

Developing Applications with C#



Developing Applications with C#

© 2023 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

First Edition - 2023



Onlinevarsity



**LEARN
ANYWHERE
ANYTIME**

Preface

The book titled "Developing Applications with C#" delves into essential aspects of C# programming language. C# is a powerful language that empowers developers to rapidly create solutions for the Microsoft .NET platform. Commencing with an introduction to the .NET Framework, it outlines fundamental characteristics of C# and elucidates its object-oriented capabilities. Additionally, the book provides an in-depth exploration of Visual Studio 2022 Integrated Development Environment (IDE).

This comprehensive Learner's Guide further delves into more advanced aspects of C#, including topics such as delegates, query expressions, advanced types such as partial types and nullable types, and more. The book also encompasses various new features of C# 10.0.

This book is the culmination of the dedicated efforts of the Design Team, consistently committed to delivering the finest and most cutting-edge content in Information Technology. The process of creating this book has adhered to the rigorous standards outlined in the ISO 9001 certification for Aptech-IT Division, Education Support Services.

Your suggestions are highly appreciated in our continuous effort to enhance the quality of our content.

Table of Contents

Sessions

Session 1: Getting Started with C#

Session 2: Basic Building Blocks in C#

Session 3: Programming Constructs and Arrays

Session 4: Classes and Methods in C#

Session 5: Inheritance and Polymorphism

Session 6: Abstract Classes and Interfaces

Session 7: Properties, Indexers, and Record Types

Session 8: Namespaces and Exception Handling

Session 9: Events, Delegates, and Collections

Session 10: Generics and Iterators

Session 11: GUI and Connectivity with SQL Database

Session 12: Advanced Concepts in C#

Session 13: Building Cross-Platform Mobile Apps Using .NET MAUI

Session 14: .NET Development and the Future

Appendix



**MANY
COURSES
ONE
PLATFORM**



Onlinevarsity App for Android devices

Download from Google Play Store



Session 1

Getting Started with C#

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

This session will give you an insight into the .NET Framework, its architecture, and different versions of .NET Framework. It also gives an overview of .NET 7.0. It then introduces C# language and explains creating and executing an application with Visual Studio 2022 and .NET 7.0.

In this Session, you will learn to:

- Define and describe the .NET Framework
- Explain Components of .NET 7.0
- Explain C# language features
- Identify various editions of Visual Studio 2022
- Outline the differences between .NET Framework 4.8.1 and .NET 7.0
- Explain the process of running a program on .NET 7.0

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 transformation in application development, it became possible for clients and servers to communicate with each other in a vendor-independent manner.

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

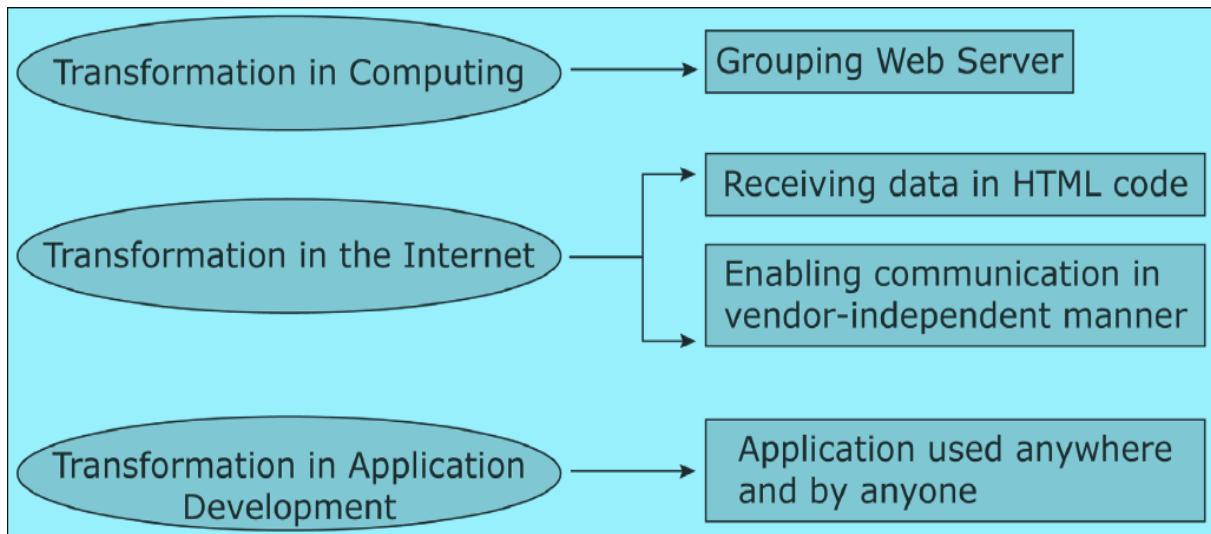


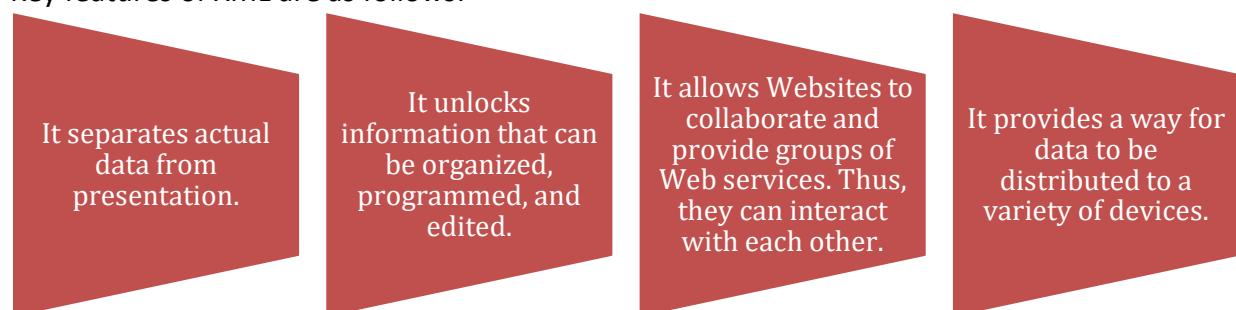
Figure 1.1: Transformations in Computing, Internet, and Application Development

All these transformations are supported by the technology platform introduced by Microsoft called .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 few 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:



Key features of XML are as follows:



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 Common Intermediate Language (CIL) and stored in a file called assembly. This assembly is then compiled by the Common Language Runtime (CLR) to the native code at runtime.

CIL was formerly called Microsoft Intermediate Language (MSIL).

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

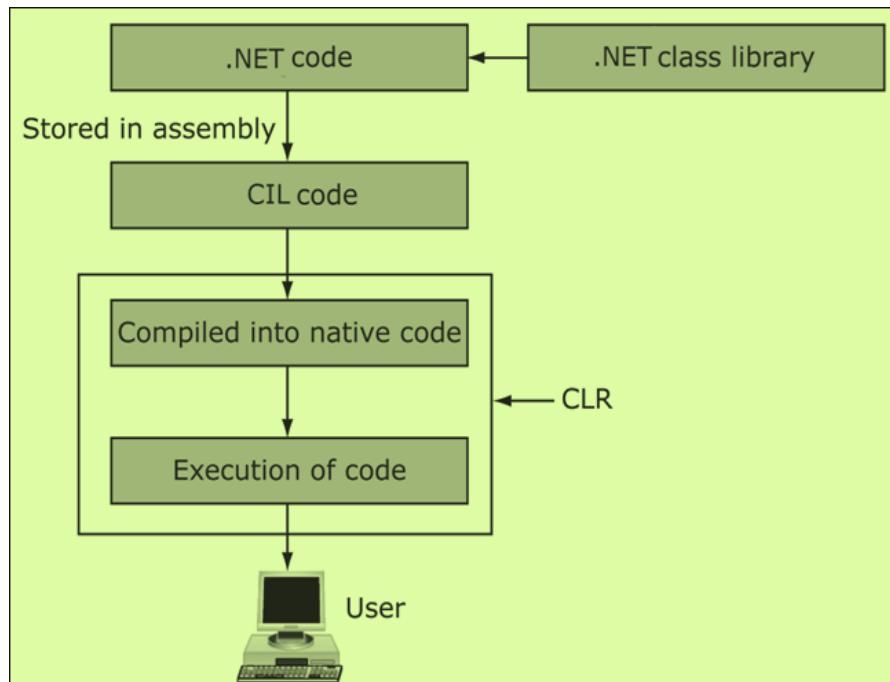


Figure 1.2: Process of Conversion of CIL Code to the Native Code

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

Microsoft has released different versions of .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, which was the first ever Integrated Development Environment (IDE) for .NET Framework. The IDE provided a comprehensive set of tools, templates, and libraries required to create .NET Framework applications. It includes CLR, class libraries of .NET Framework and ASP.NET, which is a development platform used to build Web pages.
- **.NET Framework 1.1:** This is the first upgraded version released with Microsoft Visual Studio .NET 2003.

It was incorporated with Microsoft Windows Server 2003 and included following features:

- Supported components used to create applications for mobiles as a part of the framework
- Supported Oracle databases as a repository to store information in tables
- Supported IPv6 protocol and Code Access Security (CAS) for Web-based applications
- Enabled running assemblies of Windows Forms from a Website
- Introduced .NET Compact Framework to help create applications intended for mobile phones and Personal Digital Assistants (PDAs) which were used before smartphones came into existence

Note: CAS is a Microsoft security mechanism to ensure that only code trusted by .NET Framework is allowed to perform critical actions such as requesting memory allocation and accessing databases. IPv6 stands for Internet Protocol (IP) version 6. It is a protocol that overcomes shortage of IP addresses by supporting 5 X 1028 addresses.

- **.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 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 supported to develop AJAX-enabled Websites 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 Visual Studio .NET 2010 introduced several new features, the key feature being the Dynamic Language Runtime (DLR). The DLR is a runtime environment that enables .NET programmers to create applications using dynamic languages such as Python and Ruby. Also, .NET Framework 4.0 introduced support for parallel computing that utilizes multi-core capabilities of computers. In addition, this version provided improvement in ADO.NET, WCF, and WPF. It also introduced new language features, such as dynamic dispatch, named parameters, and

optional parameters.

- **.NET Framework 4.5:** This version included with Visual Studio .NET 2012 provided enhancements to .NET Framework 4.0, such as enhancement in asynchronous programming through `async` and `await` keywords. It also provided support for Zip compression, support for regex timeout, and more efficient garbage collection.
- **.NET Framework 4.8 and 4.8.1:** .NET Framework 4.8.1 is the last version of .NET Framework and no further versions may be released. This is because Microsoft now rebranded the revamped platform into .NET 5.0.

However, .NET Framework as a platform will continue to be serviced with monthly security and reliability bug fixes even if no new versions of it are released. Additionally, it will continue to be included with Windows, with no plans to remove it. The user does not have to migrate his/her .NET Framework apps, but for any new development, .NET platform is recommended, instead of .NET Framework.

- **.NET 5.0:** .NET 5.0 was introduced by Microsoft on November 10, 2020. It combined .NET Core with .NET Framework to create a unified environment and a unified output for all things related to .NET and .NET Core.

.NET 5.0 supports all the Visual Studio versions that came after Visual Studio 2017.

- **.NET 6.0:** .NET 6.0 was planned for release in November 2021. It aimed to be a Long-Term Support (LTS) release and would continue the path of cross-platform development and unifying the .NET ecosystem.
- **.NET 7.0:** .NET 7.0 is the most recent version, as on date. It is so far the fastest version of .NET. There are noticeable performance improvements for ARM64 devices. Updating to .NET 7.0 might be a game changer if one wants to spend less money on their application operating in the cloud or use less resources to run the .NET application locally.

Table 1.1 summarizes the evolution of major .NET versions along with respective IDEs.

Year	.NET	IDE Name
2002	.NET Framework 1.0	Visual Studio .NET (2002)
2003	.NET Framework 1.1	Visual Studio .NET 2003
2005	.NET Framework 2.0	Visual Studio 2005
2006	.NET Framework 3.0	Visual Studio 2005 with .NET Framework 3.0 support
2007	.NET Framework 3.5	Visual Studio 2008
2010	.NET Framework 4	Visual Studio 2010
2012	.NET Framework 4.5	Visual Studio 2012
2015	.NET Framework 4.6	Visual Studio 2015
2017	.NET Framework 4.7	Visual Studio 2017
2019	.NET Framework 4.8	Visual Studio 2019

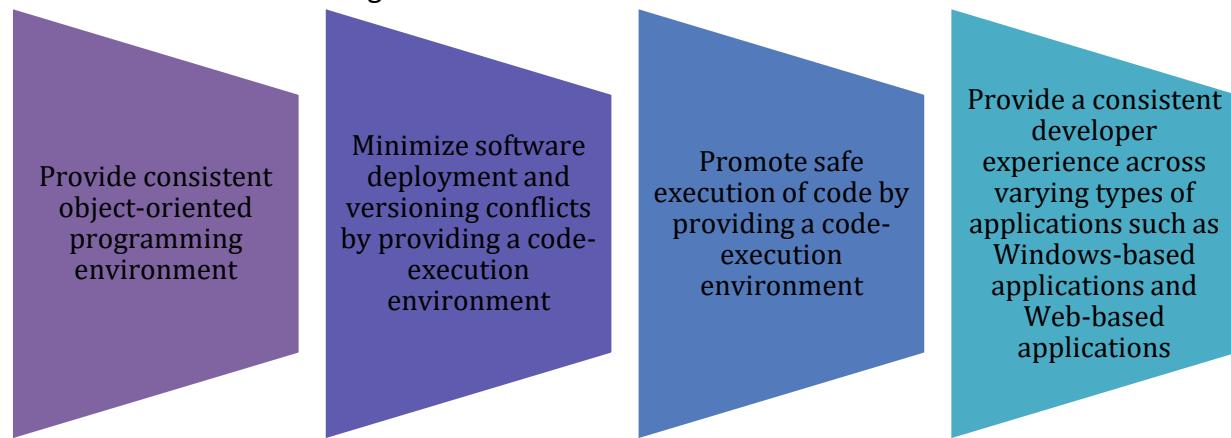
Year	.NET	IDE Name
2022	.NET Framework 4.8.1	Visual Studio 2022
2020	.NET 5.0	Visual Studio 2019
2022	.NET 6.0	Visual Studio 2022
2022	.NET 7.0	Visual Studio 2022

Table 1.1: Major Versions of .NET Framework, .NET, and Visual Studio IDE

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:



1.1.3 .NET Framework Components

The .NET Framework is made up of several components. 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.

➤ CLR

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

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

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.

➤ **.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, following command is 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 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 runtime 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 User Interface (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.

➤ **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 various components of .NET Framework.

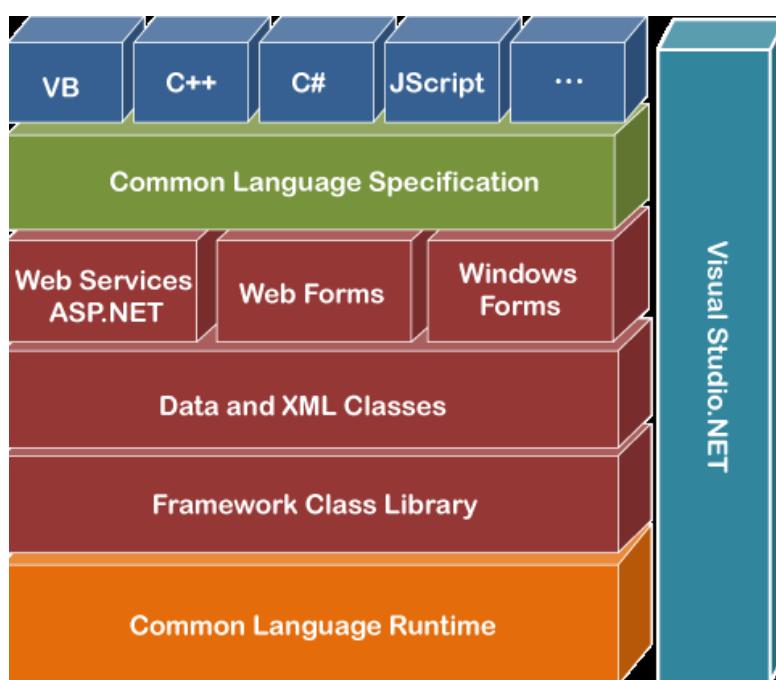


Figure 1.3: Components of .NET Framework

1.2 Introduction to .NET 7.0

As a pivotal milestone in Microsoft's technological evolution, .NET 7.0 brings forth a wealth of enhancements. This release amplifies developer's productivity with augmented language features and runtime performance. It unifies application development across Web, desktop, cloud, and more, fostering seamless cross-platform solutions. With improved tooling, security, and interoperability, .NET 7.0 empowers developers to shape innovative software experiences.

Whether building modern Web applications, robust desktop software, or cloud-native solutions, capabilities of .NET 7.0 set a new standard for efficient and versatile development in the ever-evolving technology landscape.

1.2.1 Components of .NET 7.0

.NET 7.0 boasts an amalgamation of cutting-edge elements to elevate the development landscape. It encompasses an advanced runtime, a comprehensive suite of libraries, and innovative language features that empower developers to craft efficient and modern applications. With a focus on performance optimization, cross-platform versatility, and enhanced security, .NET 7.0 sets the stage for seamless development across Web, desktop, and cloud domains. It integrates tools such as ASP.NET Core and Entity Framework Core, offering developers a unified toolkit to create sophisticated and robust software solutions. These cater to the demands of contemporary technology environments.

1.2.2 Features of .NET 7.0

Microsoft has added many new features and improved many pre-existing features in .NET 7.0.

- **Enhanced Performance:** .NET 7.0 prioritizes performance optimization, resulting in faster execution and improved responsiveness of applications.
- **Cross-Platform Compatibility:** .NET 7.0 ensures seamless deployment across various platforms, enabling consistent application behavior across different operating systems.
- **Comprehensive Libraries:** A versatile suite of libraries caters to diverse application domains, offering developers pre-built solutions for common tasks.
- **Integration of ASP.NET Core:** ASP.NET Core integration facilitates modern Web application development with enhanced performance and scalability.

1.3 Introduction to C# 10.0

C# is a programming language designed for building a wide range of applications that run on the .NET Framework. Microsoft introduced C# as a new programming language to address problems posed by traditional languages. C# was developed to provide following benefits:

- Create a 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

Purpose of C# Language

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.

C# 10.0 marks the evolution of Microsoft's programming language, introducing a host of powerful features for developers. With enhanced pattern matching, improved asynchronous programming, and refined language constructs, C# 10.0 empowers programmers to write more concise and expressive code. This version further streamlines development by introducing capabilities, classifies interpolated strings enhancements, and simplified property patterns.

Whether crafting Web-applications, desktop software, or cloud-native solutions, C# 10.0 stands as a versatile toolset enabling developers to create modern, efficient, and scalable applications across diverse domains.

1.3.1 Features of C# 10.0

- **Enhanced Pattern Matching:** C# 10.0 introduces improved pattern matching, allowing developers to write more concise and expressive conditional statements.
- **Asynchronous Programming Improvements:** The new version streamlines asynchronous programming, making it easier to write and manage asynchronous code for better performance.
- **Simplified Property Patterns:** C# 10.0 enhances code readability by introducing simplified property patterns, simplifying object deconstruction and extraction of properties.
- **Interpolated Strings Enhancements:** String formatting gets more expressive with interpolated strings enhancements, allowing dynamic insertion of variables and expressions within strings.
- **Refined Language Constructs:** C# 10.0 refines existing language constructs, optimizing their behavior and improving overall developer productivity.
- **Modernized Language Features:** New features cater to modern programming paradigms, enabling developers to adopt contemporary techniques and design patterns.
- **Versatility Across Domains:** Features of C# 10.0 empower developers to create applications across various domains, from Web and mobile to cloud-native solutions.
- **Compatibility and Interoperability:** While introducing new features, C# 10.0 maintains compatibility with previous versions and ensures seamless interoperability with existing codebases.
- **Forward-Looking Language Evolution:** C# 10.0 reflects the language's continued evolution, aligning with emerging trends and addressing developer requirements for efficient and expressive programming.

- **Elevated Software Quality:** These features collectively contribute to the creation of high-quality software solutions, improving performance, readability, and maintainability of codebases.

1.3.2 Applications of C# 10.0

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

- Web Development
- Desktop Software
- Game Development
- Mobile App Development
- Cloud-Native Solutions
- Internet of Things (IoT) Development
- Machine Learning and AI Solutions
- Financial and Enterprise Software
- Scientific and Research Applications
- Educational Tools Applications

1.4 Introduction to Visual Studio 2022

Visual Studio 2022 signifies a leap forward in software development tools. With its innovative features and enhanced performance, it empowers developers to create exceptional applications across a wide spectrum of platforms. The new 64-bit architecture improves memory usage and responsiveness, while AI-driven IntelliCode enhances code suggestions and productivity.

Visual Studio 2022 supports the latest technologies and frameworks, enabling developers to craft modern Web, mobile, cloud, and desktop applications. This version sets the stage for seamless collaboration, efficient debugging, and rich development experience, making it an indispensable toolkit for developers aiming to build cutting-edge software solutions.

1.4.1 Visual Studio 2022 Environment

Visual Studio 2022 offers a dynamic development ecosystem. Its intuitive interface integrates powerful tools for coding, debugging, and collaboration. With enhanced performance and AI-driven features, it supports diverse platforms, streamlining the creation of modern applications across Web, mobile, desktop, and cloud domains.

1.4.2 Visual Studio 2022 Editions

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

Different editions of Visual Studio 2022 are listed in Table 1.2.

Visual Studio Editions	Description
Visual Studio 2022 Community	A free edition for individual developers, students, and open-source contributors, offering essential tools and features.
Visual Studio 2022 Professional	Geared towards professional developers, it provides advanced debugging, testing, and collaboration capabilities.
Visual Studio 2022 Enterprise	Designed for large teams and enterprise-scale projects, offering advanced tools for performance optimization, architecture, and DevOps integration.
Visual Studio 2022 Test Professional	Focused on testing activities, it aids Quality Analysis (QA) professionals in delivering high-quality software through comprehensive testing tools.
Visual Studio 2022 Enterprise for Mac	Provides macOS and iOS developers with a powerful IDE.
Visual Studio 2022 Build Tools	A lightweight version with a focus on build processes and provides necessary tools for building applications without requiring to install Visual Studio.

Table 1.2: Visual Studio 2022 Editions

1.4.3 Elements of Visual Studio 2022 IDE

Following are various elements of Visual Studio 2022 IDE:

- **Code Editor:** Is the heart of the IDE, offering syntax highlighting, code completion, and intelligent suggestions to streamline coding.
- **Solution Explorer:** Provides a hierarchical view of projects, files, and folders within a solution for easy navigation and management.
- **Toolbox:** Holds controls and components for drag-and-drop design in GUI-based applications.
- **Properties Window:** Displays properties of selected elements, allowing developers to modify settings and behaviors.
- **Solution and Project Management:** Enables the creation, organization, and management of projects and solutions.
- **Debugging Tools:** Contains comprehensive tools for setting breakpoints, inspecting variables, and debugging code issues.
- **Version Control Integration:** Provides integration with Git and other version control systems for team collaboration and code management.
- **Extensions and Marketplace:** Provides access to a vast ecosystem of extensions and plugins to enhance functionality and productivity.
- **Output Window:** Displays build and runtime output, error messages, and other information.
- **Task List:** Keeps track of to-do items and comments in code, aiding in code maintenance.

- **Error List:** Provides a list of code errors and warnings for easy identification and resolution.
- **Test Explorer:** Facilitates managing and running unit tests and test cases.
- **NuGet Package Manager:** Manages third-party libraries and packages for easy integration.
- **Live Share:** Enables real-time collaborative coding sessions.

These elements collectively create a powerful and user-friendly environment for developing, testing, and managing software projects in Visual Studio 2022.

1.4.4 Csc Command

Console applications that are created in C# run in a console window that provides simple text-based output. The csc (**C Sharp Compiler**) command can be used to compile a C# program at the command prompt.

A C# program can be compiled at command prompt using following syntax:

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

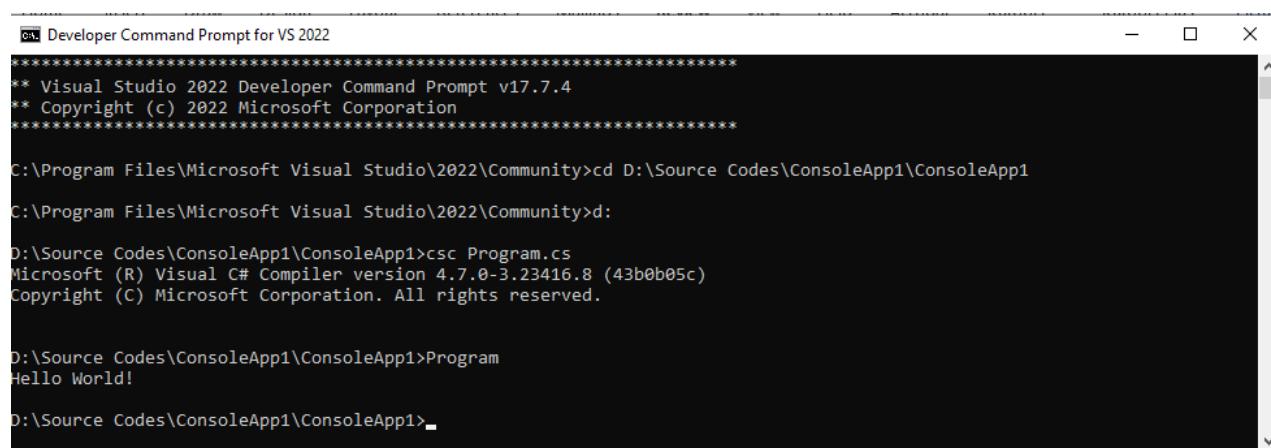
where, `file.cs`: Specifies the name of the program to be compiled.

Example: Open the Developer Command Prompt for Visual Studio 2022. Browse to the directory that contains the program to be compiled, say `Program.cs`. Then, one can type:

```
csc Program.cs
```

This command generates an executable file `Program.exe`.

Now, type the exe file name at the command prompt and it will display the output of the code. Figure 1.4 shows a sample output.



The screenshot shows a terminal window titled "Developer Command Prompt for VS 2022". The window displays the following command-line session:

```

C:\> Developer Command Prompt for VS 2022
***** 
** Visual Studio 2022 Developer Command Prompt v17.7.4
** Copyright (c) 2022 Microsoft Corporation
***** 

C:\Program Files\Microsoft Visual Studio\2022\Community>cd D:\Source Codes\ConsoleApp1\ConsoleApp1
C:\Program Files\Microsoft Visual Studio\2022\Community>d:
D:\Source Codes\ConsoleApp1\ConsoleApp1>csc Program.cs
Microsoft (R) Visual C# Compiler version 4.7.0-3.23416.8 (43b0b05c)
Copyright (C) Microsoft Corporation. All rights reserved.

D:\Source Codes\ConsoleApp1\ConsoleApp1>Program
Hello World!

D:\Source Codes\ConsoleApp1\ConsoleApp1>_

```

Figure 1.4: Developer Command Prompt

1.4.5 Creating and Executing an Application

Follow these steps to create and execute an application using Visual Studio 2022:

1. Launch Visual Studio 2022 on your computer.
2. Create a New Project.
3. Go to **File → New → Project** to create a new project.
4. Select **Project Type**. In the **Create a new project** dialog box, search for 'Console App' in the search bar. Select Console App (.NET) from the search results.
5. Select the Framework. The default will be .NET 7.0 (Long Term Support).
6. Provide a name for your project (for example, MyConsoleApp) and choose a location where it will be saved.
7. Click **Create** to proceed. The project is created. Visual Studio 2022 opens the Code Editor with the auto-generated skeleton code of a class, as shown in Code Snippet 1.

Code Snippet 1

```
using System;
namespace MyConsoleApp
{
    class Program {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Developer can replace this code with their own C# code.

8. Save the project.
9. Click the green Run button on the toolbar to run the project.

1.5 Differences Between .NET Framework 4.8.1 and .NET 7.0

Table 1.3 lists the differences between .NET Framework 4.8.1 and .NET 7.0.

Feature	Version 4.8.1	Version 7.0
Unified vs. Unified Framework	Part of the traditional .NET Framework family with Windows-centric focus	Part of the .NET unification journey, merging .NET Core and .NET Framework, with emphasis on cross-platform development
Platform Compatibility	Windows-centric, limited cross-platform support	Emphasizes cross-platform compatibility, enabling deployment on Windows, macOS, and Linux

Feature	Version 4.8.1	Version 7.0
Performance and Optimization	Limited performance improvements	Focuses on enhanced performance, introducing optimizations and runtime improvements
Language Features	Fewer modern language features	Introduces innovative language features for expressive and efficient coding
Containerization	Limited support for containerized deployment	Prioritizes containerization, enabling applications to be efficiently packaged and deployed using containers
64-Bit Support	Limited 64-bit support, primarily for Windows applications	Offers better 64-bit support, improving memory usage and responsiveness
Dependencies and NuGet	Heavily relies on Global Assembly Cache (GAC) for dependencies	Leans towards NuGet packages for managing dependencies
Modern Development Tools	Limited support for modern development workflows	Integrates seamlessly with modern tooling, CI/CD, and DevOps practices
Open-Source Contribution	Limited open-source contributions	Strong focus on open-source collaboration and community involvement
Futureproofing	Less suited for modern, cross-platform, and cloud-native development	Positions itself as a forward-looking platform for diverse application domains

Table 1.3: Differences Between .NET Framework 4.8.1 and .NET 7.0

1.6 Running a Program on .NET 7.0

Running a program on .NET 7.0, which is now part of the unified .NET platform, involves a few straightforward steps. Here is how one can run a simple C# console application using .NET 7.0:

- **Install .NET Software Development Kit (SDK) if not already installed:**
Developers should ensure they have the latest .NET SDK installed on their system. They can download it from the official .NET Website.
- **Create a New Project:**
Open a command prompt or terminal and navigate to the directory where to create a project.
Run following command to create a new console application:

```
Dotnet new console -o MyConsoleApp
```

This will create a new console application named **MyConsoleApp**.

- **Navigate to the Project Directory:**

Navigate to the newly created project directory using following command:

```
Cd MyConsoleApp
```

- **Edit the Code:**

Use Visual Studio 2022 to open and edit the '**Program.cs**' file inside the project folder.

Replace the existing autogenerated code with desired C# code, such as that given in Code Snippet 2.

Code Snippet 2
<pre>using System; namespace MyConsoleApp { class Program { static void Main(string[] args) { Console.WriteLine("Hello, .NET 7.0!"); } } }</pre>

- **Run the Application:**

Run the application by clicking green Run button in Visual Studio 2022. Alternatively, in the command prompt or terminal, run following command to build and run application:

```
Dotnet run
```

The output of the program will be displayed.

1.7 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.
- .NET Framework 4.8.1 is the last version of .NET Framework because Microsoft rebranded the platform into .NET 5.0 and later versions.
- .NET 7.0 is the most recent version of the platform as of today.
- 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 2022 provides a comprehensive environment to create, deploy, and run applications using .NET Framework.

1.8 Check Your Progress

1. Which of these statements about the .NET Framework and C# are true?

(A)	The .NET Framework is primarily useful for Java programming language
(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 .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)	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 features of 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 JIT compiler converts the operating system code to the MSIL codewhen 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 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)	C# application programming uses objects so that code written once can be reused

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

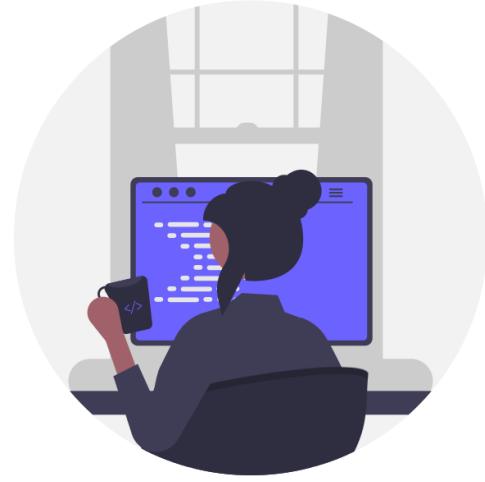
1.8.1 Answers

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

Try It Yourself

1. What are key features, enhancements, and best practices for beginners when using C# in conjunction with Visual Studio 2022 for developing desktop applications? How does this combination improve efficiency and user experience of new programmers? Present this research using a PowerPoint presentation.

2. What are the new features and enhancements introduced in C# 10.0 and Visual Studio 2022? How do these improvements impact the development process for C# programmers? Present this research using a PowerPoint presentation.



Session 2

Basic Building Blocks in C#

Welcome to the Session, **Basic Building Blocks 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 few data types that can be used to declare variables and store data. Visual Studio IDE 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. This session also explores statements and operators in C#.

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
- Describe constant interpolated strings
- List the keywords and escape sequences
- Explain input and output
- Explain different types of statements
- Explain the use of different types of operators

2.1 Variable 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 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 must 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. You can initialize a variable at the time of creating the variable or later. 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 referring to the value stored in that variable.

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.
```

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.
```

Code Snippet 1 declares two variables, namely `empNumber` and `empName`.

Code Snippet 1
<pre>int empNumber; string empName;</pre>

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:

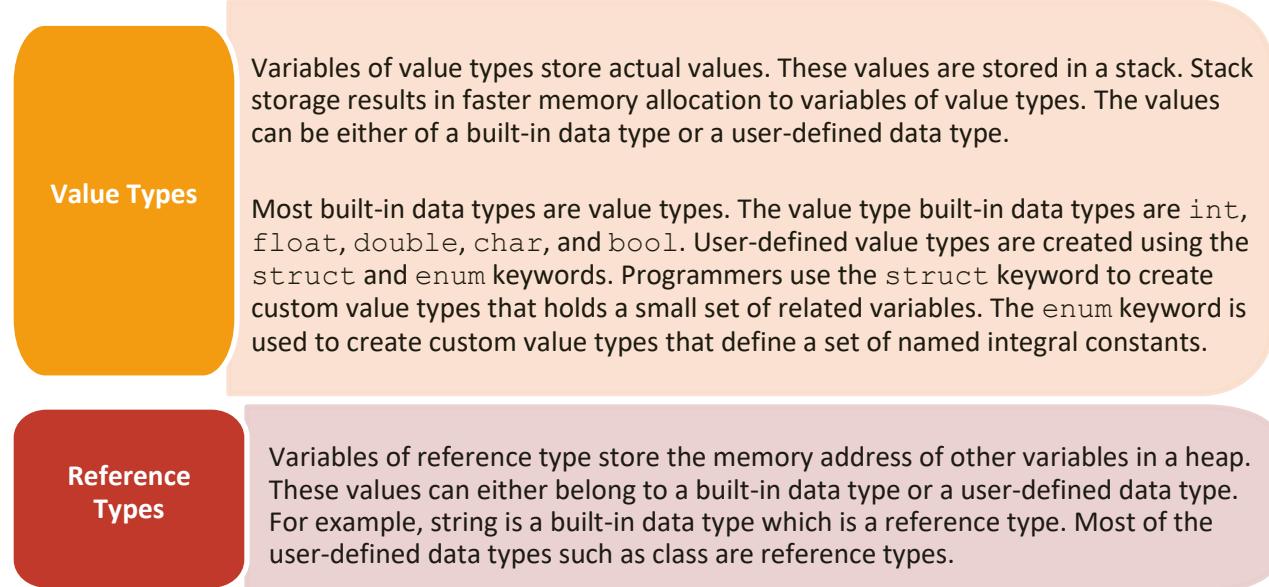


Figure 2.1 depicts a representation of these types.

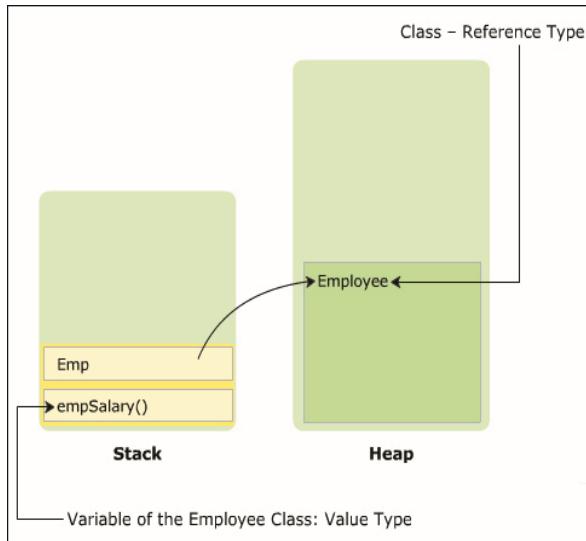


Figure 2.1: Data Types

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 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 variables hold the actual values. Reference types can be classified into 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.

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 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 of 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 2 demonstrates how to declare and initialize implicitly typed variables in C#.

Code Snippet 2

```
using System;
class VarDemo{
    static void Main(string[] args) {
        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 2, 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 2.2 displays the output.

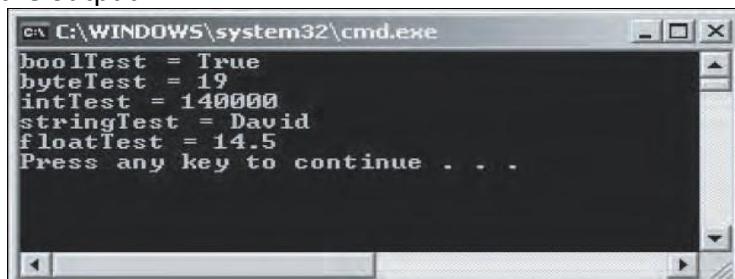


Figure 2.2: Output of Code Snippet 2

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 3.

Code Snippet 3

```
// 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 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 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 4.

Code Snippet 4

```
/* 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 must 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 5.

Code Snippet 5

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

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 IDE 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 to 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 2022 Command Prompt window.

Figure 2.3 displays 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 2.3: XML Comments

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 2.2 lists the widely used pre-defined XML tags and states their conventional use.

Pre-defined Tags	Description
<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 2.2: Pre-defined XML Tags

Code Snippet 6 demonstrates the use of XML comments.

Code Snippet 6

```
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>"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);
}
}

```

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 Necessity 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 will 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.

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

Code Snippet 7

```
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 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	<p>Boolean literals have two values, true or false. For example, <code>boolval = true;</code> where,</p> <ul style="list-style-type: none">• true: Is a boolean literal assigned to the variable <code>val</code>.
Integer Literal	<p>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, <code>longval = 53L;</code> where,</p> <ul style="list-style-type: none">• 53L: Is an integer literal assigned to the variable <code>val</code>.
Real Literal	<p>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, <code>floatval = 1.66F;</code> where,</p> <ul style="list-style-type: none">• 1.66F: Is a real literal assigned to the variable <code>val</code>.
Character Literal	<p>A character literal is assigned to a char data type. A character literal is always enclosed in single quotes. For example, <code>charval = 'A' ;</code> where,</p> <ul style="list-style-type: none">• A: Is a character literal assigned to the variable <code>val</code>.
String Literal	<p>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.</p> <p>For example, <code>string mailDomain = "@gmail.com";</code> where,</p> <ul style="list-style-type: none">• @gmail.com: Is a verbatim string literal.
Null Literal	<p>The null literal has only one value, null. For example, <code>string email = null;</code> where,</p> <ul style="list-style-type: none">• null: Specifies that e-mail does not refer to any objects (reference).

2.3.4 Constant Interpolated Strings

Before delving into constant interpolated strings, it is essential to understand interpolated strings in C#. They feature a '\$' symbol before the string literal and allow expressions enclosed in '{}' to

be dynamically evaluated and inserted into the string during runtime. This approach significantly improves string formatting, making it more concise and readable compared to traditional methods of string concatenation or formatting.

Constant interpolated strings extend this concept by enabling the creation of constant string values that incorporate expressions. In C# 10.0, developers craft string constants that are both expressive and dynamically assembled.

To declare a constant interpolated string, the developer can employ the `const` modifier in conjunction with the `$` symbol preceding the string literal, mirroring the syntax of regular interpolated strings as follows:

```
public const string Greeting = $"Hello, {GetUserName()}!";
```

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 2.3 lists the keywords used in C#.

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

Table 2.3: 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 2.4 lists the contextual keywords used in C#.

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

Table 2.4: Contextual Keywords

2.4.2 Escape Sequence Characters in C#

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

Table 2.5 displays the escape sequence characters and their corresponding non-printing characters in C#.

Escape Sequence Characters	Non-Printing Characters
\'	Single quote, required for character literals.
\"	Double quote, required for string literals.
\\	Backslash, required 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 American Standard Code for Information Interchange (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'.

Escape Sequence Characters	Non-Printing Characters
\uhhhh	Matches a Unicode character using hexadecimal representation (exactly four digits). For example, the character \u0020 represents a space.

Table 2.5: Escape Sequence Characters

Code Snippet 8 demonstrates the use of Unicode characters.

Code Snippet 8
<pre>string str = "\u0048\u0065\u006C\u006C\u006F"; Console.WriteLine("\t" + str + "!\n"); Console.WriteLine("David\u0020\"2007\" ");</pre>

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 newline 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:

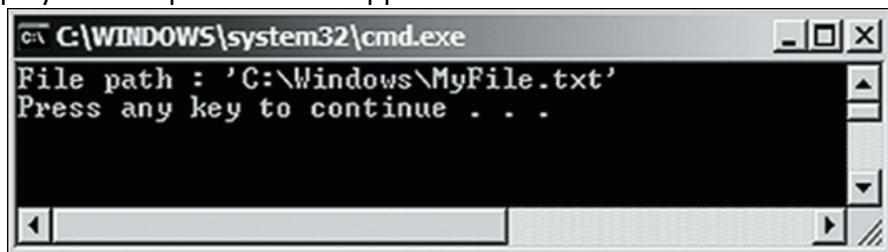
Hello!
David "2007"

Code Snippet 9 demonstrates the use of some of the commonly used escape sequences.

Code Snippet 9
<pre>using System; class FileDemo { static void Main(string[] args) { string path = "C:\\Windows\\MyFile.txt"; bool found = true; if (found) { Console.WriteLine("File path : \' " + path + " \'"); } else { Console.WriteLine("File Not Found! \a"); } } }</pre>

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 2.4 displays the output of Code Snippet 9.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window contains the text: 'File path : 'C:\Windows\MyFile.txt'', followed by 'Press any key to continue . . .'. The window has standard Windows UI elements like title bar, scroll bars, and a taskbar at the bottom.

Figure 2.4: Output of Code Snippet 9

2.5 Input and Output

Programmers often require displaying 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.

Three streams are as follows:

Standard in	Standard out	Standard err
The standard in stream takes the input and passes it to the console application for processing.	The standard out stream displays the output on the monitor.	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 require 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.

Following syntax is used for the `Console.Write()` method, which allows you to display the information on the console window:

Syntax

```
Console.Write("<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.

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 10 shows the difference between the `Console.Write()` method and `Console.WriteLine()` method.

Code Snippet 10

```
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 11 uses placeholders in the `Console.WriteLine()` method to display the result

of the multiplication operation.

Code Snippet 11

```
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 require 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 12 reads the name using the `ReadLine()` method and displays the name on the console window.

Code Snippet 12

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

In Code Snippet 12, 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 13 demonstrates the use of placeholders in the `Console.WriteLine()` method.

Code Snippet 13

```
using System;
class Loan {
    static void Main(string[] args) {
        string custName;
        double loanAmount;
        float interest = 0.09F;
        double interestAmount = 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} \n Interest rate
: {1:0%} \nInterest Amount : ${2:#,###.#0}", loanAmount,
interest, interestAmount );
        Console.WriteLine("Total amount to be paid :
${0:#,###.#0} ", totalAmount);
    }
}
```

In Code Snippet 13, 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 2.5 displays the output of Code Snippet 13.

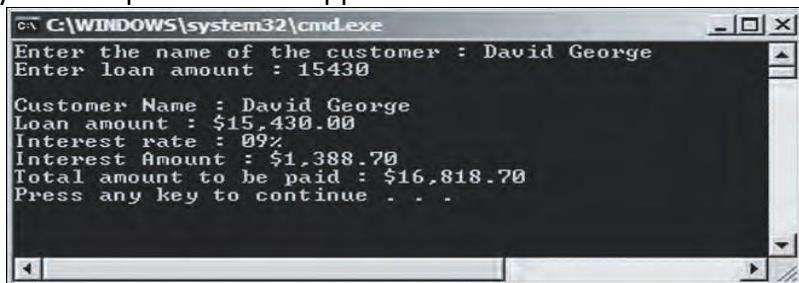


Figure 2.5: Output of Code Snippet 13

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 14 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 14

```
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.

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 2.6 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 2.6: 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 2.7 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

Format Specifier	Description
'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 2.7: Custom Numeric Format Specifiers

Code Snippet 15 demonstrates the conversion of a numeric value using C, D, and E format specifiers.

Code Snippet 15
<pre>int num = 456; Console.WriteLine("{0:C}", num); Console.WriteLine("{0:D}", num);</pre>

Output:

\$456.00 456
4.560000E+002

Code Snippet 16 demonstrates the use of custom numeric format specifiers.

Code Snippet 16
<pre>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); } }</pre>

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 2.6 displays the output of Code Snippet 16.

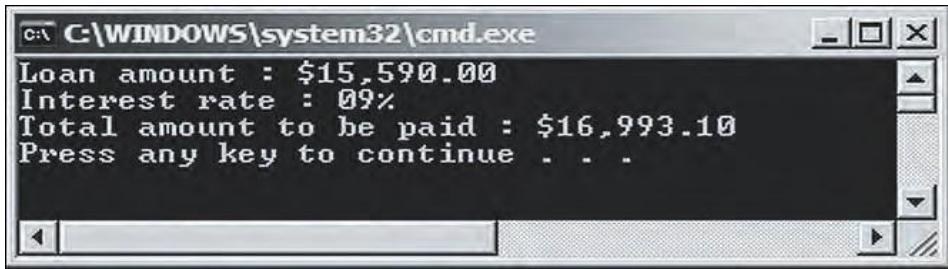


Figure 2.6: Output of Code Snippet 16

2.5.9 More Number Format Specifiers

There are some additional number format specifiers that are described in Table 2.8.

FormatSpecifier	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 2.8: Additional Numeric Format Specifiers

Code Snippet 17 demonstrates the conversion of a numeric value using F, N, and X format specifiers.

Code Snippet 17

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

Output:

456.00
456.00
1C8

2.6 Statements and Expressions

A C# program is a set of tasks that are performed 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 evaluate a value.

2.6.1 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 2.7 displays an example of a block.

```
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 2.7: Definition

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

2.6.2 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 18 shows an example of a statement in C#.

Code Snippet 18

```
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 19 shows an example of a block of statements in C#.

Code Snippet 19

```
...
{
    int side = 10;
    int height = 5;
    double area = 0.5 * side * height;
    Console.WriteLine("Area:{0}", 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 20 shows an example of nested blocks in C#.

Code Snippet 20

```
...
{
    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.

2.7 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 <code>if</code> , <code>else</code> , <code>switch</code> , and <code>case</code> .
Iteration Statements	An iteration statement helps you to repeatedly execute a block of code. The keywords associated with this statement are <code>do</code> , <code>for</code> , <code>foreach</code> , and <code>while</code> .
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 <code>break</code> , <code>continue</code> , <code>default</code> , <code>goto</code> , <code>return</code> , and <code>yield</code> .
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 <code>throw</code> , <code>try-catch</code> , <code>try-finally</code> , and <code>try-catch-finally</code> .
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 <code>checked</code> and <code>unchecked</code> .

2.8 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 runtime errors that disrupt the normal flow of the program. The `System.Exception` class is used to derive several exception classes that handle 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 following form:

```
checked statement:  
checked block
```

Code Snippet 21 creates a class **Addition** with the checked statement. This statement throws an overflow exception.

Code Snippet 21

```
using System;  
class Addition  
{  
    public static void Main(string[] args)  
    {  
        byte numOne = 255;  
        byte numTwo = 1;  
        byte result = 0;  
        try{  
            //This code throws an overflow exception  
            checked  
            {  
                result = (byte) (numOne + numTwo);  
            }  
        }  
        catch (OverflowException ex)  
        {  
            Console.WriteLine(ex);  
        }  
    }  
}
```

In Code Snippet 21, 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 2.8 shows the error message that is displayed after executing Code Snippet 21.

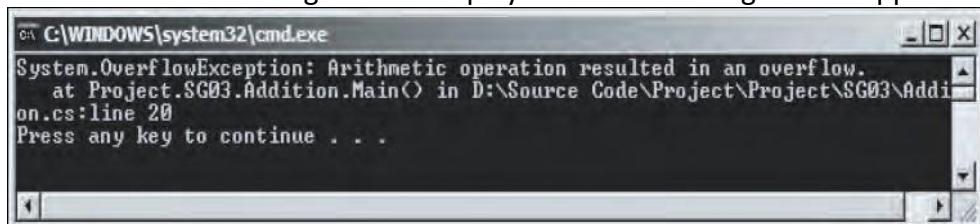


Figure 2.8: Error Message

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 following form:

```
unchecked-statement:  
unchecked block
```

Code Snippet 22 creates a class **Addition** that uses the unchecked statement.

Code Snippet 22

```
using System;  
class Addition {  
    public static void Main(string[] args) {  
        byte numOne = 255;  
        byte numTwo = 1;  
        byte result = 0;  
        try  
        {  
            //This code throws an overflow exception  
            checked  
            {  
                result = (byte) (numOne + numTwo);  
            }  
            unchecked  
            {  
                result = (byte) (numOne + numTwo);  
            }  
            Console.WriteLine("Result: " + result);  
        }  
        catch (OverflowException ex)  
        {  
            Console.WriteLine(ex);  
        }  
    }  
}
```

In Code Snippet 22, 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.

2.9 Top-Level Statements

C# introduced many new features recently and one of them is top-level statements. This feature mainly works with C# 9.0 onwards under .NET 5.0 and Visual Studio 2019 or higher versions. With top-level statements, one can easily reduce length of the program and eliminate unwanted coding lines. This is explained using the example shown in Code Snippet 23.

Code Snippet 23

```
using System;
namespace HelloWorld {
    class Program {
        static void Main (string[] args) {
            Console.WriteLine("I am Iron man") ;
        }
    }
}
```

In Code Snippet 23, the line of code, `Console.WriteLine("I am Iron man") ;` is the one that will perform action and rest of the lines are just boilerplate.

The `Main` method acts as the driver of an application. It defines the basic functionality of the object. One of the benefits of top-level statements is to enable programming without a `Main` method. A top-level statement can be implemented with the keyword `using`. This means that the `Main` method and rest of the boilerplate can be replaced with a top-level statement that has `using` keyword.

This is illustrated through Code Snippet 24.

Code Snippet 24

```
using System;
Console.WriteLine("I am Iron man") ;
```

In Code Snippet 24, the entire code is reduced to two lines just by utilizing the keyword `using`. This kind of statement is called as ‘top-level statement’. In C# 9.0 and higher versions, a top-level statement can eliminate class declaration, the main method, `args []` declaration, and more. At the time of compilation, the compiler will automatically generate the `Main` method with an `args []` string array.

Features of top-level statements are as follows:

Implicit Entry Point Method: Key points of the implicit entry point method include following:

- A C# application must always have only one entry point. This means that a project can have only one file with top-level statements. If you put top-level statements in more than one file in a project, then, it will cause a compiler error as follows:

CS8802 Only one compilation unit can have top-level statements.

- A project can have any number of additional source code files that do not have top-level statements.
- The `Main` method can be written explicitly, but it cannot function as an entry point. The compiler in this scenario will issue following warning:

CS7022 The entry point of the program is global code; ignoring 'Main()' entry point.

- Top-level statement allows users to write C# statements without explicitly creating a class. There cannot be top-level statements spread across two different classes.
- To select the entry point in a project with top-level statements, the main compiler option cannot be used even if the project includes one or more main methods.

Namespace and Type Definition: A file with top-level statements can also include namespaces and type definitions, but they must be placed after the top-level statements. This is illustrated in Code Snippet 25.

Code Snippet 25

```
using System;
Console.WriteLine("toplevel");
methodA();

public static class Program{
    public static void methodA()
    {
        Console.WriteLine("Static in Program class");
    }
}
```

Here, the second line is a top-level statement and the program includes a class declaration and definition after the top-level statement. Top-level statements can be useful for novice programmers as well as expert developers.

2.10 Expressions - Definition

Expressions are used to manipulate data. Similar to 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 26 demonstrates an example for expressions.

Code Snippet 26

```
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`.

2.11 Operators

Expressions in C# comprise one or more operators that perform some operations on variables.

An operation is an action performed on single or multiple values stored in variables 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.

2.11.1 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

2.11.2 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 2.9 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.	$100 - 47$
$*$ (Multiplication)	Performs multiplication.	$67 * 46$
$/$ (Division)	Performs division. The operator divides the first operand by the second operand and gives the quotient	$12000/10$

Operators	Description	Examples
	as the output.	
% (Modulo)	Performs modulo operation. The operator divides the two operands and gives the remainder of the division operation as the output.	100 % 33

Table 2.9: Arithmetic Operator Types

Code Snippet 27 demonstrates how to use arithmetic operators.

Code Snippet 27
<pre>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);</pre>

Output:

Addition 12
 Subtraction 8
 Multiplication 20
 Division 5
 Remainder 0

2.11.3 Relational Operators

Relational operators make a comparison between two operands and return a boolean value, true or false. Table 2.10 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

Relational Operators	Description	Examples
<=	Checks whether the first value is lesser than or equal to the second value.	75 <= 80

Table 2.10: Relational Operator Types

Code Snippet 28 demonstrates how to use relational operators.

Code Snippet 28
<pre>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));</pre>

Output:

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

2.11.4 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 namely, 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 2.11 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')

Logical Operator	Description	Example
<code>^</code> (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.	<code>(choice == 'Q') ^ (choice == 'q')</code>

Table 2.11: Logical Operator Types

Code Snippet 29 explains the use of the boolean inclusive OR operator.

Code Snippet 29

```
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 30 explains the use of the boolean AND operator.

Code Snippet 30

```
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 are evaluated to true, the complete expression returns true and the statement within the block is executed.

Code Snippet 31 explains the use of the boolean exclusive OR operator.

Code Snippet 31

```
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 2.12 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 2.12: Bitwise Logical Operators Code

2.11.5 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 following differences:

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

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

The conditional OR operator evaluates the second expression only if first expression returns false. This is because this operator returns true if either of the expressions is true. Thus, if first expression itself evaluates to true, result of the second expression is immaterial. Figure 2.9 displays the conditional operators.

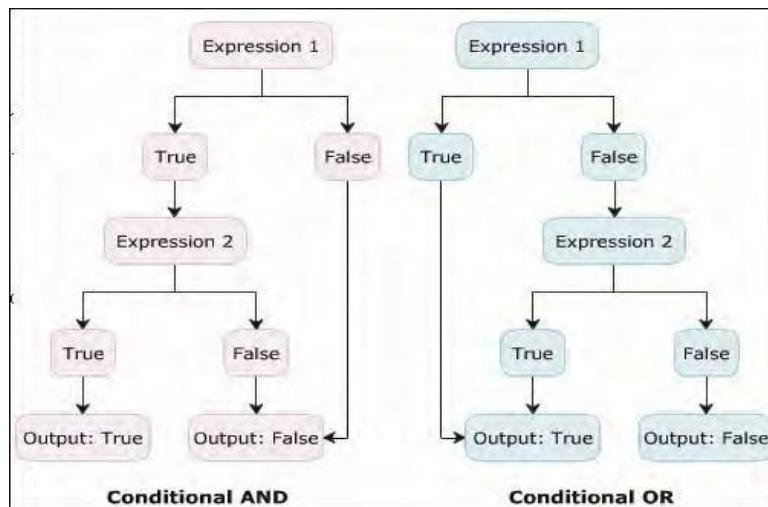


Figure 2.9: Conditional Operators

Code Snippet 32 displays the use of the conditional AND (`&&`) operator.

Code Snippet 32

```
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

2.11.6 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 2.13 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 2.13: Increment and Decrement Operators

2.11.7 Assignment Operators - Types

Assignment operators are used to assign the value of the right operand to the operand on the left 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 2.14 shows the use of assignment operators assuming the value of the variable **valueOne** is 10.

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

Table 2.14: Use of Assignment Operators

Code Snippet 33 demonstrates how to use assignment operators.

Code Snippet 33

```
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);
```

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.

2.11.8 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 2.15 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 2.15: Precedence of Operators

Code Snippet 34 demonstrates how to use logical operators.

Code Snippet 34

```
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
```

2.11.9 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 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 and the zero is added on the left. Code Snippet 35 demonstrates use of shift operators.

Code Snippet 35

```
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 35 is shown in Figure 2.10.

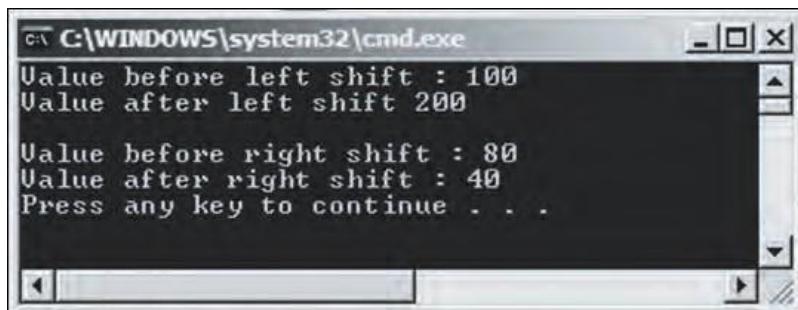


Figure 2.10: Output of Code Snippet 35

In Code Snippet 35, the `Main()` method of the class `ShiftOperator` performs shifting operations.

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

2.11.10 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 36 demonstrates the use of string concatenation operator.

Code Snippet 36

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

The output of Code Snippet 36 shows use of string concatenation operator, as shown in Figure 2.11.

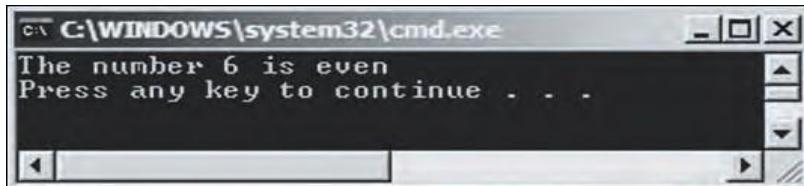


Figure 2.11: Output of Code Snippet 36

2.11.11 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 37 demonstrates the use of the ternary operator.

Code Snippet 37

```
using System;
class LargestNumber {
    public static void Main(string[] args) {
        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 37, 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 2.12 shows the output of the example using ternary operator.

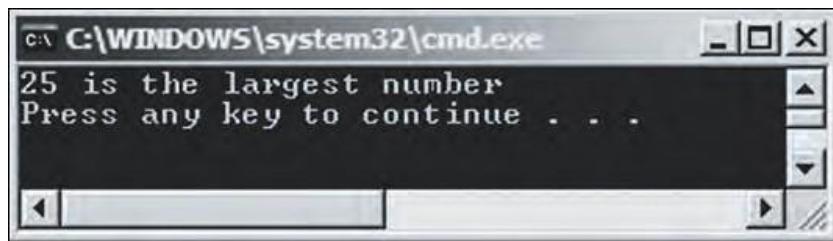


Figure 2.12: Output of Code Snippet 37

2.12 Summary

- A variable is a named location in the computer's memory and stores values.
- Comments are used to provide detailed explanation about 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.
- 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 perform mathematical and logical calculations.
- You can convert a data type into another data type implicitly or explicitly in C#.
- Top-level statements are a newly introduced feature that can aid developers to easily reduce length of a program and eliminate unwanted coding lines.

2.13 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
(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 respective escape sequence character

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

6. Match the non-printing characters against their corresponding escape sequence 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-2, C-3, D-1, E-4

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

(A)	The Read() method always inserts the new line character at the end of the read character
(B)	The data type conversion methods exist in the System 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

2.13.1 *Answers*

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

Try It Yourself

1. **Atlantis Scientific Systems** is an advanced scientific computation lab in **Colorado, USA**. The staff of this lab gathers data on various elements and phenomena and performs vast amount of scientific calculations and research on the accumulated data. This will help them to reach specific conclusions with which they can publish in journals and papers across the world.

Assume that you are one of the researchers at **Atlantis Scientific Systems** and have to gather and store following data given as input:

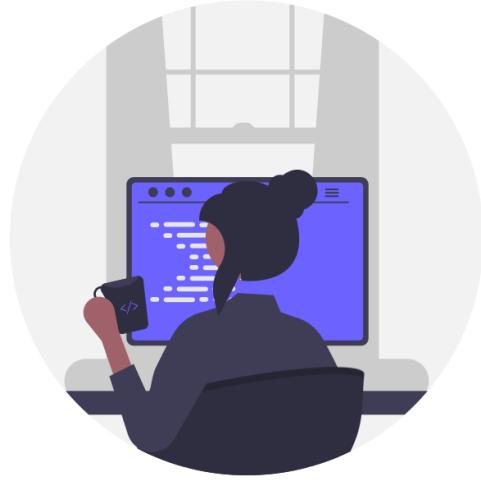
- Minimum Temperature of the locality
- Maximum Temperature of the locality
- Average Temperature of the locality
- Population of the town
- Population of the state
- Whether the city is a metropolis or not (true/false data)
- Average literacy percentage of the city
- Average qualifications of the population (Graduate/Postgraduate and so on.)

You must write a C# program that will help you to do this.

Test the output by compiling the program and executing the same through Visual Studio 2022 IDE.

2. Write a program to accept two numbers and then, display the result of following operations using the given bit-wise operators:

- Number 1 & Number 2:
- Number 1 | Number 2:
- ~ Number 1:
- Number 2:
- (\sim Number 1) & (\sim Number 2):
- (\sim Number 1) | (\sim Number 2):
- Number 1 $>>$ Number 2:
- Number 1 $<<$ Number 2:
- Number 2 $>>$ Number 1:
- Number 2 $<<$ Number 1:



Session 3

Programming Constructs and Arrays

Welcome to the Session, **Programming Constructs and Arrays**.

This session provides an overview of programming constructs, arrays in C#, and different types of arrays and briefly describes the functionality of the `Array` class.

In this Session, you will learn to:

- Explain selection constructs
- Describe loop constructs
- Explain jump statements in C#
- Define and describe arrays
- Explain different types of arrays
- Describe the `Array` class

3.1 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 following decision-making constructs:

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

3.1.1 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.

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.

Code Snippet 1 displays whether a given number is negative or not 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.

3.1.2 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.

Following is the syntax for the `if...else` statement:

Syntax

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

```
}

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

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.

3.1.3 *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 subsequent `else...if` statement is evaluated.

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;
}
```

Code Snippet 3 shows the code to determine 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.

3.1.4 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.

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;
        }
    }
}
```

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.

3.1.5 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 must 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

<pre>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; }</pre>

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

3.1.6 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 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 3.1 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 3.1: Output of Using Nested switch

3.1.7 No-Fall-Through Rule

Languages such as 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 must have a statement such as `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 3.2 shows the output of multiple `case` statements.

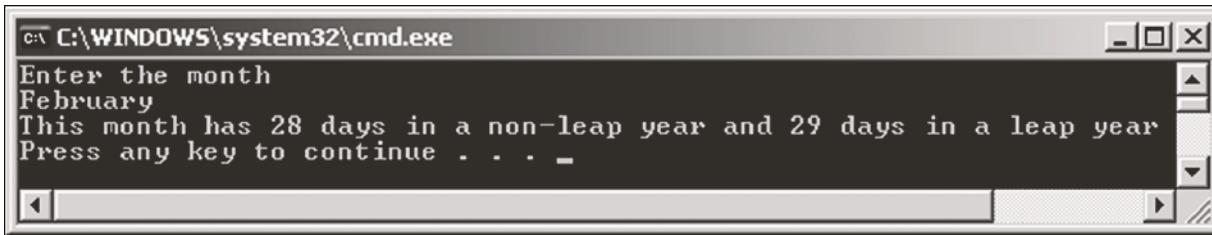


Figure 3.2: Multiple case Statements

3.2 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

3.2.1 `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.

Following is the syntax of the `while` loop:

Syntax

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

where,

condition: Specifies the boolean expression.

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

Code Snippet 8

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

3.2.2 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 3.3 shows the output of using nested while loop.

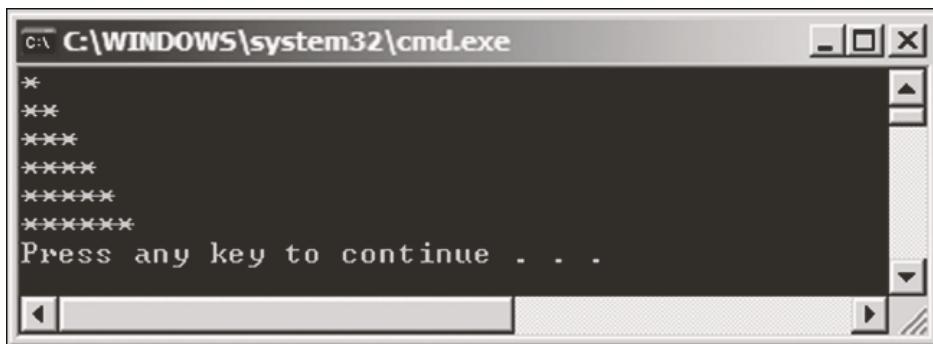


Figure 3.3: Output of Using Nested while Loop

3.2.3 *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.

Following is the syntax of the do-while loop:

Syntax

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

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.

3.2.4 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.

Following is the syntax of the `for` loop:

Syntax

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

where,

initialization: Initializes 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 such as `++`, `--`, and shortcut operators such as `+=` or `-=` are used in this section. Note that there is no semicolon at the end of the increment/decrement expressions.

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

3.2.5 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.WriteLine("{0} {1}", 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
```

3.2.6 Declaring Loop Control Variables within Loop Definition

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.

Following code demonstrates the declaration of the loop control variable within the loop definition.

```
for (int i = 1; i <= num; fact *= i++);
```

Here, the variable `i` is created just for the purpose of the loop and is of no use outside the loop. Hence, it is declared within the loop definition itself. The scope of `i` is limited to the `for` loop in which it is declared and cannot be used outside the loop.

3.2.7 for Loop with Multiple Loop Control Variables

The `for` loop also 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

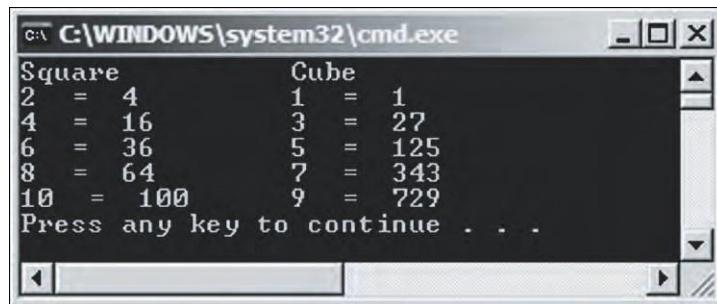
```
using System;  
class Numbers {  
static void Main(string[] args) {  
Console.WriteLine("Square \t\tCube");  
for (int i = 1, j = 0; i < 11; i++, j++) {
```

```

if ((i % 2) == 0) {
    Console.WriteLine("{0} = {1} \t", i, (i * i));
    Console.WriteLine("{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 3.4 depicts the outcome.



```

c:\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 3.4: Use of a for Loop

3.2.8 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 become 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 3.5 shows the `for` loop with missing operations.

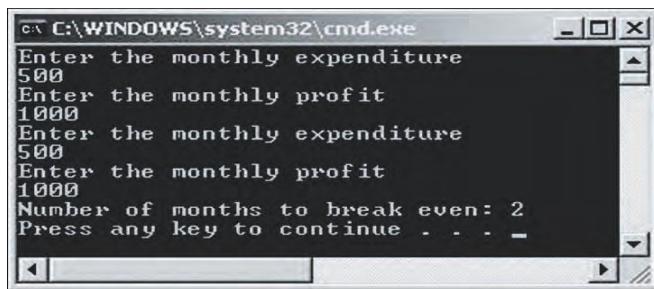


Figure 3.5: The for Loop with Missing Operations

Code Snippet 15 demonstrates a `for` loop that omits 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 3.6 shows the use of an infinite `for` loop.

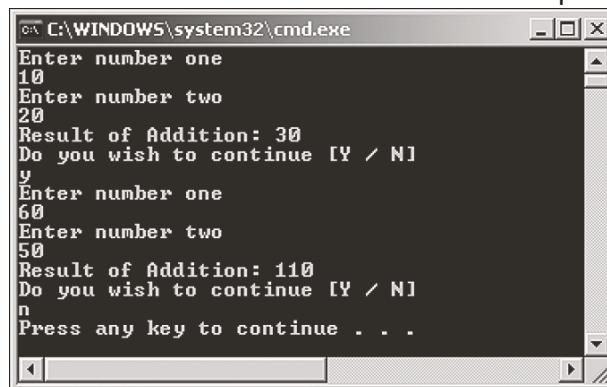


Figure 3.6: Infinite for Loop

3.2.9 *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 3.7 shows the output of using a `for` loop without a body.

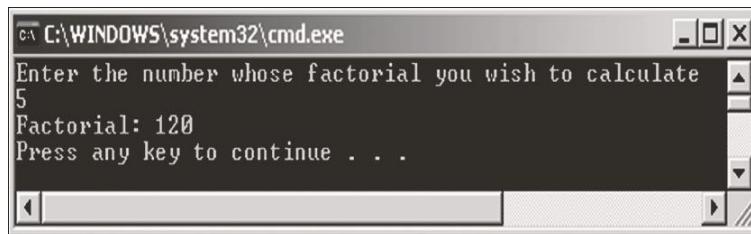


Figure 3.7: Output of Using for Loop Without Body

3.2.10 *foreach* Loop

The `foreach` loop is used to iterate over the elements of a collection, such as an array or a list of items.

Following is the syntax for declaring the `foreach` loop:

```
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.

3.3 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 must exit out of a loop prematurely and continue with the program.

In such cases, jump statements are used. Jump statements 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`

3.3.1 *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.

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.

3.3.2 *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.

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

3.3.3 *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.

Code Snippet 21 displays the output "Hello World" five times using the `goto` statement.

Code Snippet 21

```
int i = 0;
display:
Console.WriteLine("Hello World");
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

Code Snippet 22 demonstrates the use of `goto` statement to break out of a nested loop.

Code Snippet 22

```
class Factorial {
    static void Main(string[] args) {
        byte num = 0;
        while (true) {
            byte fact = 1;
            Console.Write("Please enter a number less than or
equal to 10: ");
            num = Convert.ToByte(Console.ReadLine());
            if (num < 0) {
```

```

        goto stop;
    }
    for (byte j = num; j > 0; j--) {
        if (j > 10)
        {
            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 10 are entered, the program keeps displaying their factorials. However, if a number greater than 10 is entered, the loop is exited.

Figure 3.8 shows the use of `goto` statement.

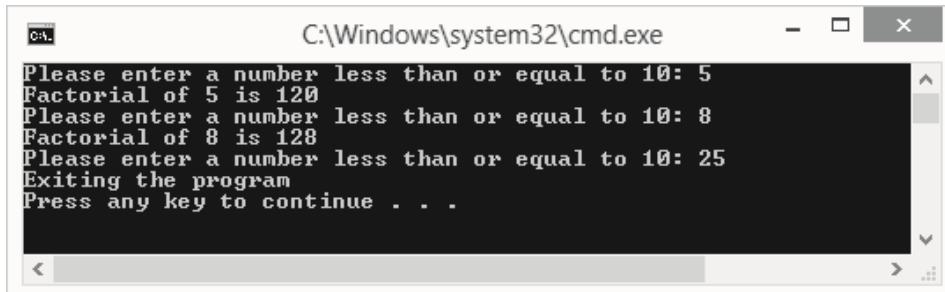


Figure 3.8: Use of `goto` Statement

3.3.4 `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.

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.

3.4 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.

3.4.1 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 3.9 shows an example to demonstrate the purpose of array.

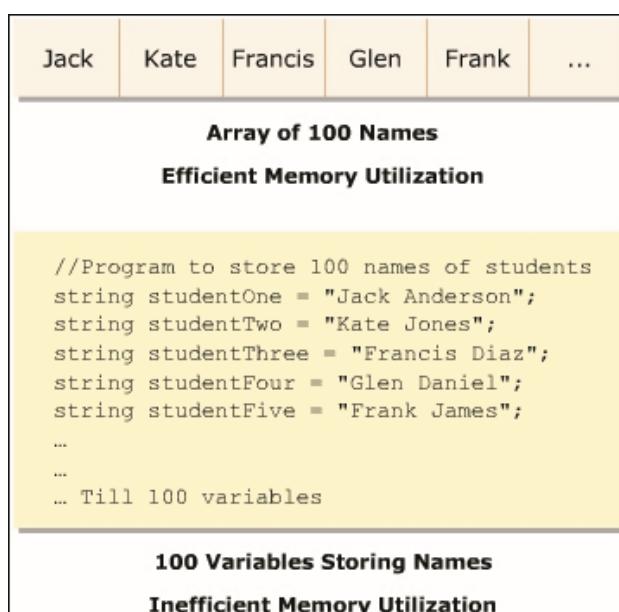


Figure 3.9: Purpose of Array

3.4.2 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 3.10 displays an example of the subscripts and elements in an array.

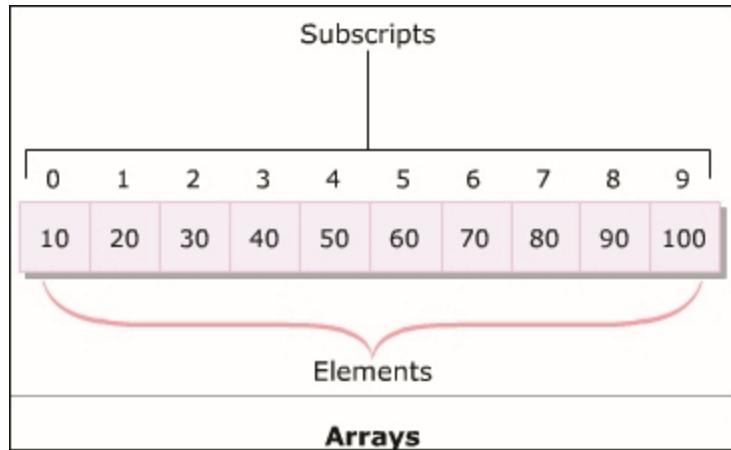


Figure 3.10: Subscripts and Elements in an Array

3.4.3 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.

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.

3.4.4 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 3.1 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 3.1: Default Values

Following syntax is used to create an array:

Syntax

```
arrayName = new type[size-value];
```

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.

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 25 creates an integer array which can have a maximum of five elements in it.

Code Snippet 25

<code>int[] number = new int[5];</code>

Code Snippet 26 initializes an array of type `string` that assigns names at appropriate index locations.

Code Snippet 26

```
string[] studNames = new string{"Allan", "Wilson", "James",
"Arnold"};
```

In Code Snippet 26, the string "Allan" is stored at subscript 0, "Wilson" at subscript 1, "James" at subscript 2 and "Arnold" at subscript 3.

Code Snippet 27 stores the string "Jack" as the name of the fifth enrolled student.

Code Snippet 27

```
studNames[4] = "Jack";
```

Code Snippet 28 demonstrates another approach for creating and initializing an array. An array called `count` is created and is assigned `int` values.

Code Snippet 28

```
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 28, 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
```

3.5 Direct Access to Array Elements

Array elements can be accessed by using an index. While defining the array, one must specify the size of an array. This is done so because based on size, an array is allocated continuous memory based on that.

The elements of an array are labeled incrementally, starting from 0 for the first element. In other words, the count of an array element starts from index value 0. For example, the seventh element of an array would be indexed at 6 and the third element of an array would be indexed at 4. By using square bracket operator, a specific element can be accessed surrounding the index with square brackets as shown in Code Snippet 29.

Code Snippet 29

```
using System;
namespace xyz {
    class ElementAccessDemo {
        public static void Main() {
            int[] scores;
            // allocating memory for scores array and initializing the array.
            scores = new int[] {79, 54, 63, 76, 22};
            Console.Write("Third element:");
            Console.Write(scores[2]);
        }
    }
}
```

Here, the output will be 63, because 63 is the third element and is accessed using index 2.

3.6 C# Array Length

The `Length` property of a C# array can be used to get the number of elements in a particular array. Code Snippet 30 demonstrates use of `Length` property for arrays of `int` and `string` type respectively.

Code Snippet 30

```
using System;
namespace xyz {
    class LenDemo {
        public static void Main() {
            int[] numbers;
            // allocating memory for days.
            numbers = new int[] {1,3,5,7,9};
            Console.Write("\nTotal Number of Elements in Integer Array: ");
            // using Length property
            Console.Write(numbers.Length);
            string[] animals;
            // allocating memory for animals
            animals = new string[] {"Dog","Cat","Rat"};
        }
    }
}
```

```

        Console.WriteLine("\nTotal Number of Elements in String Array: ");
        // using Length property
        Console.Write(animals.Length);
    }
}
}

```

The output of the code will be:

```

Total Number of Elements in Integer Array: 5
Total Number of Elements in String Array: 3

```

3.7 Types of Arrays

Based on how arrays store elements, arrays can be categorized into single-dimension and multi-dimension arrays.

3.7.1 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 five 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.

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 31 initializes a single-dimensional array to store the name of students.

Code Snippet 31

```

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 31, 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

3.7.2 Multi-dimensional Arrays

Consider a scenario where you must store the roll numbers (also called registration 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 require one array to store both roll numbers as well as marks.

A multi-dimensional array allows you to store the 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.

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 32 demonstrates the use of rectangular arrays.

Code Snippet 32

```
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 32, 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.

3.7.3 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 33 demonstrates the use of array references.

Code Snippet 33

```
using System;
class StudentReferences {
    public static void Main() {
        string[] classOne = { "Allan", "Chris", "Monica" };
        string[] classTwo = { "Katie", "Niel", "Mark" };
        Console.WriteLine("Students of Class I:\t\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 33, `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 3.11 displays the use of array references.

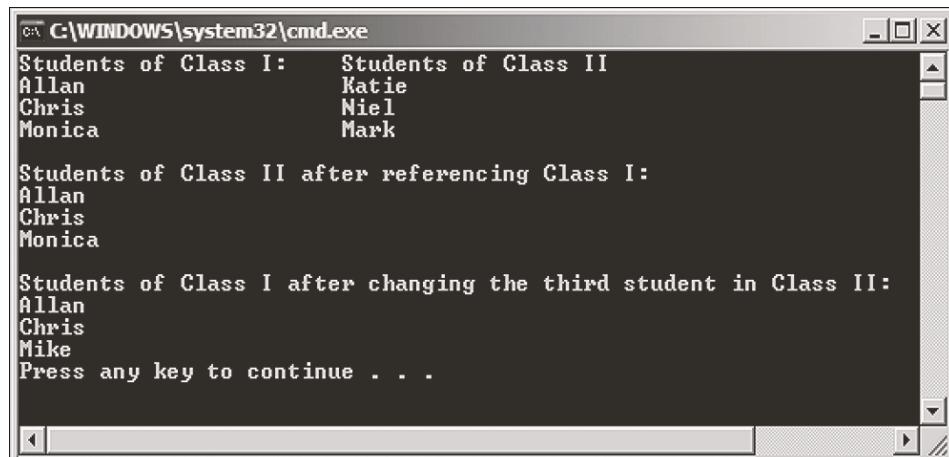


Figure 3.11: Use of Array References

3.7.4 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.

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 34 displays the name and the leave grant status of each student using the `foreach` loop.

Code Snippet 34

```
using System;
class Students{
    static void Main(string[] args) {
        string[] studentNames = new string[3] { "Ashley", "Joe",
        "Mikel" };
        foreach (string studName in studentNames) {
            Console.WriteLine("Congratulations!! " + studName + " you
have been granted an extra leave");
        }
    }
}
```

In Code Snippet 34, 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 thememory. During iteration, the elements in the list are in the read-only format. To change values in thearray, you must use `for` loop within `foreach` loop. The `foreach` loop executes once for each elementof the array.

3.7.5 Implicitly Typed Arrays

An implicitly typed array is an array in which the type of the array can be determined by the elements present in the array initializer. The major difference between a normal array and implicitly typed array is that in this array, you do not have to specify the data type. Features of implicitly typed arrays are as follows:

Syntax

```
var array_name = new[] { value1, value2, ..., valueN }
```

where,

 array_name is the name of the array
 value1, value2, ... are the values of the array.

Neither array name nor values require to have data type mentioned. The compiler will infer the type based on the data given. Refer to Code Snippet 35.

Code Snippet 35

```
using System;  
namespace xyz {  
    class ArrayDemo {  
        public static void Main(){  
            var StrikeRate = new[] {150, 196, 175, 224 };  
            var flags = new[] { true, false, false };  
            //Display the contents of the arrays  
            foreach(int i in StrikeRate)  
                Console.WriteLine("StrikeRate: {0}", i);  
            foreach(bool i in flags)  
                Console.WriteLine("Flags: {0}", i);  
        }  
    }  
}
```

When this code is compiled, the compiler infers `StrikeRate` as an `int` array based on the integer data given for the array. Similarly, the compiler infers the data for `flags` as `bool` based on the data given. To display the contents of both arrays, `foreach` loops can be used.

An important point that must be followed while using implicitly typed arrays is that the data elements of the array should be of same type.

3.8 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/she 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#. It 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.

3.8.1 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 3.2 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 3.2: Properties of Array Class

Methods

The `Array` class allows you to clear, copy, search, and sort the elements declared in the array.

Table 3.3 displays the most commonly used methods in the `Array` class.

Methods	Descriptions
Clear	Deletes all elements within the array and sets the size of the array to 0.
CopyTo	Copies all elements of the current single-dimensional array to another single-dimensional array starting from the specified index position.
GetLength	Returns number of elements in an array.
GetLowerBound	Returns the lower bound of an array.
GetUpperBound	Returns the upper bound of an array.
Initialize	Initializes each element of the array by calling the default constructor of the <code>Array</code> class.
Sort	Sorts the elements in the single-dimensional array.
SetValue	Sets the specified value at the specified index position in the array.
GetValue	Gets the specified value from the specified index position in the array.

Table 3.3: Methods in Array Class

3.8.2 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 must invoke the `CreateInstance()` method that is accessed by specifying the class name because the method is declared as static.

Code Snippet 36 creates an array of length five using the `Array` class and stores different subject names.

Code Snippet 36

```
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 36, 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 indifferent 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.

3.9 Summary

- C# supports if..else, if..else if, nested if, and switch...case selection constructs.
- C# supports while, do-while, for, and foreach loop constructs.
- Loop control variables are often created for loops such as the for loop.
- C# supports jump statements such as break, continue, goto, and return.
- Arrays are a collection of values of the same data type.
- C# supports zero-based index feature for arrays.
- 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.

3.10 Check Your Progress

1. Arrange the steps in sequence to achieve the output as "10 9 8 7 6 5".

(A)	while (num-- > 0);
(B)	if (num >= 5)
(C)	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

2. Which of these statements about 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 if statement is skipped if condition in the if statement returns false
(C)	The if..else if construct checks multiple conditions and executes an appropriate block of code for each condition
(D)	The inner if statements of the nested if construct control the execution of the outer if statement

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

3. John is a C# programmer trying to display the output 311. Which of the following codes will help him to achieve this?

(A)	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; }
-----	--

(B)

```
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)

```
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)

```
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--;
}
```

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

4. Peter is trying to display the output "Welcome to guest". Which of the following codes will help him to achieve this?

(A)	string login = "guest"; if (login == "admin") { Console.WriteLine("Welcome to admin"); } else if(login == "guest") { Console.WriteLine("Welcome to guest"); }
(B)	string login = "guest"; if (login = "admin") { Console.WriteLine("Welcome to admin"); } else if (login = "guest") { Console.WriteLine("Welcome to guest"); }
(C)	string login = "guest"; if (login = "admin") { Console.WriteLine("Welcome to admin"); } elseif (login == "guest") { Console.WriteLine("Welcome to guest"); }
(D)	string login = "guest"; if (login = "admin") { Console.WriteLine("Welcome to admin"); } elseif (login == "guest"); { Console.WriteLine("Welcome to guest"); }

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

5. Cathy is trying to display the output "0 1 1 2 3 5 8 13". Which of the following codes will help her to achieve this?

(A)	<pre>int firstNum = 0; int secondNum = 1; int result; Console.WriteLine("{0} ", firstNum); Console.WriteLine("{0} ", secondNum); for(result = 0; result < 8;){ result = firstNum + secondNum; Console.WriteLine("{0} ", result); firstNum = secondNum; secondNum = result; }</pre>
(B)	<pre>int firstNum = 0; int secondNum = 1; int result; for(result = 0; result < 10;){ result = firstNum + secondNum; firstNum = secondNum; secondNum = result; Console.WriteLine("{0} ", result); }</pre>
(C)	<pre>int firstNum = 0; int secondNum = 1; int result; Console.WriteLine("{0} ", firstNum); Console.WriteLine("{0} ", secondNum); for(result = 0; result <= 10;{ Console.WriteLine("{0} ", result); result = firstNum + secondNum; firstNum = secondNum; secondNum = result; }</pre>
(D)	<pre>int firstNum = 0; int secondNum = 1; 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; }</pre>

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

3.10.1 Answers

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

Try It Yourself

1. Write a C# program to manage a list of students and their scores. Create a class Student with following members:

Name (string): The name of the student

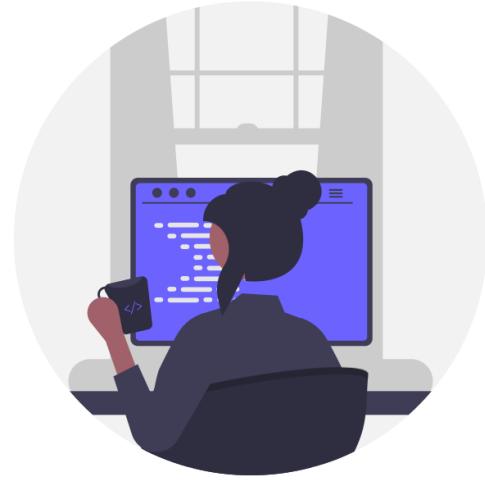
Scores (int[]): An array of integer scores for the student

The Student class should have a constructor that takes the student's name and an array of scores as parameters. The constructor should initialize the Name and Scores members.

Next, create a program that allows the user to:

- a) Enter the names and scores of several students.
- b) Display the names and average scores of all students.

2. Similarly, create a C# program to manage Players and Tournaments for a soccer match.



Session 4

Classes and Methods in C#

Welcome to the Session, **Classes and Methods in C#**.

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 a class 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

4.1 Object-Oriented Programming (OOP)

Programming languages have always been designed based on two fundamental concepts, namely, 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.

4.1.1 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 do not have to 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 the class has multiple methods declared with the same name but with different parameters and different behavior.

4.2 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.

4.2.1 Objects

An object is a tangible entity such as a car, a table, or a briefcase. Each 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 4.1 shows an example of an object with identity, state, and behavior.

Car Class
Characteristics
➤ Make ➤ Model
Behavior
➤ Driving ➤ Accelerating ➤ Braking

Figure 4.1: Object Identity, State, and Behavior

An object stores its identity and state in fields (also called variables) and exposes its behavior through methods.

4.2.2 Classes

Several objects have a common state and behavior and thus, can be grouped into 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 4.2 displays the class Car.

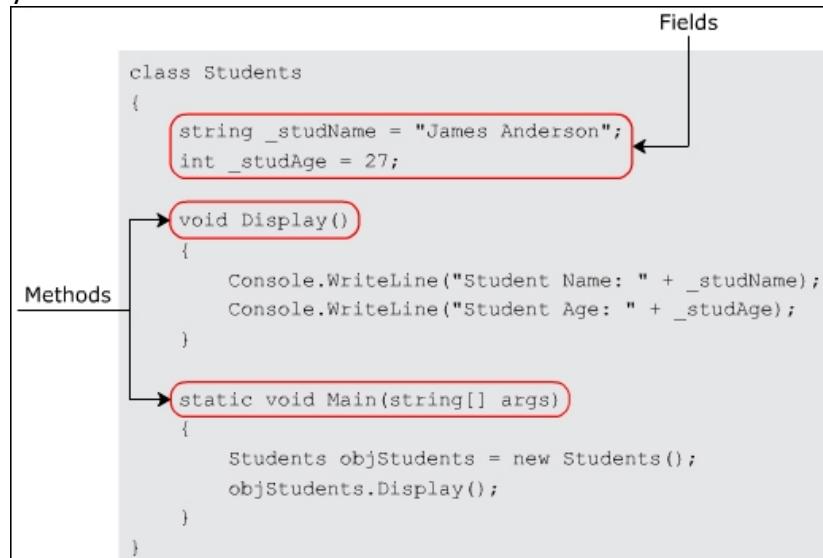


Figure 4.2: Class Car

4.2.3 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 such as 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.

Following syntax is used to declare a class:

Syntax

```
class <ClassName> {  
    // class members  
}
```

where,

ClassName: Specifies the name of the class.

4.2.4 Guidelines for Naming Classes

There are certain conventions to be followed for class names while creating a class. These conventions help the developer 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**.

Figure 4.3 displays some valid class names and invalid class names.

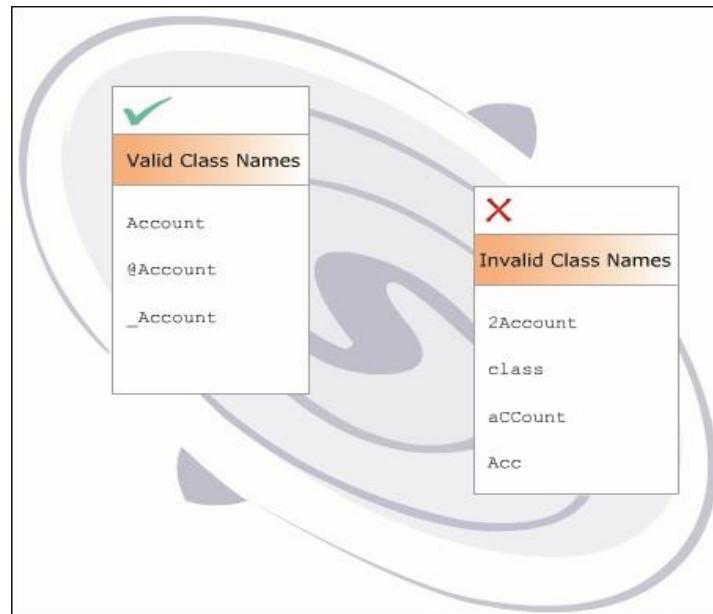


Figure 4.3: Examples of Valid Class Names and Invalid Class Names

4.2.5 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.

Each 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`.

4.2.6 Instantiating Objects

When the developer creates 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.

Following syntax is used to instantiate an object with `new`:

Syntax

```
<ClassName> <objectName> = new <ClassName>();
```

where,

`ClassName`: Specifies the name of the class.

`objectName`: Specifies the name of the object.

Figure 4.4 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 4.4: 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.

4.3 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`.

4.3.1 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 `Add`, `Add Sum()`, and `int()`.

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.

4.3.2 Invoking Methods

The developer 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 4.5 displays how a method invocation or call is stored in the stack in memory and how a method body is defined.

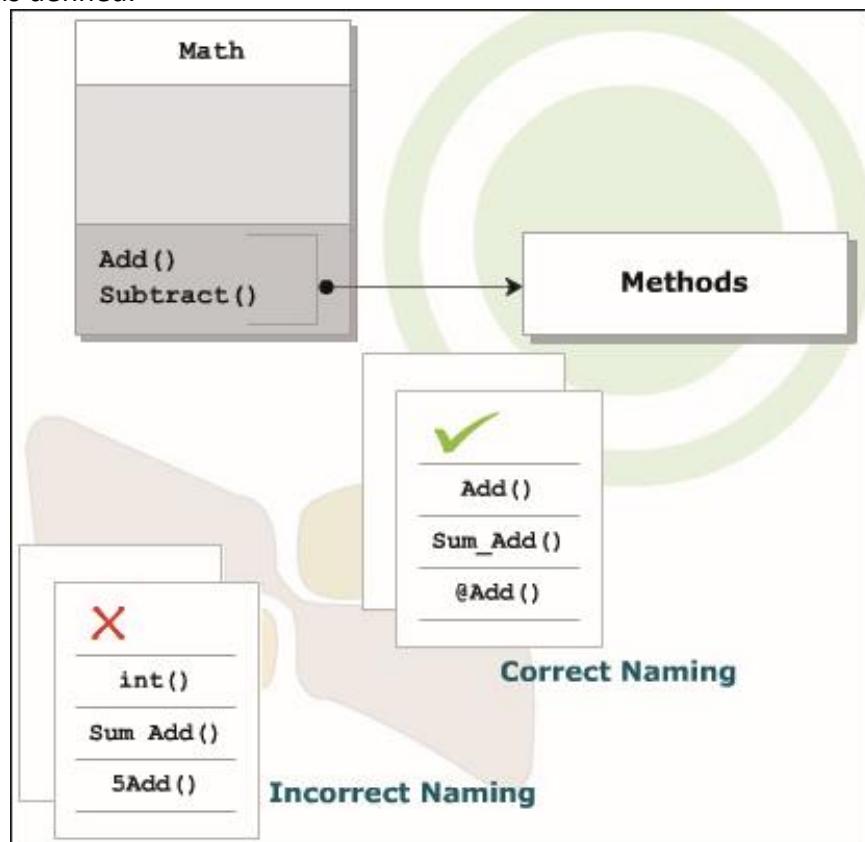


Figure 4.5: Method Invocation

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

4.3.3 Method Parameters and Arguments

Variables included in a method definition are called parameters. When the method is called, the data that the developer sends 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 4.6 shows an example of parameters and arguments.

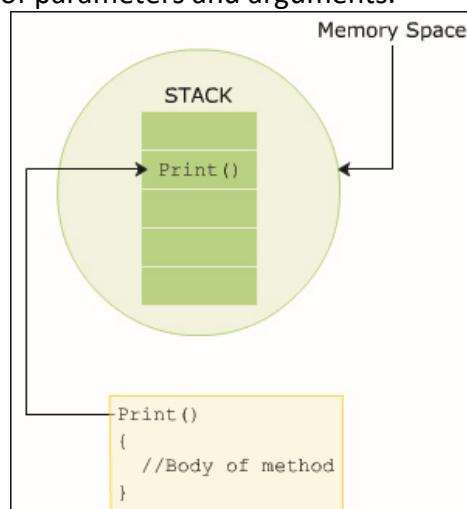


Figure 4.6: Parameters and Arguments

4.3.4 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 the developer does 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 `printName()` 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  
First Name = Henry, Last Name = Parker  
First Name = Henry, Last Name = Parker  
First Name = Henry, Last Name = Parker
```

Note: In a method call, the developer 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 {  
    void Count(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 is printed on the console.

Output:

```
Welcome User!  
Hello user!
```

4.4 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.

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 necessity 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;
    }
    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
```

4.5 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.

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.

Figure 4.7 displays invoking a static method.

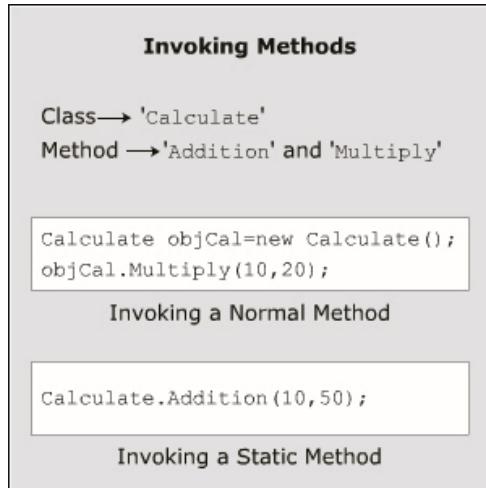


Figure 4.7: Invoking a Static Method

4.6 Static Variables

In addition to static methods, one 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 4.8 displays an example of 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 4.8: Static Variables

4.7 Access Modifiers

Object-oriented programming enables restricting 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 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 Employee{  
    // 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 4.9 displays various accessibility levels.

Accessibility Levels			
Access Modifiers	Applicable to the Application	Applicable to the Current Class	Applicable to the Derived Class
	✓	✓	✓
	✗	✓	✗
	✗	✓	✓
	✗	✓	✓

Figure 4.9: 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.

4.8 `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.

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,
```

parameter 1...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 4.10 displays the use of `ref` keyword.

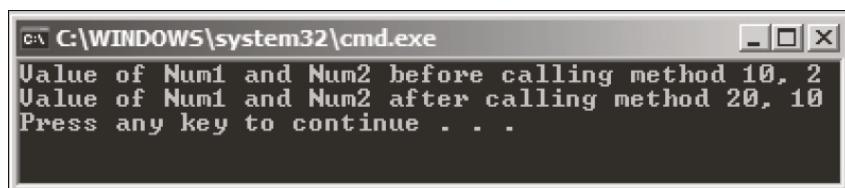


Figure 4.10: Use of ref Keyword

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.

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 4.11 displays the use of `out` keyword.

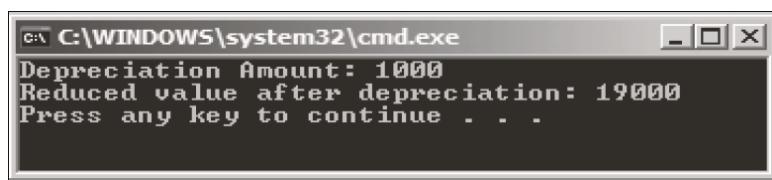


Figure 4.11: Use of out Keyword

4.9 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, so if there is a delivery of fresh vegetables for Mr. Peter, it is forwarded to the chef. Similarly, if there is a delivery of electric bulbs and wires for 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.

4.9.1 Method Overloading in C#

In object-oriented programming, each 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 4.12 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 4.12: 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.

Note: The signatures of two methods are said to be the same if all the three conditions - number of parameters passed to the method, parameter types and order in which parameters are written - are the same. The return type of a method is not a part of its signature.

4.9.2 Guidelines and Restrictions

While overloading methods in a program, the developer 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.

4.9.3 *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. The developer 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 4.13 displays the use of `this` keyword.

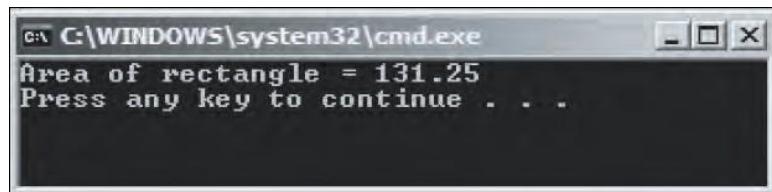


Figure 4.13: Use of this Keyword

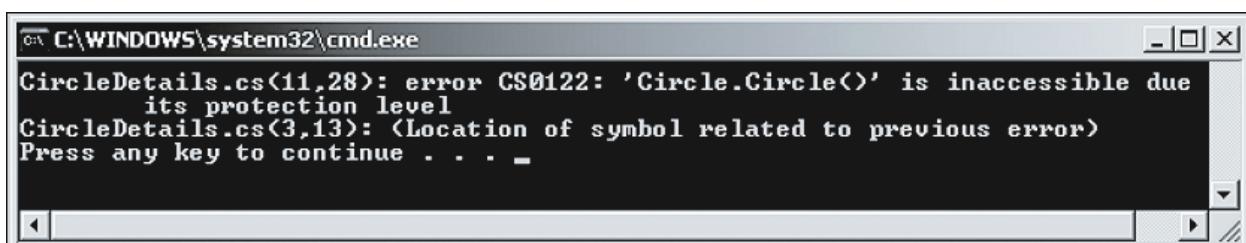
4.10 Constructors and Destructors

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.

4.10.1 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 to be performed in an application such as 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 4.14 shows the constructor declaration.



The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The error message displayed is:

```
CircleDetails.cs(11,28): error CS0122: 'Circle.Circle()' is inaccessible due
its protection level
CircleDetails.cs(3,13): <Location of symbol related to previous error>
Press any key to continue . . .
```

Figure 4.14: 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	private	protected	internal
Specifies that the constructor will be called whenever a class is instantiated. This instantiation can be done from anywhere and from any assembly.	Specifies that this constructor cannot be invoked by an instance of a class.	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.	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. 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#.

4.10.2 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 the developer defines a constructor in the class, the default constructor is no longer used.

4.10.3 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. Only one static constructor is allowed in a 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.

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
```

4.10.4 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. Each constructor has a signature similar to that of a method. The developer 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 required 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

```

4.10.5 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. The developer 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 required to be used. However, when this will happen cannot be determined and hence, the developer has no control on when the destructor is going to be executed.

4.11 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.

4.12 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 <code>class</code> keyword
(E)	Creating objects involves the use of the <code>new</code> keyword

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

2. Match the descriptions against their corresponding OOP terms.

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 <code>static</code> 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. Peter is trying to display the area of a rectangle. Which of the following codes will help him 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) {
        Dimension.SetDimension(20.1, 18.3);
        Console.WriteLine("Area of Rectangle: " +
            Dimensions.Area());
    }
}
```

(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; } 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>

```

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)
    {
        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 descriptions against their corresponding access modifiers.

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)

4.12.1 *Answers*

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

Try It Yourself

1. Create a C# program that demonstrates the functionality of a BankAccount class. The program should allow the user to create accounts, deposit, withdraw, and view account details. Use a console-based menu system to interact with the user, for example:
 - Choose 1 to Create Account
 - Choose 2 to Deposit Amount
2. Create a C# program using classes and methods and so on to model a simple library system.

The program should be able to:

- Add books to the library.
- Display a list of available books.
- Allow users to check out books.
- Allow users to return books.
- Display a list of books checked out by a particular user.
- Display a list of all books in the library, including their availability status.



Session 5

Inheritance and Polymorphism

Welcome to the Session, **Inheritance and Polymorphism**.

Inheritance is the process of creating a new class from an existing class. Inheritance allows the developer 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

5.1 *Inheritance*

A programmer does not always have 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.

5.1.1 *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 the developer 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 5.1 illustrates this example.

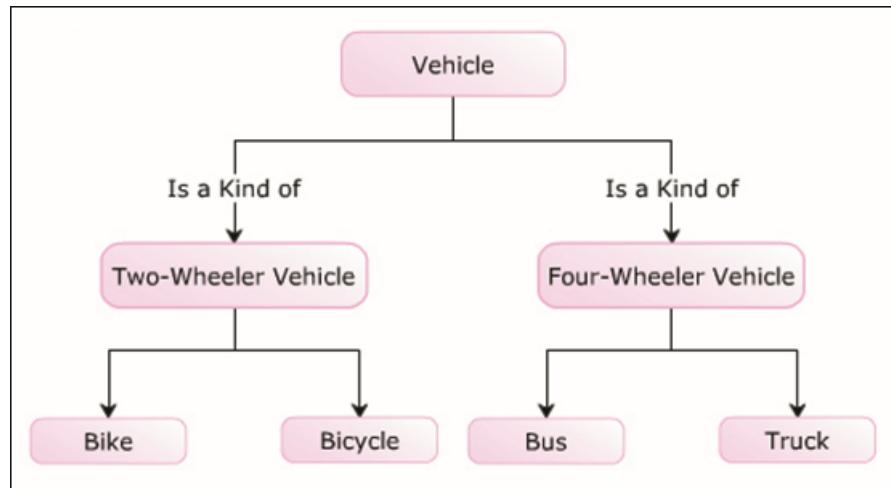


Figure 5.1: Example of Inheritance

5.1.2 Purpose

The purpose of inheritance is to reuse common methods and attributes among classes without recreating them. Reusability of a code enables the developer 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 must be created, it can be done based on **Animal** because cat is also an animal. Thus, the developer can reuse the code from the previously defined class.

Apart from reusability, inheritance is widely used for:

Generalization

Inheritance allows the developer 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 the developer 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 the developer to extend the functionalities of a derived class by creating more methods and attributes that are not present in the base class. It allows the developer to provide additional features to the existing derived class without modifying the existing code.

Figure 5.2 displays a real-world example demonstrating the purpose of inheritance.

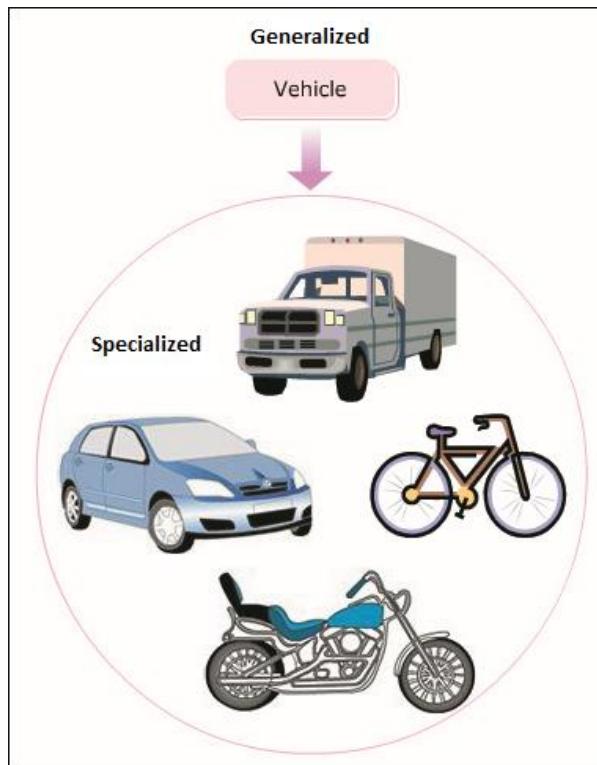


Figure 5.2: Purpose of Inheritance

5.1.3 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 5.3 depicts multi-level hierarchy of related classes.

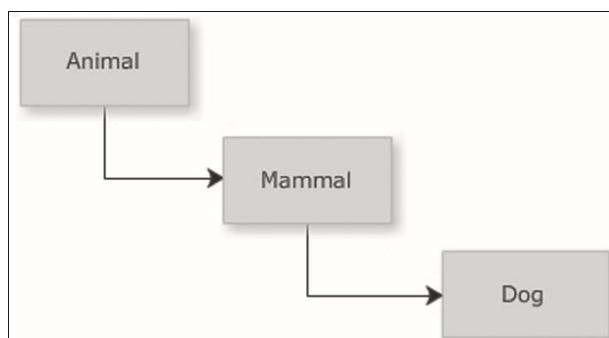


Figure 5.3: Multi-level Hierarchy

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 the young ones.
Dog Barks.
```

5.1.4 Implementing Inheritance

The syntax to derive a class from another class is quite simple in C#. The developer must 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.

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.

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.

5.1.5 *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 5.4 displays an example of using the

`protected` access modifier.

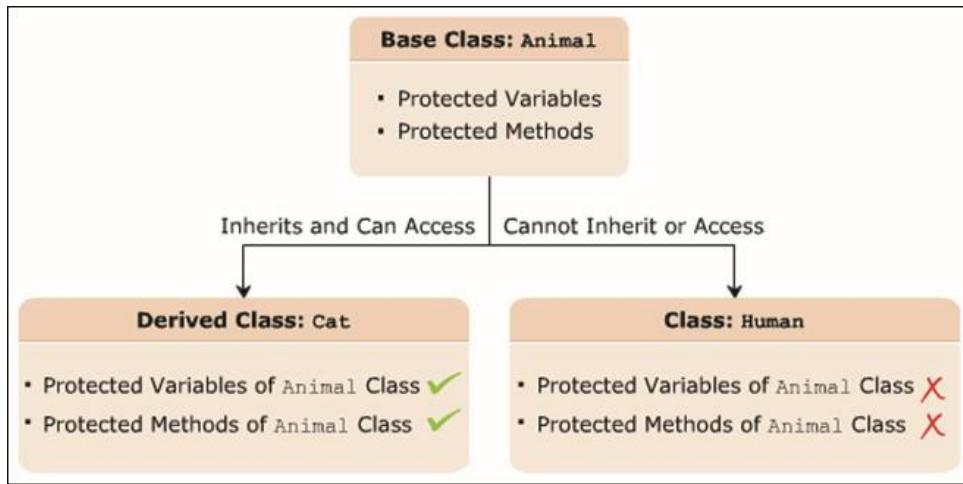


Figure 5.4: protected Access Modifier

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.

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 to 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.
```

5.1.6 base Keyword

The **base** keyword allows the developer to access the variables and methods of the base class from the derived class. When the developer inherits a class, the methods and variables defined in the base class can be re-declared in the derived class. Now, when the developer invokes methods or access variables, the derived class data members are invoked and not data members of the base class. In such situations, the developer can access the base class members using the **base** keyword.

The developer cannot use the **base** keyword for invoking the static methods of the base class.

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.
- <return type>: 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 5.5 displays an example of using the `base` keyword.

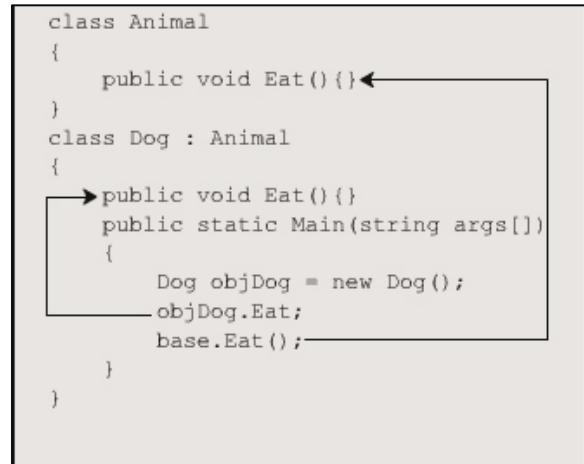


Figure 5.5: `base` Keyword

5.1.7 `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 the developer 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 the developer can access these is by using the `base` keyword.

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
<code>Employees objEmp = new Employees();</code>

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
<pre>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(); } }</pre>

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 the developer 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

5.1.8 Constructor Inheritance

In C#, the developer cannot inherit constructors similar to how the developer inherits methods. However, the developer 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, the developer can explicitly invoke the base class constructor by using the `base` keyword in the derived class constructor declaration. The `base` keyword allows the developer to pass parameters to the constructor.

Figure 5.6 displays an example of constructor inheritance.

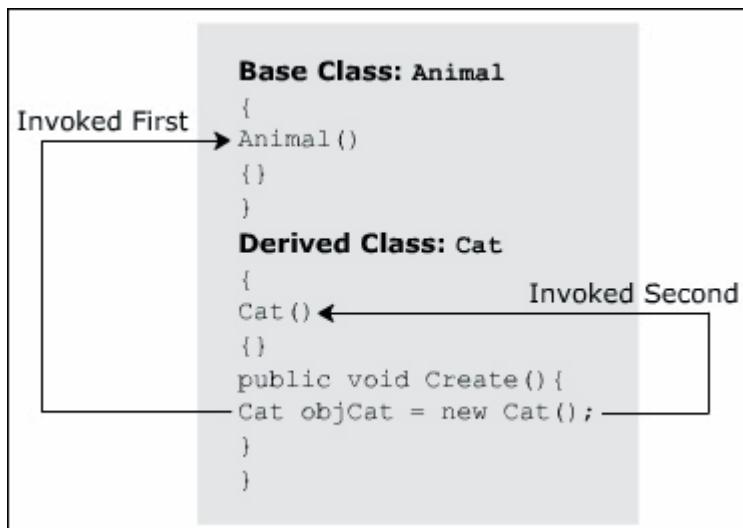


Figure 5.6: Constructor Inheritance

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

```

5.1.9 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: Chassies

5.2 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 5.7 displays the method overriding.

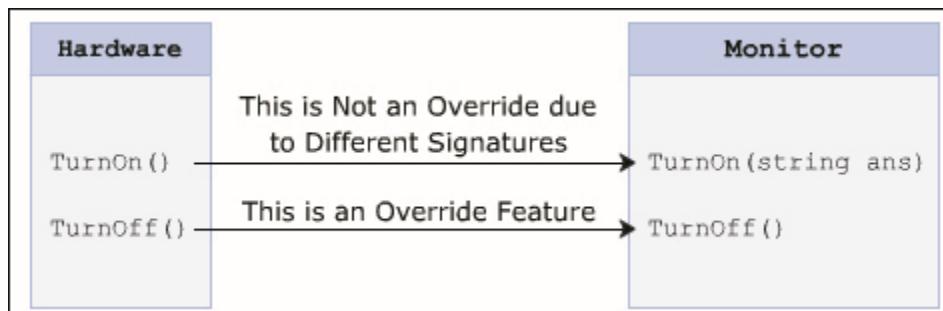


Figure 5.7: Method Overriding

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.

5.2.1 virtual and override Keywords

The developer can override a base class method in the derived class using appropriate C# keywords such as **virtual** and **override**. If the developer wants to override a particular method of the base class in the derived class, he/she must 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, the developer must 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.

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.

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.

5.2.2 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 the developer to access the new method but not the original base class method. Sometimes, the developer might want both, the base class method as well as the derived class method, to be executed. In this case, the developer can create an instance of the base class, which allows the developer 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
```

5.3 Sealed Classes

A sealed class is a class that prevents inheritance. The developer 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.

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 must 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 5.8.

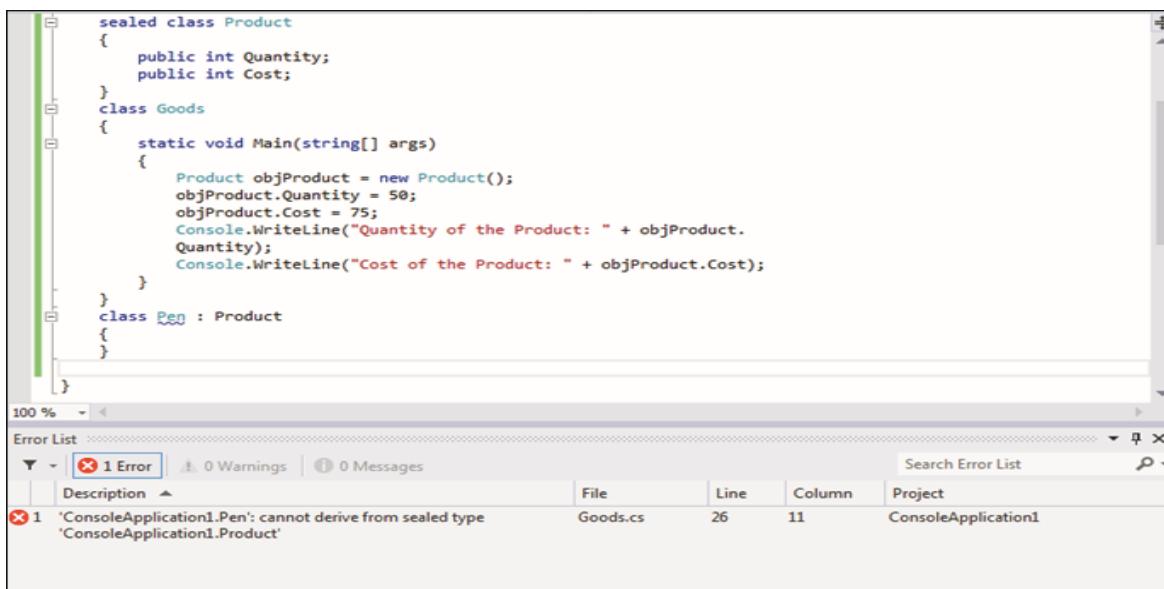


Figure 5.8: Compiler Error

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.

5.3.1 Purpose of Sealed Classes

Consider a class named **SystemInformation** that consists of critical methods that affect the working of the operating system. the developer might not want any third party to inherit the class **SystemInformation** and override its methods, thus, causing security and copyright issues. Here, the developer can declare the **SystemInformation** class as sealed to prevent any change in its variables and methods.

5.3.2 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 the developer wants to prevent any third party from modifying the developer class

5.3.3 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.

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

```
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 5.9.

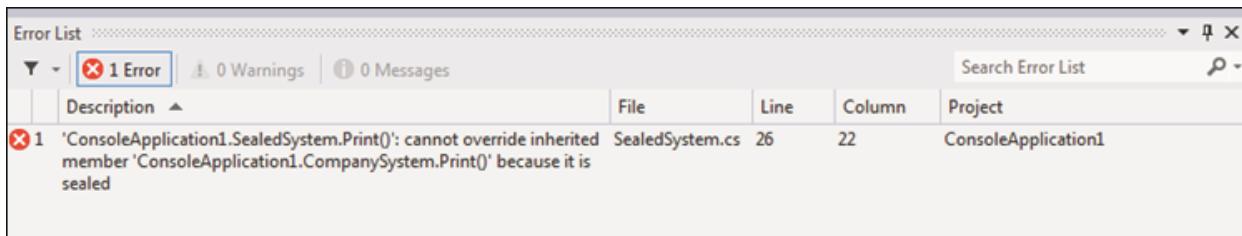


Figure 5.9: Error Due to Sealed Method

5.4 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 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.

5.4.1 Implementation

The developer can implement polymorphism in C# through method overloading and method overriding. The developer 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 5.10 displays the implementation of polymorphism.

```
• 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 5.10: Implementation of Polymorphism

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
```

5.4.2 Compile-time and Runtime Polymorphism

Polymorphism can be broadly classified into two categories, compile-time polymorphism and runtime polymorphism. Table 5.1 differentiates between compile-time and runtime polymorphisms.

Compile-time Polymorphism	Runtime 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 runtime 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 5.1: Difference between Compile-time and Runtime Polymorphism

Code Snippet 13 demonstrates the implementation of runtime polymorphism.

Code Snippet 13

```
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 runtime 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
```

5.5 Summary

- Inheritance allows the developer 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.
- The developer can declare a sealed class in C# by using the sealed keyword.
- Polymorphism is the ability of an entity to exist in two forms, compile-time polymorphism and runtime polymorphism.

5.6 Check Your Progress

1. Match the terms comprising keywords and modifiers related to inheritance against their corresponding descriptions.

Description		Term	
(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 inherited methods of derived class	(4)	base keyword
(E)	Is used to access 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. Peter is 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 Peter to achieve this?

```
(A) 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;
    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();
}
```

```
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";
(B) 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();
}
}
```

```
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";
(C) 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();
}
}
```

```

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();
    }
}

```

(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 method defined in base class is preceded by <code>virtual</code> keyword
(C)	A base class method is preceded with <code>override</code> keyword in order to override it in derived class
(D)	A method is known as overridden derived method when it is overridden in 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. John is trying to display the name, ID, designation, and salary of an employee. Which of the following codes will help John 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() { 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) { Salary objSalary = new Salary(); objSalary.Print(); } } }</pre>

```

(C) 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();
    }
}

```

```

(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 <code>seal</code> 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 <code>virtual</code> 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)	Runtime polymorphism allows the developer to execute overridden methods
(E)	Runtime polymorphism supports methods with different signatures

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

5.6.1 Answers

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

Try It Yourself

1. Create an animal hierarchy using inheritance and demonstrating polymorphism in C#.

Begin by creating a base class named `Animal` with following properties and methods:

Properties: `Name (string)`, `Species (string)`

Methods: `MakeSound ()` - a virtual method that prints the sound the animal makes.

Create two derived classes, `Dog` and `Cat`, that inherit from the `Animal` base class. Implement the `MakeSound ()` method for each of these classes to output the respective sound that a dog and a cat makes.

In your `Main ()` method, create instances of both the `Dog` and `Cat` classes. Assign them names and display their species and the sounds they make.

Demonstrate polymorphism by creating an array or a list of `Animal` objects and adding instances of both `Dog` and `Cat`. Iterate through the collection and call the `MakeSound ()` method for each animal.

Your solution should illustrate inheritance, polymorphism, and method overriding in C# by modeling an animal hierarchy and showcasing different sounds made by dogs and cats, as well as how polymorphism allows you to work with objects of the base class.

2. Create a simple program to model different types of vehicles. Use inheritance and polymorphism to achieve this.

Define a base class `Vehicle` with following properties:

`Brand (string)`: The brand or manufacturer of the vehicle.

`Model (string)`: The model name of the vehicle.

`Year (int)`: The year the vehicle was manufactured.

Create two derived classes: `Car` and `Motorcycle`. These classes should inherit from the `Vehicle` base class and have additional properties specific to each type of vehicle. For example:

`Car` should have a property `NumberOfDoors (int)`.

`Motorcycle` should have a property `EngineDisplacement (double)`.

Implement a method in the base class `Vehicle` called `DisplayDetails` that prints out the details of the vehicle, including brand, model, and year.

Override the `DisplayDetails` method in both `Car` and `Motorcycle` classes to include the specific properties of each type of vehicle.

In your `Main` method, create instances of both `Car` and `Motorcycle`, populate them with data, and use the `DisplayDetails` method to display the details of each vehicle.

Demonstrate polymorphism by creating an array of vehicle objects and adding instances of both `Car` and `Motorcycle` to the array. Iterate through the array and call the `DisplayDetails` method for each object.

Your solution should showcase inheritance and polymorphism in C# by modeling different types of vehicles and displaying their details in a unified way using the base class method.



Session 6

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, similar to 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

6.1 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).

6.1.1 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 must

be different for different animals even though they inherit the same base class. These differences can be incorporated using abstract classes.

Figure 6.1 displays an example of abstract class and subclasses.

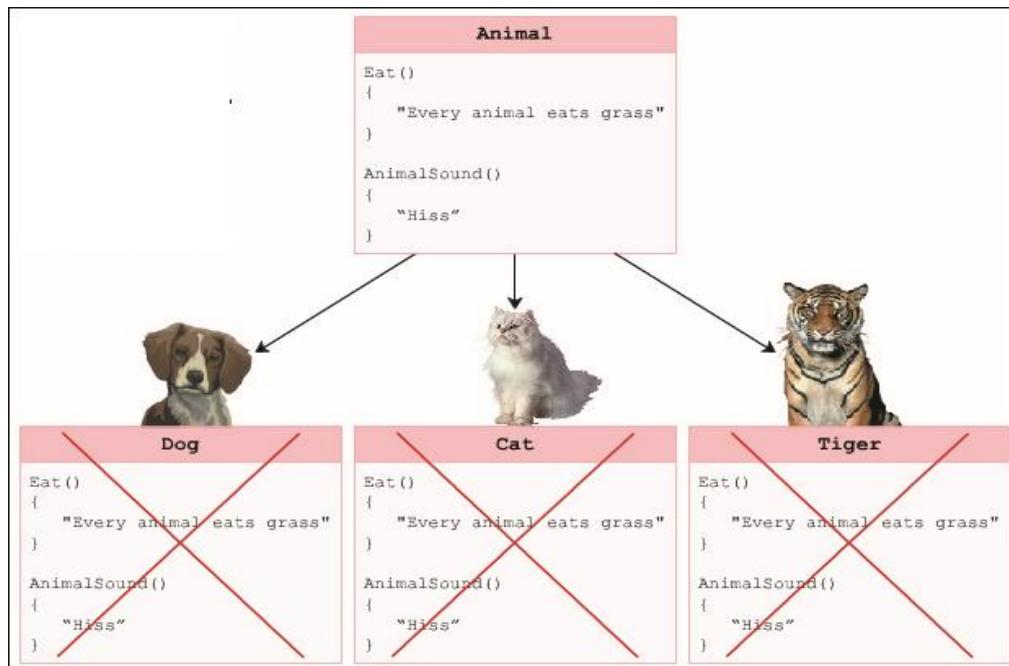


Figure 6.1: Abstract and Sub Classes

6.1.2 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 6.2 displays the contents of an abstract class.

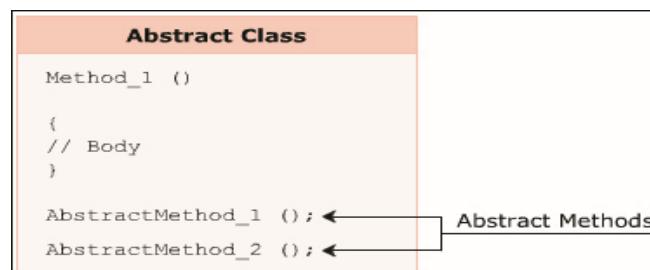


Figure 6.2: Abstract Class

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.

6.1.3 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 6.3 displays an example of inheriting an abstract class.

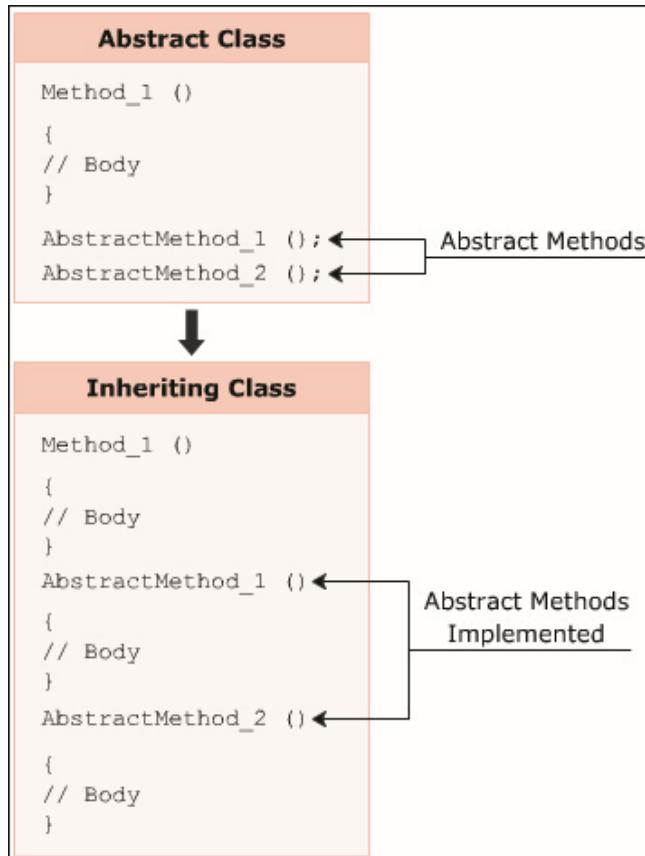


Figure 6.3: Inheriting an Abstract Class

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

```
public abstract class Animal {
    public void Eat() {
        Console.WriteLine("Every animal eats food in order to
survive");
    }

    public abstract void AnimalSound();
}
```

```

}

public 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

```

6.1.4 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.

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. Note that this program will not run as there is no `Main` in it.

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.");
        }
    }
}
```

6.1.5 Abstract Methods

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 6.4 displays an example of abstract methods.

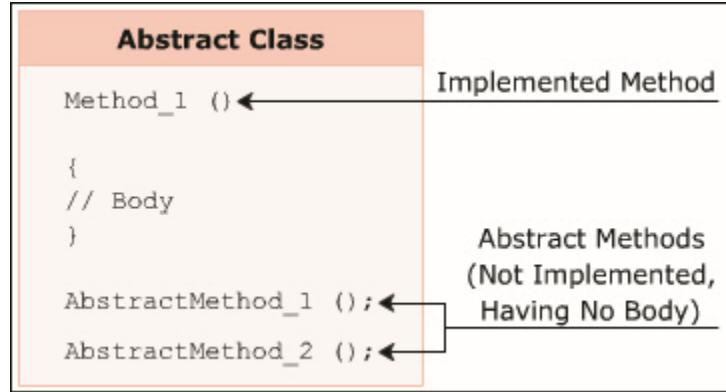


Figure 6.4: Abstract Methods

6.2 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.

6.2.1 Purpose

Consider a class **Dog** that must 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 6.5 displays an example of the subclasses in C#.

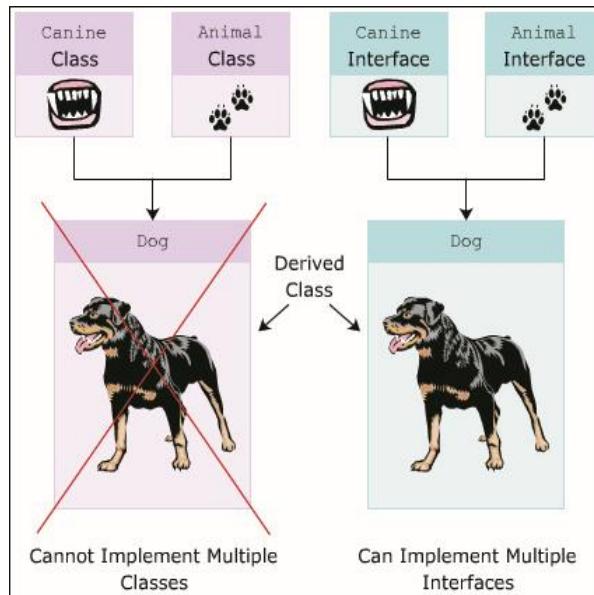


Figure 6.5: Subclasses in C#

6.2.2 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 6.6 displays an example of an interface.

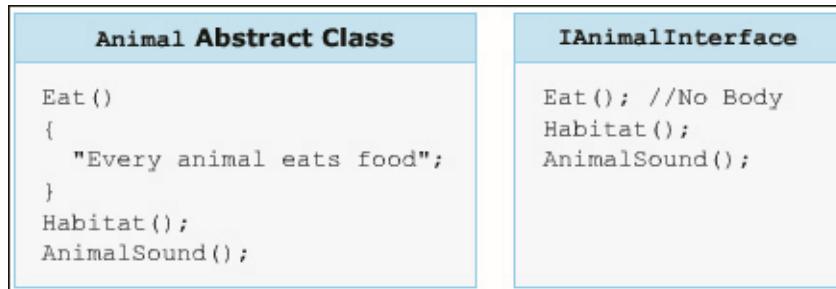


Figure 6.6: Interfaces

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.

6.2.3 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, the developer must 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 6.7 displays the implementation of an interface.

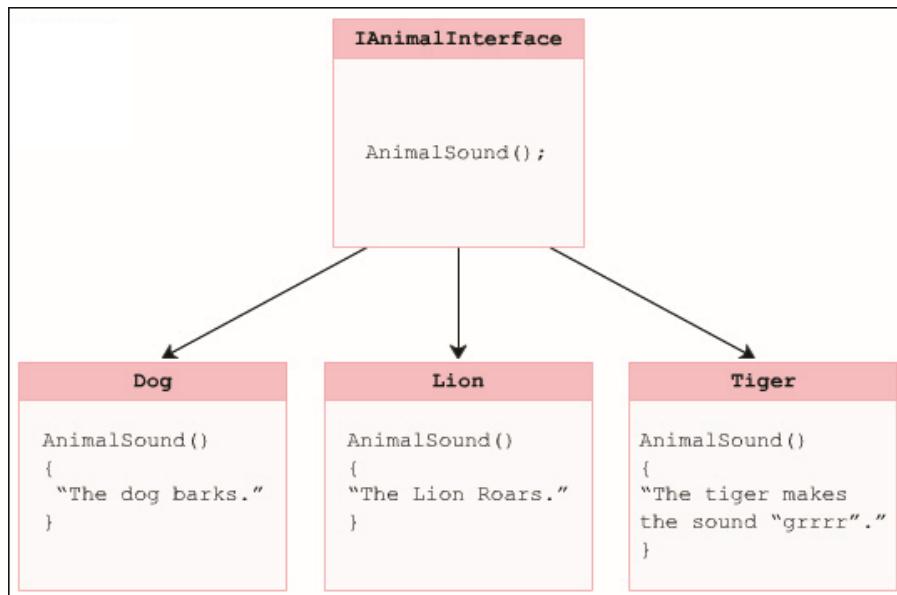


Figure 6.7: Implementation of Interface

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

6.2.4 Interfaces and Multiple Inheritance

Multiple interfaces can be implemented in a single class. This implementation provides the functionality of multiple inheritance. The developer 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 6.8 displays the concept of multiple inheritance using interfaces.

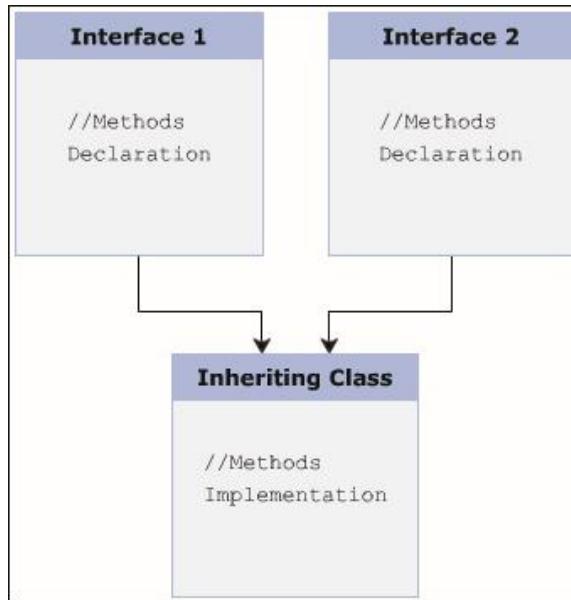


Figure 6.8: Multiple Inheritance Using Interfaces

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.

6.2.5 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, the developer cannot mention modifiers such as **abstract**, **virtual**, **override**, or **new**.

Figure 6.9 displays the explicit implementation of interface.

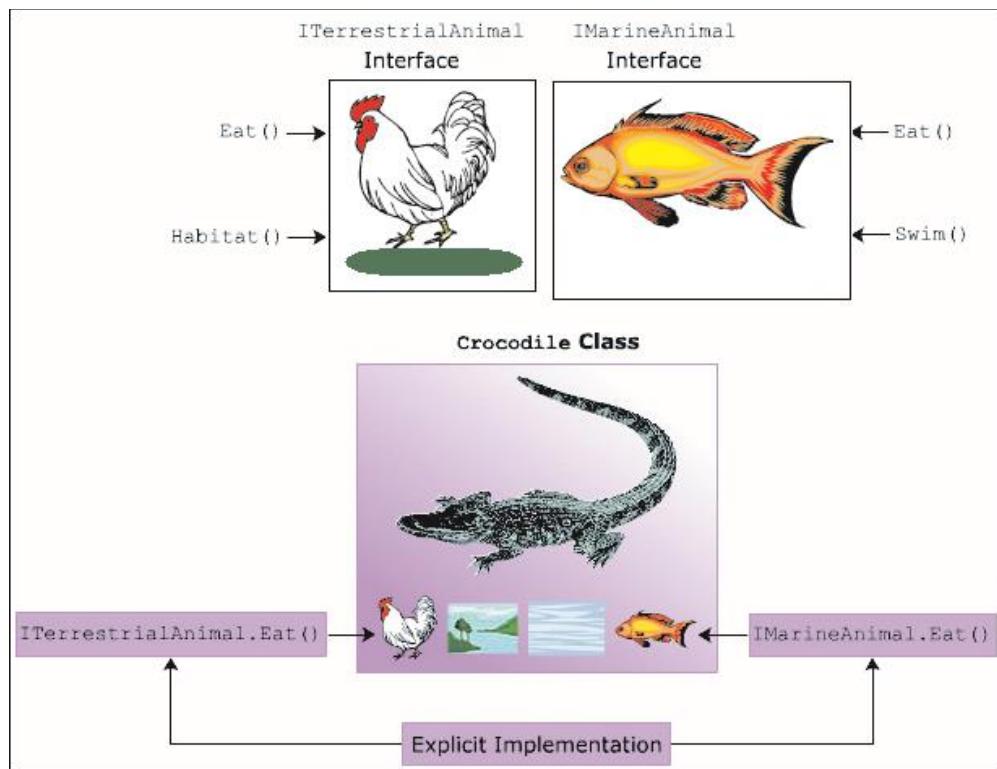


Figure 6.9: Explicit Implementation of Interface

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
```

6.2.6 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 6.10 displays an example of interface inheritance.

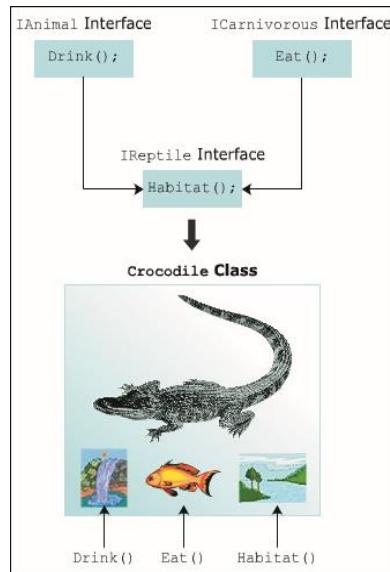


Figure 6.10: Interface Inheritance

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 fresh water");
    }

    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. 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 fresh water

```

6.2.7 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 {
    void Area();
}

class Circle : IMath {
    public const float PI = 3.14F;
    protected float Radius;
    protected double AreaOfCircle;
    public virtual void Area() {
        AreaOfCircle = (PI * Radius * Radius);
    }
}
class Sphere : Circle {
    double _areaOfSphere;
    public override void Area() {
        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 6.11 displays the output of re-implementation of an interface.

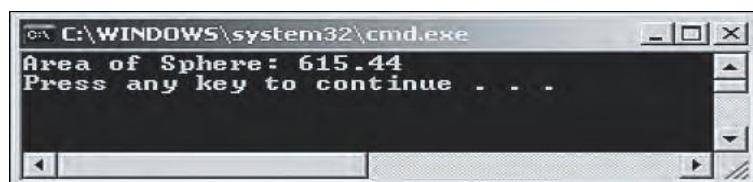


Figure 6.11: Re-implementation of an Interface

6.2.8 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 6.12 displays the output of the example using `is` operator.

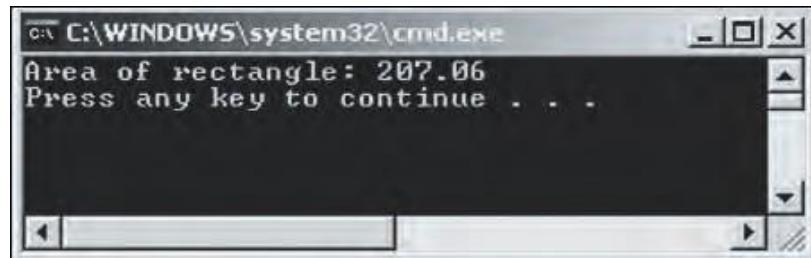


Figure 6.12: Example of is Operator

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 interface `IGet` declares a method `Display()`. The class `Employee` implements interface

`ISet` and then, implements the method declared within `ISet`. The `Main()` method creates an instance of class `Employee`. An attempt is made to retrieve an instance of `IGet` interface checks whether class `Employee` implements methods defined in the interface. Since the `as` operator returns null, the code displays the specified error message.

Figure 6.13 displays the output of the example that uses the `as` operator.



Figure 6.13: Example of as Operator

6.3 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.

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.

6.3.1 Similarities

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.

6.3.2 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 6.1 lists the differences between Abstract Classes and Interfaces.

Abstract Classes	Interfaces
An abstract class can inherit a class and multiple interfaces.	An interface can inherit multiple interfaces, but cannot inherit a class.

Abstract Classes	Interfaces
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 the developer must implement common methods and declare common abstract methods.	An interface is a better option when the developer must declare only abstract methods.
An abstract class can declare constructors and destructors.	An interface cannot declare constructors or destructors.

Table 6.1: Differences Between Abstract Classes and Interfaces

6.3.3 Recommendations for Using Abstract Classes and Interfaces

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 must 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.

6.4 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, the developer must 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.

6.5 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. Peter is trying to inherit the abstract class **Vehicle** and implement its methods in the subclass **Ferrari**. Which of the following codes will help Peter 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"); } 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"); } 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(); } }</pre>
(D)	<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>

(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. Peter is trying to inherit and implement the interface **ICar** and its methods in the subclass **Zen**. Which of the following codes will help Peter to achieve this?

(A)	<pre>interface ICar{ public void Wheels(); public 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(); Wheels(); Speed(); } }</pre>
(B)	<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>

```

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();
}

```

```

interface ICar {
    void Wheels();
    void Speed();
}
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 <code>override</code> keyword
(E)	An interface cannot be instantiated

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

6.5.1 Answers

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

Try It Yourself

1. Design a set of shapes for a graphic design application in C#. You must create a base class called **Shape** that will serve as a blueprint for various shapes. The **Shape** class should be abstract and contain an abstract method **CalculateArea()** to calculate the area of a shape.

Additionally, you must create an interface called **IDrawable** with a method **Draw()** to render a shape.

Create two classes, **Circle** and **Rectangle**, that inherit from **Shape** class and implement **IDrawable** interface. Implement **CalculateArea()** method in both **Circle** and **Rectangle** classes to calculate area specific to each shape. Implement **Draw()** method to display the shape on the screen for both **Circle** and **Rectangle** classes.

Demonstrate the use of these classes and the interface to create objects and calculate areas while rendering shapes on the screen.

Write the C# code to achieve this.

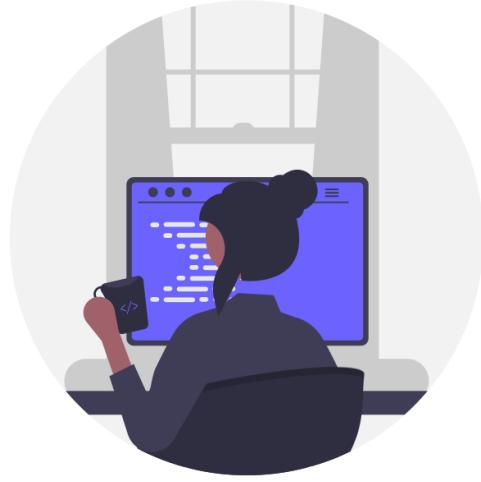
2. Implement a basic system for a library catalog. Create an abstract class **LibraryItem** that represents items that can be found in the library. This abstract class should have properties for **Title** and **DueDate**, and a method **CheckOut** that sets **DueDate** when an item is checked out.

Next, create an interface **IReservable** that includes a method **Reserve()** for reserving library items. Implement this interface in a class **Book** that derives from **LibraryItem** abstract class. The **Book** class should also have a property **Author**.

Finally, implement a class **DVD** that also derives from **LibraryItem** abstract class. It should have a field **Director**.

In your **Main()** method, create instances of the **Book** and **DVD** classes, demonstrate checking out and reserving library items and display the relevant information.

Your solution should showcase the use of abstract classes and interfaces in C# and demonstrate how they can be used in a practical scenario.



Session 7

Properties, Indexers, and Record Types

Welcome to the Session, **Properties, Indexers, and Record Types**.

Properties are data members that allow the developer to protect the private fields of a class. Property accessors known as methods allow the developer to read and assign values to fields. Apart from properties, C# also supports indexers, which allow the developer to access objects such as arrays. This session also introduces init only setters and record types.

In this Session, you will learn to:

- Define properties in C#
- Explain properties, fields, and methods
- Explain indexers
- Describe init only setters and record types

7.1 *Properties in C#*

Access modifiers such as **public**, **private**, **protected**, and **internal** are used to control 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 the developer to set and retrieve values of fields declared with any access modifier in a secured manner. This is because properties allow the developer to validate values before assigning them to fields.

For example, consider fields that store names and IDs of employees. The developer can create properties for these fields to ensure accuracy and validity of values stored in them.

7.1.1 *Applications of Properties*

Properties allow developers to protect a field in the class by reading and writing to the field

through a property declaration. Additionally, properties allow access to private fields, which would otherwise be inaccessible. Properties can validate values before allowing the developer 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.

7.1.2 Declaring Properties

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**.

7.1.3 get and set Accessors

Property accessors allow the developer 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.

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 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.

7.1.4 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 the developer to retrieve the value of a private field. To create a read-only property, the developer should define the `get` accessor.

Following syntax creates a read-only property:

Syntax

```
<access_modifier><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 the developer to change the value of a private field. To create a write-only property, the developer should define the **set** accessor.

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 the developer to set and retrieve the value of a private field. To create a read-write property, the developer should define the **set** and **get** accessors.

Following syntax creates a read-write property:

Syntax

```
<access_modifer> <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 which then 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.

7.1.5 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
        {
            _department = value;
        }
    }
}
public 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

7.1.6 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 `get` and `set` accessors.

The `get` and `set` accessors do 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;
            }
        }
    }
}
```

```

    }
    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

7.1.7 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 must 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.

7.1.8 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\$

7.1.9 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 result 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.

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);
        emp.Designation = "Sales Person";
    }
}
```

Code Snippet 8 declares three auto-implemented properties: `Name`, `Age`, and `Designation`. The `Main()` method first sets values of the properties, and then retrieves values and writes to the console. 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.

7.1.10 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

7.1.11 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

7.2 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.

7.2.1 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 7.1 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 7.1: Differences between Properties and Fields

7.2.2 Properties versus Methods

The implementation of properties covers the implementation of fields and 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 7.2.

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 7.2: Differences between Properties and Methods

7.3 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.

7.3.1 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 the index position of the student ID to locate the student record.

Figure 7.1 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 7.1: Index Position of Indexers

7.3.2 Definition of Indexers

Indexers are data members that allow the developer 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, the developer uses the index position of an object to access its value. Similarly, an indexer allows the developer 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**.

7.3.3 Declaration of Indexers

Indexers allow the developer 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.

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 two times and displays values assigned at each index position using the `get` accessor.

7.3.4 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, the developer must 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.

7.3.5 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

```
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
```

7.3.6 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.

7.3.7 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

```
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 four rows and two 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.

7.3.8 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

```
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");
    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.

7.3.9 Difference between Properties and Indexers

Indexers are syntactically similar to properties. However, there are certain differences between them. Table 7.3 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 this 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 static .	Indexers can never be declared as static .
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 base.Prop , where Prop is the name of the property.	Overridden indexers are accessed using the syntax base[indExp] , where indExp is the list of parameters separated by commas.

Table 7.3: Difference between Properties and Indexers

7.4 Init Setters and Record Types

C# recently introduced many new features such as init only setters, records, top-level statements, and pattern matching enhancements. C# also introduced support for immutability.

Immutability can make the objects thread-safe and can be used for improving the memory management. An immutable object is an object that cannot be altered once it has been created. In C#, init only setters and record types are used to support the immutability.

- **Init only Setters** - It is used to make the individual properties of an object immutable.
- **Record Types** - It is used to make the entire object immutable.

7.4.1 Init Only Setters

These setters were introduced from C# 9.0 onwards. They can be used to create an immutable field without any complexity, while still allowing them to be set in both the constructor and during Nested Object Creation.

Features of `init` only setters are as follows:

- Init accessor is a variant of `set` accessor that can be called during object initialization.
- In C# 9.0 onwards, `init` accessors can be created instead of `set` accessors for properties and indexers.
- Upon using `init` keyword, it restricts a property to only being used by a constructor. In other words, `init` only is used to make an object immutable. Code Snippet 17 demonstrates usage of `init` setters.

Code Snippet 17

```
public class Person {  
    public string? FirstName { get; init; }  
    public string? LastName { get; init; }  
}
```

In Code Snippet 17, `init` keyword is used. Upon using this keyword, one cannot change the `FirstName` and `LastName` values. If these variables are changed, then the code will show an error.

`Init` only setters can be useful to set base class properties from derived classes. They can also be used to set derived properties through helpers in a base class. Positional records declare properties using `init` only setters. These setters are used in `with`-expressions. The `init` only setters can be declared for any class, struct, or record that the developer defines.

7.4.2 Record Types

Microsoft has introduced record type feature from C# 9.0 onwards. Record types provide immutability in C#. Before C# 9.0, C# did not support immutability features. With inclusion of record types and `init` setters, C# 9.0 and higher versions now support immutability that helps to provide security and improve memory management.

A `record` type is a lightweight class or data type that has read-only properties. It cannot be changed after it has been created.

Features of `record` types are as follows:

- `record` keyword is used to define record types as shown in Code Snippet 18.

Code Snippet 18

```
public record Student (string subject, int marks);
```

Record type can also be created using mutable properties as shown in Code Snippet 19.

Code Snippet 19

```
public record Student {  
    public string Subject { get; set; }  
    public int marks {get; set; }  
};
```

- In C#, there are two ways in which space in memory is allocated. It is either based on value type or reference type of data. In value type, the data type holds the value of variable on its own memory. Whereas, in reference type, the variable value is not stored in its own memory rather, it contains a pointer that points to the memory location where the data is stored. Record types offer concise syntax for creating reference types that have immutable properties.
- Immutability in record type is defined as a property where once an object or file is created, it cannot be changed. While the records can be mutable, it can be used to create immutable data models easily. As the record type is immutable, it has read only properties.
- Unlike structure type, record type supports inheritance. A record cannot be inherited from a class and a class cannot be inherited from a record, however, a record can inherit the data from another record. Record type is lightweight as compared to structure type.

Note: A structure type includes collection of different types of data under a single unit.

How can a record type be declared in C# 9.0 and higher versions?

A record can be declared using `record` keyword. This is demonstrated in Code Snippet 20.

Code Snippet 20

```
public record Player {  
    public string Name { get; set; }  
    public string sports { get; set; }  
    public int matchPlayed { get; set; }  
    public int MatchesWon { get; set; }  
    public string Country { get; set; }  
}
```

In Code Snippet 20, the record `Player` is not immutable. To make it immutable, we must use `init` property as shown in Code Snippet 21.

Code Snippet 21

```
public record Player {  
    public string Name { get; init; }  
    public string sports { get; init; }  
    public int matchPlayed { get; init; }  
    public int MatchesWon { get; init; }  
    public string Country { get; init; }  
}
```

Record types also define behavior which is useful for a data-centric reference type that includes following features:

- Value equality
- Concise syntax for nondestructive mutation
- Built-in formatting for display

➤ **Value equality:** When two variables of a record type are equal and all the properties of variables match, it is called Value equality. In C#, the `Object` class has two methods – `Equals()` and `ReferenceEquals()`.

`Equals()` is based on a process that checks the data, whether it is equal or not whereas, `ReferenceEquals()` checks whether the object is the same or not. Reference equality is required for some data models.

For example, in the Entity Framework Core database mechanism, developers must ensure only one instance of the entity should be available throughout. Thus, this is a case where reference equality can play a key role by comparing instances. Another example to illustrate reference equality involves comparing two objects, `obj1` and `obj2`. `ReferenceEquals()` returns true if `obj1` is the same instance as `obj2` or if both are null otherwise, it returns false. The record type overrides these two methods to check the equality of values one by one recursively. This is illustrated in Code Snippet 22.

Code Snippet 22

```
using System;
namespace ConsoleApplication1 {
    public record Cricketer(string FirstName, string LastName,
    string[] HighScore) {
    public static void Main() {
        var HighScore = new string[2];
        Cricketer Cricketer1 = new("Steve", "Smith", HighScore);
        Cricketer Cricketer2 = new("Steve", "Smith", HighScore);
        Console.WriteLine(Cricketer1 == Cricketer2);
        Cricketer1.HighScore[0] = "183";
        Console.WriteLine(Cricketer1 == Cricketer2);
        Console.WriteLine(Cricketer1.Equals(Cricketer2));
        Console.WriteLine(ReferenceEquals(Cricketer1, Cricketer2));
    }
}
```

The output of the code will be as follows:

```
True
True
True
False
```

- **Nondestructive Mutation:** Nondestructive mutation is a concept used to mutate immutable properties. Immutable properties cannot be changed once created. However, nondestructive mutation can make changes in immutable properties. Nondestructive mutations allow us to create a copy of that immutable property and we can add changes in that copy.

In simple terms, nondestructive mutation enables the developer to change the state of an object by creating a copy with the change – rather than making changes to the original object.

To set positional properties or properties created by using standard property syntax, the `with` expression can be used. In the case of non-positional properties, it must have an `init` or `set` accessor that can be changed in a `with` expression.

An example of nondestructive mutation is shown in Code Snippet 23.

Code Snippet 23

```
public class NondestructiveDemo {
    public record Player(string Name, int Avg, int age);
    public static void Main() {
        var p1 = new Player("Jim Smith", 50, 30);
        Console.WriteLine($"{nameof(p1)}: {p1}");
        var p2 = p1 with { Name = "Chris Dane", Avg = 55 };
        Console.WriteLine($"{nameof(p2)}: {p2}");
        var p3 = p1 with { Name = "Andrew Flintoff", age = 38 };
        Console.WriteLine($"{nameof(p3)}: {p3}");
        Console.WriteLine($"{nameof(p1)}: {p1}");
    }
}
```

In Code Snippet 23, the `with` keyword is used to create a copy of the instance immutable properties. This way now the developer can make changes in the second and third `Player` instances, `p2`, and `p3` respectively that the developer has created using `with` keyword and the original item will remain immutable.

For reference properties, only the reference to an instance is copied and thus, the outcome of a `with` expression is a shallow copy. This means that both the original record and the copy have a reference to the same instance.

The compiler achieves this by simulating a `clone` method and a `copy constructor`. The virtual `clone` method returns a new record initialized by the `copy constructor`. When a `with` expression is used, the compiler creates code that is used to call the `clone` method. After calling the `clone` method, it then sets the properties that are specified in the `with` expression.

Output:

```
p1: Player { Name = Jim Smith, Avg = 50, age = 30 }
p2: Player { Name = Chris Dane, Avg = 55, age = 30 }
p3: Player { Name = Andrew Flintoff, Avg = 50, age = 38 }
p1: Player { Name = Jim Smith, Avg = 50, age = 30 }
```

- **Built-In Formatting for Display:** The record type has built-in compiler-generated `ToString` method which is used to display names and values of public properties.

The `ToString()` method returns a string having following syntax:

Syntax

```
<record type name> {<property name> = <value>, <property name> = <value>,
...
}
```

In the case of reference types, type name of the object referred to by the property is printed instead of property value.

Following statement shows an example where the array is a reference type, so `System.String[]` is displayed instead of actual array element values.

```
Avengers { FirstName = Tony, LastName = Stark, SuperHeroNames =
System.String[] }
```

To render the formatting, the compiler makes use of a virtual `PrintMembers` method and a `ToString()` override. The `ToString()` override helps to create a `StringBuilder` object with the type name which is followed by an opening bracket. Consider Code Snippet 24.

Code Snippet 24

```
class TestRecord2{
    public record Employee(string FirstName, string LastName);
    public static void Main(){
        Employee employee1 = new("Hank", "Williams");
        Console.WriteLine(employee1);
    }
}
```

Output will be:

```
Employee { FirstName = Hank, LastName = Williams }
```

In the code, we created an instance of an `Employee` record and provided it to the `Console.WriteLine` method, which calls its `ToString()` implementation. The output displays the name of the record type and its properties, including their values. The output format looks similar to JavaScript Object Notation (JSON), but starts with the name of the record type.

Internally, the built-in `ToString()` override looks like this:

```
public override string ToString() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Employee"); // type name
    //stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder)) {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("} ");
    return stringBuilder.ToString();
}
```

Records also support inheritance. Inheritance rules for records are as follows:

- A record can inherit only from another record
- A class cannot inherit from a record

Uses of Record Types

Records can be useful to log data, copy data structures, compare different objects' properties in business applications, and so on.

Record Types Can Seal `ToString()`

In C# 10.0, a new feature is introduced - record types can seal the `ToString()` method. This means that the developer can prevent derived classes from overriding the `ToString()` method when a record type is being declared. Sealing `ToString()` can be useful in scenarios where the developer wants to ensure that the string representation of an object remains consistent and predictable across all instances of the record type.

Code Snippet 25 depicts an example.

Code Snippet 25

```
public record Person(string FirstName, string LastName)
{
    // Sealing the ToString() method to prevent overrides
    public sealed override string ToString()
    {
        return $"{FirstName} {LastName}";
    }
}

public record Employee(string FirstName, string LastName) :
Person(FirstName, LastName) {
    public override string ToString() // Error CS0239
        //'Employee.ToString()': cannot override inherited member
        //'Person.ToString()' because it is sealed
}
```

```
        return $"My name is {FirstName} {LastName}";
    }
}
class Program{
    static void Main()
    {
        var person = new Person("John", "Doe");
        Console.WriteLine(person); // Program does not compile
                                //therefore, no output is displayed
    }
}
```

In Code Snippet 25, a record type **Person** is defined with two properties: **FirstName** and **LastName**. The **ToString()** method is overridden in the **Person** record and sealed using the **sealed** keyword. Sealing the method prevents any derived class from further overriding it.

The **ToString()** method implementation returns a string that represents the person's full name by combining the **FirstName** and **LastName** properties.

In the **Main** method, the code creates an instance of the **Person** record and prints its data to the console using **Console.WriteLine()**. Since the **ToString()** method is sealed, it will always return the formatted full name, and no derived class can change this behavior.

Sealing the **ToString()** method can be a helpful practice when the developer wants to ensure that the string representation of objects remains consistent and cannot be altered by subclasses. However, the developer should use this feature judiciously, considering the specific requirements of the application.

7.5 Summary

- Properties protect the fields of the class while accessing them.
- Property accessors enable the developer to read and assign values to fields.
- A field is a data member that stores some information.
- Properties enable the developer 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 ([]).
- Init only setters are used to make the individual properties of an object immutable whereas, record types are used to make entire object immutable.
- Init accessor is a variant of the set accessor that can be called during object initialization.
- A record type is a lightweight class or data type that has read-only properties. It cannot be changed after it has been created.
- Nondestructive mutation is a concept used to mutate immutable properties.

7.6 Check Your Progress

1. Peter is trying to display the output of the balance amount as "1005.50" using properties.
Which of the following codes will help him to achieve this?

(A)	<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 { return _balanceAmount; } set { balanceAmount = value; } } 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) { Balance objBal = new Balance(); objBal.BalanceAmount = 1005.50; Console.WriteLine(objBal.BalanceAmount); } }</pre>

<pre>(D) 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 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. Sarah is 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 her 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);
    }
}
```

(C)	<pre>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); } }</pre>
-----	--

(D)	<pre>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); } }</pre>
-----	---

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

5. Which of the following codes will help to display the output of different products as 'Mouse', 'Keyboard', and 'Speakers' using indexers?

(A)	<pre> 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]); } } } </pre>
(B)	<pre> class Products { private string[] productName = new string[3]; public string Products(int index) { get { } set { } } 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]); } } } </pre>

(C)

```

class Products {
    private string[] _productName = new string[3];
    public string this[int index] {
        get { return value[index]; } set { }
    }
    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]);
        }
    }
}

```

(D)

```

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

7.6.1 *Answers*

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

Try It Yourself

1. Create a simple inventory management system using C# for a bookstore by performing these tasks:

Create a `Book` record with following properties:

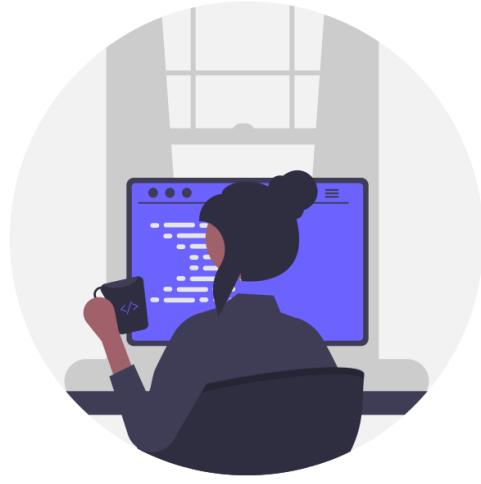
- `Title` (string): The title of the book.
- `Author` (string): The author of the book.
- `ISBNCode` (string): The International Standard Book Number (ISBN) of the book.
- `Price` (double): The price of the book.

Create an `Inventory` class that will store a collection of `Book` records. It should have following features:

- An indexer that allows users to access a book by its ISBN.
- A method `AddBook()` to add a new book to the inventory.
- A property `TotalValue()` that calculates and returns the total value of all the books in the inventory.

In your `Main()` method, demonstrate following operations:

- Create an instance of the `Inventory` class.
- Add several books to the inventory.
- Use the indexer to retrieve a book by its ISBN and display its details.
- Calculate and display the total value of the inventory.



Session 8

Namespaces and Exception Handling

Welcome to the Session, **Namespaces and Exception Handling**.

This session provides insights into C#'s built-in and custom namespaces along with their features and benefits. It also introduces different ways of handling exceptions in C# applications.

In this Session, you will learn to:

- Define and describe namespaces
- Define and describe exceptions
- Explain the process of throwing and catching exceptions
- Explain nested try and multiple catch blocks

8.1 Introduction to Namespaces in C#

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. It is possible to create multiple namespaces within a namespace.

Consider Venice, which is a city in the U.S. as well as in Italy. One 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 that 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.

8.1.1 *Contents of Namespaces*

C# allows specifying a unique identifier for each namespace. This identifier helps accessing the classes within the namespace. Apart from classes, following data structures can be declared in a namespace:

- **Interface:** A reference type that contains declarations of the events, indexers, methods, and properties.

- **Structure:** A structure is a value type that can hold values of different data types.
- **Enumeration:** An enumeration is a value type that consists of a list of named constants.
- **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.

8.2 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 interacting 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 building components that can be used to insert, modify, and delete data from multiple data sources.

`System.IO`

The `System.IO` namespace contains classes that enable developers to read from and write to data streams and files.

`System.Net`

The `System.Net` namespace contains classes that allow creating Web-based applications.

8.2.1 Using 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, developers must 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, developers can refer to classes within the namespaces in the same manner without `using` keywords. However, this results in redundancy because developers must mention the whole declaration every time while referring to the class in the code.

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 the developer wants to access.

MethodName: Is the name of the method within the class that is to be invoked.

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 the developer wants to access.

Code Snippet 3 demonstrates the use of `using` keywords while using namespaces.

Code Snippet 3
<pre>using System; class World { static void Main(string[] args) { Console.WriteLine("Hello World"); } }</pre>

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 `WriteLine` statements.

Code Snippet 4
<pre>class World { static void Main(string[] args) { System.Console.WriteLine("Hello World"); } }</pre>

```
        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#
```

Upon seeing this code, it is clear that the approach shown in Code Snippet 3 is far better than the one shown in Code Snippet 4.

8.3 Custom Namespaces

C# allows creating 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, developers do not have to worry about name clashes with classes, interfaces, and so on in other namespaces.

Custom namespaces enable developers 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.

Code Snippet 5 creates a custom namespace named `Department`.

Code Snippet 5

```
namespace Department {
    class Sales {
        . . .
    }
}
```

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 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 8.1 shows the output of the application containing the three files.



Figure 8.1: Output of the Application Demonstrating Namespaces

8.4 Access Modifiers for Namespaces

Namespaces are always public. Developers cannot apply access modifiers such as public, protected, private, or internal to namespaces. If any of the access modifiers is specified in the namespace declaration, the compiler generates an error.

8.5 Qualified Naming

C# allows using a class outside its namespace. A class outside its namespace can be accessed by specifying its namespace followed by the dot 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, a developer can access classes outside their namespaces with the using keyword. This makes the names short and meaningful. Code Snippet 9 displays the student's name, ID, subject, and marks scored using a fully qualified name.

Code Snippet 9

```
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 9, the class `ScoreReport` uses the class `StudentDetails` defined in the namespace `Examination`. The class is accessed by its fully qualified name.

Output:

Student Name: Alexander
Student ID: 30
Subject: Science
Marks: 60

8.6 Naming Conventions for Namespaces

There are certain naming conventions which must be followed for creating namespaces:

- 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 the same names.
 - Ensure that the name of a namespace is not identical to the name of the assembly.

8.7 Namespaces 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 a developer might want to access only one class from a particular namespace. The developer 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 the developer 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 8.2 displays an example of using namespace aliases.

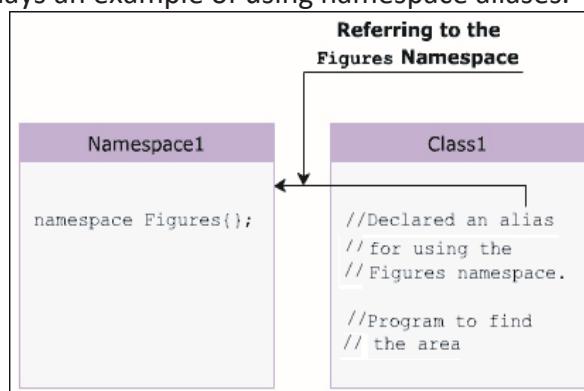


Figure 8.2: Namespace Aliases

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 10 creates a custom namespace called `Bank.Accounts.EmployeeDetails`.

Code Snippet 10

```
namespace Bank.Accounts.EmployeeDetails {
    public class Employees {
        public string EmpName;
    }
}
```

Code Snippet 11 displays the name of an employee using the aliases of the `System.Console` and `Bank.Accounts.EmployeeDetails` namespaces.

Code Snippet 11

```
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);
    }
}
```

Output:

Employee Name: Peter

In Code Snippet 11, 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.

8.7.1 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 that namespace. This is illustrated in Code Snippet 12.

Code Snippet 12

```
//Following program is saved in the Automobile.cs file under the
// Automotive project.
using System.Collections.Generic;
```

```

using System.Text;
using Utility_Vehicle.Car;
using Utility_Vehicle = Automotive.Vehicle.Jeep;
using 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 13.

Code Snippet 13

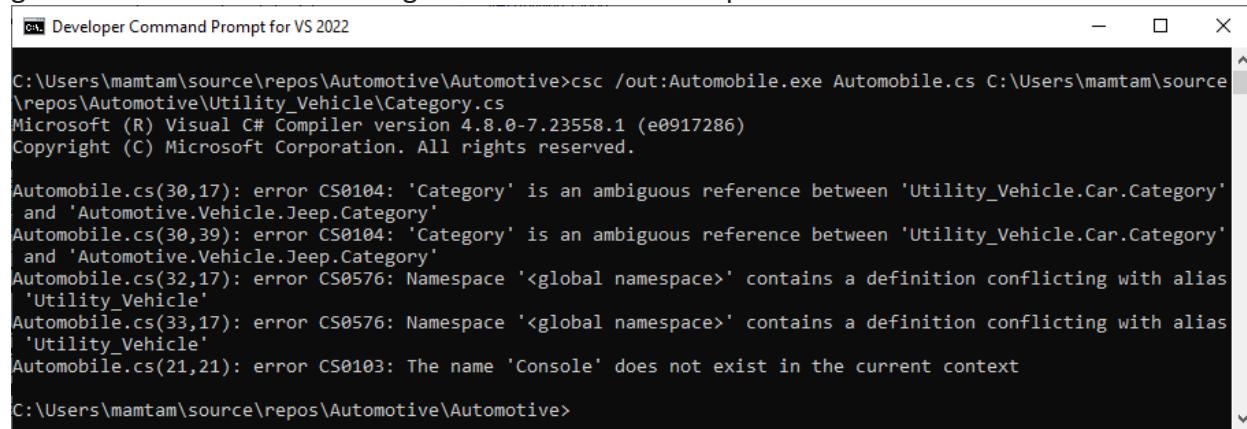
```

//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);
            }
        }
    }
}

```

```
}
```

In Code Snippets 12 and 13, 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 to by the file. Due to this name conflict, the compiler generates an error. Refer to Figure 8.3 to view the compilation error.



The screenshot shows a terminal window titled "Developer Command Prompt for VS 2022". The command entered is `csc /out:Automobile.exe Automobile.cs C:\Users\mamtam\source\repos\Automotive\Utility_Vehicle\Category.cs`. The output shows the following errors:

```
C:\Users\mamtam\source\repos\Automotive\Automotive>csc /out:Automobile.exe Automobile.cs C:\Users\mamtam\source\repos\Automotive\Utility_Vehicle\Category.cs
Microsoft (R) Visual C# Compiler version 4.8.0-7.23558.1 (e0917286)
Copyright (C) Microsoft Corporation. All rights reserved.

Automobile.cs(30,17): error CS0104: 'Category' is an ambiguous reference between 'Utility_Vehicle.Car.Category'
and 'Automotive.Vehicle.Jeep.Category'
Automobile.cs(30,39): error CS0104: 'Category' is an ambiguous reference between 'Utility_Vehicle.Car.Category'
and 'Automotive.Vehicle.Jeep.Category'
Automobile.cs(32,17): error CS0576: Namespace '<global namespace>' contains a definition conflicting with alias
'Utility_Vehicle'
Automobile.cs(33,17): error CS0576: Namespace '<global namespace>' contains a definition conflicting with alias
'Utility_Vehicle'
Automobile.cs(21,21): error CS0103: The name 'Console' does not exist in the current context

C:\Users\mamtam\source\repos\Automotive\Automotive>
```

Figure 8.3: Namespace Conflict 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 14.

Code Snippet 14

```
using System;
using System.Collections.Generic;
using System.Text;
using Automotive.Vehicle.Jeep;
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 14 is shown in Figure 8.4.

The screenshot shows the Microsoft Visual Studio Debug Console window. The output text is:

```

Microsoft Visual Studio Debug Console
Jeep Category: Multi Utility Vehicle
Jeep Category: Multi Utility Vehicle

C:\Users\...\source/repos\Automotive\bin\Debug\net6.0\Automotive.exe (process 17472) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Figure 8.4: Output of Corrected Code

In Code Snippet 14, 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.

8.8 Exceptions in C#

Exceptions in programming 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. Exception handling is a process of handling runtime errors. In C# developers can handle exceptions by using `try-catch` or `try-catch-finally` constructs. In addition, C# supports custom exceptions that allow customizing the error-handling process.

Figure 8.5 displays the types of exceptions in C#.

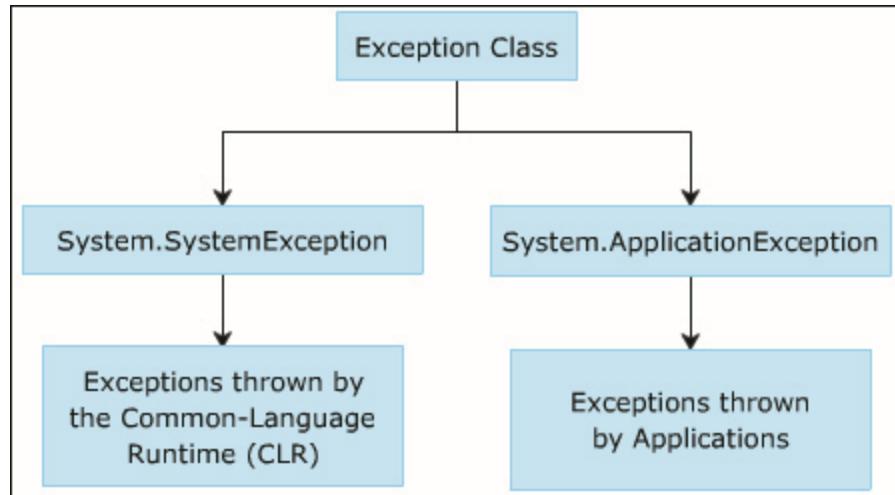


Figure 8.5: Exceptions in C#

8.9 Throwing and Catching Exceptions

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 must be caught by exception handlers. If the program does not contain the appropriate exception handler, then the program might be terminated.

8.9.1 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.

Following is the syntax 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 the exception class. It is optional.
objException: Is an instance of the particular exception class. It can be omitted if the ExceptionClass is omitted.

Code Snippet 15 demonstrates the use of try-catch blocks.

Code Snippet 15

```
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 15, the `Main()` method of the `DivisionError` class 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.

8.9.2 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, a developer might not know the type of exception the program might throw. In such a case, the developer 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, developers cannot know what appropriate action must be performed for handling the exception.

Code Snippet 16 demonstrates the way in which a general `catch` block is declared.

Code Snippet 16

```
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 16, 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 roject_New.Exception_Handling.Students.Main(String[] args) in
D:\Exception Handling\Students.cs:line 17.
```

8.10 Properties and Methods of Exception Class

The `System.Exception` class is the base class that allows handling 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 8.1 describes some of the properties of `System.Exception` class.

Property	Description
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 8.1: Properties of Exception Class

Code Snippet 17 demonstrates the use of some public properties of the System.Exception class.

Code Snippet 17

```
using System;
class ExceptionProperties {
    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 (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 17, 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 8.6 displays use of some of the public properties of the System.Exception class.

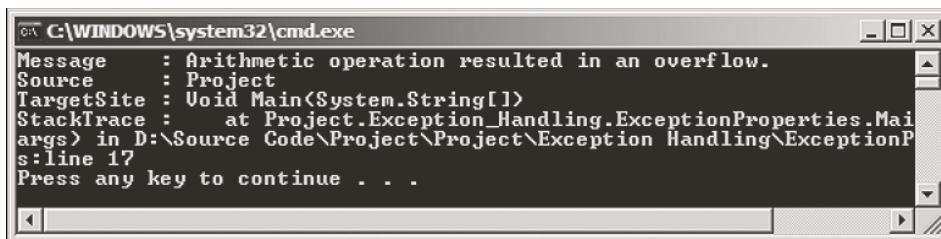


Figure 8.6: Use of Public Properties of System.Exception Class

Table 8.2 lists the public methods of the System.Exception class and their corresponding description.

Method	Description
Equals	Determines whether objects are equal

Method	Description
GetBaseException	Returns a type of Exception class when overridden in a derived class
GetHashCode	Returns a hash function for a particular type
GetObjectData	Stores information about serializing or deserializing a particular object with information about the exception when overridden in the inheriting class
GetType	Retrieves the type of the current instance
ToString	Returns a string representation of the thrown exception

Table 8.2: Public Methods of System.Exception Class

Code Snippet 18 demonstrates use of some public methods of the `System.Exception` class.

Code Snippet 18

```
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 18, 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 8.7 displays some public methods of the `System.Exception` class.

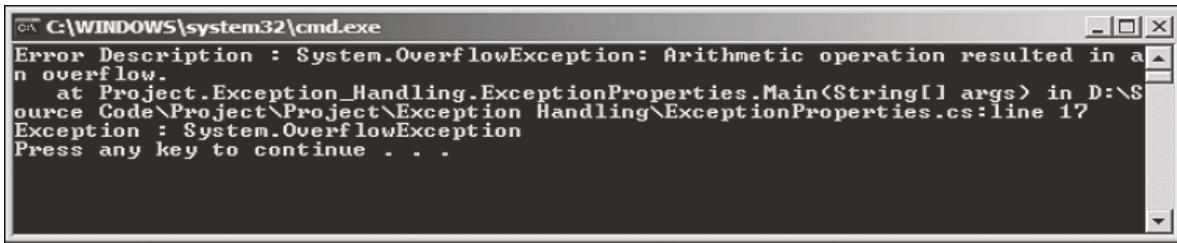


Figure 8.7: Methods of System.Exception Class

8.11 Commonly Used Exception Classes

The `System.Exception` class has a number of derived exception classes to handle different types of exceptions.

Table 8.3 lists some of the commonly used exception classes.

Method	Description
<code>System.ArithmaticException</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.DivideByZeroException</code>	This exception is thrown when an attempt is made to divide a numeric value by zero.
<code>System.IndexOutOfRangeException</code>	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.

Table 8.3: Commonly Used Exception Classes

8.11.1 `InvalidCastException` Class

The `InvalidOperationException` exception is thrown when an explicit conversion from a base type to another type fails. Code Snippet 19 demonstrates the `InvalidOperationException` exception.

Code Snippet 19

```
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 19, 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 8.8 displays the exception that is generated when the program is executed.

The screenshot shows the Microsoft Visual Studio Debug Console window. The output text is as follows:

```

Microsoft Visual Studio Debug Console
Message : Unable to cast object of type 'System.Single' to type 'System.Int32'.
Error : System.InvalidCastException: Unable to cast object of type 'System.Single' to type 'System.Int32'.
      at InvalidCastError.Main(String[] args) in C:\Users\mamtam\source\repos\ConsoleApp2\ConsoleApp2\Test.cs:line 10
C:\Users\mamtam\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\net6.0\ConsoleApp2.exe (process 30796) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Figure 8.8: InvalidCastException Generated at Runtime

8.11.2 NullReferenceException Class

The `NullReferenceException` exception is thrown when an attempt is made to operate on a null reference. Code Snippet 20 demonstrates the `NullReferenceException` exception.

Code Snippet 20

```

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 20, `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.

8.12 throw Statement

The `throw` statement in C# allows a developer 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. While throwing an exception using the `throw` keyword, the exception is handled by the `catch` block.

Code Snippet 21 demonstrates the use of the `throw` statement.

Code Snippet 21

```
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 21, the class Employee declares a static method named 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 `try` block, value of `name` is `null` and control of the program goes back to method `ThrowException`, where the exception is thrown for referencing a `null` value. The `catch` block consists of 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
```

8.13 throw Expressions

C# 7.0 onwards has a new enhancement with respect to exceptions, namely `throw` expressions. In earlier versions, `throw` was a statement and hence, code constructs such as conditional expressions, null coalescing expressions, and certain lambda expressions were not allowed to throw exceptions. However, this restriction is not applicable from C# 7.0. Developers can now throw exceptions in all such code constructs.

For example, Code Snippet 22 throws an exception inside a null coalescing expression.

Code Snippet 22

```
class Program {
    static void Evaluate(string arg)
    {
        var val = arg ?? throw new ArgumentException("Invalid argument");
        Console.WriteLine("Reached this point");
    }
    static void Main()
    {
        Evaluate("numbers");
        Evaluate(null);
    }
}
```

When the code is executed, an exception will be thrown:

```
System.ArgumentException: 'Invalid argument'
```

8.14 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.

Code Snippet 23 demonstrates the use of the `try-catch-finally` construct.

Code Snippet 23

```
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 23, 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
```

8.15 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.

8.15.1 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.

Code Snippet 24 demonstrates the use of nested `try` blocks.

Code Snippet 24

```
...
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 24, the array variable `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

8.15.2 *Multiple catch Blocks*

A `try` block can throw multiple types of exceptions, which must be handled by the `catch` block.

C# allows defining multiple `catch` blocks to handle 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. Code Snippet 25 demonstrates the use of multiple `catch` blocks.

Code Snippet 25

```
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 25, 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.

8.15.3 Exception Assistant

Exception Assistant is a recently added feature for debugging C# applications. This assistant appears whenever a runtime 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.

8.15.4 Best Practices

Best practices for exception handling are important in any programming language, including C#, because they help ensure that the code is robust, maintainable, and reliable. Here are some reasons why best practices for exception handling are required in C# 10.0:

Use Exceptions for Exceptional Situations: Exceptions should be used for handling truly exceptional situations, such as divide by zero, file not found, or network errors. They should not be used for regular control flow or as a means of handling expected conditions.

Choose the Most Specific Exception: Use the most specific exception type that accurately represents the error. Avoid catching general exceptions such as `System.Exception` unless one has a good reason to do so. Catching more specific exceptions allows to handle errors more precisely. Code Snippet 26 depicts an example of this.

Code Snippet 26

```
try
{
    // Code that may throw a specific exception
}
catch (FileNotFoundException ex)
{
    // Handle file not found error
}
catch (DivideByZeroException ex)
{
    // Handle divide by zero error
}
```

Always Include Error Messages: When throwing exceptions, provide a clear and informative error message. This helps other developers understand the cause of the exception and makes debugging easier.

```
throw new InvalidOperationException("Cannot perform operation because of X.");
```

Use Custom Exceptions Sparingly: While creating custom exception classes can be useful in some cases, do not overuse them. Only create custom exceptions when one has a specific requirement for additional information or when specific logic for handling certain types of errors has to be encapsulated.

Handle Exceptions at the Right Level: Handle exceptions at a level of an application where one can take meaningful action or provide a user-friendly error message. One should not catch exceptions at a level where nothing useful can be done with them.

Use Finally Blocks for Cleanup: If one performs cleanup operations (for example closing files or releasing resources), use a `finally` block to ensure these operations are always executed, even if an exception is thrown. Code Snippet 27 shows how a `finally` block can be used to perform cleanup.

Code Snippet 27

```
FileStream file = null;
try
{
    file = new FileStream("file.txt", FileMode.Open);
    // Perform file operations
}
catch (IOException ex)
{
    // Handle IO exception
}
finally
{
    file?.Close();
    // Ensure the file is closed, even if an exception occurs.
}
```

Avoid Swallowing Exceptions: Be cautious about catching exceptions without proper handling or rethrowing them without additional information. Swallowing exceptions can make it difficult to diagnose and fix issues.

Logging Exceptions: Consider logging exceptions to help with debugging and monitoring. Log the exception details, including the type, message, stack trace, and any relevant context.

Structured Exception Handling: Consider using structured exception handling constructs such as

`try-catch-finally` or the `using` statement when working with resources that must be cleaned up, such as files or database connections.

Fail Fast: When one encounters an unrecoverable error, it is often better to fail fast and let the application crash gracefully with an informative error message rather than trying to recover from a state that might be inconsistent.

Unit Testing for Exception Handling: Write unit tests to ensure that your exception handling code is working as expected. Test both the code paths that should throw exceptions and the code paths that handle them.

Use Patterns such as TryParse: For scenarios where one expects that a certain operation might fail, consider using patterns such as `int.TryParse` or `DateTime.TryParse` instead of relying on exceptions for control flow.

8.16 Summary

- A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names.
- System namespace is imported by default in the .NET Framework.
- Custom namespaces enable the developer to control the scope of a class by deciding the appropriate namespace for the class.
- Access modifiers such as public, protected, private, or internal are inapplicable to namespaces.
- A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- Exception-handling allows the developer 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 must be executed irrespective of whether or not an exception is thrown by the try block.
- Nested try blocks allow the developer 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.

8.17 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 built-in namespace System
(E)	Custom namespaces control scope of a class

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

2. Which of these statements about exceptions and exception classes are true?

(A)	Exceptions are compile-time errors that disrupt the program flow
(B)	InnerException is a property of the Exception class which returns an Exception object that caused the exception
(C)	Message is a property of the Exception class that displays reason for the exception
(D)	Source is a property of the Exception class that provides name of the application that caused the exception

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

3. Match the commonly used exception classes against their corresponding descriptions.

Description		Exception Class	
(A)	Raised when the developer tries to create a new object and there is not enough memory	(1)	ArrayTypeMismatchException
(B)	Raised when the developer tries 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 the developer tries 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-2, D-4, E-3

4. 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)	Is part of exception-handling statements	(3)	finally
(D)	Encloses statements that might raise exceptions	(4)	throw
(E)	Executes the code within the block irrespective of whether an exception is raised or not		

(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

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-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, and D	(D)	A and D

6. Match 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.IO
(C)	Contains classes that make up the ADO.NET architecture	(3)	System.Net
(D)	Contains classes that read from and write to data streams and files	(4)	System.Web

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

8.17.1 Answers

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

Try It Yourself

1. Build a C# program to manage a directory structure using custom namespaces. You are given the task of writing a function to calculate the total size of a directory and its subdirectories. You must also handle exceptions that may occur during the process.

Create a custom namespace named `DirectoryManager` to encapsulate your functionality. Within this namespace, define a class called `DirectorySizeCalculator`.

In the `DirectorySizeCalculator` class, create a method named `CalculateTotalSize()` with following signature:

```
public long CalculateTotalSize(string directoryPath)
```

The method should accept a `directoryPath` as input and return the total size of the directory and its subdirectories in bytes. It should use recursion to traverse the directory structure.

Implement error handling within the `CalculateTotalSize()` method to catch and handle exceptions that may occur during the directory traversal. Specifically, handle exceptions related to directory access, such as unauthorized access, and provide appropriate error messages.

In your `Main()` method, demonstrate the use of the `DirectorySizeCalculator` class by calculating the total size of a directory (for example, a folder on your system) and displaying the result. Ensure that you handle exceptions gracefully and display relevant error messages if any issues occur during the directory traversal.

Your solution should showcase the use of custom namespaces, recursion, and effective exception handling to calculate the total size of a directory and its subdirectories while providing informative error messages in case of exceptions.

2. Develop a C# program to perform a recursive directory search operation and you should handle potential exceptions that may arise during this process.

Create a C# console application that defines a custom namespace called `DirectorySearch`.

Implement a class `FileSearch` within the `DirectorySearch` namespace. This class should have a method `SearchFiles(string directoryPath, string targetFile)`.

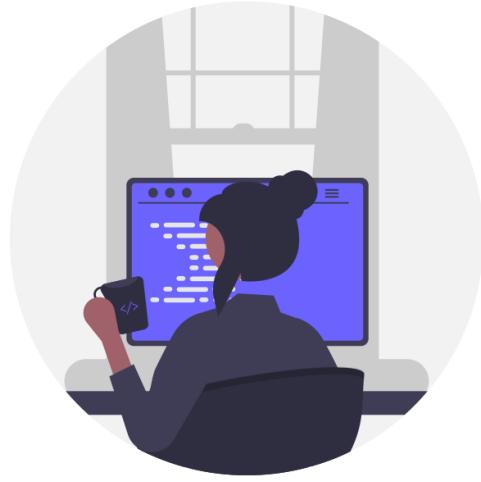
The `SearchFiles()` method should use recursion to search for a specific target file in the given directory and its subdirectories. When the target file is found, print its full path.

Implement proper exception handling for any potential issues, such as permission errors or

invalid directory paths, during the directory search operation.

In the `Main()` method of your program, demonstrate the use of `FileSearch` class by searching for a specific target file within a directory, providing appropriate exception handling for any errors that may occur during the search.

Your solution should showcase the use of namespaces, recursion, and exception handling in C# while performing a real-world task of searching for files in directories.



Session 9

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 the developer to control and manipulate a group of objects dynamically at runtime. The `System.Collections` namespace consists of collections of arrays, lists, hash tables, 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

9.1 *Delegates*

In the .NET Framework, a delegate points to one or more methods. Once the developer instantiates the delegate, the corresponding methods are invoked. Delegates are objects that contain references to methods that must be invoked instead of containing the actual method names. Using delegates, the developer can call any method which is identified only at runtime.

A delegate has a general method name that points to various methods at different times and invokes the required method at runtime. In C#, invoking a delegate will execute the referenced method at runtime.

To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.

9.1.1 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.

9.1.2 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 9.1 displays an example of declaring delegates.

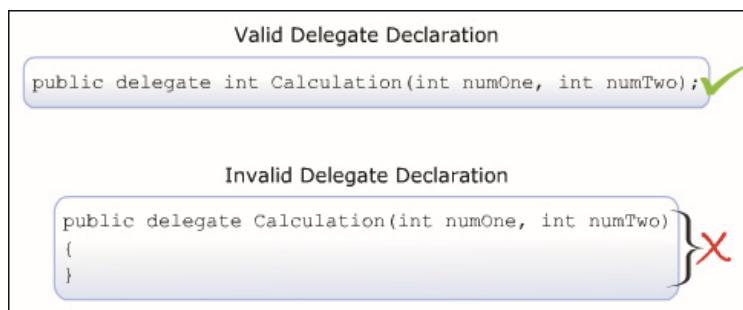


Figure 9.1: Declaring Delegates

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

9.1.3 Instantiating Delegates

The next step after declaring the delegate is to instantiate the delegate and associate it with the required method. Here, the developer 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 runtime. Figure 9.2 displays an example of instantiating delegates.

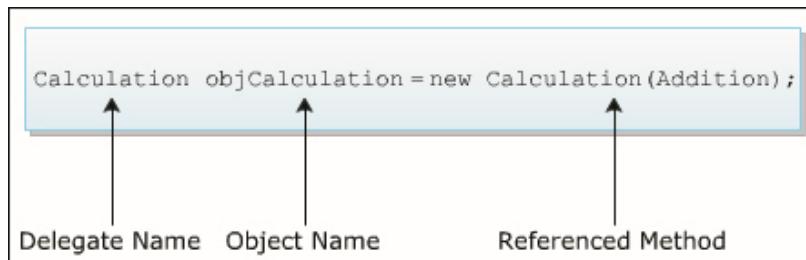


Figure 9.2: Instantiating Delegates

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.

9.1.4 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 steps is demonstrated with an example in Figure 9.3.

```

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 9.3: Using Delegates

An anonymous method is an inline block of code that can be passed as a delegate parameter. Using anonymous methods, the developer can avoid creating named methods. Figure 9.4 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();
}

```

} Anonymous Method

Figure 9.4: Anonymous Methods

9.1.5 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 must be invoked when events occur. These methods are referred to as the event handlers.

9.1.6 *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 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 9.5 shows the use of multiple delegates.

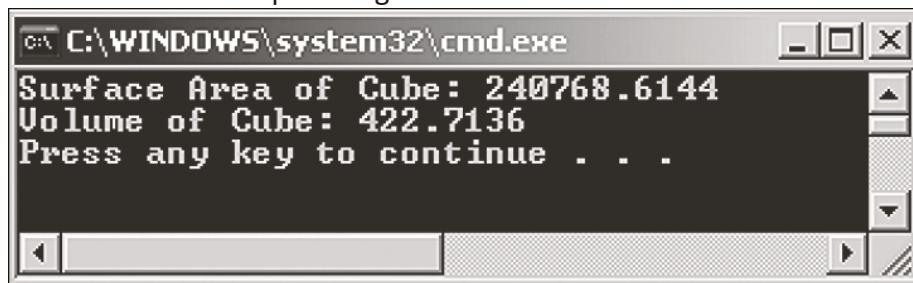


Figure 9.5: Use of Multiple Delegates

9.1.7 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 runtime 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 9.6 shows the creation of a multicast delegate.

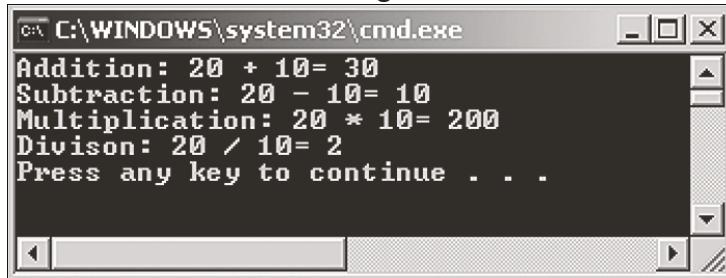


Figure 9.6: Creation of Multicast Delegate

9.1.8 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 9.1 lists the constructors defined in the `Delegate` class.

Constructor	Description
<code>Delegate(object, string)</code>	Calls a method referenced by object of the class given as parameter
<code>Delegate(type, string)</code>	Calls a static method of the class given as parameter

Table 9.1: Constructors of the Delegate Class

Table 9.2 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 delegate invokes the referenced method

Table 9.2: Properties of the Delegate Class

Table 9.3 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
<code>CreateDelegate</code>	Declares and initializes a delegate
<code>DynamicInvoke</code>	Calls the referenced method at runtime
<code>GetInvocationList</code>	Retrieves the invocation list of the current delegate

Table 9.3: Methods of the Delegate Class

Code Snippet 5 demonstrates 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);
    }
}

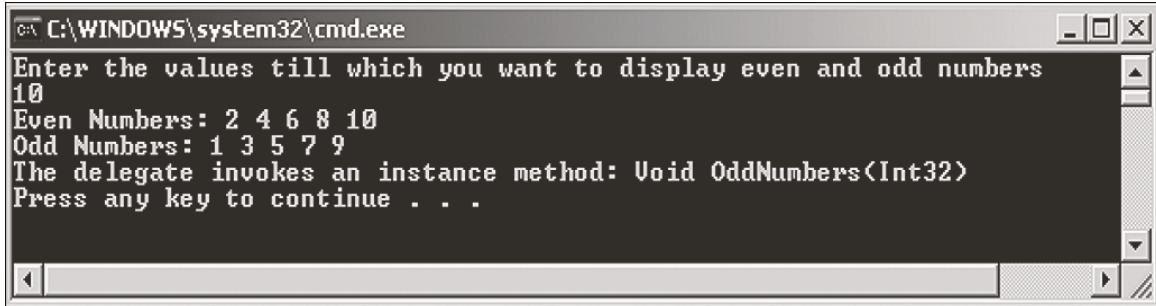
```

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 9.7 shows the use of some of the properties and methods of the Delegate class.



```
C:\WINDOWS\system32\cmd.exe
Enter the values till which you want to display even and odd numbers
10
Even Numbers: 2 4 6 8 10
Odd Numbers: 1 3 5 7 9
The delegate invokes an instance method: Void OddNumbers(Int32)
Press any key to continue . . .
```

Figure 9.7: Properties and Methods of Delegate Class

9.2 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 9.8 depicts the concept of events.

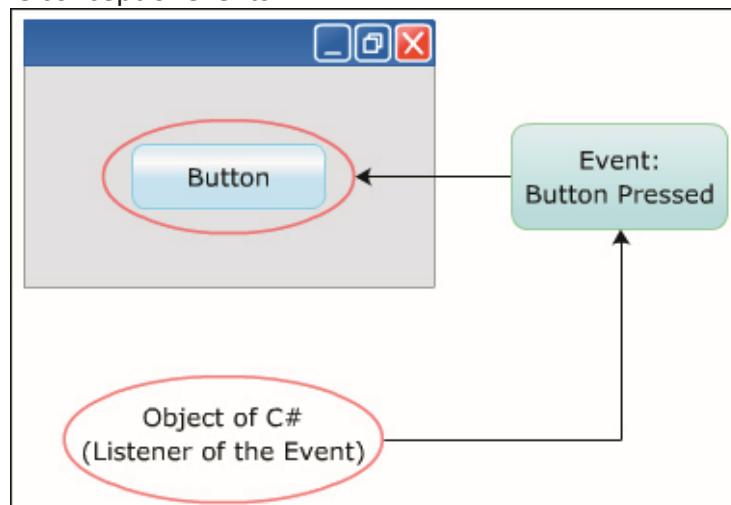


Figure 9.8: Events

9.2.1 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 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.

9.2.2 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.

9.2.3 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 9.9 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 9.9: 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.

Following syntax is used to create a method in the receiver class:

Syntax

```
<access_modifier> <return_type> <MethodName> (parameters);
```

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 with 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**.

9.2.4 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 the developer to create handlers and thus, make sure that the event is associated with the appropriate event handlers. If the event is not associated with any event handler, the declared event is considered to be **null**. Figure 9.10 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 9.10: 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.

9.2.5 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 9.11 shows the outcome of invoking the event.

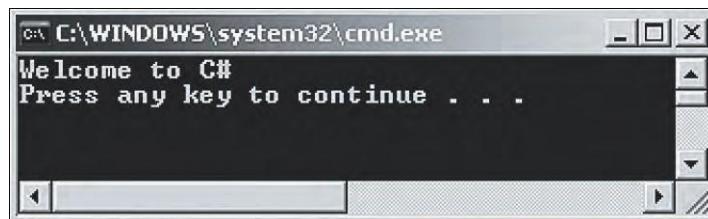


Figure 9.11: Outcome of Invoking the Event

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.

9.3 Handling Multiple Events Using Event Properties

Event properties allow the developer to encapsulate the subscription and unsubscription of event handlers within a class, providing a clean and organized way to manage multiple events. Here is a brief overview of how to use event properties:

- **Declare an Event Property:** To create an event property, the developer will first require to declare an event using the `event` keyword within class. Refer to Code Snippet 9.

Code Snippet 9

```
public class EventManager
{
    public event EventHandler<MyEventArgs> MyEvent;
}
```

- **Provide Accessors:** Event properties typically have add and remove accessors that allow the developer to subscribe and unsubscribe event handlers. Refer to Code Snippet 10.

Code Snippet 10

```
public class EventManager
{
    private event EventHandler<MyEventArgs> myEvent;
    public event EventHandler<MyEventArgs> MyEvent {
        add { myEvent += value; }
        remove { myEvent -= value; }
    }
}
```

- **Raise the Event:** Within the class, the developer can raise the event when some specific condition is met. the developer does this by invoking the event such as a method. Refer to Code Snippet 11.

Code Snippet 11

```
protected virtual void OnMyEvent (MyEventArgs e)
{
    myEvent?.Invoke(this, e);
}
```

- **Subscribe to the Event:** Outside of the class, the developer can subscribe to the event using

the `+=` operator and specify the event handler method:

```
EventManager manager = new EventManager();  
manager.MyEvent += MyEventHandlerMethod;
```

- **Unsubscribe from the Event:** To remove an event handler, the developer can use the `-=` operator:
- **Handle Multiple Events:** The developer can repeat these steps for multiple events within the class, providing a structured way to manage and handle different events.

Using event properties allows the developer to encapsulate the event management logic within the class, making the code more maintainable and reducing the chances of memory leaks. It also follows the principles of encapsulation and information hiding in object-oriented programming.

9.4 *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 runtime. 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 9.4 lists the differences between arrays and collections.

Arrays	Collections
Cannot be resized at runtime.	Can be resized at runtime.
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 9.4: Differences Between the Arrays and Collections

9.4.1 *System.Collections Namespace*

The `System.Collections` namespace in C# allows the developer 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 different collections.

Table 9.5 lists the commonly used classes and interfaces in the `System.Collections` namespace.

Class/Interface	Description
ArrayList 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
Stack 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
Hashtable Class	Provides a collection of key and value pairs that are arranged, based on the hash code of the key
SortedList Class	Provides a collection of key and value pairs where the items are sorted, based on the keys
IDictionary Interface	Represents a collection consisting of key/value pairs
IDictionaryEnumerator Interface	Lists the dictionary elements
IEnumerable Interface	Defines an enumerator to perform iteration over a collection
ICollection Interface	Specifies the size and synchronization methods for all collections
IEnumerator Interface	Supports iteration over the elements of the collection
IList Interface	Represents a collection of items that can be accessed by their index number

Table 9.5: Classes and Interfaces of the System.Collections Namespace

Code Snippet 12 demonstrates the use of the commonly used classes and interfaces of the System.Collections namespace.

Code Snippet 12

```
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("the developer are going to delete record
                           containing ID: " + id);
        Dictionary.Remove(id);
    }
    public void GetDetails() {
        IDictionaryEnumerator objEnumerate =
            Dictionary.Get.GetEnumerator();
        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 12, 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 9.12 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 9.12: System.Collections Namespace Example

9.4.2 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 the developer to define data structures that consist of functionalities which can be implemented for any data type. Thus, generics allow the developer to reuse a code for different data types.

To create generics, the developer 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.

9.4.3 Classes and Interfaces

The System.Collections.Generic namespace consists of classes and interfaces that define different generic collections.

➤ Classes

The System.Collections.Generic namespace consists of classes that allow the developer to create type-safe collections. Table 9.6 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 First In First Out (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

Table 9.6: 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 9.7 lists some of the commonly used ones.

Interface/Struct	Description
<code>ICollection Interface</code>	Defines methods to control different generic collections
<code>IEnumerable Interface</code>	Is an interface that defines an enumerator to perform an iteration of a collection of a specific type
<code>IComparer Interface</code>	Is an interface that defines a method to compare two objects
<code>IDictionary Interface</code>	Represents a generic collection consisting of the key and value pairs
<code>IEnumerator Interface</code>	Supports simple iteration over elements of a generic collection
<code>IList Interface</code>	Represents a generic collection of items that can be accessed using the index position
<code>Dictionary.Enumerator Structure</code>	Lists the elements of a Dictionary
<code>Dictionary.KeyCollection.Enumerator Structure</code>	Lists the elements of a Dictionary.KeyCollection
<code>Dictionary.ValueCollection.Enumerator Structure</code>	Lists the elements of a Dictionary.ValueCollection
<code>KeyValuePair Structure</code>	Defines a key/value pair

Table 9.7: Interfaces and Structures in the System.Collections.Generic Namespace

Code Snippet 13 demonstrates the use of the commonly used classes, interfaces, and structures of the `System.Collection.Generic` namespace.

Code Snippet 13
<pre>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;</pre>

```

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 13, 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 9.13 displays the **System.Collections.Generic** namespace example.

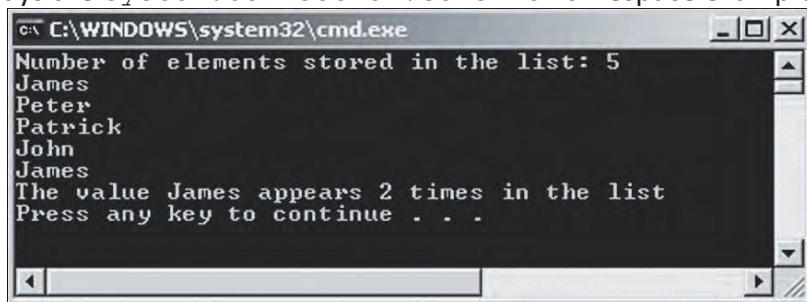


Figure 9.13: System.Collections.Generic Namespace Example

9.4.4 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 the developer to specify the size of the collection during program execution.

The `ArrayList` class allows the developer 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 the developer 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, the developer does not have to 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 the developer to perform actions such as adding, removing, and copying elements in the list. Table 9.8 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 9.8: Methods of ArrayList Class

➤ Properties

The properties of the `ArrayList` class allow the developer to count or retrieve the elements in the list. Table 9.9 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 9.9: Properties of ArrayList Class

Code Snippet 14 demonstrates the use of the methods and properties of the `ArrayList` class.

Code Snippet 14

```
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 14, 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 a developer tries referencing an element at a position greater than the list's size, the C# compiler generates an error.

Code Snippet 15 demonstrates the use of methods of the `ArrayList` class.

Code Snippet 15

```
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 : " +
                           objCustomers.IsFixedSize);
        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 15, 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 9.14 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 9.14: Methods of `ArrayList` Class

9.4.5 Hashtable Class

Consider the reception area of a hotel where the developer finds 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 9.15 demonstrates a real-world example of unique keys.

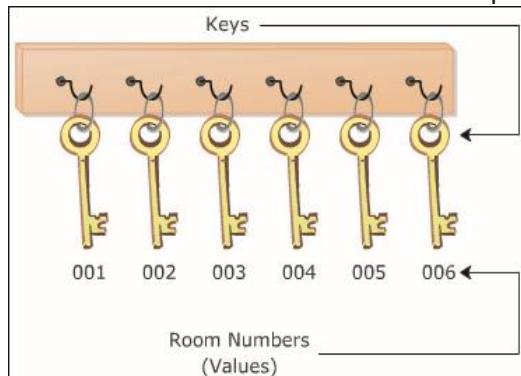


Figure 9.15: Unique Keys

Similar to the keyholder, the `Hashtable` class in C# allows the developer 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 the developer searches 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 9.16 represents the Hashtable.

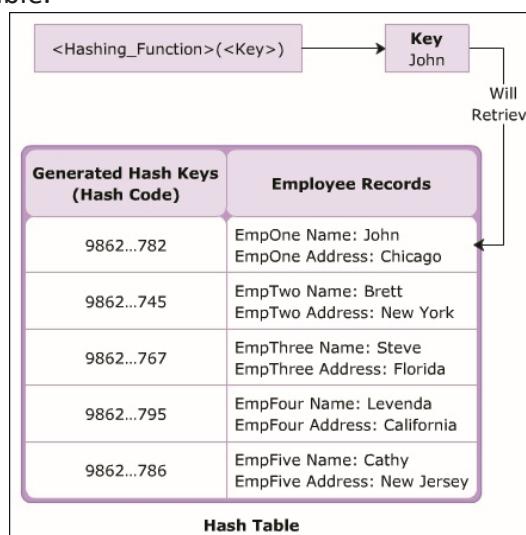


Figure 9.16: Representation of a Hashtable

The `Hashtable` class consists of different methods and properties that are used to add and manipulate the data within the hashtable.

Methods

The methods of the `Hashtable` class allow the developer to perform certain actions on the data in the hashtable. Table 9.10 displays the commonly used methods of the `Hashtable` class.

Method	Description
<code>Add</code>	Adds an element with the specified key and value
<code>Remove</code>	Removes the element having the specified key
<code>CopyTo</code>	Copies elements of the hashtable to an array at the specified index
<code>ContainsKey</code>	Checks whether the hashtable contains the specified key
<code>ContainsValue</code>	Checks whether the hashtable contains the specified value
<code>GetEnumerator</code>	Returns an <code>IDictionaryEnumerator</code> that traverses through the <code>Hashtable</code>

Table 9.10: Methods of Hashtable Class

Properties

The properties of the `Hashtable` class allow the developer to access and modify the data in the hashtable. Table 9.11 displays the commonly used properties of the `Hashtable` class.

Property	Description
<code>Count</code>	Specifies the number of key and value pairs in the hashtable
<code>Item</code>	Specifies the value, adds a new value or modifies the existing value for the specified key
<code>Keys</code>	Provides an <code>ICollection</code> consisting of keys in the hashtable
<code>Values</code>	Provides an <code>ICollection</code> consisting of values in the hashtable
<code>IsReadOnly</code>	Checks whether the <code>Hashtable</code> is read-only

Table 9.11: Properties of Hashtable Class

Code Snippet 16 demonstrates the use of methods and properties of the `Hashtable` class.

Code Snippet 16

```
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 hash table: " +
    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 16, 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 17 demonstrates the use of methods and properties of the `Hashtable` class.

Code Snippet 17

```

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\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 17, 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 9.17 displays the `Hashtable` example.

```

c:\> 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 9.17: Hashtable Example

9.4.6 *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 the developer accesses elements using their keys, the `SortedList` class behaves as a

hashtable, whereas if the developer accesses elements based on their index number, it behaves as 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

Methods of the `SortedList` class allow the developer to perform certain actions on the data in the sorted list. Table 9.12 displays the commonly used methods of the `SortedList` class.

Method	Description
Add	Adds an element to the sorted list with specified key and value
Remove	Removes the element having specified key from the sorted list
GetKey	Returns the key at specified index position
GetByIndex	Returns the value at 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 9.12: SortedList Class Methods

➤ Properties

Properties of the `SortedList` class allow the developer to access and modify the data in the sorted list. Table 9.13 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 existing value for the specified key
Keys	Returns the keys in the sorted list
Values	Returns the values in the sorted list

Table 9.13: SortedList Class Properties

Code Snippet 18 demonstrates the use of the methods and properties of the `SortedList` class.

Code Snippet 18

```
using System;
using System.Collections;
class SortedCollection {
    static void Main(string[] args) {
```

```

        SortedList objSortList = new SortedList();
        objSortList.Add("John", "Administration");
        objSortList.Add("Jack", "HumanResources");
        objSortList.Add("Peter", "Finance");
        objSortList.Add("Joel", "Marketing");
        Console.WriteLine("Original values stored in the sorted list");
        Console.WriteLine("Key \t\t Values");
        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\t Values");
        for (int i = 0; i < objSortList.Count; i++) {
            Console.WriteLine(objSortList.GetKey(i) + "\t\t" +
                objSortList.GetByIndex(i));
        }
    }
}

```

In Code Snippet 18, 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 19 demonstrates the use of methods in the `SortedList` class.

Code Snippet 19

```
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" + 
                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" + 
                objCountries.GetByIndex(i));
        }
    }
}
```

Figure 9.18 displays the `SortedList` class example.

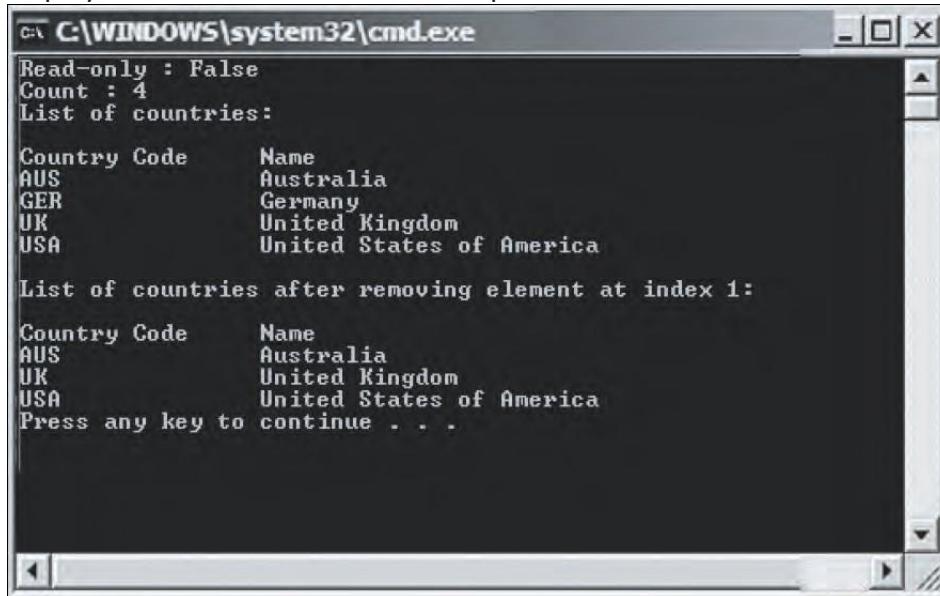


Figure 9.18: `SortedList` Class Example

In Code Snippet 19, 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.

9.4.7 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 the developer adds to the dictionary consists of a value, which is associated with its key.

The developer can retrieve a value from the dictionary by using its key. 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 the developer to perform certain actions on the data in the collection. Table 9.14 displays the commonly used methods of the `Dictionary` generic class.

Method	Description
<code>Add</code>	Adds the specified key and value in the collection
<code>Remove</code>	Removes the value associated with the specified key
<code>ContainsKey</code>	Checks whether the collection contains the specified key
<code>ContainsValue</code>	Checks whether the collection contains the specified value

Method	Description
GetEnumerator	Returns an enumerator that traverses through the Dictionary
GetType	Retrieves the Type of the current instance

Table 9.14: Methods of Dictionary Generic Class

➤ **Properties**

Properties of the Dictionary generic class allow the developer to modify the data in the collection. Table 9.15 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 9.15: Properties of Dictionary Generic Class

Code Snippet 20 demonstrates the use of the methods and properties of the Dictionary class.

Code Snippet 20
<pre>using System; using System.Collections; class DictionaryCollection { static void Main(string[] args) { Dictionary<int, string> objDictionary = new Dictionary<int, string>(); objDictionary.Add(25, "Hard Disk"); 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"); } } }</pre>

```

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 20, 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 21 demonstrates the use of methods in `Dictionary` generic class.

Code Snippet 21

```

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.Get.GetEnumerator();
        while (objDictionayEnumerator.MoveNext())
        {
            Console.WriteLine(objDictionayEnumerator.Key.ToString()
+ "\t\t" + objDictionayEnumerator.Value);
        }
    }
}

```

In Code Snippet 21, the `Add()` method inserts values into the list. The `GetType()` method returns the type of object.

Figure 9.19 displays the use of `Dictionary` generic class.

Keys	Values
201	Gear Box
220	Oil Filter
330	Engine
305	Radiator
303	Steering

Figure 9.19: Use of Dictionary Generic Class

9.4.8 Collection Initializers

Collection initializers allow adding elements to the collection classes of the `System.Collections` and `System.Collections.Generic` namespaces that implement the `IEnumerable` interface using element initializers. The element initializers can be a simple value, an expression, or an object initializer. When a programmer uses a 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 22 uses a collection initializer to initialize an `ArrayList` with integers.

Code Snippet 22

```
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 22, 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 a collection often contains objects, collection initializers accept object initializers to initialize a collection. Code Snippet 23 shows a collection initializer that initializes a generic `Dictionary` object with integer keys and `Employee` objects.

Code Snippet 23

```
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 23 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.

9.5 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 the developer 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 the developer 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.

9.6 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. Peter is trying to convert the weight of 10.2 kilograms to pounds using delegates in a C# application. Which of the following codes will help Peter to achieve this?

(A)	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)); } }
(B)	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(weightKg)); Console.WriteLine("Weight in Kilograms: " + weightKg); Console.Write("Corresponding weight in pounds: "); } }

(C)	<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 Conversion(WeightConversion); Console.WriteLine("Weight in Kilograms: " + weightKg); Console.Write("Corresponding weight in pounds: "); Console.WriteLine(objConvert(weightKg)); } } </pre>
(D)	<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)); } } </pre>

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

3. Which of these statements about events and 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 <code>abstract</code> or <code>sealed</code>
(D)	Events cannot be declared in interfaces
(E)	All 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. Peter is trying to display the output "This is a demonstration of events" using events in a C# application. Which of the following codes will help Peter to achieve this?

(A)	<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>
(B)	<pre>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"); } }</pre>
(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. Peter is trying to display output as Hard Disk, Memory, and MotherBoard. Which of the following codes will help Peter to achieve this? (assume that required namespaces are present)

(A)	<pre>class ArrayCollection { static void Main(string[] args) { ArrayList objArray = new ArrayList(); objArray.Add("Hard Disk"); 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); } } }</pre>
(B)	<pre>class ArrayCollection { static void Main(string[] args) { ArrayList objArray = new ArrayList(); objArray.AddItem("Hard Disk"); objArray.AddItem("Memory"); 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) { 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); } } }</pre>
-----	--

(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 as a hashtable when the developer accesses 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

9.6.1 Answers

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

Try It Yourself

1. Develop a simple C# program for managing events at a conference. Create a system that allows attendees to register for events and receive event notifications.

Define an Event class that contains following properties:

- EventName (string): The name of the event.
- EventDate (DateTime): The date and time of the event.

Create a Conference class to manage events. This class should include a collection (List) to store the registered events and attendees.

Implement a delegate named `EventHandler` that represents a method to handle event notifications. It should take an event name and a message as parameters.

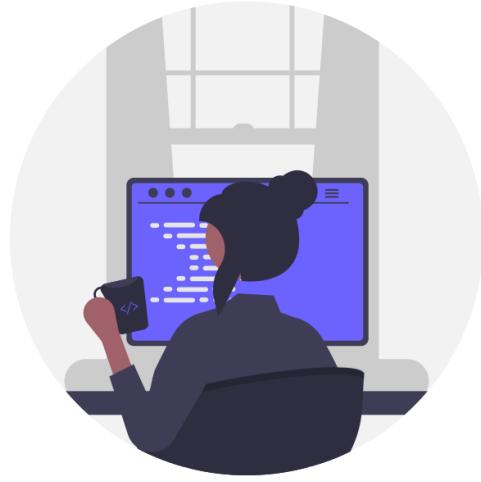
Define an event named `EventNotification` in the `Conference` class. This event should use the `EventHandler` delegate and should be triggered when an event is added to the collection of registered events.

Create a method in the `Conference` class to register attendees for events. When an attendee registers for an event, add the event to the collection, and raise the `EventNotification` event with a message containing the event name.

In the `Main()` method, demonstrate the use of this system by creating a `Conference` instance, registering attendees for various events, and handling event notifications.

Your solution should showcase the use of events, delegates, and collections in C# while providing a practical example of event management for a conference.

2. Create a C# program that simulates a simple notification system. You must use events and delegates to manage a collection of subscribers and notify them when a new message is published.



Session 10

Generics and Iterators

Welcome to the Session, **Generics and Iterators**.

Generics are data structures that allow the developer 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

10.1 Generics

Generics are a kind of parameterized data structures that can work with value types as well as reference types. A developer 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 the developer to cast any value to and from `Object`. If the developer enters 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 the developer to define generalized type templates based on which the type can be constructed later.

10.1.1 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 the developer to create dynamic and read-only generic collections. The `System.Collections.Generic` namespace consists of classes and interfaces that allow the developer to define customized generic collections.

➤ **Classes**

The `System.Collections.Generic` namespace consists of classes that allow the developer to create type-safe collections. Table 10.1 lists some of the widely used classes of the `System.Collections.Generic` namespace.

Class	Descriptions
Comparer	Is an abstract class that allows the developer 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 the developer to create a generic collection by implementing the functionalities of the <code>IEqualityComparer</code> interface

Table 10.1: Classes of System.Collections.Generic Namespace

➤ **Interfaces**

The `System.Collections.Generic` namespace consists of interfaces that allow the developer to create type-safe collections. Table 10.2 lists some of the widely used interfaces of the `System.Collections.Generic` namespace.

Interface	Description
<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 10.2: Interfaces of System.Collections.Generic Namespace

10.1.2 System.Collections.ObjectModel

The `System.Collections.ObjectModel` namespace consists of classes that can be used to create customized generic collections.

Table 10.3 shows the classes contained in the System.Collections.ObjectModel namespace.

Class	Description
Collection<>	Provides the base class for generic collections
KeyedCollection<>	Provides an abstract class for a collection whose keys are associated with values
ReadOnlyCollection<>	Is a read-only generic base class that prevents modification of collection

Table 10.3: 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 read only
            collection");
        foreach (string str in objReadOnly) {
            Console.WriteLine(str);
        }
        Console.WriteLine();
        Console.WriteLine("Total number of elements in the read only
            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 10.1 displays the output of Code Snippet 1.

```
C:\WINDOWS\system32\cmd.exe  
Values stored in the read only collection  
Francis  
James  
Baptista  
Micheal  
Total number of elements in the read only collection: 4  
Values stored in the list after modification  
Francis  
James  
Peter  
Baptista  
Micheal  
Total number of values that can be stored in the new array: 10  
Values in the new array  
null  
null  
null  
null  
null  
Francis  
James  
Peter  
Baptista  
Micheal  
Press any key to continue . . .
```

Figure 10.1: Output of Code Snippet 1

10.1.3 Creating Generic Types

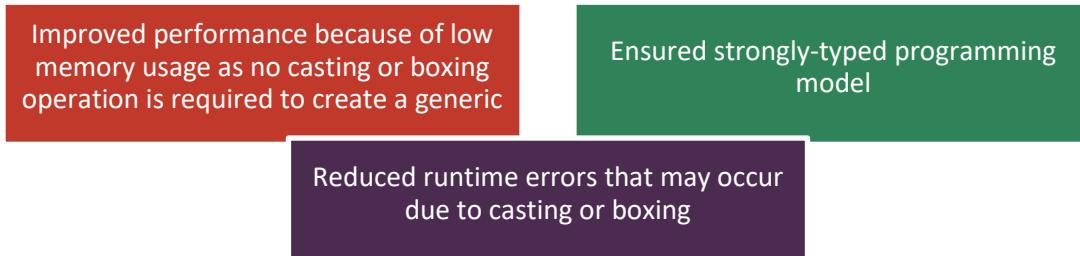
A generic declaration always accepts a **type parameter**, which is a placeholder for the required data type. The type is specified only when a generic type is referred to or constructed as a type within a program.

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.

10.1.4 Benefits

Generics ensure type-safety at compile-time. Generics allow the developer 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:



10.2 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.

10.2.1 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, the developer 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, the developer must generalize the data types into the type parameter and optionally decide the constraints to be applied on the type parameter.

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 values supplied by the user through the object. The `Main()` method of class `Students` creates an instance of generic class `General` by providing 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 do not require to 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.

10.2.2 Constraints on Type Parameters

The developer 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 10.4 lists the types of constraints that can be applied to the type parameter.

Constraint	Description
<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 10.4: 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 to 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 the developer uses 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.

10.2.3 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, the developer 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, the developer 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 10.2 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 10.2: Generic Class as Base Class

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.

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`.

10.2.4 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 of parameter list enclosed within angular brackets. Defining methods with type parameters allows the developer to call the method with a different type every time. The developer can declare a generic method within generic or non-generic class declarations. When the developer declares 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 following keywords:

<code>virtual</code>	<code>override</code>	<code>abstract</code>
The generic methods declared with the <code>virtual</code> keyword can be overridden in the derived class.	The generic method declared with the <code>override</code> keyword overrides the base class method. However, while overriding, the method does not specify the type of parameter constraints since the constraints are overridden from the overridden method.	The generic method declared with the <code>abstract</code> keyword contains only the declaration of the method. Such methods are typically implemented in a derived class.

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

10.2.5 Generic Interfaces

Generic interfaces are useful for generic collections or generic classes representing the items in the collection. The developer 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. 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
```

10.2.6 Generic Interface Constraints

The developer can specify an interface as a constraint on a type parameter. This enables him/her 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.

The developer 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
<pre>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++) {</pre>

```

        _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 as 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 10.3 creates a generic interface.

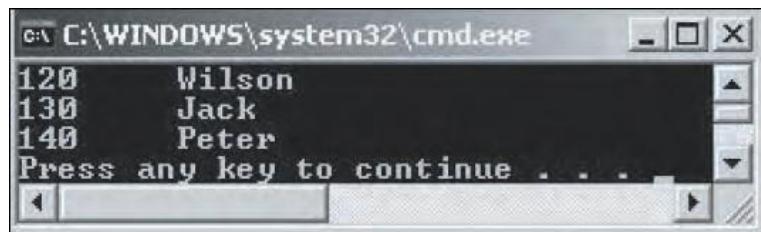


Figure 10.3: Generic Interface

10.2.7 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.

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 10.4 declares a generic delegate.

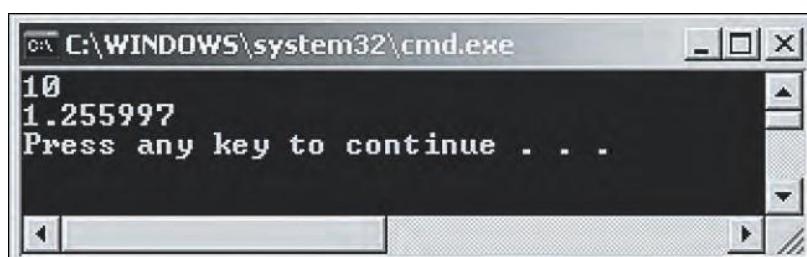


Figure 10.4: Generic Delegate

10.2.8 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

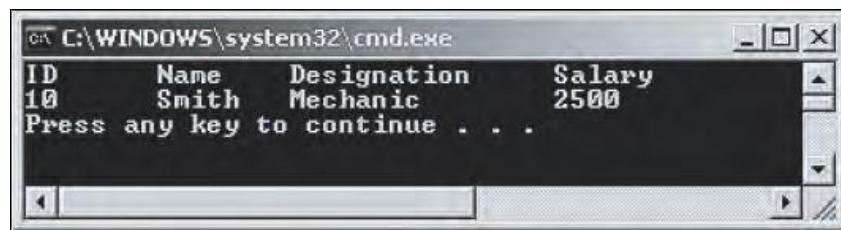
```
using System;
using System.Collections;
using System.Collections.Generic;
class General<T, U> {
    T _valOne;
    U _valTwo;
    public void AcceptValues(T item) {
        _valOne = item;
    }
    public void AcceptValues(U item) {
        _valTwo = item;
    }
    public void Display() {
        Console.WriteLine(_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.WriteLine("\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 10.5 displays the output of using overload methods with type parameters.



ID	Name	Designation	Salary
10	Smith	Mechanic	2500

Press any key to continue . . .

Figure 10.5: Overloaded Methods Using Type Parameters

10.2.9 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 T Value;
    public Student(T valOne, T valTwo) : 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\t\tGender");
        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 gender 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.

The output will display:

Name	Gender
Patrick	Male

10.3 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 must be sorted. To implement the logic manually, a programmer can iterate through each value sequentially using iterators to compare the values.

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 the developer 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 the developer 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.

10.3.1 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 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 a large number of values.

Iterators can be used to evaluate and return only those values that are required.

Iterators can return values without consuming memory by referring to each value in the list.

10.3.2 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 10.6 displays the use of iterators.

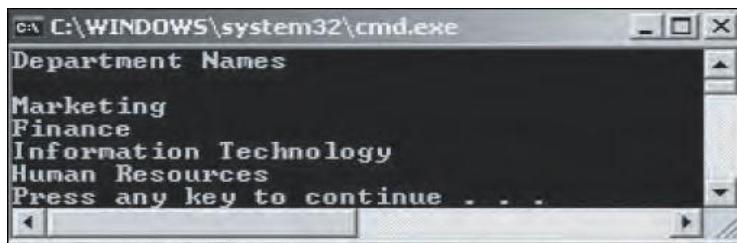


Figure 10.6: Use of Iterators

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.

10.3.3 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 10.7 creates a generic iterator.



Figure 10.7: 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 different department names within the array. Another object of the **GenericDepartment** class is created, that refers to different department IDs within the array.

10.3.4 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 the developer to fetch the required values from the collection.

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;
using System.Collections;
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
```

10.4 Summary

- Generics are data structures that allow the developer 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.

10.5 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 necessity for explicit or implicit casting
(D)	Generics are verified at runtime
(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 <code>override</code> keyword that requires to explicitly specify the type parameter constraints

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

4. Peter is trying to display the area and volume of a cube by implementing a generic interface. Which of the following codes will help Peter to achieve this? (assume required namespaces are present in the code)

(A)	<pre>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)); } }</pre>
-----	--

(B)	<pre> 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)); } } </pre>
(C)	<pre> 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)); } } </pre>

(D)	<pre> 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)); } } </pre>
-----	--

(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 <code>GetEnumerator()</code> method of the <code>IEnumerable</code> interface
(E)	Iterators use the <code>yield break</code> statement to end the iteration process

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

6. Peter is trying to display different programming languages as 'C', 'C++', 'Java', and 'C#'. Which of the following codes will help Peter to achieve this?

(A)	<pre>using System; using System.Collections; 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>
(B)	<pre>using System; using System.Collections; 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>

```

using System;
using System.Collections;
class Languages : IEnumerator
{
    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();
    }
}

```

```

using System;
using System.Collections;
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();
    }
}

```

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

10.5.1 Answers

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

Try It Yourself

1. Create a generic collection and an iterator to traverse through the elements. Develop a program that can be used to manage a collection of various types of items and iterate through them using an iterator.

Create a generic class called `ItemCollection<T>` that can store items of type `T`. This class should have methods for adding items to the collection.

Implement an iterator within the `ItemCollection<T>` class to allow iterating through the collection.

Create a simple class `Item` with properties `Name` (string) and `Price` (double).

In the `Main()` method, demonstrate the use of your `ItemCollection<T>` class. Create instances of `ItemCollection<Item>` and add a few `Item` objects to it. Then, use the iterator to traverse through the items and display their names and prices.

Ensure that the generic collection and iterator work for items of various types, not just the `Item` class.

Your solution should illustrate the use of generics and iterators in C# to create a flexible and reusable collection for managing and iterating through different types of items.

2. Create a C# program to manage a collection of items of various types. Implement a generic collection class that can store items of different data types and allows iteration through the items.

Create a generic class called `ItemCollection<T>` that can store items of type `T`. It should have methods to add items to the collection and iterate through them.

Implement an iterator within the `ItemCollection` class to allow looping through the stored items. The iterator should return items one by one.

In your `Main()` method, create instances of `ItemCollection` for different data types (`ItemCollection<int>`, `ItemCollection<string>`, and so on) and add several items to each collection.

Use the `foreach` loop to iterate through the collections and display the items.

Demonstrate that your program can handle different data types with the same generic collection and effectively iterate through the items. Your solution should illustrate how to create a generic collection class and use iterators to work with items of various types in C#.



Session 11

GUI and Connectivity with SQL Database

Welcome to the Session, **GUI and Connectivity with SQL Database**.

This session provides insights into GUI and connectivity with SQL Server databases. It describes how to develop an application using GUI tools. Thereafter, the session explains how to connect a database with an application. Finally, the session concludes with a brief overview of SQL commands.

In this Session, you will learn to:

- Define GUI
- Identify tools for creating GUI applications
- Explain Forms and Controls
- Elaborate on the implementation of GUI in C#
- Explain database connectivity
- Describe SQL Server Database Connectivity using C#
- List SQL commands
- Explain insert, update, and delete operations in an SQL database

11.1 *Introduction to GUI*

A GUI refers to the visual elements within a software application that enable users to interact with the program. GUI applications provide users with a visual and interactive method to interface with computer software. These applications typically feature elements such as windows, buttons, text boxes, labels, menus, and various other visual components. GUIs enhance user-friendliness by enabling users to directly see and interact with these elements, as opposed to relying on text-based commands.

C# is a versatile programming language that can also be used for developing applications with

rich and interactive GUIs.

Why C# for GUI Development?

C# is a popular choice for GUI development because it is part of the .NET framework, which includes libraries and tools specifically designed for creating GUIs. C# applications often use Windows Forms (WinForms), Windows Presentation Foundation (WPF), or .NET Multi-platform App UI (MAUI) for building GUIs.

These tools/frameworks are explained briefly as follows:

➤ WinForms:

WinForms is a part of the .NET Framework that allows developers to create rich and interactive desktop applications for Windows. It is a GUI framework and provides a set of controls and components for designing the user interface of an application.

It provides a drag-and-drop designer for creating forms and adding controls (buttons and text boxes, Event-driven programming is used to handle user interactions (For example, button clicks). WinForms applications have a classic and native Windows look and feel.

➤ WPF:

WPF is another GUI framework for C# that provides a more advanced and flexible way to design GUIs.

It uses eXtensible Application Markup Language (XAML) for declarative UI design, which separates the UI from code. WPF supports rich data binding, animations, and 3D graphics, making it suitable for creating modern and visually appealing applications.

➤ .NET MAUI (Multi-platform App UI):

.NET MAUI is a cross-platform framework for creating mobile and desktop applications. It allows developers to use a single codebase to target multiple platforms, including Android, iOS, macOS, and Windows. .NET MAUI is designed for building modern, responsive, and native-like user interfaces.

11.1.1 Elements of GUI in C#

GUI development in C# allows the developer to create a wide range of applications, from business software to games and multimedia applications. It is an essential aspect of modern software development that emphasizes user interaction and experience.

Various elements in a GUI application designed with C# can include:

Forms:

- The main windows or screens of an application where the developer place various controls.

Controls:

- Interactive elements such as buttons, text boxes, labels, and menus that users can interact with.

Events:

- Actions or triggers (For example, button clicks) that initiate specific behaviors in response to user actions.

Layout and Design:

- Arranging controls and elements to create an intuitive and appealing interface.

Data Binding:

- Connecting data sources (For example, databases or collections) to display information in the GUI.

User Experience (UX) Design:

- Considering usability
- Aesthetics
- User interactions to create a positive user experience

Developing GUIs in C# involves using some or all these elements - creating forms, adding controls, writing event handlers to respond to user actions, and designing the layout. A typical workflow includes using a visual designer, writing C# code to handle events, and testing the GUI to ensure a smooth user experience.

11.1.2 Key Features of WinForms Apps

Key features of WinForms applications include:

- **Drag-and-Drop UI Design:** WinForms applications make it easy to design the user interface using a visual designer, where developers can drag and drop controls such as buttons, text boxes, and labels easily onto the form.
- **Event-Driven Programming:** C# and WinForms are built around the event-driven programming model. Developers can write event handlers for user actions such as button clicks and the framework handles events and user interactions.
- **Rich Set of Controls:** WinForms provides a wide variety of controls (buttons, text boxes, list boxes, and so on.) that one can use to create complex and interactive user interfaces.
- **Data Binding:** WinForms applications support data binding, making it straightforward to connect the UI with data sources such as databases or collections.
- **Accessibility and Localization:** WinForms includes features to make applications accessible to individuals with disabilities and to localize applications for different languages and regions.
- **Compatibility:** While WinForms is a mature technology, it is still supported and widely used for building Windows desktop applications, including in the latest versions of Windows.

11.1.3 Forms and Controls

In C# and other .NET languages, Forms and Controls are fundamental concepts used in creating GUIs for desktop applications.

These are explained as follows:

1. Forms:

In C# and .NET Framework, a Form is a top-level window that serves as the primary container for user interface elements, or Controls. Forms are the windows users see and interact with inside a desktop application.

Here are some key characteristics of Forms:

Top-Level Windows	Forms are top-level windows, meaning they can be independently opened, closed, minimized, and maximized. Each Form typically represents a distinct screen or section of an application.
Containers	Forms can act as containers for other Controls. The developer can place buttons, text boxes, labels, and other GUI elements directly on a Form.
Events	Forms can handle events such as button clicks, mouse movements, keyboard input, and more. The developer writes event handlers in the code to specify how the Form should respond to these events.
Properties	Forms have properties that control their appearance and behavior. You can set properties such as size, title, icon, background color, and more to customize the Form's look and feel.
Lifecycle Events	Forms have lifecycle events such as Load, Activate, and Closing, which allows the developer to perform specific actions when the Form is created, brought to the foreground, or closed.

2. Controls:

In C# and .NET, Controls are individual graphical elements that you place on Forms to create the user interface of the application. Controls are the building blocks of your GUI.

There are various types of Controls, including:

Button	A clickable button that triggers an action when pressed.
Label	A non-editable text element used for displaying information.
TextBox	An editable text input field.
ComboBox	A drop-down list for selecting options.
ListBox	A list for displaying and selecting items.
CheckBox	A checkable box for binary choices.
RadioButton	A selectable option within a group of choices.
PictureBox	A container for displaying images.
Panel	A container for grouping and organizing Controls.
MenuStrip	A menu bar for creating menus in your application.
ToolStrip	A toolbar for adding icons and buttons.

Controls have properties that define their appearance and behavior, such as size, location, text, and event handlers. The developer can interact with Controls through events such as button clicks, text changes, or mouse movements.

How Forms and Controls Work Together?

Developer designs the user interface of the application by creating Forms and adding Controls to them. Forms act as the main windows or screens of the application, while Controls are the elements users interact with. The developer writes code to handle events raised by Controls when users interact with them. For example, the developer writes code in response to a button click event to perform a specific action.

Forms and Controls are a powerful combination for building GUI-based desktop applications in C#. They allow the developer to create rich, interactive, and user-friendly software interfaces for various purposes, from business applications to games and more.

11.1.4 Implementation of a GUI Application

Implementing a GUI in C# typically involves using one of the available GUI frameworks, such as WinForms, WPF, or .NET MAUI.

A step-by-step explanation of how to implement a GUI in C# using WinForms is as follows:

1. Create a New Project:

- Open Visual Studio 2022.
- Create a new project and select the appropriate project type for the GUI application (For example, Windows Forms App). Refer to Figures 11.1 and 11.2.

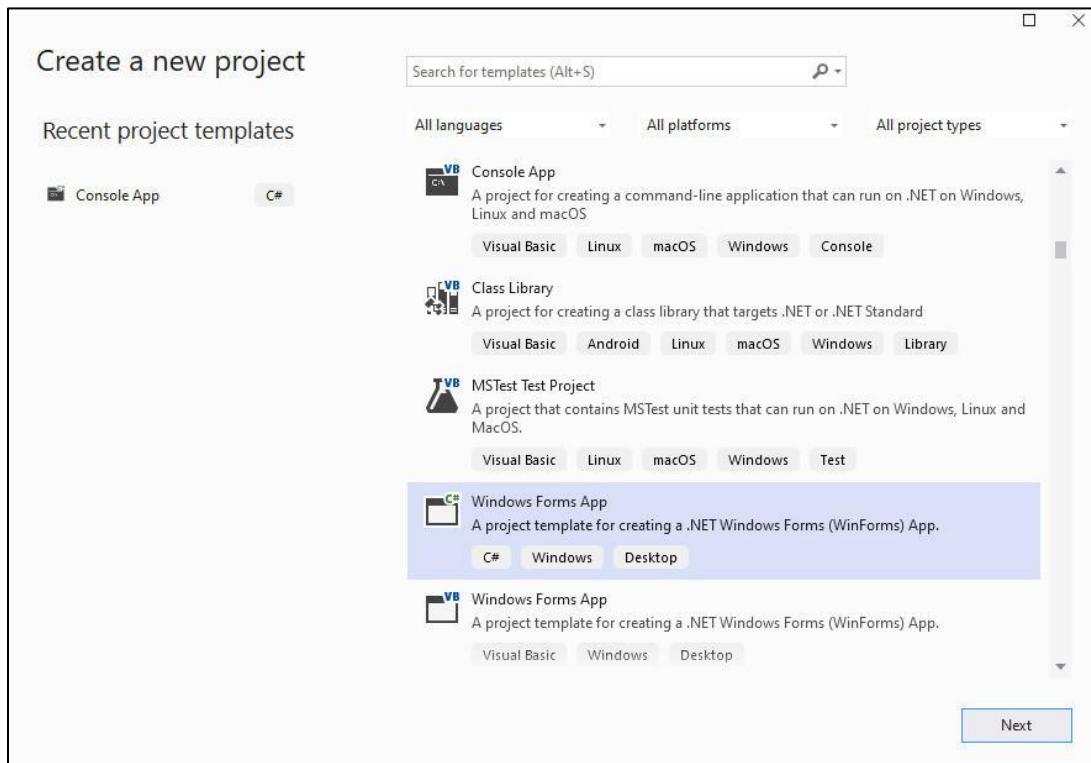


Figure 11.1: Create a Project

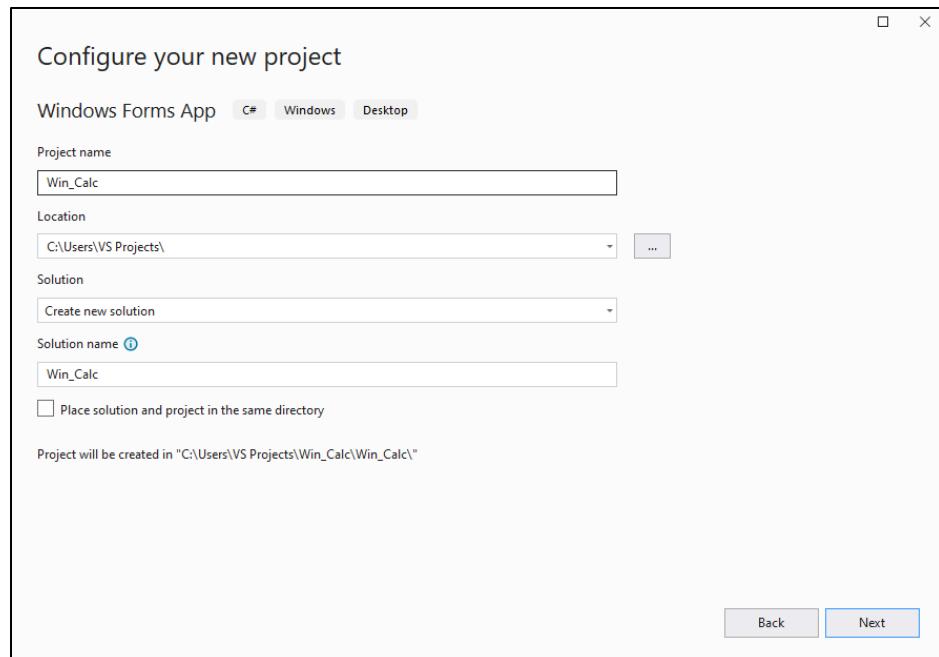


Figure 11.2: Configure the Project

2. Design the User Interface:

- Use the built-in GUI designer or form designer provided by the IDE.
- Drag and drop Controls (buttons, text boxes, labels, and so forth) onto the Form to create the desired layout and user interface.
- Set properties for the Form and Controls to define their appearance and behavior. You can use the Properties pane on each control to set its properties such as Name, Text, and so on. Refer to Figure 11.3 to see a sample UI for a Calculator application.



Figure 11.3: Designing the User Interface

3. Handle Events:

Write event handlers for the Controls to specify how the application should respond to user actions. For example, one might write code to respond to button clicks, text input changes, or mouse movements. Figure 11.4 shows the **Events** tab in the **Properties** pane. When each event in this tab is double-clicked, an event handler for that event is automatically generated in C# code. Developers can then add their desired logic. For each of the buttons in the Calculator application, an event handler must be created.

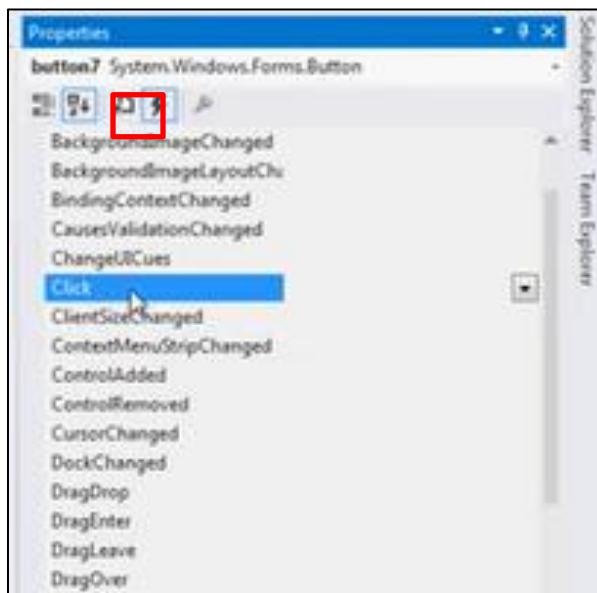


Figure 11.4: Events Tab for Adding Event Handlers

4. Implement Functionality:

Write the core logic of the application. This includes processing user input, interacting with data sources (such as databases) if required, performing calculations, and updating the user interface. In the example of the Calculator application, there is no data source involved. However, the core logic including various arithmetic operations can be added.

5. Testing and Debugging:

- Test the application thoroughly, checking for proper behavior, user interface responsiveness, and any issues that require debugging.
- Use debugging tools to identify and resolve errors in code.

6. Building and Deployment:

- Once the GUI application is complete and thoroughly tested, build the project to generate an executable file.
- Distribute the application to end-users, either by providing the executable or by creating an installer package for deployment.

7. Maintenance and Updates:

Continue to maintain and update the application as required. This includes adding new features, fixing bugs, and ensuring compatibility with new versions of the .NET Framework or operating systems.

Example:

Let us now take an example of how to build a simple calculator App using WinForms App.

Step 1: Create a new project as shown in Figures 11.1 and 11.2.

Step 2: Modify the text property of the form to ‘Win_Calc’ to ensure that the application does not display ‘Form1’ as its title upon launch.

Step 3: Drag and drop a **TextBox** control from the Toolbox onto the form and then, adjust its size. Make modifications to its properties.

Step 4: Let us begin by configuring the display. Typically, when the calculator application is launched, it should show the number 0. However, in the current example, it does not do this. To address this, adjust the **Text** property of the text box and set it to 0 without any leading or trailing spaces.

Most calculators align the number to the right. Locate the **TextAlign** property of the text box in the Properties pane and change it to **Right** to achieve the desired alignment.

Refer to Figure 11.5.

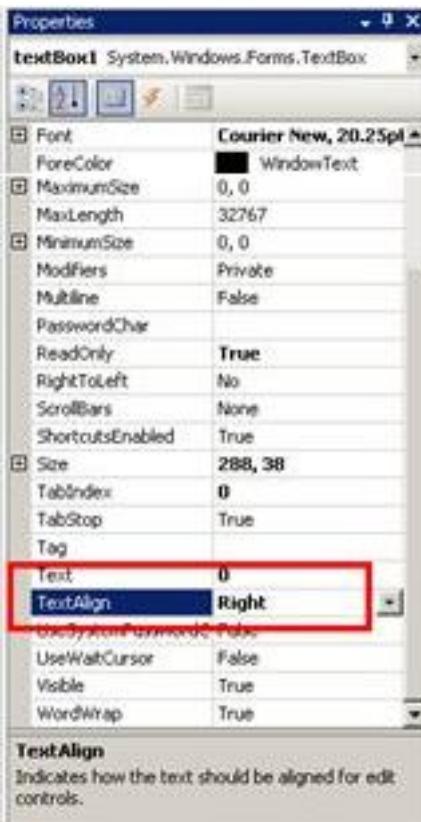


Figure 11.5: TextAlign Property

Step 5: Using the Toolbox window, drag and drop a Button onto the form. Rename it by altering its **Name** property to `btn1`. Similarly, add buttons for all the other digits and rename them as `btn2`, `btn3`, and so on. When the end user clicks a button, this naming convention will make it easier to know which number was clicked.

To add the buttons, the developer has two options to proceed: Either replicate the same process for the remaining nine buttons individually or make copies of the first button and then, rename appropriately. To copy, simply hold the `Ctrl` key and drag and drop the control to another area of the form.

Next, proceed to set the **Text** property of `btn1` as `1`. Next, access the **Font** Property and set the font to Courier New with a suitable font size.

Step 6: Double-click `btn1` to autogenerate an event handler. Alternatively, developer can go to **Events** tab of the button in **Properties** pane and then, double-click the `Click` event.

The event handler method associated with `btn1` is named `btn1_Click`, indicating that it will be executed when the user clicks the button with the name `btn1`.

Code Snippet 1 describes the event handler for `btn1`.

Code Snippet 1

```
private void btn1_Click(object sender, EventArgs e) {
    Button button = (Button)sender;
    string buttonText = button.Text;
    if (txtResult.Text == "0" && txtResult.Text != null) {
        txtResult.Text = buttonText;
    }
    else {
        txtResult.Text += buttonText;
    }
}
```

Step 7: The developer must duplicate the code similarly for the rest of the buttons.

Step 8: Create a button named `btnClear` button for clear operation and position it alongside the 0 button. Set its **Text** property to display C. Additionally, create buttons for addition, subtraction, multiplication, and division, with symbols +, -, and so on as per the calculator UI layout.

Double-click each of the events to edit their associated event handlers. The final code should be similar to Code Snippet 2.

Code Snippet 2

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

namespace CalculatorApp{
    public partial class frmCalc : Form {
        double FirstNumber;
        string Operation;
        public frmCalc() {
            InitializeComponent();
        }
        private void btn1_Click(object sender, EventArgs e) {
            Button button = (Button)sender;
            string buttonText = button.Text;

            if (txtResult.Text == "0" && txtResult.Text != null) {
                txtResult.Text = buttonText;
            }
            else{
                txtResult.Text += buttonText;
            }
        }
        private void btn2_Click(object sender, EventArgs e) {
            Button button = (Button)sender;
            string buttonText = button.Text;
            if (txtResult.Text == "0" && txtResult.Text != null)
            {
```

```
        txtResult.Text = buttonText;
    }
    else {
        txtResult.Text += buttonText;
    }
}

// replicate Click event logic for other number buttons
//similarly
private void btnOperator_Click(object sender, EventArgs e) {
    Button button = (Button)sender;
    FirstNumber = Convert.ToDouble(txtResult.Text);
    txtResult.Text = "0";
    Operation = button.Text;
}

private void btnClear_Click(object sender, EventArgs e) {
    txtResult.Text = "0";
}

private void EqualButton_Click(object sender, EventArgs e) {
    double SecondNumber = Convert.ToDouble(txtResult.Text);
    double Result = 0.0;
    switch (Operation)
    {
        case "+":
            Result = FirstNumber + SecondNumber;
            break;
        case "-":
            Result = FirstNumber - SecondNumber;
            break;
        case "*":
            Result = FirstNumber * SecondNumber;
            break;
        case "/":
            if (SecondNumber == 0) {
                txtResult.Text = "Cannot divide by zero";
                return;
            }
            Result = FirstNumber / SecondNumber;
            break;
    }
    txtResult.Text = Convert.ToString(Result);
    FirstNumber = Result;
}
}
```

Code Snippet 2 shows the code to simulate a calculator using a simple WinForms application. It allows users to enter a basic arithmetic expression and evaluate it by clicking **Equals**. The application currently supports only addition. The program defines a WinForms application named **frmCalc**. It includes a single form called **CalculatorForm**.

The form has buttons for numbers (0-9), operators (+), the equals sign (=), and a text box for input and display. When the user clicks a number or operator button, the corresponding character is appended to the input displayed in the text box.

When the user clicks **Equals**, the program attempts to evaluate the expression entered in the text box. If the expression is successfully evaluated, the result is displayed in the text box.

Based on this snippet, developers can further add logic for the other arithmetic operators and complete the creation of the whole application.

11.2 Database Connectivity

Database connectivity in C# refers to the process of establishing a connection between a C# application and a Database Management System (DBMS) to exchange data. This allows your C# application to store, retrieve, update, and manage data in a database.



Steps to implement database connectivity in a C# application are as follows:

1. Choose a DBMS:

Prior to establishing a connection with a database, the developer should make a selection of a DBMS, with popular options often including Microsoft SQL Server. Ensure databases and tables are created appropriately and data is populated in case you want to perform retrieval operations.

2. Import Database Libraries:

In C#, the developer must use libraries or data access frameworks to interact with the chosen database. The most common library for working with databases is ActiveX Data Objects for .NET (ADO.NET), which is part of the .NET Framework. Depending on the DBMS, the developer might also require a specific database provider or driver. ADO.NET is used to provide consistent and reliable access to data sources. ADO.NET consists of two main data access components namely, data providers and Dataset. The data providers are used to connect to a database, to run commands in the database, and finally, to retrieve the results of the operation performed. These

retrieved results can be processed directly or can be placed in a dataset object. Dataset helps in storing the records that are retrieved from the data source.

3. Establish a Connection:

To establish a connection to the database, one should typically use connection strings that contain information such as the database server address, credentials (username and password), and database name. The developer creates a connection object using the connection string and opens the connection.

Code Snippet 3 describes how to establish a connection with a database in C#.

Code Snippet 3

```
using System.Data.SqlClient; // For SQL Server  
...  
string connectionString = "Data Source=WorkHorse; Initial Catalog= BankDB;  
User ID=emp; Password=1234Welc0me";  
SqlConnection connection = new SqlConnection(connectionString);  
connection.Open();
```

In Code Snippet 3, the lines using `System.Data.SqlClient;` and using `System.Data:` import the necessary namespaces for working with SQL Server databases and data in C#.

The `System.Data.SqlClient` namespace provides classes for connecting to SQL Server, while `System.Data` contains general data-related classes.

Note that the namespace `System.Data.SqlClient` will not be referenced by the .NET 7.0 or .NET Core projects by default as was done by .NET Framework earlier. Hence, in .NET 7.0 and Core applications, developers must manually add the package to the project. To do this, right-click the project, select **Manage NuGet Packages** and then, search for `System.Data.SqlClient`. Once it is located, install it. The classes and interfaces in this namespace will now be available for use in the project.

Following syntax defines a connection string, which is a text-based configuration that contains all the information required to connect to a SQL Server database:

```
string connectionString = "Data Source=ServerName; Initial  
Catalog=DatabaseName; User ID=UserName; Password=Password";
```

where,

`Data Source` specifies the server or database instance to connect to (replace `"ServerName"` with the actual server name).

`Initial Catalog` indicates the initial database to use.

`User ID` and `Password` provide the authentication credentials.

For Windows authentication mode, a sample connection string would look as follows:

```
SqlConnection con = new SqlConnection("Data Source=WorkHorse;  
Database=BankDB; Integrated Security=SSPI");
```

For SQL Server authentication mode, a sample connection string would look as follows:

```
SqlConnection con = new SqlConnection("Data Source = WorkHorse;Initial  
Catalog=BankDB;User ID=sa;Password=sa123");
```

In Code Snippet 3, following lines create an instance of the `SqlConnection` class and pass the connection string as a parameter. This instance represents a connection to the SQL Server database defined in the connection string.

```
string connectionString = "Data Source=WorkHorse; Initial Catalog= BankDB;  
User ID=emp; Password=1234Welc0me";
```

```
SqlConnection connection = new SqlConnection(connectionString);
```

Finally, the line `connection.Open();` in Code Snippet 3 opens the database connection by calling the `Open()` method on the `SqlConnection` instance. Once the connection is open, the developer can execute SQL commands such as queries or updates on the connected database.

4. Create and Execute SQL Commands:

Once the connection is established, the developer can create SQL commands (For example, `SELECT`, `INSERT`, `UPDATE`, and `DELETE`) to interact with the database. The developer uses command objects to prepare and execute these commands. Code Snippet 4 describes the creation of SQL commands. It is assumed that a table named `Employees` has been created.

Code Snippet 4

```
SqlConnection con = new SqlConnection("Data Source=WorkHorse;  
Database=BankDB; Integrated Security=SSPI");  
  
string sqlQuery = "SELECT * FROM Employees";  
SqlCommand command = new SqlCommand(sqlQuery, connection);  
// Execute the query and retrieve results  
using (SqlDataReader reader = command.ExecuteReader())  
{  
    while (reader.Read())  
    {  
        // Process data  
        int id = reader.GetInt32(0);  
        Console.WriteLine("Employee ID:" + id);  
    }  
}
```

Code Snippet 4 makes use of the `SqlDataReader` class whose main function is to read a stream

of rows from a SQL Server database in a forward-only manner. The class object is created by calling the `ExecuteReader()` method of the `SqlCommand` class. The `SqlConnection` object associated with the `SqlDataReader` object cannot perform any other operations till the time it is being used by the `SqlDataReader` object.

Observe how the `using` statement has been written here. This kind of approach of writing the `using` statement works like this:

The `using` statement obtains one or more resources, executes a statement, and then, disposes of the resource.

5. Retrieve and Process Data:

The developer can use data readers or data adapters to retrieve and process data from the database. Data readers are used for reading data row by row, while data adapters can fill datasets or data tables with query results for further manipulation. Code Snippet 4 includes code for this as well.

6. Handle Transactions:

C# applications often require to perform multiple database operations as part of a single transaction. Transactions ensure that a series of operations either all succeed, or all fail, maintaining data consistency. The developer can manage transactions using `SqlTransaction()` or equivalent methods for other DBMS.

7. Close the Connection:

It is essential to close the database connection when developer has finished, to release resources and prevent potential issues with connection limits or performance. Use `connection.Close()` or `connection.Dispose()` to achieve this, assuming `connection` is the instance of `SqlConnection`. When developers have finished working with the database, they should also make sure to disconnect and dispose of database-related objects such as commands and readers to free up resources and maintain application performance.

8. Error Handling:

Implement error handling to manage exceptions that may occur during database operations. This includes catching exceptions related to network errors, query errors, and security issues. `SqlException` is the most commonly occurring exception that is thrown by the runtime. Developers should always include proper error handling in the code to manage exceptions that may occur during database operations. They should use `try-catch` blocks to handle exceptions gracefully.

9. Security Considerations:

Pay attention to security measures such as parameterized queries, input validation, and access control to protect the application and database from SQL injection attacks and unauthorized access.

Database connectivity is a crucial aspect of many C# applications, especially those that involve data storage and retrieval. The specifics of database connectivity may vary depending on the DBMS used, but general principles of connection, querying, and data handling apply across various database systems in C#.

11.2.1 SQL Commands in C#

SQL commands allow you to execute SQL statements against a database, including SELECT, INSERT, UPDATE, DELETE, and more. The primary class used to work with SQL commands in C# is SqlCommand.

The SqlCommand class is part of the System.Data.SqlClient namespace and is used to execute SQL commands against a SQL Server database. It provides methods and properties for creating and executing SQL queries and stored procedures.

To create a SqlCommand object, the developer has to provide the SQL command text and a connection to the database. Code Snippet 5 describes how the developer can create a SqlCommand.

Code Snippet 5

```
using (SqlConnection connection = new  
        SqlConnection(connectionString)) {  
    connection.Open();  
    string sqlQuery = "SELECT * FROM Employees";  
    SqlCommand command = new SqlCommand(sqlQuery, connection);  
}
```

In Code Snippet 5, connectionString is a connection string that specifies the database server, database name, and authentication details. It is assumed that these details have been specified.

Code Snippet 6 demonstrates how to execute a SQL query, retrieve data from the result set using a SqlDataReader, and process the data within a loop. It also hints at using ExecuteNonQuery() for non-query SQL commands to find out how many records were affected by the command.

Code Snippet 6

```
using (SqlDataReader reader = command.ExecuteReader())  
{  
    while (reader.Read())  
    {  
        // Process data from the reader  
        int id = reader.GetInt32(0);  
        string name = reader.GetString(1);  
    }  
}  
//For non-query commands:  
int rowsAffected = command.ExecuteNonQuery();
```

Passing Parameters

The developer can use parameters to pass values to the SQL command to make the queries more secure and prevent SQL injection attacks.

Note:

SQL injection is a type of cybersecurity attack that targets applications or Websites with vulnerabilities in their handling of SQL queries. This attack occurs when an attacker manipulates user input in a way that allows them to execute malicious SQL statements against a database.

SQL injection attacks can have severe consequences, such as unauthorized access to data, data manipulation, or even unauthorized control over the entire database.

Code Snippet 7 describes how the developer can use parameters.

Code Snippet 7

```
string sqlQuery = "SELECT * FROM Employees WHERE ID = @Id";
SqlCommand command = new SqlCommand(sqlQuery, connection);
command.Parameters.AddWithValue("@Id", 202);
```

Code Snippet 8 describes the `CommandType` property that allows the developer to specify whether the command is a text SQL query, stored procedure, or table.

Code Snippet 8

```
command.CommandType = CommandType.Text; // For text SQL queries
command.CommandType = CommandType.StoredProcedure; // For stored
//procedures
```

The developer can also execute stored procedures using `SqlCommand` class by specifying the stored procedure name and setting `CommandType` property to `CommandType.StoredProcedure`.

Note: Stored procedures are commonly used for various database tasks, including data retrieval, data modification (insert, update, delete), business logic implementation, and complex data processing. They are a valuable tool for improving database performance, security, and maintainability in applications that interact with relational databases.

11.2.2 Insert, Update, and Delete Operations in SQL Database

Developer can perform SQL database operations such as INSERT, UPDATE, and DELETE to manipulate data in an SQL Server database through C#. These operations are essential for creating, modifying, and removing records from database tables.

1. Insert Data:

Inserting data into a database table involves adding new records. Code Snippet 9 describes how the developer can perform an INSERT operation in C#.

Code Snippet 9

```
using (SqlConnection connection = new  
        SqlConnection(connectionString))  
{  
    connection.Open();  
    // Define the SQL INSERT command with parameters  
    string insertQuery = "INSERT INTO Employees (ID, Leaves) VALUES (@62270,  
@30)";  
    SqlCommand insertCommand = new SqlCommand(insertQuery, connection);  
    // Add values for the parameters  
    insertCommand.Parameters.AddWithValue("@62270", 62270);  
    insertCommand.Parameters.AddWithValue("@30", 30);  
    // Execute the INSERT command  
    int rowsInserted = insertCommand.ExecuteNonQuery();  
    Console.WriteLine(rowsInserted);  
}
```

2. Update Data:

Updating data in a database involves modifying existing records. The developer can perform an UPDATE operation in C# as shown in Code Snippet 10. The parameters will be substituted with actual values using `AddWithValue()` method.

Code Snippet 10

```
using (SqlConnection connection = new SqlConnection(connectionString))  
{  
    connection.Open();  
    // Define the SQL UPDATE command with parameters  
    string updateQuery = "UPDATE Employees SET Leaves=@leave WHERE ID =  
@id";  
    SqlCommand updateCommand = new SqlCommand(updateQuery, connection);  
    // Add values for the parameters  
    updateCommand.Parameters.AddWithValue("@id", 5000);  
    updateCommand.Parameters.AddWithValue("@leave", 15); // Execute the  
    //UPDATE command  
    int rowsUpdated = updateCommand.ExecuteNonQuery();  
    Console.WriteLine(rowsUpdated);  
}
```

3. Delete Data:

Deleting data from a database involves removing records. The developer can perform a DELETE operation in C# as shown in Code Snippet 11.

Code Snippet 11

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    // Define the SQL DELETE command with parameters
    string deleteQuery = "DELETE Employees WHERE ID = @id";
    SqlCommand deleteCommand = new SqlCommand(deleteQuery,
        connection);
    // Add values for the parameters
    deleteCommand.Parameters.AddWithValue("@Id", 6000);
    int rowsDeleted = deleteCommand.ExecuteNonQuery();
    Console.WriteLine(rowsDeleted);
}
```

Important Points:

- Always use parameterized queries to prevent SQL injection attacks and ensure that your data is properly escaped and validated.
 - Error handling is critical. Use try-catch blocks to handle exceptions that may occur during database operations.
 - Properly open, close, and dispose off database connections to manage resources efficiently.
 - After executing INSERT, UPDATE, or DELETE commands, the `ExecuteNonQuery` method returns the number of affected rows, allowing you to check if the operation was successful.
 - Consider using transactions for multiple database operations to ensure data consistency and integrity.
 - These operations are fundamental when working with SQL Server databases in C#. By correctly implementing INSERT, UPDATE, and DELETE operations, you can maintain and manage data in your database tables as part of your C# application's functionality.

Code Snippet 12 shows a complete code to perform data operations in C#.

Code Snippet 12

```
using System;
using System.Data.SqlClient;
namespace DataDemo{
class EmpDataDemo {
// Main Method
static void Main() {
using (SqlConnection connection = new
                    SqlConnection("Data Source=
WorkHorse\\SQLEXPRESS01; Database=BankDB;
Integrated Security=SSPI")) {
try{
    connection.Open();
    // Define the SQL INSERT command with parameters
    string insertQuery = "INSERT INTO Employees (ID,
EMP_NAME, Leaves) VALUES (@62270, @NAME, @30)";
    SqlCommand insertCommand = new
    SqlCommand(insertQuery, connection);
```

```
// Add values for the parameters
insertCommand.Parameters.AddWithValue("@62270",
62270);
insertCommand.Parameters.AddWithValue("@NAME",
"JAMES HANHART");
insertCommand.Parameters.AddWithValue("@30", 30);
// Execute the INSERT command
int rowsInserted = insertCommand.ExecuteNonQuery();
Console.WriteLine("Inserted: " + rowsInserted);
// Define the SQL SELECT command to retrieve data
string sqlQuery = "SELECT ID, EMP_NAME FROM
Employees";
SqlCommand command = new SqlCommand(sqlQuery,
connection);
using (SqlDataReader reader =
command.ExecuteReader()) {
while (reader.Read()) {
// Process data from the reader
int id = reader.GetInt32(0);
string name = reader.GetString(1);
}
}
// Define the SQL DELETE command with parameters
string deleteQuery = "DELETE Employees WHERE ID
=@id";
SqlCommand deleteCommand = new
SqlCommand(deleteQuery, connection);
// Add values for the parameters
deleteCommand.Parameters.AddWithValue("@Id", 62270);
int rowsDeleted = deleteCommand.ExecuteNonQuery();
Console.WriteLine(rowsDeleted);
}
catch (Exception ex) {
Console.WriteLine("Exception has occurred" + ex.Message);
}
}
```

The code inserts a record into the table Employees, retrieves data from the table, and then deletes a record.

11.3 Summary

- A GUI provides a visual and interactive way for users to interact with software.
- C# is used with .NET frameworks to build GUIs. Options include WinForms, WPF, and .NET MAUI.
- WinForms is a framework for building Windows desktop applications in C#.
- It offers a drag-and-drop designer for creating forms and controls.
- Event-driven programming handles user interactions with a native Windows look and feel.
- GUI development in C# is essential for building user-friendly applications with a variety of frameworks, allowing developers to create diverse software solutions.
- SQL Server database connectivity in C# involves establishing a robust connection to a SQL Server database and enabling effective data manipulation through a C# application.

11.4 Check Your Progress

1. Which of the following statements is true regarding Forms and Controls in C# GUI applications?

(A)	Forms represent event handlers, while Controls are main windows
(B)	Forms act as main windows and Controls are elements that users interact with
(C)	Forms are responsible for user interaction and Controls handle data storage
(D)	Forms and Controls are terms used interchangeably in C#

2. What is the primary purpose of event handling in C# GUI applications?

(A)	To design visually appealing Forms
(B)	To create databases for storing user data
(C)	To handle user interactions with Controls
(D)	To manage application resources

3. Which feature of WinForms allows developers to visually design the user interface by dragging and dropping controls?

(A)	Event-Driven Programming
(B)	Rich Set of Controls
(C)	Drag-and-Drop UI Design
(D)	Data Binding

4. Which feature of WinForms provides a wide variety of user interface elements, such as buttons, text boxes, and list boxes?

(A)	Drag-and-Drop UI Design
(B)	Event-Driven Programming
(C)	Data Binding
(D)	Rich Set of Controls

5. What operation do you perform in C# to remove records from a SQL database?

(A)	UPDATE
(B)	SELECT
(C)	DELETE
(D)	INSERT

11.4.1 Answers

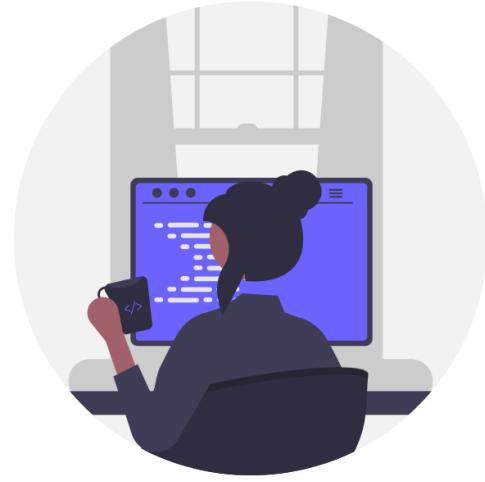
1.	B
2.	C
3.	C
4.	D
5.	C

Try It Yourself

1. Develop a WinForms application that serves as a simple address book. This application should allow users to add, view, and edit contact information, which is stored in an SQL Server database. Design a user-friendly GUI to input and display contact details. Implement connectivity with the SQL database for storing and retrieving contact data.

Ensure following functionality:

- Create a database table to store contact information, including fields for name, email, phone number, and address.
 - Design a GUI with appropriate controls for adding new contacts (text boxes, labels, and a Save button) and viewing existing contacts (a list or data grid).
 - When users enter contact information and click **Save**, the application should insert the data into the database.
 - When users view contacts, the application should retrieve and display contact details from the database.
 - Implement the ability to edit and update existing contact information in the database.
 - Add error handling and validation to ensure data integrity.
 - Test the application to verify that it properly connects to the SQL database and manages contact information.
2. Create a WinForms application for a library management system. The application should allow users to add, view, and remove books from the library's collection. Design the user interface with Forms and Controls to achieve the following:
 - Create a main form that serves as the application's primary window.
 - Add controls to the main form for navigating between different features (For example, buttons for Add Book, View Books, and Remove Book).
 - Implement a form for adding a new book. This form should include text boxes for entering book details (For example, title, author, and ISBN) and a button to save the book.
 - Design a form for viewing the list of books in the library. Use a `DataGridView` control to display the book information.
 - Create a form for removing a book. It should include a dropdown list of available books and a button to remove the selected book from the library.
 - Your task is to create necessary Forms and Controls for this library management system. Describe the Forms and Controls you would use and their functionalities.



Session 12

Advanced Concepts in C#

Welcome to the Session, **Advanced Concepts in C#**.

C# supports several advanced methods and types such as anonymous methods, extension methods, partial methods, data handling methods, and so on. These methods and types greatly enhance the object-oriented programming experience of C# developers.

To facilitate development of applications that cater to various requirements, C# provides advanced language enhancements, such as system-defined delegates, lambda expressions, and so on. The .NET Framework provides the Entity Framework which helps in developing data-centric applications. To create more responsive multi-user applications, C# supports multithreading and concurrent programming.

In this Session, you will learn to:

- Describe anonymous methods and extension methods
- Explain anonymous types, partial types, and nullable types
- Describe system-defined generic delegates
- Define lambda expressions
- Explain query expressions
- Explain parallel and dynamic programming

12.1 *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, to pass a small block of code to a delegate, the developer always had to create a method, and then pass it to the delegate. With the introduction of anonymous methods, the developer can pass an inline block of code to a delegate without actually creating a method.

Figure 12.1 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 12.1: Anonymous Method

12.1.1 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 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.

12.1.2 Creating Anonymous Methods

An anonymous method is created when a delegate is instantiated or referenced with a block of unnamed code. Following points must 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 12.2 displays syntax for 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 12.2: Syntax of Anonymous Methods

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

12.1.3 Referencing Multiple Anonymous Methods

C# allows creating and instantiating 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

12.1.4 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. The parameter values can be passed 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, a pair of empty parentheses can be used in the declaration of the delegate. The anonymous methods without a parameter list cannot be used with delegates that specify out the parameters.

12.2 Extension Methods

Extension methods allow extending 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. An extension method can be declared 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 used 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 12.3 depicts the output of Code Snippet 4.

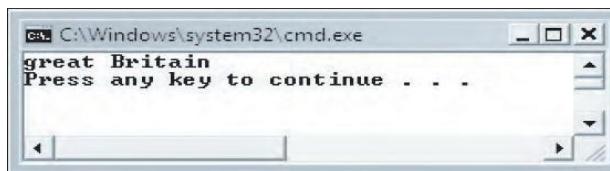


Figure 12.3: Output of Code Snippet 4

Advantages of extension methods are as follows:

- The functionality of the existing type can be extended without modification. This will avoid the problems of breaking source code in existing applications.
- Additional methods can be added 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);
        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)
```

Following lines of code iterate through each value in the collection, remove duplicate values, and store unique values in the List, **finalCities**:

```
foreach (var eachCity in allCities)
if (!finalCities.Contains(eachCity)) finalCities.Add(eachCity);
```

Figure 12.4 displays the output of Code Snippet 5.

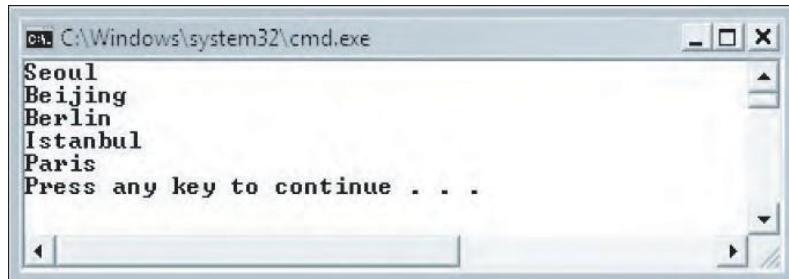


Figure 12.4: Output of Code Snippet 5

Note: Even though the extension method is declared as static, it can still be called using an object.

12.3 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, an implicitly typed variable should be declared 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 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 runtime, the C# compiler creates an instance of this type and the properties are given the values Michigan Enterprises, 1301, and 35056.75 respectively.

Figure 12.5 displays the output of Code Snippet 6.

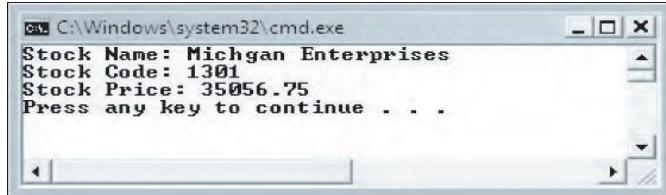


Figure 12.5: Output of Code Snippet 6

When an anonymous type is created, the C# compiler carries out following tasks:

Interprets the type

Generates a new class

Uses the new class to instantiate a new object

Assigns the object with the required parameters

12.4 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 several member definitions. The data members of the class or structure can be split and stored in different files.

These members can be combined into a single unit while executing the program. This can be done by creating partial types.

12.4.1 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 12.6 displays an example of a partial type.

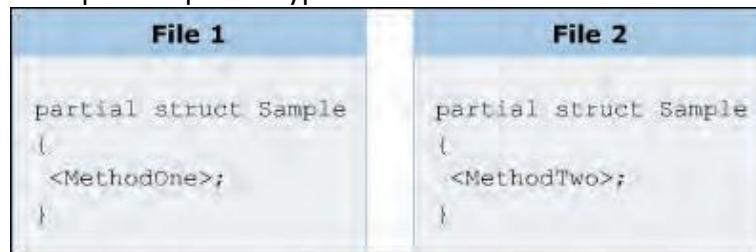


Figure 12.6: Partial Type

Note: C# does not support partial type definitions for enumerations. However, generic types can be partial.

12.4.2 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.

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 7 creates an interface with two partial interface definitions.

Code Snippet 7

```
using System;
//Program Name: MathsDemo.cs
partial interface MathsDemo {
    int Addition(int valOne, int valTwo);
}

//Program Name: 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("Subtraction of two numbers: " +
            objCalculate.Subtraction (numOne, numTwo));
    }
}
```

In Code Snippet 7, 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.

12.4.3 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.

Partial classes can be created for storing 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 8 creates two partial classes that display the name and roll number of a student.

Code Snippet 8

```
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 8, the class **StudentDetails** has its definition spread over two files, **StudentDetails.cs** and **StudentDetails2.cs**. The file **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

12.4.4 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.

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 part of the code has been auto-generated by a tool or IDE and the other parts of the code require customization.

12.5 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.

12.5.1 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.

12.5.2 Characteristics

Nullable types in C# have 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.

12.5.3 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 9 displays the employee's name, ID, and role using the nullable types.

Code Snippet 9

```
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 9, **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

```

12.5.4 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 this nullable type is assigned to a non-nullable type, the compiler generates an exception called `System.InvalidOperationException`.

To avoid this problem, a default value can be specified 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 10 demonstrates the use of **??** operator.

Code Snippet 10

```

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 10, 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
```

12.5.5 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. The conversion of a nullable type to a value type is referred to as explicit conversion.

12.6 Entity Framework

Entity Framework (EF) refers to a collection of technologies in ADO.NET that support the development of data-oriented software applications. It is an open-source Object Relational Mapper (ORM) framework intended for .NET applications. Entity Framework enables developers to work with data by using objects of domain-specific classes. There is no emphasis on the underlying database tables and columns for storing this data. By using the Entity Framework, developers can achieve a high level of abstraction when working with data. Further, they can create data-oriented applications and maintain them using lesser amount of code as compared to traditional applications.

In data-oriented applications, it is required to model entities and relationships as well as the logic of the business problems being solved and work with data engines for storing and retrieving the data. The data can span multiple storage systems, with each having its individual protocols. Further, applications working with a single storage system should be able to balance the storage system requirements against the requirement for writing efficient application code that is easily maintainable. All these requirements are fulfilled in the Entity Framework.

Through the Entity Framework, data can be dealt with as domain-specific objects and properties, for instance, customers and customer addresses. Entity Framework offers the advantage for developers to work at a high level of abstraction when dealing with the data as there is no necessity to focus on the basic database tables and columns storing the data. Further, data-oriented applications can be created using less code when compared to traditional applications, and the code can be easily maintained.

Table 12.1 lists various components of the Entity Framework architecture.

Component	Description
Conceptual Model	Is independent from the database table design. It includes model classes along with their associations.
Storage Model	Represents the design model of a database and comprises views, tables, and stored procedures along with their associations.
Mapping	Has details on how to map the conceptual and storage models.
ADO.NET Data Provider	Interacts with the database.
Object Service	Refers to the primary access point to obtain and return database's data. It takes care of converting the returned data into an entity object structure. This is known as the materialization process.
LINQ-to-Entities (L2E)	Refers to a query language that returns the conceptual model's entities.
Entity SQL	Denotes another query language, which is applicable only to EF 6. It is similar to L2E but is harder to learn than the latter.
Entity Client Data Provider	Converts Entity SQL or L2E queries into SQL, which the database can comprehend. It also interacts with the ADO.NET data provider.

Table 12.1: Components of the Entity Data Model (EDM) Architecture

Following are some features of EF:

-  EF creates an Entity Data Model (EDM) based on Plain Old CLR Object (POCO) entities with get/set properties of various data types. The data model describes the structure of data, as entities and relationships, regardless of how the data will actually be stored. This model is used for querying or saving entity data to the underlying database.
-  EF enables the use of LINQ queries using C# for retrieving data from the underlying database. LINQ queries are translated to the database-specific query language for example, SQL for a relational database, by the database provider. EF also allows executing raw SQL queries directly to the database.
-  Automatic transaction management occurs when querying or saving data. EF also provides options for customizing transaction management.

EF keeps track of changes that have occurred to instances of entities (Property values) that are given to the database.

EF offers first level of caching out of the box. Thus, repeated querying will result in data returning from the cache rather than affecting the database. This can speed up data performance in applications.

12.6.1 EF Core and EF 7

Entity Framework Core (EF Core) has consistently been a favored option among .NET developers for database-related tasks. With the recent launch of .NET 7, EF Core has undergone substantial improvements and the introduction of novel capabilities, further enhancing its efficiency and adaptability for database operations.

Some of the features of Entity Framework 7 are as follows:

- One of the standout features in EF Core 7 is the introduction of two new methods: `ExecuteUpdate()` and `ExecuteDelete()`. These methods offer developers powerful tools for making changes to the database without the necessity to load entities into memory.
- The `ExecuteUpdate()` method offers an efficient means to update database records. In contrast to the conventional `SaveChanges()` method, which monitors entity changes and transmits updates to the database, `ExecuteUpdate()` is explicit and accurate. The developer determines which properties require updating and provides their new values, resulting in improved performance.

Code Snippet 11

```
await db.Posts
    .Where(p => p.Id == 4569)
    .ExecuteUpdateAsync(s => s
        .SetProperty(b => b.AuthorName, "Sarah Jones")
        .SetProperty(b => b.Title, "EF7 can now be used!")
        .SetProperty(b => b.Text, "Some Text...")
        .SetProperty(b => b.LastUpdateDate, "2023-09-06
17:29:46.5028235"));
```

Code Snippet 11 modifies particular properties of a Post entity that shares a matching ID without requiring the entire entity to be loaded into memory.

- The `ExecuteDelete()` method enables the prompt removal of entities from the database, following predefined criteria. Whether the developer is deleting one record or multiple records, `ExecuteDelete()` efficiently executes deletions without the necessity for entity loading. Code Snippets 12 and 13 depict the code for deleting single and multiple records respectively.

Code Snippet 12: Deleting Single Record

```
await db.Posts.Where(p => p.Id == 4569).ExecuteDeleteAsync();
```

Code Snippet 13: Deleting Multiple Record

```
awaitdb.Posts.Where(p =>
p?.Text.Contains("SomeText")).ExecuteDeleteAsync();
```

- The usage of **JSON columns** in relational databases has become increasingly popular due to their versatility in storing and querying semi-structured data. With EF Core 7, there is now built-in support for JSON columns, enabling the developer to link .NET types to JSON documents. This functionality acts as a connection between relational and NoSQL databases, presenting a hybrid approach. Refer to Code Snippet 14.

Code Snippet 14

```
public class Author
{
    public Guid Id { get; set; }
    public string? Name { get; set; }
    public ContactDetails? Contact { get; set; }
}

public class ContactDetails
{
    public Address Address { get; set; } = null!;
    public string? Phone { get; set; }
}

public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
    public string Postcode { get; set; }
}
```

By using the `ToJson()` extension method in the `OnModelCreating()` method, the developer can specify which table columns should receive JSON documents as shown in Code Snippet 15.

Code Snippet 15

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>().OwnsOne(
        author => author.Contact, ownedNavigationBuilder =>
    {
        ownedNavigationBuilder.ToJson();
        ownedNavigationBuilder.OwnsOne(contactDetails =>
            contactDetails.Address);
    });
}
```

- **Faster SaveChanges Method:** Boosting performance is consistently appreciated, and EF Core 7 certainly delivers on this front. The `SaveChanges()` and `SaveChangesAsync()` methods have undergone optimizations, yielding speeds that can be up to four times faster compared to EF Core 6.0. These improvements stem from decreased database round trips and expedited SQL generation.
- **New Query Options:** In EF Core 7, there are fresh query options introduced, which include the capability to employ `GroupBy` as the concluding operation in a query. This enhancement offers greater versatility when dealing with grouped data.

Consider Code Snippet 16 where the developer can utilize the `GroupBy` extension method to group entities.

Code Snippet 16

<code>var groupByAuthor = db.Posts.GroupBy(p => p.AuthorName).ToList();</code>

Though this kind of `GroupBy` operation does not have a direct SQL equivalent, EF Core handles grouping within the retrieved results, offering the developer enhanced command over the queries.

12.7 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, following code is a lambda expression:

`word => word.Length;`

Consider a complete example to illustrate the use of lambda expressions. Assume that a developer wants to calculate the square of an integer number. The developer can use a method

`Square()` and pass the method name as a parameter to the `Console.WriteLine()` method. Code Snippet 17 uses a lambda expression to calculate the square of an integer number.

Code Snippet 17

```
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.

12.7.1 Expression Lambdas

An expression lambda is a lambda with an expression on the right. It has 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 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 there is 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, the developer 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 18 demonstrates expression lambdas.

Code Snippet 18

```
/// <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 18, 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.

12.7.2 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, a developer can specify a return statement to get the result of a lambda. `(string str, string str1) => { return (str==str1); }`

Code Snippet 19 demonstrates a statement lambda expression.

Code Snippet 19

```
/// <summary>
/// Class WordLength determines the length of a given word or phrase
/// </summary>
```

```

/// </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());
        };
        // Invoke the delegate with a string
        len("Mississippi");
    }
}

```

Note: While assigning an expression lambda to a delegate instance, the lambda expression must be compatible with the delegate instance.

12.7.3 Lambdas with Standard Query Operators

Lambda expressions can also be used with standard query operators. Table 12.2 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 12.2: Standard Query Operators

Code Snippet 20 shows how to use the `OrderBy` operator with the lambda operator to sort a list of names.

Code Snippet 20

```

/// <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 20 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 12.7 displays the output of the `OrderBy` operator example.

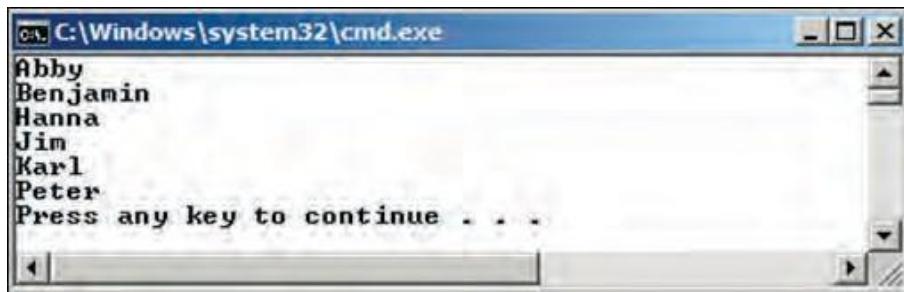


Figure 12.7: OrderBy Operator Example

12.8 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 such as `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 21 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 21

```
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 12.8 displays the output of the query expression example.

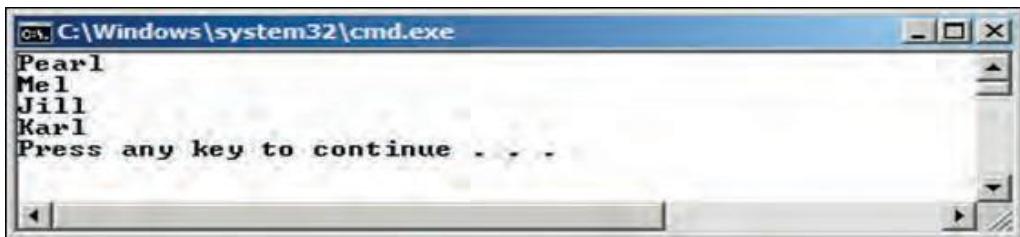


Figure 12.8: Output of the Query Expression Example

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 21 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 12.3.

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 12.3: Query Keywords Used in Query Expressions

12.9 Accessing Databases Using the Entity Framework

Most C# applications 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 requires connecting 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 must 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.

12.9.1 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 12.9 shows the role of EDM in the Entity Framework architecture.

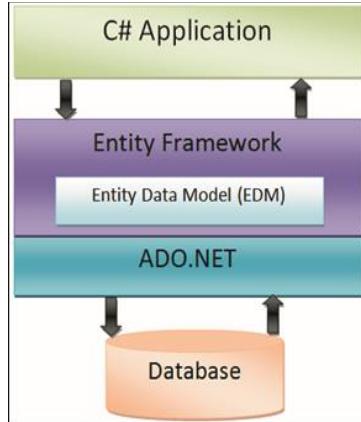


Figure 12.9: Entity Framework Architecture

12.9.2 Development Approaches

Entity Framework eliminates the necessity to write most of the data-access code that otherwise require to be written. It uses different approaches to manage data related to an application.

These approaches are as follows:

The Database-first approach	The model-first approach	The code-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 Entity Framework creates database objects based on the model that a programmer creates to represent the entities and their relationships in the application.	The Entity Framework creates database objects based on custom classes that a programmer creates to represent the entities and their relationships in the application.

12.9.3 Creating an Entity Data Model

Visual Studio 2022 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, programmers can use Visual Studio 2022 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 must perform following steps:

1. Open Visual Studio 2022.

2. Create a **Console App (.NET Framework)** project named **EDMDemo**. Ensure to select the .NET Framework option with console app otherwise the ADO.NET assemblies will not be added.
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 12.10.

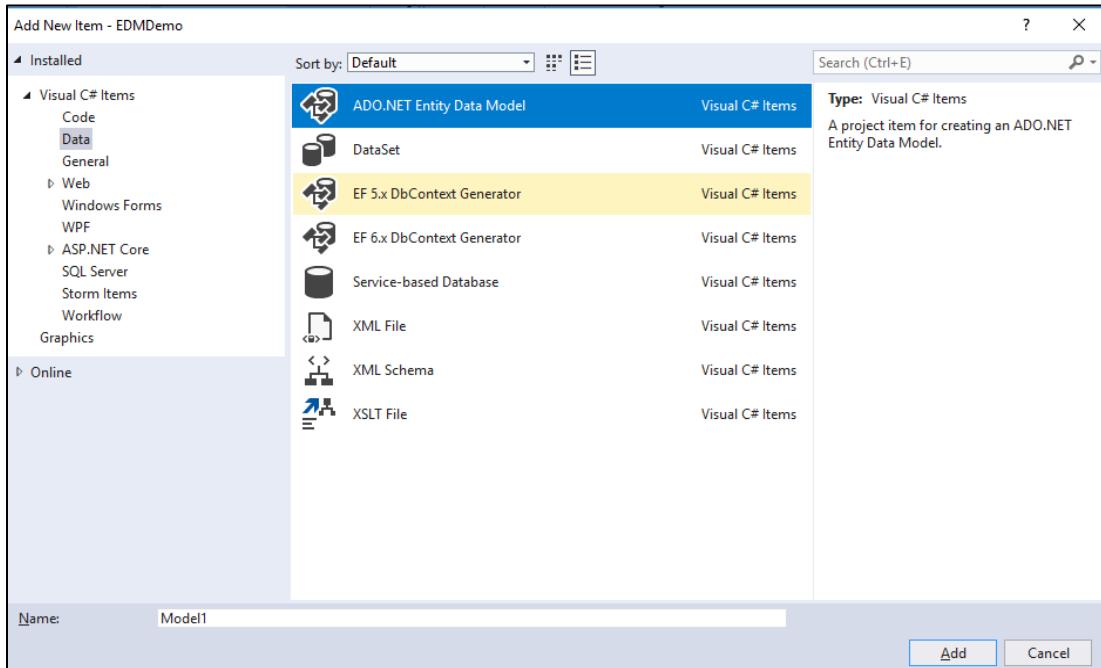


Figure 12.10: Add New Item – EDMDemo Dialog Box

1. Click **Add**. The **Entity Data Model Wizard** appears.
2. Select **Empty EF Designer model**.
3. Click **Finish**. The Entity Data Model Designer is displayed.
4. Right-click the Entity Data Model Designer and select **Add New→Entity**. The **Add Entity** dialog box is displayed.
5. Enter **Customer** in the **Entity name** field and **CustomerId** in the **Property** name field, as shown in Figure 12.11.

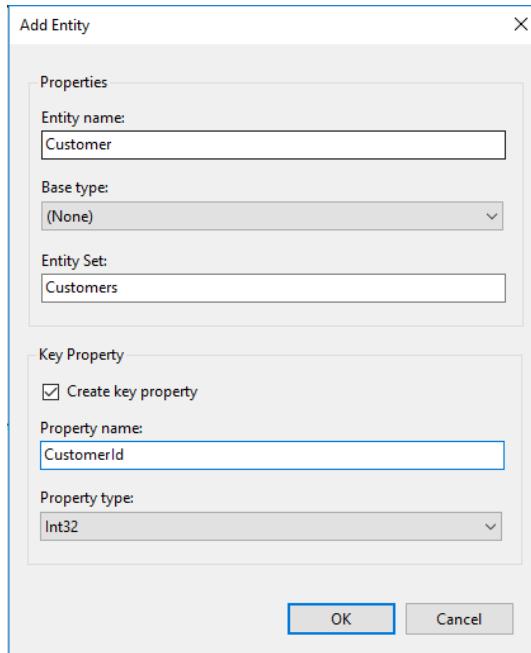


Figure 12.11: Add Entity Dialog Box

6. Click **OK**. The Entity Data Model Designer displays the new **Customer** entity, as shown in Figure 12.12.

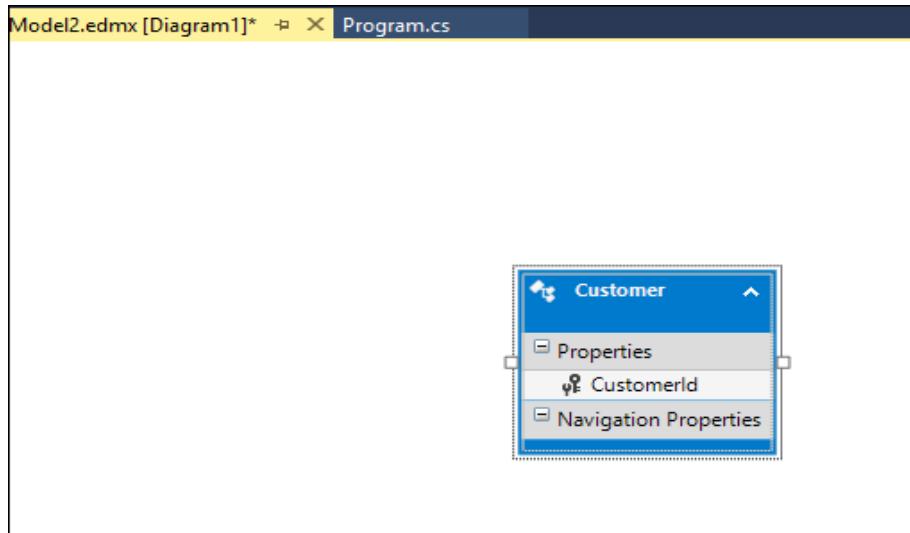


Figure 12.12: Customer Entity

7. Right-click the **Customer** entity and select **Add New→Scalar property**.
8. Enter **Name** as the name of the property.
9. Similarly, add an **Address** property to the **Customer** entity.
10. Add another entity named **Order** with an **OrderId** key property.
11. Add a **Cost** property to the **Order** entity.

12.9.4 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 points 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 12.13.

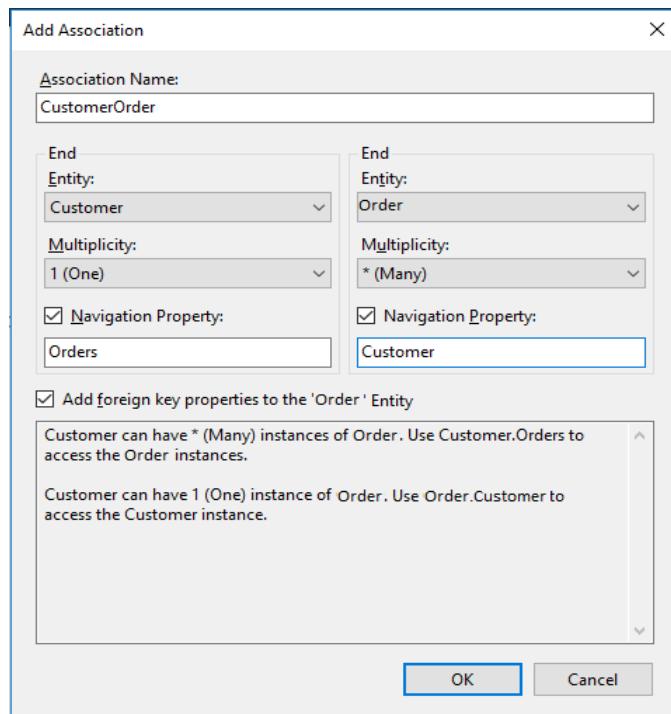


Figure 12.13: Entity Relationship

Click **OK**. The Entity Data Model Designer displays the entities with the defined relationship, as shown in Figure 12.14.

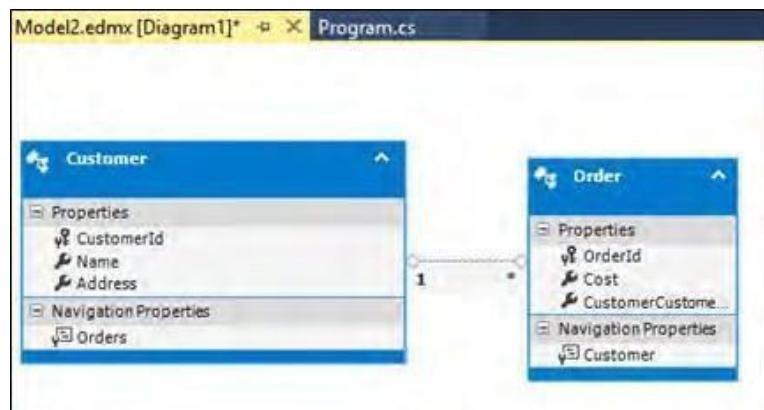


Figure 12.14: Relationship Between the Customer and Order Entities

12.9.5 Creating Database Objects

After designing the model of the application, the programmer must generate database objects based on the model. To generate 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.

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 12.15.

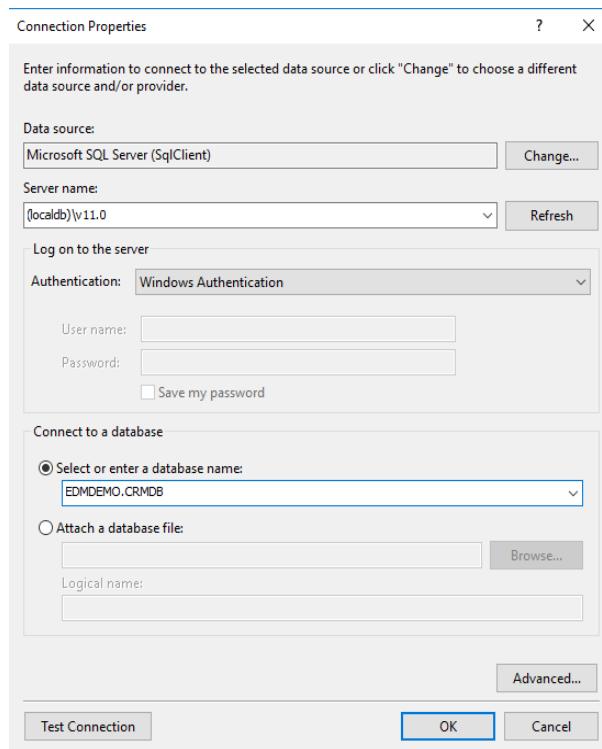


Figure 12.15: Connection Properties Dialog Box

Click **OK**. Visual Studio will prompt whether to create a new database.

3. Click **Yes**.
4. Click **Next** in the **Generate Database Wizard** window. Visual Studio generates the scripts to create database objects.
5. Click **Finish**. Visual Studio opens the file containing the scripts to create the database objects.
6. Right-click the file and select **Execute**. The **Connect to Server** dialog box is displayed.
7. Click **Connect**. Visual Studio creates database objects.

12.9.6 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 22 shows the `Main()` method that creates and persists **Customer** and **Order** entities.

Code Snippet 22

```
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.Write("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 22 prompts and accepts 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.
```

12.9.7 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 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 requires 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 23 creates and executes a query to retrieve the records of all `Customer` entities along with the associated `Order` entities.

Code Snippet 23

```
class Program {
    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 23, 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 require 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 24 shows a LINQ query that retrieves only the customer names of the **Customer** entity instances.

Code Snippet 24

```
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 24, 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 25 uses the `where` clause to filter customer records.

Code Snippet 25

```
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 25 uses the `where` clause to retrieve information of the customer with the name Alex Parker. The `foreach` statement iterates 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
```

12.9.8 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 26 uses the `Select` method to project the `Name` and `Address` properties of `Customer` entity instances into a sequence of anonymous types.

Code Snippet 26

```
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, a developer can use the other operators such as `Where`, `GroupBy`, `Max`, and so on through method-based queries.

12.10 Multithreading and Asynchronous Programming

C# applications often have to execute multiple tasks concurrently. For example, a C# application that simulates a car racing game must 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 various classes and interfaces in the `System.Threading` namespace that provides built-in support for multithreaded programming in the .NET Framework.

12.10.1 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 27 instantiates and starts a new thread.

Code Snippet 27

```
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 27, 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 execute both the infinite `while` loops, which results in the output shown in Figure 12.16.

Figure 12.16: Output of Code Snippet 27

12.10.2 Synchronizing Threads

When multiple threads must share data, their activities must 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 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 must 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.

12.10.3 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 must 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.

12.10.4 Task Class

TPL provides the `Task` class in the `System.Threading.Tasks` namespace 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 is completed 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 complete, 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 28 creates and starts two tasks.

Code Snippet 28

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") ;
    Console.ReadKey () ;
}
}

```

In Code Snippet 28, 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. The `ReadKey()` is given to make the display wait before exiting.

Output:

Executed by a Task
 Executed by a Task
 Exiting main method

12.10.5 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.

12.10.6 Task Continuation

When multiple tasks are executed 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.

12.10.7 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 must 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.

12.10.8 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 `foreach` 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 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 29 uses a traditional `for` loop and the `Parallel.For()` method.

Code Snippet 29

```
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 29, 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 12.17, 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 12.17, multiple threads indicated by the `Thread.CurrentThread.ManagedThreadId` property executes the `for` loop in parallel and the sequence of iteration is unordered.

Figure 12.17 shows one of the possible outputs of Code Snippet 29.

```
cmd C:\Windows\system32\cmd.exe  
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 12.17: Output of Code Snippet 29

12.10.9 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 30 shows using both a sequential LINQ to Object and PLINQ to query an array.

Code Snippet 30

```
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 30 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

12.10.10 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 following return types:

- `void`
- `Task`
- `Task<TResult>`

Note: By convention, a method marked with the `async` keyword ends with an `Async` suffix.

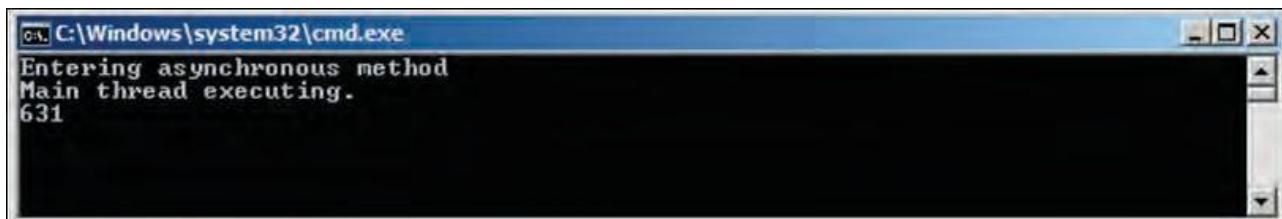
Code Snippet 31 shows the use of the `async` and `await` keywords.

Code Snippet 31

```
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;  
    }  
    public int GetResult() {  
        /*Pause Thread to simulate time consuming operation*/  
        Thread.Sleep(2000);  
        return new Random().Next(1, 1000);  
    }  
}
```

In Code Snippet 31, 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 12.18 shows the output.

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

```
Entering asynchronous method
Main thread executing.
631
```

The window has a standard Windows title bar and scroll bars on the right side.

Figure 12.18: Output of Code Snippet 31

12.11 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 runtime 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 runtime exception.

To understand how dynamic types bypasses compile type checking, consider Code Snippet 32.

Code Snippet 32

```
using System;
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 32, 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 upon encountering the `dynamic` keyword does not perform any type checking. However, on executing the program, a runtime exception will be thrown, as shown in Figure 12.19.

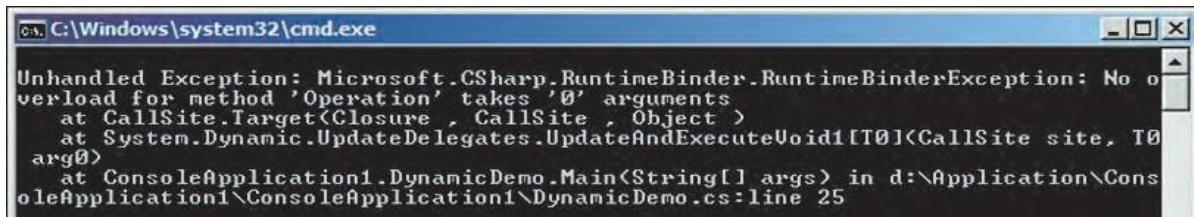


Figure 12.19: Exception Generated at Runtime

The `dynamic` keyword can also be applied to fields, properties, method parameters, and return types. Code Snippet 33 shows how the `dynamic` keyword can be applied to methods to make them reusable in a program.

Code Snippet 33

```
class DynamicDemo {
    static dynamic DynaMethod(dynamic param) {
        if (param is int) {
            Console.WriteLine("Dynamic parameter of type int has value
{0}", param);
            return param;
        }
        else if (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}")
    }
}
```

```
    \n{2}", dynaVar1, dynaVar2, dynaVar3);  
}  
}
```

In Code Snippet 33, 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 Hello World  
Dynamic parameter of unknown type has value 12.5
```

```
Returned dynamic values:  
3  
Hello World  
12.5
```

12.12 Summary

- Anonymous methods allow the developer to pass a block of unnamed code as a parameter to a delegate.
- Extension methods allow the developer to extend different types with additional static methods.
- The developer 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 the developer to split the definitions of classes, structs, and interfaces to store them in different C# files.
- The developer can define partial types using the partial keyword.
- Nullable types allow the developer to assign null values to the value types.
- Nullable types provide two public read-only properties namely, HasValue and Value.
- ADO.NET is a database technology of .NET Framework that is used to connect an application system and a database server.
- Entity Framework refers to an open-source ORM framework intended for .NET applications supported by Microsoft.
- There are many features of Entity Framework 7 that have not been added in Entity Framework Core.
- EF Core 7.0 can be used with .NET Framework applications.
- For new application development, it is recommended to use EF Core on .NET 7.0.

12.13 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, a developer is 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 in achieving this?

(A)	IQueryable<Student> query = from c in dbContext.Students orderby c.FirstName select c;
(B)	IQueryable< Student> query = select c from dbContext.Students orderby FirstName;
(C)	IQueryable<Student> query = from c in dbContext.Students select c orderby c.FirstName;
(D)	IQueryable<Students> query = from s in dbContext.Student orderby c.FirstName select c;

(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 task 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

12.13.1 Answers

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

Try It Yourself

1. Develop a C# application that interacts with a SQL Server database using ADO.NET. The application requires to perform various database operations, including inserting, updating, and retrieving data from the database. Your task is to create a class that handles these operations.

Create a `DatabaseHandler` class that encapsulates database operations. This class should include methods for the following:

`InsertData`: This method inserts a new record into the database.

`UpdateData`: This method updates an existing record in the database.

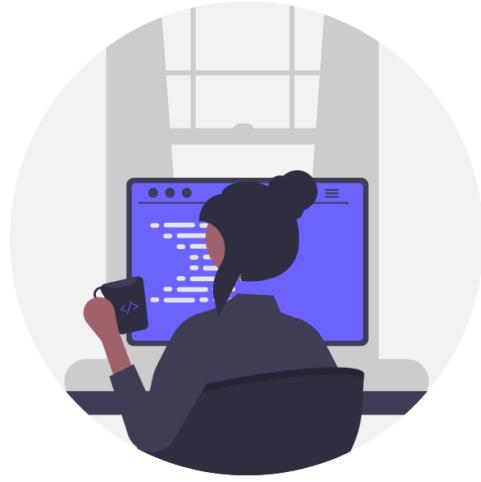
`GetData`: This method retrieves data from the database based on a provided query.

The `DatabaseHandler` class should also include a constructor that accepts a connection string as a parameter to establish a connection to the database.

In your `Main()` method, create an instance of the `DatabaseHandler` class and demonstrate how to use its methods to insert, update, and retrieve data from the database.

Implement proper exception handling for potential database-related errors, such as connection issues or query failures.

2. Given a list of '`n`' positive integers that represent coin values and a target value '`amount`', your task is to develop a C# program that computes the minimum number of coins required to reach the specified target value. In situations where it is not feasible to attain the target amount using the provided coins, your program should return -1.



Session 13

Building Cross-Platform Mobile Apps Using .NET MAUI

Welcome to the Session, **Building Cross-Platform Mobile Apps Using .NET MAUI**.

This session provides insights into .NET MAUI architecture. It describes how .NET MAUI works and explores its features. It also explains the IDEs used for developing MAUI applications. Thereafter, the session explains how to install .NET MAUI. Finally, the session concludes with a brief overview of building an application with .NET MAUI.

In this Session, you will learn to:

- Describe the .NET MAUI architecture
- List the features of MAUI
- Explain how to install and run the application

13.1 Introduction to .NET MAUI Architecture

.NET MAUI or .NET Multi-platform App UI, stands as a highly adaptable cross-platform framework engineered for the development of native mobile and desktop applications employing C# and XAML. This framework empowers developers to create applications that seamlessly operate across various platforms, such as Android, iOS, macOS, and Windows, all through a unified codebase. As an open-source framework, .NET MAUI signifies a progression beyond Xamarin.Forms, broadening its applicability from mobile to include desktop scenarios. It introduces a revamped array of UI controls designed for peak performance and enhanced extensibility.

However, notable distinctions also exist. With .NET MAUI, developers have the capability to design multi-platform applications within a single project, while maintaining the flexibility to integrate platform-specific source code and resources when required.

A central goal of .NET MAUI is to enable developers to consolidate the majority of their application's logic and UI into a single shared codebase, streamlining the development process.

13.1.1 How Does .NET MAUI Work?

.NET MAUI streamlines the development workflow by merging the APIs of Android, iOS, macOS, and Windows into a cohesive, unified API, creating a 'write once, run anywhere' development paradigm. This framework provides comprehensive access to all the intricacies of each native platform.

Starting with .NET 6.0 and onward, a family of platform-specific frameworks has been introduced for app development, including .NET Android, .NET iOS, .NET macOS, and the Windows UI 3 (WinUI 3) library. These frameworks collectively share access to the same .NET Base Class Library (BCL), which shields the code from the complexities of the underlying platform. BCL relies on the .NET runtime to provide execution environment for the code. For Android, iOS, and macOS, Mono, a .NET runtime implementation, oversees the execution environment, while on Windows, it is the .NET CoreCLR that manages execution.

While BCL eases sharing of business logic across apps on diverse platforms, each platform offers unique approaches for defining UIs and presents different models for governing how UI elements interact. To create the UI for each platform, the developer can utilize the corresponding platform-specific framework (.NET Android, .NET iOS, .NET macOS, or WinUI 3). However, this approach requires the upkeep of separate codebases for each device family.

On the other hand, .NET MAUI presents a consolidated framework for crafting UIs for both mobile and desktop apps, simplifying the development workflow. Figure 13.1 offers a broad view of the architecture of a .NET MAUI application.

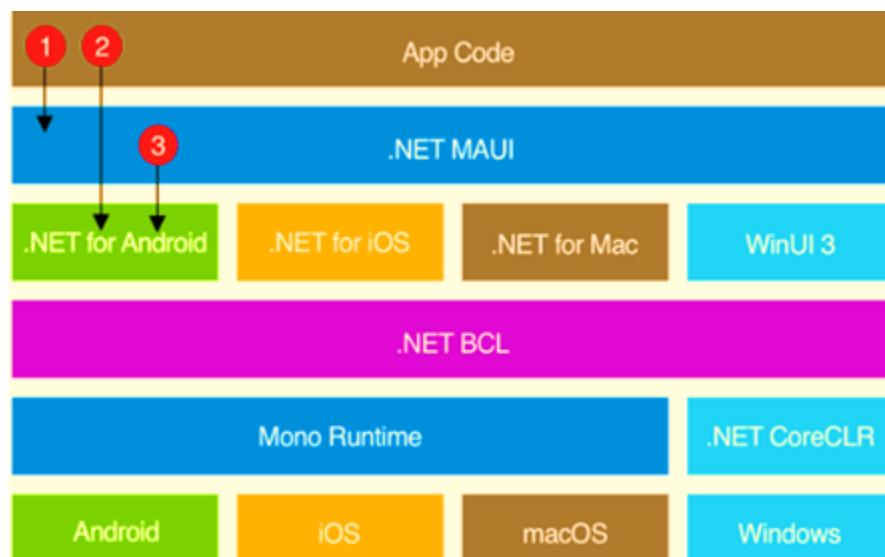


Figure 13.1: .NET MAUI Architecture

Explanation for highlighted items in Figure 13.1 is as follows:

- 1) In a .NET MAUI application, the code primarily interfaces with the .NET MAUI API.
- 2) In cases where required, the app code can also directly engage with platform APIs.
- 3) The .NET MAUI framework subsequently interfaces directly with the native platform APIs.

.NET MAUI applications can be crafted on both desktop PCs and Mac and can compile into native app packages in following manner:

- Android apps produced using .NET MAUI undergo compilation from C# into an Intermediate Language (IL). Subsequently, during app launch, this IL code undergoes Just-In-Time (JIT) compilation into native assembly code.
- iOS applications crafted using .NET MAUI undergo complete Ahead-Of-Time (AOT) compilation, where they are converted from C# directly into native ARM assembly code.
- macOS applications built with .NET MAUI make use of Mac Catalyst, an Apple solution that extends the iOS app, initially designed with UIKit, to the desktop environment. Additional integration of AppKit and platform APIs is performed as required.
- Windows applications crafted using .NET MAUI utilize the WinUI 3 library to generate native applications tailored for the Windows desktop environment.

13.1.2 Features of MAUI

.NET MAUI provides cross-platform APIs for native device features. Here are some important features provided by MAUI:

- Access to sensors, such as the accelerometer, compass, and gyroscope on devices
- Ability to check the device's network connectivity state and detect changes
- Information about the device the app is running on
- Ability to copy and paste text to the system clipboard, between apps
- Ability to pick single or multiple files from the device
- Storage of data securely as key/value pairs
- Utilization of built-in text-to-speech engines to read text from the device
- Browser-based authentication flows that listen for a callback to a specific app registered Uniform Resource Locator (URL)
- Support for .NET hot reload

13.1.3 IDEs for Developing MAUI Applications

With the availability of Visual Studio 2022 and Visual Studio Code IDEs, the developer can build native applications for Android, iOS, macOS, and Windows using MAUI.

13.2 Getting Started with MAUI

Following are pre-requisites for installing MAUI:

- Framework – .NET latest version with .NET MAUI
- Platform SDKs – Android and iOS
- IDE – Visual Studio 2022 or Visual Studio Code

Figure 13.2 depicts the workloads required for MAUI using Visual Studio 2022. This can be selected during Visual Studio 2022 installation.

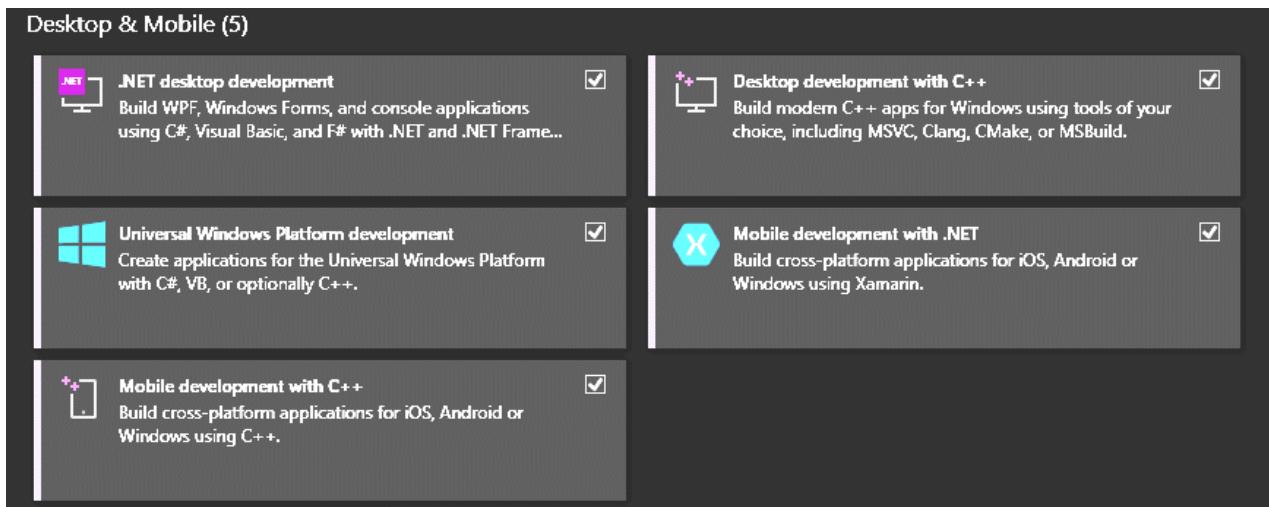


Figure 13.2: Workloads Required for MAUI Using Visual Studio 2022

Alternatively, one can do it via command prompt.

Steps to install MAUI using the Visual Studio Developer Command Prompt are given as follows (type each of the given commands at the prompt):

➤ **Install MAUI workloads using .NET CLI**

```
dotnet workload install maui
```

➤ **Install Missing Components**

```
dotnet tool install -g redth.net.maui.check
```

➤ **Check If Anything Still Missing**

```
maui-check
```

If any tools and SDKs required by .NET MAUI are still missing, `maui-check` will install them.

A new dotnet workload install command is available for installing the mobile workloads. Developers can also install MAUI Workloads using an alternative approach.

At the Administrator Command Prompt, they can type

```
dotnet workload install maui
```

Then, install the given workloads one by one:

```
dotnet workload install maui-android
dotnet workload install maui-ios
dotnet workload install maui-maccatalyst
dotnet workload install maui-windows
dotnet workload install microsoft-android-sdk-full
dotnet workload install microsoft-ios-sdk-full
dotnet workload install microsoft-maccatalyst-sdk-full
dotnet workload install microsoft-maccatalyst-sdk-full
dotnet workload install microsoft-macos-sdk-full
dotnet workload install microsoft-tvos-sdk-full
```

13.3 Building Sample Hello World MAUI Application

Consider an example to build a MAUI application utilizing Visual Studio 2022. Initially, generate a new application by executing following command in the Developer Command Prompt:

```
dotnet new maui -n HelloMaui
```

This will autogenerate a complete application named HelloMaui. Now, launch Visual Studio 2022 and open the created project using **Open a project or solution** and browse for the project. Figure 13.3 depicts this.

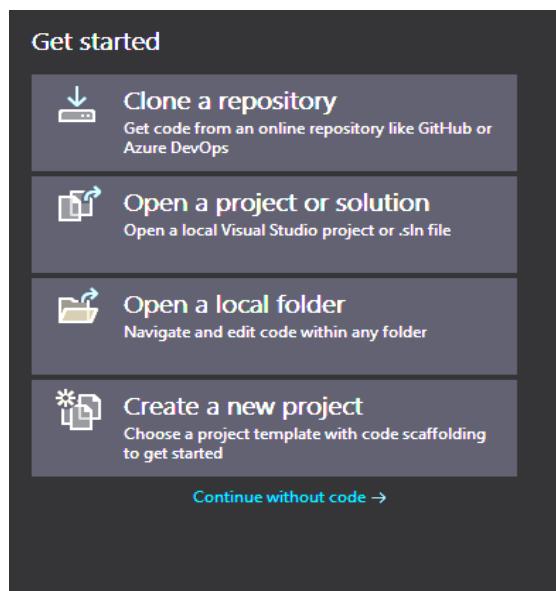


Figure 13.3: Opening Newly Created MAUI Project

Wait for a while to load the dependencies. Retain the autogenerated files as is.

Figure 13.4 shows the project structure of the MAUI application.

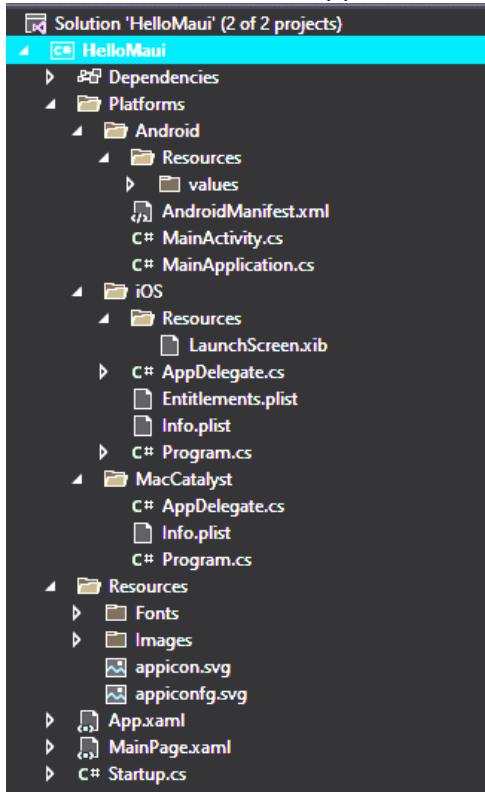


Figure 13.4: Project Structure of MAUI Application

13.3.1 Running the MAUI Application

To execute the MAUI Application, either an emulator or a local Android device connected to the system is required for mobile application testing. If the application is intended to be run on a local Android device, it is necessary to activate **Developer Settings** on the device.

In the Visual Studio 2022 toolbar, click the **Start** button to launch the app in the chosen Android emulator or Android Local Device. Figure 13.5 depicts this.

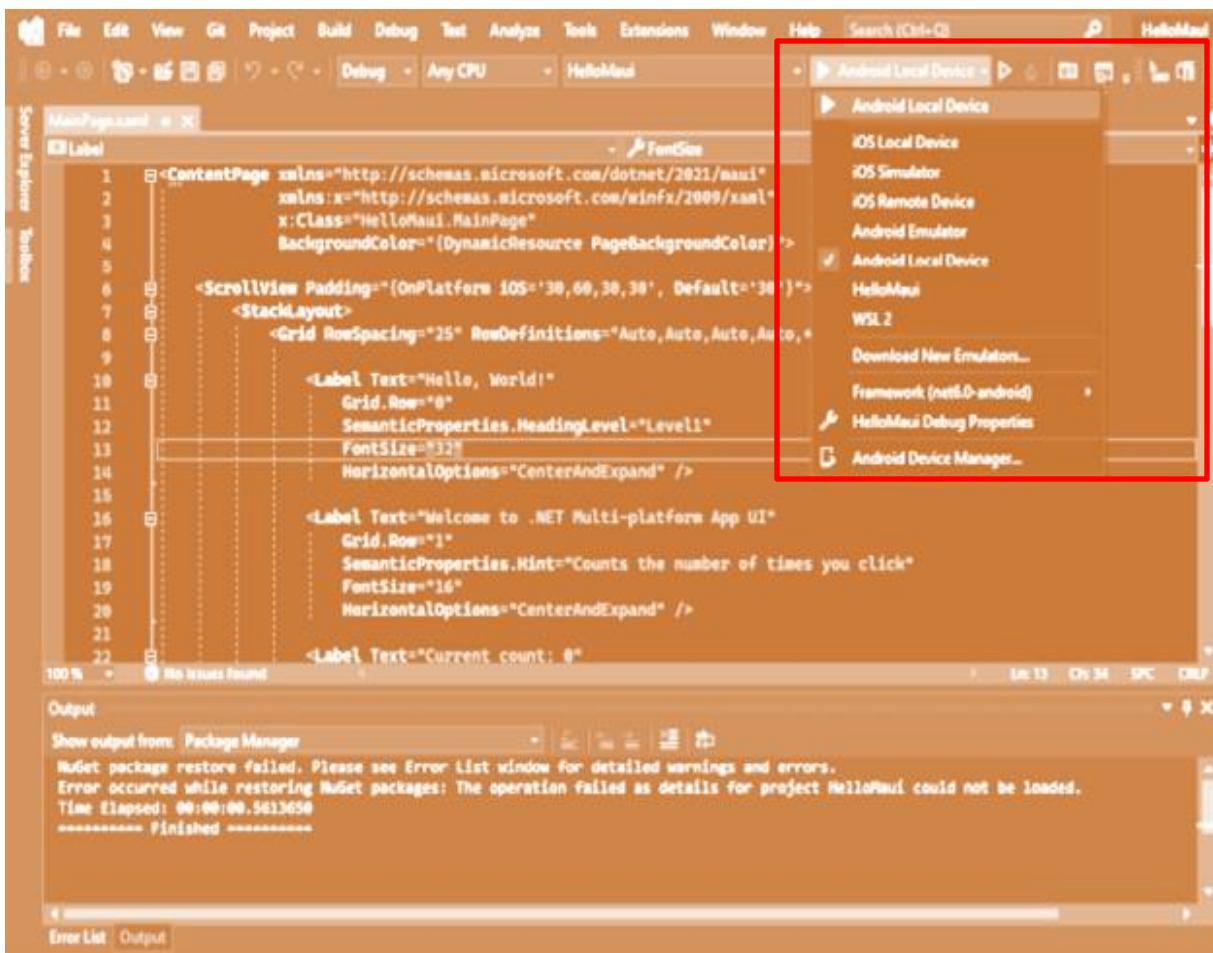


Figure 13.5: Running MAUI Application Using Start

In the running app, click the **CLICK ME** button several times and observe that the count of the number of button clicks is incremented. Figure 13.6 depicts a sample count.

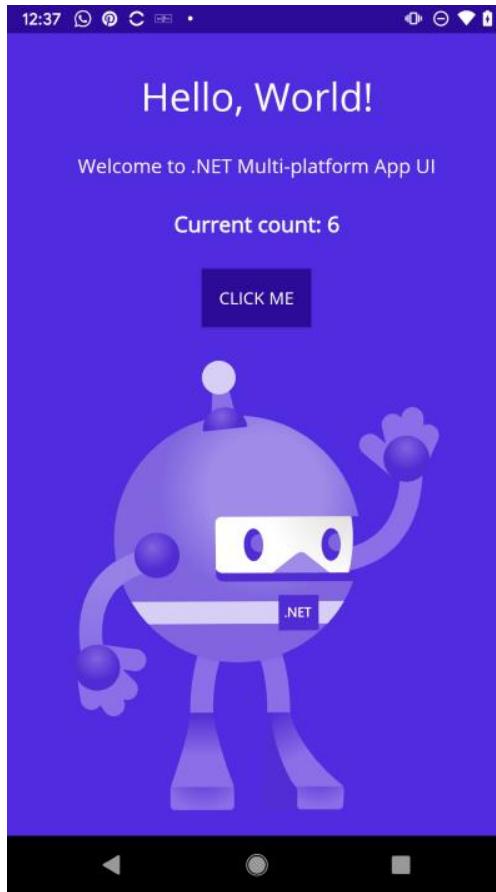


Figure 13.6: Output of MAUI Application

13.4 Creating, Loading, and Saving a Page for a Note App

Consider that the developer has been given a requirement to build a Note App which allows end users to create and keep notes on their device.

Building a complete .NET MAUI application for creating, loading, and saving notes requires multiple files and components. A simplified example is explained here demonstrating the core functionalities: creating a note, loading it, and saving it. The developer can expand on this to build a more robust application later with additional features.

Basic steps are as follows:

- **Create a new .NET MAUI project:** Start by creating a new .NET MAUI project in Visual Studio 2022 or the preferred IDE. Refer to Figure 13.7.

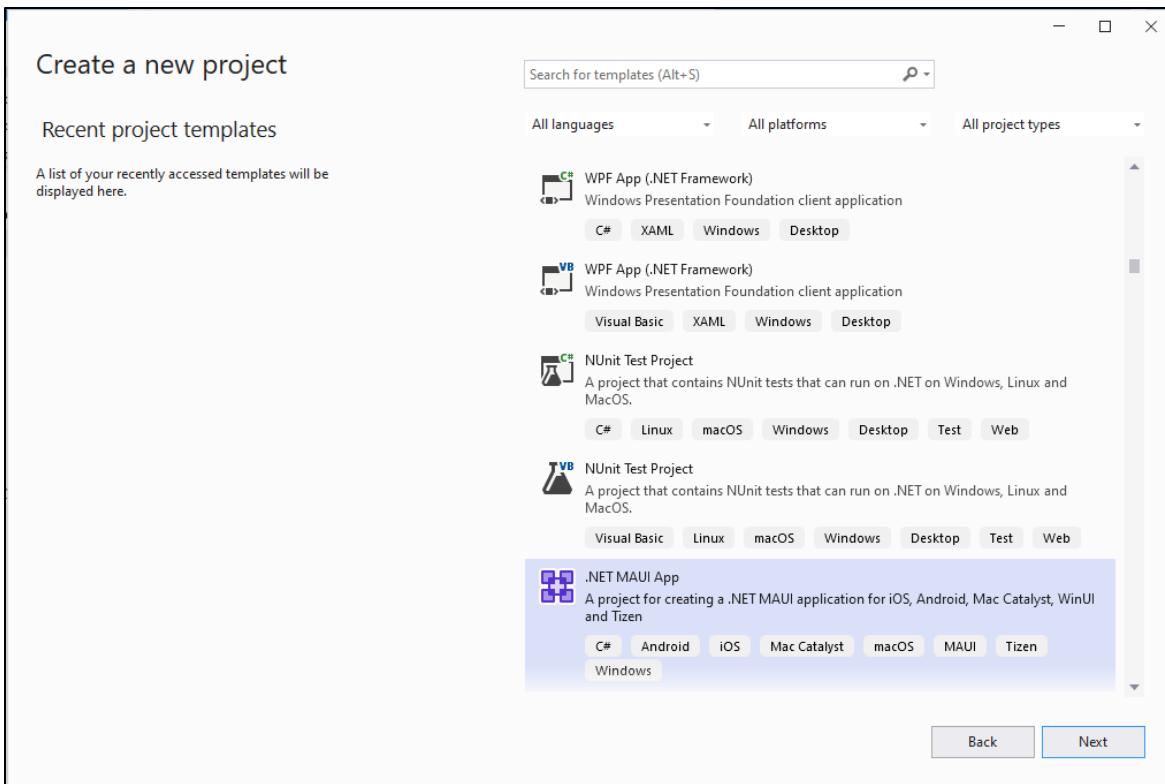


Figure 13.7: .NET MAUI App Template in VS 2022

- **Edit MainPage.xaml:** Define the UI for the application's main page. Code Snippet 1 depicts a simple XAML code.

Code Snippet 1

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="NoteApp.MainPage">
    <Stack>
        <Entry x:Name="NoteEntry" Placeholder="Enter the note..." />
        <Button Text="Save Note" Clicked="SaveNote_Clicked" />
        <Button Text="Load Note" Clicked="LoadNote_Clicked" />
        <Label x:Name="NoteLabel" Text="" />
    </Stack>
</ContentPage>
```

- **Edit MainPage.xaml.cs:** Implement the code-behind for MainPage.xaml to handle saving and loading notes. Refer to Code Snippet 2.

Code Snippet 2

```
using Microsoft.Maui.Controls;
using System;
using System.IO;
using System.Threading.Tasks;
namespace NoteApp{
    public partial class MainPage : ContentPage {
        private string noteFilePath =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder
.LocalApplicationData), "note.txt");
        public MainPage() {
            InitializeComponent();
        }

        private async void SaveNote_Clicked(object sender, EventArgs e) {
            string noteText = NoteEntry.Text;
            await File.WriteAllTextAsync(noteFilePath, noteText);
            NoteEntry.Text = ""; // Clear the entry field
            await DisplayAlert("Note Saved", "the note has been saved.",
                "OK");
        }

        private async void LoadNote_Clicked(object sender, EventArgs e) {
            if (File.Exists(noteFilePath)) {
                string noteText = await File.ReadAllTextAsync(noteFilePath);
                NoteLabel.Text = noteText;
            }
            else {
                await DisplayAlert("No Note Found", "There's no saved note.",
                    "OK");
            }
        }
    }
}
```

- **Add Required Permissions:** In the Android and iOS projects, make sure to request appropriate permissions to read and write to local storage in the MainActivity.cs (Android) and AppDelegate.cs (iOS) files.
- **Build and Run:** Build and run the .NET MAUI application on an emulator or physical device.

With this, the developer has a basic .NET MAUI application ready that allows users to create, save, and load notes. Users can enter text in the 'Enter the note...' field, save it, and later load it to view.

13.5 Summary

- .NET MAUI allows developers to create native applications for multiple platforms, including Android, iOS, macOS, and Windows using a single codebase.
- .NET MAUI provides a unified API surface that simplifies cross-platform development by offering a single set of APIs for building UIs, accessing device features, and more.
- .NET MAUI leverages familiar technologies such as C# and XAML, making it accessible to a broad range of developers with existing .NET experience.
- .NET MAUI supports adaptive UIs that automatically adjust to different screen sizes and orientations, enhancing the user experience across devices.
- .NET MAUI includes modern and customizable UI controls, making it easier to create visually appealing and interactive applications.
- The framework is designed for high performance, with AOT compilation for iOS and JIT compilation for Android, ensuring optimal app responsiveness.
- .NET MAUI is open-source, allowing the community to contribute, extend, and improve the framework.
- It integrates seamlessly with popular development tools such as Visual Studio, Visual Studio Code, and Xamarin.Forms, making it easy for developers to get started.

13.6 Check Your Progress

1. What does .NET MAUI stand for?

(A)	.NET Mobile App Integration
(B)	.NET Multi-platform App UI
(C)	.NET Mobile Application Interface
(D)	.NET Mobile App User Interface

2. Which platforms can the developer target with .NET MAUI?

(A)	Android and iOS only
(B)	Android, iOS, macOS, and Windows
(C)	Android and Windows only
(D)	iOS and macOS only

3. What is the purpose of .NET MAUI's adaptive UI capabilities?

(A)	To create a fixed user interface for all devices
(B)	To provide a consistent user interface only for iOS devices
(C)	To adjust the user interface to different screen sizes and orientations
(D)	To improve app performance

4. Which of the following development tools can be used for .NET MAUI development?

(A)	Visual Studio only
(B)	Visual Studio Code only
(C)	Both Visual Studio and Visual Studio Code
(D)	None of these

5. What does .NET MAUI offer for platform-specific customization?

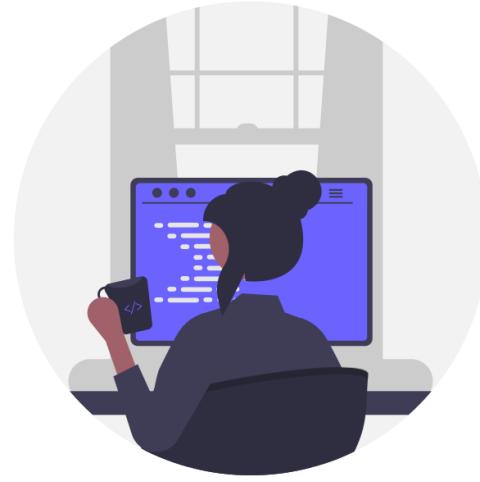
(A)	Access to platform-specific features when required
(B)	Automatic platform-specific adaptation
(C)	No option for platform-specific customization
(D)	Complete isolation from platform-specific features

13.6.1 *Answers*

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

Try It Yourself

1. Create a basic To-Do List mobile application using .NET MAUI and C#. The application should allow users to add, edit, and remove tasks from their to-do list.
2. Build a cross-platform mobile app using .NET MAUI and C# that provides weather forecasts for various locations. The app should allow users to select a city or location, view the current weather conditions, and access five days forecast.



Session 14

.NET Development and the Future

Welcome to the Session, **.NET Development and the Future**.

This session provides insights into .NET Core and .NET Framework. It lists the differences between the two platforms. The session introduces .NET 7.0 and describes its features. Thereafter, the session explains how to create an application with .NET 7.0 and VS 2022. Finally, the session concludes with a brief overview of .NET 8.0.

In this Session, you will learn to:

- Explain what .NET Core is
- Explain the history of .NET Core and .NET
- Identify differences between .NET Framework and .NET Core
- Explain .NET 7.0 and its key features
- Describe how to create an application with .NET 7.0 and VS 2022
- Outline .NET 8.0 and the Future

14.1 Introduction to .NET Core

.NET Core is an open-source .NET Framework platform available for Windows, Linux, and Mac OS. As it is a lightweight version of the .NET Framework, not all the libraries of the original framework can be accessed.

Following are the key features of .NET Core:

➤ Cross-platform and Container Support

Most developers prefer to write their business logic code once and reuse it later. This is an easier approach as compared to building different apps to target multiple platforms. Therefore, developers can create an app that will not only run on Windows, but also run on Linux, macOS, and on different architectures, such as x86 and Advanced RISC Machine (ARM). This is ideal in

many instances, including desktop applications.

➤ **High Performance**

.NET Core has relatively higher performance due to the lightweight assemblies and some equally exciting improvements throughout the runtime and the base class libraries.

➤ **Asynchronous via Async/Await**

C# offers a streamlined approach known as async programming capable of leveraging asynchronous support in .NET Framework 4.5 onwards, .NET Core, and Windows Runtime. In this approach, the compiler performs complex tasks, which otherwise would have been done by the developers. Further, a logical structure is maintained by the application that is similar to synchronous code. Thus, users are presented with all benefits of asynchronous programming with little effort.

➤ **Unified Model View Controller (MVC) and Web API Frameworks**

MVC controllers and Web API controllers are now exactly the same. They have now been unified from MVC 6.0 onwards. .NET Core offers this feature for efficient programming.

➤ **Multiple Environments and Development Mode**

In IT environments, there are many development environments not supported by the three default environments - Development, Staging, and Production - offered by Microsoft in the ASP.NET Core libraries.

.NET Core offers other environments ahead of these three, such as Test, DevelopmentExternal, TestExternal, and ProductionExternal.

➤ **Dependency Injection (DI)**

DI is an approach that helps in obtaining loose coupling among objects and their dependencies. Instead of instantiating the dependencies directly or with static references, the objects required by a class to accomplish a task are given to the class in a certain fashion. In simpler words, Dependency Injection is used to make applications independent of its objects or to make classes independent of how its objects are created. Classes declare their dependencies with the help of the constructor.

➤ **Support for Microservices**

.NET Core supports microservices, which is a type of service-oriented architecture. Microservices are software applications containing small and modular business services. Each service is capable of executing a unique process, of being deployed independently, and being developed in various programming applications. .NET Core supports a mix of technologies that can be minimized for each microservice. It can be scaled up as and when new microservices are included.

After .NET Core 3.1, Microsoft decided to rebrand the framework without the suffix Core.

Thus, .NET 5.0 was launched and subsequently, further versions such as .NET 6.0 onwards were released.

14.1.1 .NET Core and .NET Version History

Table 14.1 summarizes the history of .NET Core releases and how .NET has emerged in recent years.

Version	Release Date
.NET Core 1.0	June 27, 2016
.NET Core 1.1	November 16, 2016
.NET Core 2.0	August 14, 2017
.NET Core 2.1	May 30, 2018
.NET Core 2.2	December 4, 2018
.NET Core 3.0	September 23, 2019
.NET Core 3.1	December 3, 2019
.NET 5.0	November 10, 2020
.NET 6.0	November 8, 2021
.NET 7.0	November 8, 2022
.NET 8.0	Preview available, Stable version yet to be released

Table 14.1: Version History of .NET Core and .NET

14.2 Differences Between .NET Framework and .NET Core

Table 14.2 lists the major differences between .NET Core and .NET Framework.

.NET Framework	.NET Core
It was made available as a licensed and proprietary software framework. Later, Microsoft made some of its components open source.	Microsoft released .NET Core as an open-source software framework.
It allows users to create applications for a single platform - Windows.	.NET Core is cross-platform and can accommodate three different OSes - Windows, OS X, and Linux.
It must be set up as a single package and runtime environment for Windows.	As it is cross-platform, it can be packaged and set up irrespective of the underlying OS.
It uses CLR for managing libraries and compilation purposes.	It utilizes a redesigned CLR known as CoreCLR and presents a modular collection of libraries known as CoreFX.
If utilizing Web applications with .NET Framework, users can employ a robust Web application framework, such as ASP.NET.	It has a redesigned version of ASP.NET.

.NET Framework	.NET Core
With .NET Framework, users must deploy Web applications only on Internet Information Server.	The Web applications created with ASP.NET Core can be run in several ways. These applications can be deployed directly in the Cloud or users can self-host them by creating specific hosting processes.
It has extensive APIs for creating cloud-based applications.	It has features that ease the creation and deployment of Cloud-based application.
It does not have any robust framework or tools to simplify mobile app development.	It is compatible with Xamarin via the .NET Standard Library. Hence, users can exploit Xamarin to script cross-platform mobile apps in C# using a shared code base and same set of APIs.
It provides few options for developing microservice oriented systems as compared to .NET Core.	It allows the creation of microservice oriented systems rapidly.
It requires additional infrastructure to be installed for achieving scalability.	It is more effective when compared to .NET Framework in terms of enhancing the performance and scalability of applications without having to deploy extra hardware or infrastructure.
It can use all the features in .NET Framework.	It does not support all the features and functionalities offered by the latest version of .NET Framework.

Table 14.2: Differences Between .NET Core and .NET Framework

14.3 .NET 7.0 and its Key Features

Microsoft unveiled .NET version 7.0 in 2022, which includes a Standard-Term Support (STS) or shorter supported version. Software products that are STS have a comparatively short life cycle and may be provided new features omitted from the Long-Term Support (LTS) edition. This is done so as to avoid the possibility of compromising the LTS release.

.NET 7.0 brings substantial changes and performance enhancements for developers.

Here are key enhancements available in .NET 7.0:

- **On-stack Replacement (OSR)** is introduced, allowing code modification during its execution, enabling long-running methods to transition to a faster version midway through execution.
- **Regex improvements** enhance regular expressions in .NET 7.0.

- **Simplified LINQ** ordering streamlines LINQ operations.
- **Dynamic Profile-guided Optimization (PGO) Improvements** deviates from Static PGO, eliminating the necessity for separate tools and training. Collecting optimization data is as simple as running the application.
- **Reflection Improvements** are included wherein .NET 7.0 reduces the overhead when invoking members using reflection.
- **Application Trimming Improvements** optimize applications by removing unnecessary components, resulting in smaller .exe sizes.
- **Trimming Libraries** that are redundant or unnecessary for specific use cases.
- **Nanoseconds and Microseconds** in Date Time offer more precise time values from the Date Time object.
- **Memory Caching Improvements**, particularly in ASP.NET, enhance memory caching techniques.
- **TAR File Creation is available.** In addition to .ZIP file creation in .NET 6.0, .NET 7.0 now supports .TAR file creation.
- **Blazor Changes are included** wherein updates include Blank Templates and a Sample Blazor Template with no pre-populated sample data.

14.4 Creating an Application with .NET 7.0 and VS 2022

Following are the steps to create a simple ‘HelloWorld’ console application using .NET 7.0:

1. Start Visual Studio 2022. Click **File → New → Project**. The **New Project** dialog box appears.

Choose **C#** followed by **Windows** and then, **Console**. Next, choose the **Console App** project template as shown in Figure 14.1.

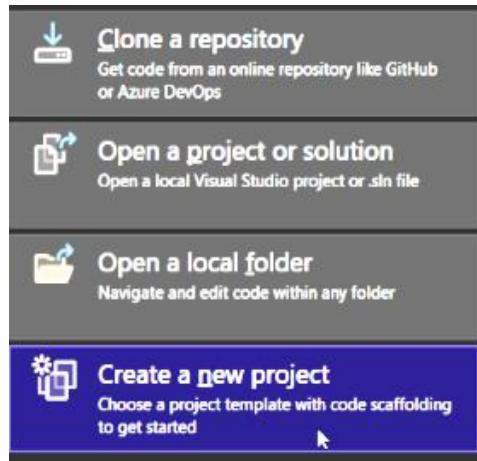


Figure 14.1: New Project Dialog Box

2. Click **Next**.
3. Provide the name 'HelloWorld' in the **Project name** field as shown in Figure 14.2 and click **Next**.

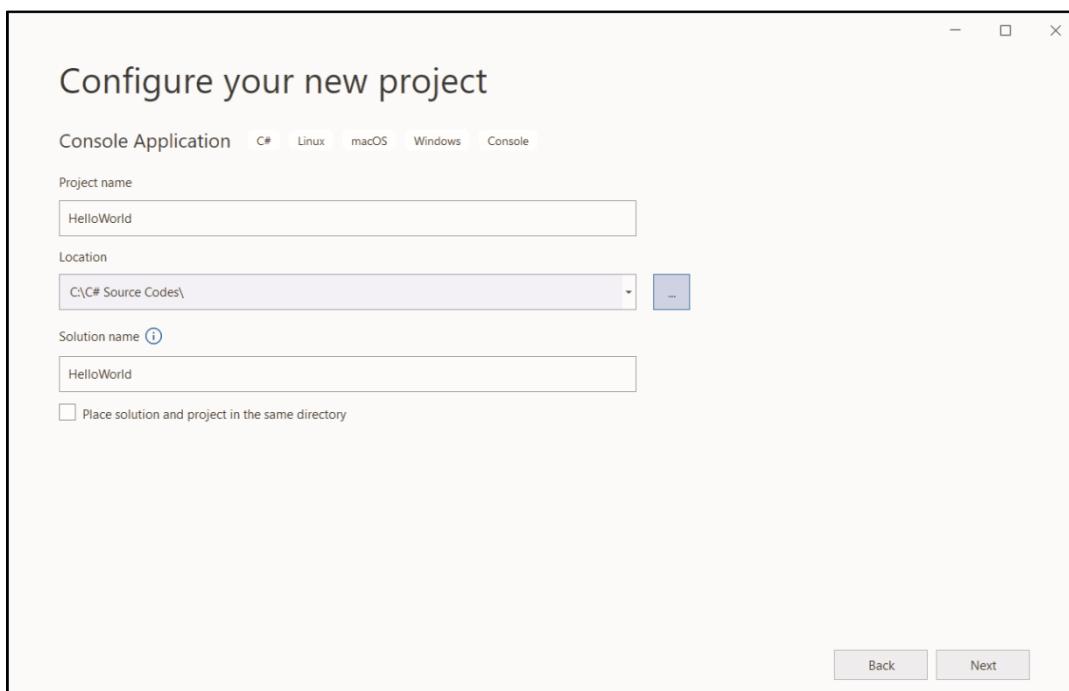


Figure 14.2: Configuring New Project

4. Select the Target Framework as shown in Figure 14.3 and then, click **Create**.

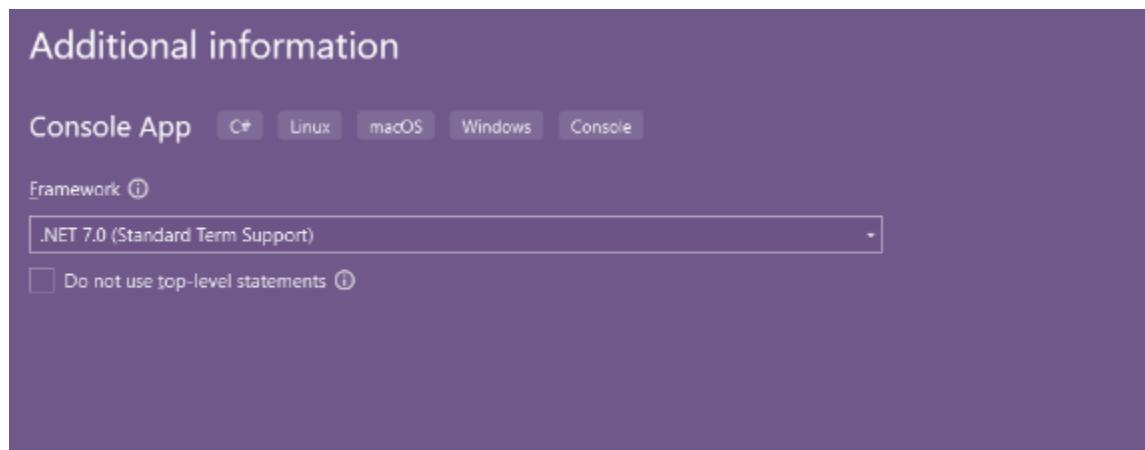


Figure 14.3: New Project Dialog Box - Selecting Target Framework

The template generates a simple HelloWorld application as shown in Figure 14.4.

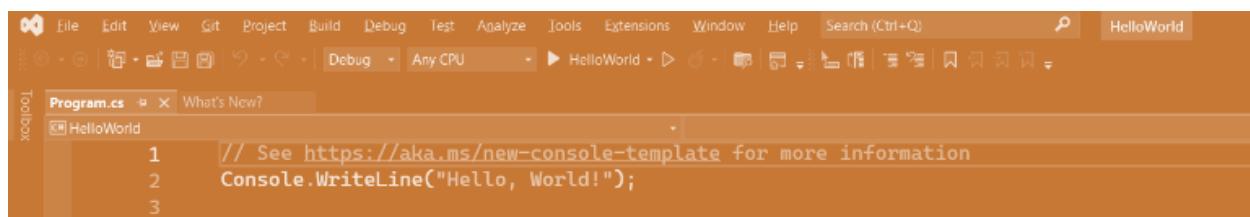


Figure 14.4: Default HelloWorld Template

The `Console.WriteLine()` method is called, which is intended to show the literal string ‘Hello World!’ in the console window. In the toolbar, clicking the project name (in this case, `HelloWorld`) button that is preceded by a green arrow will execute the program in Debug mode.

5. On the menu bar, click **Build → Build Solution**. This compiles the program into an IL code, which is transformed into binary code by a Just-In-Time (JIT) compiler, similar to how it happens in regular .NET applications.
6. Execute the program. Figure 14.5 shows the output of the program in the debug console window.



Figure 14.5: Output of the Program

Thereafter, the developer will be prompted to press any key to exit the console window. These steps illustrated the creation of a basic console application using .NET. In a similar manner, the developer can also create complex applications using this platform.

14.5 .NET 8.0 and the Future

Establishing a dependable release schedule for .NET offers notable benefits. In earlier times, when developers depended on the older .NET Framework, releases were infrequent, and alterations were typically limited. Although this was not troublesome, the fact that the development platform was synchronized with the Windows release cycle occasionally resulted in extended waits for anticipated features.

The move to an open-source framework resulted in combining .NET Core and .NET Standard into a single .NET version. Developers now receive annual .NET updates aligned with the annual .NET Conf event in November, with 18 months of support for STS versions and three years for LTS versions. Odd-numbered versions are STS and even-numbered versions are LTS.

.NET 8.0, arriving in November 2023, is an LTS version, offering stability and support. It builds on .NET 7.0's changes and adds more features.

As the .NET platform evolves, Microsoft is actively making a range of adjustments to the underlying compiler to seamlessly integrate emerging language capabilities and features. These modifications are predominantly materialized through the introduction of a new version of C#.

Further enhancements anticipated are:

- **Containerization and DevOps:** Containerization is a key trend and .NET is likely to further integrate with container technologies such as Docker and Kubernetes. This will make it easier to build, deploy, and manage .NET applications in containerized environments.
- **Security:** .NET will continue to adapt to evolving security requirements. Expect improvements in security features, best practices, and libraries to help developers write more secure applications.
- **IoT and Edge Computing:** As Internet of Things (IoT) and edge computing gain importance, .NET may provide better support for developing applications in these domains.
- **Machine Learning and AI:** Integration with machine learning and artificial intelligence frameworks may become more seamless, allowing developers to leverage AI capabilities in their .NET applications.
- **Enhanced Tooling:** Visual Studio and Visual Studio Code are central to the .NET developer experience. Expect improvements in these tools and the development experience they offer.
- **Open-Source Contributions:** The .NET ecosystem will likely continue to benefit from contributions from the open-source community. This includes both enhancements to the core .NET runtime and the development of open-source libraries and frameworks.

14.6 Summary

- .NET Core is an open-source .NET Framework platform available for Windows, Linux, and Mac OS.
- The key features of .NET Core are namely, cross-platform and container support, high performance, asynchronous operations via `async/await`, unified MVC and Web API Frameworks, multiple environments and development mode, dependency injection, and support for microservices.
- All the libraries from the original .NET Framework are not available in .NET Core due to its lightweight nature and only some are available.
- Microsoft rebranded .NET Core to simply .NET, starting with .NET 5.0 and continuing with subsequent versions such as .NET 6.0.
- .NET 7.0 provides many enhancements and features.
- .NET 8.0, an LTS release, is expected to offer stability and additional features.

14.7 Check Your Progress

1. Which aspect of .NET 7.0 allows code modification during its execution, enabling long-running methods to transition to a faster version midway through execution?

(A)	Entity Framework Updates
(B)	Cloud Integration
(C)	Cross-Platform Development
(D)	On-stack Replacement (OSR)

2. _____ are software applications containing small and modular business services.

(A)	Microservices
(B)	System services
(C)	Xamarin services
(D)	Cloud services

3. Which of the following is not a feature of .NET Core?

(A)	It must be set up as a single package and runtime environment for Windows
(B)	It can be packaged and set up irrespective of the underlying OS
(C)	It utilizes a redesigned CLR known as CoreCLR and presents a modular collection of libraries
(D)	It is compatible with Xamarin via the .NET Standard Library 5

4. Which of these statements are true for .NET 7.0?

(A)	Dynamic PGO Improvements
(B)	Application Trimming Improvements
(C)	Trimming Libraries
(D)	TAR File Creation is available

14.7.1 Answers

1.	D
2.	A
3.	A
4.	A, B, C, and D

Try It Yourself

1. Write a C# program using .NET 7.0 that calculates the factorial of a given integer. The factorial of a non-negative integer 'n' is the product of all positive integers from 1 to 'n'. Implement this calculation using recursion.

The program should:

- Prompt the user to input a non-negative integer.
- Use recursion to calculate the factorial of the input integer.
- Display the result.

2. Implement a C# program in .NET 7.0 that reads a list of integers from the user, calculates the sum of all even numbers in the list, and then displays the result. Ensure that the program handles user input errors gracefully, such as non-integer inputs or empty lists.

Appendix

Sr. No.	Case Studies
1.	<p>Problem Statement: City General Hospital is a large healthcare facility in California, United States. It is struggling to efficiently manage patient information, appointments, and medical records. The existing paper-based system is error-prone, time-consuming, and often leads to confusion and delays in patient care. The hospital requires a comprehensive Hospital Management System (HMS) to streamline hospital operations, improve patient care, and enhance administrative efficiency.</p> <p>Project Scope: Develop an HMS using C# 10.0, .NET 7.0/.NET Framework 4.8, SQL Server 2022, and Visual Studio 2022 IDE that encompasses following features:</p> <p>Patient Registration: Capture and store patient demographics, medical history, and contact information.</p> <p>Appointment Scheduling: Enable patients to book appointments online and facilitate appointment management for healthcare providers.</p> <p>Medical Records: Digitize patient records, including diagnoses, treatments, and prescriptions, ensuring easy access for authorized personnel.</p> <p>Billing and Invoicing: Generate and manage invoices for medical services, enabling online payment.</p> <p>Staff Management: Manage staff details, roles, and shifts and track attendance.</p> <p>Inventory Management: Maintain an inventory of medical supplies and equipment, with alerts for low stock.</p> <p>Reporting: Generate reports on patient statistics, billing, and inventory.</p>
2.	<p>Problem Statement: Aura Bus Services, a prominent bus transportation company, faces challenges in managing ticket reservations, passenger information, and route scheduling. The current manual system is prone to errors, resulting in overbookings, confusion, and operational inefficiencies.</p>

The client seeks a robust Bus Ticket Management System (BTMS) to automate ticket booking, streamline passenger information management, and improve route scheduling.

Project Scope:

Develop a Bus Ticket Management System using C# 10.0, .NET 7.0/.NET Framework 4.8, SQL Server 2022, and Visual Studio 2022 IDE that encompasses following features:

Ticket Booking: Enable passengers to book tickets online, specifying the date, route, and seat preference.

Passenger Management: Capture and store passenger details, including name, contact information, and booking history.

Route and Schedule Management: Define and manage bus routes, schedules, and stops.

Seat Allocation: Automatically assign and manage seat bookings based on availability.

Pricing and Discounts: Implement dynamic pricing and discounts based on demand and booking history.

Ticket Printing: Generate e-tickets with QR codes for easy validation.

Payment Integration: Integrate with payment gateways for online payment processing.

Reporting: Generate reports on ticket sales, passenger demographics, and route performance.

Admin Dashboard: Provide administrators with a dashboard for managing buses, routes, and bookings.

Security: Ensure data security and privacy, including passenger information.

3. **Project Title:** School Management System (EducationPro)

Problem Statement:

Global International School, a leading educational institution, is grappling with the challenges of manual record-keeping, communication gaps, and complex administrative processes. The

current system leads to inefficiencies, data redundancy, and difficulties in managing student information, staff, and academic operations. The School management, who is now a client of yours, seeks a comprehensive School Management System (EducationPro) to streamline school operations, improve communication, and enhance the overall educational experience.

Project Scope:

Develop a School Management ERP using C# 10.0, .NET 7.0/.NET Framework 4.8, SQL Server 2022, and Visual Studio 2022 IDE that encompasses following features:

Student Information Management: Capture and maintain student records, including personal details, attendance, and academic performance.

Teacher and Staff Management: Manage staff details, roles, attendance, and payroll information.

Admissions and Enrolment: Facilitate student admissions and enrolment processes.

Attendance and Timetable: Track student and teacher attendance and generate class timetables.

Grades and Assessment: Record and calculate student grades and manage assessments.

Communication Portal: Provide a platform for parents, teachers, and administrators to communicate and share information.

Library Management: Automate library operations, including cataloging, borrowing, and returning books.

Finance and Billing: Manage school finances, including fees, invoices, and payroll.

Inventory Management: Maintain inventory of school supplies and equipment.

Security: Implement role-based access control for data privacy.

Reporting: Generate reports on student performance, staff management, finances, and more.