String[] args)
{
    Shape s = new Shape();
    s.Test();
}
```

It is possible to have only the signature of `Create()` in one part of `Shape` and no implementation of `Create()` anywhere. In that case, the compiler removes all references to `Create()`, including any method calls.

A partial method must always include the `partial` keyword.

Also, partial methods can be defined only within a partial class or type. If the class containing the definition or implementation of a partial method does not have the `partial` keyword, then a compile-time error would be raised.

Some of the restrictions when working with partial methods are as follows:

- The `partial` keyword is a must when defining or implementing a partial method
- Partial methods must return `void`
- They are implicitly `private`
- Partial methods can return `ref` but not `out`
- Partial methods cannot have any access modifier such as `public`, `private`, and so forth, or keywords such as `virtual`, `abstract`, `sealed`, or so forth

Partial methods are useful when you have part of the code auto-generated by a tool or IDE and want to customize the other parts of the code.

14.4.7 Using Partial Types

A large project in an organization involves creation of multiple structures, classes, and interfaces. If these types are stored in a single file, their modification and maintenance becomes very difficult. In addition, multiple programmers working on the project cannot use the file at the same time for modification. Thus, partial types can be used to split a type over separate files, allowing the programmers to work on them simultaneously.

Partial types are also used with the code generator in Visual Studio 2012. You can add the auto-generated code into your file without recreation of the source file. You can use partial types for both these codes.

Note - When a GUI-based project is created using the Windows Forms technology in Visual Studio 2012, a partial class is automatically created to hold the form design.

14.4.8 Inheriting Partial Classes

A partial class can be inherited just like any other class in C#. It can contain virtual methods defined in different files which can be overridden in its derived classes. In addition, a partial class can be declared as an abstract class using the `abstract` keyword. Abstract partial classes can be inherited.

Code Snippets 16 and 17 demonstrate how to inherit a partial class.

Code Snippet 16:

```
//The following code is stored in Geometry.cs file
using System;
abstract partial class Geometry
{
    public abstract double Area(double val);
}
```

Code Snippet 17:

```
//The following code is stored in Cube.cs file
using System;
abstract partial class Geometry
{
    public virtual void Volume(double val)
    {
    }
}
class Cube : Geometry
{
    public override double Area (double side)
    {
        return 6 * (side * side);
    }
    public override void Volume (double side)
```

```

{
    Console.WriteLine("Volume of cube: " + (side * side));
}

static void Main(string[] args)
{
    double number = 20.56;
    Cube objCube = new Cube();
    Console.WriteLine("Area of Cube: " + objCube.Area(number));
    objCube.Volume(number);
}
}

```

In Code Snippets 16 and 17, the abstract partial class **Geometry** is defined across two C# files. It defines an abstract method called **Area()** and a virtual method called **Volume()**. Both these methods are inherited in the derived class called **Cube**.

Output:

Area of Cube: 2536.2816
Volume of cube: 422.7136

14.5 Nullable Types

C# provides nullable types to identify and handle value type fields with null values. Before this feature was introduced, only reference types could be directly assigned null values. Value type variables with null values were indicated either by using a special value or an additional variable. This additional variable indicated whether or not the required variable was null.

Special values are only beneficial if the decided value is followed consistently across applications. Creating and managing additional fields for such variables leads to more memory space and becomes tedious. These problems are solved by the introduction of nullable types.

14.5.1 Creating Nullable Types

A nullable type is a means by which null values can be defined for the value types. It indicates that a variable can have the value **null**. Nullable types are instances of the **System.Nullable<T>** structure.

A variable can be made nullable by adding a question mark following the data type. Alternatively, it can be declared using the generic **Nullable<T>** structure present in the **System** namespace.

14.5.2 Characteristics

Nullable types in C# have the following characteristics:

- They represent a value type that can be assigned a null value.
- They allow values to be assigned in the same way as that of the normal value types.
- They return the assigned or default values for nullable types.
- When a nullable type is being assigned to a non-nullable type and the assigned or default value has to be applied, the ?? operator is used.

Note - One of the uses of a nullable type is to integrate C# with databases that include null values in the fields of a table. Without nullable types, there was no way to represent such data accurately. For example, if a `bool` variable contained a value that was neither true nor false, there was no way to indicate this.

14.5.3 Implementing Nullable Types

A nullable type can include any range of values that is valid for the data type to which the nullable type belongs. For example, a `bool` type that is declared as a nullable type can be assigned the values `true`, `false`, or `null`. Nullable types have two public read-only properties that can be implemented to check the validity of nullable types and to retrieve their values.

These are as follows:

→ The `HasValue` Property

`HasValue` is a `bool` property that determines validity of the value in a variable. The `HasValue` property returns a `true` if the value of the variable is `not null`, else it returns `false`.

→ The `Value` Property

The `Value` property identifies the value in a nullable variable. When the `HasValue` evaluates to `true`, the `Value` property returns the value of the variable, otherwise it returns an exception.

Code Snippet 18 displays the employee's name, ID, and role using the nullable types.

Code Snippet 18:

```
using System;
class Employee
{
    static void Main(string[] args)
```

```
{
    int empId = 10;
    string empName = "Patrick";
    char? role = null;
    Console.WriteLine("Employee ID: " + empId);
    Console.WriteLine("Employee Name: " + empName);
    if (role.HasValue == true)
    {
        Console.WriteLine("Role: " + role.Value);
    }
    else
    {
        Console.WriteLine("Role: null");
    }
}
```

In Code Snippet 18, **empId** is declared as an integer variable and it is initialized to value 10 and **empName** is declared as a string variable and it is assigned the name **Patrick**. Additionally, **role** is defined as a nullable character with **null** value. The output displays the role of the employee as **null**.

Output:

Employee ID: 10

Employee Name: Patrick

Role: null

14.5.4 Nullable Types in Expressions

C# allows you to use nullable types in expressions that can result in a **null** value. Thus, an expression can contain both, nullable types and non-nullable types. An expression consisting of both, the nullable and non-nullable types, results in the value **null**.

Code Snippet 19 demonstrates the use of nullable types in expressions.

Code Snippet 19:

```
using System;
```

```
class Numbers
{
    static void Main (string[] args)
    {
        System.Nullable<int> numOne = 10;
        System.Nullable<int> numTwo = null;
        System.Nullable<int> result = numOne + numTwo;
        if (result.HasValue == true)
        {
            Console.WriteLine ("Result: " + result);
        }
        else
        {
            Console.WriteLine ("Result: null");
        }
    }
}
```

In Code Snippet 19, **numOne** and **numTwo** are declared as integer variables and initialized to values 10 and null respectively. In addition, **result** is declared as an integer variable and initialized to a value which is the sum of **numOne** and **numTwo**. The result of this **sum** is a null value and this is indicated in the output.

Output:

Result: null

14.5.5 The ?? Operator

A nullable type can either have a defined value or the value can be undefined. If a nullable type contains a null value and you assign this nullable type to a non-nullable type, the compiler generates an exception called `System.InvalidOperationException`.

To avoid this problem, you can specify a default value for the nullable type that can be assigned to a non-nullable type using the ?? operator. If the nullable type contains a null value, the ?? operator returns the default value.

Code Snippet 20 demonstrates the use of ?? operator.

Code Snippet 20:

```
using System;

class Salary
{
    static void Main(string[] args)
    {
        double? actualValue = null;
        double marketValue = actualValue ?? 0.0;
        actualValue = 100.20;
        Console.WriteLine("Value: " + actualValue);
        Console.WriteLine("Market Value: " + marketValue);
    }
}
```

In Code Snippet 20, the variable **actualValue** is declared as `double` with a `? symbol` and initialized to value `null`. This means that **actualValue** is now a nullable type with a value of `null`.

When it is assigned to `marketValue`, a `??` operator has been used. This will assign `marketValue` the default value of `0.0`.

Output:

Value: 100.2

Market Value: 0

14.5.6 Converting Nullable Types

C# allows any value type to be converted into nullable type, or a nullable type into a value type. C# supports two types of conversions on nullable types:

- ➔ Implicit conversion
- ➔ Explicit conversion

The storing of a value type into a nullable type is referred to as implicit conversion. A variable to be declared as nullable type can be set to null using the `null` keyword.

This is illustrated in Code Snippet 21.

Code Snippet 21:

```
using System;

class ImplicitConversion

{
    static void Main(string[] args)
    {
        int? numOne = null;

        if (numOne.HasValue == true)
        {
            Console.WriteLine("Value of numOne before
conversion: " + numOne);
        }
        else
        {
            Console.WriteLine("Value of numOne: null");
        }
        numOne = 20;

        Console.WriteLine("Value of numOne after implicit
conversion: " + numOne);
    }
}
```

In Code Snippet 21, the variable **numOne** is declared as nullable. The `HasValue` property is being used to check whether the variable is of a null type. Then, **numOne** is assigned the value 20, which is of `int` type stored in a nullable type. This is implicit conversion.

Output:

```
Value of numOne: null
Value of numOne after implicit conversion: 20
```

The conversion of a nullable type to a value type is referred to as explicit conversion. This is illustrated in Code Snippet 22.

Code Snippet 22:

```
using System;

class ExplicitConversion
{
    static void Main(string[] args)
    {
        int? numOne = null;
        int numTwo = 20;
        int? resultOne = numOne + numTwo;
        if (resultOne.HasValue == true)
        {
            Console.WriteLine("Value of resultOne before conversion: " + resultOne);
        }
        else
        {
            Console.WriteLine("Value of resultOne: null");
        }
        numOne = 10;
        int result = (int) (numOne + numTwo);
        Console.WriteLine("Value of result after implicit conversion: " +
            result);
    }
}
```

In Code Snippet 22, the **numOne** and **resultOne** variables are declared as null. The **HasValue** property is being used to check whether the **resultOne** variable is of a null type. Then, **numOne** is assigned the value 10, which is of **int** type stored in a nullable type. The values in both the variables are added and the result is stored in the **result** variable of **int** type. This is explicit conversion.

Output:

```
Value of resultOne: null
Value of resultTwo after explicit conversion: 30
```

14.5.7 Boxing Nullable Types

An instance of the `object` type can be created as a nullable type that can hold both null and non-null values. The instance can be boxed only if it holds a non-null value and the `HasValue` property returns true. In this case, only the data type of the nullable variable is converted to type `object`. While boxing, if the `HasValue` property returns false, the object is assigned a null value.

Code Snippet 23 demonstrates how to box nullable types.

Code Snippet 23:

```
using System;

class Boxing

{
    static void Main(string[] args)
    {
        int? number = null;
        object objOne = number;
        if (objOne != null)
        {
            Console.WriteLine("Value of object one: " +
                objOne);
        }
        else
        {
            Console.WriteLine("Value of object one: null");
        }
        double? value = 10.26;
        object objTwo = value;
        if (objTwo != null)
        {
            Console.WriteLine("Value of object two: " +
                objTwo);
        }
        else
```

```
{  
    Console.WriteLine("Value of object one: null");  
}  
}  
}  
}
```

In Code Snippet 23, the **number** variable declared as nullable is boxed and its value is stored in **objOne** as null. The **value** variable declared as nullable is boxed and its value is stored in **objTwo** as 10.26.

Output:

Value of object one: null

Value of object two: 10.26

14.6 Check Your Progress

1. Which of these statements about the anonymous methods of C# are true?

(A)	Anonymous methods are blocks of code that can be included with a delegate instantiation.
(B)	The parameters of an anonymous method are always of generic type.
(C)	Anonymous methods are recommended for small as well as large blocks of code.
(D)	Variables declared in an anonymous method are called outer variables.
(E)	The values of parameters for an anonymous method are passed when the delegate is invoked.

(A)	A, D, E	(C)	C, E
(B)	B, C	(D)	D

2. You are trying to display the following output:

"Student Name: James"

"Student Id: 20"

Which of the following codes will help you to achieve this?

(A)	<pre>public delegate void Show(); class Student { string _studName; int _studId; public Student(string name, int id) { _studName = name; _studId = id; } static void Main(string[] args) { Student objStudent = new Student("James", 20); Show objShow = delegate() {</pre>
-----	---

(A)	<pre> Console.WriteLine("Student Name: " + objStudent._studName); Console.WriteLine("Student Id: " + objStudent._studId); } objShow(); } } </pre>
(B)	<pre> public delegate void Show(); class Student { string _studName; int _studId; public Student(string name, int id) { _studName = name; _studId = id; } static void Main(string[] args) { Student objStudent = new Student("James", 20); Show objShow = new delegate() { Console.WriteLine("Student Name: " + objStudent._studName); Console.WriteLine("Student Id: " + objStudent._studId); }; objShow(); } } </pre>

(C)	<pre> public delegate void Show(); class Student { string _studName; int _studId; public Student(string name, int id) { _studName = name; _studId = id; } static void Main(string[] args) { Student objStudent = new Student("James", 20); Show objShow = new Show() { Console.WriteLine("Student Name: " + objStudent._studName); Console.WriteLine("Student Id: " + objStudent._studId); }; objShow(); } } </pre>
(D)	<pre> public delegate void Show(); class Student { string _studName; int _studId; public Student(string name, int id) { _studName = name; } } </pre>

```

(D)           _studId=id;
             }

             static void Main(string[] args)
             {
                 Student objStudent = new Student ("James", 20);
                 Show objShow = delegate ()
                 {
                     Console.WriteLine ("Student Name: " + objStudent._studName);
                     Console.WriteLine ("Student Id: " + objStudent._studId);
                 };
                 objShow ();
             }
         }
    
```

(A)	A	(C)	C
(B)	B	(D)	D

3. Which of these statements about partial types in C# are true?

(A)	The partial types in C# 2.0 allow you to store different methods of an interface in different source files.
(B)	Partial types allow you to split a class definition over different files by specifying different access modifiers.
(C)	Partial classes allow you to store the members of a class in different files in different namespaces.
(D)	The declaration of partial types involves the use of <code>partialtype</code> keyword.
(E)	Partial types enable the compiler to merge only the scattered private data members of a class during compilation.

(A)	A	(C)	C
(B)	B	(D)	D

4. You are trying to display the output 'Book Name: Freakonomics'. Assuming that the interface definitions are stored in separate files, which of the following codes will help you achieve this?

(A)

```
Partial interface Details
{
    void AddDetails(string val);
}

partial interface Details
{
    void Display();
}

class Book : Details
{
    string _bookName;
    public void AddDetails(string name)
    {
        _bookName = name;
    }

    public void Display()
    {
        Console.WriteLine("Book Name: " + _bookName);
    }

    static void Main(string[] args)
    {
        Book objBook = new Book();
        objBook.AddDetails("Freakonomics");
        objBook.Display();
    }
}
```

```
(B)
partial interface Details
{
    void AddDetails(string val);
}
interface Details
{
    void Display();
}
class Book : Details
{
    string _bookName;
    public void AddDetails(string name)
    {
        _bookName = name;
    }
    public void Display()
    {
        Console.WriteLine("Book Name: " + _bookName);
    }
    static void Main(string[] args)
    {
        Book objBook = new Book();
        objBook.AddDetails("Freakonomics");
        objBook.Display();
    }
}
```

```
(C)
partial interface Details
{
    void AddDetails(string val);
}
partial interface Details
{
    void Display();
}
```

(C)

```

class Book : Details
{
    string _bookName;
    void AddDetails(string name)
    {
        _bookName = name;
    }
    void Display()
    {
        Console.WriteLine("Book Name: " + _bookName);
    }
    static void Main(string[] args)
    {
        Book objBook = new Book();
        objBook.AddDetails("Freakonomics");
        objBook.Display();
    }
}

```

(D)

```

partial interface Details
{
    void AddDetails(string val);
}

partial interface Details
{
    void Display();
}

class Book : Details
{
    string _bookName;
}

```

(D)

```

void AddDetails(string name)
{
    _bookName = name;
}

void Display()
{
    Console.WriteLine("Book Name: " + _bookName);
}

static void Main(string[] args)
{
    Book objBook = new Book();
    objBook.AddDetails("Freakonomics");
    objBook.Display();
}
}

```

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of these statements about nullable types in C# are true?

(A)	Nullable types identify the fields that contain the value zero.
(B)	Nullable types use the generic structure System.Nullable<T> for their declaration.
(C)	Nullable types return the assigned or default values using the ?? operator.
(D)	The HasValue property identifies the value in a variable.
(E)	The Value property determines the validity of the value in a variable.

(A)	A	(C)	C
(B)	B	(D)	D

6. You are trying to display the following output:

"Working Capital: 1000.20"

"Cash in hand: 0.0"

Which of the following codes will help you to achieve this?

(A)	static void Main(string[] args) { double capital = null; double cash = capital ?? 0.0; capital = 1000.20; Console.WriteLine("Working Capital: " + capital); Console.WriteLine("Cash in hand: " + cash); }
(B)	static void Main(string[] args) { double? capital = null; double cash = capital ?? 0.0; capital = 1000.20; Console.WriteLine("Working Capital: " + capital); Console.WriteLine("Cash in hand: " + cash); }
(C)	static void Main(string[] args) { double capital = null; double cash = capital ? 0.0; capital = 1000.20; Console.WriteLine("Working Capital: " + capital); Console.WriteLine("Cash in hand: " + cash); }

(D)

```
static void Main(string[] args)
{
    double capital = null;
    double cash = capital;
    capital = 1000.20;
    Console.WriteLine("Working Capital: " + capital);
    Console.WriteLine("Cash in hand: " + cash);
}
```

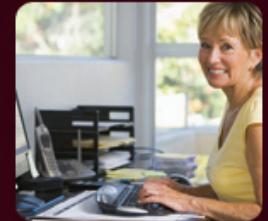
14.6.1 Answers

1.	A
2.	D
3.	A
4.	A
5.	B
6.	B



Summary

- ➔ Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- ➔ Extension methods allow you to extend different types with additional static methods.
- ➔ You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.
- ➔ Partial types allow you to split the definitions of classes, structs, and interfaces to store them in different C# files.
- ➔ You can define partial types using the partial keyword.
- ➔ Nullable types allow you to assign null values to the value types.
- ➔ Nullable types provide two public read-only properties, HasValue and Value.



Session -15

Advanced Concepts in C#

Welcome to the Session, **Advanced Concepts in C#**.

To facilitate development of applications that caters to various needs, C# provides advanced language enhancements, such as system-defined delegates, lambda expressions, and so on. To develop data-centric and distributed applications, the .NET Framework provides Entity Framework and Windows Communication Framework (WCF). To create more responsive multi-user applications, C# supports multithreading and concurrent programming. C# applications can also interact with dynamic languages, such as IronRuby and IronPython.

In this session, you will learn to:

- Describe system-defined generic delegates
- Define lambda expressions
- Explain query expressions
- Describe Windows Communication Framework (WCF)
- Explain parallel programming
- Explain dynamic programming

15.1 System-Defined Generic Delegates

A delegate is a reference to a method. Consider an example to understand this. You create a delegate `CalculateValue` to point to a method that takes a `string` parameter and returns an `int`. You need not specify the method name at the time of creating the delegate. At some later stage in your code, you can instantiate the delegate by assigning it a method name.

The .NET Framework and C# have a set of predefined generic delegates that take a number of parameters of specific types and return values of another type. The advantage of these predefined generic delegates is that they are ready for reuse with minimal coding.

Some of the commonly used predefined generic delegates are as follows:

→ **`Func<TResult>() Delegate`**

It represents a method having zero parameters and returns a value of type `TResult`.

→ **`Func<T, TResult>(T arg) Delegate`**

It represents a method having one parameter of type `T` and returns a value of type `TResult`.

→ **`Func<T1, T2, TResult>(T1 arg1, T2 arg2) Delegate`**

It represents a method having two parameters of type `T1` and `T2` respectively and returns a value of type `TResult`.

Code Snippet 1 determines the length of a given word or phrase. It makes use of the `Func<T, TResult>(T arg)` predefined generic delegate that takes one parameter and returns a result.

Code Snippet 1:

```
/// <summary>
/// Class WordLength determines the length of a given word or phrase
/// </summary>
public class WordLength
{
    public static void Main()
    {
        // Instantiate delegate to reference Count method
        Func<string, int> count = Count;
        string location = "Netherlands";
        // Use delegate instance to call Count method
        Console.WriteLine("The number of characters in the input is:
{0}", count(location).ToString());
```

```

    }

    private static int Count (string inputString)
    {
        return inputString.Length;
    }

}

```

Figure 15.1 shows the use of a predefined generic delegate.

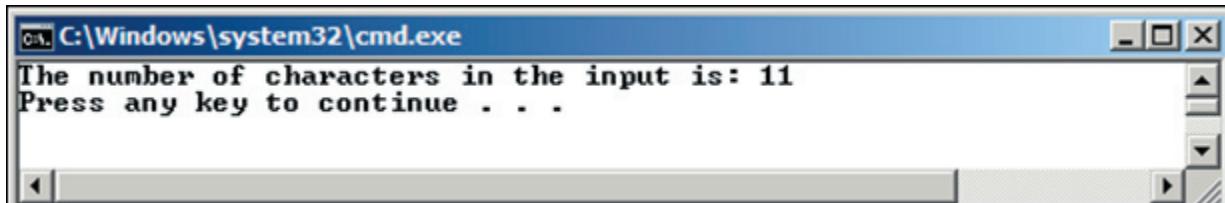


Figure 15.1: Predefined Generic Delegate

15.2 Lambda Expressions

A method associated with a delegate is never invoked by itself, instead, it is only invoked through the delegate. Sometimes, it can be very cumbersome to create a separate method just so that it can be invoked through the delegate. To overcome this, anonymous methods and lambda expressions can be used. Anonymous methods allow unnamed blocks of code to be created for representing a method referred by a delegate. A lambda expression is an anonymous expression that can contain expressions and statements and enables to simplify development through inline coding. In simple terms, a lambda expression is an inline expression or statement block having a compact syntax and can be used wherever a delegate or anonymous method is expected.

Syntax:

parameter-list => expression or statements

where,

parameter-list: is an explicitly typed or implicitly typed parameter list

=> : is the lambda operator

expression or statements : are either an expression or one or more statements

For example, the following code is a lambda expression:

```
word => word.Length;
```

Consider a complete example to illustrate the use of lambda expressions. Assume that you want to calculate the square of an integer number. You can use a method `square()` and pass the method name as a parameter to the `Console.WriteLine()` method.

Code Snippet 2 uses a lambda expression to calculate the square of an integer number.

Code Snippet 2:

```
class Program
{
    delegate int ProcessNumber(int input);
    static void Main(string[] args)
    {
        ProcessNumber del = input => input * input;
        Console.WriteLine(del(5));
    }
}
```

The `=>` operator is pronounced as ‘goes to’ or ‘go to’ in case of multiple parameters. Here, the expression, `input => input * input` means: given the value of `input`, calculate `input` multiplied by `input` and return the result.

This example returns the square of an integer number as shown in figure 15.2.

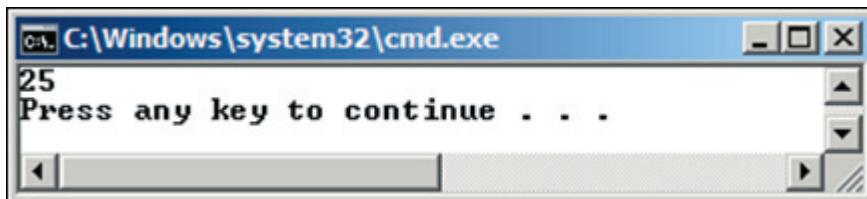


Figure 15.2: Square of an Integer

15.2.1 Expression Lambdas

An expression lambda is a lambda with an expression on the right side. It has the following syntax:

Syntax:

`(input_parameters) => expression`

where,

`input_parameters`: one or more input parameters, each separated by a comma

`expression`: the expression to be evaluated

The input parameters may be implicitly typed or explicitly typed.

When there are two or more input parameters on the left side of the lambda operator, they must be enclosed within parentheses. If there is no parameter at all, a pair of empty parentheses must be used. However, if you have only one input parameter and its type is implicitly known, then the parentheses can be omitted.

Consider the lambda expression,

```
(str, str1) => str == str1
```

It means **str** and **str1** go into the comparison expression which compares **str** with **str1**. In simple terms, it means that the parameters **str** and **str1** will be passed to the expression **str == str1**.

Here, it is not clear what are the types of **str** and **str1**. Hence, it is best to explicitly mention their data types:

```
(string str, string str1)=> str==str1
```

To use a lambda expression, declare a delegate type which is compatible with the lambda expression. Then, create an instance of the delegate and assign the lambda expression to it. After this, you will invoke the delegate instance with parameters, if any. This will result in the lambda expression being executed. The value of the expression will be the result returned by the lambda.

Code Snippet 3 demonstrates expression lambdas.

Code Snippet 3:

```
/// <summary>
/// Class ConvertString converts a given string to uppercase
/// </summary>
public class ConvertString
{
    delegate string MakeUpper(string s);
    public static void Main()
    {
        // Assign a lambda expression to the delegate instance
        MakeUpper con = word => word.ToUpper();
        // Invoke the delegate in Console.WriteLine with a string parameter
        Console.WriteLine(con("abc"));
    }
}
```

In Code Snippet 3, a delegate named **MakeUpper** is created and instantiated. At the time of instantiation, a lambda expression, **word => word.ToUpper()** is assigned to the delegate instance. The meaning of this lambda expression is that, given an input, **word**, call the **ToUpper()** method on it. **ToUpper()** is a built-in method of **String** class and converts a given string into uppercase.

Figure 15.3 displays the output of using expression lambdas.

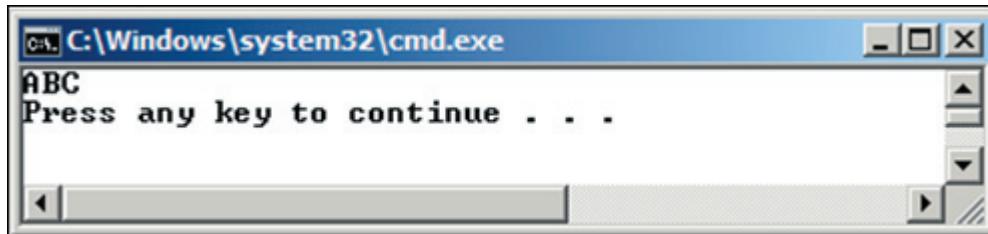


Figure 15.3: Expression Lambdas

Note - When you assign an expression lambda to a delegate instance, the lambda expression must be compatible with the delegate instance.

15.2.2 Statement Lambdas

A statement lambda is a lambda with one or more statements. It can include loops, if statements, and so forth.

```
(input_parameters) => { statement; }
```

where,

input_parameters: one or more input parameters, each separated by a comma
statement: a statement body containing one or more statements

Optionally, you can specify a return statement to get the result of a lambda.

```
(string str, string str1)=> { return (str==str1); }
```

Code Snippet 4 demonstrates a statement lambda expression.

Code Snippet 4:

```
/// <summary>
/// Class WordLength determines the length of a given word or phrase
/// </summary>
public class WordLength
{
    // Declare a delegate that has no return value but accepts a string
    delegate void GetLength(string s);
    public static void Main()
    {
        // Here, the body of the lambda comprises two entire statements
        GetLength len = name => { int n = name.Length; Console.WriteLine(
            n.ToString()); };
    }
}
```

```

        name.Length; Console.WriteLine(n.ToString()); } ;

    // Invoke the delegate with a string

    len("Mississippi");
}

}

```

Figure 15.4 displays the statement lambda.

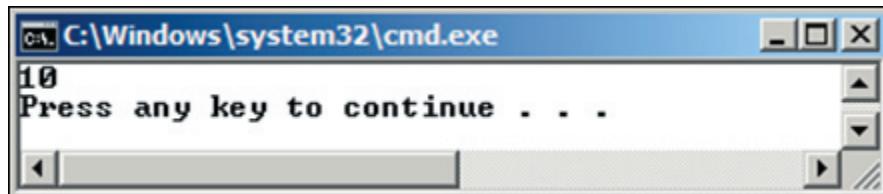


Figure 15.4: Statement Lambda

15.2.3 Lambdas with Standard Query Operators

Lambda expressions can also be used with standard query operators.

Table 15.1 lists the standard query operators.

Operator	Description
Sum	Calculates sum of the elements in the expression
Count	Counts the number of elements in the expression
OrderBy	Sorts the elements in the expression
Contains	Determines if a given value is present in the expression

Table 15.1: Standard Query Operators

Code Snippet 5 shows how to use the `OrderBy` operator with the lambda operator to sort a list of names.

Code Snippet 5:

```

/// <summary>
/// Class NameSort sorts a list of names
/// </summary>
public class NameSort
{
    public static void Main()
    {
        // Declare and initialize an array of strings

```

```

string [ ] names = { "Hanna", "Jim", "Peter", "Karl", "Abby", "Benjamin" };
foreach (string n in names.OrderBy(name => name) )
{
    Console.WriteLine(n);
}
}
}

```

Code Snippet 5 sorts the list of names in the string array alphabetically and then displays them one by one. It makes use of the lambda operator along with the standard query operator `OrderBy`.

Figure 15.5 displays the output of the `OrderBy` operator example.

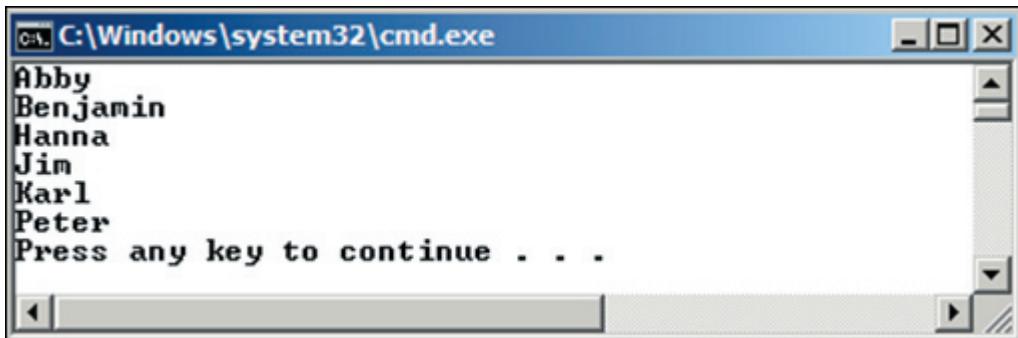


Figure 15.5: OrderBy Operator Example

15.3 Query Expressions

A query is a set of instructions that retrieves data from a data source. The source may be a database table, an ADO.NET dataset table, an XML file, or even a collection of objects such as a list of strings. A query expression is a query that is written in query syntax using clauses like `from`, `select`, and so forth. These clauses are an inherent part of a LINQ query. LINQ is a set of technologies introduced in Visual Studio 2008. It simplifies working with data present in various formats in different data sources. LINQ provides a consistent model to work with such data.

Through LINQ, developers can now work with queries as part of the C# language. Developers can create and use query expressions, which are used to query and transform data from a data source supported by LINQ. A `from` clause must be used to start a query expression and a `select` or `group` clause must be used to end the query expression.

Code Snippet 6 shows a simple example of a query expression. Here, a collection of strings representing names is created and then, a query expression is constructed to retrieve only those names that end with 'l'.

Code Snippet 6:

```
class Program
{
    static void Main(string[] args)
    {
        string[] names = { "Hanna", "Jim", "Pearl", "Mel", "Jill", "Peter",
                           "Karl", "Abby", "Benjamin" };
        IEnumerable<string> words = from word in names
                                     where word.EndsWith("l")
                                     select word;
        foreach (string s in words)
        {
            Console.WriteLine(s);
        }
    }
}
```

Figure 15.6 displays the output of the query expression example.

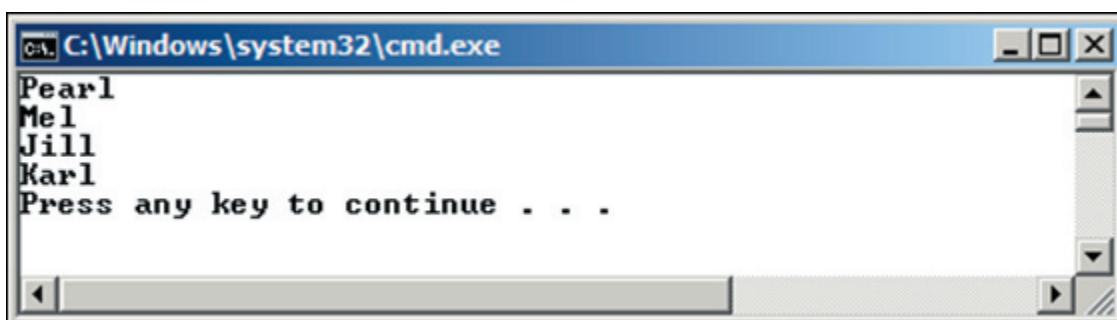


Figure 15.6: Output of the Query Expression Example

Whenever a compiler encounters a query expression, internally it converts it into a method call, using extension methods.

So, an expression such as the one shown in Code Snippet 6 is converted appropriately.

```
IEnumerable<string> words = from word in names
    where word.EndsWith("l")
    select word;
```

After conversion:

```
IEnumerable<string> words = names.Where(word => word.EndsWith("l"));
```

Though this line is more compact, the SQL way of writing the query expression is more readable and easier to understand.

Some of the commonly used query keywords seen in query expressions are listed in table 15.2.

Clause	Description
from	Used to indicate a data source and a range variable
where	Used to filter source elements based on one or more boolean expressions that may be separated by the operators && or
select	Used to indicates how the elements in the returned sequence will look like when the query is executed
group	Used to group query results based on a specified key value
orderby	Used to sort query results in ascending or descending order
ascending	Used in an orderby clause to represent ascending order of sort
descending	Used in an orderby clause to represent descending order of sort

Table 15.2: Query Keywords Used in Query Expressions

15.4 Accessing Databases Using the Entity Framework

Most of the C# applications need to persist and retrieve data that might be stored in some data source, such as a relational database, XML file, or spreadsheet. In an application, data is usually represented in the form of classes and objects. However, in a database, data is stored in the form of tables and views. Therefore, an application in order to persist and retrieve data first needs to connect with the data store. The application must also ensure that the definitions and relationships of classes or objects are mapped with the database tables and table relationships. Finally, the application needs to provide the data access code to persist and retrieve data. All these operations can be achieved using ADO.NET, which is a set of libraries that allows an application to interact with data sources. However, in enterprise data-centric applications, using ADO.NET results in development complexity, which subsequently increases development time and cost. In addition, as the data access code of an application increases, the application becomes difficult to maintain and often leads to performance overhead.

To address data access requirements of enterprise applications, Object Relationship Mapping (ORM) frameworks have been introduced. An ORM framework simplifies the process of accessing data from applications. An ORM framework performs the necessary conversions between incompatible type systems in relational databases and object-oriented programming languages.

The Entity Framework is an ORM framework that .NET applications can use.

15.4.1 The Entity Data Model

The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application. EDM allows a programmer to handle data access logic by programming against entities without having to worry about the structure of the underlying data store and how to connect with it. For example, in an order placing operation of a customer relationship management application, a programmer using the EDM can work with the **Customer** and **Order** entities in an object-oriented manner without writing database connectivity code or SQL-based data access code.

Figure 15.7 shows the role of EDM in the Entity Framework architecture.

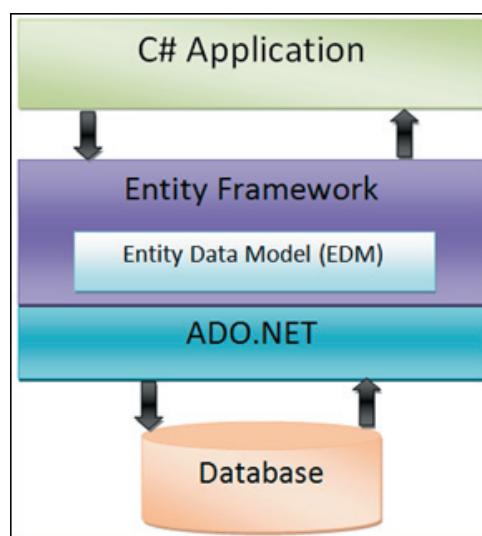


Figure 15.7: The Entity Framework Architecture

15.4.2 Development Approaches

Entity Framework eliminates the need to write most of the data-access code that otherwise need to be written. It uses different approaches to manage data related to an application. These approaches are as follows:

→ **The database-first approach**

The Entity Framework creates the data model containing all the classes and properties corresponding to the existing database objects, such as tables and columns.

→ **The model-first approach**

The Entity Framework creates database objects based on the model that a programmer creates to represent the entities and their relationships in the application.

→ **The code-first approach**

The Entity Framework creates database objects based on custom classes that a programmer creates to represent the entities and their relationships in the application.

15.4.3 Creating an Entity Data Model

Visual Studio 2012 provides support for creating and using EDM in C# application. Programmers can use the Entity Data Model wizard to create a model in an application. After creating a model, programmer can add entities to the model and define their relationship using the Entity Framework designer. The information of the model is stored in an .edmx file. Based on the model, programmer can use Visual Studio 2012 to automatically generate the database objects corresponding to the entities. Finally, the programmer can use LINQ queries against the entities to retrieve and update data in the underlying database.

To create an entity data model and generate the database object, a programmer needs to perform the following steps:

1. Open Visual Studio 2012.
2. Create a Console Application project, named **EDMDemo**.
3. Right-click **EDMDemo** in the **Solution Explorer** window, and select **Add→New Item**. The **Add New Item – EDMDemo** dialog box is displayed.
4. Select **Data** from the left menu and then select **ADO.NET Entity Data Model**, as shown in figure 15.8.

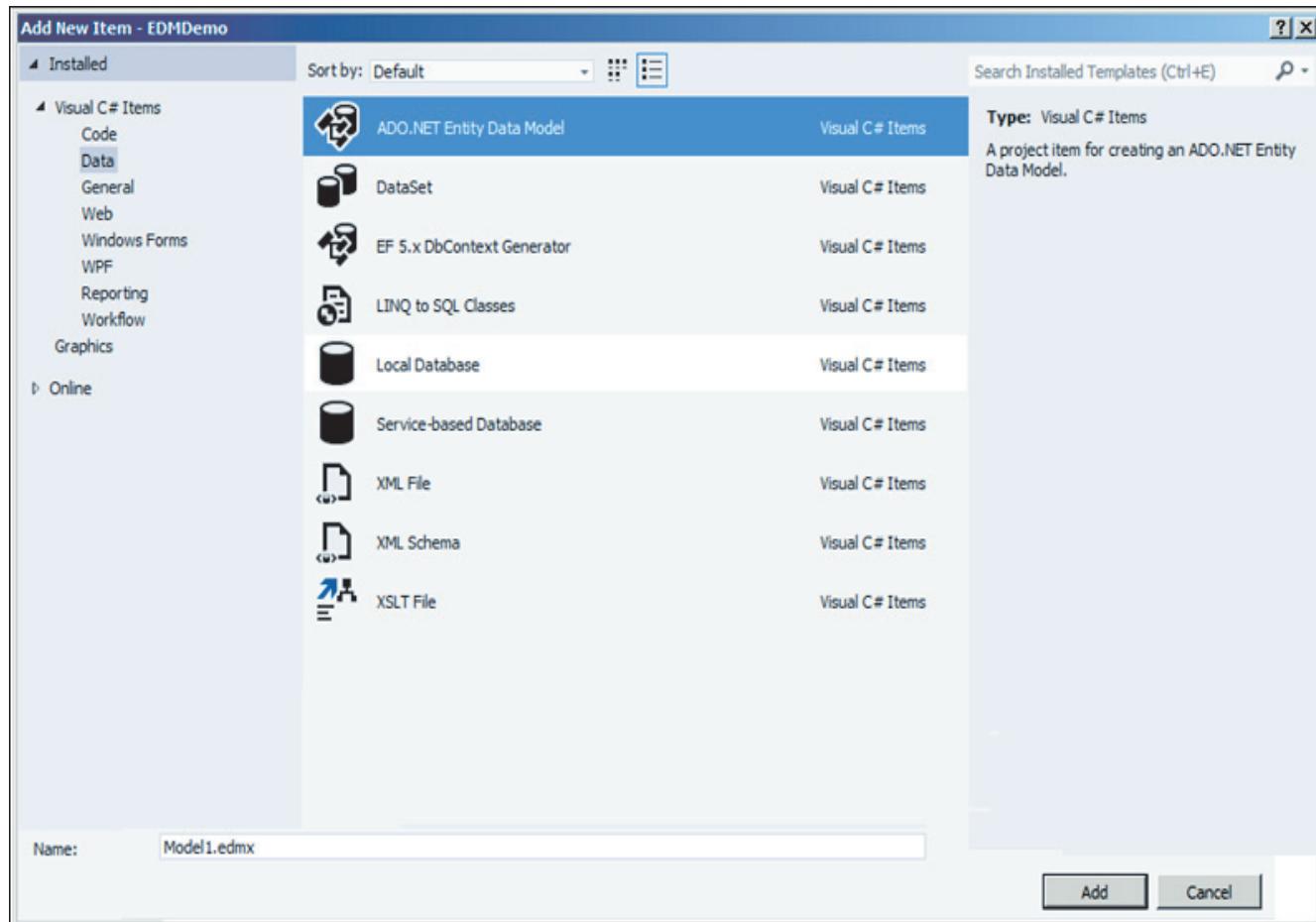


Figure 15.8: The Add New Item – EDMDemo Dialog Box

5. Click **Add**. The **Entity Data Model Wizard** appears.
6. Select **Empty model**.
7. Click **Finish**. The Entity Data Model Designer is displayed.
8. Right-click the Entity Data Model Designer, and select **Add New→Entity**. The **Add Entity** dialog box is displayed.
9. Enter **Customer** in the **Entity name** field and **CustomerId** in the **Property** name field, as shown in figure 15.9.

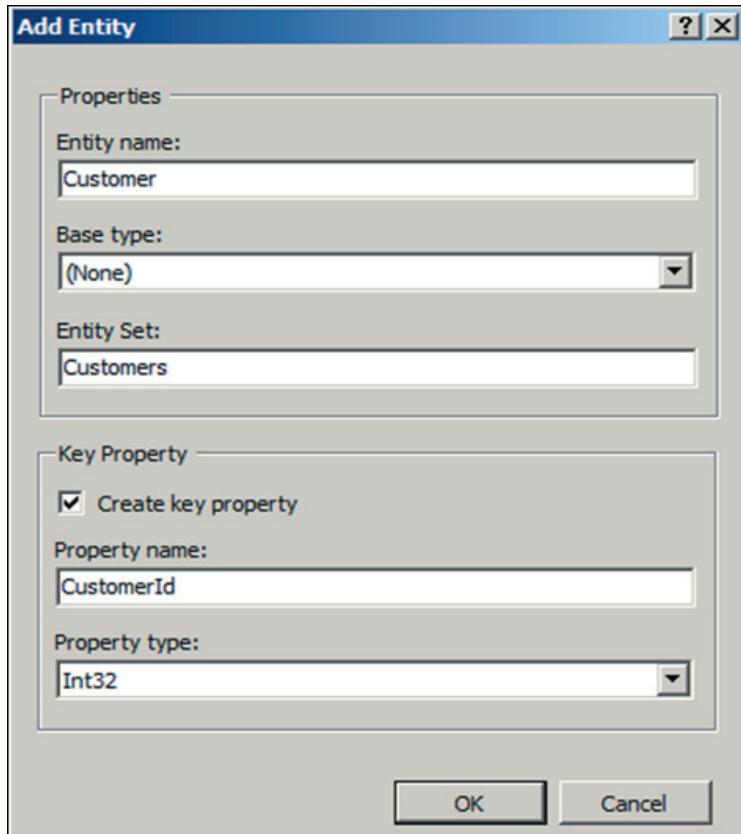


Figure 15.9: The Add Entity Dialog Box

10. Click **OK**. The Entity Data Model Designer displays the new **Customer** entity, as shown in figure 15.10.

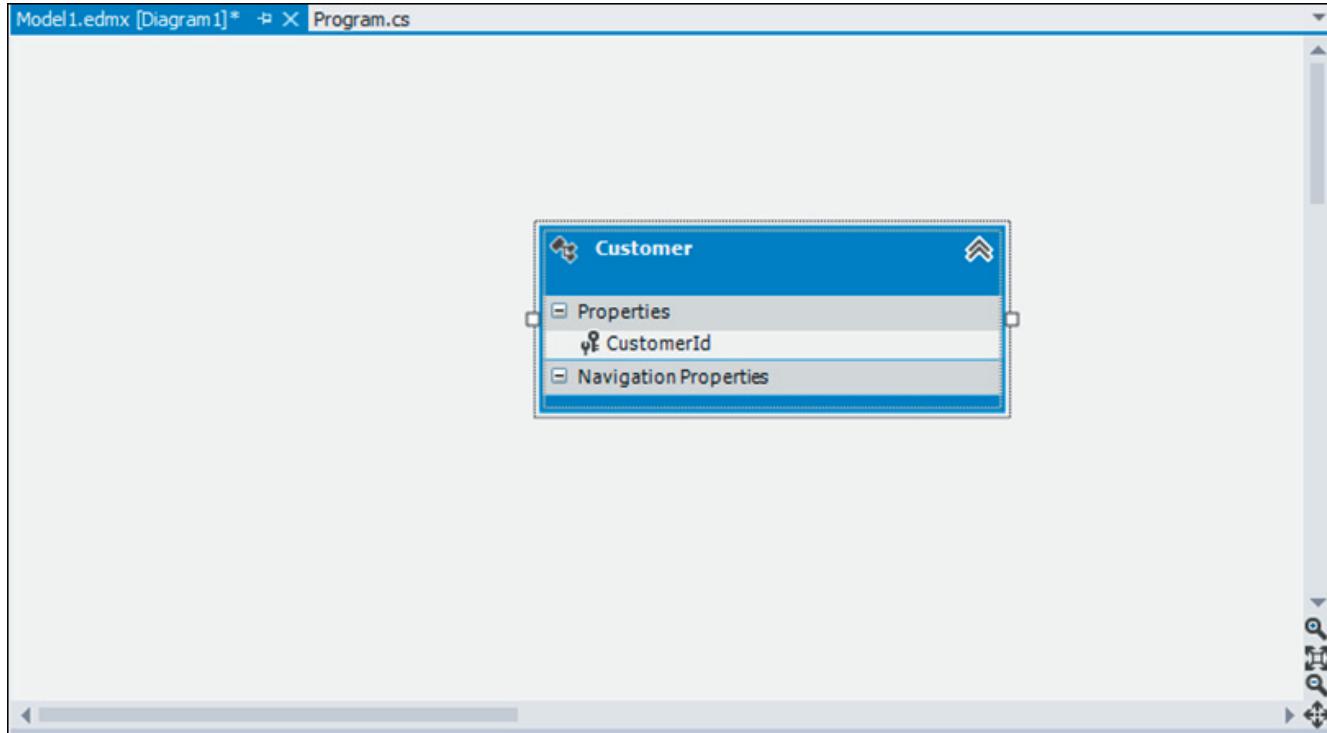


Figure 15.10: The Customer Entity

11. Right-click the **Customer** entity and select **Add New→Scalar property**.
12. Enter **Name** as the name of the property.
13. Similarly, add an **Address** property to the **Customer** entity.
14. Add another entity named **Order** with an **OrderId** key property.
15. Add a **Cost** property to the **Order** entity.

15.4.4 Defining Relationships

After creating an EDM and adding the entities to the EDM, the relationships between the entities can be defined using the Entity Data Model Designer. As a customer can have multiple orders, the **Customer** entity will have a one-to-many relationship with the **Order** entity. To create an association between the **Customer** and **Order** entities:

1. Right-click the Entity Data Model Designer, and select **Add New→Association**. The **Add Association** dialog box is displayed.
2. Ensure that the left-hand **End** section of the relationship point to **Customer** with a multiplicity of **1 (One)** and the right-hand **End** section point to **Post** with a multiplicity of ***(Many)**.

Accept the default setting for the other fields, as shown in figure 15.11.

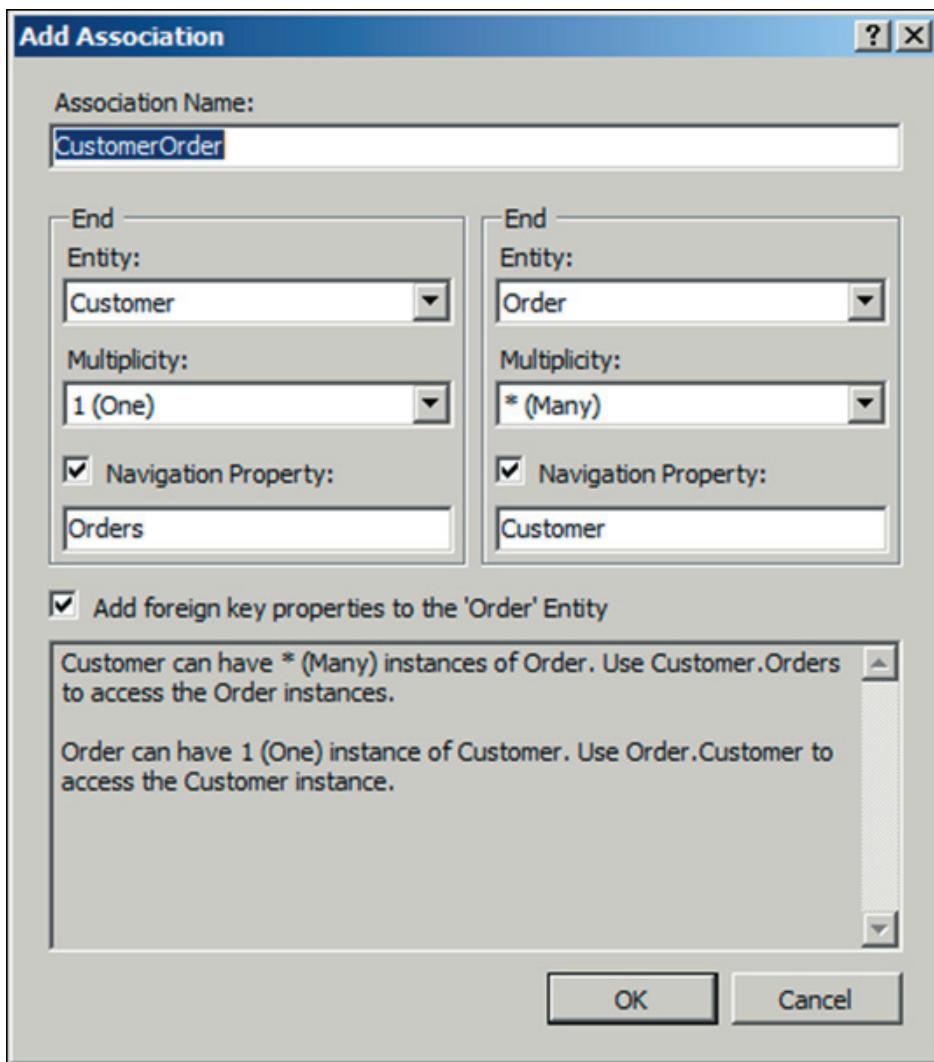


Figure 15.11: Entity Relationship

3. Click **OK**. The Entity Data Model Designer displays the entities with the defined relationship, as shown in figure 15.12.

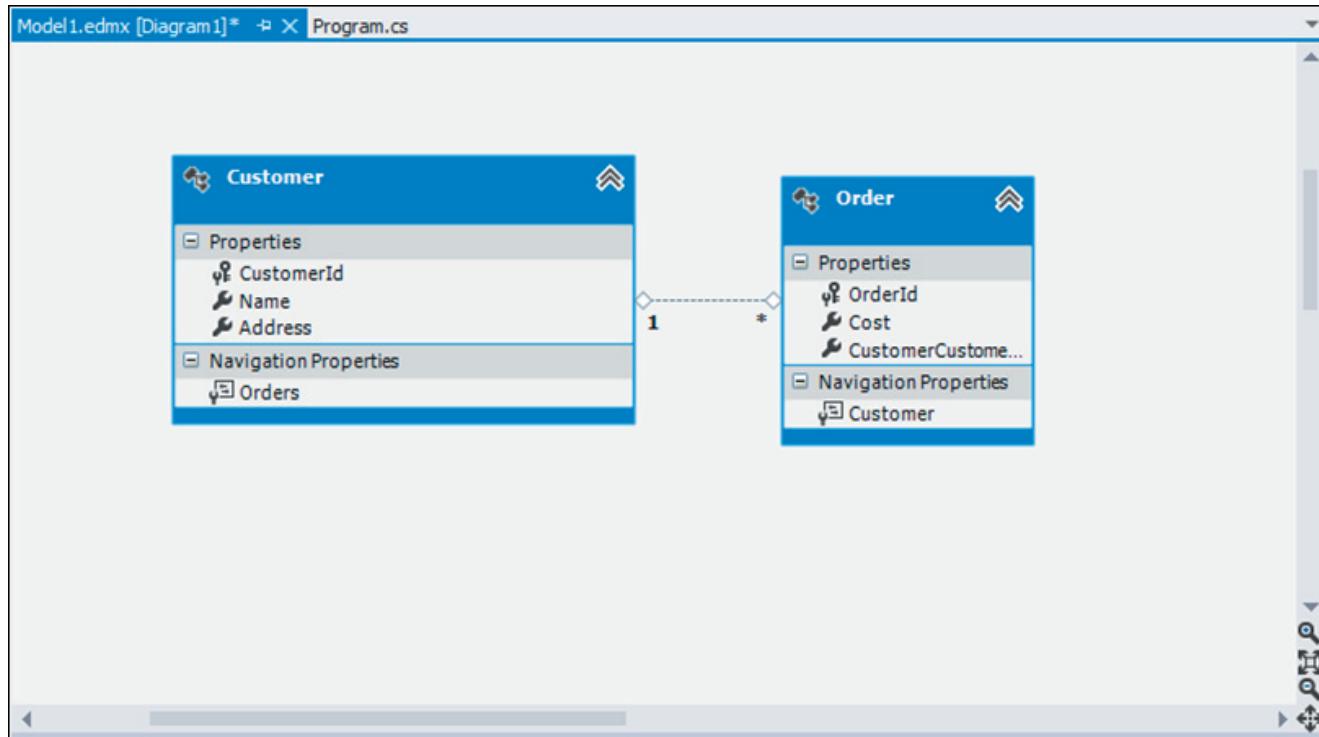


Figure 15.12: Relationship between the Customer and Order Entities

15.4.5 Creating Database Objects

After designing the model of the application, the programmer needs to generate the database objects based on the model. To generate the database objects:

1. Right-click the Entity Data Model Designer, and select **Generate Database from Model**. The **Generate Database Wizard** dialog box appears.
2. Click **New Connection**. The **Connection Properties** dialog box is displayed.

3. Enter `(localdb)\v11.0` in the **Server name** field and `EDMDEMO.CRMDB` in the **Select or enter a database name** field, as shown in figure 15.13.

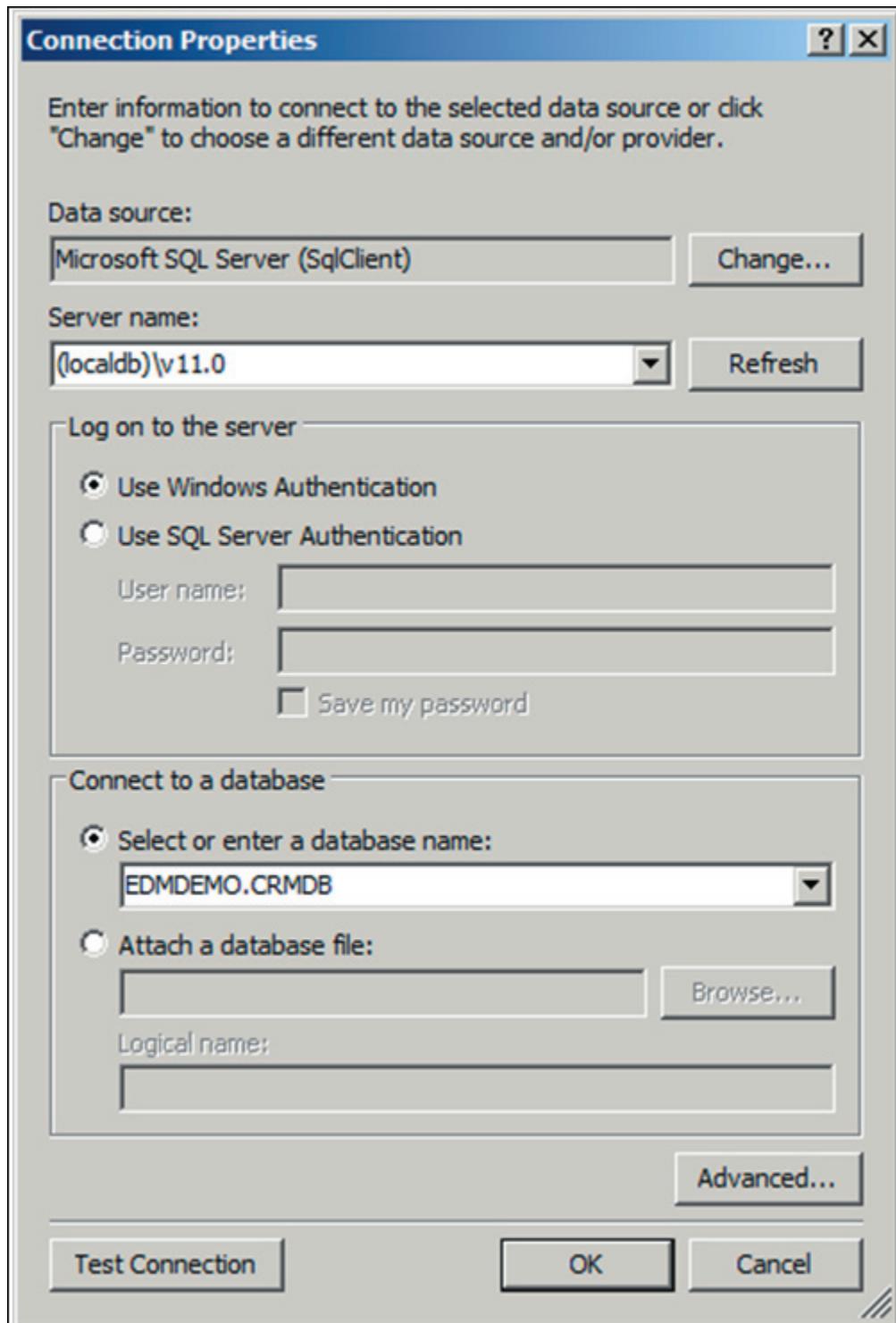


Figure 15.13: The Connection Properties Dialog Box

4. Click **OK**. Visual Studio will prompt whether to create a new database.
5. Click **Yes**.
6. Click **Next** in the **Generate Database Wizard** window. Visual Studio generates the scripts to create the database objects.
7. Click **Finish**. Visual Studio opens the file containing the scripts to create the database objects.
8. Right-click the file and select **Execute**. The **Connect to Server** dialog box is displayed.
9. Click **Connect**. Visual Studio creates the database objects.

15.4.6 Using the EDM

When a programmer uses Visual Studio to create an EDM with entities and their relationships, Visual Studio automatically creates several classes. The important classes that a programmer will use are as follows:

→ **Database Context Class**

This class extends the `DbContext` class of the `System.Data.Entity` namespace to allow a programmer to query and save the data in the database. In the `EDMDemo` Project, the `Model1Container` class present in the `Model1.Context.cs` file is the database context class.

→ **Entity Classes**

These classes represent the entities that programmers add and design in the Entity Data Model Designer. In the `EDMDemo` Project, `Customer` and `Order` are the entity classes.

Code Snippet 7 shows the `Main()` method that creates an persists `Customer` and `Order` entities.

Code Snippet 7:

```
class Program
{
    static void Main(string[] args)
    {
        using (Model1Container dbContext = new Model1Container())
        {
            Console.Write("Enter Customer name: ");
            var name = Console.ReadLine();
            Console.Write("Enter Customer Address: ");
            var address = Console.ReadLine();
```

```
Console.WriteLine("Enter Order Cost:");
var cost = Console.ReadLine();
var customer = new Customer { Name=name, Address=address };
var order = new Order { Cost=cost };
customer.Orders.Add(order);
dbContext.Customers.Add(customer);
dbContext.SaveChanges();
Console.WriteLine("Customer and Order Information added
successfully.");
}
}
```

Code Snippet 7 prompts and accept customer and order information from the console. Then, the **Customer** and **Order** objects are created and initialized with data. The **Order** object is added to the **Orders** property of the **Customer** object. The **Orders** property which is of type, **ICollection<Order>** enables adding multiple **Order** objects to a **Customer** object, based on the one-to-many relationship that exists between the **Customer** and **Order** entities. Then, the database context object of type **Model1Container** is used to add the **Customer** object to the database context. Finally, the call to the **SaveChanges ()** method persists the **Customer** object to the database.

Output:

```
Enter Customer name: Alex Parker
Enter Customer Address: 10th Park Street, Leo Mount
Enter Order Cost:575
Customer and Order Information added successfully.
```

15.4.7 Querying Data by Using LINQ Query Expressions

LINQ provides a consistent programming model to create standard query expression syntax to query different types of data sources. However, different data sources accept queries in different formats. To solve this problem, LINQ provides the various LINQ providers, such as LINQ to Entities, LINQ to SQL, LINQ to Objects, and LINQ to XML. To create and execute queries against the conceptual model of Entity Framework, programmers can use LINQ to Entities.

In LINQ to Entities, a programmer creates a query that returns a collection of zero or more typed entities. To create a query, the programmer needs a data source against which the query will execute. An instance of the `ObjectQuery` class represents the data source. In LINQ to entities, a query is stored in a variable. When the query is executed, it is first converted into a command tree representation that is compatible with the Entity Framework. Then, the Entity Framework executes the query against the data source and returns the result.

Code Snippet 8 creates and executes a query to retrieve the records of all `Customer` entities along with the associated `Order` entities.

Code Snippet 8:

```
public static void DisplayAllCustomers()
{
    using (Model1Container dbContext = new Model1Container())
    {
        IQueryable<Customer> query = from c in
            dbContext.Customers
            select c;

        Console.WriteLine("Customer Order Information:");
        foreach (var cust in query)
        {
            Console.WriteLine("Customer ID: {0}, Name: {1}, Address: {2}",
                cust.CustomerId, cust.Name, cust.Address);
            foreach (var cst in cust.Orders)
            {
                Console.WriteLine("Order ID: {0}, Cost: {1}", cst.OrderId, cst.Cost);
            }
        }
    }
}
```

In Code Snippet 8, the `from` clause specifies the data source from where the data has to be retrieved. `dbContext` is an instance of the data context class that provides access to the `Customers` data source, and `c` is the range variable. When the query is executed, the range variable acts as a reference to each successive element in the data source. The `select` clause in the LINQ query specifies the type of the returned elements as an `IQueryable<Customer>` object. The `foreach` loops iterate through the results of the query returned as an `IQueryable<Customer>` object to print the customer and order details.

Output:

Customer Order Information:

Customer ID: 1, Name: Alex Parker, Address: 10th Park Street, Leo Mount

Order ID: 1, Cost: 575

Customer ID: 2, Name: Peter Milne, Address: Lake View Street, Cheros Mount

Order ID: 2, Cost: 800

In addition to simple data retrieval, programmers can use LINQ to perform various other operations, such as forming projections, filtering data, and sorting data.

→ Forming Projections

When using LINQ queries, the programmer might only need to retrieve specific properties of an entity from the data store; for example only the **Name** property of the **Customer** entity instances. The programmer can achieve this by forming projections in the **select** clause.

Code Snippet 9 shows a LINQ query that retrieves only the customer names of the Customer entity instances.

Code Snippet 9:

```
public static void DisplayCustomerNames ()  
{  
    using (Model1Container dbContext = new Model1Container ())  
    {  
        IQueryable<String> query = from c in dbContext.Customers  
        select c.Name;  
        Console.WriteLine ("Customer Names:");  
        foreach (String custName in query)  
        {  
            Console.WriteLine (custName);  
        }  
    }  
}
```

In Code Snippet 9, the **select** method retrieves a sequence of customer names as an **IQueryable<String>** object. The **foreach** loop iterates through the result to print out the names.

Output:

Customer Names:
Alex Parker
Peter Milne

→ Filtering Data

The `where` clause in a LINQ query enables filtering data based on a Boolean condition, known as the predicate. The `where` clause applies the predicate to the range variable that represents the source elements and returns only those elements for which the predicate is true. Code Snippet 10 uses the `where` clause to filter customer records.

Code Snippet 10:

```
public static void DisplayCustomerByName()
{
    using (Model1Container dbContext = new Model1Container())
    {
        IQueryable<Customer> query = from c in dbContext.Customers
                                         where c.Name == "Alex Parker" select c;
        Console.WriteLine("Customer Information:");
        foreach (Customer cust in query)
        {
            Console.WriteLine("Customer ID: {0}, Name: {1}, Address: {2}",
                cust.CustomerId, cust.Name, cust.Address);
        }
    }
}
```

Code Snippet 10 uses the `where` clause to retrieve information of the customer with the name Alex Parker. The `foreach` statement iterate through the result to print the information of the customer.

Output:

Customer Information:

Customer ID: 1, Name: Alex Parker, Address: 10th Park Street, Leo Mount

15.4.8 Querying Data by Using LINQ Method-Based Queries

The LINQ queries used so far are created using query expression syntax. Such queries are compiled into method calls to the standard query operators, such as `select`, `where`, and `orderby`. Another way to create LINQ queries is by using method-based queries where programmers can directly make method calls to the standard query operator, passing lambda expressions as the parameters.

Code Snippet 11 uses the `Select` method to project the `Name` and `Address` properties of `Customer` entity instances into a sequence of anonymous types.

Code Snippet 11:

```
public static void DisplayPropertiesMethodBasedQuery()
{
    using (Model1Container dbContext = new Model1Container())
    {
        var query = dbContext.Customers.Select(c => new
        {
            CustomerName = c.Name,
            CustomerAddress = c.Address
        });
        Console.WriteLine("Customer Names and Addresses:");
        foreach (var custInfo in query)
        {
            Console.WriteLine("Name: {0}, Address: {1}",
                custInfo.CustomerName, custInfo.CustomerAddress);
        }
    }
}
```

Output:

Customer Names and Addresses:

Name: Alex Parker, Address: 10th Park Street, Leo Mount

Name: Peter Milne, Address: Lake View Street, Cheros Mount

Similarly, you can use the other operators such as `Where`, `GroupBy`, `Max`, and so on through method-based queries.

15.5 Web Services with Windows Communication Framework (WCF)

WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA). SOA is an extension of distributed computing based on the request/response design pattern. SOA allows creating interoperable services that can be accessed from heterogeneous systems. Interoperability enables a service provider to host a service on any hardware or software platform that can be different from the platform on the consumer end.

15.5.1 WCF Fundamentals

Microsoft has several service-oriented distributed technologies, such as ASP.NET Web Services, .NET Remoting, Messaging, and Enterprise Services. Each of these technologies has their own set of infrastructure, standards, and Application Programming Interfaces (APIs) to enable solving different distributed computing requirements. However, there was a need of an interoperable solution that provides the benefits of the distributed technologies in a simple and consistent manner. As a solution, Microsoft introduced WCF services, which is a unified programming model for service-oriented applications.

In WCF, a client communicates with a service using messages. In a WCF communication, a service defines one or more service endpoints, that defines the information required to exchange messages. A service provides the information through service endpoints in the form of metadata. Based on the metadata exposed by a service, the client initiates a communication with the service by sending a message to the service endpoint. The service on receiving a communication message responds to the message.

15.5.2 Service Endpoints

In a WCF service, a service endpoint provides the following information, also known as the ABC's of WCF:

→ **Address**

Specifies the location where messages can be sent.

→ **Binding**

Specifies the communication infrastructure that enables communication of messages. Binding is formed by a stack of components implemented as channels. At the minimum, a binding defines the transport mechanism, such as HTTP or TCP and the encoding, such as text or binary that is being used to communicate messages.

→ **Contract**

Defines a set of messages that can be communicated between a client and a service.

Figure 15.14 shows service endpoints in a WCF Service.

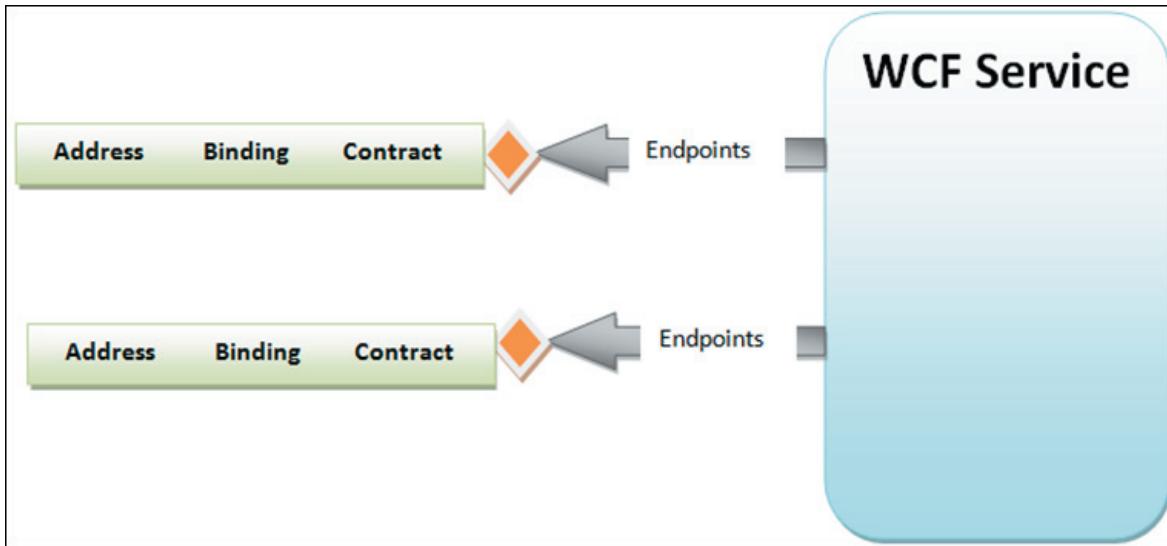


Figure 15.14: Service Endpoints in a WCF Service

15.5.3 WCF Contracts

WCF operations are based on standard contracts that a WCF service provides. The standard contracts are as follows:

- ➔ **Service Contract:** Describes what operations a client can perform on a service.

Code Snippet 12 shows a service contract applied to an `IProductService` interface.

Code Snippet 12:

```
[ServiceContract]
public interface IProductService
{
}
```

Code Snippet 12 uses the `ServiceContract` attribute on the `IProductService` interface to specify that the interface will provide one or more operations that client can invoke using WCF.

→ Operational Contract

Code Snippet 13 shows two operational contracts applied to the `IProductService` interface.

Code Snippet 13:

```
[ServiceContract]  
public interface IProductService  
{  
    [OperationContract]  
    String GetProductName (int productId);  
    [OperationContract]  
    IEnumerable<ProductInformation> GetProductInfo (int productId);  
}
```

Code Snippet 13 uses the `OperationContract` attribute on the `GetProductName()` and `GetProductInfo()` methods to specify that these methods can service WCF clients. In the first operational contract applied to the `GetProductName()` method, no additional data contract is required as both the parameter and return types are primitives. However, the second operational contract returns a complex `ProductInfo` type and therefore, must have a data contract defined for it.

→ Data Contract and Data Members

Specifies how the WCF infrastructure should serialize complex types for transmitting its data between the client and the service. A data contract can be specified by applying the `DataContract` attribute to the complex type, which can be a class, structure, or enumeration. Once a data contact is specified for a type, the `DataMember` attribute must be applied to each member of the type.

Code Snippet 14 shows a data contract applied to the `ProductInformation` class.

Code Snippet 14:

```
[DataContract]  
public class ProductInformation  
{  
    [DataMember]  
    public int ProductId  
    {  
        get; set;  
    }
```

```
[DataMember]
public String ProductName
{
    get; set;
}

[DataMember]
public int ProductPrice
{
    get; set;
}
}
```

Code Snippet 14 applies the `DataContract` attribute to the `ProductInformation` class and the `DataMember` attribute to the `ProductId`, `ProductName`, and `ProductPrice` attributes.

Figure 15.15 shows the relationships between the different contracts in a WCF application.

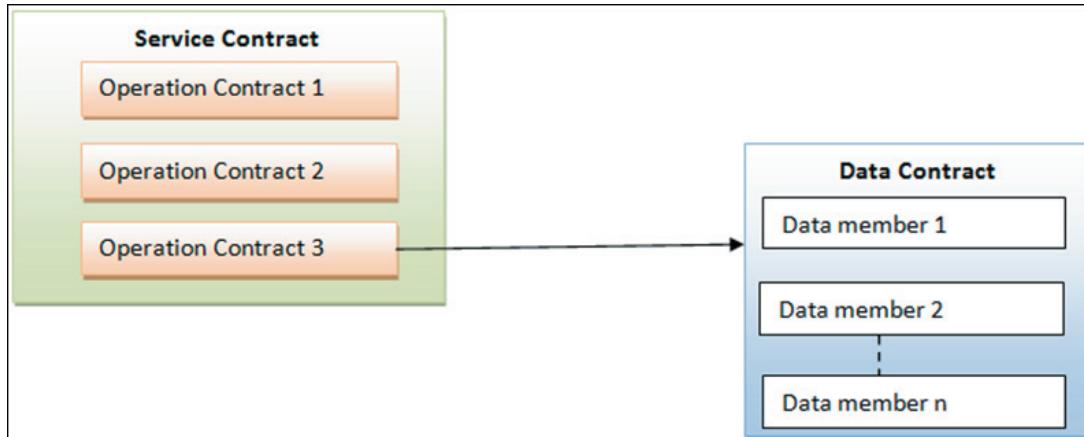


Figure 15.15: Relationships between the Different Contracts

15.5.4 Creating the Implementation Class

After defining the various contracts in a WCF application, the next step is to create the implementation class. This class implements the interface marked with the `ServiceContract` attribute.

Code Snippet 15 shows the `ProductService` class that implements the `IProductService` interface.

Code Snippet 15:

```
public class ProductService : IProductService
{
    List<ProductInformation> products = new
    List<ProductInformation>();
    public ProductService()
    {
        products.Add(new ProductInformation{ ProductId=001,
            ProductName = "Hard Drive", ProductPrice=175 });
        products.Add(new ProductInformation { ProductId=002,
            ProductName = "Keyboard", ProductPrice = 15 });
        products.Add(new ProductInformation { ProductId=003,
            ProductName = "Mouse", ProductPrice = 15 });
    }
    public string GetProductName(int productId)
    {
        IEnumerable<string> Product = from product in products
            where product.ProductId == productId
            select product.ProductName;
        return Product.FirstOrDefault();
    }
    public IEnumerable<ProductInformation> GetProductInfo(int productId)
    {
        IEnumerable<ProductInformation> Product = from product in products
            where product.ProductId == productId
            select product;
        return Product;
    }
}
```

Code Snippet 15 creates the `ProductService` class that implements the `IProductService` interface marked as the service contract. The constructor of the `ProductService` class initializes a `List` object with `ProductInformation` objects. The `ProductService` class implements the `GetProductName()` and `GetProductInfo()` methods marked as operational contract in the `IProductService` interface. The `GetProductName()` method accepts a product ID and performs a LINQ query on the `List` object to retrieve the name and price of the `ProductInformation` object. The `GetProductInfo()` method accepts a product ID and performs a LINQ query on the `List` object to retrieve a `ProductInfo` object in an `IEnumerable` object.

15.5.5 Creating a Client to Access a WCF Service

After creating a WCF service, a client application can access the service using a proxy instance. To use the proxy instance and access the service, the client application requires a reference to the service. In the Visual Studio 2012 IDE, you need to perform the following steps to add a service reference to a client project:

1. In the **Solution Explorer**, right-click the **Service References** node under the project node and select **Add References**. The **Add Service Reference** dialog box is displayed.
2. Click **Discover**. The **Add Service Reference** dialog box displays the hosted WCF service, as shown in figure 15.16.

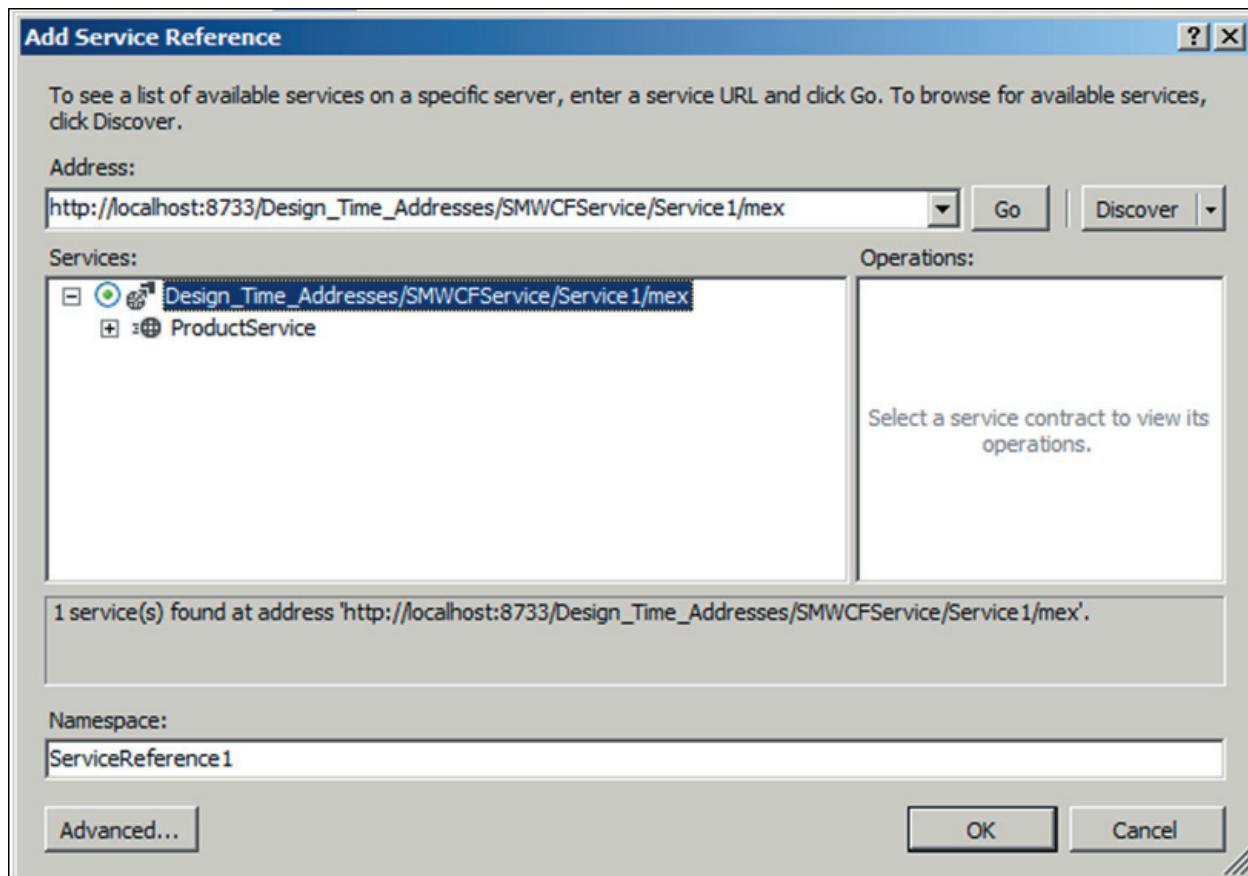


Figure 15.16: The Add Service Reference Dialog Box

- Click **OK**. A reference to the WCF service is added to the project.

Code Snippet 16 shows the `Main()` method of a client that accesses a WCF service using the proxy instance.

Code Snippet 16:

```
ServiceReference1.ProductServiceClient client = new
    ServiceReference1.ProductServiceClient();
Console.WriteLine("Name of product with ID 001");
Console.WriteLine(client.GetProductName(001));
Console.WriteLine("Information of product with ID 002");
ServiceReference1.ProductInformation[] productArr = client.
    GetProductInfo(002);
foreach (ServiceReference1.ProductInformation product in productArr) {
    Console.WriteLine("Product name: {0}, Product price: {1}",
        product.ProductName, product.ProductPrice);
}
Console.ReadLine();
```

Code Snippet 16 creates a proxy instance of a WCF client of type `ServiceReference1.ProductServiceClient`. The proxy instance is then used to invoke the `GetProductName()` and `GetProductInfo()` service methods. The results returned by the service methods are printed to the console.

Figure 15.17 shows the output of the client application.

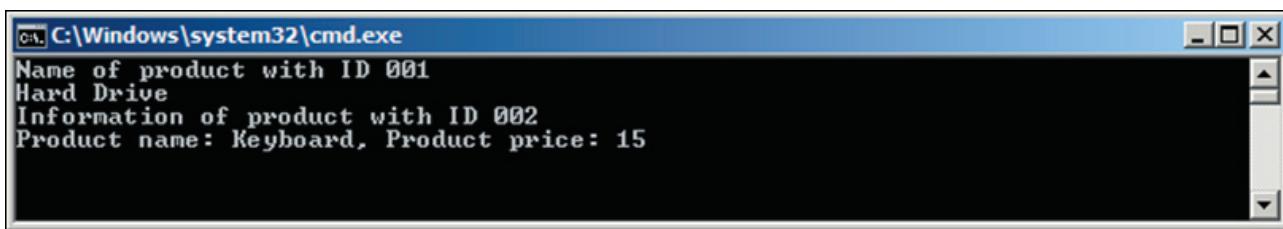


Figure 15.17: Output of the WCF Client

15.6 Multithreading and Asynchronous Programming

C# applications often need to execute multiple tasks concurrently. For example, a C# application that simulates a car racing game needs to gather user inputs to navigate and move a car while concurrently moving the other cars of the game towards the finish line. Such applications, known as multi-threaded applications use multiple threads to execute multiple parts of code concurrently. In the context of programming language, a thread is a flow of control within an executing application. An application will have at least one thread known as the main thread that executes the application. A programmer can create multiple threads spawning the main thread to concurrently process tasks of the application.

A programmer can use the various classes and interfaces in the `System.Threading` namespace that provides built-in support for multithreaded programming in the .NET Framework.

15.6.1 The Thread Class

The `Thread` class of the `System.Threading` namespace allows programmers to create and control a thread in a multithreaded application. Each thread in an application passes through different states that are represented by the members of the `ThreadState` enumeration. A new thread can be instantiated by passing a `ThreadStart` delegate to the constructor of the `Thread` class. The `ThreadStart` delegate represents the method that the new thread will execute. Once a thread is instantiated, it can be started by making a call to the `Start()` method of the `Thread` class.

Code Snippet 17 instantiates and starts a new thread.

Code Snippet 17:

```
class ThreadDemo
{
    public static void Print()
    {
        while (true)
        {
            Console.Write("1");
        }
    }

    static void Main (string [] args)
    {
        Thread newThread = new Thread (new ThreadStart (Print));
        newThread.Start();
        while (true)
        {
            Console.Write("2");
        }
    }
}
```

In Code Snippet 17, the `Print()` method uses an infinite while loop to print the value 1 to the console. The `Main()` method instantiates and starts a new thread to execute the `Print()` method. The `Main()` method then uses an infinite while loop to print the value 2 to the console.

In this program, two threads simultaneously executes both the infinite `while` loops, which results in the output shown in figure 15.18.

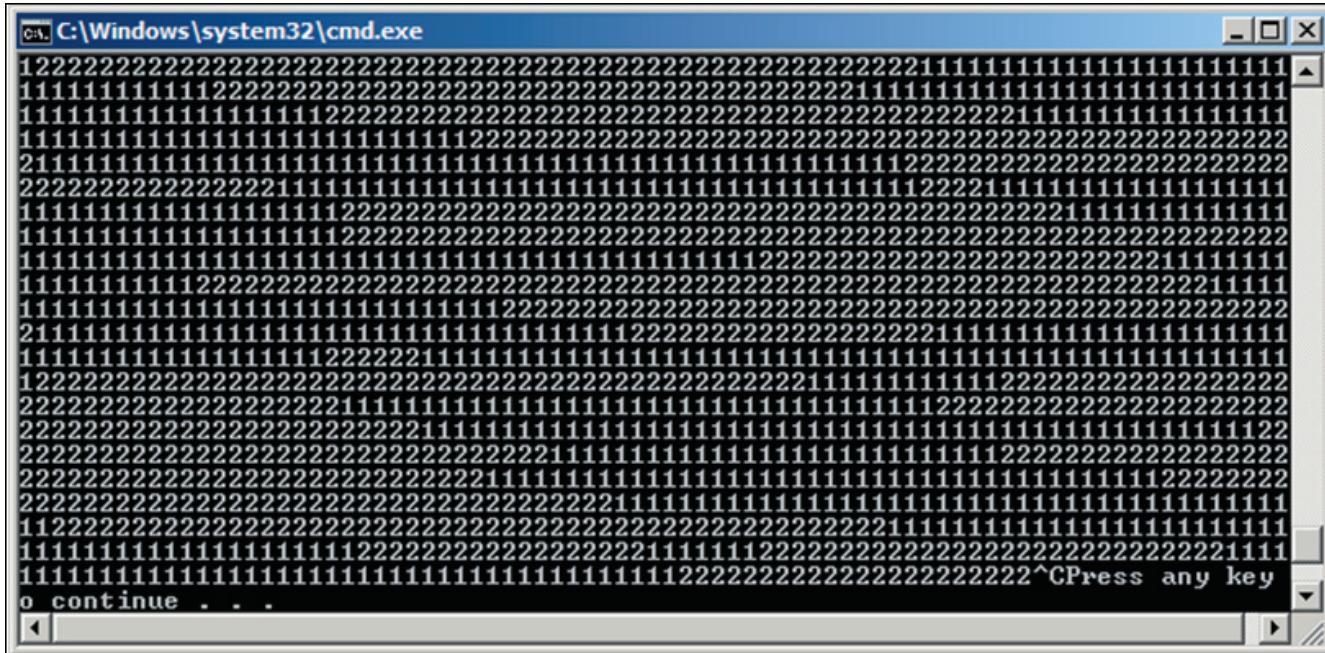


Figure 15.18: Output

15.6.2 The ThreadPool Class

The `System.Threading` namespace provides the `ThreadPool` class to create and share multiple threads as and when required by an application. The `ThreadPool` class represents a thread pool, which is a collection of threads in an application. Based on the request from the application, the thread pool assigns a thread to perform a task. When the thread completes execution, it is put back in the thread pool to be reused for another request.

The `ThreadPool` class contains a `QueueUserWorkItem()` method that a programmer can call to execute a method in a thread from the thread pool. This method accepts a `WaitCallback` delegate that accepts `Object` as its parameter. The `WaitCallback` delegate represents the method that needs to execute in a separate thread of the thread pool.

15.6.3 Synchronizing Threads

When multiple threads need to share data, their activities needs to be coordinated. This ensures that one thread does not change the data used by the other thread to avoid unpredictable results. For example, consider two threads in a C# program. One thread reads a customer record from a file and the other tries to update the customer record at the same time. In this scenario, the thread that is reading the customer record might not get the updated value as the other thread might be updating the record at that instance.

To avoid such situations, C# allows programmers to coordinate and manage the actions of multiple threads at a given time using the following thread synchronization mechanisms.

→ Locking using the `lock` Keyword

Locking is the process that gives control to execute a block of code to one thread at one point of time. The block of code that locking protects is referred to as a critical section.

Locking can be implemented using the lock keyword. When using the lock keyword, the programmer needs to pass an object reference that a thread must acquire to execute the critical section. For example, to lock a section of code within an instance method, the reference to the current object can be passed to the lock.

→ Synchronization Events

The locking mechanism used for synchronizing threads is useful for protecting critical sections of code from concurrent thread access. However, locking does not allow communication between threads. To enable communication between threads while synchronizing them, C# supports synchronization events. A synchronization event is an object that has one of two states: signaled and un-signaled. When a synchronized event is in un-signaled state, threads can be made suspended until the synchronized event comes to the signaled state.

The `AutoResetEvent` class of the `System.Threading` namespace represents a synchronization event that changes automatically from signaled to un-signaled state any time a thread becomes active. The `AutoResetEvent` class provides the `WaitOne()` method that suspends the current thread from executing until the synchronized event comes to the signaled state. The `Set()` method of the `AutoResetEvent` class changes the state of the synchronized event from un-signaled to signaled.

Note - The `System.Threading` namespace also contains a `ManualResetEvent` class that is similar to the `AutoResetEvent` class. It represents a synchronization event. However, unlike the `AutoResetEvent` class, an object of the `ManualResetEvent` class needs to be manually changed from signaled to un-signaled state by calling its `Reset()` method.

15.6.4 Task Parallel Library

Modern computers contain multiple CPUs. In order to take advantage of the processing power that computers with multiple CPUs deliver, a C# application needs to execute tasks in parallel on multiple CPUs. This is known as parallel programming. To make parallel and concurrent programming simpler, the .NET Framework introduced Task Parallel Library (TPL). TPL is a set of public types and APIs in the `System.Threading` and `System.Threading.Tasks` namespaces.

15.6.5 The Task Class

TPL provides the `Task` class in the `System.Threading.Tasks` namespace. It represents an asynchronous task in a program. Programmers can use this class to invoke a method asynchronously. To create a task, the programmer provides a user delegate that encapsulates the code that the task will execute.

The delegate can be a named delegate, such as the `Action` delegate, an anonymous method, or a lambda expression.

After creating a `Task`, the programmer calls the `Start()` method to start the task. This method passes the task to the task scheduler that assigns threads to perform the work. To ensure that a task completes before the main thread exits, a programmer can call the `Wait()` method of the `Task` class. To ensure that all the tasks of a program completes, the programmer can call the `WaitAll()` method passing an array of the `Tasks` objects that have started.

The `Task` class also provides a `Run()` method to create and start a task in a single operation.

Code Snippet 18 creates and starts two tasks.

Code Snippet 18:

```
class TaskDemo
{
    private static void printMessage()
    {
        Console.WriteLine("Executed by a Task");
    }

    static void Main(string[] args)
    {
        Task task1 = new Task(new Action(printMessage));
        task1.Start();
        Task task2 = Task.Run(() => printMessage());
        task1.Wait();
        task2.Wait();
        Console.WriteLine("Exiting main method");
    }
}
```

In Code Snippet 18, the `Main()` method creates a `Task`, named `task1` using an `Action` delegate and passing the name of the method to execute asynchronously. The `Start()` method is called to start the task. The `Run()` method is used to create and start another task, named `task2`. The call to the `Wait()` method ensures that both the tasks complete before the `Main()` method exits.

Output:

Executed by a Task

Executed by a Task

Exiting main method

15.6.6 Obtaining Results from a Task

Often a C# program would require some results after a task completes its operation. To provide results of an asynchronous operation, the .NET Framework provides the `Task<T>` class that derives from the `Task` class. In the `Task<T>` class, `T` is the data type of the result that will be produced. To access the result, call the `Result` property of the `Task<T>` class.

15.6.7 Task Continuation

When multiple tasks execute in parallel, it is common for one task, known as the antecedent to complete an operation and then invoke a second task, known as the continuation task. Such task continuation can be achieved by calling a `ContinueWith()` overloaded methods of the antecedent task. The simplest form of the `ContinueWith()` method accepts a single parameter that represents the task to be executed once the antecedent completes. The `ContinueWith()` method returns the new task. A programmer can call the `Wait()` method on the new task to wait for it to complete.

15.6.8 Task Cancellation

TPL provides the `CancellationTokenSource` class in the `System.Threading` namespace that can be used to cancel a long running task. The `CancellationTokenSource` class has a `Token` property that returns an object of the `CancellationToken` struct. This object propagates notification that a task should be canceled. While creating a task that can be canceled, the `CancellationToken` object needs to be passed to the task.

The `CancellationToken` struct provides an `IsCancellationRequested` property that returns true if a cancellation has been requested. A long running task can query the `IsCancellationRequested` property to check whether a cancellation request is being made, and if so elegantly end the operation. A cancellation request can be made by calling the `Cancel()` method of the `CancellationTokenSource` class.

While canceling a task, a programmer can call the `Register()` method of the `CancellationToken` struct to register a callback method that receives a notification when the task is canceled.

15.6.9 Parallel Loops

TPL introduces a `Parallel` class in the `System.Threading.Tasks` namespace which provides methods to perform parallel computation of loops, such as `for` loops and `for each` loops. The `For()` method is a static method in the `Parallel` class that enables executing a `for` loop with parallel iterations. As the iterations of a loop done using the `For()` method are parallel, the order of iterations might vary each time the `For()` method executes. The `For()` method has several overloaded versions. The most commonly used overloaded `For()` method accepts the following three parameters in the specified order:

1. An `int` value representing the start index of the loop.
2. An `int` value representing the end index of the loop.

3. A `System.Action<Int32>` delegate that is invoked once per iteration.

Code Snippet 19 uses a traditional for loop and the `Parallel.For()` method.

Code Snippet 19:

```
static void Main (string [] args)
{
    Console.WriteLine ("\nUsing traditional for loop");
    for (int i = 0; i <= 10; i++)
    {
        Console.WriteLine ("i = {0} executed by thread with ID {1}", i,
                           Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep (100);
    }
    Console.WriteLine ("\nUsing Parallel For");
    Parallel.For (0, 10, i =>
    {
        Console.WriteLine ("i = {0} executed by thread with ID {1}", i,
                           Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep (100);
    });
}
```

In Code Snippet 19, the `Main()` method first uses a traditional `for` loop to print the identifier of the current thread to the console. The `Sleep()` method is used to pause the main thread for 100 ms for each iteration. As shown in figure 15.19, the `Console.WriteLine()` method prints the results sequentially as a single thread is executing the `for` loop. The `Main()` method then performs the same operation using the `Parallel.For()` method. As shown in figure 15.19, multiple threads indicated by the `Thread.CurrentThread.ManagedThreadId` property executes the `for` loop in parallel and the sequence of iteration is unordered.

Figure 15.19 shows one of the possible outputs of Code Snippet 19.

```
Using traditional for loop
i = 0 executed by thread with ID 1
i = 1 executed by thread with ID 1
i = 2 executed by thread with ID 1
i = 3 executed by thread with ID 1
i = 4 executed by thread with ID 1
i = 5 executed by thread with ID 1
i = 6 executed by thread with ID 1
i = 7 executed by thread with ID 1
i = 8 executed by thread with ID 1
i = 9 executed by thread with ID 1
i = 10 executed by thread with ID 1

Using Parallel For
i = 0 executed by thread with ID 1
i = 5 executed by thread with ID 3
i = 1 executed by thread with ID 4
i = 6 executed by thread with ID 3
i = 2 executed by thread with ID 1
i = 4 executed by thread with ID 4
i = 7 executed by thread with ID 3
i = 8 executed by thread with ID 4
i = 3 executed by thread with ID 1
i = 9 executed by thread with ID 4
Press any key to continue . . .
```

Figure 15.19: Output of Code Snippet 19

15.6.10 Parallel LINQ (PLINQ)

LINQ to Object refers to the use of LINQ queries with enumerable collections, such as `List<T>` or arrays. PLINQ is the parallel implementation of LINQ to Object. While LINQ to Object sequentially accesses an in-memory `IEnumerable` or `IEnumerable<T>` data source, PLINQ attempts parallel access to the data source based on the number of processor in the host computer. For parallel access, PLINQ partitions the data source into segments, and then executes each segment through separate threads in parallel.

The `ParallelEnumerable` class of the `System.Linq` namespace provides methods that implement PLINQ functionality.

Code Snippet 20 shows using both a sequential LINQ to Object and PLINQ to query an array.

Code Snippet 20:

```
string[] arr = new string[] { "Peter", "Sam", "Philip", "Andy", "Philip", "Mary",
"John", "Pamela" };

var query = from string name in arr
            select name;

Console.WriteLine("Names retrieved using sequential LINQ");
foreach (var n in query)
{
}
```

```
Console.WriteLine(n);  
}  
  
var plinqQuery = from string name in arr.AsParallel()  
                  select name;  
  
Console.WriteLine("Names retrieved using PLINQ");  
  
foreach (var n in plinqQuery)  
{  
    Console.WriteLine(n);  
}
```

Code Snippet 20 creates a string array initialized with values. A sequential LINQ query is used to retrieve the values of the array that are printed to the console in a `foreach` loop. The second query is a PLINQ query that uses the `AsParallel()` method in the `from` clause. The PLINQ query also performs the same operations as the sequential LINQ query. However, as the PLINQ query is executed in parallel the order of elements retrieved from the source array is different.

Output:

Names retrieved using sequential LINQ

Peter

Sam

Philip

Andy

Philip

Mary

John

Pamela

Names retrieved using PLINQ

Peter

Philip

Sam

Mary

Philip

John

Andy

Pamela

15.6.11 Concurrent Collections

The collection classes of the `System.Collections.Generic` namespace provides improved type safety and performance compared to the collection classes of the `System.Collections` namespace. However, the collection classes of the `System.Collections.Generic` namespace are not thread safe. As a result, a programmer needs to provide thread synchronization code to ensure integrity of the data stored in the collections. To address thread safety issues in collections, the .NET Framework provides concurrent collection classes in the `System.Collections.Concurrent` namespace. These classes being thread safe relieves programmers from providing thread synchronization code when multiple threads simultaneously accesses these collections. The important classes of the `System.Collections.Concurrent` namespace are as follows:

→ **`ConcurrentDictionary<TKey, TValue>`**

Is a thread-safe implementation of a dictionary of key-value pairs.

→ **`ConcurrentQueue<T>`**

Is a thread-safe implementation of a queue.

→ **`ConcurrentStack<T>`**

Is a thread-safe implementation of a stack.

→ **`ConcurrentBag<T>`**

Is a thread-safe implementation of an unordered collection of elements.

Code Snippet 21 uses multiple threads to add elements to an object of `ConcurrentDictionary<string, int>` class.

Code Snippet 21:

```
class CollectionDemo
{
    static ConcurrentDictionary<string, int> dictionary = new
        ConcurrentDictionary<string, int>();

    static void AddToDictionary()
    {
        for (int i = 0; i < 100; i++)
        {
            dictionary.TryAdd(i.ToString(), i);
        }
    }
}
```

```

static void Main(string[] args)
{
    Thread thread1 = new Thread(new ThreadStart(AddToDictionary));
    Thread thread2 = new Thread(new ThreadStart(AddToDictionary));
    thread1.Start();
    thread2.Start();
    thread1.Join();
    thread2.Join();
    Console.WriteLine("Total elements in dictionary: {0}",
        dictionary.Count());
}
}

```

Code Snippet 21 calls the `TryAdd()` method of the `ConcurrentDictionary` class to concurrently add elements to a `ConcurrentDictionary<string, int>` object using two separate threads. The `TryAdd()` method, unlike the `Add()` method of the `Dictionary` class does not throw an exception if a key already exists. The `TryAdd()` method instead returns false if a key exist and allows the program to exit normally, as shown in figure 15.20.



Figure 15.20: Output of Code Snippet 21

15.6.12 Asynchronous Methods

TPL provides support for asynchronous programming through two new keywords: `async` and `await`. These keywords can be used to asynchronously invoke long running methods in a program. A method marked with the `async` keyword notifies the compiler that the method will contain at least one `await` keyword. If the compiler finds a method marked as `async` but without an `await` keyword, it reports a compilation error.

The `await` keyword is applied to an operation to temporarily stop the execution of the `async` method until the operation completes. In the meantime, control returns to the `async` method's caller. Once the operation marked with `await` completes, execution resumes in the `async` method.

A method marked with the `async` keyword can have either one of the following return types:

- `Void`
- `Task`
- `Task<TResult>`

Note - By convention, a method marked with the `async` keyword ends with an `Async` suffix.

Code Snippet 22 shows the use of the `async` and `await` keywords.

Code Snippet 22:

```
class AsyncAwaitDemo
{
    static async void PerformComputationAsync()
    {
        Console.WriteLine("Entering asynchronous method");
        int result = await new ComplexTask().AnalyzeData();
        Console.WriteLine(result.ToString());
    }

    static void Main(string[] args)
    {
        PerformComputationAsync();
        Console.WriteLine("Main thread executing.");
        Console.ReadLine();
    }
}

class ComplexTask
{
    public Task<int> AnalyzeData()
    {
        Task<int> task = new Task<int>(GetResult);
        task.Start();
        return task;
    }

    private int GetResult()
    {
        // Simulate a long-running computation
        System.Threading.Thread.Sleep(5000);
        return 42;
    }
}
```

```

    }

    public int GetResult()
    {
        /*Pause Thread to simulate time consuming operation*/
        Thread.Sleep(2000);

        return new Random().Next(1, 1000);
    }
}

```

In Code Snippet 22, the `AnalyzeData()` method of the `ComplexTask` class creates and starts a new task to execute the `GetResult()` method. The `GetResult()` method simulates a long running operation by making the thread sleep for two seconds before returning a random number. In the `AsyncAwaitDemo` class, the `PerformComputationAsync()` method is marked with the `async` keyword. This method uses the `await` keyword to wait for the `AnalyzeData()` method to return. While waiting for the `AnalyzeData()` method to return, execution is returned to the calling `Main()` method that prints the message ‘Main thread executing’ to the console. Once the `AnalyzeData()` method returns, execution resumes in the `PerformComputationAsync()` method and the retrieved random number is printed on the console.

Figure 15.21 shows the output.



Figure 15.21: Output of Code Snippet 22

15.7 Dynamic Programming

C# provides dynamic types to support dynamic programming for interoperability of .NET applications with dynamic languages, such as IronPython and Component Object Model (COM) APIs such as the Office Automation APIs.

The C# compiler does not perform static type checking on objects of a dynamic type. The type of a dynamic object is resolved at run-time using the DLR. A programmer using a dynamic type is not required to determine the source of the object’s value during application development. However, any error that escapes compilation checks causes a run-time exception.

To understand how dynamic types bypasses compile type checking, consider Code Snippet 23.

Code Snippet 23:

```
class DemoClass
{
    public void Operation(String name)
    {
        Console.WriteLine("Hello {0}", name);
    }
}

class DynamicDemo
{
    static void Main(string[] args)
    {
        dynamic dynaObj = new DemoClass();
        dynaObj.Operation();
    }
}
```

In Code Snippet 23, the **DemoClass** class has a single **Operation()** method that accepts a **String** parameter. The **Main()** method in the **DynamicDemo** class creates a dynamic type and assigns a **DemoClass** object to it. The dynamic type then calls the **Operation()** method without passing any parameter.

However, the program compiles without any error as the compiler on encountering the `dynamic` keyword does not perform any type checking. However, on executing the program, a run-time exception will be thrown, as shown in figure 15.22.

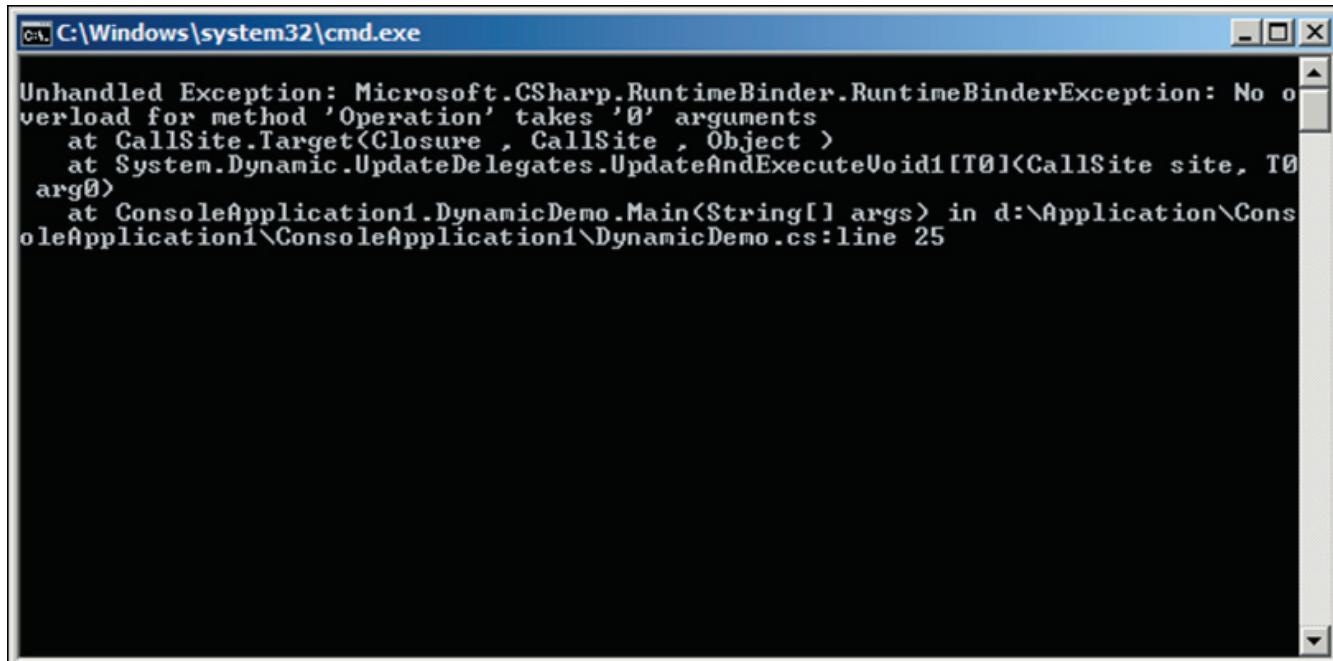


Figure 15.22: Exception Generated at Run-time

The `dynamic` keyword can also be applied to fields, properties, method parameters, and return types. Code Snippet 24 shows how the `dynamic` keyword can be applied to methods to make them reusable in a program.

Code Snippet 24:

```
class DynamicDemo
{
    static dynamic DynaMethod(dynamic param)
    {
        if (param is int)
        {
            Console.WriteLine("Dynamic parameter of type int has value
{0}", param);
            return param;
        }
        elseif (param is string)
        {
```

```
Console.WriteLine("Dynamic parameter of type string has value {0}", param);
    return param;
}
else
{
    Console.WriteLine("Dynamic parameter of unknown type has value {0}", param);
    return param;
}
}

static void Main(string[] args)
{
    dynamic dynaVar1 = DynaMethod(3);
    dynamic dynaVar2 = DynaMethod("HelloWorld");
    dynamic dynaVar3 = DynaMethod(12.5);
    Console.WriteLine("\nReturned dynamic values:\n{0}\n{1}\n{2}", dynaVar1, dynaVar2, dynaVar3);
}
```

In Code Snippet 24, the `DynaMethod()` method accepts a dynamic type as a parameter. Inside the `DynaMethod()` method, the `is` keyword is used in an `if-elseif-else` construct to check for the parameter type and accordingly, returns a value. As the return type of the `DynaMethod()` method is also dynamic, there is no constraint on the type that the method can return. The `Main()` method calls the `DynaMethod()` method with integer, string, and decimal values and prints the return values on the console.

Output:

```
Dynamic parameter of type int has value 3
Dynamic parameter of type string has value HelloWorld
Dynamic parameter of unknown type has value 12.5
Returned dynamic values:
3
HelloWorld
12.5
```

15.8 Check Your Progress

1. Which of the following statements about expression lambdas are true?

(A)	In an expression lambda, the parentheses can be omitted if there are two or more input parameters.
(B)	An expression lambda cannot include a loop statement within it.
(C)	The return type of the expression lambda is the return value of the lambda expression.
(D)	It is mandatory that at least one input parameter must be present in an expression lambda.

(A)	A, B	(C)	C, E
(B)	B, C, E	(D)	D, E

2. Which one of the following delegate syntax corresponds to the statement given here?

It represents a method having two parameters of type T1 and T2 respectively and returns a value of type TResult.

(A)	public delegate TResult Func<T1, TResult>(T1 arg1, T2 arg2)
(B)	public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2)
(C)	public delegate TResult Func<T1, T2>(T1 arg1, T2 arg2)
(D)	public delegate Func<T1, T2, TResult>(T1 arg1, T2 arg2)

(A)	A	(C)	C
(B)	B	(D)	D

3. Match the descriptions given on the right against the clauses given on the left.

Clause		Description	
A.	where	1.	Used to indicate how the elements in the returned sequence will look like when the query is executed
B.	select	2.	Used to filter source elements based on one or more Boolean expressions that may be separated by the operators && or
C.	orderby	3.	Used to sort query results in ascending or descending order
D.	from	4.	Used to indicate a data source and a range variable

(A)	A-2, B-1, C-3, D-4
(B)	A-1, B-3, C-4, D-2
(C)	A-3, B-2, C-1, D-4
(D)	A-4, B-3, C-2, D-1

4. In an Entity Framework application, you are trying to retrieve all **Student** entities stored in a **Students** table sorted by the **FirstName** property. Assuming **dbContext** is a valid database context object, which of the following code will help you achieve this?

(A)	<code>IQueryable<Student> query = from c in dbContext.Students orderby c.FirstName select c;</code>
(B)	<code>IQueryable<Student> query = select c from dbContext.Students orderby c.FirstName;</code>
(C)	<code>IQueryable<Student> query = from c in dbContext.Students select c orderby c.FirstName;</code>
(D)	<code>IQueryable<Students> query = from s in dbContext.Student orderby c.FirstName select c;</code>

(A)	A	(C)	C
(B)	B	(D)	D

5. Regarding the `async` and `await` keywords, which of the following statements are true?

(A)	A class marked with the <code>async</code> keyword notifies the compiler that the class will contain at least one asynchronous method.
(B)	The <code>await</code> keyword when applied to a tasks in a method stops executing the method until the task completes.
(C)	A method marked with the <code>async</code> keyword can either have the <code>Void</code> , <code>Task</code> , or <code>Task<TResult></code> as the return type
(D)	A method marked with the <code>async</code> keyword notifies the compiler that the method will contain at least one <code>await</code> keyword.

(A)	A, D	(C)	C
(B)	B, C	(D)	B, C, and D

15.8.1 Answers

1.	B
2.	B
3.	A
4.	A
5.	D



Summary

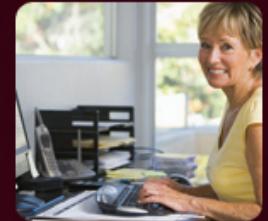
- System-defined generic delegates take a number of parameters of specific types and return values of another type.
- A lambda expression is an inline expression or statement block having a compact syntax and can be used in place of a delegate or anonymous method.
- A query expression is a query that is written in query syntax using clauses like from, select, and so forth.
- The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- Various classes and interfaces in the System.Threading namespace provide built-in support for multithreaded programming in the .NET Framework.
- To make parallel and concurrent programming simpler, the .NET Framework introduced TPL, which is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces.



Need
HELP
on a topic? = **FAQs**



www.onlinevarsity.com



Session - 16

Encrypting and Decrypting Data

Welcome to the Session, **Encrypting and Decrypting Data**.

Encryption and decryption are cryptographic techniques to secure data in storage and in transit. Encryption is the process of transforming data into a secured unreadable format. This data, being in the encrypted form, cannot be read and understood by any user or application without being converted into the original format. Transformation of this encrypted data into the original format is known as decryption.

In this session, you will learn to:

- Explain symmetric encryption
- Explain asymmetric encryption
- List the various types in the System.Security.Cryptography namespace that supports symmetric and asymmetric encryptions

16.1 Introducing Cryptography and Encryption

All organizations need to handle sensitive data. This data can be either present in storage or may be exchanged between different entities within and outside the organization over a network. As an example, consider Steve, who is the CEO of a company. Steve while travelling needs to access performance appraisal reports of the top management of the company. These reports contain data that are confidential and are stored in the company's server. Such data is often prone to misuse either intentionally with malicious intent or unintentionally. To avoid such misuse, there should be some security mechanism that can ensure confidentiality and integrity of data. Cryptography is a security mechanism to ensure data confidentiality and integrity.

One of the commonly used ways to secure sensitive data is through encryption.

16.1.1 Encryption Basics

The primary objective of cryptography is to secure data exchanged between entities, each of which can be a person or an application.

Consider a scenario where User A transmits sensitive data to User B. The transmission needs to be confidential to ensure that even if a third party obtains the data, the data is incomprehensible. In addition, User B before using the data must be sure about the integrity of the data, which means that User B must be sure that the data has not been modified in transit. Also, User B must be able to authenticate that the data is actually been send by User A. On the other hand, after sending the data, User A must not be able to deny sending the data to User B.

In the given scenario, all the necessary security mechanisms for the data communication can be achieved using cryptographic techniques.

Encryption is one such cryptographic technique that ensures data confidentiality. Encryption converts data in plain text to cipher (secretly coded) text. The opposite process of encryption is called decryption, which is a process that converts the encrypted cipher text back to the original plain text. Figure 16.1 shows the process of encryption and decryption of a password as an example.

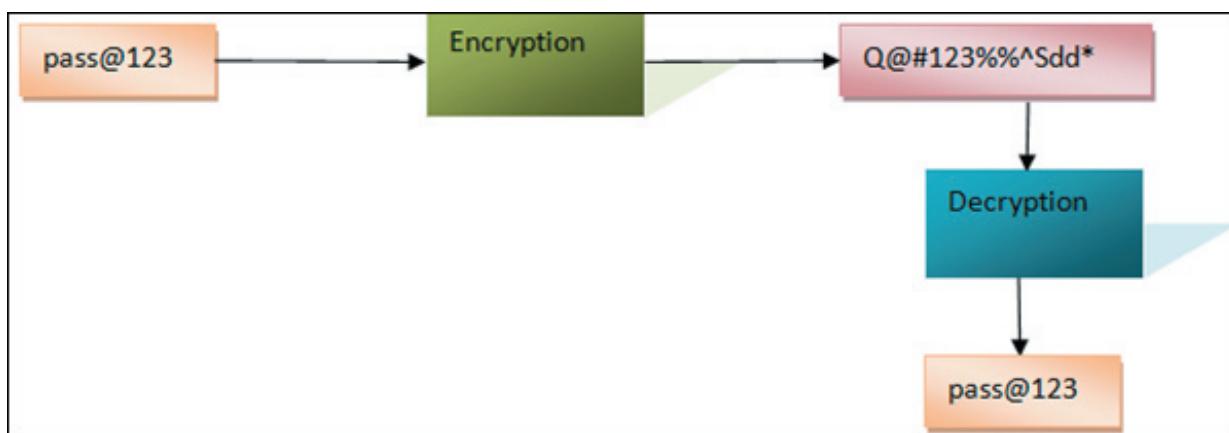


Figure 16.1: Encryption and Decryption

In figure 16.1, the plain text, pass@123 is encrypted to a cipher text. The cipher text is decrypted back to the original plain text.

16.1.2 Symmetric Encryption

Symmetric encryption, or secret key encryption, uses a single key, known as the secret key both to encrypt and decrypt data. The following steps outline an example usage of symmetric encryption:

1. User A uses a secret key to encrypt a plain text to cipher text.
2. User A shares the cipher text and the secret key with User B.
3. User B uses the secret key to decrypt the cipher text back to the original plain text.

Figure 16.2 shows the symmetric encryption and decryption process.

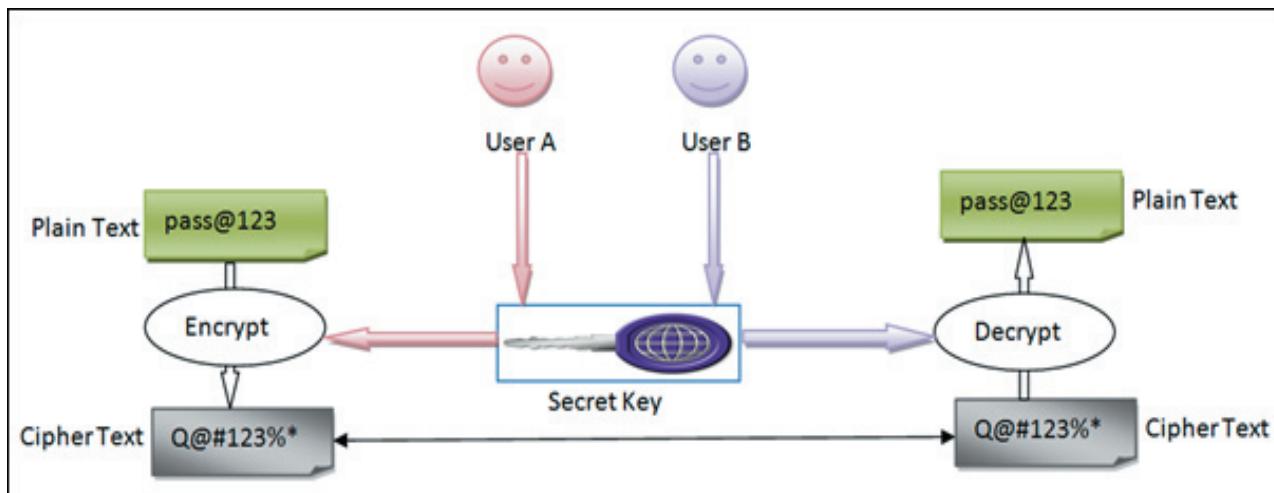


Figure 16.2: Symmetric Encryption and Decryption

Note - To make symmetric encryption more secure, an Initialization Vector (IV) can be used with the secret key. An IV is a random number that generates different sequence of encrypted text for identical sequence of text present in the plain text. When you use an IV with a key to symmetrically encrypt data, you will need the same IV and key to decrypt data.

16.1.3 Asymmetric Encryption

Asymmetric encryption uses a pair of public and private key to encrypt and decrypt data. The following steps outline an example usage of asymmetric encryption:

1. User A generates a public and private key pair.
2. User A shares the public key with User B.
3. User B uses the public key to encrypt a plain text to cipher text.
4. User A uses the private key to decrypt the cipher text back to the original plain text.

Figure 16.3 shows the asymmetric encryption and decryption process.

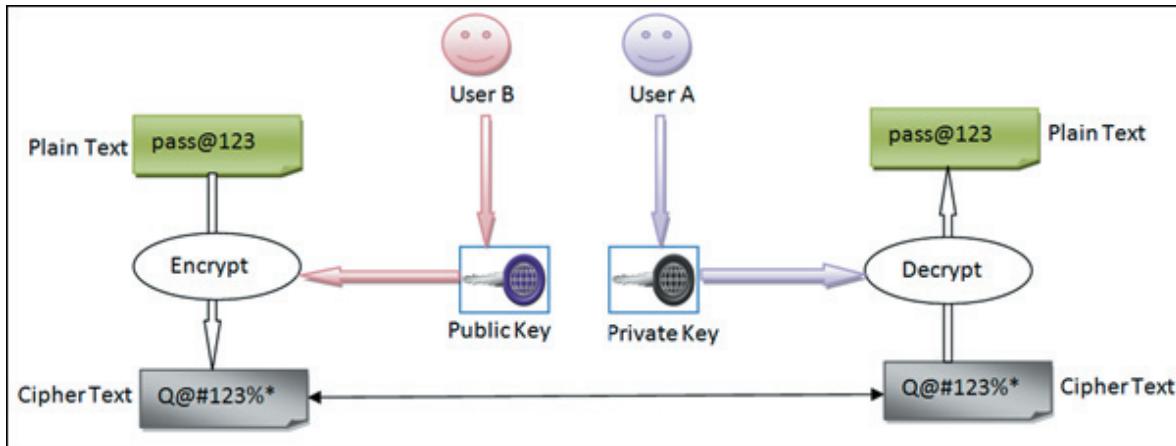


Figure 16.3: Asymmetric Encryption and Decryption

16.2 Encryption Support in the .NET Framework

The .NET Framework provides various types in the `System.Security.Cryptography` namespace to support symmetric and asymmetric encryptions.

16.2.1 Symmetric Encryption Algorithms

The `System.Security.Cryptography` namespace provides the `SymmetricAlgorithm` base class for all symmetric algorithm classes. The derived classes of the `SymmetricAlgorithm` base class are as follows:

→ **RC2**

Is an abstract base class for all classes that implement the RC2 algorithm. This algorithm is a proprietary algorithm developed by RSA Data Security, Inc in 1987. RC2 supports key sizes of from 40 bits to 128 bits in 8-bit increments for encryption. RC2 was designed for the old generation processors and currently have been replaced by more faster and secure algorithms. The `RC2CryptoServiceProvider` class derives from the `RC2` class to provide an implementation of the RC2 algorithm.

→ **DES**

Is an abstract base class for all classes that implement the Data Encryption Standard (DES) algorithm. This algorithm was developed by IBM but as of today available as a U.S. Government Federal Information Processing Standard (FIPS 46-3). The DES algorithm works on the data to encrypt as blocks where each block is of 64 bit. To perform the encryption, DES uses a key of 64 bits. Because of its small key size DES encrypts data faster as compared to other asymmetric algorithms. However, DES is prone to brute force security attacks because of its smaller key size. The `DESCryptoServiceProvider` class derives from the `DES` class to provide an implementation of the DES algorithm.

→ **TripleDES**

Is an abstract base class for all classes that implement the TripleDES algorithm. This algorithm is an enhancement to the DES algorithm for the purpose of making the DES algorithm more secure against security threats. Similar to the DES algorithm, the TripleDES algorithm also works on 64 bit blocks. However, the TripleDES algorithm supports key sizes of 128 bits to 192 bits. The `TripleDESCryptoServiceProvider` class derives from the `TripleDES` class to provide an implementation of the TripleDES algorithm.

→ **Aes**

Is an abstract base class for all classes that implement the Advanced Encryption Standard (AES) algorithm. This algorithm is a successor of DES and is currently considered as one of the most secure algorithm. In addition, AES is more efficient in encrypting large volume of data in the size of several gigabytes. AES works on 128-bits blocks of data and key sizes of 128, 192, or 256 bits for encryption. The `AesCryptoServiceProvider` and `AesManaged` classes derive from the `Aes` class to provide an implementation of the AES algorithm.

→ **Rijndael**

Is an abstract base class for all classes that implement the `Rijndael` algorithm. This algorithm is a superset of the `Aes` algorithm. Similar to the `Aes` algorithm, `Rijndael` supports key sizes of 128, 192, or 256 bits. However, unlike `Aes` which has a fixed block size of 128 bits, `Rijndael` can have block sizes of 128, 192, or 256 bits. By supporting different block sizes, `Rijndael` provides the flexibility to select an appropriate block size based on the volume of data to encrypt. The `RijndaelManaged` class derives from the `Rijndael` class to provide an implementation of the `Rijndael` algorithm.

16.2.2 Asymmetric Encryption Algorithm

The `System.Security.Cryptography` namespace provides the `AsymmetricAlgorithm` base class for all asymmetric algorithm classes. `RSA` is an abstract class that derives from the `AsymmetricAlgorithm` class. The `RSA` class is the base class for all the classes that implement the RSA algorithm.

The RSA algorithm was designed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman and till now is the most widely adopted algorithm to perform asymmetric encryption and decryption. This algorithm functions in three steps: key generation, encryption, and decryption. The RSA algorithm generates a public key as a product of two large prime numbers, along with a public (or encryption) value. The algorithm generates a private key as a product of the same two large prime numbers, along with a private (or decryption) value. The public key is used to perform encryption while the private key is used to perform decryption.

In the .NET Framework, the `RSACryptoServiceProvider` class derives from the `RSA` class to provide an implementation of the RSA algorithm.

16.3 Performing Symmetric Encryption

You need to use one of the symmetric encryption implementation classes of the .NET Framework to perform symmetric encryption.

The first step to perform symmetric encryption is to create the symmetric key.

16.3.1 Generating Symmetric Keys

When you use the default constructor of the symmetric encryption classes, such as `RijndaelManaged` and `AesManaged`, a key and IV are automatically generated. The generated key and the IV can be accessed as byte arrays using the `Key` and `IV` properties of the encryption class.

Code Snippet 1 creates a symmetric key and IV using the `RijndaelManaged` class.

Code Snippet 1:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
...
RijndaelManaged symAlgo = new RijndaelManaged();
Console.WriteLine("Generated key: {0}, \nGenerated IV: {1}", Encoding.Default.GetString(symAlgo.Key), Encoding.Default.GetString(symAlgo.IV));
```

Code Snippet 1 uses the default constructor of the `RijndaelManaged` class to generate a symmetric key and IV. The `Key` and `IV` properties are accessed and printed as strings using the default encoding to the console.

Figure 16.4 shows the output of Code Snippet 1.

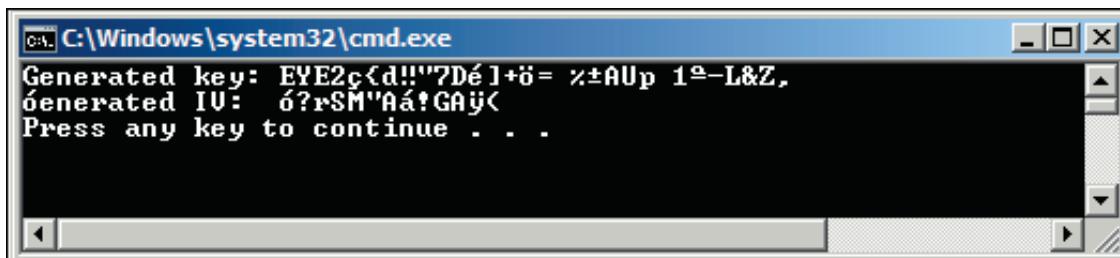


Figure 16.4: Output of Code Snippet 1

The symmetric encryption classes, such as `RijndaelManaged` also provide the `GenerateKey()` and `GenerateIV()` methods that you can use to generate keys and IVs, as shown in Code Snippet 2.

Code Snippet 2:

```
using System;  
using System.Security.Cryptography;  
using System.Text;
```

```

...
.

RijndaelManaged symAlgo = new RijndaelManaged();
symAlgo.GenerateKey();
symAlgo.GenerateIV();
byte[] generatedKey = symAlgo.Key;
byte[] generatedIV = symAlgo.IV;
Console.WriteLine("Generated key through GenerateKey(): {0}, \nGenerated IV
through GenerateIV(): {1}",
Encoding.Default.GetString(generatedKey),
Encoding.Default.GetString(generatedIV));

```

Code Snippet 2 creates a `RijndaelManaged` object and calls the `GenerateKey()` and `GenerateIV()` methods to generate a key and an IV. The Key and the IV properties are then accessed and printed as strings using the default encoding to the console. Figure 16.5 shows the output of Code Snippet 2.

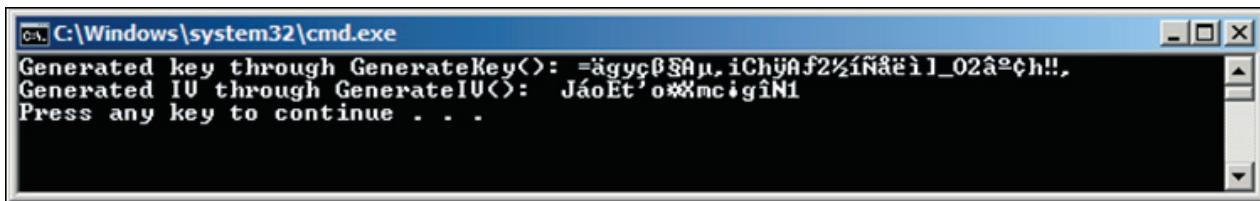


Figure 16.5: Output of Code Snippet 2

16.3.2 Encrypting Data

The symmetric encryption classes of the .NET Framework provides the `CreateEncryptor()` method that returns an object of the `ICryptoTransform` interface. The `ICryptoTransform` object is responsible for transforming the data based on the algorithm of the encryption class. Once you have obtained an `ICryptoTransform` object, you can use the `CryptoStream` class to perform encryption. The `CryptoStream` class acts as a wrapper of a stream-derived class, such as `FileStream`, `MemoryStream`, and `NetworkStream`. A `CryptoStream` object operates in one of the following two modes defined by the `CryptoStreamMode` enumeration:

→ Write

This mode allows write operation on the underlying stream. Use this mode to perform encryption.

→ **Read**

This mode allows read operation on the underlying stream. Use this mode to perform decryption.

You can create a `CryptoStream` object by calling the constructor that accepts the following three parameters:

- The underlying stream object
- The `ICryptoTransform` object
- The mode defined by the `CryptoStreamMode` enumeration

After creating the `CryptoStream` object, you can call the `Write()` method to write the encrypted data to the underlying stream.

Code Snippet 3 encrypts data using the `RijndaelManaged` class and writes the encrypted data to a file.

Code Snippet 3:

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
    static void EncryptData(String plainText, RijndaelManaged algo)
    {
        byte[] plaindataArray=ASCIIEncoding.ASCII.GetBytes(plainText);

        ICryptoTransform transform=algo.CreateEncryptor();
        using (var fileStream=new FileStream("D:\\CipherText.txt", FileMode.
OpenOrCreate, FileAccess.Write))
        {
            using (var cryptoStream=new CryptoStream(fileStream, transform,
CryptoStreamMode.Write))
            {
                cryptoStream.Write(plaindataArray, 0, plaindataArray.GetLength(0));
                Console.WriteLine("Encrypted data written to:");
            }
        }
    }
}
```

```

        D:\\CipherText.txt");
    }
}
}

static void Main()
{
    RijndaelManaged symAlgo = new RijndaelManaged();
    Console.WriteLine("Enter data to encrypt.");
    string dataToEncrypt = Console.ReadLine();
    EncryptData(dataToEncrypt, symAlgo);
}
}
}

```

In Code Snippet 3, the `Main()` method creates a `RijndaelManaged` object and passes it along with the data to encrypt to the `EncryptData()` method. In the `EncryptData()` method, the call to the `CreateEncryptor()` method creates the `ICryptoTransform` object. Then, a `FileStream` object is created to write the encrypted text to the `CipherText.txt` file. Next, the `CryptoStream` object is created and its `Write()` method is called. Figure 16.6 shows the output of Code Snippet 3.

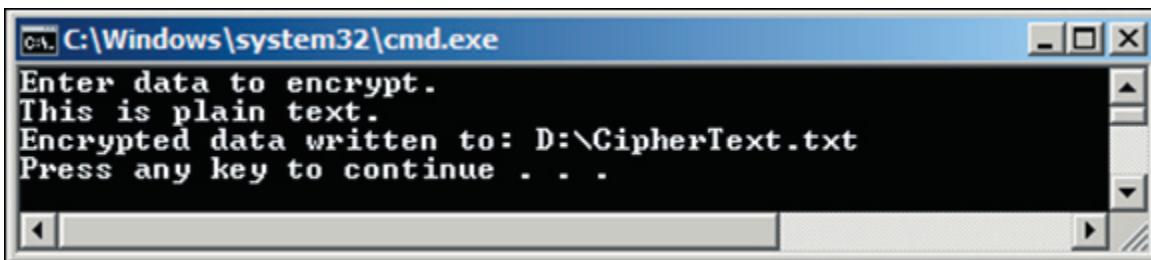


Figure 16.6: Output of Code Snippet 3

16.3.3 Decrypting Data

To decrypt data you need to use the same symmetric encryption class, key, and IV used for encrypting the data. You call the `CreateDecryptor()` method to obtain a `ICryptoTransform` object that will perform the transformation. You then need to create the `CryptoStream` object in Read mode and initialize a `StreamReader` object with the `CryptoStream` object. Finally, you need to call the `ReadToEnd()` method of the `StreamReader` that returns the decrypted text as a string.

Code Snippet 4 creates a program that performs both encryption and decryption.

Code Snippet 4:

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
    static void EncryptData(String plainText, RijndaelManaged algo)
    {
        byte[] plaindataArray = ASCIIEncoding.ASCII.GetBytes(plainText);
        ICryptoTransform transform = algo.CreateEncryptor();
        using (var fileStream = new FileStream("D:\\CipherText.txt", FileMode.
OpenOrCreate, FileAccess.Write))
        {
            using (var cryptoStream = new CryptoStream(fileStream, transform,
CryptoStreamMode.Write))
            {
                cryptoStream.Write(plaindataArray, 0, plaindataArray.GetLength(0));
                Console.WriteLine("Encrypted data written to: D:\\CipherText.txt");
            }
        }
    }
    static void DecryptData(RijndaelManaged algo)
    {
        ICryptoTransform transform = algo.CreateDecryptor();
        using (var fileStream = new FileStream("D:\\CipherText.txt", FileMode.
Open, FileAccess.Read))
        {
            using (CryptoStream cryptoStream = new CryptoStream(fileStream,
transform, CryptoStreamMode.Read))
            {

```

```

using (var streamReader = new StreamReader(cryptoStream))
{
    string decryptedData = streamReader.ReadToEnd();
    Console.WriteLine("Decrypted data: \n{0}", decryptedData);
}
}

static void Main()
{
    RijndaelManaged symAlgo = new RijndaelManaged();
    Console.WriteLine("Enter data to encrypt.");
    string dataToEncrypt = Console.ReadLine();
    EncryptData(dataToEncrypt, symAlgo);
    DecryptData(symAlgo);
}
}

```

In Code Snippet 4, the `Main()` method creates a `RijndaelManaged` object and passes it along with the data to encrypt the `EncryptData()` method. The encrypted data is saved to the `CipherText.txt` file. Next, the `Main()` method calls the `DecryptData()` method passing the same `RijndaelManaged` object created for encryption. The `DecryptData()` method creates the `ICryptoTransform` object and uses a `FileStream` object to read the encrypted data from the file. Then, the `CryptoStream` object is created in the `Read` mode initialized with the `FileStream` and `ICryptoTransform` objects. Next, a `StreamReader` object is created by passing the `CryptoStream` object to the constructor. Finally, the `ReadToEnd()` method of the `StreamReader` object is called. The decrypted text returned by the `ReadToEnd()` method is printed to the console, as shown in figure 16.7.

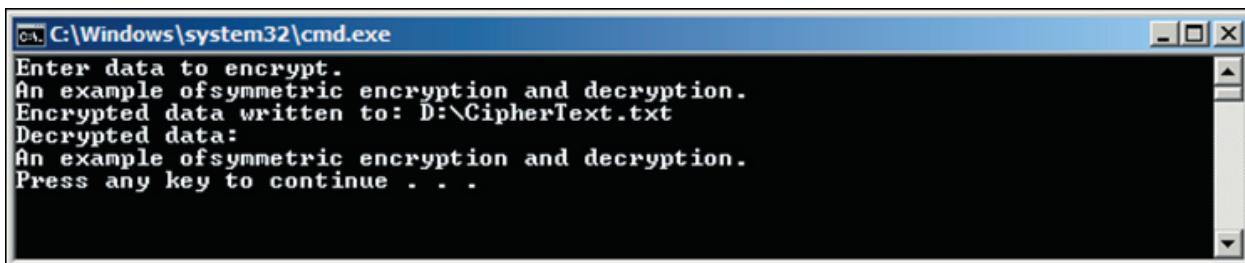


Figure 16.7: Output of Code Snippet 4

16.4 Performing Asymmetric Encryption

You can use the `RSACryptoServiceProvider` class of the `System.Security.Cryptography` namespace to perform asymmetric encryption.

16.4.1 Generating Asymmetric Keys

When you call the default constructor of the `RSACryptoServiceProvider` class, a new public/private key pair is automatically generated. After you create a new instance of the class, you can export the key information by using one of the following methods:

→ `ToXMLString()`

Returns an XML representation of the key information

→ `ExportParameters()`

Returns an `RSAParameters` structure that holds the key information

Both the `ToXMLString()` and `ExportParameters()` methods accept a Boolean value. A `false` value indicates that the method should return only the public key information while a `true` value indicates that the method should return information of both the public and private keys.

Code Snippet 5 shows how to create and initialize an `RSACryptoServiceProvider` object and then export the public key in XML format.

Code Snippet 5:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new
RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
```

Code Snippet 6 shows how to create and initialize an `RSACryptoServiceProvider` object and then export both the public and private keys as an `RSAParameters` structure.

Code Snippet 6:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
```

```
RSACryptoServiceProvider rSAKeyGenerator = new  
RSACryptoServiceProvider();  
  
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
```

16.4.2 Key Containers

Private keys used to decrypt data in asymmetric encryption should be stored using a secured mechanism. To achieve this, you can use a key container that is a logical structure to securely store asymmetric keys. The .NET Framework provides the `CspParameters` class to create a key container and to add and remove keys to and from the container. To create a key container for an `RSACryptoServiceProvider` object, you first need to use the default constructor of the `CspParameters` class to create a key container instance. Then, you need to set the container name using the `KeyContainerName` property of the `CspParameters` class. Finally, to store the key pair in the key container, you need to pass the `CspParameters` object to the constructor while creating the `RSACryptoServiceProvider` object.

Code Snippet 7 uses a key container to store a key pair.

Code Snippet 7:

```
using System;  
  
using System.Security.Cryptography;  
  
using System.Text;  
  
...  
  
...  
  
CspParameters cspParams = new CspParameters();  
cspParams.KeyContainerName = "RSA_CONTAINER";  
RSACryptoServiceProvider rSAKeyGenerator = new  
RSACryptoServiceProvider(cspParams);  
  
Console.WriteLine("RSA key added to the container, \n\n{0}",  
rSAKeyGenerator.ToXmlString(true));
```

Figure 16.8 shows the output of Code Snippet 7.

```
C:\Windows\system32\cmd.exe
RSA key added to the container.

<RSAKeyValue><Modulus>tnooy3WnR/ZFYgS9UxE3xUKhBDqA+c5B195s05IEgxXVGzoUJSq+3mkmHE
WZF0JHm9vMsIz0LL6dB0Qm0+gSF1Ri1oPpAUi2+Ue5Bn6c58jKupoY17cS5PEWw81MIJozUgAQQftNiF
IK9ObExsQsw27UJDNeWELlce9+hanZmm8=</Modulus><Exponent>AQAB</Exponent><P>64HMUdro
rhosErKifQT7mYSNrJZmcv4CqI9+3W6U9SGjLC809d27Yupxf8yk0Ntr0d6czolBo60nWhirf iARw==
</P><Q>xls00BQ6kEwLNB7oe9LW53kGaNFSvt4n4QnABRyim50MC5dDw+34LdBshSL6DNPEiFt0XLepe
wyUifzileuQmQ==</Q><DP>IZW3D41KUVnCI0B2nm2Ue8Uddx1Irt5uFp91Fkf09rY/UNPC17UxPZ/1Y
6ZJcsd3zswSTJj6gYmRFEmLTq37jQ==</DP><DQ>s/y0eEOi0ITwNZ9GPFWXY/sBweMPPnq/nQgH4ZuW
QbZD09CygPtSk9/oWPbgBvux/jleYTOW6j0EHCIH39etIQ==</DQ><InverseQ>5bhIIcrr8Q13WZ5xh
XPHiYoS/oxzjUDM7MTemsPillgSrql9wgmKFO9IUDjYkPVDzCD458iqLFPRDqhhyp/UMYA==</Inverse
Q><D>U1eHeUN6a1aA90C/gigmor0usIB0RZ21BeUAH9CvejMUhXm6NMQOEhUQWc3yZhCih/agNg1f jkk
6.jkTPtPRLDXmiWd3epKoZU5qRzp7i46a3jFdw9UGcHYL1sI5RAX+tJUFrB/rj8vslo?stNt9Uc4xpUR
rE3MnRZDphGnkPhE=</D></RSAKeyValue>
Press any key to continue . . .
```

Figure 16.8: Output of Code Snippet 7

To retrieve a key pair from the container, you need to create a new object of the `CspParameters` class and initialize it with the name of the key container that contains the key pair. Then, you need to create a new object of the `RSACryptoServiceProvider` initialized with the `CspParameters` object. The `RSACryptoServiceProvider` object will now contain the keys stored in the key container.

Code Snippet 8 retrieves keys from the key container, named `RSA_CONTAINER`.

Code Snippet 8:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
CspParameters cp = new CspParameters();
cp.KeyContainerName = "RSA_CONTAINER";
RSACryptoServiceProvider rsaEncryptor = new
RSACryptoServiceProvider(cp);
```

16.4.3 Encrypting Data

To encrypt data, you need to create a new instance of the `RSACryptoServiceProvider` class and call the `ImportParameters()` method to initialize the instance with the public key information exported to an `RSAPublicKey` structure.

Code Snippet 9 shows how to initialize an `RSACryptoServiceProvider` object with the public key exported to an `RSAParameters` structure.

Code Snippet 9:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(false);
RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider();
rsaEncryptor.ImportParameters(rSAKeyInfo);
```

If the public key information is exported to XML format, you need to call the `FromXmlString()` method to initialize the `RSACryptoServiceProvider` object with the public key, as shown in Code Snippet 10.

Code Snippet 10:

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider();
rsaEncryptor.FromXmlString(publicKey);
```

After the `RSACryptoServiceProvider` object is initialized with the public key, you can encrypt data by calling the `Encrypt()` method of the `RSACryptoServiceProvider` class. The `Encrypt()` method accepts the following two parameters:

- ➔ byte array of the data to encrypt
- ➔ A Boolean value that indicates whether or not to perform encryption using Optimal Asymmetric Encryption Padding (OAEP) padding. A true value uses OAEP padding while a false value uses PKCS#1 v1.5 padding

Note - In encryption, padding is used with an encryption algorithm to make the encryption stronger. Padding prevents predictability to find patterns that might aid in breaking the encryption. For example, OAEP is a padding scheme often used together with RSA encryption.

The `Encrypt()` method after performing encryption returns a byte array of the encrypted text, as shown in Code Snippet 11.

Code Snippet 11:

```
byte[] plainbytes = new UnicodeEncoding().GetBytes("Plain text to encrypt.");
byte[] cipherbytes = rsaEncryptor.Encrypt(plainbytes, true);
```

16.4.4 Decrypting Data

To decrypt data, you need to initialize an `RSACryptoServiceProvider` object using the private key of the key pair whose public key was used for encryption.

Code Snippet 12 shows how to initialize an `RSACryptoServiceProvider` object with the private key exported to an `RSAParameters` structure.

Code Snippet 12:

```
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
RSACryptoServiceProvider rsaDecryptor = new RSACryptoServiceProvider();
rsaDecryptor.ImportParameters(rSAKeyInfo);
```

Code Snippet 13 shows how to initialize an `RSACryptoServiceProvider` object with the private key exported to XML format.

Code Snippet 13:

```
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
string keyPair = rSAKeyGenerator.ToXmlString(true);
RSACryptoServiceProvider rsaDecryptor = new RSACryptoServiceProvider();
rsaDecryptor.FromXmlString(keyPair);
```

When the `RSACryptoServiceProvider` object is initialized with the private key, you can decrypt data by calling the `Decrypt()` method of the `RSACryptoServiceProvider` class. The `Decrypt()` method accepts the following two parameters:

- byte array of the encrypted data
- A Boolean value that indicates whether or not to perform encryption using OAEP padding. A `true` value uses OAEP padding while a `false` value uses PKCS#1 v1.5 padding

The Decrypt() method returns a byte array of the original data, as shown in the following code snippet:

Code Snippet:

```
byte[] plainbytes = rsaDecryptor.Decrypt(cipherbytes, false);
```

Code Snippet 14 shows a program that performs asymmetric encryption and decryption.

Code Snippet 14:

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class AsymmetricEncryptionDemo
{
    static byte[] EncryptData(string plainText, RSAParameters
        rsaParameters)
    {
        byte[] plainTextArray = new
            UnicodeEncoding().GetBytes(plainText);
        RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
        RSA.ImportParameters(rsaParameters);
        byte[] encryptedData =
            RSA.Encrypt(plainTextArray, true);
        return encryptedData;
    }
    static byte[] DecryptData(byte[] encryptedData, RSAParameters
        rsaParameters)
    {
        RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
        RSA.ImportParameters(rsaParameters);
        byte[] decryptedData = RSA.Decrypt(encryptedData, true);
        return decryptedData;
    }
}
```

```
static void Main(string[] args)
{
    Console.WriteLine("Enter text to encrypt:");
    String inputText = Console.ReadLine();
    RSACryptoServiceProvider RSA = new
        RSACryptoServiceProvider();
    RSAParameters RSAParam = RSA.ExportParameters(false);
    byte[] encryptedData = EncryptData(inputText, RSAParam);
    string encryptedString =
        Encoding.Default.GetString(encryptedData);
    Console.WriteLine("\nEncrypted data
\n{0}", encryptedString);
    byte[] decryptedData = DecryptData(encryptedData,
        RSA.ExportParameters(true));
    String decryptedString = new
        UnicodeEncoding().GetString(decryptedData);
    Console.WriteLine("\nDecrypted data \n{0}", decryptedString);
}
```

In Code Snippet 14, the `Main()` method creates a `RSACryptoServiceProvider` object and exports the public key as a `RSAParameters` structure. The `EncryptData()` method is then called passing the user entered plain text and the `RSAParameters` object. The `EncryptData()` method uses the exported public key to encrypt the data and returns the encrypted data as a byte array. The `Main()` method then exports both the public and private key of the `RSACryptoServiceProvider` object into a second `RSAParameters` object. The `DecryptData()` method is called passing the encrypted byte array and the `RSAParameters` object. The `DecryptData()` method performs the decryption and returns the original plain text as a string.

Figure 16.9 shows the output of Code Snippet 14.

The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
Enter text to encrypt:  
This is plain text.  
  
Encrypted data  
y$0*x10ùqP?'ëIíxbxA?►ç?Iv♦ôN]z_1\oeZt*,E0_♦,♦j=IF@E zlsçI♦OL'KOç.,x,0xE.êzêDâ  
,¬F-b‡?+?ßTäd<08Ü0f4Px?¶z0-N_P^"*,>(ioxÜXoU',%  
  
Decrypted data  
This is plain text.  
Press any key to continue . . .
```

Figure 16.9: Output of Code Snippet 14

16.5 Check Your Progress

1. Which of the following statements regarding asymmetric encryption are true?

(A)	Asymmetric encryption uses a public key to encrypt data and the corresponding private key to decrypt data.
(B)	Asymmetric encryption uses a single key to both encrypt and decrypt data.
(C)	Asymmetric encryption can be performed using the <code>RSACryptoServiceProvider</code> class.
(D)	Asymmetric encryption can be performed using the <code>DSACryptoServiceProvider</code> class.
(E)	Asymmetric encryption uses the <code>CryptoStream</code> class to perform encryption and decryption.

(A)	A, C, D	(C)	C
(B)	B	(D)	D

2. You are trying to retrieve the key and IV generated using the `Rijndael` algorithm. Which of the following codes will help you achieve this?

(A)	<pre>SymmetricAlgorithm symAlgo = new RijndaelManaged(); symAlgo.Generate(); symAlgo.GenerateIV(); byte[] generatedKey = symAlgo.Key; byte[] generatedIV = symAlgo.IV;</pre>
(B)	<pre>Rijndael symAlgo = new Rijndael(); symAlgo.GenerateKey(); symAlgo.GenerateIV(); byte[] generatedKey = symAlgo.Key; byte[] generatedIV = symAlgo.IV;</pre>
(C)	<pre>AsymmetricAlgorithm symAlgo = new RijndaelManaged(); byte[] generatedKey = symAlgo.Key; byte[] generatedIV = symAlgo.IV;</pre>

(D) SymmetricAlgorithm symAlgo = new RijndaelManaged();
 byte[] generatedKey = symAlgo.Key;
 byte[] generatedIV = symAlgo.IV;

(A)	A	(C)	C
(B)	B	(D)	D

3. You are trying to initialize an RSACryptoServiceProvider with a private key exported to an RSAParameters structure. Which of the following codes will help you achieve this?

(A)	RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider(); RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(false); RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider(); rsaEncryptor.ImportParameters(rSAKeyInfo);
(B)	RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider(); RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true); RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider(); rsaEncryptor.ImportParameters(rSAKeyInfo);
(C)	RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider(); RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(); RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider(); rsaEncryptor.ImportParameters(rSAKeyInfo);
(D)	RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider(); RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(); RSACryptoServiceProvider rsaEncryptor = new RSACryptoServiceProvider(); rsaEncryptor.ImportParameters(true);

(A)	A	(C)	C
(B)	B	(D)	D

4. You are trying to export a key pair generated using the `RSACryptoServiceProvider` to a key container. Which of the following codes will help you to initialize a `RSACryptoServiceProvider` object with a `CspParameters` object?

(A)	<pre>CspParameters cspParams = new CspParameters(); cspParams.KeyContainerName = "RSA_CONTAINER"; RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider(cspParams);</pre>
(B)	<pre>CspParameters cspParams = new CspParameters("RSA_CONTAINER"); RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider(cspParams);</pre>
(C)	<pre>CspParameters cspParams = new CspParameters(); cspParams.KeyContainerName = "RSA_CONTAINER"; RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider("RSA_CONTAINER");</pre>
(D)	<pre>CspParameters cspParams = new CspParameters(); cspParams.ContainerName = "RSA_CONTAINER"; RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider(cspParams);</pre>

(A)	A	(C)	C
(B)	B	(D)	D

5. Which of the following statements are true regarding symmetric encryption?

(A)	Symmetric encryption uses a key pair to encrypt and decrypt data.
(B)	Symmetric encryption can be performed using the <code>RijndaelManaged</code> class.
(C)	Symmetric encryption can be performed using the <code>RSACryptoServiceProvider</code> class.
(D)	Symmetric encryption key is automatically generated when a call to the default constructor of the encryption implementation class is made.

(A)	A	(C)	C
(B)	B	(D)	D

16.5.1 Answers

1.	A
2.	D
3.	B
4.	A
5.	C



Summary

- ➔ Encryption is a security mechanism that converts data in plain text to cipher text.
- ➔ An encryption key is a piece of information or parameter that determines how a particular encryption mechanism works.
- ➔ The .NET Framework provides various types in the System.Security.Cryptography namespace to support symmetric and asymmetric encryptions.
- ➔ When you use the default constructor to create an object of the symmetric encryption classes, a key and IV are automatically generated.
- ➔ The ICryptoTransform object is responsible for transforming the data based on the algorithm of the encryption class.
- ➔ The CryptoStream class acts as a wrapper of a stream-derived class, such as FileStream, MemoryStream, and NetworkStream.
- ➔ When you call the default constructor of the RSACryptoServiceProvider and DSACryptoServiceProvider classes, a new public/private key pair is automatically generated.