

Building Applications Using C#

Building Applications Using C#

Trainer's Guide

© 2014 Aptech Limited

All rights reserved.

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

All trademarks acknowledged.

APTECH LIMITED

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

Edition 1 - 2014



Dear Learner,

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

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

- Scanning the user system and needs assessment

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

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

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

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

- Needs analysis and design of curriculum

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

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

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

- Design & development of instructional materials

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

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

➤ Strategies for delivery of instruction

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

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

➤ Assessment of learning

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

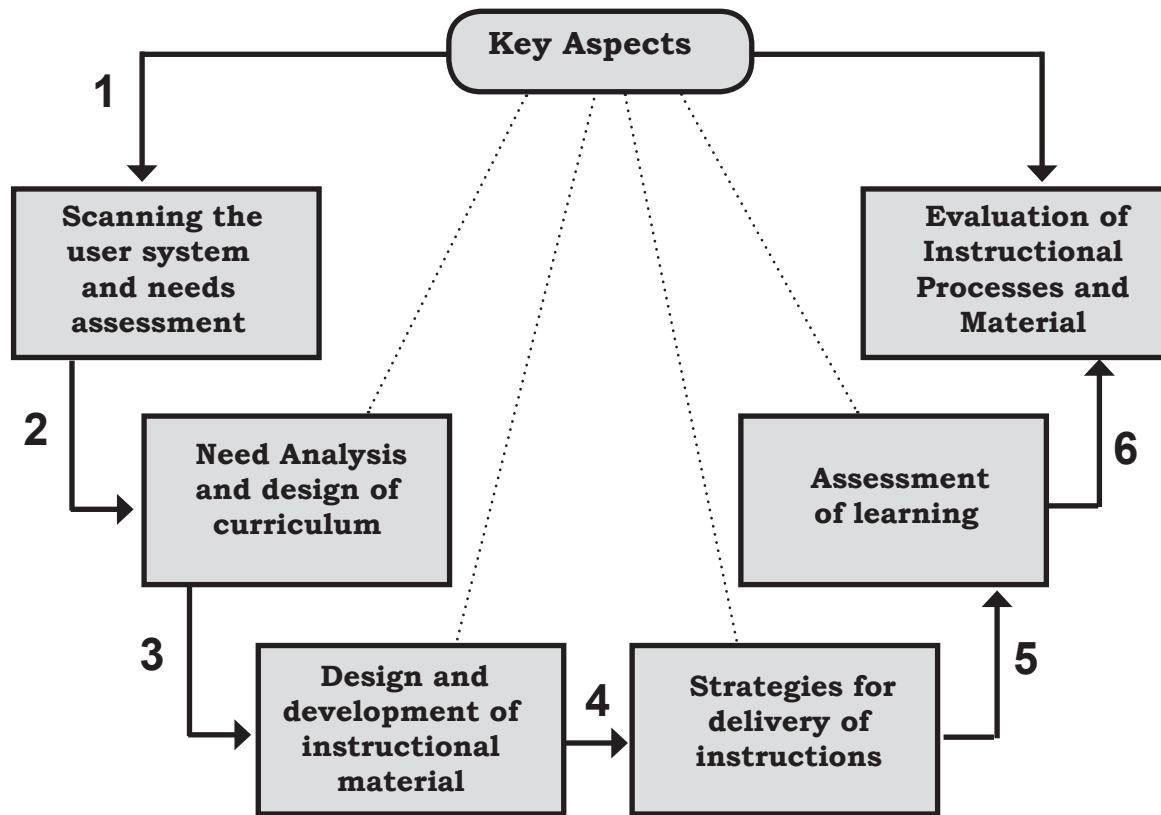
➤ Evaluation of instructional process and instructional materials

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

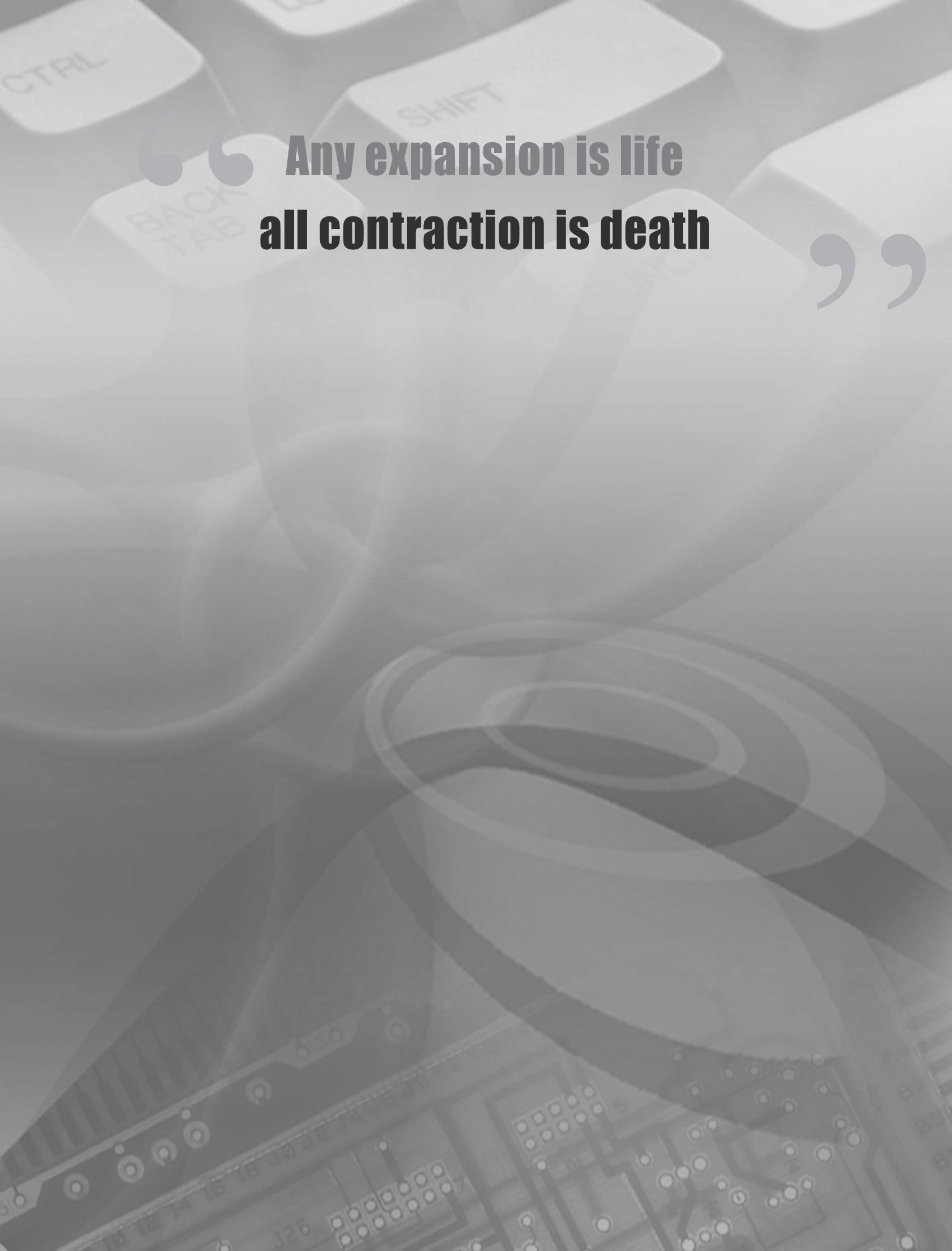
*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

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

Aptech New Products Design Model



“ Any expansion is life
all contraction is death ”



Preface

The Trainer's Guide for **Building Applications Using C#** covers the features of the C# language. The book begins by introducing .NET Framework 4.5, describing the basic features of C#, and then, explaining the object-oriented capabilities of C#. The book also describes the Visual Studio 2012 Integrated Development Environment (IDE). The book explains various advanced features of C# such as delegates, query expressions, advanced types such as partial types, nullable types, and so on. The book also describes parallel programming and enforcing data security through encryption. The faculty/trainer should teach the concepts in the theory class using the slides. This Trainer's Guide will provide guidance on the flow of the session and also provide tips and additional examples wherever necessary. The trainer can ask questions to make the session interactive and also to test the understanding of the students.

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

We will be glad to receive your suggestions.

Design Team

**“Learning how to learn is
life’s most important skill”**

Table of Contents

Sessions

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

“Knowing is not enough

we must apply;

Willing is not enough,

we must do”



Session 1 - Getting Started with C#

1.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth.

1.1.1 Objectives

By the end of this session, the learners will be able to:

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

1.1.2 Teaching Skills

To teach this session successfully, you must know about the .NET Framework. You should be aware of the C# language features. You should also be familiar with the Visual Studio 2012 environment and the elements of Microsoft Visual Studio 2012 IDE.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here for the In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to the .NET Framework. They will learn about the C# language features. This session will also discuss the Visual Studio 2012 environment and the elements of Microsoft Visual Studio 2012 IDE.

1.2 In-Class Explanations

Slide 3

Understand the .NET Framework.

The slide has a blue header bar with the title 'Introduction to .NET Framework'. Below the header is a white content area containing a bulleted list. At the bottom of the slide is a dark footer bar with the copyright information '© Aptech Ltd.' and the page number '3'.

Introduction to .NET Framework

- ◆ The .NET Framework is an infrastructure that is used to:
 - ❖ Build, deploy, and run different types of applications and services using .NET technologies.
 - ❖ Minimize software development, deployment, and versioning conflicts.

© Aptech Ltd. Building Applications Using C# / Session 1 3

Use slide 3 to explain that .NET Framework is an infrastructure that enables building, deploying, and running different types of applications and services using .NET technologies.

Explain to the students that, typically, for creating an application, a developer may use a tool to create software, another tool to compile, yet another tool to deploy on a client machine, and finally one more tool to manage the software. Instead of using a number of such tools, a developer can now use a single product – the Microsoft .NET Framework.

Tell the students that the .NET Framework can be used to minimize software development effort as it provides many features that help faster and efficient coding. The .NET Framework minimizes deployment issues as it offers tools for deployment. Finally, the .NET Framework minimizes versioning conflicts as it supports the execution of multiple versions of applications side by side.

Slides 4 to 6

Understand the evolution of the .NET Framework architecture.

The .NET Framework Architecture 1-8

- ◆ With improvements in networking technology:
 - ◆ Distributed computing has provided the most effective use of processing power of both client and server processors.
 - ◆ Applications became platform-independent with the emergence of Internet, which ensured that they could be run on PCs with different hardware and software combination.
 - ◆ Communication with each other has become possible for the clients and servers in a vendor-independent manner.

© Aptech Ltd. Building Applications Using C# / Session 1 4

The .NET Framework Architecture 2-8

- ◆ The following figure shows the different features accompanying the transformation in computing, Internet, and application development:

```

graph LR
    A([Transformation in Computing]) --> B[Grouping Web Server]
    B --- C[Receiving data in HTML code]
    B --- D[Enabling communication in vendor-independent manner]
    E([Transformation in the Internet]) --> D
    F([Transformation in Application Development]) --> G[Application used anywhere and by anyone]
  
```

© Aptech Ltd. Building Applications Using C# / Session 1 5



The .NET Framework Architecture 3-8

- ◆ All the transformations are supported by the technology platform introduced by Microsoft called as **.NET Framework**.
- ◆ Data stored using the .NET Framework is accessible to a user from any place, at any time, through any .NET compatible device.
- ◆ The .NET Framework:
 - ◆ Is a programming platform that is used for developing Windows, Web-based, and mobile software.
 - ◆ Has a number of pre-coded solutions that manage the execution of programs written specifically for the framework.
 - ◆ Is based on two basic technologies for communication of data:
 - eXtensible Markup Language (XML)
 - The suite of Internet protocols

© Aptech Ltd.

Building Applications Using C# / Session 1 6

Through slides 4 to 6, trace the evolution of the .NET framework and explain the technological advancements which led to its emergence.

In slide 4, explain to the students the various improvements in networking technology. Distributed computing is a technique in which the different parts of an application exist independently on multiple computers, which can be invoked simultaneously over a network.

For example, a banking application can be split to have the loan processing module on one machine, account processing module on another machine and so on.

Distributed computing provided the most effective use of processing power of both client and server processors. Later, applications became platform-independent with the emergence of Internet. Communication between clients and servers became possible in a vendor-independent manner.

During this phase, various kinds of transformations started to take place. Use slide 5 to explain the different features accompanying the transformation in computing, Internet, and application development.

Using slide 6, explain to the students that all the transformations led to the emergence of the .NET Framework. Explain that the data stored using the .NET Framework is accessible to a user from any place, at any time, through any .NET compatible device. Tell the students that the .NET Framework is a programming platform that is used for developing Windows, Web-based, and mobile software. Explain its features as given on the slide 6.

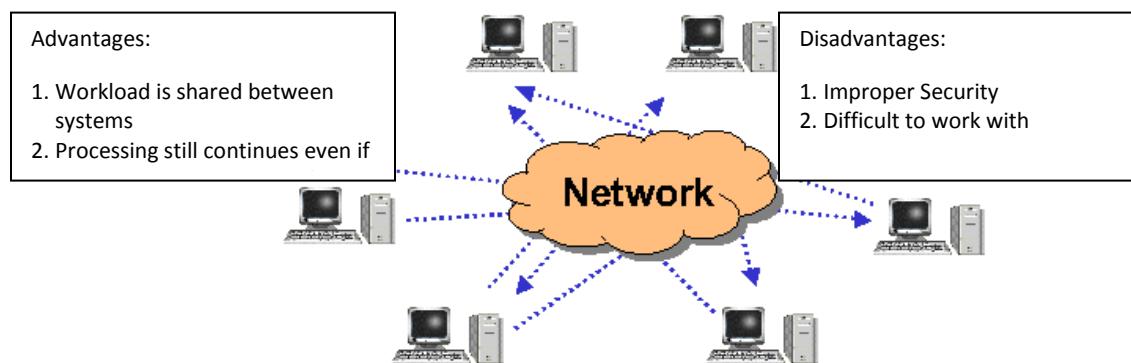
Additional Information

When PCs were introduced, they became highly popular within a short time. The PC became a ‘client’ that requested information from the ‘server’, which could be a mainframe or even a high-end PC. Thus, client-server computing was born.

While such a scenario reduced the load on the server, the clients still depended on it. This meant that the load on the network continued to be present. Moreover, with time, the processors became more powerful. However, their potential was not being totally utilized in such a client-server scenario. As a result, for most of the time, the processor was idle. Therefore, the need for a computing architecture was felt, where the processor time would be completely utilized, and the dependence on a server would be removed.

The next phase of computing namely ‘Distributed Computing’, evolved out of the client-server architecture. The processing power of the PC was much more than what was utilized in the Client-Server architecture so it was assumed that if a job could be distributed between several computers, it would be done more quickly. The distribution of jobs between computers in a network enhanced the utilization of the system’s processing capability as different computers could end up doing several jobs at the same time in parallel, and the workload would also be shared. Thus, a job that would take a supercomputer some time to do could be completed in much lesser time using such a scenario.

The following figure illustrates the concept of “Distributed Computing”, and lists its advantages and disadvantages. You can draw an illustration or sketch similar to this to explain to the students.



Such a scenario was popular in a local network. However, with the popularity of the Internet, distributed computing was also used on the Internet.

The transformation in computing ensured that distributed computing was here to stay. On the other hand, the transformation in the Internet led to the blending of communications and computing because of the same applications being available on devices other than computers. Moreover, Web sites evolved into constellations from the isolated islands that they were. In this context, it can be said that .NET is a whole new platform centered on the Internet. With .NET, user data lives on the net. This data is accessible to the user from any place, at any time,

and through any .NET compatible device. It also enables the user to create applications that harness the power of the Internet.

Slide 7

Understand the .NET Framework architecture.

The slide has a blue header bar with the title "The .NET Framework Architecture 4-8". Below the title is a list of bullet points:

- ◆ Following are the key features of XML:
 - ❖ It separates actual data from presentation.
 - ❖ It unlocks information that can be organized, programmed, and edited.
 - ❖ It allows Web sites to collaborate and provide groups of Web services. Thus, they can interact with each other.
 - ❖ It provides a way for data to be distributed to a variety of devices.
- ◆ Apart from XML, the .NET platform is also built on Internet protocols such as:
 - ❖ Hypertext Transfer Protocol (HTTP)
 - ❖ Open Data Protocol (OData)
 - ❖ Simple Object Access Protocol (SOAP)

At the bottom left is the copyright notice "© Aptech Ltd." and at the bottom right is the page number "7".

Use slide 7 to explain the students the key features of XML. Tell them that XML is a markup language and is used to separate actual data from presentation. Explain that XML was designed in order to describe data. In XML, tags are not predefined. You must define your own tags. XML uses a Document Type Definition (DTD) or an XML Schema to describe the data.

Tell the students that XML unlocks information that can be organized, programmed, and edited. Then, tell that it allows Web sites to collaborate and provide groups of Web services. Thus, they can interact with each other.

Also, tell that XML provides a way for data to be distributed to a variety of devices.

Mention that apart from XML, the .NET platform is also built on Internet protocols such as:

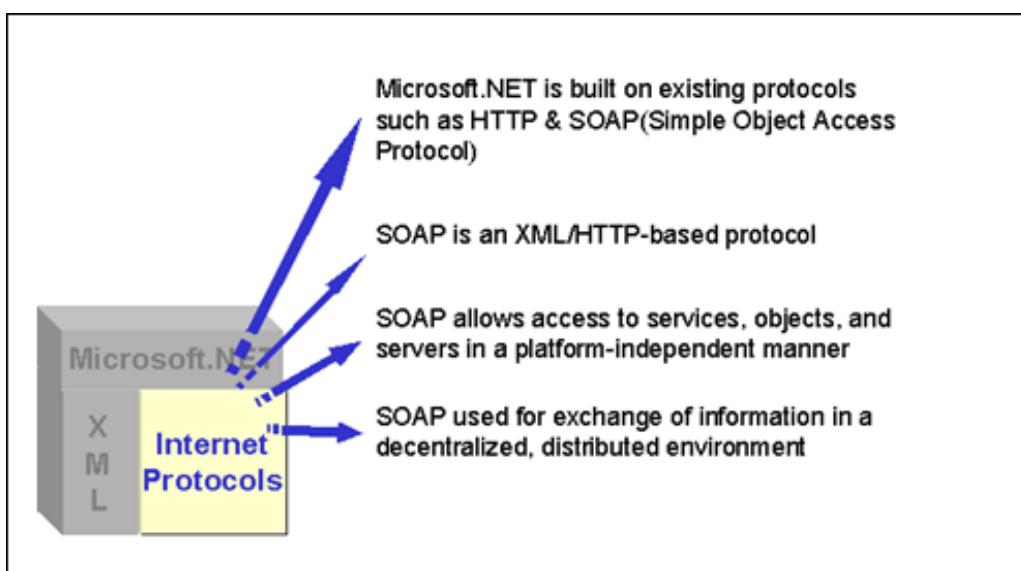
- Hypertext Transfer Protocol (HTTP)
- Open Data Protocol (OData)
- Simple Object Access Protocol (SOAP)

Early distributed computing technologies relied on binary protocols. The problems with these protocols are that they provide different encoding data formats and risky communication with the destination point, if the point is within the firewall. Dealing with different encoding formats prevented 100 percent interoperability over the Web. Communicating with the destination point within the firewall required opening multiple ports for communication. This made the system susceptible to unauthorized access. To solve these problems, World Wide Web (WWW) provides a global solution by introducing platform-independent standards. These standards include Hypertext Transfer Protocol (HTTP), and others. HTTP uses HTML or XML to request and store information on Intranet or Internet and can safely communicate with the firewall.

XML deals with plain text and thereby ensures interoperability.

Web Services use Simple Object Access Protocol (SOAP) to communicate with heterogeneous systems by exchanging messages. SOAP is a platform-independent protocol that uses XML to exchange information. A Web Service is the ideal solution to all the problems of traditional distributed computing.

The following figure lists the features of SOAP. These suit the requirements of the .NET platform perfectly, and hence this protocol has been adopted as one of the core components.



Slide 8

Understand the .NET Framework architecture.

The .NET Framework Architecture 5-8

- ◆ In traditional Windows applications:
 - ❖ Codes were directly compiled into the executable native code of the operating system.
- ◆ Using the .NET Framework:
 - ❖ The code of a program is compiled into CIL (formerly called MSIL) and stored in a file called assembly.
 - ❖ This assembly is then compiled by the CLR to the native code at run-time.
- ◆ The following figure represents the process of conversion of CIL code to the native code:

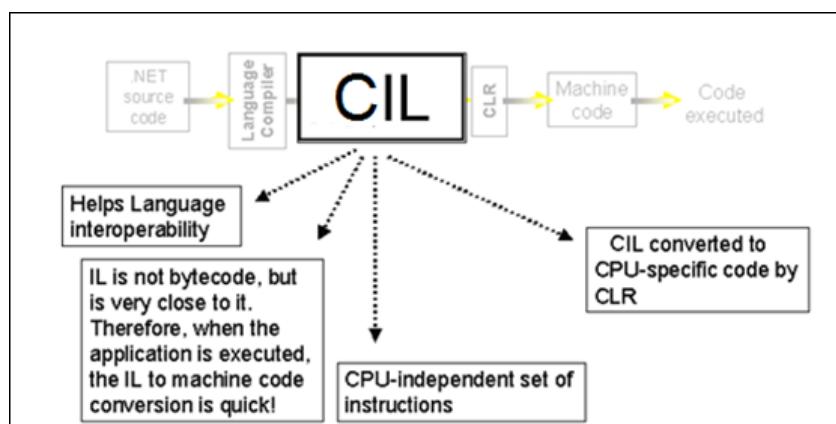
```

graph TD
    A[.NET code] --> B[.NET class library]
    A -. "Stored in assembly" .-> C[CIL code]
    C --> D[Compiled into native code]
    D --> E[Execution of code]
    E --> F[User]
    F -. "CLR" .-> D
  
```

© Aptech Ltd. Building Applications Using C# / Session 1 8

Using slide 8, explain the process of conversion of Common Intermediate Language (CIL) code to native code. Tell the students that in traditional Windows applications, the codes were directly compiled into the executable native code of the operating system.

Explain that using the .NET Framework, the code of a program is compiled into CIL (formerly called MSIL) and stored in a file called assembly. This assembly is then compiled by the CLR to the native code at run-time. You can show students the figure in slide 8 that represents the process of conversion of CIL code to the native code. You can further explain about the role of CIL using the following figure.



In-Class Question:

Ask the students to name the three Internet protocols mentioned earlier.

Answer: HTTP, OData, and SOAP

Slide 9

Understand the features of CLR.

The .NET Framework Architecture 6-8

- ◆ The CLR provides many features such as:
 - ◆ Memory management
 - ◆ Code execution
 - ◆ Error handling
 - ◆ Code safety verification
 - ◆ Garbage collection
- ◆ The applications that run under the CLR are called managed code.

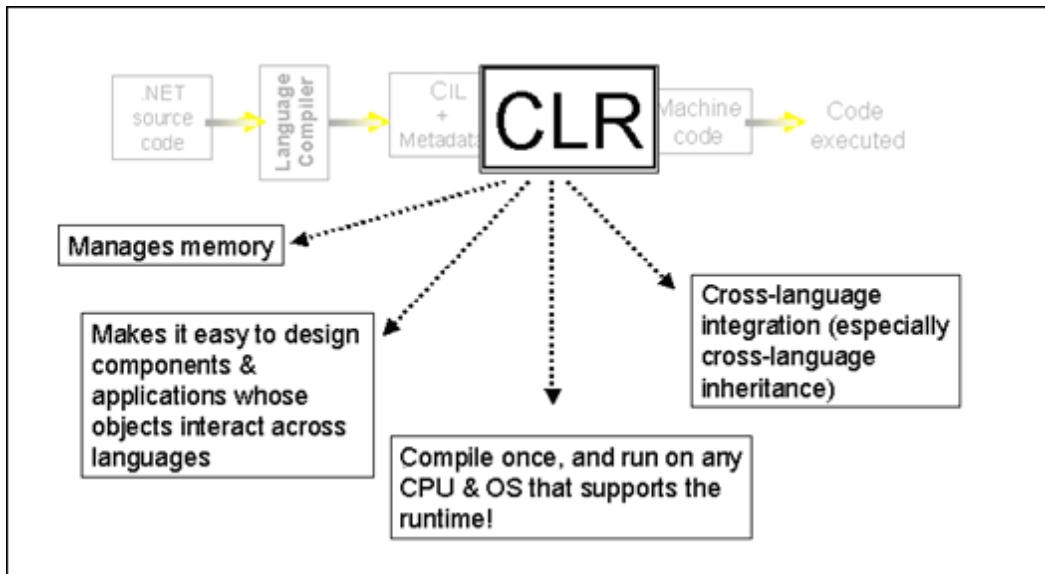
© Aptech Ltd.

Building Applications Using C# / Session 1 9

In slide 9, explain about CLR and managed code. Explain to the students that the Common Language Runtime which is called CLR in short is one of the key components of the .NET Framework. Tell the students that the CLR provides many features such as memory management, code execution, error handling, code safety verification, and garbage collection.

Explain that the applications that run under the CLR are called managed code.

The role of the CLR is outlined in the following figure:



Slide 10

Understand the different versions of the .NET Framework.

The .NET Framework Architecture 7-8

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

.NET Framework 1.0	Red
.NET Framework 1.1	Green
.NET Framework 2.0	Purple
.NET Framework 3.0	Cyan
.NET Framework 3.5	Orange
.NET Framework 4.0	Red
.NET Framework 4.5	Green

© Aptech Ltd. Building Applications Using C# / Session 1 10

In slide 10, tell the students that Microsoft has released different versions of the .NET Framework to include additional capabilities and functionalities with every newer version. Explain the different versions of the .NET Framework.

Tell that the .NET Framework 1.0 is the first version released with Microsoft Visual Studio .NET 2002. It includes CLR, class libraries of .NET Framework, and ASP. NET a development platform used to build Web pages.

Then, tell that .NET Framework 1.1 is first upgraded version released with Microsoft Visual Studio .NET 2003. It was incorporated with Microsoft Windows Server 2003 that supports components used to create applications for mobiles as a part of the framework, Oracle databases as a repository to store information in tables, IPv6 protocol and Code Access Security (CAS) for Web-based applications, and mention that it enables running assemblies of Windows Forms from a Web site.

Tell that the .NET compact framework provides components to create applications to be used in mobile phones and PDAs.

Also, tell that the .NET Framework 2.0 is the successor to .NET Framework 1.1 and next upgraded version included with Microsoft Visual Studio .NET 2005 and Microsoft SQL Server 2005.

Explain the new features of this version.

Tell that its support for 64-bit hardware platforms, generic data structures, new Web controls used to design Web applications, and exposure to .NET Micro framework which allows developers to create graphical devices in C#.

Explain that the .NET Framework 3.0 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.

Also, tell that the .NET Framework 3.5 is the next upgraded version and is included with Visual Studio .NET 2008. The primary features of this release are support to develop AJAX-enabled Web sites and a new technology named 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.

Mention that the .NET Framework 4.0 version included with 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 provides improvement in ADO.NET, WCF, and WPF, and introduces new language features, such as dynamic dispatch, named parameters, and optional parameters.

Then, tell that the .NET Framework 4.5 version included with Visual Studio .NET 2012 provides enhancements to .NET Framework 4.0, such as enhancement in asynchronous programming

through the `async` and `await` keywords, support for Zip compression, support for `regex` timeout, and more efficient garbage collection.

Slide 11

Understand the different versions of the .NET Framework and Visual Studio.

The .NET Framework Architecture 8-8

- The following table shows various versions of .NET Framework and Visual Studio:

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

© Aptech Ltd. Building Applications Using C# / Session 1 11

Use slide 11 to refer to the table that summarizes the evolution of .NET Framework versions that were distributed with Windows OS and the corresponding IDE name. Ever since 2002, various versions of Visual Studio IDE have been released to support .NET Framework application development. Mention to the student that the name of the IDE is not necessarily relative to the year of the release.

Slide 12

Understand the .NET Framework fundamentals.

The .NET Framework Fundamentals

- ◆ The .NET Framework is an essential Windows component for building and running the next generation of software applications and XML Web services.
- ◆ The .NET Framework is designed to:
 - ◆ Provide consistent object-oriented programming environment.
 - ◆ Minimize software deployment and versioning conflicts by providing a code-execution environment.
 - ◆ Promote safe execution of code by providing a code-execution environment.
 - ◆ Provide a consistent developer experience across varying types of applications such as Windows-based applications and Web-based applications.

VB C++ C# JScript ...

Common Language Specification

Application Class Libraries & Services

Base Class Library

Common Language Runtime

Visual Studio .Net

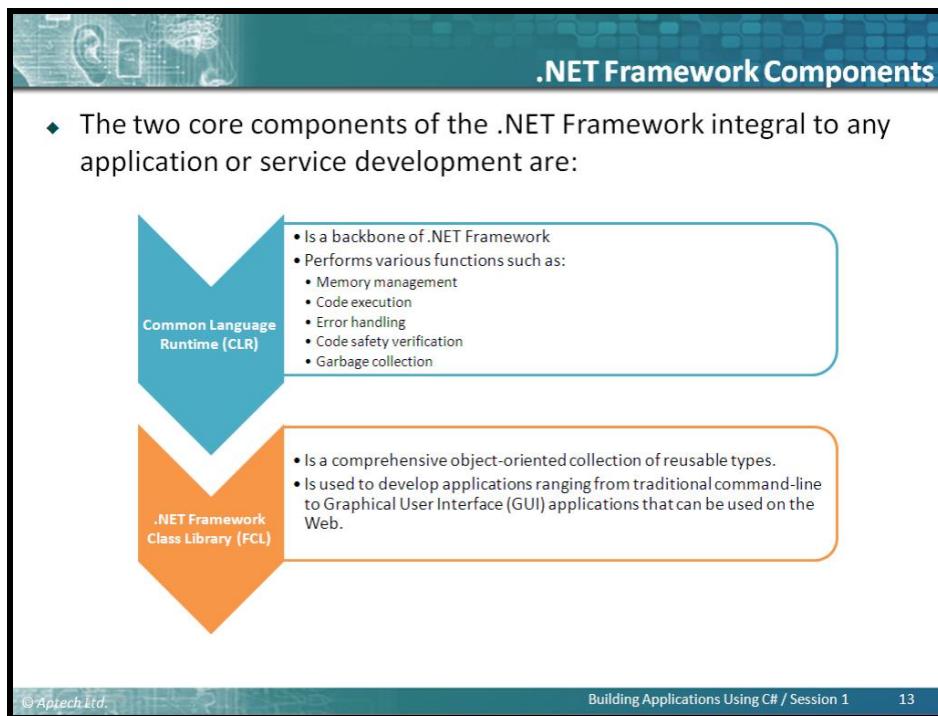
© Aptech Ltd. Building Applications Using C# / Session 1 12

In slide 12, explain to the students that the .NET Framework is an essential Windows component for building and running the next generation of software applications and XML Web services.

Explain to them that the .NET Framework is designed for various goals. It provides a consistent object-oriented programming environment by supporting languages such as C# and VB.NET. It helps minimize software deployment and versioning conflicts by providing a code-execution environment. It promotes safe execution of code by providing a code-execution environment and the CLR. It provides a consistent developer experience across varying types of applications such as Windows-based applications and Web-based applications.

Slide 13

Understand the .NET Framework components.



In slide 13, explain to the students that the .NET Framework is made up of several components. The two core components of the .NET Framework which are integral to any application or service development are, the CLR and the .NET Framework class library.

Tell them that the CLR is the backbone of .NET Framework. As mentioned in an earlier slide, it performs various functions such as:

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

Tell them that the .NET Framework Class Library (FCL) 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.

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

Slide 14

Understand the concept of using the .NET Framework.

The slide has a teal header bar with the title 'Using .NET Framework'. Below the header, there is a list of bullet points. A blue rectangular box labeled 'Example' contains another list of bullet points and a code snippet. At the bottom of the slide, there is a footer bar with copyright information and page numbers.

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.

Example

- ◆ To display a text message on the screen, the following command can be used:

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

- ◆ The same `WriteLine()` method will be used across all .NET languages.
- ◆ This is done by making the Framework Class Library as a common class library for all .NET languages.

© Aptech Ltd. Building Applications Using C# / Session 1 14

In slide 14, explain to the students that a programmer can develop applications using one of the languages supported by .NET such as C# and VB.NET. These applications make use of the base class libraries provided by the .NET Framework.

Give the students an example for this as shown on the slide. Tell them that the Framework Class Library acts as a common class library for all .NET languages and because of this, the same `WriteLine()` method can be used across all .NET languages.

Slides 15 to 18

Understand other components of .NET Framework.

Other Components of .NET Framework 1-4

- ◆ Following are some other important components:
 - ❖ Common Language Specification (CLS)
 - ❖ Common Type System (CTS)
 - ❖ Base Framework Classes
 - ❖ ASP.NET
 - ❖ ADO.NET
 - ❖ WPF
 - ❖ WCF
 - ❖ LINQ
 - ❖ ADO.NET Entity Framework
 - ❖ Parallel LINQ
 - ❖ Task Parallel Library

The diagram illustrates the structure of the .NET Framework. It is organized into several layers:

- .NET Framework** (Orange background): Contains Parallel LINQ, Task Parallel Library, LINQ, ADO.NET Entity Framework, WPF, WCF, WF, CardSpace, WinForms, ASP.NET, and ADO.NET.
- Base Framework Classes** (Light Blue background): Contains DLR.
- CLR** (Yellow background): Contains CLS and CTS.

Other Components of .NET Framework 2-4

- ◆ **Common Language Specification (CLS):**
 - ❖ Is 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.
- ◆ **ADO.NET:**
 - ❖ Provides classes to interact with databases.



Other Components of .NET Framework 3-4

- ◆ **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 the HTML.
 - ❖ Supports Web services that can be accessed using a standard set of protocols.
- ◆ **WPF:**
 - ❖ Is a UI framework based on XML and vector graphics.
 - ❖ Uses 3D computer graphics hardware and Direct3D technologies to create desktop applications with rich UI on the Windows platform.
- ◆ **WCF:**
 - ❖ Is a service-oriented messaging framework.
 - ❖ Allows creating service endpoints and allows programs to asynchronously send and receive data from the service endpoint.

© Aptech Ltd. Building Applications Using C# / Session 1 17



Other Components of .NET Framework 4-4

- ◆ **LINQ:**
 - ❖ Is a component that provides data querying capabilities to a .NET application.
- ◆ **ADO.NET Entity Framework:**
 - ❖ Is a set of technologies built upon ADO.NET that enables creating data-centric applications in object-oriented manner.
- ◆ **Parallel LINQ:**
 - ❖ Is a set of classes to support parallel programming using LINQ.
- ◆ **Task Parallel Library:**
 - ❖ Is a library that simplifies parallel and concurrent programming in a .NET application.

© Aptech Ltd. Building Applications Using C# / Session 1 18

In slide 15, explain to the students that while CLR and FCL are major components of the .NET Framework, there are several other important components as well.

Using slides 16 to 18, explain the various components. Tell the students that Common Language Specification (CLS) are a set of rules that any .NET language should follow to create applications that are interoperable with other languages. Explain that 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.

Explain that Base Framework Classes provide basic functionality such as input/output, string manipulation, security management, network communication, and so on.

Then, explain that 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 the HTML. ASP.NET also supports Web services that can be accessed using a standard set of protocols.

Also, tell that ADO.NET is the technology that enables data access and provides classes to interact with databases.

Mention that WPF is a UI framework based on XML and vector graphics. WPF uses 3D computer graphics hardware and Direct3D technologies to create desktop applications with rich UI on the Windows platform.

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

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

Explain that ADO.NET Entity Framework is a set of technologies built upon ADO.NET that enables creating data-centric applications in object-oriented manner.

Tell that Parallel LINQ is a set of classes to support parallel programming using LINQ.

Mention that Task Parallel Library is a library that simplifies parallel and concurrent programming in a .NET application.

You can refer to the figure in slide 15 that displays the various components of .NET Framework.

Additional Information

Refer the following links for further information:

<http://msdn.microsoft.com/en-us/library/vstudio/12a7a7h3%28v=vs.100%29.aspx>
<http://www.codeproject.com/Articles/7333/Understanding-NET-Framework-at-a-glance>

Slides 19 and 20

Understand the Common Intermediate Language (CIL).

Common Intermediate Language (CIL) 1-2

- ◆ Every .NET programming language generally has a compiler and a runtime environment of its own.
- ◆ The compiler converts the source code into executable code that can be run by the users.
- ◆ One of the primary goals of .NET Framework is to combine the runtime environments so that the developers can work with a single set of runtime services.
- ◆ When the code written in a .NET compatible language such as C# or VB is compiled, the output code is in the form of MSIL code.
- ◆ MSIL is composed of a specific set of instructions that indicate how the code should be executed.
- ◆ MSIL is now called as **Common Intermediate Language (CIL)**.

© Aptech Ltd. Building Applications Using C# / Session 1 19

Common Intermediate Language (CIL) 2-2

- ◆ The following figure depicts the concept of CIL:

```

graph LR
    VB((VB)) --> VB_Compiler[Compiler]
    CSharp((C#)) --> CSharp_Compiler[Compiler]
    VB_Compiler -- "Output Code" --> CIL_Box[CIL Code]
    CSharp_Compiler -- "Output Code" --> CIL_Box
  
```

© Aptech Ltd. Building Applications Using C# / Session 1 20

In slide 19, explain to the students that all .NET programming languages have a compiler and a runtime environment of their own. It is the job of the compiler to convert the source code in each language into executable code that can be run by the users. Mention that one of the

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

Tell them that when the code written in a .NET compatible language such as C# or VB is compiled, the output code is in the form of MSIL code. MSIL is composed of a specific set of instructions that indicate how the code should be executed. You can refer to the figure of slide 20, which depicts the concept of Microsoft Intermediate Language. Inform the students that MSIL is now called as CIL.

Additional Information

The CIL promotes language interoperability. Any language compilers targeting the .NET Framework produces an intermediate code named CIL, which, in turn, is compiled at run time by the CLR. With this feature, programs or modules written in one language become accessible to other languages. This enables developers to create applications in their preferred language or languages without worrying about the need to convert it to another language.

Slides 21 to 23

Understand Common Language Runtime (CLR).

Common Language Runtime (CLR) 1-3

- ◆ The CLR:
 - ❖ Is the foundation of the .NET Framework.
 - ❖ Acts as an execution engine for the .NET Framework.
 - ❖ Manages the execution of programs and provides a suitable environment for programs to run.
 - ❖ Provides a multi-language execution environment.
- ◆ The runtime manages code at execution time and performs operations such as:
 - ❖ Memory management
 - ❖ Thread management
 - ❖ Remoting
- ◆ The .NET Framework supports a number of development tools and language compilers in its Software Development Kit (SDK).

© Aptech Ltd. Building Applications Using C# / Session 1 21

Common Language Runtime (CLR) 2-3

- ◆ 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 runtime 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.
- ◆ The following figure depicts the working of the CLR:

```

graph LR
    CIL[CIL Code] -- CLR --> Executed[Code Executed]
  
```

© Aptech Ltd.

Building Applications Using C# / Session 1

22

Common Language Runtime (CLR) 3-3

- ◆ The following figure shows a more detailed look at the working of the CLR:

```

graph LR
    subgraph CLR [Common Language Runtime]
        direction TB
        S1[C#] --- C1[Compiler]
        S2[VB.NET] --- C2[Compiler]
        S3[F#] --- C3[Compiler]
        S4[Other] --- C4[Compiler]
        
        C1 --- CIL[CIL]
        C2 --- CIL
        C3 --- CIL
        C4 --- CIL
        
        CIL --- JIT[JIT Compilation]
        JIT --- EC[Executable Code]
        
        EC --- B1[100010001]
        EC --- B2[001010101]
        EC --- B3[010101010]
        EC --- B4[1010101]
    end
  
```

© Aptech Ltd.

Building Applications Using C# / Session 1

23

In slide 21, tell the students that the CLR is the foundation of the .NET Framework. The runtime manages code at execution time and performs operations such as memory management, thread management, and remoting.

Explain to them that in simple terms, the CLR acts as an execution engine for the .NET Framework. It manages the execution of programs and provides a suitable environment for

programs to run. The .NET Framework supports a number of development tools and language compilers in its Software Development Kit (SDK). Hence, the CLR provides a multi-language execution environment.

Using slides 22 and 23, explain the working of CLR. 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 runtime 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.

In slide 23, you can refer to the figure that shows a more detailed look at the working of the CLR.

All the code in .NET is managed by the CLR and is therefore, referred to as managed code. In managed code, developers allocate memory wherever and whenever required by declaring variables and the runtime garbage collector determines when the memory is no longer needed and cleans it all up. The garbage collector may also move memory around to improve efficiency.

The runtime manages it all for you and hence, the term managed code is used for such programs. On the other hand, code that runs without the CLR, such as C programs, is called unmanaged code.

In-Class Question:

Ask the students the following question to test their understanding of managed versus unmanaged code.



Is working with managed code better from a developer's point of view?

Answer: Yes, because the CLR takes care of many important tasks, thus, allowing the developer to focus on the application development.

Slide 24

Understand the Dynamic Language Runtime (DLR).

The slide has a blue header bar with the title 'Dynamic Language Runtime (DLR)' in white. Below the header is a white content area containing a bulleted list. At the bottom is a dark blue footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 1' in the center, and '24' on the right.

- ◆ Dynamic Language Runtime (DLR):
 - ◆ Is a runtime environment built on top of the CLR to enable interoperability of dynamic languages such as Ruby and Python with the .NET Framework.
 - ◆ Allows creating and porting dynamic languages to the .NET Framework.
 - ◆ Provides dynamic features to the existing statically typed languages. For example, C# relies on the DLR to perform dynamic binding.
- ◆ The .NET Framework languages, such as C#, VB, and J# are statically typed languages.
- ◆ In dynamic languages, programmers are not required to specify object types in the development phase.

In slide 24, tell the students that DLR is a runtime environment built on top of the CLR to enable interoperability of dynamic languages such as Ruby and Python with the .NET Framework.

Explain to them that the .NET Framework languages, such as C#, VB, and J# are statically typed languages, which mean that the programmer needs to specify object types while developing a program. On the other hand, in dynamic languages, programmers are not required to specify object types in the development phase. DLR allows creating and porting dynamic languages to the .NET Framework. In addition, DLR provides dynamic features to the existing statically typed languages. For example, C# relies on the DLR to perform dynamic binding.

Additional Information

Ruby and Python are open source languages that focus on simplicity and productivity.

For more information on the DLR, refer the following link:

<http://msdn.microsoft.com/en-us/library/dd233052%28v=vs.110%29.aspx>

Slide 25

Understand the need for a new language.

The slide has a blue header bar with the title 'Need for a New Language'. The main content area contains a bulleted list of reasons for introducing C#. The footer bar includes the Aptech logo, the slide title, and page number.

Need for a New Language

- ◆ Microsoft introduced C# as a new programming language to address the problems posed by traditional languages.
- ◆ C# was developed to:
 - ❖ Create a very simple and yet powerful tool for building interoperable, scalable, and robust applications.
 - ❖ Create a complete object-oriented architecture.
 - ❖ Support powerful component-oriented development.
 - ❖ Allow access to many features previously available only in C++ while retaining the ease-of-use of a rapid application development tool such as Visual Basic.
 - ❖ Provide familiarity to programmers coming from C or C++ background.
 - ❖ Allow to write applications that target both desktop and mobile devices.

© Aptech Ltd. Building Applications Using C# / Session 1 25

In slide 25, explain to the students that Microsoft introduced C# as a new programming language to address the problems posed by traditional languages. Explain the benefits of C# as mentioned on the slide.

Slide 26

Understand the purpose of C# language.

The slide has a blue header bar with the title 'Purpose of C# Language'. Below the title is a list of bullet points. At the bottom of the slide, there is a footer bar with the text '©Aptech Ltd.' on the left and 'Building Applications Using C# / Session 1 26' on the right.

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

In slide 26, tell the students that Microsoft .NET was formerly known as Next Generation Windows Services (NGWS). It is a completely new platform for developing the next generation of Windows/Web applications.

Explain to them that these applications transcend device boundaries and fully harness the power of the Internet. However, building the new platform required a language that could take full advantage. This is one of the factors that led to the development of C#.

Mention that 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.

C# has evolved from C and 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.

Slide 27

Understand the language features.

The slide has a blue header bar with the title 'Language Features'. The main content area contains a bulleted list of features. At the bottom, there is a footer bar with the Aptech logo, the slide title, and page number.

Language Features

- ◆ C# has features common to most object-oriented languages.
- ◆ It has language-specific features, such as:
 - ◆ Type safety checking
 - ◆ Generics
 - ◆ Indexers
- ◆ These features make the C# as a preferred language to create a wide variety of applications.

©Aptech Ltd. Building Applications Using C# / Session 1 27

Show slide 27 and tell the students that C# has features common to most object-oriented languages and in addition, it has language-specific features, such as type safety checking, generics, and indexers that make it the preferred language to create a wide variety of applications.

Slides 28 to 30

Understand the basic features of C#.

Basic Features of C# 1-3

- ◆ C# is a programming language designed for building a wide range of applications that run on the .NET Framework.
- ◆ Following are some basic key features of C#:
 - ◆ Object-oriented Programming
 - ◆ Type-safety Checking
 - ◆ Garbage Collection
 - ◆ Standardization by European Computer Manufacturers Association (ECMA)
 - ◆ Generic Types and Methods
 - ◆ Iterators
 - ◆ Static Classes
 - ◆ Partial Classes
 - ◆ Anonymous Methods
 - ◆ Methods with named Arguments
 - ◆ Methods with optional Arguments
 - ◆ Nullable Types
 - ◆ Accessor Accessibility
 - ◆ Auto-implemented Properties
 - ◆ Parallel Computing

©Aptech Ltd. Building Applications Using C# / Session 1 28

Basic Features of C# 2-3

- ◆ **Object-oriented Programming:**
 - ◆ Focuses on objects so that code written once can be reused. This helps reduce time and effort on the part of developers.
- ◆ **Type-safety Checking:**
 - ◆ Checked the overflow of types because uninitialized variables cannot be used in C# as C# is a case-sensitive language.
- ◆ **Garbage Collection:**
 - ◆ Performs automatic memory management from time to time and spares the programmer the task.
- ◆ **Standardization by European Computer Manufacturers Association (ECMA):**
 - ◆ Specifies the syntax and constraints used to create standard C# programs.
- ◆ **Generic Types and Methods:**
 - ◆ Are a type of data structure that contains code that remains the same throughout but the data type of the parameters can change with each use.
- ◆ **Iterators:** Enable looping (or iterations) on user-defined data types with the for each loop.
- ◆ **Static Classes:** Contain only static members and do not require instantiation.

©Aptech Ltd. Building Applications Using C# / Session 1 29

Basic Features of C# 3-3

- ◆ **Partial Classes:** Allow the user to split a single class into multiple source code (.cs) files.
- ◆ **Anonymous Methods:** Enable the user to specify a small block of code within the delegate declaration.
- ◆ **Methods with named Arguments:** Enable the user to associate a method argument with a name rather than its position in the argument list.
- ◆ **Methods with optional Arguments:** Allow the user to define a method with an optional argument with a default value.
- ◆ **Nullable Types:** Allow a variable to contain a value that is undefined.
- ◆ **Accessor Accessibility:** Allows the user to specify the accessibility levels of the get and set accessors.
- ◆ **Auto-implemented Properties:** Allow the user to create a property without explicitly providing the methods to get and set the value of the property.
- ◆ **Parallel Computing:** Support for parallel programming using which develop efficient, fine-grained, and scalable parallel code without working directly with threads or the thread pool.

© Aptech Ltd. Building Applications Using C# / Session 1 30

In slide 28, explain to the students that C# is a programming language designed for building a wide range of applications that run on the .NET Framework.

List and explain some of its key features as given on slides 28, 29, and 30.

Explain to the students that object-oriented programming focuses on objects so that code written once can be reused. This helps reduce time and effort on the part of developers.

Explain that type-safety checking are uninitialized variables that cannot be used in C#. Overflow of types can be checked. C# is a case-sensitive language. Also, tell that garbage collection is a process that performs automatic memory management from time to time and spares the programmer the task. Users cannot control when garbage collection will take place. Then, tell that standardization by the ECMA lays down guidelines, syntax, and constraints that will be used to create standard C# programs.

Mention that generic types and methods are a type of data structure that contains code that remains the same throughout but the data type of the parameters can change with each use. Then, mention that Iterators enable looping (or iterations) on user-defined data types with the foreach loop.

Explain that static classes contain only static members and do not require instantiation. Then, explain that partial classes allow the user to split a single class into multiple source code (.cs) files. Mention that anonymous methods enable the user to specify a small block of code within the delegate declaration. Tell that methods with named arguments enable the user to associate a method argument with a name rather than its position in the argument list.

Explain that methods with optional arguments allow the user to define a method with an optional argument with a default value. The caller of the method may or may not pass the optional argument value during the method invocation.

Tell that nullable types allow a variable to contain a value that is undefined. Tell that accessor accessibility allows the user to specify the accessibility levels of the get and set accessors. Explain that auto-implemented properties allow the user to create a property without explicitly providing the methods to get and set the value of the property.

Tell the students that in .NET Framework and C#, there is strong support for parallel programming using which you can develop efficient, fine-grained, and scalable parallel code without working directly with threads or the thread pool.

Slide 31

Understand the applications of C#.

The slide has a teal header bar with the title "Applications of C#" and decorative icons of a computer monitor, smartphone, and tablet. The main content area contains a bulleted list of applications:

- ◆ C# is an object-oriented language that can be used in a number of applications.
- ◆ Following are some of the applications:
 - ❖ Web applications
 - ❖ Web services
 - ❖ Gaming applications
 - ❖ Large-scale enterprise applications
 - ❖ Mobile applications for pocket PCs, PDAs, and cell phones
 - ❖ Simple standalone desktop applications such as Library Management Systems, Student Mark Sheet generation, and so on
 - ❖ Complex distributed applications that can spread over a number of cities or countries
 - ❖ Cloud applications

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 1", and "31".

In slide 31, tell the students that C# is an object-oriented language that can be used in a number of applications. Explain some of the applications as mentioned on the slide.

Additional Information

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

Slide 32

Understand the advantages of C#.

Advantages of C#

- ◆ C# has become a preferred programming language over C++ because of its simplicity and user-friendliness.
- ◆ The advantages of C# are as follows:

Cross Language Support:

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

Common Internet Protocols:

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

Simple Deployment:

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

XML Documentation:

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

© Aptech Ltd. Building Applications Using C# / Session 1 32

In slide 32, tell the students that C# has become a preferred programming language over C++ because of its simplicity and user-friendliness.

Explain the advantages of C# as seen on the slide.

Additional Information

For more information on other advantages of C#, refer the following link:

<http://www.techrepublic.com/article/why-should-you-add-c-to-your-skill-set/>

Slide 33

Understand memory management.

The slide has a blue header bar with the title "Memory Management". Below the header is a white content area containing a bulleted list. At the bottom of the slide is a footer bar with the copyright information "© Aptech Ltd.", the course name "Building Applications Using C# / Session 1", and the page number "33".

- ◆ In programming languages like C and C++, the allocation and de-allocation of memory is done manually.
- ◆ Performing these tasks manually is both, time-consuming and difficult.
- ◆ The C# language provides the feature of allocating and releasing memory using automatic memory management.
- ◆ This means that there is no need to write code to allocate memory when objects are created or to release memory when objects are not required in the application.
- ◆ Automatic memory management increases the code quality and enhances the performance and the productivity.

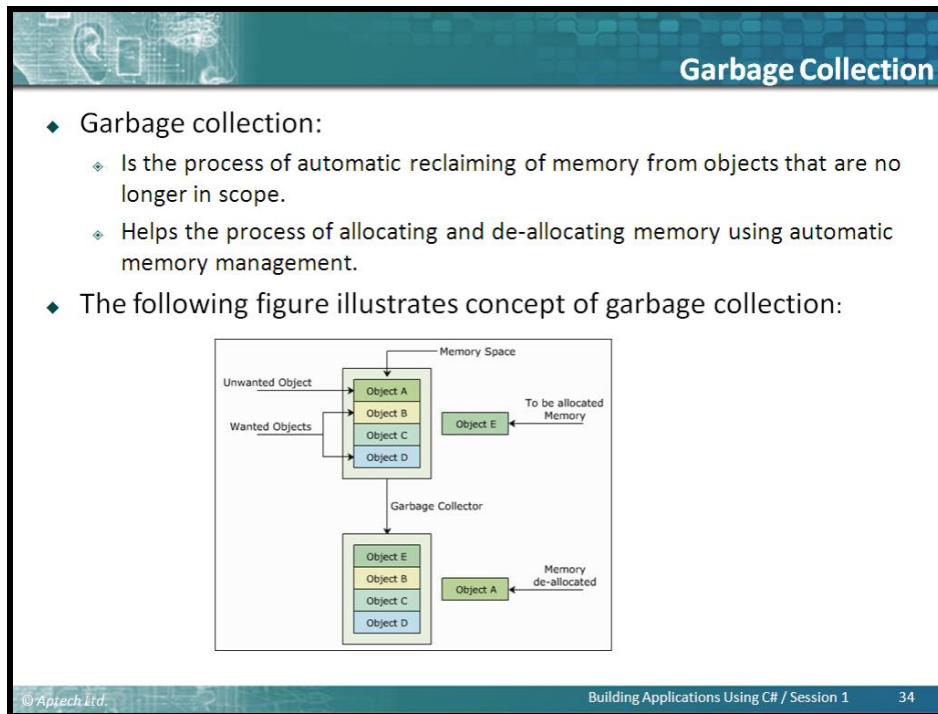
In slide 33, explain to the students that in programming languages such as C and C++, the allocation and de-allocation of memory are done manually. Performing these tasks manually is both, time-consuming and difficult.

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

Then, tell that automatic memory management increases the code quality and enhances the performance and the productivity.

Slide 34

Understand garbage collection.



The slide has a title 'Garbage Collection' at the top right. Below it, there are two main bullet points:

- ◆ Garbage collection:
 - ❖ Is the process of automatic reclaiming of memory from objects that are no longer in scope.
 - ❖ Helps the process of allocating and de-allocating memory using automatic memory management.
- ◆ The following figure illustrates concept of garbage collection:

Below the text is a diagram titled 'Memory Space' showing the process of garbage collection:

```

graph TD
    subgraph Memory_Space [Memory Space]
        direction TB
        A[Unwanted Object] --> B[Object A  
Object B  
Object C  
Object D]
        B --> C[Object E]
        C -- "To be allocated Memory" --> D[Object E  
Object B  
Object C  
Object D]
        D --> E[Object A]
        E -- "Memory de-allocated" --> F[Object A]
    end
    subgraph Garbage_Collector [Garbage Collector]
        direction TB
        B --> G[Object E  
Object B  
Object C  
Object D]
        G --> H[Object A]
        H -- "Memory de-allocated" --> I[Object A]
    end

```

The diagram shows a 'Memory Space' containing objects A through E. Object E is labeled 'To be allocated Memory'. The 'Garbage Collector' section shows object E being collected, resulting in object A being freed from memory.

At the bottom left is the copyright notice '© Aptech Ltd.' and at the bottom right are the page numbers 'Building Applications Using C# / Session 1' and '34'.

In slide 34, tell the students that the process of allocating and de-allocating memory using automatic memory management is done with the help of a garbage collector. Thus, garbage collection is the automatic reclaiming of memory from objects that are no longer in scope.

Then, tell them that this means that when the object is out of scope, the memory will be freed for use so that other objects can be allotted the memory.

You can refer to the figure in slide 34 to explain the concept of garbage collection.

Additional Information

Garbage collection process involves two steps:

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

For more information on garbage collection, refer the following link:

<http://msdn.microsoft.com/en-us/library/ee787088%28v=vs.110%29.aspx>

Slide 35

Understand Visual Studio 2012 environment.

Visual Studio 2012 Environment

- ◆ Visual Studio 2012 Environment:
 - ❖ Provides the environment to create, deploy, and run applications developed using the .NET framework.
 - ❖ Comprises the Visual Studio Integrated Development Environment (IDE), which is a comprehensive set of tools, templates, and libraries required to create .NET framework applications.
 - ❖ Is a complete set of development tools for building high performance desktop applications, XML Web Services, mobile applications, and ASP Web applications.
 - ❖ Is also used to simplify team-based design, development, and deployment of enterprise solutions.
 - ❖ Is an IDE used to ease the development process of .NET applications such as Visual C# 2012 and Visual Basic 2012.
 - ❖ Uses the same IDE, debugger, Solution Explorer, Properties tab, Toolbox, standard menus, and toolbars for all the .NET compatible languages.

© Aptech Ltd. Building Applications Using C# / Session 1 35

In slide 35, explain to the students that Visual Studio 2012 provides the environment to create, deploy, and run applications developed using the .NET Framework.

Also, tell them that the Visual Studio 2012 environment comprises the Visual Studio Integrated Development Environment (IDE), which is a comprehensive set of tools, templates, and libraries required to create .NET Framework applications.

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

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

In-Class Question:

After you finish explaining the .NET Framework, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is a debugger?

Answer:

In Visual Studio, a debugger works as a source-level debugger as well as a machine-level debugger. It also works with managed code and native code for debugging applications that are written in any language that Visual Studio supports. It attaches to running processes, monitors and debugs those processes.

Slides 36 and 37

Understand the features of IDE.

The screenshot shows a slide titled "Introduction to Visual Studio 2012 1-3". The slide content is as follows:

- ◆ The following features of IDE make it useful for an easier development process:
 - ❖ A single environment is provided for developing the .NET applications.
 - ❖ Several programming languages are available to choose from for developing applications.
 - ❖ A built-in browser is available in the IDE that is used for browsing the Internet without launching another application.
 - ❖ A program can be executed with or without a debugger.
 - ❖ The application can be published over the Internet or onto a disk.
 - ❖ The application provides Dynamic Help on a number of topics using the MSDN library.
 - ❖ The syntax of the code is checked as the user is typing it and the appropriate error notification is provided in case an error is encountered.

At the bottom of the slide, there is footer text: "© Aptech Ltd.", "Building Applications Using C# / Session 1", and "36".

Introduction to Visual Studio 2012 2-3

- ◆ The IDE:
 - ❖ Can be customized, based on the user's preferences.
 - ❖ Provides a standard code editor to write the .NET applications.
 - ❖ Has an integrated compiler to compile and execute the application. The user can either compile a single source file or the complete project.
 - ❖ Has a set of visual designers that simplifies application developments.
- ◆ Some commonly used visual designers are as follows:
 - ❖ **Windows Form Designer:** Allows programmers to design the layout of Windows forms and drag and drop controls to it.
 - ❖ **Web Designer:** Allows programmers to design the layout and visual elements of ASP.NET Web pages.
 - ❖ **WPF Designer:** Allows programmers to create user interfaces targeting WPF.
 - ❖ **Class Designer:** Allows programmers to use UML modeling to design classes, its members, and relationships between classes.
 - ❖ **Data Designer:** Allows programmer to edit database schemas graphically.

Use slide 36, to explain the students the following features of IDE. Explain that, a single environment is provided for developing the .NET applications. There are several programming languages are available to choose from for developing application. Then, tell that a built-in browser is available in the IDE that is used for browsing the Internet without launching another application.

Also tell that, a program can be executed with or without a debugger.

Use slide 37 to explain that the IDE provides a standard code editor to write the .NET applications. When a keyword is used or a dot (.) is typed after objects or enumerations, the text editor has the ability to suggest options (methods or properties) that automatically completes the required text.

Then, tell that the IDE has a set of visual designers that simplifies application developments.

Explain to the students that Windows Form Designer allows programmers to design the layout of Windows forms and drag and drop controls to it. The Web Designer allows programmers to design the layout and visual elements of ASP.NET Web pages.

Explain that WPF Designer allows programmers to create user interfaces targeting WPF.

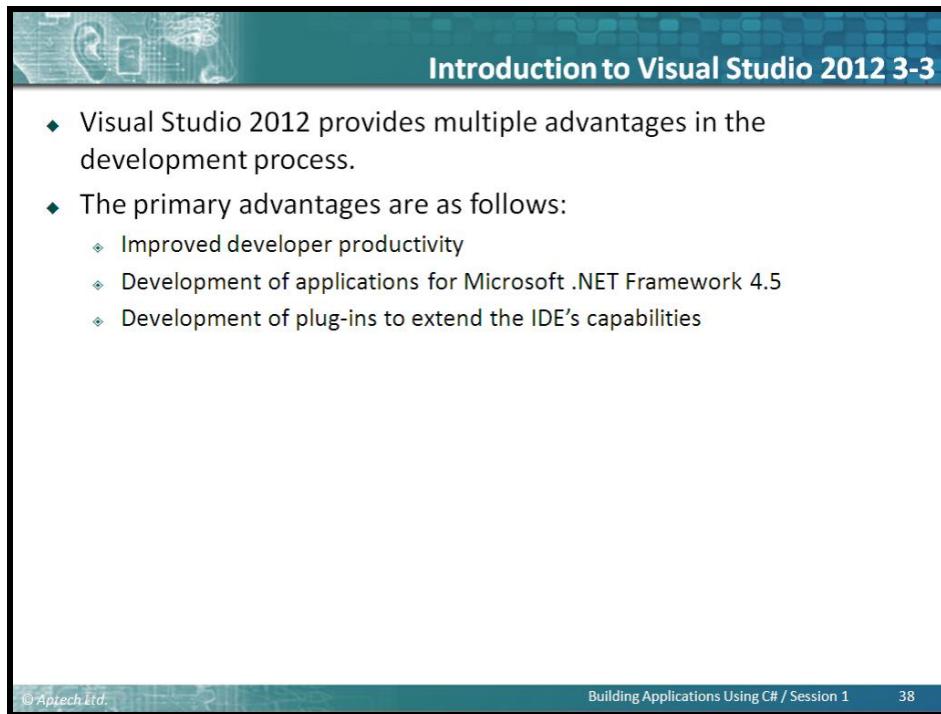
Then, tell that Class Designer allows programmers to use UML modeling to design classes, its members, and relationships between classes.

Also, mention that Data Designer allows programmer to edit database schemas graphically.

In this slide, also, ask students if they know what is meant by 'MSDN Library' and explain the same to them.

Slide 38

Understand the advantages of Visual Studio 2012.



The slide has a blue header bar with the title "Introduction to Visual Studio 2012 3-3". Below the header is a white content area containing a bulleted list of advantages. At the bottom is a dark blue footer bar with copyright and session information.

Introduction to Visual Studio 2012 3-3

- ◆ Visual Studio 2012 provides multiple advantages in the development process.
- ◆ The primary advantages are as follows:
 - ◆ Improved developer productivity
 - ◆ Development of applications for Microsoft .NET Framework 4.5
 - ◆ Development of plug-ins to extend the IDE's capabilities

© Aptech Ltd. Building Applications Using C# / Session 1 38

Use slide 38 to explain that the IDE has an integrated compiler to compile and execute the application. The user can either compile a single source file or the complete project. Visual Studio 2012 provides multiple advantages in the development process.

Tell the primary advantages,

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

Slide 39

Understand the Visual Studio 2012 editions.

Visual Studio 2012 Editions

- ◆ The IDE of Microsoft Visual Studio is a result of extensive research by the Microsoft team.
- ◆ The different editions of Visual Studio 2012 are:

Visual Studio Professional 2012	This is the entry-level edition that provides support for developing and debugging applications, such as Web, desktop, cloud-based, and mobile applications.
Visual Studio Professional 2012 with MSDN	-This edition provides all the features of the Visual Studio Professional 2012 edition along with an MSDN subscription. -In addition, this edition includes Team Foundation Server and provides access to cloud, Windows Store, and Windows Phone Marketplace.
Visual Studio Test Professional 2012 with MSDN	-This edition targets testers and Quality Assurance (QA) professionals by providing project management tools, testing tools, and virtual environment to perform application testing.
Visual Studio Premium 2012 with MSDN	-This edition provides all the features of the combined Visual Studio Professional 2012 and Visual Studio Test Professional 2012 with MSDN editions. -In addition, this edition supports peer code review, User Interface (UI) validation through automated tests, and code coverage analysis to determine the amount of code being tested.
Visual Studio Ultimate 2012 with MSDN	-This edition has all the features of the other editions. In addition, this edition supports designing architectural layer diagrams, performing Web performance and load testing, and analyzing diagnostic data collected from runtime systems.

© Aptech Ltd. Building Applications Using C# / Session 1 39

In slide 39, tell the students that the IDE of Microsoft Visual Studio is a result of extensive research by the Microsoft team.

Then, mention the different editions of Visual Studio 2012 and explain the slide.

Additional Information

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

Slide 40

Understand the languages in Visual Studio 2012.

The screenshot shows a presentation slide with a blue header bar. The title 'Languages in Visual Studio 2012' is centered in the header. Below the header is a white content area containing a bulleted list. The list includes:

- ◆ Visual Studio 2012 supports multiple programming languages such as:
 - ◆ Visual Basic .NET
 - ◆ Visual C++
 - ◆ Visual C#
 - ◆ Visual J#
- ◆ The **classes** and **libraries** used in the Visual Studio 2012 IDE are common for all the languages in Visual Studio 2012.
- ◆ It makes Visual Studio 2012 more flexible.

At the bottom of the slide, there is a footer bar with the text '©Aptech Ltd.' on the left, 'Building Applications Using C# / Session 1' in the center, and the number '40' on the right.

In slide 40, explain the students that Visual Studio 2012 supports multiple programming languages such as:

- Visual Basic.NET
- Visual C++
- Visual C#
- Visual J#

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

Slides 41 and 42

Understand the features of Visual Studio 2012.

Features of Visual Studio 2012 1-2

- ◆ The features of Visual Studio 2012 are:
 - ❖ **Comprehensive Tools Platform:**
 - In Visual Studio 2012, developers of all knowledge levels can make use of developer tools, which offer a development experience tailored for their unique needs.
 - ❖ **Reduced Development Complexity:**
 - Visual Studio 2012 enables customers to deliver more easily a broad range of .NET Framework-based solutions including Windows, Office, Web, and mobile applications.
 - ❖ **Edit Marks:**
 - Visual Studio 2012 provides a visual indication of the changes that are made and not saved and changes that are made during the current session that have been saved to the disk.
 - ❖ **Code Snippets:**
 - Code Snippets are small units of C# source code that the developer can use quickly with the help of certain keystrokes.

©Aptech Ltd. Building Applications Using C# / Session 1 41

Features of Visual Studio 2012 2-2

- ❖ **AutoRecover:**
 - Visual Studio 2012 automatically saves the work on a regular basis and thus, minimizes loss of information due to unexpected closing of unsaved files.
 - In case of an IDE crash, Visual Studio 2012 will also prompt you to recover your work after you restart.
- ❖ **IntelliSense:**
 - Visual Studio 2012 has the IntelliSense feature in which syntax tips, lists of methods, variables and classes pop up continually when entering code in the Code Editor, making the process of entering code more efficient.
- ❖ **Refactoring:**
 - Refactoring enables developers to automate common tasks when restructuring code.
 - It changes the internal structure of the code, specifically the design of its objects, to make it more comprehensible, maintainable, and efficient without changing its behavior.

©Aptech Ltd. Building Applications Using C# / Session 1 42

In slides 41 and 42, tell the students that Visual Studio 2012 provides a number of new and improved features and explain the slide.

Tell that the comprehensive tools platform helps the developers of all knowledge levels to make use of developer tools, which offer a development experience tailored for their unique needs.

Explain that the reduced development complexity feature enables customers to deliver more easily a broad range of .NET Framework-based solutions including Windows, Office, Web, and mobile applications.

Then, tell that the Edit Marks feature provides a visual indication of the changes that are made and not saved and changes that are made during the current session that have been saved to the disk.

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

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

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

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

Additional Information

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

Slide 43

Understand the various elements of Microsoft Visual Studio 2012 IDE.

The screenshot shows a slide titled "Elements of Microsoft Visual Studio 2012 IDE". The slide content is as follows:

- ◆ Visual Studio 2012 contains an extensive set of elements, comprising of editors, toolbox, and different windows to assist developers in creating .NET applications.
- ◆ The key elements in Visual Studio 2012 IDE are:
 - ❖ **Solution Explorer**
 - ❖ **Code Editor**
 - ❖ **Properties Window**
 - ❖ **Toolbox**
 - ❖ **Server Explorer**
 - ❖ **Output Window**
 - ❖ **Error List**
 - ❖ **Dynamic Help**

At the bottom of the slide, there is footer text: "©Aptech Ltd.", "Building Applications Using C# / Session 1", and "43".

Show slide 43 and tell the students that Visual Studio 2012 contains an extensive set of elements, comprising editors, toolbox, and different windows to assist developers in creating .NET applications.

Then tell them the key elements in Visual Studio 2012 IDE:

- Solution Explorer
- Code Editor
- Properties Window
- Toolbox
- Server Explorer
- Output Window
- Error List
- Dynamic Help

Slide 44

Understand the Solution Explorer in Visual Studio 2012 and its purpose.

◆ Solution Explorer:

- ◆ Provides you with an organized view of your projects and their files.
- ◆ Gives you ready access to the commands that pertain to these projects.
- ◆ Helps you to use the Visual Studio editors to work on files outside the context of a solution or project.
- ◆ The reference node consists of the assemblies referenced in the current project. Form1.cs is the name of the source file.
- ◆ When a code is written and saved, the .NET solution is saved in a .sln file, the C# project is saved in a .csproj file and the source code is saved in a .cs file.

Building Applications Using C# / Session 1 44

In slide 44, explain that the Solution Explorer provides an organized view of your projects and their files. It also gives ready access to the commands that pertain to these projects.

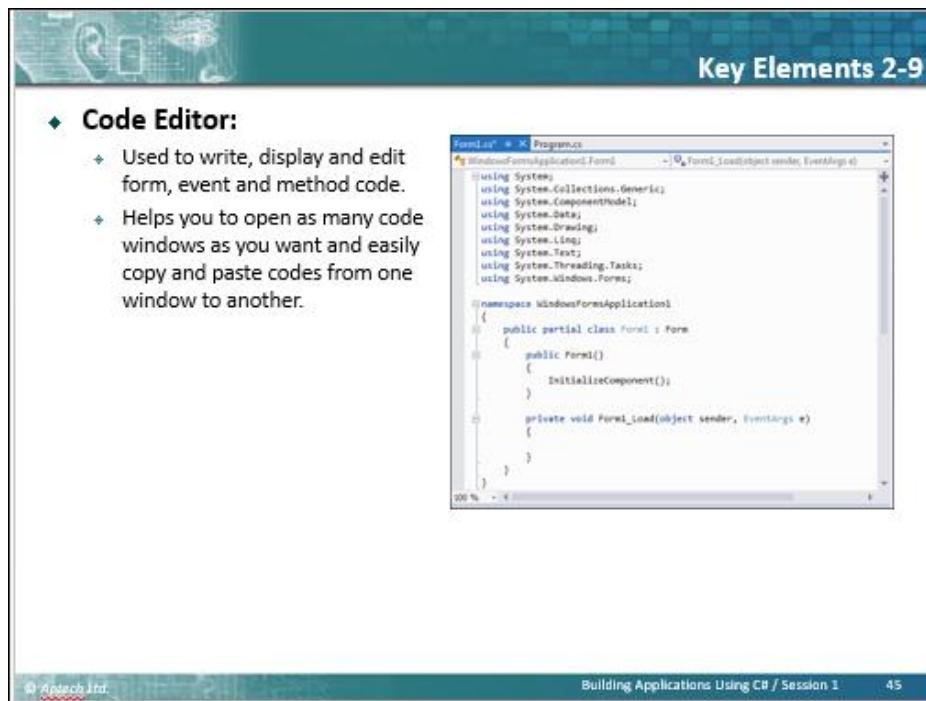
Tell them that the Solution Explorer can use the Visual Studio editors to work on files outside the context of a solution or project.

You can refer to the figure in slide 44 that displays the snapshot of Solution Explorer. Also, mention that the reference node consists of the assemblies referenced in the current project. Form1.cs is the name of the source file.

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

Slide 45

Understand the Code Editor.



In slide 45, tell the students that the code editor is one of the key components of Visual Studio 2012 as it is used to write, display, and edit code.

Tell that many code windows can be opened. Copying and pasting codes can be done easily from one window to another.

You can refer to the figure in slide 45 that displays the code editor.

Slide 46

Understand properties window and its purpose in Visual Studio 2012.

The screenshot shows the 'Properties' window in Visual Studio 2012. The title bar says 'Key Elements 3-9'. The window lists various properties for 'Form1 System.Windows.Forms.Form'. The 'Text' property is selected, showing 'Form1' in the value field. A tooltip below the 'Text' entry states: 'The text associated with the control.' Other visible properties include Localizable, Location, Locked, MainMenuStrip, MaximizeBox, MaximumSize, MinimizeBox, MinimumSize, Opacity, Padding, RightToLeft, RightToLeftLayout, ShowIcon, ShowInTaskbar, Size, SizeGripStyle, StartPosition, Tag, TopMost, TransparencyKey, UseWaitCursor, and WindowState.

In slide 46, tell the students that the properties window is used to view and change the design-time properties and events of selected objects that are located in editors and designers.

Mention that it displays different types of editing fields depending on the needs of a particular property. These fields include edit boxes, drop-down lists, and links to custom editor dialog boxes.

You can refer to the figure in slide 46 that displays the properties window.

Slides 47 and 48

Understand the Toolbox and what it is used for.

Key Elements 4-9

Toolbox:

- ◆ Displays the controls and components that can be added to the Design mode of the form.
- ◆ Displays the contents of the Toolbox window and change according to the type of form the user is creating or editing.

For example, if the user is adding tools onto a Web form, the Toolbox displays the server controls, HTML controls, data controls, and other components that the Web form may require.

Key Elements 5-9

◆ To use the controls or components from the Toolbox:

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

In slide 47, tell the students that the toolbox window displays the controls and components that can be added to the Design mode of the form. However, the contents of the Toolbox window change according to the type of form the user is creating or editing.

Give them an example for this. Tell them, if the user is adding tools onto a Web form, the Toolbox displays the server controls, HTML controls, data controls, and other components that the Web form may require.

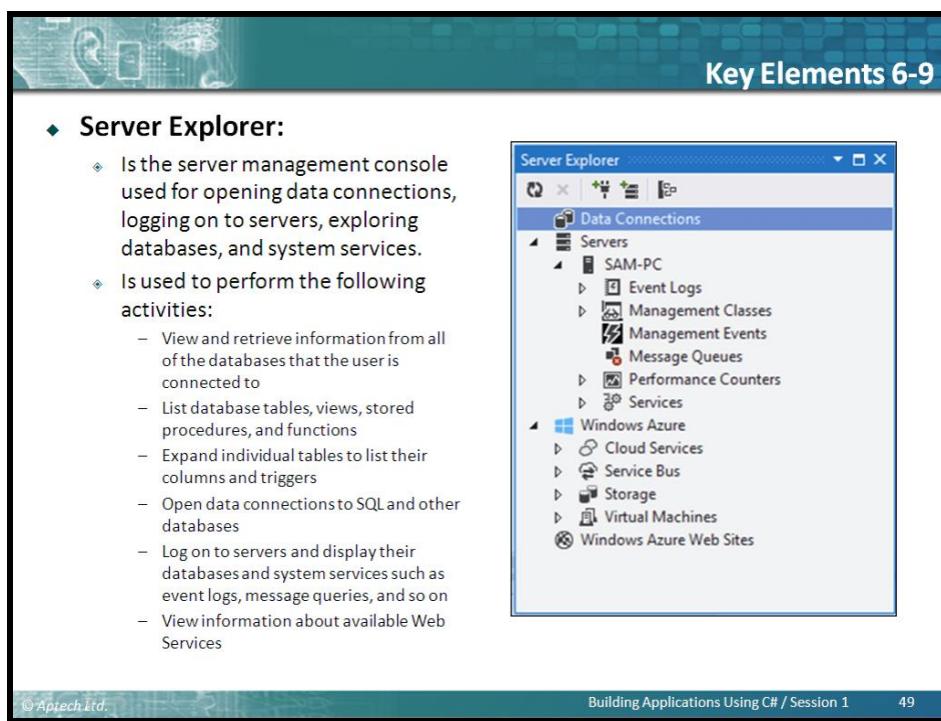
You can refer to the figure in slide 47 that displays the toolbox.

Use slide 48 to explain that to use the controls or components from the Toolbox, the user can drag and drop the required control or component onto a form.

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

Slide 49

Understand the Server Explorer and what it is used for.



The screenshot shows the Microsoft Visual Studio interface with the title bar "Key Elements 6-9". On the left, there is a callout box containing the following text:

- ◆ **Server Explorer:**
- ◆ Is the server management console used for opening data connections, logging on to servers, exploring databases, and system services.
- ◆ Is used to perform the following activities:
 - View and retrieve information from all of the databases that the user is connected to
 - List database tables, views, stored procedures, and functions
 - Expand individual tables to list their columns and triggers
 - Open data connections to SQL and other databases
 - Log on to servers and display their databases and system services such as event logs, message queries, and so on
 - View information about available Web Services

On the right, the "Server Explorer" window is open, showing the following tree structure:

- Server Explorer
- Data Connections
- Servers
 - SAM-PC
 - Event Logs
 - Management Classes
 - Management Events
 - Message Queues
 - Performance Counters
 - Services
 - Windows Azure
 - Cloud Services
 - Service Bus
 - Storage
 - Virtual Machines
 - Windows Azure Web Sites

Use slide 49 to explain the students that the server explorer is the server management console used for opening data connections, logging on to servers, exploring databases, and system services.

Also, mention that using the Server Explorer, the user can perform the following activities:

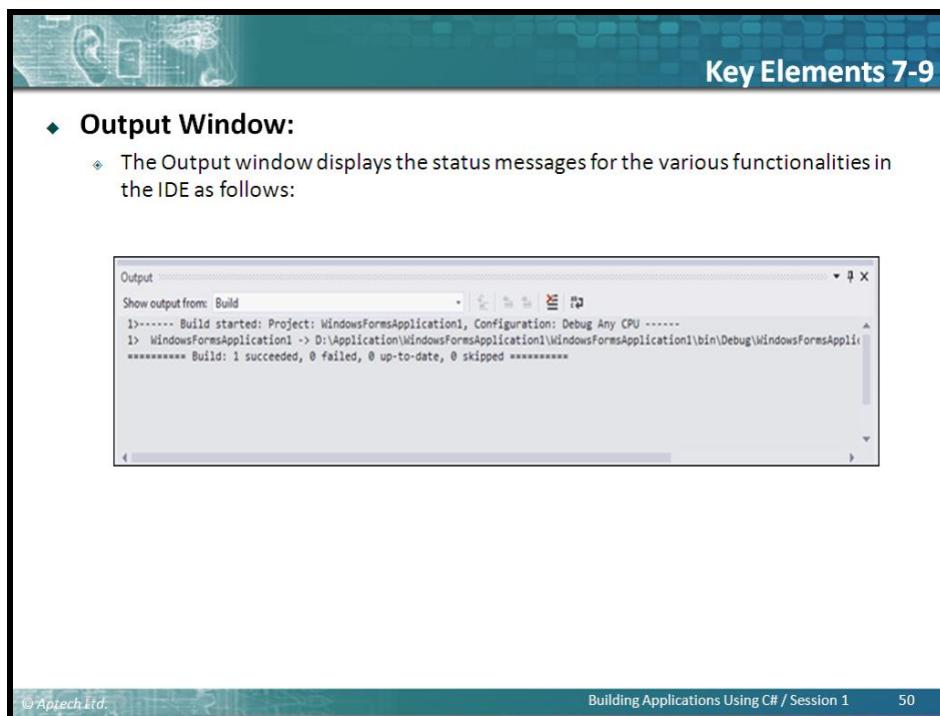
- View and retrieve information from all of the databases that the user is connected to
- List database tables, views, stored procedures, and functions
- Expand individual tables to list their columns and triggers

- Open data connections to SQL and other databases
- Log on to servers and display their databases and system services such as event logs, message queries, and so on
- View information about available Web Services

You can refer to the figure in slide 49 that displays the Server Explorer.

Slide 50

Understand the Output Window.



In slide 50, tell the students that the output window displays the status messages for the various functionalities in the IDE.

You can refer to the figure in slide 50 that displays the output window.

Slide 51

Understand the Error List window.

Key Elements 8-9

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

Error List

Description		File	Line	Column	Project
1	: expected	Form1.cs	17	34	WindowsFormsApplication1

©Aptech Ltd. Building Applications Using C# / Session 1 51

In slide 51, explain the students that with the help of the error list, debugging and troubleshooting the application becomes easier and thus, the speed of application development increases.

Explain the Error List window using following points:

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

You can refer to the figure in slide 51 that displays the error list window.

Slide 52

Understand Dynamic Help and its purpose.

The screenshot shows a slide titled 'Key Elements 9-9'. The main content area contains a bullet point under a heading '◆ Dynamic Help:' followed by a descriptive text. The bottom navigation bar includes icons for back, forward, and search, along with the text 'Building Applications Using C# / Session 1' and the number '52'.

◆ **Dynamic Help:**

- ◆ Provides a list of topics specific to the area of the IDE you are working in or the task you are working on.

Building Applications Using C# / Session 1 52

In slide 52, explain to the students that the Dynamic Help window provides a list of topics specific to the area of the IDE.

Slides 53 to 56

Understand `csc` command and its usage.

csc Command 1-4

- ◆ Console applications that are created in C# run in a console window. This window provides simple text-based output.
- ◆ The `csc` (C Sharp Compiler) command can be used to compile a C# program.
- ◆ Following are the steps to compile and execute a program:
 1. **Create a New Project.**
 2. **Compile a C# Program.**
 3. **Execute the Program.**

© Aptech Ltd. Building Applications Using C# / Session 1 53

csc Command 2-4

- ◆ **Create a New Project:**
 - ◆ Following are the steps to create a new project:
 - Start **Visual Studio 2012**.
 - Select **New → Project** from the **File** menu.
 - Expand the **Templates → Visual C#** nodes in the left pane and select **Console Application** in the right pane of the New Project dialog box.
 - Specify the name and location for the project and click **OK**. Visual Studio 2012 opens the Code Editor with the skeleton code of a class, as shown in the following code:

Snippet

```

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

```

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


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

© Aptech Ltd. Building Applications Using C# / Session 1 54

csc Command 3-4

- ◆ **Compile a C# Program:**
 - ❖ A C# program can be compiled using the following syntax:
`csc <file.cs>`

Example

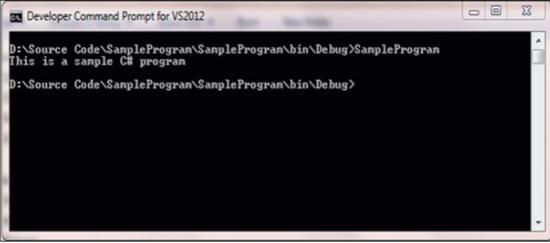
```
csc SampleProgram.cs
```

- ❖ In the example:
 - `SampleProgram`: Specifies the name of the program to be compiled.
 - This command generates an executable file `SampleProgram.exe`.

© Aptech Ltd. Building Applications Using C# / Session 1 55

csc Command 4-4

- ◆ **Execute the Program:**
 - ❖ Open the Developer Command Prompt for VS2012, and browse to the directory that contains the .exe file.
 - ❖ Type the file name at the command prompt.
 - ❖ The following figure shows the developer command prompt for VS2012 window:



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

© Aptech Ltd. Building Applications Using C# / Session 1 56

In slide 53, explain to the students that the console applications that are created in C# run in a console window. This window provides simple text-based output.

The **csc (C Sharp Compiler)** command can be used to compile a C# program.

Tell them the steps to compile and execute a program are as follows:

- Create a New Project.
- Compile a C# Program.
- Execute the Program.

In slide 54, explain to the students the steps to create a new project. Make use of the code to explain them the structure of a simple C# program. A template is a ready-made built-in skeleton structure for a program provided by an IDE. In Visual Studio 2012, you have both project as well as item templates that act as reusable skeleton structures for users with basic code that can be changed as per requirement.

In slide 55, explain to the students that a C# program can be compiled using the syntax as shown in the slide. The example command shown on the slide generates an executable file SampleProgram.exe. The .exe file is known as portable EXE as it contains machine-independent instructions. The portable EXE works on any operating system that supports the .NET platform.

Use slide 56 and tell the students that to execute the program, open the Developer Command Prompt for VS2012, and browse to the directory that contains the .exe file. Then, type the file name at the command prompt.

You can refer to the figure in slide 56 that displays the developer command prompt for VS2012 window.

Additional Information

For more information on the `csc` command, refer the following link:

<http://msdn.microsoft.com/en-us/library/78f4aasd.aspx>

Slide 57

Understand the process to build and execute a C# program.

The slide is titled "Build and Execute". It contains the following text and a screenshot:

- ◆ The IDE also provides the necessary support to compile and execute C# programs.
- ◆ Following are the steps to compile and execute C# programs:
 - ❖ **Compiling the C# Program**
 - Select Build <application name> from the Build menu. This action will create an executable file (.exe).
 - ❖ **Executing the Program:**
 - Select Start Without Debugging from the Debug menu.

Screenshot Description: A screenshot of a Windows Command Prompt window titled "cmd C:\Windows\system32\cmd.exe". The window displays the text: "This is a sample C# program" and "Press any key to continue . . ." at the bottom.

At the bottom of the slide, there is footer text: "©Aptech Ltd." and "Building Applications Using C# / Session 1 57".

In slide 57, tell the students that in Visual Studio 2012, apart from the use of the `csc` command, the Integrated Development Environment provides the necessary support to compile and execute C# programs.

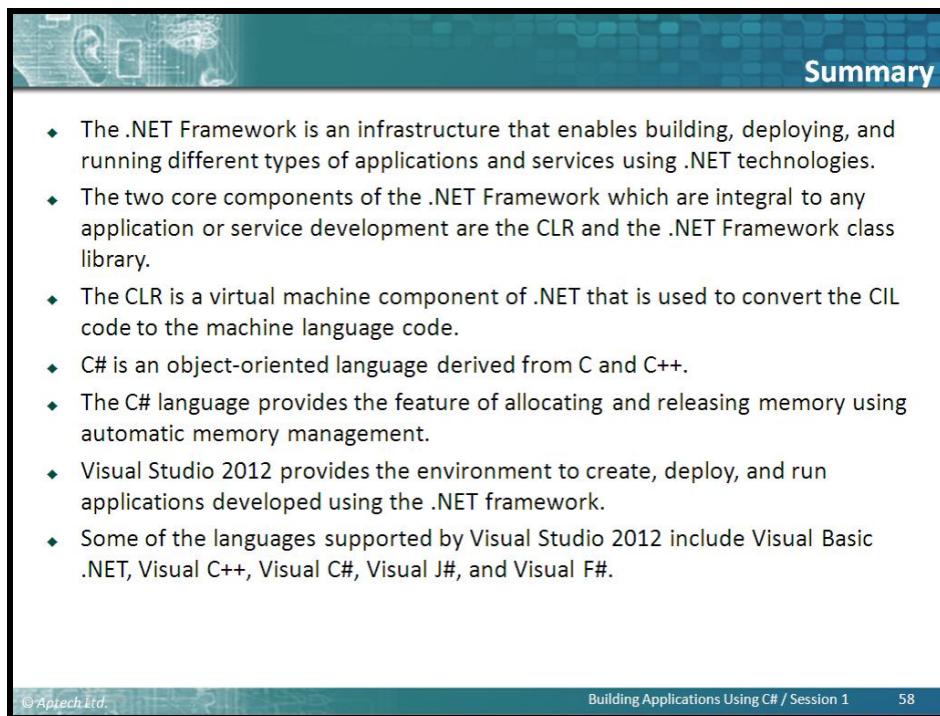
Explain the steps:

- **Compiling the C# Program:** Select **Build <application name>** from the **Build** menu. This action will create an executable file (.exe).
- **Executing the Program:** From the **Debug** menu, select **Start Without Debugging**.

You can refer to the figure in slide 57 that shows the output of the program.

Slide 58

Summarize the session.



The slide features a blue header bar with the word "Summary" on the right. Below the header is a white content area containing a bulleted list of points about .NET Framework components and Visual Studio 2012. At the bottom of the slide is a dark footer bar with the text "©Aptech Ltd.", "Building Applications Using C# / Session 1", and the number "58".

◆ The .NET Framework is an infrastructure that enables building, deploying, and running different types of applications and services using .NET technologies.

◆ The two core components of the .NET Framework which are integral to any application or service development are the CLR and the .NET Framework class library.

◆ The CLR is a virtual machine component of .NET that is used to convert the CIL code to the machine language code.

◆ C# is an object-oriented language derived from C and C++.

◆ The C# language provides the feature of allocating and releasing memory using automatic memory management.

◆ Visual Studio 2012 provides the environment to create, deploy, and run applications developed using the .NET framework.

◆ Some of the languages supported by Visual Studio 2012 include Visual Basic .NET, Visual C++, Visual C#, Visual J#, and Visual F#.

In slide 58, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

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

- Some of the languages supported by Visual Studio 2012 include Visual Basic.NET, Visual C++, Visual C#, Visual J#, and Visual F#.

1.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. What is CLR?
2. Explain Framework Class Library.
3. How does .NET Framework differ from traditional compilers?

Session 2 - Variables and Data Types

2.1 Pre-Class Activities

Before you commence the session, you should revisit topics of the previous session for a brief review before the session begins. The summary of the previous session is as follows:

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

Here, you can ask students the key topics they can recall from previous session. Prepare a question or two which will be a key point to relate the current session objectives. Ask them to explain the .NET Framework in C# and explain the C# language features. You can also ask the students to describe the Visual Studio 2012 environment and the elements of Microsoft Visual Studio 2012 IDE.

2.1.1 Objectives

By the end of this session, the learners will be able to:

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

2.1.2 Teaching Skills

To teach this session successfully, you must know about variables and data types in C#. You should be aware of the constants and literals. You should also be familiar with input and output.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to the variables and data types in C#. They will learn about the comments and XML documentation, constants and literals, keywords and escape sequences, and input and output.

2.2 In-Class Explanations

Slide 3

Understand variables and data types in C#.

Variables and Data Types in C#

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

`int a;`

Data Type Variable

© Aptech Ltd. Building Applications Using C# /Session 2 3

Use slide 3 to explain that a variable is used to store data in a program and is declared with an associated data type.

Tell them that the variable has a name and may contain a value and a data type defines the type of data that can be stored in a variable.

For introducing the concept of variables and data types, you can use the following explanation:

An application may handle more than one piece of data. In such a case, the application has to assign memory for each piece of data. While assigning memory, consider the following two factors:

- How much memory is to be assigned?
- Remembering at what memory address, data is stored.

Earlier, programmers had to write their programs in the language of the machine that is, in 1's and 0's. If the programmer wanted to temporarily store a value, the exact location where the data is stored inside the memory of the computer had to be assigned. This storage location had to be a specific number or memory address.

Modern day languages enable us to use symbolic names known as variables, to refer to the memory location where a particular value is to be stored.

The data type decides the amount of memory to be assigned. The names that we assign to variables help us in reusing the data as and when required.

Slide 4

Understand the definition of a variable.

The slide has a blue header bar with the word 'Definition' on the right. Below the header, there is a bulleted list of points. A blue box labeled 'Example' contains a sub-list of entities that can be represented by variables. At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# /Session 2' in the center, and the number '4' on the right.

- ◆ A variable is an entity whose value can keep changing during the course of a program.
 - ◆ 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#, 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.
- ◆ Different types of data such as a character, an integer, or a string can be stored in variables. Based on the type of data that needs to be stored in a variable, variables can be assigned different data types.

Use slide 4 to explain that a variable is an entity whose value can keep changing during the course of a program. Give an example. Tell them that 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.

Mention that in C#, 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.

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

Slides 5 and 6

Understand the usage of variables.

Using Variables 1-2

- In C#, memory is allocated to a variable at the time of its creation and a variable is given a name that uniquely identifies the variable within its scope.
- On initializing a variable, the value of a variable can be changed as required to keep track of data being used in a program. When referring to a variable, you are actually referring to the value stored in that variable.
- The following figure illustrates the concept of a variable:

- The following syntax is used to declare variables in C#:

Syntax

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

where,

- datatype: Is a valid data type in C#.
- variableName: Is a valid variable name.

© Aptech Ltd. Building Applications Using C# /Session 2 5

Using Variables 2-2

- The following syntax is used to initialize variables in C#:

Syntax

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

- where,
 - =: Is the assignment operator used to assign values.
 - value: Is the data that is stored in the variable.
- The following code declares two variables, namely, **empNumber** and **empName**:

Snippet

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

- In the code:
 - an integer variable declares **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;**

© Aptech Ltd. Building Applications Using C# /Session 2 6

In slide 5, tell the students that in C#, memory is allocated to a variable at the time of its creation and a variable is given a name that uniquely identifies the variable within its scope.

Explain them that on initializing a variable, the value of a variable can be changed as required to keep track of data being used in a program. When referring to a variable, you are actually referring to the value stored in that variable.

You can refer to the figure in slide 5 that illustrates the concept of a variable.

Then, explain the syntax used to declare variables in C# as given on the slide.

In slide 6, tell the students that the syntax is used to initialize variables in C#, where, the = is the assignment operator used to assign values and value is the data that is stored in the variable.

Then, tell the students that the code declares two variables, namely, empNumber and empName.

Tell them that in the code, an integer variable declares empNumber, and a string variable, empName. Memory is allocated to hold data in each variable in the code. Values can be assigned to variables by using the assignment operator (=), as empNumber = 100; and empName = "David Blake";

Mention that the students can also assign a value to a variable upon creation, as shown in following statement:

```
int empNumber = 100;
```

Additional Information

For more information on variables, refer the following link:

<http://msdn.microsoft.com/en-us/library/ms173104.aspx>

Slide 7

Understand data types.

Data Types

- Different types of values such as numbers, characters, or strings can be stored in different variables. To identify the type of data that can be stored in a variable, C# provides different data types.
- When a variable is declared, a data type is assigned to the variable. This allows the variable to store values of the assigned data type.
- In C# programming language, data types are divided into two categories:

Value Types

- Variables of value types store actual values that are stored in a stack that results in faster memory allocation to variables of value types.
- Most of the built-in data types are value types.
- The value type built-in data types are `int`, `float`, `double`, `char`, and `bool`. User-defined value types are created using the `struct` and `enum` keywords.

Reference Types

- Variables of reference type store the memory address of other variables in a heap.
- These values can either belong to a built-in data type or a user-defined data type.

© Aptech Ltd. Building Applications Using C# /Session 2 7

In slide 7, explain that different types of values such as numbers, characters, or strings can be stored in different variables. To identify the type of data that can be stored in a variable, C# provides different data types.

Mention that 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.

Then, tell them that data types are divided into two categories such as value types and reference types in C# programming language. The runtime deals with the two in different ways.

Explain that variables of value types contain actual values which are stored in a stack. This results in faster memory allocation to variables of value types.

Tell that most of the built-in data types are value types. The value type built-in data types are `int`, `float`, `double`, `char`, and `bool`. User-defined value types are created using the `struct` and `enum` keywords.

When a variable of value-type is created, a single space in memory is allocated to store the value. When the runtime deals with a value type, it is dealing directly with its underlying data and this leads to more efficiency. Tell that variables of reference types store the memory address of other variables (like a pointer) in a heap. These values can either belong to a built-in data type or a user-defined data type.

You can refer to the figure in slide 7 that displays the data types. To understand the difference between value types and reference types in a better manner, let's take an example.

Consider the following two variables:

```
AReferenceType aa;
AValueType bb;
```

Here, the value of `aa` is a reference, hence, it will either be null or a reference to an object which is itself an instance of `AReferenceType` or a derived class. The value of `aa` is not the object. However, the value of `bb` is the data itself.

Additional Information

For more information on types, refer the following link:

<http://msdn.microsoft.com/en-us/library/ms173104.aspx>

Slide 8

Understand the pre-defined data types.

The slide has a title 'Pre-defined Data Types' at the top right. Below the title is a bulleted list of three points:

- The pre-defined data types are referred to as basic data types in C# that 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.
- The following table 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	+1.5e-45 to +3.4e38
double	64-bit floating point with 15-16 digits precision	+5.0e-324 to +1.7e308
decimal	128-bit floating point with 28-29 digits precision	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$
char	Unicode 16-bit character	U+0000 to U+ffff
bool	Stores either true or false	true or false

At the bottom left is a copyright notice '© Aptech Ltd.' and at the bottom right is the page number '8'.

Use slide 8 to tell the students that the pre-defined data types are referred to as basic data types in C# that has a pre-defined range and size.

Mention that 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. You can refer to table in slide 8 and explain that summarizes the pre-defined data types in C#.

Arrays are of reference types even if the elements they contain may be of value types. 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 can represent both positive and negative numbers. 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.

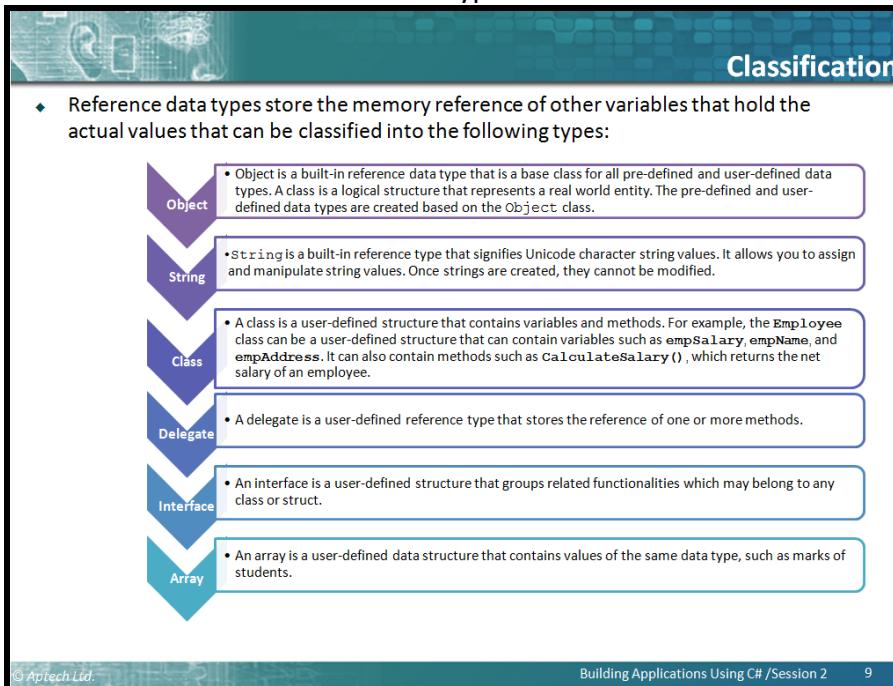
Additional Information

For more information on built-in types, refer the following link:

<http://msdn.microsoft.com/en-us/library/1dhd7f2x.aspx>

Slide 9

Understand the classification of reference data types.



In slide 9, explain that reference data types store the memory reference of other variables that hold the actual values that can be classified into the types such as Object, String, Class, Delegate, Interface, and Array.

Explain that in Object data type, an object is a built-in reference data type that 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. The pre-defined and user-defined data types are created based on the Object class. Mention that in String data type, String is a built-in reference type that signifies Unicode character string values. It allows you to assign and manipulate string values. Once strings are created, they cannot be modified.

Explain that in Class data type, a class is a user-defined structure that contains variables and methods. For example, the Employee class can be a user-defined structure that can contain variables such as empSalary, empName, and empAddress. It can also contain methods such as CalculateSalary(), which returns the net salary of an employee.

Then, explain to the students that in Delegate data type, a delegate is a user-defined reference type that stores the reference of one or more methods.

Explain that in Interface data type, an interface is a user-defined structure that groups related functionalities which may belong to any class or struct.

Also, mention that in Array data type, an array is a user-defined data structure that contains values of the same data type, such as marks of students.

Slide 10

Understand the rules for variables.

Rules

- ◆ A variable needs to be declared before it can be referenced by following certain rules as follows:

A variable name can begin with an uppercase or a lowercase letter. The name can contain letters, digits, and the underscore character (_).

The first character of the variable name must be a letter and not a digit. The underscore is also a legal first character, but it is not recommended at the beginning of a name.

C# is a case-sensitive language; hence, variable names count and Count refer to two different variables.

C# keywords cannot be used as variable names. If you still need to use a C# keyword, prefix it with the '@' symbol.

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

© Aptech Ltd. Building Applications Using C# /Session 2 10

Use slide 10 to explain that a variable needs to be declared before it can be referenced by certain rules.

Explain to the students that a variable name can begin with an uppercase or a lowercase letter. The name can contain letters, digits, and the underscore character (_).

Then, tell the students that the first character of the variable name must be a letter and not a digit. The underscore is also a legal first character, but it is not recommended at the beginning of a name.

Also, tell that C# is a case-sensitive language; hence, variable names count and Count refer to two different variables. Mention that C# keywords cannot be used as variable names. If you still need to use a C# keyword, prefix it with the '@' symbol. Mention that it is always advisable to give meaningful names to variables such that the name gives an idea about the content that is stored in the variable.

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

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

Slide 11

Understand the validity of the variables.

Validity

- ◆ Mentioning a variable's type and identifier (name) at the time of declaring a variable indicates to the compiler, the name of the variable and the type of data that will be stored.
- ◆ An undeclared variable generates an error message.
- ◆ The following table displays a list of valid and invalid variable names in C#:

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

© Aptech Ltd. Building Applications Using C# /Session 2 11

Use slide 11 to tell the students that mentioning a variable's type and identifier (name) at the time of declaring a variable indicates to the compiler, the name of the variable and the type of data that will be stored. An undeclared variable generates an error message.

You can refer and explain table in slide 11 that displays a list of valid and invalid variable names in C#.

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

Slides 12 to 14

Understand the declaration of the variables.



Declaration 1-3

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

Syntax

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

where,

- ◆ data type: Is a valid variable type.
- ◆ variable name: Is a valid variable name or identifier.
- ◆ value: Is the value assigned to the variable.

© Aptech Ltd.

Building Applications Using C# /Session 2 12



Declaration 2-3

- ◆ The following is the syntax to declare multiple variables:

Syntax

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

where,

- ◆ data type: Is a valid variable type.
- ◆ variable name1, variable name2, variable nameN: Are valid variable names or identifiers.

- ◆ The following is the syntax to declare and initialize multiple variables:

Syntax

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

© Aptech Ltd.

Building Applications Using C# /Session 2 13

Declaration 3-3

- The following code demonstrates how to declare and initialize variables in C#:

Snippet

```
bool boolTest = true;
short byteTest = 19;
int intTest;
string stringTest = "David";
float floatTest;
int Test = 140000;
floatTest = 14.5f;
Console.WriteLine("boolTest = {0}", boolTest);
Console.WriteLine("byteTest = " + byteTest);
Console.WriteLine("intTest = " + intTest);
Console.WriteLine("stringTest = " + stringTest);
Console.WriteLine("floatTest = " + floatTest);
```

- In the code:
 - Variables of type `bool`, `byte`, `int`, `string`, and `float` are declared. Values are assigned to each of these variables and are displayed using the `WriteLine()` method of the `Console` class.
- The code displays the following output:

Output

```
PS C:\WINDOWS\system32\cmd.exe
boolTest = True
byteTest = 19
intTest = 140000
stringTest = David
floatTest = 14.5
Press any key to continue . . .
```

Use slide 12 to explain that in C#, multiple variables can be declared at the same time in the same way a single variable is declared. After declaring variables, assign values to them.

Assigning a value to a variable is called initialization. A value can be assigned to a variable while declaring it or at a later time.

Tell the students that the syntax declares and initializes a single variable. In the syntax, `data type` is a valid variable type, `variable name` is a valid variable name or identifier, and `value` is the value assigned to the variable. In slide 13, tell the students that instead of declaring every variable of a type in individual statements, one can combine multiple variables belonging to the same type in a single statement. This improves readability and also decreases the lines of code in a program.

Tell the students that the syntax shown on the slide 13 declares multiple variables using a single statement. In the given syntax, `data type` is a valid variable type and `<variable name1>`, `<variable name2>`, and `<variable nameN>` are syntax of valid variable names or identifiers.

You cannot combine multiple variables of various data types in a single statement. Thus, declaring the following statement

```
int x, y, float z;
```

will result in a compiler error as the statement is invalid.

Use slide 14 to tell the students that the code demonstrates how to declare and initialize variables in C#.

Explain the code as follows:

In the code, variables of type `bool`, `byte`, `int`, and `string` are declared and initialized respectively. The `float` variable is declared and then assigned a value in a separate statement.

Values of the variables are displayed using the `WriteLine()` method of the `Console` class.

You can refer to the figure in slide 14 that displays the output.

You may also discuss additional examples such as:

```
char firstLetter = 'M';
var price = 350;
```

Slide 15

Understand implicitly typed variables.

Iimplicitly Typed Variables

- When you declare and initialize a variable in a single step, you can use the `var` keyword in place of the type declaration. Variables declared using the `var` keyword are called implicitly typed variables. For implicitly typed variables, the compiler infers the type of the variable from the initialization expression. The following code demonstrates how to declare and initialize implicitly typed variables in C#:

Snippet

```
var boolTest = true;
var byteTest = 1;
var intTest = 140000;
var stringTest = "David";
var floatTest = 14.5f;
Console.WriteLine("boolTest = {0}", boolTest);
Console.WriteLine("byteTest = {1}", byteTest);
Console.WriteLine("intTest = {2}", intTest);
Console.WriteLine("stringTest = {3}", stringTest);
Console.WriteLine("floatTest = {4}", floatTest);
```

- In the code, 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.

Output

```
C:\WINDOWS\system32\cmd.exe
boolTest = True
byteTest = 19
intTest = 140000
stringTest = David
floatTest = 14.5
Press any key to continue . . .
```

© Aptech Ltd. Building Applications Using C# /Session 2 15

In slide 15, tell the students that to declare and initialize a variable in a single step, use the `var` keyword in place of the type declaration. Variables declared using the `var` keyword are called

implicitly typed variables. For implicitly typed variables, the compiler infers the type of the variable from the initialization expression.

Explain to the students that the code demonstrates how to declare and initialize implicitly typed variables in C#. Then, explain the code. Tell the students that in the code, 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. You can refer to the figure in slide 15 that displays the output.

Additional Information

- 1) The ‘implicitly typed variables’ feature was first introduced in C# 3.0 with .NET Framework 3.5 and Visual studio 2008.
- 2) You must declare and initialize implicitly typed variables at the same time. Not doing so will result in the compiler reporting an error.

For more information on implicitly typed variables, refer the following link:

<http://msdn.microsoft.com/en-us/library/bb384061.aspx>

Slides 16 to 18

Understand the purpose and usage of comments in C# programs.



Comments

- ◆ Comments help in reading the code of a program to understand the functionality of the program.
- ◆ In C#, comments are given by the programmer to:
 - Provide information about a piece of code.
 - Make the program more readable.
 - Explain the purpose of using a particular variable or method to a programmer.
 - Help to identify comments as they are marked with special characters.
- ◆ Comments are ignored by the compiler, during the execution of the program.

© Aptech Ltd.

Building Applications Using C# /Session 2 16



Types of Comments 1-2

- ◆ **Single-line Comments:** Begin with two forward slashes (//).

Snippet

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

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


```
// This block of code will add two numbers and then put the result in the
// variable, doSum
int doSum = 4 + 3;
```
 - ◆ You can also write the single-line comment in the same line as shown in the following code:


```
int doSum = 4 + 3; // Adding two numbers
```
- ◆ **Multi-line Comments:** Begin with a forward slash followed by an asterisk /*) and end with an asterisk followed by a forward slash (*/).

Snippet

```
/* 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)
```

© Aptech Ltd.

Building Applications Using C# /Session 2 17

Types of Comments 2-2

- ◆ **XML Comments:** Begin with three forward slashes (///). Unlike single-line and multi-line comments, the XML comment must be enclosed in an XML tag. You need to create XML tags to insert XML comments. Both the XML tags and XML comments must be prefixed with three forward slashes.
 - ❖ You can insert an XML comment, as shown in the following code:

Snippet

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

- ❖ The following figure displays a complete example of using XML comments:

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

    ///<summary>
    //Main method begins here

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

© Aptech Ltd.

Building Applications Using C# /Session 2 18

Use slide 16 to tell the students that comments help in reading the code of a program to understand the functionality of the program.

Explain that in C#, comments are given by the programmer to provide information about a piece of code, make the program more readable, explain the purpose of using a particular variable or method to a programmer, and help to identify comments as they are marked with special characters.

Mention that the comments are ignored by the compiler, during the execution of the program. In slide 17, tell the students that there are three types of comments: Single-line Comments, Multi-line Comments, and XML Comments.

Explain to the students that the single-line comments begin with two forward slashes (//). Then, tell the students that Multi-line Comments begin with a forward slash followed by an asterisk /*) and end with an asterisk followed by a forward slash (*/). In slide 18, explain the XML comments.

Explain to the students that the XML comments begin with three forward slashes (///). Unlike single-line and multi-line comments, the XML comment must be enclosed in an XML tag. They need to create XML tags to insert XML comments. Both the XML tags and XML comments must be prefixed with three forward slashes. You can refer to the figure in slide 18 that displays a complete example of using XML comments.

When you press the Enter key, after the opening characters /* in a multi-line comment, an '*' character appears at the beginning of the new line. This character indicates that a new line is inserted in the comment. When the multi-line comment is closed, Visual Studio 2012 will

change all text contained in the comment as **BOLD** text. This will highlight the text that is written as a comment. In C#, comments are also known as remarks.

Slides 19 and 20

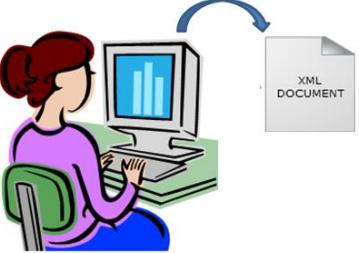
Understand XML documentation.

XML Documentation 1-2

- ◆ 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.

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.



© Aptech Ltd. Building Applications Using C# /Session 2 19

XML Documentation 2-2

- The following code displays the XML comments that can be extracted to an XML file:

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

- The following syntax is used to create an XML document from the C# source file Visual Studio 2012 Command Prompt window:

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.

© Aptech Ltd. Building Applications Using C# /Session 2 20

Use slide 19 to tell the students that in C#, an XML document can be created that will contain all the XML comments. This document is useful when multiple programmers want to view information of the program. Then, give them an example.

Tell them to 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 2012 Command Prompt window. You can refer to the figure in slide 20 that displays the XML comments that can be extracted to an XML file.

In slide 20, tell the students that the given syntax creates an XML document from the C# source file where, **XMLfilename.xml** is the name of the XML file that is to be created and **CSharpfilename.cs** is the name of the C# file from where the XML comments will be extracted.

In the Visual Studio 2012 IDE, when you type in three slashes: **///**, the IDE automatically fills in this snippet,

```
/// <summary>
///
/// </summary>
```

This is the skeleton structure for an XML comment. The XML comment for a class consists of one block. When the IDE adds the skeleton for a constructor or another method, it adds tags for each of the parameters.

When you add an XML comment for a method, the IDE automatically adds a `<param>` tag for each parameter, and a `<returns>` tag if it has a return value. When you use the class later in a program, the IDE automatically displays the XML comment for the class along with its IntelliSense window. As soon as you type (for the constructor), the IDE shows you the parameters for the constructor. This is because of the XML comments and documentation.

Slides 21 and 22

Understand pre-defined XML tags.

The slide has a blue header bar with the title "Pre-defined XML Tags 1-6". Below the title is a bulleted list: "XML comments are inserted in XML tags that can either be pre-defined or user-defined. The following table lists the widely used pre-defined XML tags and states their conventional use:". A table follows, with columns "Pre-defined Tags" and "Descriptions".

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

At the bottom left is the copyright notice "© Aptech Ltd." and at the bottom right is the page number "Building Applications Using C# /Session 2 21".

The following figure displays an example of pre-defined XML tags:

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

In slide 21, tell the students that the XML comments are inserted in XML tags that can either be pre-defined or user-defined. You can refer slide 21 and explain table that lists widely used pre-defined XML tags and states their conventional use. Use slide 22 to refer to the figure that displays an example of pre-defined XML tags.

Slides 23 to 26

Understand the code that demonstrates the use of XML comments.

Pre-defined XML Tags 3-6

- The following code demonstrates the use of XML comments:

Snippet

```
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 href="Employee(int,string)">Employee(int,string)</seealso>
    /// </remarks>
}
```

Pre-defined XML Tags 4-6

```
private string _name;
/// <summary>
/// This constructor initializes the id and name to -1 and null.
/// </summary>
/// <remarks>
/// <seealso href="Employee(int,string)">Employee(int,string)</seealso>
/// </remarks>
public Employee()
{
    _id = -1;
    _name = null;
}
/// <summary>
/// This constructor initializes the id and name to
/// (<paramref name="id"/>,<paramref name="name"/>).
/// </summary>
/// <param name="id">Employee ID</param>
/// <param name="name">Employee Name</param>
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); }
```

Pre-defined XML Tags 5-6

- ◆ The following figure displays the XML document:

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

Pre-defined XML Tags 6-6

- ◆ In the code:

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

In slides 23 and 24, explain the code and tell the students that it demonstrates the use of XML comments.

In slide 25, you can show the outcome of the code and say that this resultant XML document is achieved through the code.

Use slide 26 to explain what was done in the code. Tell the students that in the code, the <remarks>, <seealso>, and <paramref> XML documentation tags are used. The <remarks> tag is used to provide information about a specific class. The <seealso> tag is used to specify the text that should appear in the See Also section. The <paramref> tag is used to indicate that the specified word is a parameter.

Additional Information

For more information on XML comments, refer the following link:
<http://msdn.microsoft.com/en-us/library/b2s063f7.aspx>

In-Class Question:

After you finish explaining pre-defined XML tags, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are the types of comments in C#?

Answer:

Single-line comments, Multi-line comments, and XML comments are the types of comments in C#.

Slides 27 to 29

Understand constants.

Constants 1-3

- ◆ 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.

Example

- ◆ 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.

© Aptech Ltd. Building Applications Using C# /Session 2 27

Constants 2-3

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

Syntax

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

where,

- ◆ `const`: Keyword denoting that the identifier is declared as constant.
- ◆ `data type`: Data type of constant.
- ◆ `identifier name`: Name of the identifier that will hold the constant.
- ◆ `value`: Fixed value that remains unchanged throughout the execution of the code.

© Aptech Ltd. Building Applications Using C# /Session 2 28

Constants 3-3

- The following code declares a constant named `_pi` and a variable named `radius` to calculate the area of the circle:

Snippet

```
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.

In slide 27, tell the students that 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.

Then, to understand the constants, give them an example. Tell them to 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.

Mention that 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.

Use slide 28 to tell the students that constants for all data types can be declared. They have to initialize a constant at the time of its declaration.

Explain that the 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.

Then, explain that the given syntax initializes a constant. In the syntax, `const` is a keyword denoting that the identifier is declared as constant, `data type` is a data type of constant, `identifier name` is the syntax for a name of the identifier that will hold the constant, and `value` is a fixed value that remains unchanged throughout the execution of the code.

In slide 29, tell the students that the code declares a constant named `_pi` and a variable named `radius` to calculate the area of the circle. Explain to the students that 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.

Slide 30

Understand literals.



Using Literals

Literals

- ◆ A literal is a static value assigned to variables and constants.
- ◆ 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.

Example

For example, in the following declaration,

```
string bookName = "Csharp"
```

Csharp is a literal assigned to the variable `bookName` of type string.

Building Applications Using C# /Session 2 30

Use slide 30 to tell the students that a literal is a static value assigned to variables and constants.

Explain to the students that 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.

Give an example. Tell that, in the declaration `Csharp` is a literal assigned to the variable `bookName` of type `string`.

Slides 31 and 32

Understand types of literals.



Types of Literals 1-2

- ◆ **Boolean Literal:** Boolean literals have two values, `true` or `false`. For example,


```
boolval = true;
```

 where,
`true`: Is a Boolean literal assigned to the variable `val`.
- ◆ **Integer Literal:** An integer literal can be assigned to `int`, `uint`, `long`, or `ulong` data types. Suffixes for integer literals include `U`, `L`, `UL`, or `LU`. `U` denotes `uint` or `ulong`, `L` denotes `long`. `UL` and `LU` denote `ulong`. For example,


```
longval = 53L;
```

 where,
`53L`: Is an integer literal assigned to the variable `val`.
- ◆ **Real Literal:** A real literal is assigned to `float`, `double` (default), and `decimal` data types. This is indicated by the suffix letter appearing after the assigned value. A real literal can be suffixed by `F`, `D`, or `M`. `F` denotes `float`, `D` denotes `double`, and `M` denotes `decimal`. For example,


```
floatval = 1.66F;
```

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

 Aptech Ltd. Building Applications Using C# /Session 2 31



Types of Literals 2-2

- ◆ **Character Literal:** A character literal is assigned to a `char` data type. A character literal is always enclosed in single quotes. For example,


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

 where,
`A`: Is a character literal assigned to the variable `val`.
- ◆ **String Literal:** There are two types of string literals in C#, regular and verbatim. A regular string literal is a standard string. A verbatim string literal is similar to a regular string literal but is prefixed by the '@' character. A string literal is always enclosed in double quotes. For example,


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

 where,
`@gmail.com`: Is a verbatim string literal.
- ◆ **Null Literal:** The null literal has only one value, `null`. For example,


```
string email = null;
```

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

 Aptech Ltd. Building Applications Using C# /Session 2 32

Use slide 31 to tell the students that there are total six types of literals:

- Boolean Literal
- Integer Literal

- Real Literal
- Character Literal
- String Literal
- Null Literal

Begin with explaining the Boolean literal. Tell that the Boolean literals have two values, `true` or `false`. Give an example. Tell that in `boolval = true;`, the `true` is a Boolean literal assigned to the variable `val`.

Then, explain the Integer literal. Explain that the 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, in the `longval = 53L;` where, the `53L` is an integer literal assigned to the variable `val`.

Then, explain the Real literal. Tell that 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 in the `floatval = 1.66F;` the `1.66F` is a real literal assigned to the variable `val`.

Use slide 32 to tell the students that Character literal is assigned to a `char` data type. A character literal is always enclosed in single quotes. For example, in the `charval = 'A' ;`, the `A` is a character literal assigned to the variable `val`.

After this, explain the String literal. Explain that there are two types of string literals in C#, regular and verbatim. A regular String literal is a standard string. A Verbatim String literal is similar to a regular String literal but is prefixed by the '@' character. A String literal is always enclosed in double quotes. For example, in the `stringmailDomain = "@gmail.com";`, the `@gmail.com` is a verbatim string literal.

Finally, explain Null literal. The null literal has only one value, `null`. For example, in the `string email = null;`, the `null` specifies that e-mail does not refer to any objects (reference).

Tips:

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

Slides 33 to 35

Understand about keywords and escape sequences in C#.



Keywords and Escape Sequences 1-3

- ◆ Keywords are reserved words that are separately compiled by the compiler and convey a pre-defined meaning to the compiler and hence, cannot be created or modified.

Example int is a keyword that specifies that the variable is of data type integer.

- ◆ Escape sequence characters in C# are characters preceded by a backslash (\) and denote a special meaning to the compiler.
- ◆ You cannot use keywords as variable names, method names, or class names, unless you prefix the keywords with the '@' character.

© Aptech Ltd. Building Applications Using C# /Session 2 33



Keywords and Escape Sequences 2-3

- ◆ The following table lists the keywords used in C#:

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

© Aptech Ltd. Building Applications Using C# /Session 2 34



Keywords and Escape Sequences 3-3

- ◆ 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.
- ◆ The following table lists the contextual keywords used in C#:

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

© Aptech Ltd.

Building Applications Using C# /Session 2 35

Use slide 33 to tell the students that keywords are reserved words that are separately compiled by the compiler and convey a pre-defined meaning to the compiler and hence, cannot be created or modified.

Give an example. Tell the students that `int` is a keyword that specifies that the variable is of data type integer.

Explain that the escape sequence characters in C# are characters preceded by a backslash (\) and denote a special meaning to the compiler. Tell that keywords cannot be used as variable names, method names, or class names, unless you prefix the keywords with the '@' character.

In slide 34, refer and explain table that lists the keywords used in C#.

Use slide 35 to tell the students that 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.

You can refer and explain table in slide 35 that lists the contextual keywords used in C#.

Slides 36 to 38

Understand escape sequence characters.

Need of Escape Sequence Characters

Example

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




© Aptech Ltd. Building Applications Using C# /Session 2 36

Definition of Escape Sequences

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

Example

- ◆ \n is used to insert a new line similar to the Enter key of the keyboard.
- ◆ In C#, the escape sequence characters must always be enclosed in double quotes.

© Aptech Ltd. Building Applications Using C# /Session 2 37



Escape Sequence Characters in C# 1-3

- There are multiple escape sequence characters in C# that are used for various kinds of formatting. The following table displays the escape sequence characters and their corresponding non-printing characters in C#:

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

© Aptech Ltd. Building Applications Using C# /Session 2 38

Use slide 36 to bring out the need for escape sequence characters. Tell the students to consider an example of a payroll system of an organization. One of its functions is to display the monthly salary as output with the salary displayed on the next line. However, the salary should always be printed on the next line irrespective of the length of string to be displayed with the salary amount. To achieve this, the programmer will need to use escape sequences.

Use slide 37 to explain that an escape sequence character is a special character that is prefixed by a backslash (\) and are used to implement special non-printing characters such as a new line, a single space, or a backspace.

Explain to the students that these non-printing characters are used while displaying formatted output to the user to maximize readability. The backslash character tells the compiler that the following character denotes a non-printing character.

Mention that the \n is used to insert a new line similar to the Enter key of the keyboard. In C#, the escape sequence characters must always be enclosed in double quotes.

Use slide 38 to tell the students that there are several escape sequence characters in C# that are used for various kinds of formatting.

You can refer and explain table in slide 38 that displays the escape sequence characters and their corresponding non-printing characters in C#.

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

Slide 39

Understand the use of Unicode characters.

Escape Sequence Characters in C# 2-3

- The following code demonstrates the use of Unicode characters:

Snippet

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

Output

```
Hello!
David "2007"
```

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

© Aptech Ltd. Building Applications Using C# /Session 2 39

In slide 39, tell the students that the code demonstrates the use of Unicode characters. Explain the code. Tell in the code, the variable `str` is declared as type `string` and stores Unicode characters for the letters H, e, l, l, and o. The method uses the horizontal tab escape sequence character to display the output leaving one tab space.

The new line escape sequence character used in the string of the method displays the output of the next statement in the next line.

Mention that 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.

Slide 40

Understand some commonly used escape sequences.

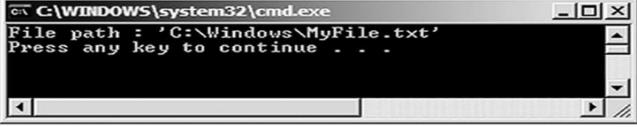
Escape Sequence Characters in C# 3-3

- The following code demonstrates the use of some of the commonly used escape sequences:

```
using System;
classfileDemo
{
    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");
        }
    }
}
```

Snippet

- 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.
- The following figure displays the output of the example of escape sequences:



Output

Building Applications Using C# /Session 2 40

In slide 40, tell the students that the code demonstrates the use of some of the commonly used escape sequences.

Explain the code. Tell the students that 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.

You can refer to the figure on slide 40 that displays the output of the example of escape sequences.

In-Class Question:

After you finish explaining escape sequence characters in C#, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are the types of Literals?

Answer:

The types of literals are Boolean Literal, Integer Literal, Real Literal, Character Literal, String Literal, and Null Literal.

Slide 41

Understand input and output.

Input and Output

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



© Aptech Ltd. Building Applications Using C# /Session 2 41

Use slide 41 to tell the students that the programmers often need to display the output of a C# program to users.

Explain to the students that they can use the command line interface to display the output and similarly accept inputs from a user through the command line interface. Such input and output operations are also known as console operations.

Slides 42 and 43

Understand the features of console operations.

Console Operations 1-2

- Following are the features of the 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.

© Aptech Ltd. Building Applications Using C# /Session 2 42

Console Operations 2-2

- The three streams are as follows:

Standard in	<ul style="list-style-type: none"> The standard in stream takes the input and passes it to the console application for processing.
Standard out	<ul style="list-style-type: none"> The standard out stream displays the output on the monitor.
Standard err	<ul style="list-style-type: none"> The standard err stream displays error messages on the monitor.

© Aptech Ltd. Building Applications Using C# /Session 2 43

In slide 42, explain the features of the console operations to the students.

Tell them that the 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.

Mention 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.

Use slide 43 to tell the three streams of console operations. Explain to the students that Standard `in` stream takes the input and passes it to the console application for processing. Then, tell the students that Standard `out` stream displays the output on the monitor. Mention that Standard `err` stream displays error messages on the monitor.

Slides 44 and 45

Understand output methods.

Output Methods 1-3

- ◆ In C#, all console operations are handled by the `Console` class of the `System` namespace.
- ◆ A namespace is a collection of classes having similar functionalities.
- ◆ To write data on the console, you need the standard output stream, provided by the `Console` class.
- ◆ There are two output methods that write to the standard output stream as follows:
 - ◆ `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.

© Aptech Ltd. Building Applications Using C# /Session 2 44

The slide has a teal header bar with the title "Output Methods 2-3". Below the header, there is a bulleted list of text. A blue button labeled "Syntax" is positioned above a code block. The code block contains the C# syntax for writing to the console.

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

Syntax

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

where,

- ◆ `data`: Specifies strings or escape sequence characters enclosed in double quotes.
- ◆ `variables`: Specify variable names whose value should be displayed on the console.

© Aptech Ltd. Building Applications Using C# /Session 2 45

Use slide 44 to tell the students that 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.

Tell them that to write data on the console, you need the standard output stream, provided by the `Console` class.

Explain that there are two output methods that write to the standard output stream 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.

Use slide 45 to tell the students that syntax used for the `Console.WriteLine()` method displays the information on the console window.

Then, tell the students that in the syntax, the `data` specifies strings or escape sequence characters enclosed in double quotes and `variables` specify variable names whose value should be displayed on the console.

Slide 46

Understand the syntax and code of output methods.

The slide has a teal header bar with the title "Output Methods 3-3". Below the title, there are two bullet points:

- The following syntax is used for the `Console.WriteLine()` method, which allows you to display the information on a new line in the console window:
Syntax
`Console.WriteLine("<data>" + variables);`
- The following code shows the difference between the `Console.Write()` and `Console.WriteLine()` methods:
Snippet
`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

At the bottom left is the copyright notice "© Aptech Ltd." and at the bottom right is the page number "Building Applications Using C#/Session 2 46".

In slide 46, tell the students that the syntax used for the `Console.WriteLine()` method displays the information on a new line in the console window.

Then, tell the students that the code displays the difference between the `Console.Write()` and `Console.WriteLine()` methods.

Slide 47

Understand placeholders.

The slide has a blue header bar with the title "Placeholders". Below the header, there is a list of bullet points explaining 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.
- The following code uses placeholders in the `Console.WriteLine()` method to display the result of the multiplication operation:

Snippet

```
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.
```

At the bottom left, there is a watermark for "Aptech Ltd.". At the bottom right, it says "Building Applications Using C# / Session 2" and "47".

Use slide 47 to tell the students that the `WriteLine()` and `Write()` methods accept a list of parameters to format text before displaying the output.

Explain to the students that 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.

Then, tell the students that the code uses placeholders in the `Console.WriteLine()` method to display the result of the multiplication operation.

Explain the output in the slide.

Slide 48

Understand input methods.

The slide has a blue header bar with the title "Input Methods 1-3". The main content area has a teal background. It contains a bulleted list of points about input methods in C#, followed by a code snippet in a yellow box, and then an output example below it.

- In C#, to read data, you need the standard input stream. This stream is provided by the input methods of the `Console` class. There are two input methods that enable the software to take in the input from the standard input stream.
- These methods are as follows:
 - `Console.Read()`: Reads a single character,
 - `Console.ReadLine()`: Reads a line of strings.
- The following code reads the name using the `ReadLine()` method and displays the name on the console window:

Snippet

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

Output

```
Enter your name: David Blake
You are David Blake
```

At the bottom left is the Aptech logo, and at the bottom right are the words "Building Applications Using C# / Session 2" and the number "48".

Use slide 48 to tell the students that in C#, to read data, the standard input stream is needed. 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.

Tell them that these methods are as follows:

- `Console.Read()`: Reads a single character.
- `Console.ReadLine()`: Reads a line of strings.

Tell the students that the code reads the name using the `ReadLine()` method and displays the name on the console window.

Tell them that in the code, the `ReadLine()` method reads the name as a string and the string that is given is displayed as output using placeholders.

Explain the output of the code.

Slides 49 and 50

Understand the code that shows the use of placeholders in the `Console.WriteLine()` method.

Input Methods 2-3

- The following code demonstrates the use of placeholders in the `Console.WriteLine()` method:

Snippet

```
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:#,###.##} \nInterest rate : {(1:0%)}\nInterest Amount : ${2:#,###.##}",
            loanAmount, interest, interestAmount );
        Console.WriteLine("Total amount to be paid : ${0:#,###.##} ", totalAmount);
    }
}
```

© Aptech Ltd. Building Applications Using C# /Session 2 49

Input Methods 3-3

- In the code:
 - 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.
- The following figure displays the example of placeholders:

© Aptech Ltd. Building Applications Using C# /Session 2 50

In slide 49, tell the students that the code demonstrates the use of placeholders in the `Console.WriteLine()` method.

Use slide 50 to explain the code and the output. Explain that in the code, 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.

You can refer to the figure in slide 50 that displays the example of placeholders.

Slide 51

Understand the various Convert methods.

Convert Methods

Snippet

- 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.
- The following code 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:

```
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
```

© Aptech Ltd. Building Applications Using C# /Session 2 51

Use slide 51 to tell the students that 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 with methods to convert one base data type to another base data type.

Tell the students that 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.

Tell that the code 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. Explain the output of the code.

Slide 52

Understand the different numeric format specifiers.

The slide has a blue header bar with the title 'Numeric Format Specifiers'. Below the title is a bulleted list of four points. A 'Syntax' section follows, containing a code example. Below the code, 'where,' is followed by two bullet points. The bottom of the slide shows a footer with the Aptech logo and page number.

- ◆ Format specifiers are special characters that are used to display values of variables in a particular format. For example, you can display an octal value as decimal using format specifiers.
- ◆ In C#, you can convert numeric values in different formats. For example, you can display a big number in an exponential form.
- ◆ To convert numeric values using numeric format specifiers, you should enclose the specifier in curly braces. These curly braces must be enclosed in double quotes. This is done in the output methods of the `Console` class.
- ◆ The following is the syntax for the numeric format specifier:

Syntax

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

where,

- ◆ `format specifier`: Is the numeric format specifier.
- ◆ `variable name`: Is the name of the integer variable.

Aptech Ltd. Building Applications Using C#/Session 2 52

In slide 52, tell the students that the format specifiers are special characters that are used to display values of variables in a particular format.

Give an example. Tell the students that you can display an octal value as decimal using format specifiers.

Mention that in C#, you can convert numeric values in different formats.

Give an example. Tell that you can display a big number in an exponential form.

Explain to the students that 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.

Tell the syntax of the numeric format specifier to the students where, the `format specifier` is the numeric format specifier and the `variable name` is the name of the integer variable.

Slide 53

Understand how to use the different numeric format specifiers.

The slide has a teal header bar with the title 'Using Numeric Format Specifiers'. Below the title is a list of bullet points. A blue box labeled 'Example' contains another list of bullet points. To the right is a table with three rows, each representing a numeric format specifier: C or c, D or d, and E or e. The table has three columns: 'FormatSpecifier', 'Name', and 'Description'. At the bottom of the slide is a footer bar with the text '© Aptech Ltd.' and 'Building Applications Using C# /Session 2 53'.

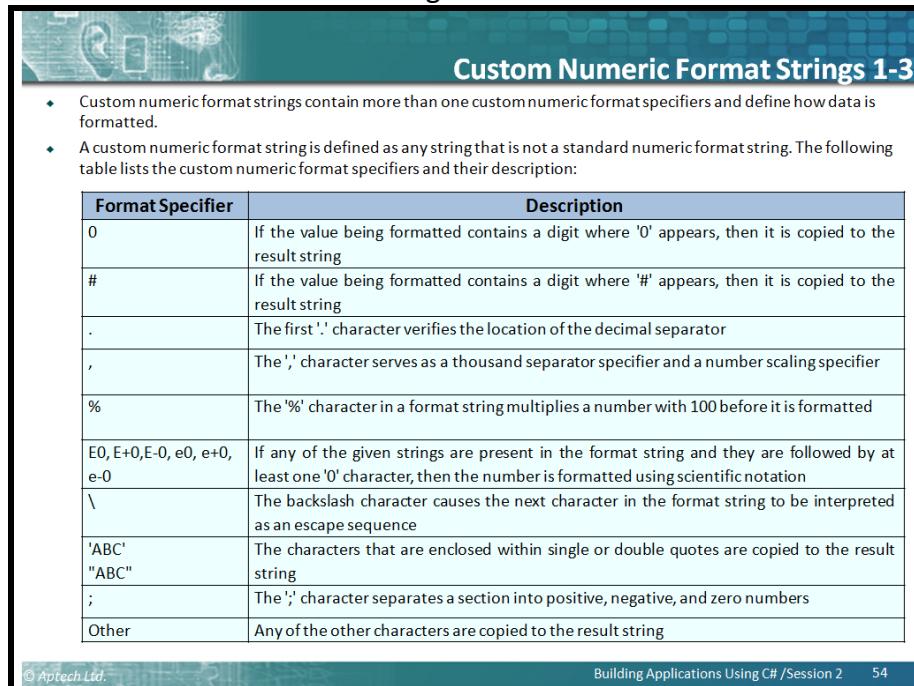
FormatSpecifier	Name	Description
C or c	Currency	The number is converted to a string that represents a currency amount.
D or d	Decimal	The number is converted to a string of decimal digits (0-9), prefixed by a minus sign in case the number is negative. The precision specifier indicates the minimum number 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).

Use slide 53 to tell the students that the numeric format specifiers work only with numeric data that can be suffixed with digits. The digits specify the number of zeroes to be inserted, after the decimal location. If they use a specifier such as C3, three zeroes will be suffixed, after the decimal location of the given number.

You can refer and explain table in slide 53 that lists some of the numeric format specifiers in C#.

Slides 54 to 56

Understand the custom numeric format strings.

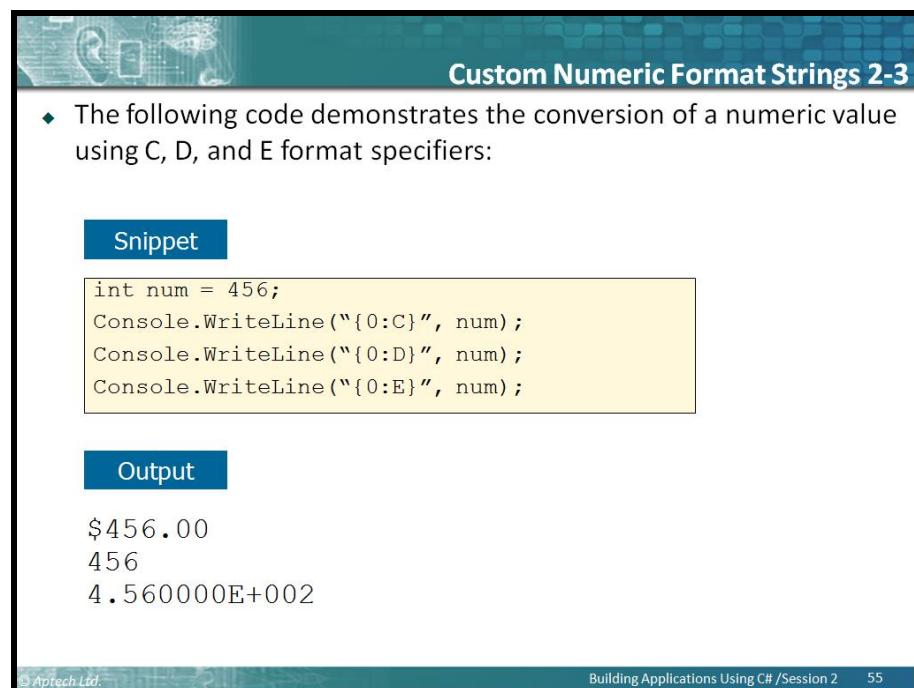


Custom Numeric Format Strings 1-3

- 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. The following table lists the custom numeric format specifiers and their description:

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

© Aptech Ltd. Building Applications Using C# /Session 2 54



Custom Numeric Format Strings 2-3

- The following code demonstrates the conversion of a numeric value using C, D, and E format specifiers:

Snippet

```
int num = 456;
Console.WriteLine("{0:C}", num);
Console.WriteLine("{0:D}", num);
Console.WriteLine("{0:E}", num);
```

Output

```
$456.00
456
4.560000E+002
```

© Aptech Ltd. Building Applications Using C# /Session 2 55

Custom Numeric Format Strings 3-3

- The following code demonstrates the use of custom numeric format specifiers:

Snippet

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

- In the code:
 - The #, %, ., and 0 custom numeric format specifiers are used to display the loan details of the customer in the desired format.
- The following figure displays the example of custom numeric format specifiers:

© Aptech Ltd. Building Applications Using C# /Session 2 56

In slide 54, tell the students that the 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.

You can refer and explain table in slide 54 that lists the custom numeric format specifiers and their description. Use slide 55 to tell the students that the code demonstrates the conversion of a numeric value using C, D, and E format specifiers to the students.

Explain the output of the code as shown on slide 55.

Use slide 56 to tell the students that the code shows the use of custom numeric format specifier. Explain the code. Tell that in the code, the custom numeric format specifiers such as #, %, ., and 0 are used to display the loan details of the customer in the desired format. You can refer to the figure in slide 56 that displays the example of custom numeric format specifiers to them.

Slides 57 and 58

Understand additional number format specifiers.

More Number Format Specifiers 1-2

- ◆ There are some additional number format specifiers that are described in the following table:

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

© Aptech Ltd. Building Applications Using C# /Session 2 57

More Number Format Specifiers 2-2

- ◆ The following figure demonstrates the conversion of a numeric value using F, N, and X format specifiers:

Snippet

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

Output

```
456.00
456.00
1C8
```

© Aptech Ltd. Building Applications Using C# /Session 2 58

In slide 57, explain some additional number format specifiers such as F or f , N or n, and X or x. You can refer and explain table in slide 57. In slide 58, tell the students that the code displays the conversion of a numeric value using F, N, and X format specifiers. Then, explain the output of the code.

Slide 59

Understand standard date and time format specifiers.

Standard Date and Time Format Specifiers

- ◆ A date and time format specifier is a special character that enables you to display the date and time values in different formats.

Example

- ◆ You can display a date in `mm-dd-yyyy` format and time in `hh : mm` format.
- ◆ If you are displaying GMT time as the output, you can display the GMT time along with the abbreviation GMT using date and time format specifiers.
- ◆ The date and time format specifiers allow you to display the date and time in 12-hour and 24-hour formats.
- ◆ The following is the syntax for date and time format specifiers:

Syntax

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

- ◆ where,
 - ◆ `format specifier`: Is the date and time format specifier.
 - ◆ `datetime object`: Is the object of the `DateTime` class.

In slide 59, explain that a date and time format specifier is a special character that enables you to display the date and time values in different formats. Tell them that they can display a date in `mm-dd-yyyy` format and time in `hh : mm` format. If they are displaying GMT time as the output, they can display the GMT time along with the abbreviation GMT using date and time format specifiers.

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

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

Explain to the students that in the syntax for date and time format specifiers, the `format specifier` is the date and time format specifier and `datetime object` is the object of the `DateTime` class.

Slide 60

Understand using standard date and time format specifiers.

Using Standard Date and Time Format Specifiers 1-2

- Standard date and time format specifiers are used in the `Console.WriteLine()` method with the `datetime` object that can be created using an object of the `DateTime` class and initialize it.
- The formatted date and time are always displayed as strings in the console window. The following table displays some of the standard date and time format specifiers in C#:

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

Aptech Ltd.

Building Applications Using C#/Session 2 60

In slide 60, tell the students that standard date and time format specifiers are used in the `Console.WriteLine()` method with the `datetime` object that can be created using an object of the `DateTime` class and initialize it.

Mention that the formatted date and time are always displayed as strings in the console window.

You can refer and explain table in slide 60 that displays some of the standard date and time format specifiers in C# to the students.

Slide 61

Understand the conversion of a specified date and time.

The following code demonstrates the conversion of a specified date and time using the d, D, f, F, and g date and time format specifiers:

Snippet

```
DateTimedt = DateTime.Now;
// Returns short date (MM/DD/YYYY)
Console.WriteLine("Short date format (d): {(0:d)}", dt);
// Returns long date (Day, Month Date, Year)
Console.WriteLine("Long date format (D): {(0:D)}", dt);
// Returns full date with time without seconds
Console.WriteLine("Full date with time without seconds (f):{(0:f)}", dt);
// Returns full date with time with seconds
Console.WriteLine("Full date with time with seconds (F):{(0:F)}", dt);
// Returns short date and short time without seconds
Console.WriteLine("Short date and short time without seconds (g):{(0:g)}", dt);
```

Output

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

Aptech Ltd. Building Applications Using C#/Session 2 61

In slide 61, tell the students that the code demonstrates the conversion of a specified date and time using the d, D, f, F, and g date and time format specifiers. Then, explain the output of the code.

Slides 62 and 63

Understand some more standard date and time format specifiers in C#.

Additional Standard Date and Time Format Specifiers 1-2

- The following table displays the standard date and time format specifiers in C#:

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

© Aptech Ltd. Building Applications Using C# /Session 2 62

Additional Standard Date and Time Format Specifiers 2-2

- The following code demonstrates the conversion of a specified date and time using the G, m, t, T, and y date and time format specifiers:

Snippet

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

Output

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

© Aptech Ltd. Building Applications Using C# /Session 2 63

In slide 62, explain with the help of the table the additional standard date and time format specifiers in C#. In slide 63, tell the students that the code demonstrates the conversion of a specified date and time using the G, m, t, T, and y date and time format specifiers. Then, explain the output of the code.

Slides 64 and 65

Understand the custom DateTime format strings.

Custom DateTime Format Strings 1-2

- Any non-standard DateTime format string is referred to as a custom DateTime format string. Custom DateTime format strings consist of more than one custom DateTime format specifiers. The following table lists some of the custom DateTime format specifiers:

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

- The following code demonstrates the use of custom DateTime format specifiers:

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

© Aptech Ltd. Building Applications Using C# /Session 2 64

Custom DateTime Format Strings 2-2

- In the code:
 - The date and time is displayed using the different DateTime format specifiers.
- The following figure displays the output of the code:

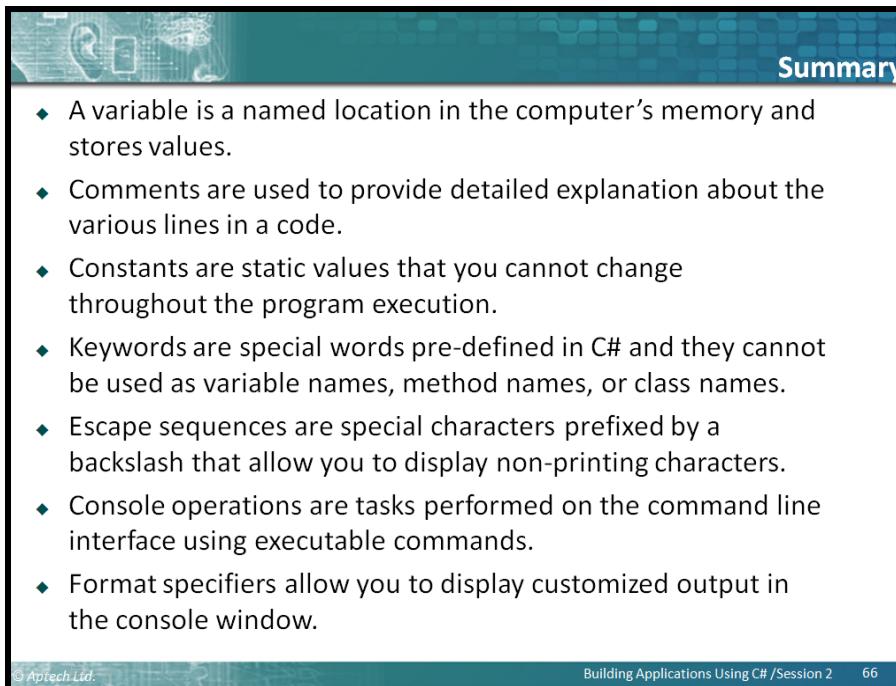
© Aptech Ltd. Building Applications Using C# /Session 2 65

In slide 64, tell the students that any non-standard DateTime format string is referred to as a custom DateTime format string. Custom DateTime format strings consist of more than one custom DateTime format specifiers.

You can refer slide 64 and explain table that lists some of the custom `DateTime` format specifiers to them. Explain to the students that the code demonstrates the use of custom `DateTime` format specifiers. Use slide 65 to explain the code. Tell that in the code, the date and time is displayed using the different `DateTime` format specifiers. You can refer to the figure in slide 65 that displays the output of the code.

Slide 66

Summarize the session.



The slide has a teal header bar with the word "Summary" in white. Below the header is a list of nine bullet points. At the bottom of the slide is a footer bar with the text "© Aptech Ltd." on the left and "Building Applications Using C# /Session 2 66" on the right.

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

In slide 66, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Explain to the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief.

Tell them that:

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

2.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. What are pointer type variables?
2. What are converting data types?
3. Explain convert and parse methods?

Session 3 - Statements and Operators

3.1 Pre-Class Activities

Before you commence the session, you should revisit topics of the previous session for a brief review before the session begins. The summary of the previous session is as follows:

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

Here, you can ask students the key topics they can recall from previous session. Prepare a question or two which will be a key point to relate the current session objectives. Ask them to explain variables and data types in C# and explain comments and XML documentation.

3.1.1 Objectives

By the end of this session, the learners will be able to:

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

3.1.2 Teaching Skills

To teach this session successfully, you must know about statements and expressions. You should be aware of the types of operators. You should also be familiar with the process of performing data conversions in C#.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

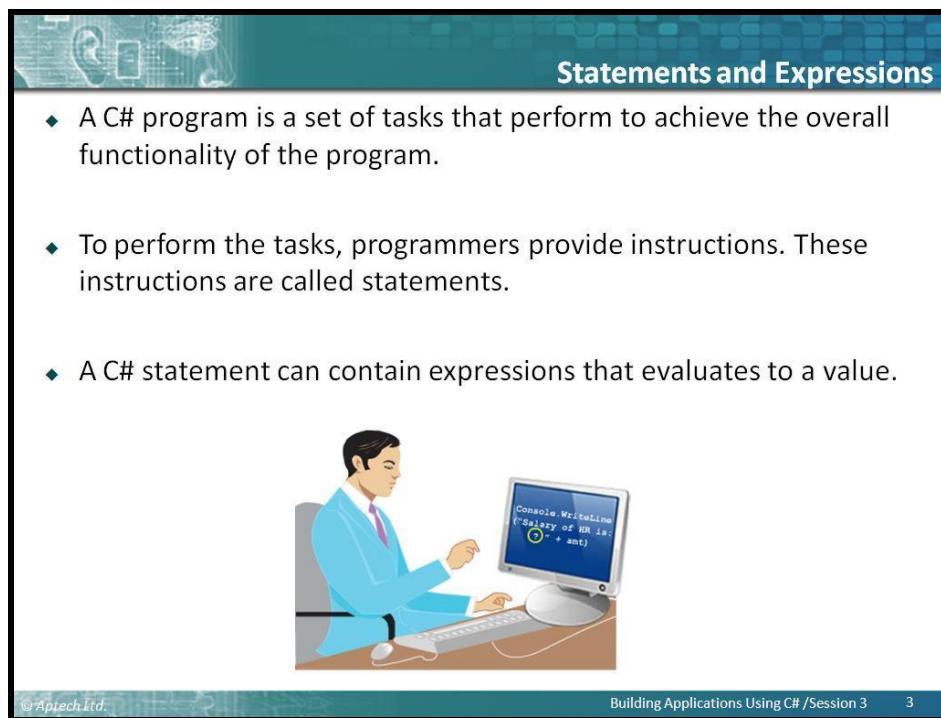
Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell the students that they will be introduced to statements and expressions. They will learn about the types of operators. This session will also discuss the process of performing data conversions in C#.

3.2 In-Class Explanations

Slide 3

Understand statements and expressions.



Statements and Expressions

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



© Aptech Ltd. Building Applications Using C# /Session 3 3

Use slide 3 to explain that files in a C# program is a set of tasks that perform to achieve the overall functionality of the program.

Mention that to perform the tasks, programmers provide instructions. These instructions are called statements. A C# statement can contain expressions that evaluates to a value.

Give some examples of C# statements so that the students get clear idea about what is meant by a statement.

A statement can be:

```
Console.WriteLine("How are you?");  
int age=Convert.ToInt32(Console.ReadLine());
```

An expression can be:

```
int no1=10, no2=15, no3;
```

```
no3 = no1 + no2;
```

Give similar such examples.

Slide 4

Understand statements.

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.
- ◆ The following figure displays an example of a block of statements:

```

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);
    }
}

```

© Aptech Ltd. Building Applications Using C# /Session 3 4

In slide 4, explain the students that statements are referred to as logical grouping of variables, operators, and C# keywords that perform a specific task.

Give them an example for this. Tell the students, the line which initializes a variable by assigning it a value is a statement.

Explain them that 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. Give them an example for this. Tell the students that the set of statements included in the `Main()` method of a C# code is a block. A method in C# is equivalent to a function in earlier programming languages such as C and C++.

You can refer to the figure in slide 4 that displays an example of a block of statements. A block is used to mark the beginning and end of a method, a loop, a conditional statement, a structure, or a class. It sets the boundaries for the mentioned concepts.

In slide 4, the block is used to group all the statements that are to be executed by the `Main()` method.

Slides 5 to 7

Understand the uses of statements.

Statements – Uses 1-3

- ◆ 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
 - ◆ Display the output

© Aptech Ltd. Building Applications Using C# /Session 3 5

Statements – Uses 2-3

- ◆ The following code shows an example of a statement in C#:

Snippet

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

- ◆ This line of code is an example of a C# statement that calculates the area of the circle and stores the value in the variable area.
- ◆ The following code shows an example of a block of statements in C#.

Snippet

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

- ◆ In the code:
 - ◆ A block of code is enclosed within curly braces.
 - ◆ The first statement from the top will be executed first followed by the next statement and so on.

© Aptech Ltd. Building Applications Using C# /Session 3 6

Statements – Uses 3-3

- The following code shows an example of nested blocks in C#:

Snippet

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

- In the code:
 - Another block of code is 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.

© Aptech Ltd. Building Applications Using C# /Session 3 7

In slide 5, explain to the students that statements are used to specify the input, the process, and the output tasks of a program.

Tell that statements consist of:

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

Then, tell them that statements help to build a logical flow in the program.

Mention that statements initialize variables and objects, take the input, call a method of a class, and display the output. These are types of categories in which the statements can be classified. Give them brief idea about each type of statements and what they are supposed to do. Tell the students the difference between a variable name and a keyword.

Use slide 6 to tell the students that the code demonstrates an example of a statement in C#. Tell the students 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`.

Then, tell the students that the next code is an example of a block of statements in C#.

Mention that a block of code is enclosed within curly braces and the first statement from the top will be executed first followed by the next statement and so on.

Ask students what type of statements are shown in the given snippet. Give them some examples and ask them to identify the statements and expressions. Tell the students to list some of the keywords from the given code snippet. Use slide 7 to tell that the code demonstrates an example of nested blocks in C#. Mention to them that the lines of code show another block of code nested within a block of statements.

Explain that the first three statement from the top will be executed in sequence followed by the line of code within the inside braces that will be executed to calculate the area. The execution is terminated at the last statement in the block of the code displaying the area.

Additional Information

Give them some more examples of block codes. You can use various methods or functions and write some statements inside the blocks and demonstrate the same. Tell the students that we can have nested code blocks also. Explain how nested code blocks are written and explains its execution order.

In-Class Question:

After you finish explaining statements, i.e. statements are block of code given in a program, which can be nested one inside another. What are the Statements, different types of statements and using statements in code blocks.



Question: What is code block?

Answer: A code block is generally denoted by the open brace and close brace. Open brace marks the beginning of the code block and the close brace marks the end of the code block. The code blocks are used with methods, loops, conditional statements, structures, and classes.

Slide 8

Understand types of statements.

The slide has a teal header bar with the title "Types of Statements". The main content area contains a bulleted list of seven categories of statements. At the bottom, there is a footer bar with the copyright information "© Aptech Ltd." and the slide number "8".

Types of Statements

- ◆ Similar to statements in C and C++, the C# statements are classified into seven categories:
 - ◆ Selection Statements
 - ◆ Iteration Statements
 - ◆ Jump Statements
 - ◆ Exception Handling Statements
 - ◆ Checked and Unchecked Statements
 - ◆ Fixed Statement
 - ◆ Lock Statement

© Aptech Ltd. Building Applications Using C# /Session 3 8

In slide 8, explain to the students that C# statements are similar to statements in C and C++.

Explain that C# statements are classified into seven categories according to the function they perform.

List and explain the various categories of statements as given on the slide.

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

Then, tell that iteration statements help to repeatedly execute a block of code. The keywords associated with this statement are: `do`, `for`, `foreach`, and `while`.

Mention that jump statements help to transfer the flow from one block to another block in the program. The keywords associated with this statement are `break`, `continue`, `default`, `goto`, `return`, and `yield`.

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

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

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

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

Slides 9 to 12

Understand checked and unchecked statements.

Checked and Unchecked Statements 1-4

- Following are the features of the checked and unchecked statements:

The checked statement checks for an arithmetic overflow in arithmetic expressions and 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 causing 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 the different types of exceptions.

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

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

```
checked-statement:  
checked block
```

© Aptech Ltd. Building Applications Using C# /Session 3 9

Checked and Unchecked Statements 2-4

- The following code creates a class `Addition` with the `checked` statement that throws an overflow exception.

Snippet

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

When the statement in the `checked` block is executed, it gives an error. This is because the result of the addition of the two numbers results in 256. This value is too large to be stored in the `byte` variable causing an arithmetic overflow to occur.

Output

```
C:\WINDOWS\system32\cmd.exe
System.OverflowException: Arithmetic operation resulted in an overflow.
at Project.SG03.Addition.Main() in D:\Source Code\Project\Project\SG03\Addition.cs:line 20
Press any key to continue . . .


```

© Aptech Ltd. Building Applications Using C# /Session 3 10



Checked and Unchecked Statements 3-4

- ◆ The unchecked statement is associated with the unchecked keyword.
- ◆ The unchecked statement ignores the arithmetic overflow and assigns junk data to the target variable.
- ◆ An unchecked statement creates an unchecked context for a block of statements and has the following form:

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



Checked and Unchecked Statements 4-4

- ◆ The following code creates a class **Addition** that uses the unchecked statement:

Snippet

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

- ◆ In the code:
 - ◆ 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.

In slide 9, explain to the students that the checked statement checks for an arithmetic overflow in arithmetic expressions. On the contrary, the unchecked statement does not check for an arithmetic overflow.

Tell that 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 the different types of exceptions.

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

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

checked-statement:
checked block



Question : Why do we use a checked statement?

Answer : A checked statement is used to check if there is any arithmetic overflow.

In slide 10, explain to the students that the code creates a class **Addition** with the checked statement. This statement throws an overflow exception.

Explain to them that in the code, 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.

Use slide 11 to refer to figure that demonstrates the error message that is displayed after executing the code.

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

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

unchecked-statement:
unchecked block

In slide 12, explain to the students that the code creates a class **Addition** that uses the unchecked statement. Mention to them that in the code, 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.

Explain that 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.

Slide 13

Understand expressions.

The slide has a decorative header with icons of a brain, a gear, and a smartphone. The title 'Expressions – Definition' is centered at the top. Below the title is a bulleted list of points about expressions. A 'Snippet' box contains a small block of C# code. At the bottom, there is a footer bar with the Aptech logo and navigation icons.

Expressions – Definition

- ◆ Expressions are used to manipulate data. Like in mathematics, expressions in programming languages, including C#, are constructed from the operands and operators.
- ◆ An expression statement in C# ends with a semicolon (;).
- ◆ Expressions are used to:
 - ◆ Produce values.
 - ◆ Produce a result from an evaluation.
 - ◆ Form part of another expression or a statement.
- ◆ The following code demonstrates an example for expressions:

Snippet

```
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`.

© Aptech Ltd.

Building Applications Using C# /Session 3 13

In slide 13, explain to the students that expressions are used to manipulate data. Like in mathematics, expressions in programming languages, including C#, are constructed from the operands and operators. An expression statement in C# ends with a semicolon (;).

Tell the students that expressions are used to:

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

Then tell the students that the code demonstrates an example for expressions and tell them that 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`.

You can give your own examples to clarify the expressions to the students. Use various operators to write different types of expressions that produce different values. A few more examples of expressions are as follows:

```
dividend/divisor  
x % y  
b >= 3
```

Additional Information

For more information on expressions, refer the following link:

<http://msdn.microsoft.com/en-us/library/ms173144.aspx>

Slide 14

Understand the differences between statements and expressions.

Differences between Statements and Expressions

◆ Some fundamental differences between statements and expressions are listed in the table.

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

© Aptech Ltd.

Building Applications Using C# /Session 3 14

In slide 14, refer and explain the table that lists the fundamental differences between statements and expressions.

With this slide, you will finish explaining statements and expressions.

Give them some examples of statements and expressions randomly and ask the students to differentiate between the statements and expression. Students should be able to differentiate it.

Ask them to tell you the values after evaluation of each of the expressions.

Slides 15 and 16

Understand operators.

The slide has a blue header bar with the title "Operators". Below the header, there is a bulleted list: "Following are the features of operators in C#:". The slide is divided into five colored boxes:

- Red box:** Expressions in C# comprise one or more operators that performs some operations on variables.
- Green box:** An operation is an action performed on single or multiple values stored in variables in order to modify them or to generate a new value with the help of minimum one symbol and a value.
- Purple box:** The symbol is called an operator and it determines the type of action to be performed on the value.
- Orange box:** 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.
- Cyan box:** The value on which the operation is to be performed is called an operand.

At the bottom left, it says "© Aptech Ltd." and at the bottom right, it says "Building Applications Using C# /Session 3 15".

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.
- ◆ These are classified into six categories based on the action they perform on values:
 - ◆ Arithmetic Operators
 - ◆ Relational Operators
 - ◆ Logical Operators
 - ◆ Conditional Operators
 - ◆ Increment and Decrement Operators
 - ◆ Assignment Operators

Building Applications Using C# /Session 3 16

In slide 15, explain to the students that 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 in order to modify them or to generate a new value.

Explain that 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.

Mention that 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.

In slide 16, explain to the students that the 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.

Mention the six categories of operators:

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

Slides 17 and 18

Understand the types of operators.

Arithmetic Operators – Types 1-2

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

© Aptech Ltd.

Building Applications Using C# /Session 3 17

Arithmetic Operators – Types 2-2

- ◆ The following code demonstrates how to use the arithmetic operators:

Snippet

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

Output

```
Addition 12
Subtraction 8
Multiplication 20
Division 5
Remainder 0
```

© Aptech Ltd.

Building Applications Using C# /Session 3 18

In slide 17, explain to the students that 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.

You can refer to table from slide 17 that lists the arithmetic operators along with their descriptions and an example of each type.

In slide 18, explain to the students that the code demonstrates how to use the arithmetic operators and also show the output.

Explain them about the operator's precedence also. Tell the students how it is that the results are affected if we don't pay attention to operator precedence. You can demonstrate the same by making use of () with other arithmetic operators and show the output by placing the () at different locations/positions in the same expression.

Slides 19 and 20

Understand relational operators.


Relational Operators 1-2

- ◆ Relational operators make a comparison between two operands and return a boolean value, true, or false.
- ◆ The following table lists the relational operators along with their descriptions and an example of each type.

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

©Aptech Ltd.

Building Applications Using C# /Session 3 19

Relational Operators 2-2

- The following code demonstrates how to use the relational operators.

Snippet

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

Output

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

©Aptech Ltd. Building Applications Using C# /Session 3 20

In slide 19, explain to the students that relational operators make a comparison between two operands and return a boolean value, true or false. You can refer and explain table from slide 19 that lists the relational operators along with their descriptions and an example of each type. In slide 20, explain to the students that the code demonstrates how to use the relational operators.

Slide 21

Understand logical operators.

The slide has a blue header bar with the title "Logical Operators 1-7". Below the title, there is a bulleted list:

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

Two rounded rectangular callout boxes are positioned below the list:

- A purple box containing the text "Boolean Logical Operators".
- A red box containing the text "Bitwise Logical Operators".

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# /Session 3", and the number "21".

In slide 21, explain to the students that logical operators are binary operators that perform logical operations on two operands and return a boolean value. C# supports two types of logical operators, Boolean and Bitwise.

Slide 22

Understand boolean logical operators.

Logical Operators 2-7

- ◆ **Boolean Logical Operators:**
 - ◆ Boolean logical operators perform boolean logical operations on both the operands. They return a boolean value based on the logical operator used.
 - ◆ The following table lists the boolean logical operators along with their descriptions and an example of each type:

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

© Aptech Ltd. Building Applications Using C# /Session 3 22

In slide 22, explain to the students that Boolean logical operators perform boolean logical operations on both the operands. They return a boolean value based on the logical operator used. Refer and explain the table in slide 22 that lists the boolean logical operators along with their descriptions and an example of each type.

Slide 23

Understand the use of the boolean inclusive OR operator.

Logical Operators 3-7

- The following code explains the use of the boolean inclusive OR operator:

Snippet

```
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.
- The following code explains the use of the boolean AND operator:

Snippet

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

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

In slide 23, explain to the students that 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.

Also, explain to the students that the code explains the use of the boolean AND operator in the code, the boolean AND operator checks both the expressions. If both the expressions evaluate to true, the complete expression returns true and the statement within the block is executed.

Tell the students that when we use AND (&&) operator actually internally multiplication of two values takes place. So '0' multiplied by '1' gives you '0'. And when we use OR (||) operator actually the addition operation is performed on the given variables. Thus, '0' added to '1' gives you '1' as the result. You can also make use of the truth table to make it more clear to the students.

Slide 24

Understand the boolean exclusive OR operator.

The slide has a blue header bar with the title "Logical Operators 4-7". Below the header, there is a list of bullet points and a code snippet.

- ◆ The following code explains the use of the boolean exclusive OR operator:

Snippet

```
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.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# /Session 3", and "24".

In slide 24, explain the use of the boolean exclusive OR operator.

Then, tell them that 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.

Tell the students that sometimes we need to check for the condition where both the conditions evaluates to true or both the conditions evaluates to false. When one condition is true and other condition is false then only we need to execute certain steps in such cases we can use exclusive OR.

Slides 25 and 26

Understand bitwise logical operators.

Logical Operators 5-7

- Bitwise Logical Operators:**
 - The bitwise logical operators perform logical operations on the corresponding individual bits of two operands.
 - The following table lists the bitwise logical operators along with their descriptions and an example of each type:

Logical Operators	Description	Examples
& (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
- The following code explains the working of the bitwise AND operator:

Snippet

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

© Aptech Ltd. Building Applications Using C# /Session 3 25

Logical Operators 6-7

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

© Aptech Ltd. Building Applications Using C# /Session 3 26

In slide 25, explain to the students that bitwise logical operators perform logical operations on the corresponding individual bits of two operands.

You can refer and explain table from slide 25 that lists the bitwise logical operators along with their descriptions and an example of each type.

Tell that the code explains the working of the bitwise AND operator.

In slide 26, mention that in the code, the bitwise AND operator compares the corresponding bits of the two operands. It returns 1 if both the bits in that position are 1 or else returns 0. This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number. This number is automatically converted to integer, which is displayed as the output. The resultant output for the code is “24”.

You can refer and explain figure from slide 26 that displays the bitwise logical operators.

Tell the students how to make use of &&, ||, and ! operators. You can also demonstrate to the students the use of bitwise operators & and | operators. Show them with real time examples that shows what difference does it make if you use ‘&&’ and ‘&’. Explain what happens internally in the memory when we use logical operators and bitwise operators. Tell the students that bitwise operators actually do the shifting of ‘0’ and ‘1’ at bit level and thus changes the values and logical operators checks for the boolean results.

Slide 27

Understand how a code explains the working of the bitwise inclusive OR operator.

Logical Operators 7-7

- ◆ The following code explains the working of the bitwise inclusive OR operator:

Snippet

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

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

Snippet

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

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

© Aptech Ltd. Building Applications Using C# /Session 3 27

In slide 27, explain to the students that bitwise inclusive OR operator compares the corresponding bits of the two operands. It returns 1 if either of the bits in that position is 1, else it returns 0. This comparison is performed on each of the individual bits and the results of these comparisons form an 8-bit binary number. This number is automatically converted to integer, which is displayed as the output. The resultant output for the code is 60.

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

Slides 28 to 30

Understand the conditional operators.

Conditional Operators 1-3

- There are two types of conditional operators, conditional AND (`&&`) and conditional OR (`||`).
- Conditional operators are similar to the boolean logical operators but have the following differences:
 - The conditional AND operator evaluates the second expression only if the first expression returns true because this operator returns true only if both expressions are true.
 - The conditional OR operator evaluates the second expression only if the first expression returns false because this operator returns true if either of the expressions is true.

```

graph TD
    subgraph Conditional_AND [Conditional AND]
        A1[Expression 1] --> B1[True]
        A1 --> B2[False]
        B1 --> C1[Expression 2]
        B2 --> C1
        C1 --> D1[True]
        C1 --> D2[False]
        D1 --> E1[Output: True]
        D2 --> E2[Output: False]
    end

    subgraph Conditional_OR [Conditional OR]
        A2[Expression 1] --> B3[True]
        A2 --> B4[False]
        B3 --> C2[Expression 2]
        B4 --> C2
        C2 --> D3[True]
        C2 --> D4[False]
        D3 --> E3[Output: True]
        D4 --> E4[Output: False]
    end
  
```

© Aptech Ltd. Building Applications Using C# /Session 3 28

Conditional Operators 2-3

- The following code displays the use of the conditional AND (`&&`) operator:

Snippet

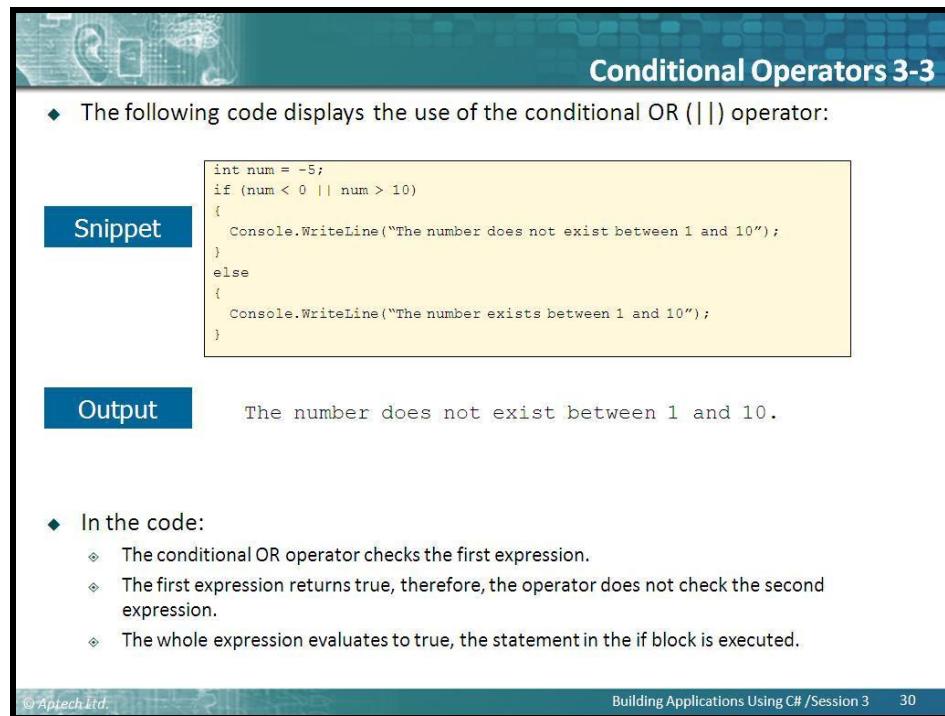
```
int num = 0;
if (num >= 1 && num <= 10)
{
    Console.WriteLine("The number exists between 1 and 10");
}
else
{
    Console.WriteLine("The number does not exist between 1 and 10");
}
```

- 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

© Aptech Ltd. Building Applications Using C# /Session 3 29



Conditional Operators 3-3

◆ The following code displays the use of the conditional OR (||) operator:

Snippet

```
int num = -5;
if (num < 0 || num > 10)
{
    Console.WriteLine("The number does not exist between 1 and 10");
}
else
{
    Console.WriteLine("The number exists between 1 and 10");
}
```

Output The number does not exist between 1 and 10.

◆ In the code:

- ◊ The conditional OR operator checks the first expression.
- ◊ The first expression returns true, therefore, the operator does not check the second expression.
- ◊ The whole expression evaluates to true, the statement in the if block is executed.

©Aptech Ltd. Building Applications Using C# /Session 3 30

Use slide 28 to explain that there are two types of conditional operators, conditional AND (&&) and conditional OR (||).

Explain that conditional operators are similar to the boolean logical operators but have some differences.

Mention the differences.

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

Then, explain to the students that the conditional OR operator evaluates the second expression only if the first expression returns false. This is because this operator returns true if either of the expressions is true. Thus, if the first expression itself evaluates to true, the result of the second expression is immaterial. The figure displays the working of conditional operators.

In slide 29, tell the students that the code displays the use of the conditional AND (&&) operator.

Explain to them that 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.

In slide 30, tell the students how the code displays the use of the conditional OR (||) operator.

Then, tell them that in the code, the conditional OR operator checks the first expression. The first expression returns true, therefore, the operator does not check the second expression. The whole expression evaluates to true, the statement in the `if` block is executed.

In-Class Question:

After you finish explaining conditional operators, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are relational operators?

Answer:

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

Slide 31

Understand the increment and decrement operators.

The slide has a blue header bar with the title 'Increment and Decrement Operators'. Below the title is a bulleted list of six points. At the bottom is a table comparing four expressions with their types and results. The footer contains copyright and page information.

- ◆ 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.
- ◆ The following table 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>

© Aptech Ltd. Building Applications Using C# /Session 3 31

In slide 31, explain to the students that two of the most common calculations performed in programming are increasing and decreasing the value of the variable by 1.

Explain that 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.

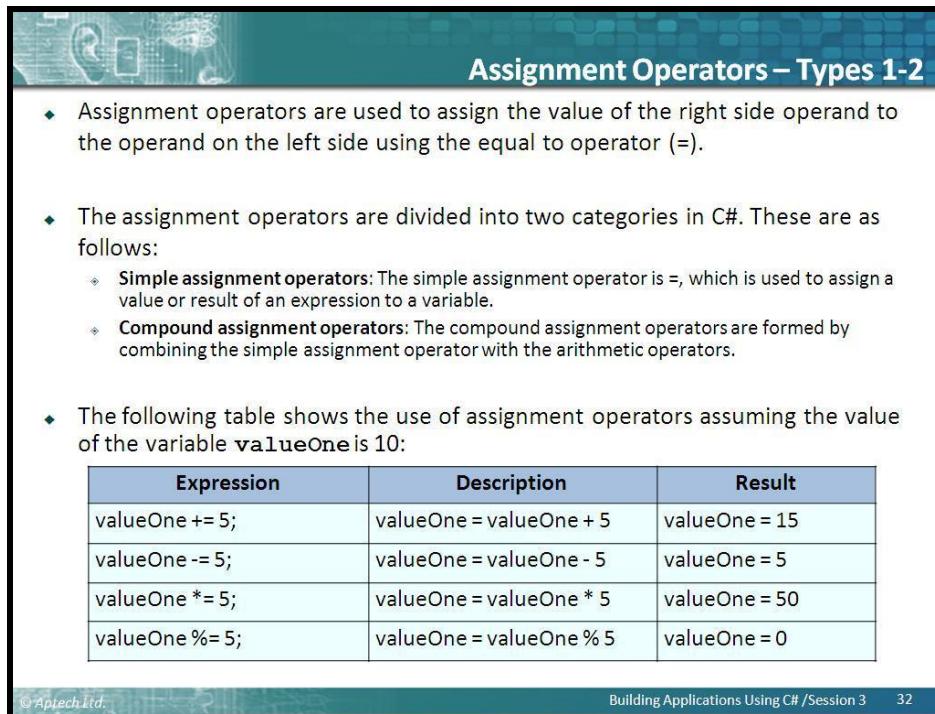
Tell that if the operator is placed before the operand, the expression is called pre-increment or pre-decrement. If the operator is placed after the operand, the expression is called post-increment or post-decrement.

You can refer to table from slide 31 that depicts the use of increment and decrement operators assuming the value of the variable **valueOne** is 5.

Show them a practical example by using a variable. Assign value 10 into that variable. Then print the value. Use ++ operator to increment the value and print the variable. Again use – operator and print the value. Thus, you can show the output line by line by putting `Console.ReadLine()` statement in between to stop the process, so that it becomes clear to the students.

Slides 32 and 33

Understand assignment operators.

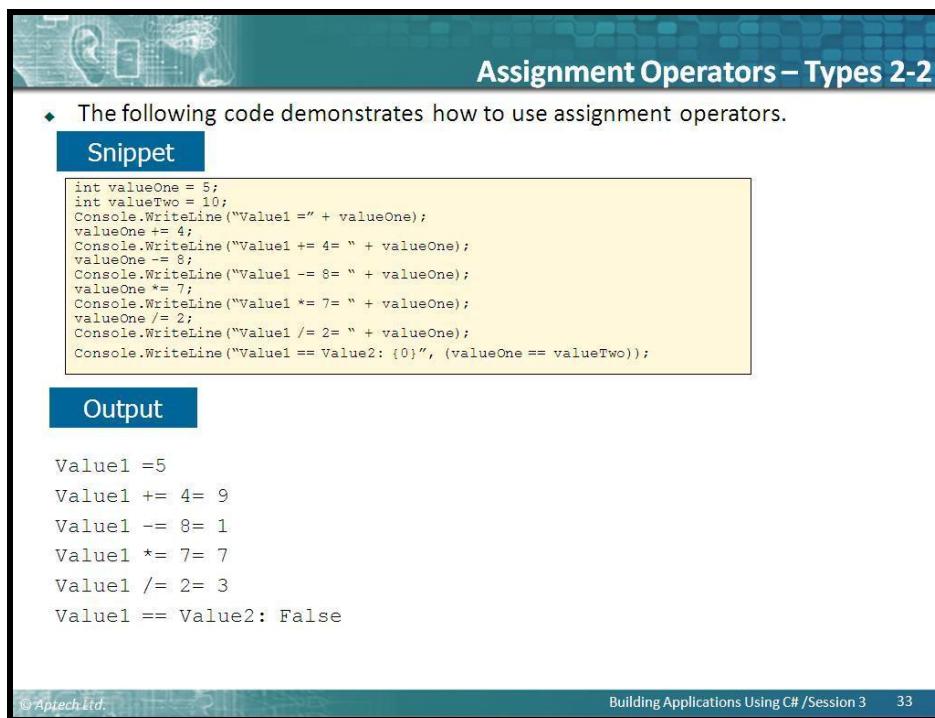


Assignment Operators – Types 1-2

- Assignment operators are used to assign the value of the right side operand to the operand on the left side using the equal to operator (=).
- The assignment operators are divided into two categories in C#. These are as follows:
 - Simple assignment operators:** The simple assignment operator is =, which is used to assign a value or result of an expression to a variable.
 - Compound assignment operators:** The compound assignment operators are formed by combining the simple assignment operator with the arithmetic operators.
- The following table 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

© Aptech Ltd. Building Applications Using C# /Session 3 32



Assignment Operators – Types 2-2

- The following code demonstrates how to use assignment operators.

Snippet

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

Output

```
Value1 =5
Value1 += 4= 9
Value1 -= 8= 1
Value1 *= 7= 7
Value1 /= 2= 3
Value1 == Value2: False
```

© Aptech Ltd. Building Applications Using C# /Session 3 33

In slide 32, explain that assignment operators are used to assign the value of the right side operand to the operand on the left side using the equal to operator (`=`). The assignment operators are divided into two categories in C#.

Tell that the simplest assignment operator is `=`, which is used to assign a value or result of an expression to a variable.

Then, explain to the students that compound assignment operators are formed by combining the simple assignment operator with the arithmetic operators.

You can refer to the table from slide 32 that shows the use of assignment operators assuming the value of the variable **valueOne** is 10.

In slide 33, explain to the students that the code demonstrates how to use assignment operators and also show the output.

Explain that 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.

Slides 34 and 35

Understand precedence and associativity.

Precedence and Associativity 1-2

- Operators in C# have certain associated priority levels.
- The C# compiler executes operators in the sequence defined by the priority level of the operators.

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.
- The following table 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

© Aptech Ltd. Building Applications Using C# /Session 3 34

Precedence and Associativity 2-2

- The following code demonstrates precedence of operators:

Snippet

```
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

```
True
False
```

© Aptech Ltd. Building Applications Using C# /Session 3 35

In slide 34, explain that operators in C# have certain associated priority levels. The C# compiler executes operators in the sequence defined by the priority level of the operators.

Give an example for this. Tell the students that 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.

You can refer to table from slide 34 that lists the precedence of the operators, from the highest to the lowest precedence, their description and their associativity.

Use slide 35, explain that the code demonstrates precedence of operators.

Tell the students that 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.

Slides 36 and 37

Understand the shift operators.

Shift Operators 1-2

- The shift operators allow the programmer to perform shifting operations. The two shifting operators are as follows:
 - The Left Shift (<<) Operators**
 - The Right Shift (>>) Operators**
- The left shift operator allows shifting the bit positions towards the left side where the last bit is truncated and zero is added on the right.
- The right shift operator allows shifting the bit positions towards the right side and the zero is added on the left.
- The following code demonstrates the use of shift operators.

Snippet

```
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 = 80
        result = num >> 1; // 01010000 = 40
        Console.WriteLine("\nValue before right shift : " + num);
        Console.WriteLine("Value after right shift : " + result);
    }
}
```

© Aptech Ltd. Building Applications Using C# /Session 3 36

Shift Operators 2-2

- In the code:
 - The `Main()` method of the class **ShiftOperator** performs the shifting operations.
 - The `Main()` method initializes the variable `num` to 100.
 - After shifting towards left, the value of `num` is doubled. Now, the variable `num` is initialized to 80.
 - After shifting towards right, the value of `num` is halved.
- Following is the output of code:

```
C:\WINDOWS\system32\cmd.exe
Value before left shift : 100
Value after left shift 200

Value before right shift : 80
Value after right shift : 40
Press any key to continue . . .
```

© Aptech Ltd. Building Applications Using C# /Session 3 37

In slide 36, explain the students, the shift operators allow the programmer to perform shifting operations. The two shifting operators are the left shift (<<) and the right shift (>>) operators.

Tell the students that the left shift operator allows shifting the bit positions towards the left side where the last bit is truncated and zero is added on the right.

Mention that the right shift operator allows shifting the bit positions towards the right side and the zero is added on the left.

Then, explain to the students that the code demonstrates the use of shift operators.

Use slide 37 to explain the students that in the code, the `Main()` method of the class **ShiftOperator** performs the shifting operations. Tell that the `Main()` method initializes the variable `num` to 100.

Also, tell the students that after shifting towards left, the value of `num` is doubled. Now, the variable `num` is initialized to 80.

Then, tell the students that after shifting towards right, the value of `num` is halved. You can refer to the figure in slide 37 that displays the output.

Slides 38 and 39

Understand the string concatenation operator.

String Concatenation Operator 1-2

- ◆ 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.
- ◆ The following code demonstrates the use of string concatenation operator.

Snippet

```
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);
    }
}
```

© Aptech Ltd. Building Applications Using C# /Session 3 38

String Concatenation Operator 2-2

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

The number 6 is even
Press any key to continue . . .

© Aptech Ltd. Building Applications Using C# /Session 3 39

In slide 38, explain to the students that 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.

Then, tell the students that the code demonstrates the use of string concatenation operator.

Use slide 39 to explain that in the code, the `Main()` method of the class **Concatenation** uses the construct to check whether a number is even, odd, or negative. Depending upon the condition, the code displays the output by using the string concatenation operator (+) to concatenate the strings with numbers.

You can refer to the figure in slide 39 that shows the use of string concatenation operator.

Slides 40 to 42

Understand the ternary or conditional operator.

The slide has a blue header bar with the title "Ternary or Conditional Operator 1-3". Below the title is a list of bullet points explaining the ternary operator. A "Syntax" section follows, showing the code structure. Below that is another list of bullet points explaining the components of the syntax.

Ternary or Conditional Operator 1-3

- ◆ 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.

Ternary or Conditional Operator 2-3

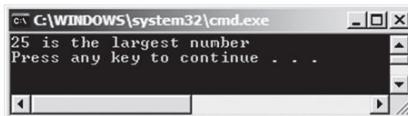
- The following code demonstrates the use of the ternary operator.

Snippet

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

Ternary or Conditional Operator 3-3

- In the code:
 - 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.
- The following figure shows the output of the example using ternary operator:



In slide 40, explain to the students that the `? :` is referred to as the conditional operator. It is generally used to replace the `if-else` constructs.

Explain that 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.

Explain that 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.

Explain the syntax shown on the slide. In the slide 41, explain to the students that the code demonstrates the use of the ternary operator.

Use slide 42 to explain the code. In the code, 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.

Explain that 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`.

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

You can refer to the figure in slide 42 that shows the output of the example using ternary operator.

Slide 43

Understand data conversions.


Data Conversions in C#

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

Example

- ◆ Consider the payroll system of an organization.
- ◆ The gross salary of an employee is calculated and stored in a variable of `float` type.
- ◆ The payroll department wants the salary amount as a whole number and thus, wants any digits after the decimal point of the calculated salary to be ignored.
- ◆ The programmer can achieve this using the typecasting feature of C#, which allows you to change the data type of a variable.
- ◆ C# supports two types of casting, namely **Implicit** and **Explicit**.
- ◆ Typecasting is mainly used to:
 - ◆ Convert a data type to another data type that belongs to the same or a different hierarchy.
 - ◆ Display the exact numeric output. For example, you can display exact quotients during mathematical divisions.
 - ◆ Prevent loss of numeric data if the resultant value exceeds the range of its variable's data type.



© Aptech Ltd. Building Applications Using C# /Session 3 43

In slide 43, explain that data conversion is performed in C# through casting, a mechanism to convert one data type to another.

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

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

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

Slide 44

Understand implicit conversions for C# data types - definition.

Implicit Conversions for C# Data Types – Definition

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

Snippet

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

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

© Aptech Ltd. Building Applications Using C# /Session 3 44

In slide 44, tell them that implicit typecasting refers to an automatic conversion of data types. This is done by the C# compiler. Implicit typecasting is done only when the destination and source data types belong to the same hierarchy.

Tell that the destination data type must hold a larger range of values than the source data type. Implicit conversion prevents the loss of data as the destination data type is always larger than the source data type. For example, if you have a value of `int` type, you can assign that value to the variable of `long` type.

Tell that the code displays an example of implicit conversion and explain the code. In the code, the compiler generates code that automatically converts the value in `valueOne` into a floating-point value before storing the result in `valueTwo`. Converting an integer value to a floating point value is safe.

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

Slide 45

Understand implicit conversions for C# data types - rules.

Implicit Conversions for C# Data Types – Rules

- ◆ Implicit typecasting is carried out automatically by the compiler.
- ◆ The C# compiler automatically converts a lower precision data type into a higher precision data type when the target variable is of a higher precision than the source variable.
- ◆ The following figure illustrates the data types of higher precision to which they can be converted:

```

graph TD
    byte[byte] --> ushort[ushort]
    ushort --> char[char]
    ushort --> uint:uint
    uint --> ulong[ulong]
    ulong --> float[float]
    float --> double[double]
    short[short] --> int[int]
    int --> long[long]
    sbyte[sbyte] --> short
  
```

© Aptech Ltd. Building Applications Using C# /Session 3 45

In slide 45, explain to the students that implicit typecasting is carried out automatically by the compiler.

Explain that the C# compiler automatically converts a lower precision data type into a higher precision data type when the target variable is of a higher precision than the source variable.

You can refer to the figure in slide 45 that illustrates the data types of higher precision to which they can be converted.

Slides 46 and 47

Understand explicit type conversion.

Explicit Type Conversion – Definition 1-2

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

Syntax

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

where,

- `target data type`: Is the resultant data type.
- `variable name`: Is the name of the variable, which is of the target data type.
- `target data type`: Is the target data type in parentheses.
- `source data type`: Is the data type which is to be converted.

© Aptech Ltd. Building Applications Using C# /Session 3 46

Explicit Type Conversion – Definition 2-2

- The following code displays the use of explicit conversion for calculating the area of a square:

Snippet

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

Output

```
Area of the square = 110
```

© Aptech Ltd. Building Applications Using C# /Session 3 47

In slide 46, explain to the students that explicit typecasting refers to changing a data type of higher precision into a data type of lower precision.

Give an example for this. Tell the students that using explicit typecasting, you can manually convert the value of float type into int type. This typecasting might result in loss of data.

This is because when you convert the float data type into the int data type, the digits after the decimal point are lost. Explain the syntax for explicit conversion as given on the slide. In slide 47, explain to the students that the code displays the use of explicit conversion for calculating the area of a square.

Slide 48

Understand the use of explicit type conversion - implementation.

Explicit Type Conversion – Implementation

- ◆ There are two ways to implement explicit typecasting in C# using the built-in methods:
 - ◆ **Using System.Convert class:** This class provides useful methods to convert any built-in data type to another built-in data type.
 - ◆ **Using ToString() method:** This method belongs to the Object class and converts any data type value into string.
- ◆ The code displays a float value as string using the ToString() method:

Snippet

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

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

Output 500.25

©Aptech Ltd. Building Applications Using C# /Session 3 48

Use slide 48 to explain the students that there are two ways to implement explicit typecasting in C# using the built-in methods. These are using:

- **System.Convert class:** This class provides useful methods to convert any built-in data type to another built-in data type.
For example, Convert.ToChar(float) method converts a float value into a char value.
- **ToString() method:** This method belongs to the Object class and converts any data type value into string.

Tell that the code displays a float value as string using the ToString() method and explain the code.

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

Slide 49

Understand boxing and unboxing.

Boxing and Unboxing 1-6

- ◆ Boxing is a process for converting a value type, like integers, to its reference type, like objects that is useful to reduce the overhead on the system during execution because all value types are implicitly of object type.
- ◆ To implement boxing, you need to assign the value type to an object.
- ◆ While boxing, the variable of type object holds the value of the value type variable which means that the object type has the copy of the value type instead of its reference.
- ◆ Boxing is done implicitly when a value type is provided instead of the expected reference type.
- ◆ The figure illustrates with an analogy the concept of boxing:

```

graph LR
    A[int num = 100] -- Boxing --> B[Reference Type]
    subgraph Value_Type [Value Type]
        A
    end
    subgraph Reference_Type [Reference Type]
        B
    end
  
```

© Aptech Ltd. Building Applications Using C# /Session 3 49

In slide 49, explain to the students that boxing is a process for converting a value type, like integers, to its reference type, such as objects.

Explain that this conversion is useful to reduce the overhead on the system during execution. This is because all value types are implicitly of Object type.

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

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

You can refer to the figure in slide 49 to illustrate with an analogy the concept of boxing.

Additional Information

For more information on boxing, refer the following link:

<http://msdn.microsoft.com/en-us/library/yz2be5wk.aspx>

Slide 50

Understand the syntax for boxing.

The slide is titled "Boxing and Unboxing 2-6". It contains the following content:

- The syntax for boxing is as follows:
Syntax:
object <instance of the object class> = <variable of value type>;
- where,
 - object: Is the base class for all value types.
 - instance of the object class: Is the name referencing the Object class.
 - variable of value type: Is the identifier whose data type is of value type.
- The following code demonstrates the use of implicit boxing:
Snippet:
int radius = 10;
double area;
area = 3.14 * radius * radius;
object boxed = area;
Console.WriteLine("Area of the circle = {0}",boxed);
- In the code:
 - Implicit boxing occurs when the value of double variable, **area**, is assigned to an object, **boxed**.

Output: Area of the circle = 314

© Aptech Ltd. Building Applications Using C# /Session 3 50

In slide 50, explain the syntax for boxing as given on the slide. Tell that the code demonstrates the use of implicit boxing. In the code, implicit boxing occurs when the value of double variable, **area**, is assigned to an object, **boxed**.

Slide 51

Understand the use of explicit boxing.

The slide has a blue header bar with the title "Boxing and Unboxing 3-6". Below the header, there is a bulleted list: "The following code demonstrates the use of explicit boxing:". A "Snippet" box contains the following C# code:

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

Below the code, a note says: "In the code: Explicit boxing occurs by casting the variable, **circumference**, which is assigned to object, **boxed**."

A "Output" box shows the console output: "Circumference of the circle = 28.26".

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd." on the left and "Building Applications Using C# /Session 3 51" on the right.

In slide 51, explain to the students that the code demonstrates the use of explicit boxing. In the code, explicit boxing occurs by casting the variable, **circumference**, which is assigned to object, **boxed**.

Slide 52

Understand unboxing.

Boxing and Unboxing 4-6

- ◆ Unboxing refers to converting a reference type to a value type.
- ◆ The stored value inside an object is unboxed to the value type.
- ◆ The object type must be of the destination value type. This is explicit conversion, without which the program will give an error.
- ◆ The following figure illustrates with an analogy the concept of unboxing:

- ◆ The syntax for unboxing is as follows:

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

- ◆ where,
 - ◆ target value type: Is the resultant data type.
 - ◆ variable name: Is the name of the variable of value type.
 - ◆ target value type: Is the resultant value type in parentheses.
 - ◆ object type: Is the reference name of the Object class.

© Aptech Ltd. Building Applications Using C# /Session 3 52

In slide 52, explain that unboxing refers to converting a reference type to a value type. The stored value inside an object is unboxed to the value type. The object type must be of the destination value type. This is explicit conversion, without which the program will give an error.

You can refer to the figure in slide 52 to illustrate with an analogy the concept of unboxing.

Tell the syntax for unboxing as given on the slide.

Additional Information

For more information on unboxing, refer the following link:

<http://msdn.microsoft.com/en-us/library/yz2be5wk.aspx>

Slide 53

Understand use of unboxing while calculating the area of the rectangle.

The slide has a blue header bar with the title "Boxing and Unboxing 5-6". Below the title, there is a bulleted list and a code snippet box. The code snippet contains C# code for calculating the area of a rectangle and then unboxing it. At the bottom of the slide, there is a footer bar with the copyright information "© Aptech Ltd." and the page number "Building Applications Using C# /Session 3 53".

◆ The following code demonstrates the use of unboxing while calculating the area of the rectangle:

Snippet

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

◆ In the code:

- ❖ Boxing is done when the value of the variable **area** is assigned to an object, **boxed**. However, when the value of the object, **boxed**, is to be assigned to **num**, unboxing is done explicitly because **boxed** is a reference type and **num** is a value type.

In slide 53, explain to the students that the code demonstrates the use of unboxing while calculating the area of the rectangle.

Explain the code. In the code, boxing is done when the value of the variable **area** is assigned to an object, **boxed**. However, when the value of the object, **boxed**, is to be assigned to **num**, unboxing is done explicitly because **boxed** is a reference type and **num** is a value type.

Slide 54

Understand how boxing and unboxing occur.

Boxing and Unboxing 6-6

- The following code demonstrates how boxing and unboxing occur:

Snippet

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

- In the code:
 - The `Main()` method calculates the square of the specified value.
 - Boxing occurs when the variable `num`, which is of value type, is passed to the `Square()` method whose input parameter, `inum`, is of reference type.
 - Unboxing occurs when this reference type variable, `inum`, is converted to value type, `int`, before its square is returned to variable `result`.
- The following figure shows the output of the code:

© Aptech Ltd. Building Applications Using C# /Session 3 54

In slide 54, explain the code to the students that demonstrates how boxing and unboxing occur.

Explain the code to them. In the code, the `Main()` method calculates the square of the specified value. Boxing occurs when the variable `num`, which is of value type, is passed to the `Square()` method whose input parameter, `inum`, is of reference type. Unboxing occurs when this reference type variable, `inum`, is converted to value type, `int`, before its square is returned to variable `result`.

You can refer to the figure in slide 54 that shows the output of the code.

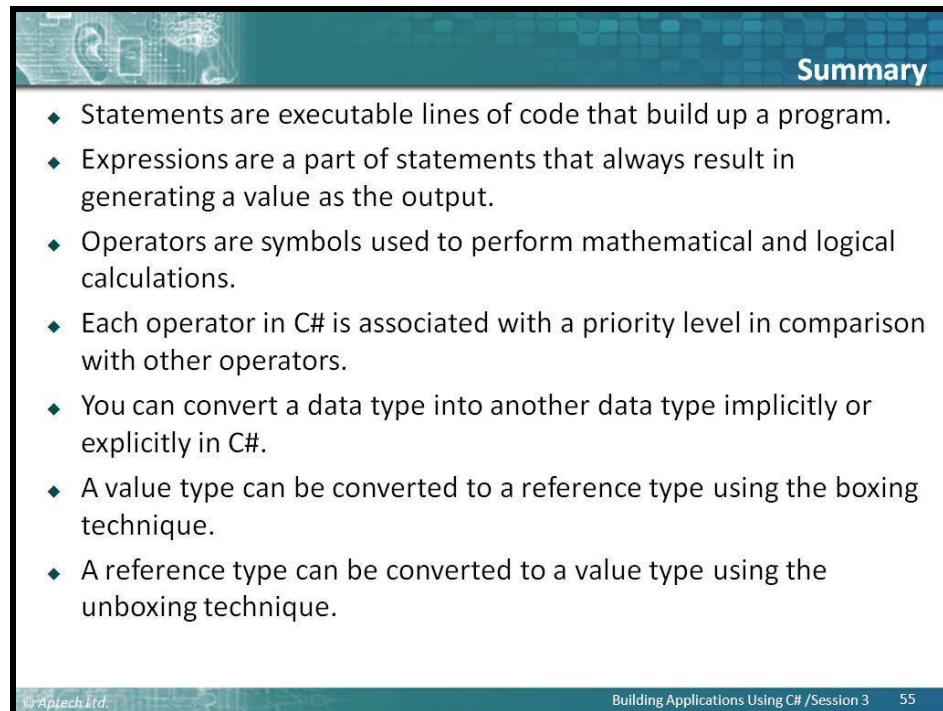
Additional Information

For more information on boxing and unboxing, refer the following link:

<http://msdn.microsoft.com/en-us/library/yz2be5wk.aspx>

Slide 55

Summarize the session.



The slide features a decorative header with a blue gradient and a gear icon. The word "Summary" is centered in a white box. Below the header is a list of nine bullet points. At the bottom of the slide is a footer bar with the text "©Aptech Ltd." on the left and "Building Applications Using C# /Session 3 55" on the right.

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

In slide 55, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell the students that:

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

3.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a question to students to search additional information, such as:

1. What are special operators?
2. Explain the break statement.
3. Describe the purpose of jump statements.

Session 4 - C# Programming Constructs

4.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the previous session for a review. Prepare the background knowledge/summary to be discussed with students in the class. The summary of the previous session is as follows:

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

Familiarize yourself with the topics of this session in-depth.

Here, you can ask students the key topics they can recall from previous session. Prepare a question or two which will be a key point to relate the current session objectives. Ask them to explain types of operators in C# and explain statements and expressions. Also ask about how to perform data conversions in C#.

4.1.1 Objectives

By the end of this session, the learners will be able to:

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

4.1.2 Teaching Skills

To teach this session successfully, you must know about various selection constructs in C#. You should be aware of basic and advanced concepts of loop constructs.

You should teach the concepts in the theory class using the concepts, tables, and codes provided.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced the concepts of selection constructs and loop constructs. They will also learn about the jump statements in C#.

4.2 In-Class Explanations**Slide 3**

Understand the selection constructs.

Selection Constructs

- ◆ A selection construct:
 - ❖ Is a programming construct supported by C# that controls the flow of a program.
 - ❖ Executes a particular block of statements based on a boolean condition, which is an expression returning true or false.
 - ❖ Is referred to as a decision-making construct.
 - ❖ Allow you to take logical decisions about executing different blocks of a program to achieve the required logical output.
- ◆ C# supports the following decision-making constructs:
 - ❖ if...else
 - ❖ if...else...if
 - ❖ Nested if
 - ❖ switch...case

```

graph TD
    Expression1[Expression 1] --> True[True]
    Expression1 --> False[False]
  
```

© Aptech Ltd. Building Applications Using C# / Session 4 3

In slide 3, introduce a selection construct. Tell the students that 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.

Explain to them that the selection construct is referred to as a decision-making construct and it allow the students to take logical decisions about executing different blocks of a program to achieve the required logical output.

Explain that the C# supports the decision-making constructs such as `if...else`, `if...else...if`, Nested `if`, and `switch...case`.

Slides 4 to 6

Understand the `if` statement.

The slide has a blue header bar with the title 'The if Statement 1-3'. Below the title is a bulleted list of points about the `if` statement. A 'Syntax' section contains a code snippet for the `if` statement. Below the syntax, there is a note about where the condition and statements are defined.

The if Statement 1-3

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

Syntax

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

where,

- ◆ condition: Is the boolean expression.
- ◆ statements: Are set of executable instructions executed when the boolean expression return true.

© Aptech Ltd. Building Applications Using C# / Session 4 4

The if Statement 2-3

- The following figure displays an example of the if construct:

```

graph TD
    START([START]) --> Init[num = -4]
    Init --> Cond{num < 0 ?}
    Cond -- No --> STOP([STOP])
    Cond -- Yes --> Print[PRINT "The number is negative"]
    Print --> STOP
  
```

©Aptech Ltd.

Building Applications Using C# / Session 4

5

The if Statement 3-3

- The following code displays whether the number is negative using the if statement:

Snippet

```

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

- In the code:
 - 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.

©Aptech Ltd.

Building Applications Using C# / Session 4

6

In slide 4, explain the if statement to the students.

Tell that 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.

Mention that if the condition evaluates to false, the block of statements following the `if` statement is ignored and the statement after the block is executed.

Then, tell that the `if` statement syntax to the students and tell them that the condition is the boolean expression and the statements are set of executable instructions executed when the boolean expression returns true in the syntax.

Refer to the figure in slide 5 that displays the flowchart example of the `if` statement. Understand the code of `if` statement. In slide 6, tell that the code displays whether the number is negative using the `if` statement. Then, explain the code to the students.

Tell the students that in the code, `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. Mention that the condition evaluates to `true` and the output “The number is negative” is displayed in the console window.

Slides 7 and 8

Understand the `if...else` construct.

The `if...else` Construct 1-2

- In some situations, it is required to define an action for a false condition by using an `if...else` construct.
- The `if...else` construct starts with the `if` block followed by an `else` block and the `else` block starts with the `else` keyword followed by a block of statements.
- If the condition specified in the `if` statement evaluates to false, the statements in the `else` block are executed.
- The following is the syntax for the `if...else` statement:

Syntax
<pre>if (condition) { // one or more statements; } else { //one or more statements; }</pre>

```

graph TD
    START([START]) --> SetNum[num = 10]
    SetNum --> Cond{num < 0 ?}
    Cond -- No --> PrintPos[PRINT "The number is Positive"]
    PrintPos --> STOP([STOP])
    Cond -- Yes --> PrintNeg[PRINT "The number is negative"]
    PrintNeg --> STOP
  
```

© Aptech Ltd. Building Applications Using C# / Session 4 7

The slide has a blue header bar with the title 'The if...else Construct 2-2'. Below the header, there is a list of bullet points and a code snippet.

- The following code displays whether a number is positive or negative using the if...else construct:

Snippet

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

Output

The number is positive.

© Aptech Ltd. Building Applications Using C# / Session 4 8

In slide 7, explain to the students that 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.

Tell the students that 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.

Explain the syntax for the `if...else` statement. You can refer figure from slide 8 that demonstrates the `if-else` construct with an example.

In slide 8, tell that the code displays whether a number is positive or negative using the `if...else` construct.

Tell the students that in the code, `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.

Slides 9 to 11

Understand the if...else...if construct.

The if...else...if Construct 1-3

- ◆ The if...else...if construct allows you to check multiple conditions to execute a different block of code for each condition.
- ◆ It is also referred to as if-else-if ladder.
- ◆ The construct starts with the if statement followed by multiple else if statements followed by an optional else block.
- ◆ The conditions specified in the if...else...if construct are evaluated sequentially.
- ◆ The execution starts from the if statement. If a condition evaluates to false, the condition specified in the following else...if statement is evaluated.
- ◆ The syntax for the if...else...if construct is as follows:

Syntax

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

© Aptech Ltd. Building Applications Using C# / Session 4 9

The if...else...if Construct 2-3

- ◆ The following figure displays an example of the if...else...if construct:

```

graph TD
    START([START]) --> Init[num=13]
    Init --> Cond1{num < 0 ?}
    Cond1 -- No --> Cond2{num % 2 == 0 ?}
    Cond2 -- Yes --> Even[Print "The number is even"]
    Cond2 -- No --> Odd[Print "The number is odd"]
    Cond1 -- Yes --> Neg[Print "The number is negative"]
    Even --> STOP([STOP])
    Odd --> STOP
    Neg --> STOP
  
```

© Aptech Ltd. Building Applications Using C# / Session 4 10

The if...else...if Construct 3-3

- The following code displays whether a number is negative, even, or odd using the if...else...if construct:

Snippet

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

- In the code:
 - 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.

© Aptech Ltd. Building Applications Using C# / Session 4 11

In slide 9, explain the if...else...if construct. Tell the students that 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.

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

Then, explain the syntax for the if...else...if construct. You can refer to the figure in slide 10 that displays an example of the if...else...if construct.

In slide 11, tell that the code displays whether a number is negative, even, or odd using the if...else...if construct.

Tell them that in the code, 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.

Explain to the students that 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.

Slides 12 to 14

Understand the features of the nested `if` construct.

Nested if Construct 1-3

- Following are the features of the 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.

© Aptech Ltd.

Building Applications Using C# / Session 4 12

Nested if Construct 2-3

- The following is the syntax for the nested `if` construct:

`Syntax`

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

- The following figure displays an example of the nested `if` construct:

```

graph TD
    START([START]) --> Init["yrsOfService=3  
salary=1500  
bonus=0"]
    Init --> Cond1{yrsOfService < 5 ?}
    Cond1 -- No --> Bonus700[bonus=700]
    Bonus700 --> STOP([STOP])
    Cond1 -- Yes --> Cond2{salary < 500 ?}
    Cond2 -- No --> Bonus200[bonus=200]
    Bonus200 --> STOP
    Cond2 -- Yes --> Bonus100[bonus=100]
    Bonus100 --> STOP
  
```

© Aptech Ltd.

Building Applications Using C# / Session 4 13

Nested if Construct 3-3

- The following figure displays the bonus amount using the nested `if` construct:

Snippet

```

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 the code:
 - `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.

Output Bonus amount: 200

© Aptech Ltd. Building Applications Using C# / Session 4 14

In slide 12, explain the nested `if` construct.

Explain the student that 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.

Tell the students that 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`.

Mention that each inner `if` statement is executed only if the condition in its previous inner `if` statement is `true`.

In slide 13, explain the syntax for the nested `if` construct. You can refer figure from slide 13 that displays nested `if` construct.

Tell them that when they wish to do something when only one condition is satisfied, then we can use `if....else` statement.

When there are two actions to be taken based on a certain condition then we can use, `if....else` statement.

When multiple actions are to be taken by checking multiple possibilities, then we can use `if....else....if` statements.

Also tell them that use of nested `if` statements is also possible.

In slide 14, tell that the code displays the bonus amount using the nested `if` construct. Then, tell the students that in the code, `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.

Mention that 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.

Slide 15

Understand the `switch...case` construct.

switch...case Construct 1-5

- ◆ A program is difficult to comprehend when there are too many `if` statements representing multiple selection constructs.
- ◆ To avoid using multiple `if` statements, in certain cases, the `switch...case` approach can be used as an alternative.
- ◆ The `switch...case` statement is used when a variable needs to be compared against different values.
- ◆ The following figure depicts the `switch...case` as a flow chart.

```

graph TD
    D{ } --> B1[ ]
    B1 --> B2[ ]
    B2 --> B3[ ]
    B3 --> F[ ]
  
```

© Aptech Ltd. Building Applications Using C# / Session 4 15

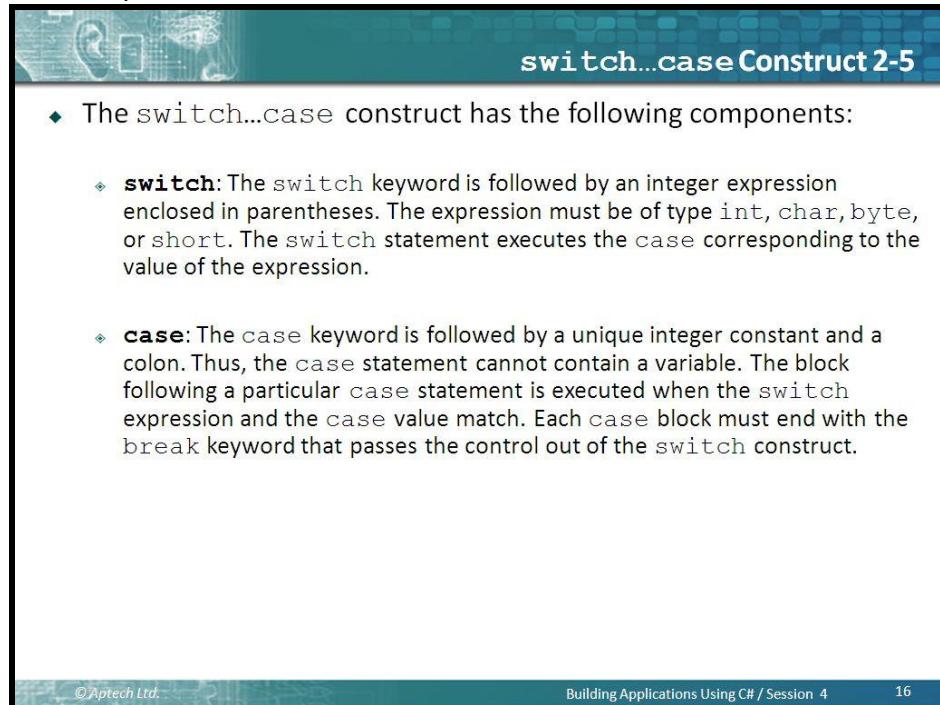
In slide 15, explain the `switch...case` construct to the students.

Tell the students that 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.

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

Slides 16 and 17

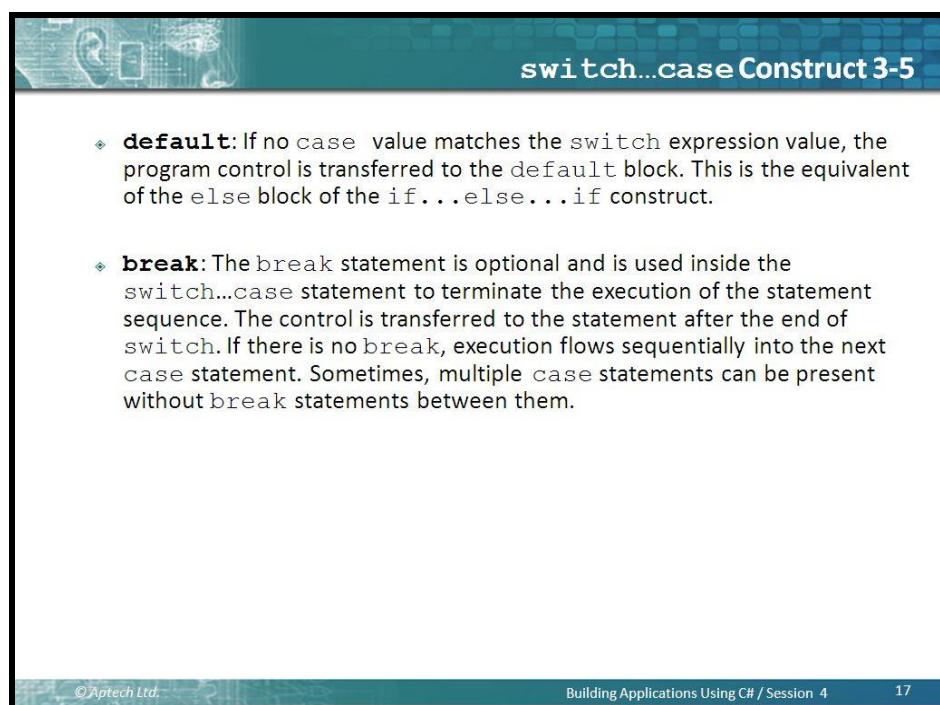
Understand the components of switch...case construct.



switch...case Construct 2-5

- ◆ The switch...case construct has the following components:
 - ◆ **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.

© Aptech Ltd. Building Applications Using C# / Session 4 16



switch...case Construct 3-5

- ◆ **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 `case` statements can be present without `break` statements between them.

© Aptech Ltd. Building Applications Using C# / Session 4 17

In slide 16, tell the students that the switch...case construct has the components such as `switch`, `case`, `default`, and `break`.

Explain the `switch` component.

Tell the students that 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.

Explain the `case` component.

Explain to the students that 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.

Use slide 17, explain the `default` and the `break` components.

Explain the `default` component.

Tell the students that 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.

Explain the `break` component. Explain to the students that 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 `case` statements can be present without `break` statements between them.

In C#, a `switch....case` construct needs a `break` statement after each condition. Even the `default case` needs a `break` statement at the end.

Slides 18 and 19

Understand the code that displays the day of the week using the `switch...case` construct.

switch...case Construct 4-5

- The following code displays the day of the week using the `switch...case` construct:

Snippet

```

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;
}

```

© Aptech Ltd.

Building Applications Using C# / Session 4

18

switch...case Construct 5-5

- In the code:
 - `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

© Aptech Ltd.

Building Applications Using C# / Session 4

19

In slide 18, tell that the code displays the day of the week using the `switch...case` construct.

Use slide 19 to tell that the students that in the code, **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.

Mention that when the `break` statement is encountered, the control passes out of the `switch...case` construct.

Slides 20 to 23

Understand the nested `switch...case` construct.

Nested switch...case Construct 1-4

- ◆ 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.
- ◆ The following code demonstrates the use of nested `switch`:

Snippet

```
namespace Samsung
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());
```

© Aptech Ltd. Building Applications Using C# / Session 4 20

Nested switch...case Construct 2-4

```

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;
    switch (input)
    {
        case 1:
            result = numOne + numTwo;
            break;

        case 2:
            result = numOne - numTwo;
            break;
        case 3:
            result = numOne * numTwo;
            break;
        case 4:
            Console.WriteLine("Do you want to calculate
the quotient or remainder?");
            Console.WriteLine("(1) Quotient");
            Console.WriteLine("(2) Remainder");
            int choice = Convert.ToInt32
            (Console.ReadLine());
    }
}

```

© Aptech Ltd.

Building Applications Using C# / Session 4 21

Nested switch...case Construct 3-4

```

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);
}
}

```

© Aptech Ltd.

Building Applications Using C# / Session 4 22

Nested switch...case Construct 4-4

- ◆ In the code:
 - ❖ The user is asked to choose the desired arithmetical 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.
- ◆ The following figure demonstrates the nested switch:

© Aptech Ltd.

Building Applications Using C# / Session 4 23

In slide 20, tell that the 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.

Mention that the case constants of the inner switch...case construct can have values that are identical to the case constants of the outer construct.

In slides 21 and 22, tell that the code demonstrates the use of nested switch.

In slide 23, explain the code to the students.

Tell that the user is asked to choose the desired arithmetical operation in the code. 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.

Explain to them that 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.

You can refer figure from slide 21 that demonstrates the nested switch.

In-Class Question:

After you finish explaining the Nested switch...case Construct, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which are the components of switch...case construct?

Answer:

The components of switch...case construct are switch, case, default, and break.

Slide 24

Understand the no-fall-through rule.


No-Fall-Through Rule 1-3

- ◆ Following are the 'no-fall-through' rules:

In C#, the flow of execution from one case statement is not allowed to continue to the next case statement and is referred to as the 'no-fall-through' rule of C#.

Thus, the list of statements inside a case block generally ends with a break or a goto statement, which causes the control of the program to exit the switch...case construct and go to the statement following the construct.

The last case block (or the default block) also needs to have a statement like break or goto to explicitly take the control outside the switch...case construct.

C# introduced the no fall-through rule to allow the compiler to rearrange the order of the case blocks for performance optimization.
- ◆ 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.

©Aptech Ltd. Building Applications Using C# / Session 4 24

In slide 24, explain the 'no-fall-through' rule to the students.

Tell the students that 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.

Explain that 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#.

Explain to the students that 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. Mention that the

last case block (or the default block) also needs to have a statement such as break or goto to explicitly take the control outside the switch...case construct.

Tell them that the 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.

Explain them that 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.

Slides 25 and 26

Understand the multiple case statements can be made to execute the same code sequence.

No-Fall-Through Rule 2-3

- A multiple case statement can be made to execute the same code sequence, as shown in the following code:

Snippet

```
namespace Samsung
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":
```

© Aptech Ltd. Building Applications Using C# / Session 4 25

No-Fall-Through Rule 3-3

```

        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 the code:
 - ❖ 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.
- ◆ The figure shows the output of multiple `case` statements:

© Aptech Ltd.

Building Applications Using C# / Session 4 26

In slide 25, tell that the multiple `case` statements can be made to execute the same code sequence.

Use slide 26 to tell that in the code, 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.

Also, tell that 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.

You can refer figure from slide 26 that shows the output of multiple `case` statements.

Slide 27

Understand the loop constructs.

The slide has a header 'Loop Constructs' with a purple circular icon. On the left is a list of points about loops, and on the right is a flowchart and a Venn diagram.

- 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 such as:
 - The while loop
 - The do..while loop
 - The for loop
 - The foreach loop

Flowchart:

```

graph TD
    Start(( )) --> Cond{Condition Expression}
    Cond -- True --> Execute[Execute Body of loop]
    Execute --> Cond
    Cond -- False --> Exit[Exit Loop]
  
```

Venn Diagram:

A Venn diagram with three overlapping circles labeled 'while' (purple), 'do-while' (orange), and 'for' (pink).

© Aptech Ltd. Building Applications Using C# / Session 4 27

In slide 27, explain the loop constructs.

Tell the students that the loop executes 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.

Mention that in software programming, a loop construct contains a condition that helps the compiler identify the number of times a specific block will be executed.

Then, explain that 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.

Explain to the students that C# supports four types of loop constructs such as:

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

Tell the students that they can use any of the loop mentioned here. All the loops are used to repeat certain steps n number of times. Explain the main difference between the `do...while` and `while` loop. Tell them that `do...while` is used when we need to

execute the block of code at least once even if the given condition evaluates to false. The loops evaluate because the condition is checked after execution of the loop block whereas in case of while loop, first the condition is checked and the loop is executed only when the condition evaluates to true.

Slides 28 to 30

Understand the while loop.

The slide has a blue header bar with the title 'The while Loop 1-3'. Below the title is a bulleted list of six points about the while loop. A 'Syntax' section is shown with a code example, followed by a 'where:' section with a note about the 'condition'. The footer contains copyright information and page numbers.

The while Loop 1-3

- ◆ The while loop is used to execute a block of code repetitively as long as the condition of the loop remains true.
- ◆ The while loop consists of the while statement, which begins with the while keyword followed by a boolean condition.
- ◆ If the condition evaluates to true, the block of statements after the while statement is executed.
- ◆ After each iteration, the control is transferred back to the while statement and the condition is checked again for another round of execution.
- ◆ When the condition is evaluated to false, the block of statements following the while statement is ignored and the statement appearing after the block is executed by the compiler.
- ◆ The following is the syntax of the while loop:

Syntax

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

where:

- ◆ condition: Specifies the boolean expression.

©Aptech Ltd. Building Applications Using C# / Session 4 28

The while Loop 2-3

- The following figure depicts an example of a `while` loop using a flowchart:

```

graph TD
    START([START]) --> num1[num=1]
    num1 --> cond1{num<=11?}
    cond1 -- No --> STOP([STOP])
    cond1 -- Yes --> cond2{num%2==0?}
    cond2 -- No --> STOP
    cond2 -- Yes --> print[PRINT num]
    print --> numplus1[num=num+1]
    numplus1 --> cond1
  
```

- The following code displays even numbers from 1 to 10 using the `while` loop:

Snippet

```

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

The while Loop 3-3

- In the following code:
 - `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
  
```

In slide 28, explain the `while` loop.

Tell that 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.

Explain them that after each iteration, the control is transferred back to the `while` statement and the condition is checked again for another round of execution. Mention that 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.

Then, explain the syntax of the `while` loop to the students and tell them that the condition specifies the boolean expression.

Use slides 29 and 30 to refer figure that depicts an example of a `while` loop using a flowchart and also explains the code that displays even numbers from 1 to 10 using the `while` loop.

Then, explain the code. Tell that in the code, `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.

Explain that 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.

Mention that 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.

Tell the students that when the value of `num` becomes 12, the `while` loop terminates as the loop condition becomes false.

Tips:

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.

Slide 31

Understand the nested `while` loop.

Nested while Loop

- ◆ A `while` loop can be created within another `while` loop to create a nested `while` loop structure.
- ◆ The following code demonstrates the use of nested `while` loops to create a geometric pattern:

Snippet

```
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 the code:
 - ◆ A pattern of a right-angled triangle is created using the asterisk (*) symbol. This is done using the nested `while` loop.

© Aptech Ltd. | Building Applications Using C# / Session 4 | 31

In slide 31, tell the students that a `while` loop can be created within another `while` loop to create a nested `while` loop structure.

Then, tell that the code demonstrates the use of nested `while` loops to create a geometric pattern. Tell that a pattern of a right-angled triangle is created using the asterisk (*) symbol. This is done using the nested `while` loop. You can refer figure from slide 31 that shows the nested `while` loop.

Slides 32 and 33

Understand the `do-while` loop.

The do-while Loop 1-2

- The `do-while` loop is similar to the `while` loop; however, it is always executed at least once without the condition being checked.
- The loop starts with the `do` keyword and is followed by a block of executable statements.
- The `while` statement along with the condition appears at the end of this block.
- The statements in the `do-while` loop are executed as long as the specified condition remains true.
- When the condition evaluates to false, the block of statements after the `do` keyword are ignored and the immediate statement after the `while` statement is executed.
- The following is the syntax of the `do-while` loop:

Syntax	<pre>do { // one or more statements; } while (condition);</pre>
--------	---

```

graph TD
    START([START]) --> num1[num=num=1]
    num1 --> cond1{num%2=0 ?}
    cond1 -- No --> STOP([STOP])
    cond1 -- Yes --> print1[/PRINT num/]
    print1 --> numplus1[num=num+1]
    numplus1 --> cond2{num<=11 ?}
    cond2 -- Yes --> print1
    cond2 -- No --> STOP
  
```

©Aptech Ltd. Building Applications Using C# / Session 4 32

The do-while Loop 2-2

- The following code displays even numbers from 1 to 10 using the `do-while` loop:

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

- In the code:
 - `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
```

© Aptech Ltd. Building Applications Using C# / Session 4 33

In slide 32, explain the `do-while` loop.

Explain to the students that the `do-while` loop is similar to the `while` loop.

Mention that 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.

Tell the students that 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.

Then, explain the syntax of the `do-while` loop.

You can refer figure from slide 33 that shows an example of a `do-while` loop using a flowchart.

In slide 33, tell that the code displays even numbers from 1 to 10 using the `do-while` loop. Then, explain the code.

Tell the students that in the code, **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. Mention that when the value of **num** becomes 12, the **do-while** loop terminates.

Tips:

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.

Slides 34 to 36

Understand **for** loop.

The for Loop 1-3

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

Syntax	<pre>for (initialization; condition; increment/decrement) { // one or more statements; }</pre>
---------------	--

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.

© Aptech Ltd. Building Applications Using C# / Session 4 34

◆ The working of the `for` loop can be depicted using the following flowchart:

```

graph TD
    START([START]) --> Init[/num=1<br/>num=num+1/]
    Init --> Cond{num≤11?}
    Cond -- No --> STOP([STOP])
    Cond -- Yes --> Decision{num%2=0?}
    Decision -- No --> Init
    Decision -- Yes --> Print[/PRINT num/]
    Print --> Init
    
```

© Aptech Ltd. Building Applications Using C# / Session 4 35

◆ The following code displays even numbers from 1 to 10 using the `for` loop:

Snippet

```

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

```

◆ In the code:

- ◆ `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.

© Aptech Ltd. Building Applications Using C# / Session 4 36

In slide 34, tell the students that the `for` statement is similar to the `while` statement in its function.

Tell them that the statements within the body of the loop are executed as long as the condition is true. The condition is checked before the statements are executed.

Then, explain the syntax of the `for` loop where, the initialization initializes the variable(s) that will be used in the condition and the condition comprises the condition that is tested before the statements in the loop are executed.

Explain that the 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.

Use slide 35 to refer figure that depicts an example of a `for` loop using a flowchart.

Tell the students that many developers prefer `for` loop to other loops because it allows initialization, condition, and increment or decrement part to be mentioned at one location only. Even understanding this loop also becomes easy, since all the components of a loop are present at one place. In slide 36, tell that the code displays the even numbers from 1 to 10 using the `for` loop.

Explain the code to the students. Tell that in the code, `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.

Slide 37

Understand nested `for` loops.

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.
- The following code demonstrates a 2X2 matrix using nested `for` loops:

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

Snippet

- In the code:
 - The 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.

© Aptech Ltd. Building Applications Using C# / Session 4 37

In slide 37, explain the nested `for` loops.

Tell that 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.

Mention that 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.

Then, tell that the code demonstrates a 2X2 matrix using nested `for` loops.

Explain the code. Tell that in the code, 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.

Explain that 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.

Slide 38

Understand the `for` loop with multiple loop control variables.

The for Loop with Multiple Loop Control Variables

- ◆ The `for` loop allows the use of multiple variables to control the loop.
- ◆ The following code demonstrates the use of the `for` loop with two variables:

Snippet

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

- ◆ In the code:
 - ◆ 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.
- ◆ The following figure shows the use of the `for` loop:

Number	Square	Cube
1		1
2	4	
3		27
4	16	
5		125
6	36	
7		343
8	64	
9		729
10	100	

Press any key to continue . . .

In slide 38, explain the `for` loop with multiple loop control variables to the students.

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

Tell that the code demonstrates the use of the `for` loop with two variables.

Explain that 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. You can refer figure from slide 38 that shows the use of the `for` loop.

Slide 39

Understand the `for` loop with missing portions.


The for Loop with Missing Portions 1-5

- ◆ 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.
- ◆ The following code demonstrates a `for` loop that leaves out or omits the increment/decrement portion of the loop definition:

Snippet

```
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);
    }
}
```

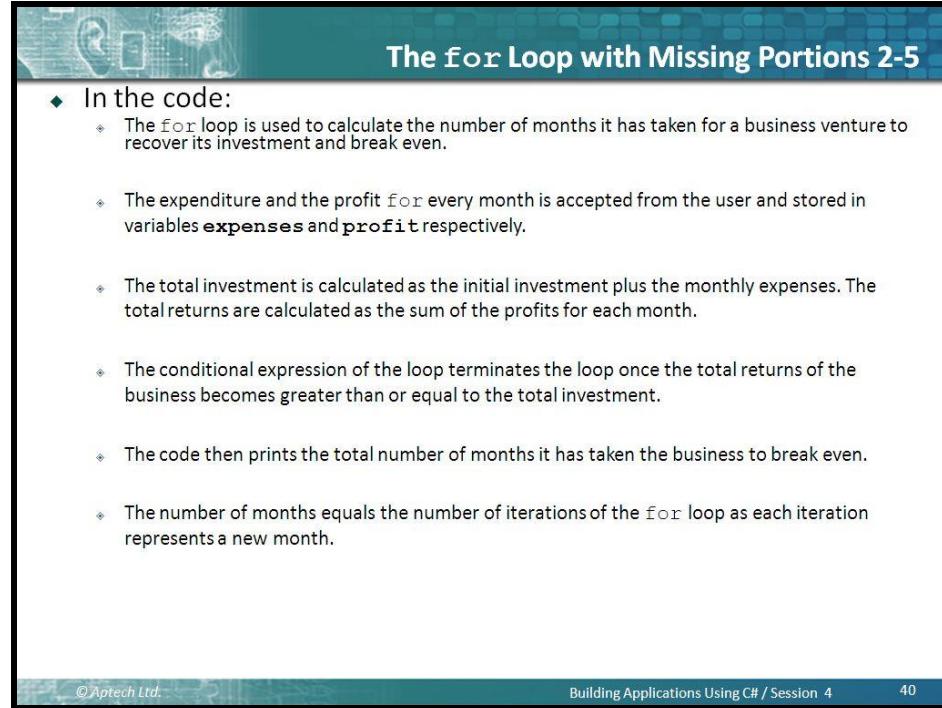
©Aptech Ltd.
Building Applications Using C# / Session 4
39

In slide 39, explain the `for` loop with missing portions.

Tell that the `for` loop definition can be divided into three portions, the initialization, the conditional expression, and the increment/decrement portion. Tell that 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.

Slide 40

Understand the code that demonstrates a `for` loop that leaves out or omits the increment or decrement portion of the loop definition.



The For Loop with Missing Portions 2-5

- ◆ In the code:
 - ❖ 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 conditional expression of the loop terminates the loop once the total returns of the business becomes greater than or equal to the total investment.
 - ❖ The code then prints the total number of months it has taken the business to break even.
 - ❖ The number of months equals the number of iterations of the `for` loop as each iteration represents a new month.

© Aptech Ltd. Building Applications Using C# / Session 4 40

In slide 40, explain the code. Explain that in the code, 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.

Mention that 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`.

Tell that the conditional expression of the loop terminates the loop once the total returns of the business becomes greater than or equal to the total investment. The code then prints the total number of months it has taken the business to break even. The number of months equals the number of iterations of the `for` loop as each iteration represents a new month.

Slides 41 to 43

Understand the code that demonstrates a `for` loop that omits the initialization portion as well as the increment/decrement portion of the loop definition.



The `for` Loop with Missing Portions 3-5

- The following code demonstrates a `for` loop that omits the initialization portion as well as the increment/decrement portion of the loop definition:

Snippet

```
int investment = 1000;
int returns = 0;
for (; returns < investment; )
{
    // for loop statements
    ...
}
```

- In the code:
 - 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.
- The following code demonstrates the use of an infinite `for` loop:

Snippet

```
using System;
class Summation
{
    static void Main(string[] args)
    {
        char c;
        int numOne;
        int numTwo;
        int result;
        for ( ; ; )
```

© Aptech Ltd. Building Applications Using C# / Session 4 41



The `for` Loop with Missing Portions 4-5

```
{
    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 the code:
 - 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.

© Aptech Ltd. Building Applications Using C# / Session 4 42

The for Loop with Missing Portions 5-5

- The following figure shows the use of an infinite for loop:

```

C:\WINDOWS\system32\cmd.exe
Enter number one
10
Enter number two
20
Result of Addition: 30
Do you wish to continue [Y / N]
y
Enter number one
60
Enter number two
50
Result of Addition: 110
Do you wish to continue [Y / N]
n
Press any key to continue . . .

```

©Aptech Ltd.

Building Applications Using C# / Session 4

43

In slide 41 tell that the code demonstrates a `for` loop that omits the initialization portion as well as the increment/decrement portion of the loop definition.

Tell that in the code 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.

Mention that 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.

Use slides 41 and 42 to tell that the code demonstrates the use of an infinite `for` loop.

Tell that in the code, 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.

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

You can refer figure from slide 43 that shows the use of an infinite `for` loop.

Slide 44

Understand the `for` loop without a body.

The `for` Loop without a Body

- ◆ In C#, a `for` loop can be created without a body.
- ◆ Such a loop is created when the operations performed within the body of the loop can be accommodated within the loop definition itself.
- ◆ The following code demonstrates the use of a `for` loop without a body:

Snippet

```
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 the code:
 - ◆ The process of calculating the factorial of a user-given number is done entirely in the `for` loop definition; the loop has no body.
- ◆ The following figure shows the `for` loop without a body:

© Aptech Ltd. Building Applications Using C# / Session 4 44

In slide 44, explain the `for` loop without a body.

Tell that 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.

Then, tell the code that demonstrates the use of a `for` loop without a body and explain it. Explain that in the code, the process of calculating the factorial of a user-given number is done entirely in the `for` loop definition; the loop has no body.

You can refer slide 44 that shows the `for` loop without a body.

With this slide, you will finish explaining loop constructs.

In-Class Question:

After you finish explaining the `for` loop without a body, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which are the four types of loop constructs that are supported by C#?

Answer:

The four types of loop constructs that are supported by C# are the `while` loop, the `do..while` loop, the `for` loop, and the `foreach` loop.

Slide 45

Understand declaring loop control variables within the `for` loop definition.

Declaring Loop Control Variables in the Loop Definition 1-3

- ◆ The loop control variables are often created for loops such as the `for` loop. Once the loop is terminated, there is no further use of these variables. In such cases, these variables can be created within the initialization portion of the `for` loop definition.
- ◆ The following is the syntax for declaring the loop control variable within the loop definition:

Syntax

```
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.

© Aptech Ltd. Building Applications Using C# / Session 4 45

In slide 45, explain the declaring loop control variables within the `for` loop definition.

Explain that the loop control variables are often created for loops such as the `for` loop. Once the loop is terminated, there is no further use of these variables. In such cases, these variables can be created within the initialization portion of the `for` loop definition.

Explain the syntax to the students that is used for declaring the loop control variable within the loop definition. In the syntax, `datatype` specifies the data type of the elements in the list. The `identifier` is an appropriate name for the collection of elements and the `list` specifies the name of the list.

Slides 46 and 47

Understand the code that displays the employee names using the `foreach` loop.

Declaring Loop Control Variables in the Loop Definition 2-3

- The following figure displays the employee names using the `foreach` loop:

Snippet

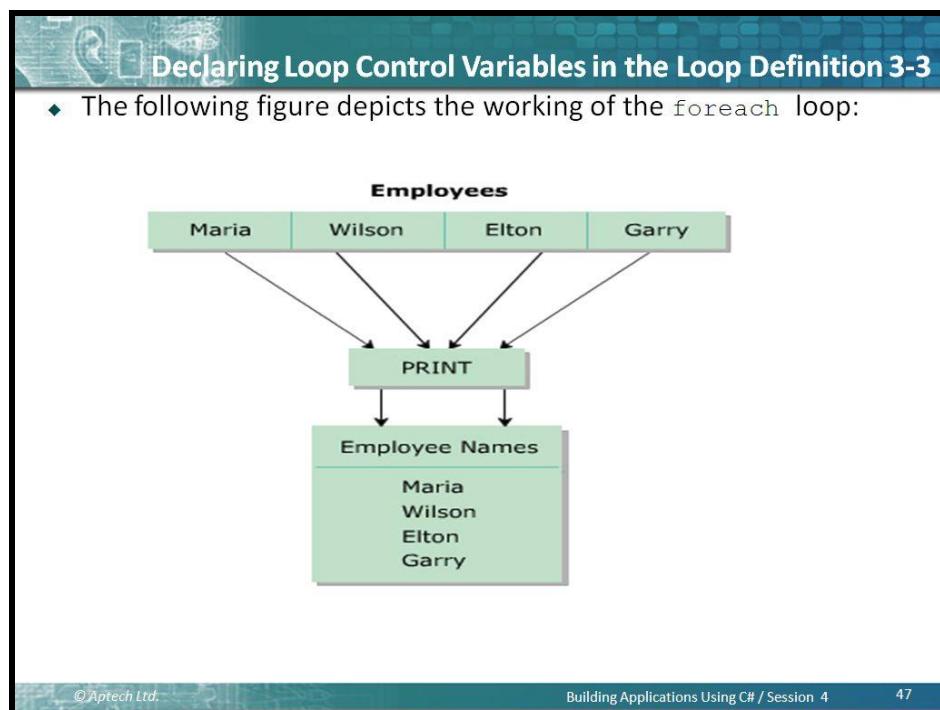
```
string[] employeeNames = { "Maria", "Wilson", "Elton", "Garry" };
Console.WriteLine("Employee Names");
foreach (string names in employeeNames)
{
    Console.WriteLine("{0} ", names);
}
```

- In the code:
 - The list of employee names is declared in an array of `string` variables called `employeeNames`.
 - In the `foreach` statement, the data type is declared as `string` and the identifier is specified as `names`.
 - This variable refers to all values from the `employeeNames` array.
 - The `foreach` loop displays names of the employees in the order in which they are stored.

Output

```
Employee Names
Maria
Wilson
Elton
Garry
```

© Aptech Ltd. Building Applications Using C# / Session 4 46



In slide 46, tell that the code displays the employee names using the `foreach` loop. Then, explain the code.

Tell the students that in the code, 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.

Refer to the figure on slide 47 that depicts the working of the `foreach` loop.

`foreach` loop is the new way of iterating within collections of objects. We can use this loop to iterate within collections, arrays, or lists. The main advantage of this loop is that we need not know the number of times the loop should run. This loop will not throw any error even if there is no item in the collection. It is also helpful when there are chances of the list items to increase or reduce. Any such increment or decrement in the collection of items does not affect the performance of this loop. It will function properly without any problem. Even if the number of items change at any given time. So use it when you are not sure about how many items will be present in the collection.

Slide 48

Understand the jump statements in C#.



Jump Statements in C#

- ◆ Jump statements are used to transfer control from one point in a program to another.
- ◆ There will be situations where you need to exit out of a loop prematurely and continue with the program.
- ◆ In such cases, jump statements are used. Jump statement unconditionally transfer control of a program to a different location.
- ◆ The location to which a jump statement transfers control is called the target of the jump statement.
- ◆ C# supports four types of jump statements. These are as follows:
 - ◆ break
 - ◆ continue
 - ◆ goto
 - ◆ return

© Aptech Ltd.

Building Applications Using C# / Session 4 48

In slide 48, explain the jump statements in C# to the students.

Tell the students that the jump statements are used to transfer control from one point in a program to another to exit out of a loop prematurely and continue with the program.

Mention that in such cases, jump statements are used.

Tell that 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.

Tell that the C# supports four types of jump statements such as break, continue, goto, and return.

Additional Information

The problem with goto is that it may hide the flow of execution in the program and make the code confusing. But if you create easy to read/maintain code, then it is not a problem. Make use of goto statement to jump forward or to jump backward. There is no harm in doing so, but the problem arises when you make use of too many jumping statements within your code. If certain code needs to be called again and again, you can better create a separate function and make call to such a function any number of times. Debugging also become little difficult with goto statement, if it is used at too many places.

Slides 49 and 50

Understand the `break` statement.

The break Statement 1-2

- 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.
- The following figure depicts the `break` statement:

©Aptech Ltd.

Building Applications Using C# / Session 4

49

The break Statement 2-2

- The following figure displays a prime number using the `while` loop and the `break` statement:

Snippet

```
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 the code:
 - The `numOne` and `numTwo` are declared as integer variables and are initialized to values 17 and 2 respectively.
 - In the `while` statement, if the condition is true, the inner `if` condition is checked. If this condition evaluates to true, the program control passes to the `if` statement outside the `while` loop.
 - If the condition is false, the value of `numTwo` is incremented and the control passes to the `while` statement again.

Output
Prime Number

©Aptech Ltd.

Building Applications Using C# / Session 4

50

In slide 49, explain the `break` statement. Explain that 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.

Tell them that in the `switch...case` construct, it is used to terminate the execution of the construct.

Mention that 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. You can refer figure from slide 49 that displays the `break` statement.

In slide 50, tell that the code displays a prime number using the `while` loop and the `break` statement. Then, explain the code.

Tell the students that in the code `numOne` and `numTwo` are declared as integer variables and are initialized to values 17 and 2 respectively. Mention that 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.

Slides 51 and 52

Understand the `continue` statement.

The continue Statement 1-2

- The `continue` statement is most widely used in the loop constructs and 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.
- The following figure displays the working of the `continue` statement:

```

while (condition)
{
    ...
    if (TrueCondition)
        continue;
    ...
}

```

Continues Loop with the Next Value

© Aptech Ltd. Building Applications Using C# / Session 4 51

The continue Statement 2-2

- The following code displays the even numbers in the range of 1 to 10 using the `for` loop and the `continue` statement:

Snippet

```

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 the code:
 - `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.

Even numbers in the range of 1-10.

Output

```

2 4 6 8 10

```

© Aptech Ltd. Building Applications Using C# / Session 4 52

In slide 51, explain the `continue` statement.

Tell that the `continue` statement is most widely used in the loop constructs. This statement is denoted by the `continue` keyword.

Mention that 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.

You can refer figure from slide 51 that displays the working of the `continue` statement.

In slide 52, show the code that displays the even numbers in the range of 1 to 10 using the `for` loop and the `continue` statement. Then, explain the code.

Tell the students that in the code, `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.

Mention that 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.

Slide 53

Understand the `goto` statement.

The goto Statement 1-4

- 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.
- The following code displays the `goto` statement:

```

if (Truecondition)
{
    goto Display;
}
Display:←
Console.WriteLine("goto statement is executed");

```

Control Transferred to Display

- The following figure displays the output "Hello World" five times using the `goto` statement:

Snippet

```

int i = 0;
display:
Console.WriteLine("Hello World");
i++;
if (i < 5)
{
    goto display;
}

```

©Aptech Ltd. Building Applications Using C# / Session 4 53

In slide 53, explain the `goto` statement to the students.

Tell the students that 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.

Mention that the `goto` statement is denoted by the `goto` keyword. Show the figure that displays the `goto` statement.

Then, tell that the code displays the output "Hello World" five times using the `goto` statement.

You can refer figure from slide 53 that displays the `goto` statement.

Slide 54

Understand the code.

The goto Statement 2-4

- ◆ In the code:
 - ◆ `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

©Aptech Ltd. Building Applications Using C# / Session 4 54

Using slide 54, tell the students that in the code, `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, tell that 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.

Tell that they cannot use the `goto` statement for moving inside a block under the `for`, `while` or `do-while` loops.

Slides 55 and 56

Understand the code that demonstrates the use of `goto` statement to break out of a nested loop.

The goto Statement 3-4

- The following code demonstrates the use of `goto` statement to break out of a nested loop:

Snippet

```
using System;
class Factorial
{
    static void Main(string[] args)
    {
        byte num = 0;
        while (true)
        {
            byte fact = 1;
            Console.WriteLine("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");
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 4 55

The goto Statement 4-4

- The following figure shows the use of `goto` statement:

```
C:\Windows\system32\cmd.exe
Please enter a number less than or equal to 10: 5
Factorial of 5 is 120
Please enter a number less than or equal to 10: 8
Factorial of 8 is 128
Please enter a number less than or equal to 10: 25
Exiting the program
Press any key to continue . . .
```

© Aptech Ltd. Building Applications Using C# / Session 4 56

In slide 55, tell that the code demonstrates the use of `goto` statement to break out of a nested loop.

Tell that in the code, 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. Use slide 56 to refer figure that shows the use of `goto` statement.

Slides 57 to 59

Understand the `return` statement.

The return Statement 1-3

- 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.
- The following figure displays the working of the `return` statement:

```

if (TrueCondition)
{
    ...
    return;
}
else
{
    ...
}
Console.WriteLine("return statement");

```

©Aptech Ltd.

Building Applications Using C# / Session 4

57

The return Statement 2-3

- The following code displays the cube of a number using the `return` statement:

Snippet

```

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 the code:
 - 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

©Aptech Ltd.

Building Applications Using C# / Session 4

58

The return Statement 3-3

- The following code demonstrates the use of the `return` statement to terminate the program:

Snippet

```

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 the code:
 - `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.

In slide 57, explain the `return` statement to the students.

Tell the students that 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.

You can refer figure from slide 58 that displays the working of the `return` statement.

In slide 58, tell that the code displays the cube of a number using the `return` statement. Then, explain the code to the students.

Tell that in the code, 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. Mention that the `return` statement returns the calculated cube value back to the `Console.WriteLine()` method, which displays the calculated cube of 23.

In slide 59, tell that the code demonstrates the use of the `return` statement to terminate the program. Explain the code to the students.

Explain to the students that in the code, `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.

Tell them that 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.

In-Class Question:

After you finish explaining the `return` statement, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



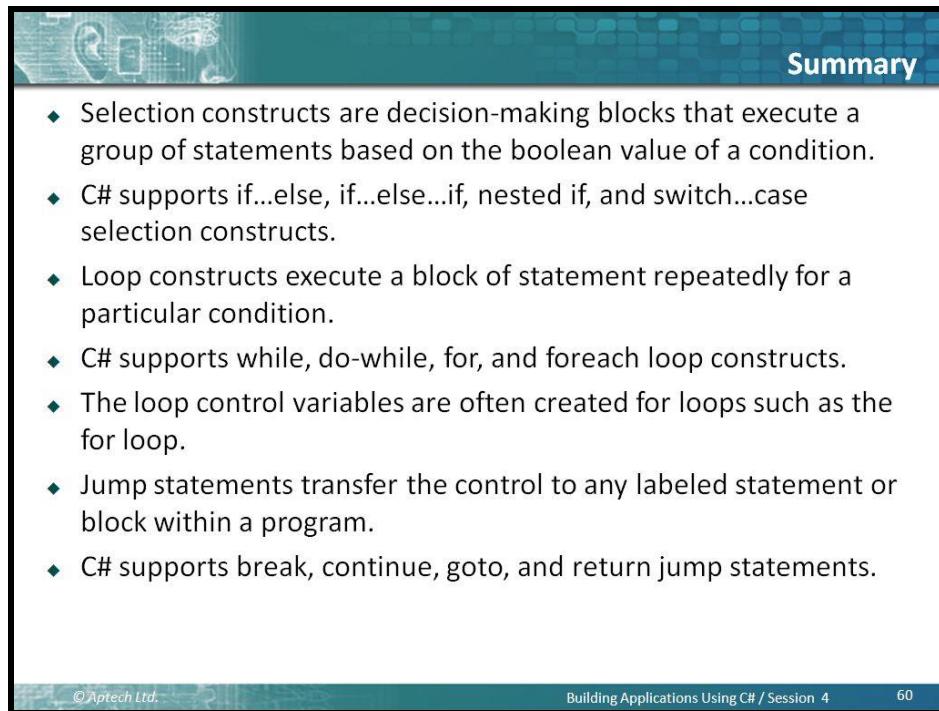
Where is the `return` statement used?

Answer:

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.

Slide 60

Summarize the session.



The slide features a blue header bar with the word "Summary" on the right. Below the header is a list of bullet points detailing various programming constructs. At the bottom of the slide, there is a footer bar with the text "©Aptech Ltd.", "Building Applications Using C# / Session 4", and the number "60".

- ◆ Selection constructs are decision-making blocks that execute a group of statements based on the boolean value of a condition.
- ◆ C# supports if...else, if...else...if, nested if, and switch...case selection constructs.
- ◆ Loop constructs execute a block of statement repeatedly for a particular condition.
- ◆ C# supports while, do-while, for, and foreach loop constructs.
- ◆ The loop control variables are often created for loops such as the for loop.
- ◆ Jump statements transfer the control to any labeled statement or block within a program.
- ◆ C# supports break, continue, goto, and return jump statements.

In slide 60, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- Selection constructs are decision-making blocks that execute a group of statements based on the boolean value of a condition.
- C# supports if..else, if..else if, nested if, and switch...case selection constructs.
- Loop constructs execute a block of statements repeatedly for a particular condition.
- C# supports while, do-while, for, and foreach loop constructs.
- The loop control variables are often created for loops such as the for loop.
- Jump statements transfer the control to any labeled statement or block within a program.
- C# supports break, continue, goto, and return jump statements.

4.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. Why does C# include programming constructs that are not CLS-compliant?
2. List the difference between Loop and Statements?
3. How can you reverse a `foreach` loop?

Session 5 - Arrays

5.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- Selection constructs are decision-making blocks that execute a group of statements based on the Boolean value of a condition.
- C# supports `if...else`, `if...else if`, nested-`if`, and `switch...case` selection constructs.
- Loop constructs execute a block of statements repeatedly for a particular condition.
- C# supports `while`, `do-while`, `for`, and `foreach` loop constructs.
- The loop control variables are often created for loops such as the `for` loop.
- Jump statements transfer the control to any labeled statement or block within a program.
- C# supports `break`, `continue`, `goto`, and `return` jump statements.

Here, you can ask students the key topics they can recall from previous session. Ask them to briefly describe selection constructs and jump statements. Ask them to describe loop constructs. Prepare a question or two which will be a key point to relate the current session objectives.

5.1.1 Objectives

By the end of this session, the learners will be able to:

- Define and describe arrays
- List and explain the types of arrays
- Explain the `Array` class

5.1.2 Teaching Skills

To teach this session successfully, you must know arrays. You should be aware of the types of arrays. You should also be familiar with the `Array` class in C#.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here for the In-Class activities.

Overview of the Session:

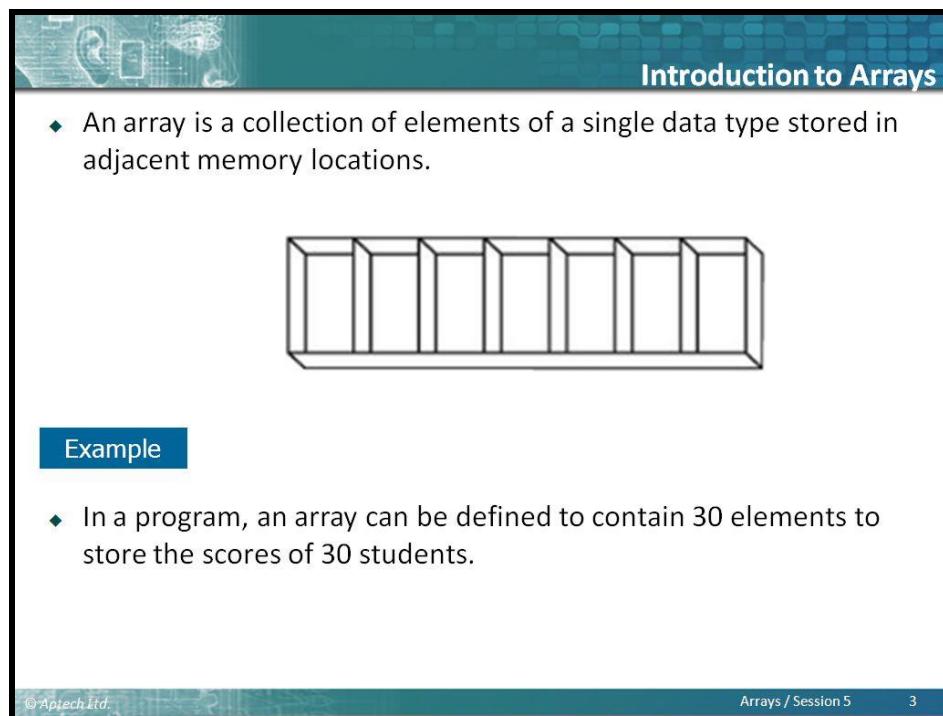
Give the students a brief overview of the current session in the form of session objectives.

Show the students slide 2 of the presentation. Tell them that they will be introduced to arrays. They will learn about the types of arrays. This session will also discuss the `Array` class in C#.

5.2 In-Class Explanations

Slide 3

Understand arrays.



The slide has a blue header bar with a gear icon and the title "Introduction to Arrays". The main content area contains a bulleted definition of an array, a diagram of an array as a row of boxes, and an "Example" section with its own bullet point. The footer bar includes the copyright notice "© Aptech Ltd.", the page number "3", and the session title "Arrays / Session 5".

Introduction to Arrays

- An array is a collection of elements of a single data type stored in adjacent memory locations.

Example

- In a program, an array can be defined to contain 30 elements to store the scores of 30 students.

© Aptech Ltd. 3 Arrays / Session 5

Use slide 3 to explain that an array is a collection of elements of a single data type stored in adjacent memory locations.

Here, you can give an example such that, in a program, an array can be defined to contain 30 elements to store the scores of 30 students.

Explain to the students with illustrations and analogies how the array elements are created in the memory. Tell them about the name, address, and value of an array element. You can show them how the values are stored and retrieved from the array.

Additional Information

For more information on arrays, refer the following link:

<http://msdn.microsoft.com/en-us/library/aa288453%28v=vs.71%29.aspx>

Slides 4 and 5

Understand the purpose of arrays.

Purpose 1-2

Example

- ◆ 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.

Array of 100 Names					
Steve	David	John	Klen	Stefen

Proper Utilization of Memory

100 Variables Storing Names	
Program to store 100 names of students	
var empOne	Steve
var studentTwo	David
var studentThree	John
var studentFour	Klen
var studentFive	Stefen
...	...
... Till 100 variables	

Inefficient Memory Utilization

© Aptech Ltd. Arrays / Session 5 4

Purpose 2-2

- ◆ An array:
 - ◆ Is a collection of related values placed in contiguous memory locations and these values are referenced using a common array name.
 - ◆ Simplifies the task of maintaining these values.
- ◆ 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 an index number zero while the last element has an 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.

© Aptech Ltd. Arrays / Session 5 5

In slide 4, explain the purpose of arrays with an example. For example, 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.

Use slide 5 to explain that 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.

Mention that C# supports zero-based index values in an array. This means that the first array element has an index number, zero, while the last element has an 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.

Additional Information

If a large amount of data were to be stored, it would be a long and tedious task to think of a separate variable name for each data element.

Consider maintaining the score of a cricket team during a match. The score of each batsman is added to the total score of the team. A team consists of 11 players. Hence, we need to keep a record of the total number of players, the score of each player and the total score of the team. Consider the case if we were to store the score of a player for 25 different innings. Our code would thus look somewhat like:

```
int i_score1, i_score2, i_score3, ..., i_scoreN  
i_score1 = 45 ;  
i_score2 = 35 ;  
i_score3 = 50 ;  
i_score4 = 0 ;  
:  
:  
:  
i_score25 = 45 ;
```

In the example given, note that declaring 25 different elements to store 25 different scores becomes a very tedious task. It is an equally painstaking job to try to assign 25 different values to 25 different variable names. Using this method of data management would make the job of programming an extremely difficult task and this method of storing data may serve the purpose as long as the number of data elements to be stored is known. However, this method does not

allow for the common case where there are an indeterminate number of values, which have to be operated upon or if we need to add large number of values.

The use of arrays provides a solution to this problem. Arrays form an important part of any programming language structure and are hence included in every high-level programming language.

An array is a list of variables that are all of the same type and are referenced through a common name. An individual variable in the array is called an array element. Arrays are thus used to store a large variety of common data under a single reference. For example, suppose a batsman scores the following runs in 8 innings:

Score

34

54

34

54

65

34

21

0

To store these scores in the order of their occurrence, a serial number should be attached to assert the order of precedence. This order can be indicated by means of a subscript. For example,

Score₁ = 34

Score₂ = 54

Score₃ = 34

Score₄ = 54

Score₅ = 65

Score₆ = 34

Score₇ = 21

Score₈ = 0

While defining/declaring an array, we basically create a series of memory locations reserved for values of the integer data type (in this case 8 such memory locations would be reserved). In other words, an array declared to be of type **int**, cannot contain elements that are not of type integer.

Slide 6

Understand an example of array.

Definition

- Following figure is an example of the subscripts and elements in an array:

Subscripts

0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	60	70	80	90	100

Elements

Arrays

© Aptech Ltd. Arrays / Session 5 6

Show slide 6 to demonstrate an example of the subscripts and elements in an array.

Slide 7

Explain array declaration.

The slide has a teal header bar with the title 'Declaring Arrays'. The main content area contains the following text:

- ◆ Arrays are reference type variables whose creation involves two steps:
 - ◆ **Declaration:**
 - An array declaration specifies the type of data that it can hold and 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.
 - ◆ **Memory allocation:**
 - Declaring an array does not allocate memory to the array.
- ◆ Following is the syntax for declaring an array:

Syntax

```
type[] arrayName;
```
- ◆ In the syntax:
 - ◆ **type:** Specifies the data type of the array elements (for example, int and char).
 - ◆ **arrayName:** Specifies the name of the array.

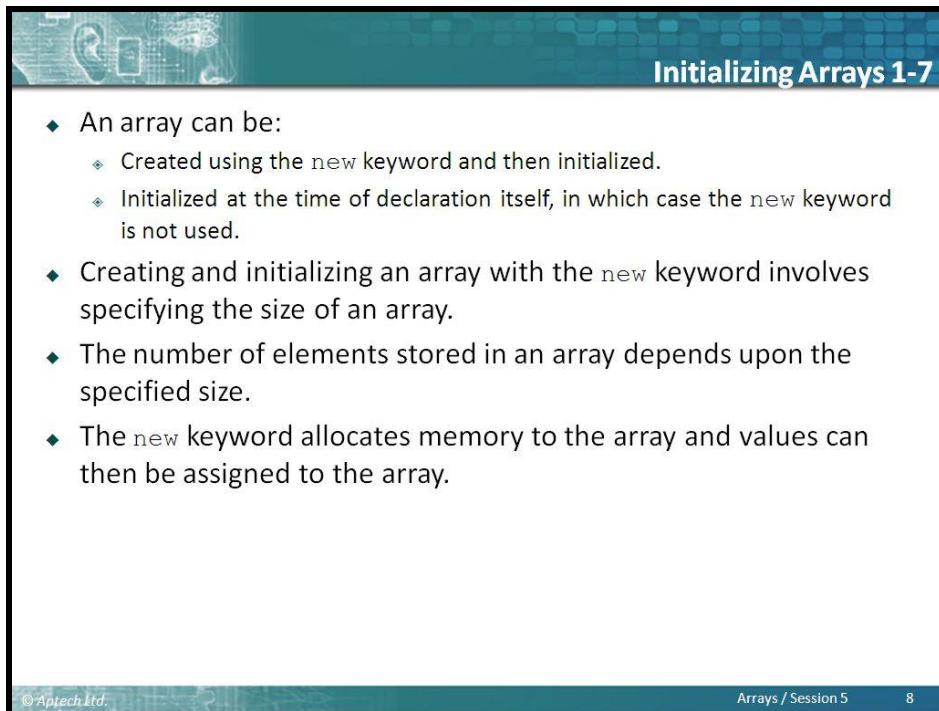
In slide 7, explain that arrays are reference type variables whose creation involves two steps, declaration and memory allocation.

Also, tell that an array declaration specifies the type of data that it can hold and 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.

Explain the syntax for declaring an array as given on the slide.

Slides 8 and 9

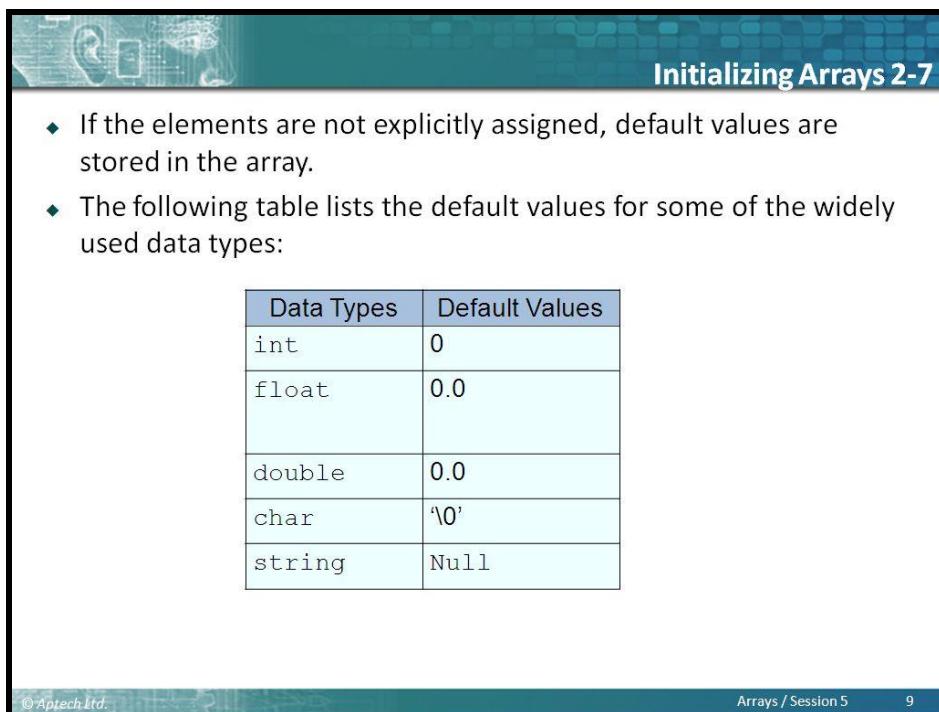
Explain initializing arrays.



Initializing Arrays 1-7

- ◆ An array can be:
 - ❖ Created using the `new` keyword and then initialized.
 - ❖ 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.

© Aptech Ltd. Arrays / Session 5 8



Initializing Arrays 2-7

- ◆ If the elements are not explicitly assigned, default values are stored in the array.
- ◆ The following table lists the default values for some of the widely used data types:

Data Types	Default Values
<code>int</code>	0
<code>float</code>	0.0
<code>double</code>	0.0
<code>char</code>	'\0'
<code>string</code>	Null

© Aptech Ltd. Arrays / Session 5 9

In slide 8, tell the students that 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.

Also, mention that 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.

Use slide 9 to explain that if the elements are not explicitly assigned, default values are stored in the array.

The table in the slide lists the default values for some of the widely used data types.

Slides 10 and 11

Explain how to create an array.

Initializing Arrays 3-7

- The following syntax is used to create an array:

Syntax
`arrayName = new type[size-value];`
- The following syntax is used to declare and create an array in the same statement using the `new` keyword:

Syntax
`type[] arrayName = new type[size-value];`
- In the syntax:
 - `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.

© Aptech Ltd. Arrays / Session 5 10

Initializing Arrays 4-7

- Once an array has been created using the syntax, its elements can be assigned values using either a subscript or using an iteration construct such as a `for` loop.
- The following syntax is used to create and initialize an array without using the `new` keyword:

Syntax
`type[] arrayIdentifier = {val1, val2, val3, ..., valN};`
- In the syntax:
 - `val1`: It is the value of the first element.
 - `valN`: It is the value of the nth element.

© Aptech Ltd. Arrays / Session 5 11

In slide 10, explain the syntax, `arrayName = new type[size-value];`

Explain the syntax used to declare and create an array in the same statement using the `new` keyword.

In slide 11, explain that once an array has been created using the syntax, its elements can be assigned values using either a subscript or an iteration construct such as a `for` loop.

Then, explain the syntax used to create and initialize an array without using the `new` keyword.

Slides 12 to 14

Explain codes that can be created in different ways.

The slide has a blue header bar with the title "Initializing Arrays 5-7". Below the header, there is a list of bullet points and code snippets.

- ◆ The following code creates an integer array which can have a maximum of five elements in it:
Snippet

```
public int[] number = new int[5];
```
- ◆ The following code initializes an array of type `string` that assigns names at appropriate index locations:
Snippet

```
public string[] studNames = new string{"Allan", "Wilson",  
"James", "Arnold"};
```
- ◆ In the code:
 - ◆ The string 'Allan' is stored at subscript 0, 'Wilson' at subscript 1, 'James' at subscript 2, and 'Arnold' at subscript 3.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Arrays / Session 5", and "12".

Initializing Arrays 6-7

- ◆ The following code stores the string 'Jack' as the name of the fifth enrolled student:

Snippet

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

- ◆ The following code demonstrates another approach for creating and initializing an array. An array called **count** is created and is assigned int values:

```
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
        }
    }
}
```

© Aptech Ltd.

Arrays / Session 5 13

Initializing Arrays 7-7

- ◆ In the code:

- ◆ 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
```

© Aptech Ltd.

Arrays / Session 5 14

In slide 12, tell them that an integer array which can have a maximum of five elements in it.

Then, explain the code that initializes an array of type string that assigns names at appropriate index locations.

Explain that the string 'Allan' is stored at subscript 0, 'Wilson' at subscript 1, 'James' at subscript 2, and 'Arnold' at subscript 3.

Use slide 13 to explain that the code stores the string 'Jack' as the name of the fifth enrolled student.

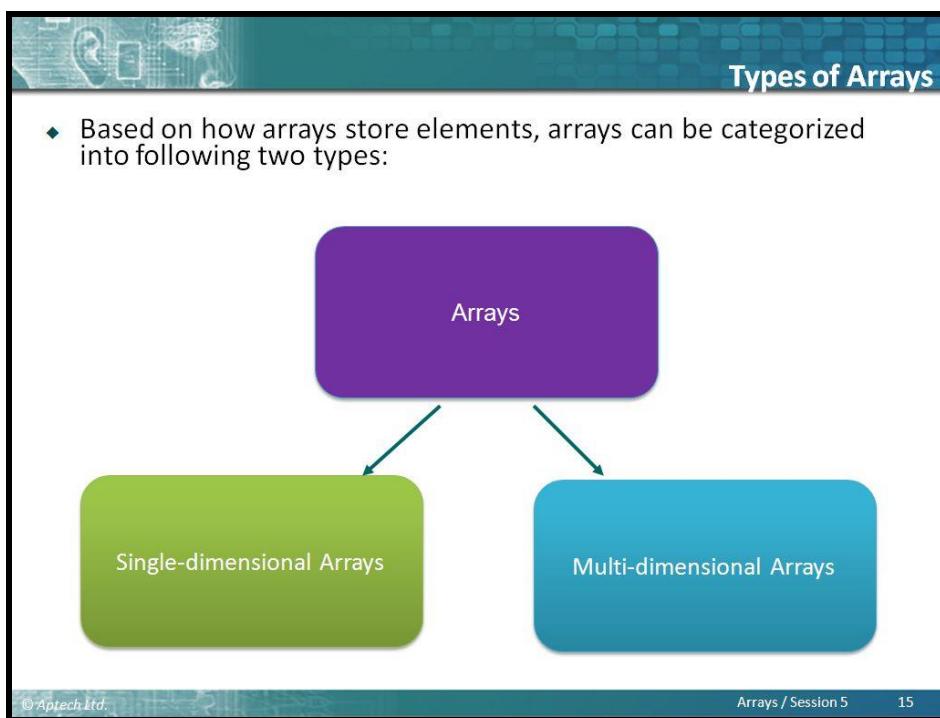
Also, explain that the code demonstrates another approach for creating and initializing an array. An array called `count` is created and is assigned `int` values.

Use slide 14 to explain that 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` and shows the output.

In this slide, ask a question to the students. Ask them what are mixed arrays?

Slide 15

Explain types of arrays.



In slide 15, explain how arrays store elements.

Mention that arrays can be categorized into two types:

- Single-dimension arrays
- Multi-dimension arrays

Slides 16 to 18

Explain single-dimensional arrays.

Single-dimensional Arrays 1-3

- ◆ Single-dimensional arrays:
 - ◆ Elements of a single-dimensional array stored in a single row in allocated memory.
 - ◆ Declaration/initialization same as standard declaration/initialization of arrays.
 - ◆ Elements indexed from 0 to (n-1), where n is the total number of elements in the array.

Example

45	22	600	71	84
----	----	-----	----	----

1st Element
 Index: 0

5th Element
 Index: 4

Syntax

- ◆ The following syntax is used for declaring and initializing a single-dimensional array:

```

type[] arrayName; //declaration
arrayName = new type[length]; // creation

```

© Aptech Ltd. Arrays / Session 5 16

Single-dimensional Arrays 2-3

- ◆ In the syntax:
 - ◆ 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.
- ◆ The following code initializes a single-dimensional array to store the name of students:

```

using System;

classSingleDimensionArray
{
    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]);
        }
    }
}

```

© Aptech Ltd. Arrays / Session 5 17

Single-dimensional Arrays 3-3

- ◆ In the code:
 - ◆ 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
```

© Aptech Ltd. | [Home](#) | [Contact Us](#) | [Feedback](#) | [Sitemap](#) | [Privacy Policy](#) | [Terms & Conditions](#) | [Help](#)

Arrays / Session 5 18

Use slide 16 to tell the students about single-dimensional array. Tell them that:

- 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.
- The elements are indexed from 0 to $(n-1)$, where n is the total number of elements in the array. For example, an array of 5 elements will have the elements indexed from 0 to 4 such that the first element is indexed 0 and the last element is indexed 4.

In slides 16 and 17, explain the syntax given on the slide. Also, tell them how a code initializes a single-dimensional array. Then, show slide 18 and explain that in the code, 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.

Slide 19

Explain multi-dimensional array.

Multi-dimensional Arrays 1-5

Example

- ◆ Consider a scenario where you need to store the roll numbers of 50 students and their marks in three exams.
- ◆ Using a single-dimensional array, you require two separate arrays for storing roll numbers and marks respectively.
- ◆ However, using a multi-dimensional array, you just need one array to store both roll numbers as well as marks.

COLUMNS

ROWS

© Aptech Ltd. Arrays / Session 5 19

In slide 19, tell them an example. Consider a scenario where you need to store the roll numbers of 50 students and their marks in three exams. Using a single-dimensional array, you require two separate arrays for storing roll numbers and marks respectively. However, using a multi-dimensional array, you just need one array to store both roll numbers as well as marks.

Multidimensional arrays are defined in the same manner as one-dimensional arrays, except that a comma is added between each pair of subscripts. Thus, a two dimensional array will require one comma with two subscripts.

As the number of dimensions of an array increases, the complexity associated with referring to the data elements of that array also increases.

Slide 20

Explain types of multi-dimensional arrays.

Multi-dimensional Arrays 2-5

- ◆ A multi-dimensional array allows you to store combination of values of a single type in two or more dimensions.
- ◆ The dimensions of the array are represented as rows and columns similar to the rows and columns of a Microsoft Excel sheet.
- ◆ Following are the two types of multi-dimensional arrays:

Rectangular Array <ul style="list-style-type: none"> • Is a multi-dimensional array where all the specified dimensions have constant values. • Will always have the same number of columns for each row. 	Jagged Array <ul style="list-style-type: none"> • Is a multidimensional array where one of the specified dimensions can have varying sizes. • Can have unequal number of columns for each row.
---	---

© Aptech Ltd. Arrays / Session 5 20

In slide 20, tell the students that a multi-dimensional array allows you to store combination of values of a single type in two or more dimensions. The dimensions of the array are represented as rows and columns, similar to the rows and columns of a Microsoft Excel sheet.

Explain to them the two types of multi-dimensional arrays, rectangular array and jagged array.

Mention that in the rectangular array, all the specified dimensions have constant values and will always have the same number of columns for each row.

Tell them that in a jagged array, one of the specified dimensions can have varying sizes and can have unequal number of columns for each row.

In-Class Question:

After you finish explaining the types of arrays, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are multi-dimensional arrays?

Answer:

Multi-dimensional arrays allow you to store combination of values of a single type in two or more dimensions. The dimensions of the array are represented as rows and columns similar to the rows and columns of a Microsoft Excel sheet.

Slide 21

Explain how to create a rectangular array.

Multi-dimensional Arrays 3-5

- ◆ The following is the syntax for creating a rectangular array:

Syntax

```
type[,] <arrayName>; //declaration
arrayName = new type[value1 , value2]; //initialization
```

- ◆ In the syntax:
 - ◆ **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.

© Aptech Ltd.

Arrays / Session 5 21

In slide 21, tell them the syntax for creating a rectangular array and explain the code as shown in the slide.

Slides 22 and 23

Explain the use of rectangular arrays.

Multi-dimensional Arrays 4-5

- The following code demonstrates the use of rectangular arrays:

Snippet

```
using System;
classRectangularArray
{
    static void Main (string [] args)
    {
        int[,] dimension = new int [4, 5];
        intnumOne = 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();
        }
    }
}
```

© Aptech Ltd. Arrays / Session 5 22

Multi-dimensional Arrays 5-5

- In the code:

- 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
```

© Aptech Ltd. Arrays / Session 5 23

In slide 22, show the code to demonstrate the use of rectangular arrays.

Use slide 23 to explain the code. Tell them that in the code, 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.

A multi-dimensional array can have a maximum of eight dimensions.

Slide 24

Explain fixed and dynamic arrays.

The slide has a teal header bar with the title 'Fixed and Dynamic Arrays'. Below the header, there's a bullet point: '◆ Arrays can be either:' followed by two circular icons: a green one labeled 'Fixed-length arrays' and a purple one labeled 'Dynamic arrays'. The 'Fixed-length arrays' section contains the following text:
 The number of elements is defined at the time of declaration.
 For example, an array declared for storing days of the week will have exactly seven elements.
 The number of elements is known and hence, can be defined at the time of declaration. Therefore, a fixed-length array can be used.
 The 'Dynamic arrays' section contains the following text:
 The size of the array is not fixed at the time of the array declaration and can dynamically increase at runtime or whenever required.
 For example, an array declared to store the e-mail addresses of all users who access a particular Web site cannot have a predefined length.
 In such a case, the length of the array cannot be specified at the time of declaration and a dynamic array has to be used.
 Can add more elements to the array as and when required.
 Created using built-in classes of the .NET Framework.

At the bottom left is the copyright notice '© Aptech Ltd.' and at the bottom right are the page numbers 'Arrays / Session 5' and '24'.

In slide 24, tell the students that arrays can be either fixed-length arrays or dynamic arrays.

Explain that in a fixed-length array, the number of elements is defined at the time of declaration. Give an example such that, an array declared for storing days of the week will have exactly seven elements. Here, the number of elements is known and hence, can be defined at the time of declaration. Therefore, a fixed-length array can be used.

In a dynamic array, the size of the array is not fixed at the time of the array declaration and can dynamically increase at runtime or whenever required.

Give an example such that, an array declared to store the e-mail addresses of all users who access a particular Web site, cannot have a predefined length. In such a case, the length of the array cannot be specified at the time of declaration. Here, a dynamic array has to be used. A dynamic array can add more elements to the array as and when required. Dynamic arrays are created using built-in classes of the .NET Framework.

Slides 25 and 26

Explain use of fixed arrays.

Multi-dimensional Arrays 1-2

- The following code demonstrates the use of fixed arrays:

```
using System;
class DaysOfWeek
{
    static void Main(string[] args)
    {
        string[] days = new string[7];
        days[0] = "Sunday";
        days[1] = "Monday";
        days[2] = "Tuesday";
        days[3] = "Wednesday";
        days[4] = "Thursday";
        days[5] = "Friday";
        days[6] = "Saturday";
        for(int i = 0; i < days.Length; i++)
        {
            Console.WriteLine(days[i]);
        }
    }
}
```

© Aptech Ltd. Arrays / Session 5 25

Multi-dimensional Arrays 2-2

- In this code:
 - A fixed-length array variable, **days**, of data type **string**, is declared to store the seven days of the week.
 - The days from Sunday to Saturday are stored in the index positions 0 to 6 of the array and are displayed on the console using the **Console.WriteLine()** method.

Output

- The following output displays the use of fixed arrays:

© Aptech Ltd. Arrays / Session 5 26

In slide 25, show the students a code that demonstrates use of fixed arrays. In slide 26, explain the code and output to the students.

- A fixed-length array variable, **days**, of data type **string**, is declared to store the seven days of the week.
- The days from Sunday to Saturday are stored in the index positions 0 to 6 of the array and are displayed on the console, using the `Console.WriteLine()` method.

In the given example, an array called `days` is created with size 7. This shows that the array can hold maximum 7 elements only. That is why, it is said to be a fixed array. The array is declared and defined with the names of week days from 'Sunday' to 'Saturday'. At the end, all the values are printed using the `for` loop.

Slides 27 and 28

Explain array references.

Array References 1-2

- ◆ 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.
- ◆ The following code demonstrates the use of array references:

```
using
using System;
classStudentReferences
{
    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++)
        {
```

© Aptech Ltd. Arrays / Session 5 27

Array References 2-2

```

        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 the code:
 - ❖ **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**.
- ◆ The following figure displays the use of array references:

© Aptech Ltd. Arrays / Session 5 28

In slide 27, explain to the students that 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.

Use slide 28 to explain the code that demonstrates the use of array references.

Tell them that in the code:

- **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**.

Then, tell them that the figure on the slide displays the use of array references.

Slide 29

Explain rectangular arrays.

The slide has a decorative header with icons like a gear, a smartphone, and a person. The title 'Rectangular Arrays 1-4' is centered in a blue header bar. The main content area contains two bullet points and a 'Syntax' section. A code example is shown in a yellow box. The footer contains copyright information and page numbers.

Rectangular Arrays 1-4

- ◆ A rectangular array is a two-dimensional array where each row has an equal number of columns.
- ◆ The following syntax displays the marks stored in a rectangular array:

Syntax

```
type [,]<variableName>;  
variableName = new type[value1 , value2];
```

- ◆ In the syntax:
 - ◆ **type**: Specifies the data type of the array elements.
 - ◆ **[,]**: Specifies that the array is a two-dimensional array.
 - ◆ **variableName**: Specifies the name of the two-dimensional array.
 - ◆ **new**: Is the operator used to instantiate the array.
 - ◆ **value1**: Specifies the number of rows in the two-dimensional array.
 - ◆ **value2**: Specifies the number of columns in the two-dimensional array.

© Aptech Ltd. Arrays / Session 5 29

In slide 29, explain to the students that a rectangular array is a two-dimensional array, where each row has an equal number of columns.

Tell them that the syntax displays the marks stored in a rectangular array.

Slides 30 to 32

Explain the code that demonstrates the use of a rectangular array.

Rectangular Arrays 2-4

- The following code allows the user to specify the number of students, their names, the number of exams, and the marks scored by each student in each exam.
- All these marks are stored in a rectangular array.

Snippet

```
using System;
class StudentsScore
{
    void StudentDetails()
    {
        Console.WriteLine("Enter the number of Students: ");
        int noOfStds = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter the number of Exams: ");
        int exams = Convert.ToInt32(Console.ReadLine());
        string[] stdName = new string[noOfStds];
        string[,] details = new string[noOfStds, exams];
        for (int i = 0; i < noOfStds; i++)
        {
            Console.WriteLine();
            Console.Write("Enter the Student Name: ");
            stdName[i] = Convert.ToString(Console.ReadLine());
            for (int y = 0; y < exams; y++)
            {
                Console.Write("Enter Score in Exam " + (y + 1) + ": ");
                details[i, y] = Convert.ToString(Console.ReadLine());
            }
        }
    }
}
```

© Aptech Ltd. Arrays / Session 5 30

Rectangular Arrays 3-4

Snippet

```
        }
    }
    static void Main()
    {
        StudentsScore objStudentsScore = new StudentsScore();
        objStudentsScore.StudentDetails();
    }
}
```

© Aptech Ltd. Arrays / Session 5 31

Rectangular Arrays 4-4

- ◆ In the code:
 - ◆ The **StudentsScore** class allows the user to enter the number of students in the class, the names of the students, the number of exams conducted, and the marks scored by each student in each exam.
 - ◆ The class declares a method **StudentDetails**, which accepts the student and the exam details.
 - ◆ The variable **noOfStds** stores the number of students whose details are to be stored.
 - ◆ The variable **exams** stores the number of exams the students have appeared in. The array **stdName** stores the names of the students.
 - ◆ The dimensions of the rectangular array **details** are defined by the variables **noOfStds** and **exams**.
 - ◆ This array stores the marks scored by students in the various exams. A nested **for** loop is used for displaying the student details.
 - ◆ In the **Main** method, an object is created of the class **StudentsScore** and the method **StudentDetails** is called through this object.

© Aptech Ltd. Arrays / Session 5 32

In slides 30 and 31, mention that code allows the user to specify the number of students, their names, the number of exams, and the marks scored by each student in each exam. All these marks are stored in a rectangular array.

Use slide 32 to explain that in the code,

- The **StudentsScore** class allows the user to enter the number of students in the class, the names of the students, the number of exams conducted, and the marks scored by each student in each exam.
- The class declares a method **StudentDetails**, which accepts the student and the exam details. The variable **noOfStds** stores the number of students whose details are to be stored.
- The variable **exams** stores the number of exams, the students has appeared in. The array **stdName** stores the names of the students. The dimensions of the rectangular array **details** are defined by the variables, **noOfStds** and **exams**.
- This array stores the marks scored by students in the various exams. A nested-for loop is used for displaying the student details.
- In the **Main** method, an object is created of the class **StudentsScore** and the method **StudentDetails** is called through this object.

Tell them that the code snippet on the slide displays the use of a rectangular array.

The details are printed using two nested-for loops that display the marks obtained in the exam. This is an example of two-dimensional rectangular array that holds the number of student and score of that student in the exam.

Tell the students how the rectangular array is declared, initialized, and printed. It is very clear from the given explanation how the number of student and their scores are accepted and then, displayed.

Slide 33

Explain jagged arrays.

Jagged Arrays 1-3

- ◆ A jagged array:
 - ❖ Is a multi-dimensional array and is referred to as an array of arrays.
 - ❖ Consists of multiple arrays where the number of elements within each array can be different. Thus, rows of jagged arrays can have different number of columns.
 - ❖ Optimizes the memory utilization and performance because navigating and accessing elements in a jagged array is quicker as compared to other multi-dimensional arrays.

Example

- ❖ Consider a class of 500 students where each student has opted for a different number of subjects.
- ❖ Here, you can create a jagged array because the number of subjects for each student varies.
- ❖ The following figure displays the representation of jagged arrays:

© Aptech Ltd. Arrays / Session 5 33

Use slide 33 to explain the students that a jagged array is a multi-dimensional array and is referred to as an array of arrays. It consists of multiple arrays where the number of elements within each array, can be different. Thus, rows of jagged arrays can have different number of columns.

A jagged array optimizes the memory utilization and performance, because navigating and accessing elements in a jagged array is quicker as compared to other multi-dimensional arrays.

Give an example such that, consider a class of 500 students where each student has opted for a different number of subjects. Here, you can create a jagged array because the number of subjects for each student varies.

The figure displays the representation of jagged arrays.

Slide 34

Explain the use of jagged arrays.

Jagged Arrays 2-3

- The following code demonstrates the use of jagged arrays to store the names of companies:

```
using System;
class JaggedArray
{
    static void Main (string[] args)
    {
        string[][] companies = new string[3][];
        companies[0] = new string[] {"Intel", "AMD"};
        companies[1] = new string[] {"IBM", "Microsoft", "Sun"};
        companies[2] = new string[] {"HP", "Canon", "Lexmark",
        "Epson"};
        for (int i=0; i<companies.GetLength (0); i++)
        {
            Console.Write("List of companies in group " + (i+1) +
            ":\t");
            for (int j=0; j<companies[i].GetLength (0); j++)
            {
                Console.Write(companies [i] [j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

© Aptech Ltd.

Arrays / Session 5 34

In slide 34, show the students the code that demonstrates the use of jagged arrays to store the names of companies.

Additional Information

The given code shows an example of jagged array. In the given array, first array called 'companies' is created with three rows but no fixed value is given for the columns. Again, on the next 3 lines, you can see that the companies[0], companies[1], and companies [2] elements are again initialized with string values. This is how the jagged array is created. Explain the loops that are used to print the values from the jagged array.

Slide 35

Explain the code that uses a jagged array.

The screenshot shows a presentation slide with a blue header bar. The title 'Jagged Arrays 3-3' is in the top right corner. Below the title, there is a bulleted list under the heading 'In the code:' and a section labeled 'Output' containing sample code and its execution results.

In the code:

- ◆ A jagged array called **companies** is created that has three rows.
- ◆ The values 'Intel' and 'AMD' are stored in two separate columns of the first row.
- ◆ Similarly, the values 'IBM', 'Microsoft', and 'Sun' are stored in three separate columns of the second row.
- ◆ Finally, the values 'HP', 'Canon', 'Lexmark', and 'Epson' are stored in four separate columns of the third row.

Output

```
List of companies in group 1: Intel AMD
List of companies in group 2: IBM Microsoft Sun
List of companies in group 3: HP Canon Lexmark Epson
```

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Arrays / Session 5' in the center, and the number '35' on the right.

In slide 35, tell the students that in the code, a jagged array called **companies** is created that has three rows. The values 'Intel' and 'AMD' are stored in two separate columns of the first row.

Similarly, the values 'IBM', 'Microsoft', and 'Sun' are stored in three separate columns of the second row. Finally, the values 'HP', 'Canon', 'Lexmark', and 'Epson' are stored in four separate columns of the third row.

Slides 36 to 38

Explain the `foreach` loop in arrays.


Using the `foreach` Loop for Arrays 1-3

- ◆ The `foreach` loop:
 - ❖ In C# is an extension of the `for` loop.
 - ❖ Is used to perform specific actions on large data collections and can even be used on arrays.
 - ❖ Reads every element in the specified array.
 - ❖ Allows you to execute a block of code for each element in the array.
 - ❖ Is particularly useful for reference types, such as strings.
- ◆ The following is the syntax for the `foreach` loop:

Syntax

```
foreach(type<identifier> in <list>
{
    // statements
}
```

- ◆ In the code:
 - ❖ type: Is the variable type.
 - ❖ identifier: Is the variable name.
 - ❖ list: Is the array variable name.

© Aptech Ltd. Arrays / Session 5 36


Using the `foreach` Loop for Arrays 2-3

- ◆ The following code displays the name and the leave grant status of each student using the `foreach` loop:

Snippet

```
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");
        }
    }
}
```

© Aptech Ltd. Arrays / Session 5 37

Using the `foreach` Loop for Arrays 3-3

- ◆ In the code:
 - ❖ 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
```

© Aptech Ltd. | Arrays / Session 5 38

In slide 36, tell the students that 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.

Explain the syntax for the `foreach` loop. Explain the code using the `foreach` loop in arrays.

Slide 37 displays the name and the leave grant status of each student using the `foreach` loop.

Use slide 38 to explain the code and the output to the students. In the code, 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.

The `foreach` loop allows you to navigate through an array without re-loading the array in the memory. During iteration, the elements in the list are in the read-only format. To change the values in the array, you need to use the `for` loop within the `foreach` loop. The `foreach` loop executes once for each element of the array.

Slide 39

Explain the Array class.

The slide has a teal header bar with the title 'Array Class'. Below the header is a white content area. On the left side of the content area, there is a blue rectangular box labeled 'Example'. Inside this box, there is a bulleted list of points. At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Arrays / Session 5' in the center, and the number '39' on the right.

- ◆ The `Array` class:
 - ◆ Is a built-in class in the `System` namespace and is the base class for all arrays in C#.
 - ◆ Provides methods for various tasks such as creating, searching, copying, and sorting arrays.

Example

- ◆ 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.

In slide 39, explain to the students with an example. Consider a code that stores the marks of a particular subject for 100 students. The programmer wants to sort the marks and to do this, he has to manually write the code to perform sorting. This can be tedious and result in increased lines of code. However, if the array is declared as an object of the `Array` class, the built-in methods of the `Array` class can be used to sort the array.

The `Array` class is a built-in class in the `System` namespace and is the base class for all arrays in C#.

The `Array` class provides methods for various tasks such as creating, searching, copying, and sorting arrays.

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.

Slides 40 and 41

Explain properties and methods.

Properties and Methods 1-2

- ◆ 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.
 - The following table 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.

© Aptech Ltd. Arrays / Session 5 40

Properties and Methods 2-2

- ❖ **Methods:**
 - The `Array` class allows you to clear, copy, search, and sort the elements declared in the array.
 - The following table displays the most commonly used methods in the `Array` class:

© Aptech Ltd. Arrays / Session 5 41

Use slide 40 to explain the students that 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.

Explain properties by referring to the table on the slide.

The properties of the `Array` class modify the elements declared in the array as displayed in the table of properties of the `Array` class.

Use slide 41 to explain the methods.

The `Array` class allows you to clear, copy, search, and sort the elements declared in the array. `Array` class allows you to easily search any element in an array. Sorting is also easy by using the `sort` method of `Array` class. You can sort the array both in ascending order as well as in descending order by passing object of `IComparer` interface.

In-Class Question:

After you finish explaining `Array` class, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is an `Array` class?

Answer:

The `Array` class is a built-in class in the `System` namespace and is the base class for all arrays in C#. It also provides methods for various tasks such as creating, searching, copying, and sorting arrays.

Slide 42

Explain creating arrays using the `Array` class.

Using the Array Class 1-5

- ◆ The `Array` class allows you to create arrays using the `CreateInstance()` method.
- ◆ This method can be used with different parameters to create single-dimensional and multi-dimensional arrays.
- ◆ For creating an array using this class, you need to invoke the `CreateInstance()` method that is accessed by specifying the class name because the method is declared as static.
- ◆ The following is the syntax for signature of the `CreateInstance()` method used for creating a single-dimensional array:

Syntax

```
public static Array CreateInstance(Type elementType, int length)
```

- ◆ In the syntax:
 - ◆ `Array`: Returns a reference to the created array.
 - ◆ `Type`: Uses the `typeof` operator for explicit casting.
 - ◆ `elementType`: Is the resultant data type in casting.
 - ◆ `Length`: Specifies the length of the array.

© Aptech Ltd. Arrays / Session 5 42

In slide 42, explain the students that the `Array` class allows you to:

- Create arrays using the `CreateInstance()` method.
- This method can be used with different parameters to create single-dimensional and multi-dimensional arrays.
- For creating an array using this class, you need to invoke the `CreateInstance()` method that is accessed by specifying the class name, because the method is declared as static.

Explain to them the syntax used for creating a single-dimensional array.

Slide 43

Explain the syntax used for creating a multi-dimensional array.

The slide has a blue header bar with the title "Using the Array Class 2-5". Below the header, there is a bulleted list of points. One point is highlighted with a yellow background and a black border, containing a code snippet. The code is:

```
public static Array CreateInstance(Type elementType, int length1,  
int length2)
```

The list below the code contains several bullet points explaining the syntax and usage of the method.

- ◆ The following is the syntax for signature of the `CreateInstance()` method used for creating a multi-dimensional array.
- Syntax**

```
public static Array CreateInstance(Type elementType, int length1,  
int length2)
```
- ◆ In the syntax:
 - ◆ `length1`: Specifies the row length.
 - ◆ `length2`: Specifies the column length.
- ◆ These syntax determine how the method is declared in the `Array` class.
- ◆ To create single-dimensional and multi-dimensional arrays, you must explicitly invoke the method with the appropriate parameters.

At the bottom left is the copyright notice "©Aptech Ltd." and at the bottom right are the page numbers "Arrays / Session 5" and "43".

Use slide 43 to explain the syntax used for creating a multi-dimensional array.

Tell them that how the method is declared in the `Array` class. To create single-dimensional and multi-dimensional arrays, you must explicitly invoke the method with the appropriate parameters.

Slides 44 and 45

Explain the code to create an array using an `Array` class.

Using the Array Class 3-5

- The following code creates an array of length 5 using the `Array` class and stores the different subject names:

Snippet

```
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));
        }
    }
}
```

© Aptech Ltd. Arrays / Session 5 44

Using the Array Class 4-5

- In the code:
 - 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.

© Aptech Ltd. Arrays / Session 5 45

In slide 44, tell the students that the code creates an array of length 5 using the `Array` class and stores the different subject names.

Use slide 45 to explain the code.

In the code, 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.

More than one **CreateInstance()** method is declared in the **Array** class, which takes in different parameters. The parameters can accept 64-bit integers to accommodate large-capacity arrays.

Slide 46

Explain the four interfaces of the **Array** class.

Using the Array Class 5-5

- ◆ For manipulating an array, the **Array** class uses the following four interfaces:

ICloneable	ICollection	IList	IEnumerable
<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • 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.

© Aptech Ltd. Arrays / Session 5 46

In slide 46, explain the students that for manipulating an array, the **Array** class uses four interfaces:

- **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.

Slides 47 to 49

Explain rank of an array.

Rank of an Array 1-3

- ◆ Rank is a read-only property that specifies the number of dimensions of an array.

Example A three-dimensional array has rank three.

- ◆ The following code demonstrates the use of the Rank property:

Snippet

```
using System;
class Employee
{
    public static void Main()
    {
        Array objEmployeeDetails = Array.CreateInstance(typeof(string), 2, 3);
        objEmployeeDetails.SetValue("141", 0, 0);
        objEmployeeDetails.SetValue("147", 0, 1);
        objEmployeeDetails.SetValue("154", 0, 2);
        objEmployeeDetails.SetValue("Joan Fuller", 1, 0);
        objEmployeeDetails.SetValue("Barbara Boxen", 1, 1);
        objEmployeeDetails.SetValue("Paul Smith", 1, 2);
        Console.WriteLine("Rank : " + objEmployeeDetails.Rank);
        Console.WriteLine("Employee ID \tName");
    }
}
```

© Aptech Ltd. Arrays / Session 5 47

Rank of an Array 2-3

```

for (int i = 0; i < 1; i++)
{
    for (int j = 0; j < 3; j++)
    {
        Console.Write(objEmployeeDetails.GetValue(i, j) + "\t\t");
        Console.WriteLine(objEmployeeDetails.GetValue(i+1, j));
    }
}
}

```

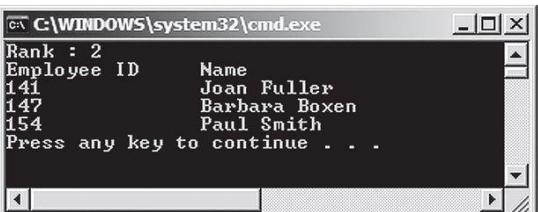
- ◆ In the code:
 - ❖ The `CreateInstance()` method creates a two-dimensional array of the specified type and dimension lengths.
 - ❖ Since this array has two dimensions, its rank will be 2.
 - ❖ An instance of this class `objEmployeeDetails` is created and two sets of values are then inserted in the object `objEmployeeDetails` using the method `SetValue()`.
 - ❖ The values stored in the array are employee ID and the name of the employee. The `Rank` property retrieves the rank of the array which is displayed by the `WriteLine()` method.

© Aptech Ltd.

Arrays / Session 5 48

Rank of an Array 3-3

- ◆ The following figure displays the use of Rank property:



© Aptech Ltd.

Arrays / Session 5 49

In slide 47, show the code to the students that Rank is a read-only property that specifies the number of dimensions of an array. For example, a three-dimensional array has rank three. Tell them that the code demonstrates the use of the Rank property and explain the code. In slide 48, explain the code. In the code, the `CreateInstance()` method creates a two-dimensional array of the specified type and dimension lengths. Since this array has two dimensions, its rank will be 2. An instance of this class `objEmployeeDetails` is created and two sets of values are then inserted in the object, `objEmployeeDetails` using the method `SetValue()`. The values stored in the array are employee ID and the name of the employee.

The Rank property retrieves the rank of the array which is displayed by the `WriteLine()` method. Refer to the figure on slide 49 that demonstrates the use of Rank property.

In-Class Question:

After you finish explaining `Array` class, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is Rank?

Answer:

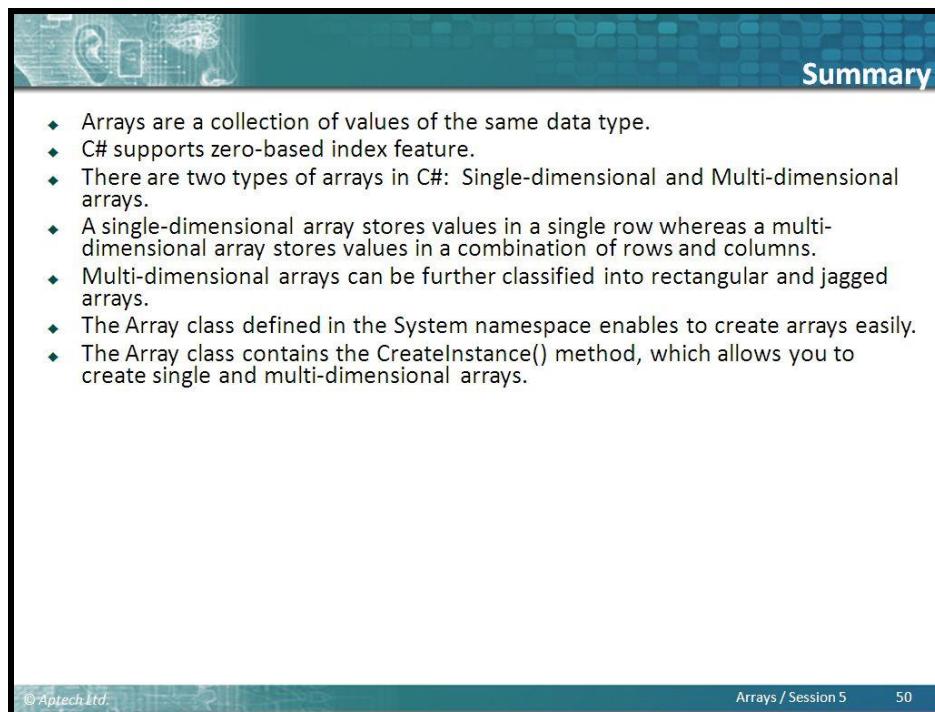
Rank is a read-only property that specifies the number of dimensions of an array.

You can also put a few questions to students to search additional information, such as:

1. What is the difference between length and count?
2. What are array properties?
3. What are vectors?
4. Explain different types of arrays.

Slide 50

Summarize the session.



Summary

- ◆ Arrays are a collection of values of the same data type.
- ◆ C# supports zero-based index feature.
- ◆ There are two types of arrays in C#: Single-dimensional and Multi-dimensional arrays.
- ◆ A single-dimensional array stores values in a single row whereas a multi-dimensional array stores values in a combination of rows and columns.
- ◆ Multi-dimensional arrays can be further classified into rectangular and jagged arrays.
- ◆ The `Array` class defined in the `System` namespace enables to create arrays easily.
- ◆ The `Array` class contains the `CreateInstance()` method, which allows you to create single and multi-dimensional arrays.

© Aptech Ltd. Arrays / Session 5 50

In slide 50, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief.

Tell them that:

- Arrays are a collection of values of the same data type.
- C# supports zero-based index feature.
- There are two types of arrays in C#, single-dimensional and multi-dimensional arrays.
- A single-dimensional array stores values in a single row whereas a multi-dimensional array stores values in a combination of rows and columns.
- Multi-dimensional arrays can be further classified into rectangular and jagged arrays.
- The `Array` class defined in the `System` namespace enables to create arrays easily.
- The `Array` class contains the `CreateInstance()` method, which allows you to create single and multi-dimensional arrays.

5.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. What are classes and objects?
2. What are methods, how to use methods in class?
3. Explain the concept of method overloading?

Session 6 - Classes and Methods

6.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the previous session for a review. Prepare the background knowledge/summary to be discussed with students in the class. The summary of the previous session is as follows:

- Arrays are a collection of values of the same data type.
- C# supports zero-based index feature.
- There are two types of arrays in C#, single-dimensional and multi-dimensional arrays.
- A single-dimensional array stores values in a single row whereas a multi-dimensional array stores values in a combination of rows and columns.
- Multi-dimensional arrays can be further classified into rectangular and jagged arrays.
- The `Array` class defined in the `System` namespace enables to create arrays easily.
- The `Array` class contains the `CreateInstance()` method, which allows you to create single and multi-dimensional arrays.

Familiarize yourself with the topics of this session in-depth.

6.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain classes and objects
- Define and describe methods
- List the access modifiers
- Explain method overloading
- Define and describe constructors and destructors

6.1.2 Teaching Skills

To teach this session successfully, you must know about the classes and objects. You should be aware of the basic concepts of access modifiers, method overloading, constructors, and destructors.

You should teach the concepts in the theory class using the concepts, tables, and codes provided. You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced the concept of classes and objects. They will learn about the access modifiers and method overloading.

This session will also discuss the concepts of constructors and destructors.

6.2 In-Class Explanations

Slide 3

Understand Object-Oriented Programming (OOP).

Object-Oriented Programming

- ◆ Programming languages are based on two fundamental concepts, data and ways to manipulate data.
- ◆ Traditional languages such as Pascal and C used the procedural approach which focused more on ways to manipulate data rather than on the data itself.
- ◆ This approach had several drawbacks such as lack of re-use and lack of maintainability.
- ◆ To overcome these difficulties, OOP was introduced, which focused on data rather than the ways to manipulate data.
- ◆ The object-oriented approach defines objects as entities having a defined set of values and a defined set of operations that can be performed on these values.

© Aptech Ltd. Classes and Methods / Session 6 3

In slide 3, explain the concept of object-oriented programming. Explain to the students that the programming languages have always been designed based on two fundamental concepts, data and ways to manipulate data. Traditional languages such as Pascal and C used the procedural approach which focused more on ways to manipulate data rather than on the data itself. This approach had several drawbacks such as lack of re-use and lack of maintainability.

Tell the students that 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.

Slide 4

Understand the features of OOP.

Features

- ◆ Object-oriented programming provides the following features:

Abstraction	• Abstraction is the feature of extracting only the required information from objects. For example, consider a television as an object. It has a manual stating how to use the television. However, this manual does not show all the technical details of the television, thus, giving only an abstraction to the user.
Encapsulation	• Details of what a class contains need not be visible to other classes and objects that use it. Instead, only specific information can be made visible and the others can be hidden. This is achieved through encapsulation, also called data hiding. Both abstraction and encapsulation are complementary to each other.
Inheritance	• Inheritance is the process of creating a new class based on the attributes and methods of an existing class. The existing class is called the base class whereas the new class created is called the derived class. This is a very important concept of object-oriented programming as it helps to reuse the inherited attributes and methods.
Polymorphism	• Polymorphism is the ability to behave differently in different situations. It is basically seen in programs where you have multiple methods declared with the same name but with different parameters and different behavior.

© Aptech Ltd. Classes and Methods / Session 6 4

In slide 4, tell the students that object-oriented programming provides a number of features that distinguish it from the traditional programming approach.

Tell them that the features of the object-oriented programming are: abstraction, encapsulation, inheritance, and polymorphism. Explain each feature in detail to the students.

Explain abstraction as follows:

Tell the students that the abstraction is the feature of extracting only the required information from objects.

Give them an example. Tell them to 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.

Explain encapsulation as follows:

Tell the students that all details of what a class contains need not be visible to other classes and objects that use it. Instead, only specific information can be made visible and the others can be hidden. This is achieved through encapsulation, also called data hiding. Both abstraction and encapsulation are complementary to each other.

Explain inheritance as follows:

Explain the students that the 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.

Explain polymorphism as follows:

Tell the students that the polymorphism is the ability to behave differently in different situations. It is basically seen in programs where you have multiple methods declared with the same name but with different parameters and different behavior.

Additional Information

Data abstraction involves concentrating on the essential details while ignoring the non-essential ones. When we create a class, we have to concentrate on the details essential for our system that we are trying to develop.

Encapsulation allows us to hide the non-essential details from the user of the current class. We can decide what should be visible and what should be invisible to the user. This can be implemented by using private, public, and protected keywords.

Inheritance helps us to develop new classes based on existing classes. It saves our coding time.

By giving rise to code reusability, the class which is derived is called as child class, derived class or sub class and from which it is derived that class is called a parent class or super class.

You can refer the following links for more information on OOP:

<http://studytipsandtricks.blogspot.in/2012/04/features-of-object-oriented-programming.html>
<http://www.durofy.com/the-basics-of-object-oriented-programming/>
<http://codebetter.com/raymondlewallen/2005/07/19/4-major-principles-of-object-oriented-programming/>

Slides 5 and 6

Understand classes and objects.

Classes and Objects 1-2

- ◆ C# programs are composed of classes that represent the entities of the program which 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.
- ◆ An object is a tangible entity such as a car, a table, or a briefcase. Every object has some characteristics and is capable of performing certain actions.
- ◆ The concept of objects in the real world can also be extended to the programming world. An object in a programming language has a unique identity, state, and behavior.
- ◆ The state of the object refers to its characteristics or attributes whereas the behavior of the object comprises its actions.
- ◆ An object has various features that can describe it which could be the company name, model, price, mileage, and so on.

Classes and Objects 2-2

- ◆ The following figure shows an example of objects:

 Identity: AXA 43 S State: Color-Red, Wheels-Four Behavior: Running	 Identity: T002 State: Color-Brown Behavior: Stable
---	--

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

In slide 5, introduce the classes and objects to the students. Tell the students that the 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.

Tell them that the object is a tangible entity such as a car, a table, or a briefcase. Every object has some characteristics and is capable of performing certain actions.

Explain them that 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.

Mention that these features could be the company name, model, price, mileage, and so on.

Explain examples of objects. In slide 6, show the figure to the students and tell them that the car and the rounded table are the objects. As shown in the figure, explain to the students that the car's identity is AXA 43 S, state is color-red, wheels-four, and the behavior is running. Whereas, table's Identity is T002, state is color-brown, and behavior is stable.

Tell the students that an object stores its identity and state in fields (also called variables) and exposes its behavior through methods.

Give examples of real life objects around you. You can include various object seen at home or you can include object seen in the office or classroom. Things such as Television, Computer, Pen, Table, and Chair are some of the examples with which the students are already familiar. It becomes easy for them to understand the concepts if you explain it with the context to the things they already know. The chair may have color, material, shape, and other features. These are the characteristics of that object. The volume up or down, changing the channels, putting it on or off, these are some of the actions. Explain the concepts with the real life examples.

Additional Information

Refer the following links for more information on classes and objects:

<http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep>
<http://msdn.microsoft.com/en-us/library/dd460654.aspx>

Slide 7

Understand classes.

◆ Several objects have a common state and behavior and thus, can be grouped under a single class.

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.

◆ The following figure displays the class Car:

Car Class
Characteristics <ul style="list-style-type: none">➤ Make➤ Model
Behavior <ul style="list-style-type: none">➤ Driving➤ Accelerating➤ Braking

© Aptech Ltd. Classes and Methods / Session 6 7

In slide 7, tell the students that several objects have a common state and behavior and thus, can be grouped under a single class.

To understand this, explain the following example to them.

Tell them that 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.

Show the figure on the slide that displays the class Car.

Explain the various characteristics and behavior of the car. Relate this example with the car that they might be familiar with.

Slides 8 and 9

Understand the process of creating classes.


Creating Classes 1-2

- ◆ The concept of classes in the real world can be extended to the programming world, similar to the concept of objects.
- ◆ In object-oriented programming languages like C#, a class is a template or blueprint which defines the state and behavior of all objects belonging to that class.
- ◆ A class comprises fields, properties, methods, and so on, collectively called data members of the class. In C#, the class declaration starts with the `class` keyword followed by the name of the class.
- ◆ The following syntax is used to declare a class:

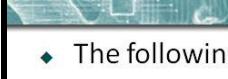
Syntax

```
{  
    // class members  
}
```

where,

`ClassName`: Specifies the name of the class.


Classes and Methods / Session 6 8


Creating Classes 2-2

- ◆ The following figure displays a sample class:

Fields
Methods

```
class Students  
{  
    string _studName = "James Anderson";  
    int _studAge = 27;  
  
    void Display()  
    {  
        Console.WriteLine("Student Name: " + _studName);  
        Console.WriteLine("Student Age: " + _studAge);  
    }  
  
    static void Main(string[] args)  
    {  
        Students objStudents = new Students();  
        objStudents.Display();  
    }  
}
```


Classes and Methods / Session 6 9

In slide 8, tell the students that 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.

Explain them that 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.

Then, show the syntax that is used to declare a class and tell the students that the `ClassName` specifies the name of the class.

In slide 9, show a sample class to the students as given in the figure.

Slide 10

Understand the guidelines for naming classes.

Guidelines for Naming Classes

- ◆ There are certain conventions to be followed for class names while creating a class that help you to follow a standard for naming classes.
- ◆ These conventions state that a class name:
 - Should be a noun and written in initial caps, and not in mixed case.
 - Should be simple, descriptive, and meaningful.
 - Cannot be a C# keyword.
 - Cannot begin with a digit but can begin with the '@' character or an underscore (_).

Example

- ◆ Valid class names are: `Account`, `@Account`, and `_Account`.
- ◆ Invalid class names are: `2account`, `class`, `Acccount`, and `Account123`.

© Aptech Ltd. 2011

Classes and Methods / Session 6 10

In slide 10, explain the guidelines for naming classes.

Tell the students that there are certain conventions to be followed for class names while creating a class. These conventions help you to follow a standard for naming classes.

Explain them that 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.

Mention that some examples of valid class names are **Account**, **@Account**, and **_Account** and the examples of invalid class names are **2account**, **class**, **Acccount**, and **Account123**.

Slide 11

Understand the Main() method.

The slide has a blue header bar with the title "Main Method". The main content area contains a bulleted list of three points about the Main() method. The footer bar is also blue and contains the copyright information "© Aptech Ltd." and the page number "11".

- ◆ The Main () method indicates to the CLR that this is the first method of the program which is declared within a class and specifies where the program execution begins.
- ◆ Every C# program that is to be executed must have a Main () method as it is the entry point to the program.
- ◆ The return type of the Main () in C# can be int or void.

© Aptech Ltd. Classes and Methods / Session 6 11

In slide 11, tell the students that the Main() method indicates to the CLR that this is the first method of the program. This method is declared within a class and specifies where the program execution begins. Every C# program that is to be executed must have a Main() method as it is the entry point to the program. The return type of the Main() in C# can be int or void.

Additional Information

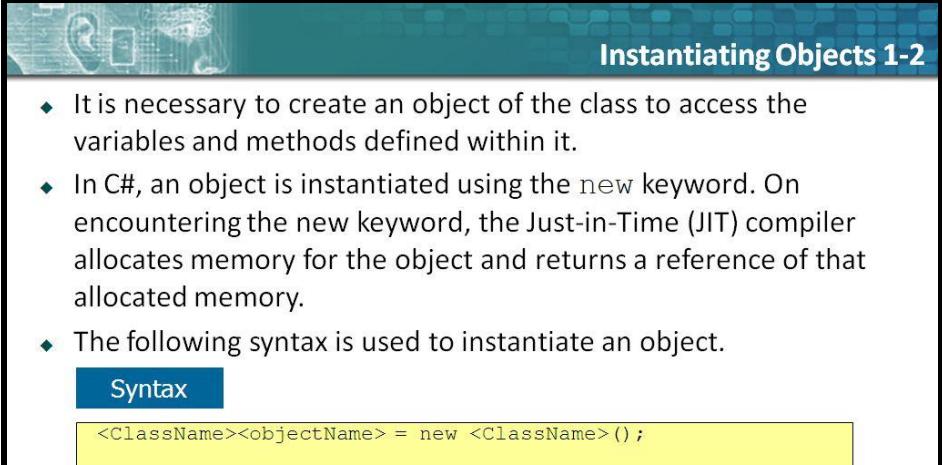
Refer the following links for more information on the Main() method:

<http://www.codeproject.com/Articles/479467/Main-Method-in-Csharp>

<http://msdn.microsoft.com/en-us/library/acy3edy3.aspx>

Slides 12 and 13

Understand instantiating objects.



Instantiating Objects 1-2

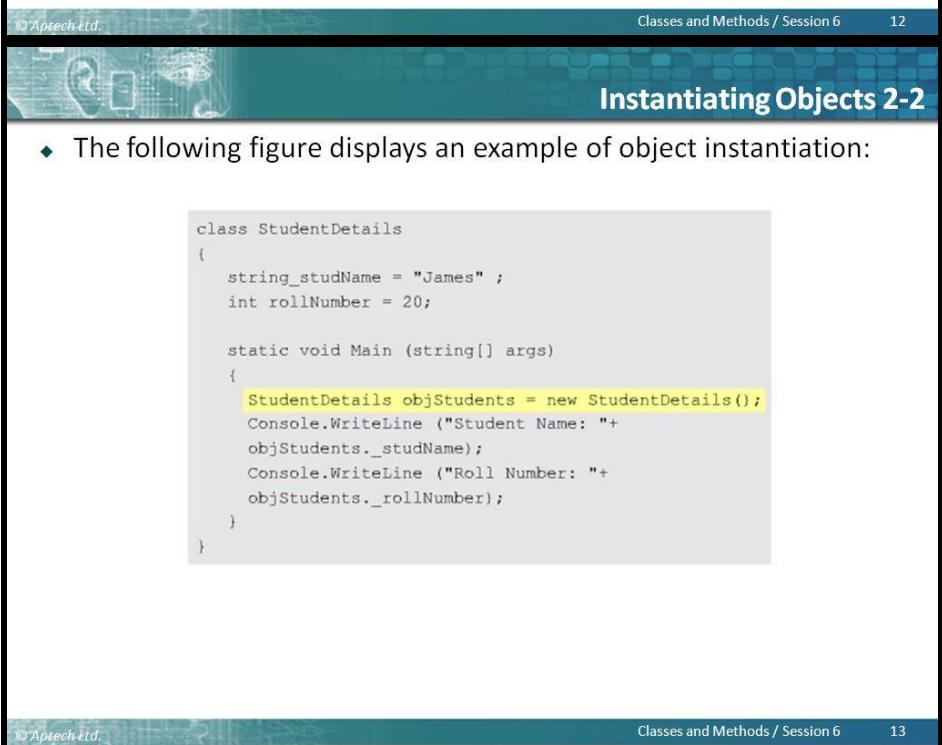
- ◆ It is necessary to create an object of the class to access the variables and methods defined within it.
- ◆ In C#, an object is instantiated using the `new` keyword. On encountering the `new` keyword, the Just-in-Time (JIT) compiler allocates memory for the object and returns a reference of that allocated memory.
- ◆ The following syntax is used to instantiate an object.

Syntax

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

where,

- ◆ `ClassName`: Specifies the name of the class.
- ◆ `objectName`: Specifies the name of the object.



Instantiating Objects 2-2

- ◆ The following figure 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);
    }
}
```

In slide 12, tell the students that when you create a class, it is necessary to create an object of the class to access the variables and methods defined within it.

Tell them that in C#, an object is instantiated using the `new` keyword. On encountering the `new` keyword, the Just-in-Time (JIT) compiler allocates memory for the object and returns a reference of that allocated memory.

Then, show how the syntax is used to instantiate an object and tell them that the `ClassName` specifies the name of the class and the `objectName` specifies the name of the object.

In slide 13, show the figure that displays an example of object instantiation.

Tell the students that 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.

Slide 14

Understand methods.

The slide features a blue header bar with the word 'Methods' on the right. Below the header is a large white area containing bulleted text and a callout box. At the bottom is a dark blue footer bar with copyright and page number information.

- ◆ 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.
- ◆ Methods specify the manner in which a particular operation is to be carried out on the required data members of the class.

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**.

© Aptech Ltd. Classes and Methods / Session 6 14

In slide 14, introduce methods to the students.

Explain them that the 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.

Tell them that methods specify the manner in which a particular operation is to be carried out on the required data members of the class.

Give them an example.

Tell them that 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**.

Slides 15 and 16

Understand the process of creating methods.


Creating Methods 1-2

- ◆ Conventions to be followed for naming methods state that a method name:
 - Cannot be a C# keyword, cannot contain spaces, and 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()**.

Example

Invalid method names include **5Add**, **AddSum()**, and **int()**.

- ◆ The following syntax is used to create a method:

Syntax

```
<access_modifier><return_type><MethodName> ([list of parameters]) {
  // body of the method
}
```

- ◆ where,
 - ◆ **access_modifier**: Specifies the scope of access for the method.
 - ◆ **return_type**: Specifies the data type of the value that is returned by the method and it is optional.
 - ◆ **MethodName**: Specifies the name of the method.
 - ◆ **list of parameters**: Specifies the arguments to be passed to the method.

© Aptech Ltd.
Classes and Methods / Session 6 15


Creating Methods 2-2

- ◆ The following code shows the definition of a method named **Add()** that adds two integer numbers:

```
using System;
class Sum
{
    int Add(int numOne, int numTwo)
    {
        int addResult = numOne + numTwo;
        Console.WriteLine("Addition = " + addResult);
        ...
    }
}
```

- ◆ In the code:
 - ◆ 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.

© Aptech Ltd.
Classes and Methods / Session 6 16

In slide 15, tell the students that the 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.

 ©Aptech Limited

Explain them that 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

Mention that there are some examples of valid method names such as **Add ()**, **Sum_Add()**, and **@Add()**. The examples of invalid method names include **5Add**, **\$5AddSum()**, and **int()**.

Then, show the syntax used to create a method and tell the students that the `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`. The `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.

The `MethodName` specifies the name of the method and the `list of parameters` specifies the arguments to be passed to the method.

Explain the code that shows the definition of a method named **Add ()** that adds two integer numbers.

In slide 16, show the code that shows the definition of a method named **Add ()** that adds two integer numbers. Then, explain the code.

Tell them that 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.

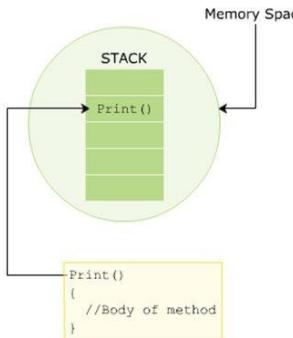
Mention that the `Main ()` method of a class is mandatory if the program is to be executed. If an application has two or more classes, one of them must contain a `Main ()`, failing which the application cannot be executed. Tell them that the `Main ()` is written with capitalized 'M' and not as `main ()` in lowercase as is done with C/C++ and Java. `Main()` method is the entry point of the application from where the application starts executing. That is why it is necessary that in every application at least one class must have `Main ()` method.

Slides 17 and 18

Understand how to invoke a method.


Invoking Methods 1-2

- ◆ A method can be invoked in a class by creating an object of the class where 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. This is referred to as the **calling** method and the invoked method is referred to as the **called** method.
- ◆ The following figure displays how a method invocation or call is stored in the stack in memory and how a method body is defined:



The diagram illustrates the state of memory during a method invocation. A large green circle labeled "STACK" contains a stack frame for the `Print()` method. The stack frame is represented as a vertical stack of four colored rectangles: light blue at the top, followed by yellow, red, and green. Below the stack, a pointer points to a separate box containing the method's body: `Print()
{
 //Body of method
}`. An arrow also points from the stack frame back to this code box, indicating they are linked.


Classes and Methods / Session 6 **17**


Invoking Methods 2-2

Snippet

```
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 the code:
 - ◆ 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


Classes and Methods / Session 6 **18**

In slide 17, tell the students that they 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.

Explain them that 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.

Then, show the figure that displays how a method invocation or call is stored in the stack in memory and how a method body is defined.

In slide 18, show the code to the students and tell them that the code is used to define methods `Print()` and `Input()` in the `Book` class and then invoke them through an object `objBook` in the `Main()` method. Explain the code to the students.

Tell the students that in the code, 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, mention that the `Print()` method is called from the `Main()` method and it displays the name of the book as the output.

Tell the students that you can even pass another string and show them the change in the output. Explain to them how the object is created. How the call to `Input()` and `Print()` method is done. And the output is printed as the string that was passed by `Input()` method because the `Print()` method returns the variable `_bookName`.

Slide 19

Understand method parameters and arguments.

Method Parameters and Arguments

- Method parameters and arguments:
 - Parameters**: The variables included in a method definition are called parameters. Which 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.
 - Arguments**: When the method is called, the data that you send into the method's parameters are called arguments.
- The following figure shows an example of parameters and arguments:

```

class Student{
    public void Display(string myParam)
    {
    ...
    }
    public static void Main()
    {
        string myArg1 = "this is my argument";
        ...
        objStudent.Display(myArg1);
    }
}
  
```

Parameter
Argument

© Aptech Ltd. Classes and Methods / Session 6 19

In slide 19, tell the students what is a parameter and an argument.

Tell them that the variables included in a method definition are called parameters. When the method is called, the data that you send into the method's parameters are called arguments.

Explain them that 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.

Then, show the figure of an example of parameters and arguments to the students.

In-Class Question:

After you finish explaining the method parameters and arguments, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are the features of object-oriented programming?

Answer:

The features of the object-oriented programming are: abstraction, encapsulation, inheritance, and polymorphism.

Slide 20

Understand named and optional arguments.

Q
Classes and Methods / Session 6
20

Named and Optional Arguments 1-6

- ◆ A method in a C# program can accept multiple arguments that are passed based on the position of the parameters in the method signature.
- ◆ A method caller can explicitly name one or more arguments being passed to the method instead of passing the arguments based on their position.
- ◆ 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.
- ◆ Named arguments are beneficial because you do not have to remember the exact order of parameters in the parameter list of methods.

In slide 20, explain the named and the optional arguments. Tell them that 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.

Explain them that while passing named arguments, the order of the arguments declared in the method do not matter. The second argument of the method can be passed ahead of the first argument. Also a named argument can follow positional arguments. Mention that named arguments are beneficial because you do not have to remember the exact order of parameters in the parameter list of methods.

Slides 21 and 22

Understand how to use named arguments.

Named and Optional Arguments 2-6

- The following code demonstrates how to use named arguments:

Snippet

```
using System;

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");

    }
}
```

©Aptech Ltd.

Classes and Methods / Session 6 21

Named and Optional Arguments 3-6

- ◆ In the code:
 - ❖ The first call to the **printNamed()** method passes positional arguments.
 - ❖ The second and third call passes named arguments in different orders.
 - ❖ The fourth call passes a positional argument followed by a named argument.

Output

```
First Name = Henry, Last Name = Parker
```

©Aptech Ltd. Classes and Methods / Session 6 22

In slide 21, show the code that demonstrates how to use named arguments.

Use slide 22 to explain the code.

Tell the students that in the code, 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.

Show the output of the code to the students.

Slide 23

Understand another example of using named arguments.

The slide has a blue header bar with the title "Named and Optional Arguments 4-6". Below the header, there is a bulleted list of three items. The first item is enclosed in a yellow box labeled "Snippet". The snippet contains C# code for a class named "TestProgram" with a method "Count" and a static method "Main". The "Count" method takes two parameters: "boys" and "girls", both of type int. It prints the sum of boys and girls to the console. The "Main" method creates an instance of "TestProgram" and calls its "Count" method with named arguments: "boys: 16, girls: 24".

- ◆ The following code shows another example of using named arguments:
- ◆ C# also supports optional arguments in methods and can be emitted by the method caller.
- ◆ Each optional argument has a default value.

© Aptech Ltd. Classes and Methods / Session 6 23

In slide 23, explain the code that shows another example of using named arguments.

Tell the students that 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.

Slides 24 and 25

Understand how to use optional arguments.

Named and Optional Arguments 5-6

- The following code shows how to use optional arguments:

Snippet

```
using System;
classOptionalParameterExample
{
    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();
    }
}
```

Classes and Methods / Session 6 24

Named and Optional Arguments 6-6

- In the code:
 - The **printMessage()** method declares an optional argument **message** with a default value **Hello user!**.
 - The first call to the **printMessage()** method passes an argument value that is printed on the console.
 - The second call does not pass any value and therefore, the default value gets printed on the console.

Output

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

Classes and Methods / Session 6 25

In slide 24, show the code that demonstrates how to use optional arguments.

In slide 25, tell the students that in the code, the **printMessage()** method declares an optional argument **message** with a default value **Hello user!**. The first call to the **printMessage()** method passes an argument value that is printed on the console. The second call does not pass any value and therefore, the default value gets printed on the console. Show the output of the code to the students.

Slides 26 to 28

Understand static classes.


Static Classes 1-3

- ◆ Classes that cannot be instantiated or inherited are known as classes and the **static** keyword is used before the class name that consists of static data members and static methods.
- ◆ It is not possible to create an instance of a static class using the **new** keyword. The main features of static classes are as follows:
 - ◆ They can only contain static members.
 - ◆ They cannot be instantiated or inherited and cannot contain instance constructors. However, the developer can create static constructors to initialize the static members.
- ◆ The code 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.
- ◆ Since there is no need to create objects of the static class to call the required methods, the implementation of the program is simpler and faster than programs containing instance classes.


Classes and Methods / Session 6 **26**


Static Classes 2-3

Snippet

```
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();
    }
}
```


Classes and Methods / Session 6 **27**

Static Classes 3-3

- ◆ In the code:
 - ❖ 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
```

© Aptech Ltd. Classes and Methods / Session 6 28

In slide 26, explain static classes to the students.

Explain the students that the 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.

Tell them that it is not possible to create an instance of a static class using the `new` keyword.

The main features of static classes are as follows:

- They can only contain static members.
- They cannot be instantiated.
- They cannot be inherited.
- They cannot contain instance constructors. However, the developer can create static constructors to initialize the static members.

Mention that since there is no need to create objects of the static class to call the required methods, the implementation of the program is simpler and faster than programs containing instance classes.

Tell the students that the code shown 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.

In slide 27, show the code that creates a static class **Product** having static variables `_productId` and `price` and a static method called `Display()`.

Following are rules to create a static class:

- a) A static class must contain the static keyword in the class declaration.
- b) A method should be static i.e. the method declaration must contain the static keyword.
- c) The constructor must be static because a static class doesn't allow an instance constructor.
- d) The constructor must not contain an access modifier.
- e) A static class can't implement an interface.
- f) A static class can't be a base class i.e. it cannot be derived from one class to another.

In slide 28, tell the students that in the code, 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.

Then, show the output of the code to the students.

With this slide you have finished explaining static classes to the students.

Slides 29 to 31

Understand static methods.


Static Methods 1-3

- ◆ A method is called using an object of the class but it is possible for a method to be called without creating any objects of the class 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 but can refer to non-static methods and variables by using an instance of the class.
- ◆ The following syntax is used to create a static method:

Syntax

```
static<return_type><MethodName>()
{
    // body of the method
}
```


Classes and Methods / Session 6 **29**

Static Methods 2-3

- ◆ The following code creates a static method `Addition()` in the class `Calculate` and then invokes it in another class named `StaticMethods`:

Snippet

```
using System;

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);
    }
}
```


Classes and Methods / Session 6 **30**

Static Methods 3-3

- In the code, 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 operations are displayed in the console window:

Output

```
60
200
```

- The following figure displays invoking a static method:

Class → 'Calculate'
Method → 'Addition' and 'Multiply'

Calculate objCal=new Calculate();
objCal.Multiply(10,20);

Invoking a Normal Method

Calculate.Addition(10,50);

Invoking a Static Method

Classes and Methods / Session 6 31

In slide 29, explain static methods.

Explain the students that 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.

Tell them that a static method is declared using the `static` keyword.

To understand this, give an example to the students.

Tell them that 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.

Then, show the syntax that is used to create a static method.

In slide 30, show the code that creates a static method **Addition()** in the class **Calculate** and then invokes it in another class named **StaticMethods**.

In slide 31, tell the students that in the code, the static method **Addition()** is invoked using the class name whereas the **Multiply()** method is invoked using the instance of the class.

Finally, mention that the results of the addition and multiplication operation are displayed in the console window. Show the figure that displays the process of invoking a static method to the students.

Slide 32

Understand static variables.

Static Variables

- ◆ In addition to static methods, you can also have static variables in C#.
- ◆ A static variable is a special variable that is accessed without using an object of a class.
- ◆ A variable is declared as static using the static keyword. When a static variable is created, it is automatically initialized before it is accessed.
- ◆ Only one copy of a static variable is shared by all the objects of the class.
- ◆ Therefore, a change in the value of such a variable is reflected by all the objects of the class.
- ◆ An instance of a class cannot access static variables. Figure 6.8 displays the static variables.
- ◆ The following figure displays the static variables:

```
class Employee
{
    public static int EmpId = 20 ;
    public static string EmpName = "James";

    static void Main (string[] args)
    {
        Console.WriteLine ("Employee ID: " + EmpId);
        Console.WriteLine ("Employee Name: " + EmpName);
    }
}
```

© Aptech Ltd. Classes and Methods / Session 6 32

In slide 32, tell the students that in addition to static methods, they 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.

Explain them that 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.

Show the figure that displays the static variables `EmpID` and `EmpName` to the students.

Slide 33

Understand access modifiers.



Access Modifiers 1-4

- ◆ C# provides you with access modifiers that allow you to specify which classes can access the data members of a particular class.
- ◆ In C#, there are four commonly used access modifiers.



- ◆ These are described 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.

The following code declares a `public` string variable called `Name` to store the name of the person which can be publicly accessed by any other class:

```

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.

In slide 33, explain the various access modifiers.

Tell the students that the object-oriented programming enables you to restrict access of data members defined in a class so that only specific classes can access them. To specify these restrictions, C# provides you with access modifiers that allow you to specify which classes can access the data members of a particular class. Mention that the access modifiers are specified using C# keywords.

Explain to the students that the four commonly used access modifiers are `public`, `private`, `protected`, and `internal`. Explain these four modifiers to the students.

Tell the students that 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.

Then, show the code that 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.

Tell the students that the `private` access modifier provides the least permissive access level. Private members are accessible only within the class in which they are declared.

Slide 34

Understand the **protected** modifier.

The slide has a teal header bar with the title "Access Modifiers 2-4". Below the header, there are two sections, each containing a "Snippet" heading and a code block.

Snippet (Private Access Modifier):

```
class Employee
{
    // No access restrictions.
    public string Name = "Wilson";
}
```

Snippet (Protected Access Modifier):

```
class Employee
{
    // Accessible only within the class
    protected float _salary;
}
```

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Classes and Methods / Session 6", and the number "34".

In slide 34, show the code that declares a variable called **_salary** as **private**, which means it cannot be accessed by any other class except for the **Employee** class.

Tell the students that the **protected** access modifier allows the class members to be accessible within the class as well as within the derived classes.

Slide 35

Understand the `internal` modifier.

Access Modifiers 3-4

- The following code declares a variable called **Salary** as protected, which means it can be accessed only by the **Employee** class and its derived classes:

Snippet

```
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. The code declares a variable called **NumOne** as `internal`, which means it has only assembly-level access.

Snippet

```
public class Sample
{
    // Only accessible within the same assembly
    internal static int NumOne = 3;
}
```

© Aptech Ltd. Classes and Methods / Session 6 35

In slide 35 show the code that declares a variable called **Salary** as `protected`, which means it can be accessed only by the **Employee** class and its derived classes.

Tell the students that 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.

Then, show the code that declares a variable called **NumOne** as `internal`, which means it has only assembly-level access.

Additional Information

Tell the students that `internal` is a keyword introduced only in C#. Other languages do not have this keyword. This keyword allows us to declare a variable that allows the other classes within current assembly only to be accessible. Any other class outside the current assembly will not be able to access the variable declared with `internal` keyword as access specifier. Tell the students that `internal` keyword is also accessible in the derived class.

Slide 36

Understand the figure that displays the various accessibility levels.

The slide has a blue header bar with the title "Access Modifiers 4-4". Below the title is a green decorative graphic featuring a stylized face and a gear. The main content is a table titled "Accessibility Levels" with three columns: "Applicable to the Application", "Applicable to the Current Class", and "Applicable to the Derived Class". The rows represent access modifiers: public, private, protected, and internal. A vertical legend on the left indicates that green checkmarks mean "Applicable" and red X's mean "Not Applicable".

Access Modifiers	Accessibility Levels		
	Applicable to the Application	Applicable to the Current Class	Applicable to the Derived Class
public	✓	✓	✓
private	✗	✓	✗
protected	✗	✓	✓
internal	✗	✓	✓

In slide 36, show the figure that displays the various accessibility levels.

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.

Slides 37 to 39

Understand the `ref` and `out` keywords.

ref and out Keywords 1-7

- The `ref` keyword causes arguments to be passed in a method by reference.
- In call by reference, the called method changes the value of the parameters passed to it from the calling method.
- Any changes made to the parameters in the called method will be reflected in the parameters passed from the calling method when control passes back to the calling method.
- It is necessary that both the called method and the calling method must explicitly specify the `ref` keyword before the required parameters.
- The variables passed by reference from the calling method must be first initialized.
- The following syntax is used to pass values by reference using the `ref` keyword.

Syntax

```
<access_modifier><return_type><MethodName> (ref parameter1, ref parameter2,
parameter3, parameter4, ...parameterN)
{
// actions to be performed
}
```

where,

`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.

© Aptech Ltd. Classes and Methods / Session 6 37

ref and out Keywords 2-7

- The following code uses the `ref` keyword to pass the arguments by reference:

Snippet

```
using System;

classRefParameters
{
    static void Calculate(ref int numValueOne, ref int
numValueTwo)
    {
        numValueOne = numValueOne * 2;
        numValueTwo = numValueTwo / 2;
    }
    static void Main(string[] args)
    {
        intnumOne = 10;
        intnumTwo = 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);
    }
}
```

© Aptech Ltd. Classes and Methods / Session 6 38

ref and out Keywords 3-7

- ◆ In the code:
 - ❖ 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()**.

In slide 37, tell the students that 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.

Tell them that 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.

Show the syntax used to pass values by reference using the **ref** keyword and tell the students that 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.

In slide 38, show the code that uses the **ref** keyword to pass the arguments by reference.

In slide 39, explain the code to the students.

Explain the students that in the code, 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()**.

Slides 40 to 43

Understand the `ref` and `out` keywords.

ref and out Keywords 4-7

- ◆ The following figure displays the use of `ref` keyword:

```
Value of Num1 and Num2 before calling method 10, 2
Value of Num1 and Num2 after calling method 20, 10
Press any key to continue . . .
```

- ◆ 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.

ref and out Keywords 5-7

- ◆ The following syntax is used to pass values by reference using the `out` keyword:

Syntax

```
<access_modifier><return_type><MethodName> (out parameter1, out
parameter2, ...parameterN)
{
// actions to be performed
}
```

- ◆ where,
 - parameter 1...parameterN: Specifies that there can be any number of parameters and it is not necessary for all the parameters to be `out` parameters.

ref and out Keywords 6-7

- The following code uses the `out` keyword to pass the parameters by reference:

Snippet

```
using System;
classOutParameters
{
    static void Depreciation(out intval)
    {
        int 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);
    }
}
```

© Aptech Ltd. Classes and Methods / Session 6 42

ref and out Keywords 7-7

- In the code:
 - 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.
- The following figure shows the use of `out` keyword:

© Aptech Ltd. Classes and Methods / Session 6 43

In slide 40 show the figure that displays the use of `ref` keyword.

Then, explain the `out` keyword to the students.

Tell them that 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.

In slide 41, show the syntax that is used to pass values by reference using the `out` keyword and explain to the students that the 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.

In slide 42, show the code that uses the `out` keyword to pass the parameters by reference.

In slide 43, explain the code.

Tell the students that in the code 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.

Then, show the figure that displays the use of `out` keyword.

With this slide you have finished explaining the `ref` and `out` Keywords to the students.

In-Class Question:

After you finish explaining the `ref` and `out` Keywords, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which are the four commonly used access modifiers?

Answer:

The four commonly used access modifiers are `public`, `private`, `protected`, and `internal`.

Slides 44 to 46

Understand method overloading in C#.

Method Overloading in C# 1-3

- ◆ In object-oriented programming, every method has a signature which includes:
 - 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, but 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.

© Aptech Ltd. Classes and Methods / Session 6 44

Method Overloading in C# 2-3

- ◆ The following figure 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;
}

```

 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#

© Aptech Ltd. Classes and Methods / Session 6 45

Method Overloading in C# 3-3

- The following code overloads the **Square()** method to calculate the square of the given **int** and **float** values:

Snippet

```
using System;
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 the code:
 - Two methods with the same name but with different parameters are declared in the class.
 - The two **Square()** methods take parameters of **int** type and **float** type respectively.
 - Within the **Main()** method, depending on the type of value passed, the appropriate method is invoked and the square of the specified number is displayed in the console window.

Output

```
Square of integer value 25
Square of float value 6.25
```

Classes and Methods / Session 6 46

In slide 44, introduce method overloading in C#.

Explain the following scenario to the students.

Here, tell the students to consider a holiday resort having two employees with the same name, Peter. One is the chef while the other is in charge of maintenance. The delivery man knows which Peter does what, so if there is a delivery of fresh vegetables for a Mr. Peter, it is forwarded to the chef. Similarly, if there is a delivery of electric bulbs and wires for a Mr. Peter, it is forwarded to the maintenance person.

Tell them that though there are two Peters, the items delivered give a clear indication to which Peter the delivery has to be made.

Tell them that similarly, in C#, two methods can have the same name as they can be distinguished by their signatures. This feature is called method overloading.

Then tell the students that in object-oriented programming, every method has a signature. This comprises the number of parameters passed to the method, the data types of parameters, and the order in which the parameters are written. While declaring a method, the signature of the method is written in parentheses next to the method name.

Tell them that 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.

Mention that the concept of declaring more than one method with the same method name but different signatures is called method overloading.

Method overloading involves having more than one method varying number or type of parameters.

In slide 45, `Addition ()` method is provided with same name but with different number of parameters.

In slide 46, show the code that overloads the `Square ()` method to calculate the square of the given `int` and `float` values. Then, explain the code.

Explain the students that in the code, 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.

Then, show the code output to the students.

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

Slide 47

Understand the guidelines and restrictions to be followed while overloading methods in a program.

Guidelines and Restrictions

- ◆ Guidelines to be followed while overloading methods in a program, to ensure that the overloaded methods function accurately are as follows:

The methods to be overloaded should 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.

In slide 47, explain the guidelines to be followed while overloading methods in a program. Tell the students that while overloading methods in a program, they should follow certain guidelines to ensure that the overloaded methods function accurately.

Explain the following guidelines to the students:

- 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.

Slides 48 to 50

Understand this keyword.

The this keyword 1-3

- ◆ The `this` keyword is used to refer to the current object of the class to resolve conflicts between variables having same names and to pass the current object as a parameter.
- ◆ You cannot use the `this` keyword with static variables and methods.

© Aptech Ltd. 2011-12

Classes and Methods / Session 6 48

The this keyword 2-3

- In the following code, the `this` keyword refers to the `_length` and `_breadth` fields of the current instance of the class `Dimension`:

Snippet

```
using System;

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));
    }
}
```

The this keyword 3-3

- In the code:
 - 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.
- The following figure displays the use of the `this` keyword:

In slide 48, tell the students that 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.

Mention that the students cannot use the `this` keyword with static variables and methods.

In slide 49, show the code where the `this` keyword refers to the `_length` and `_breadth` fields of the current instance of the class `Dimension`.

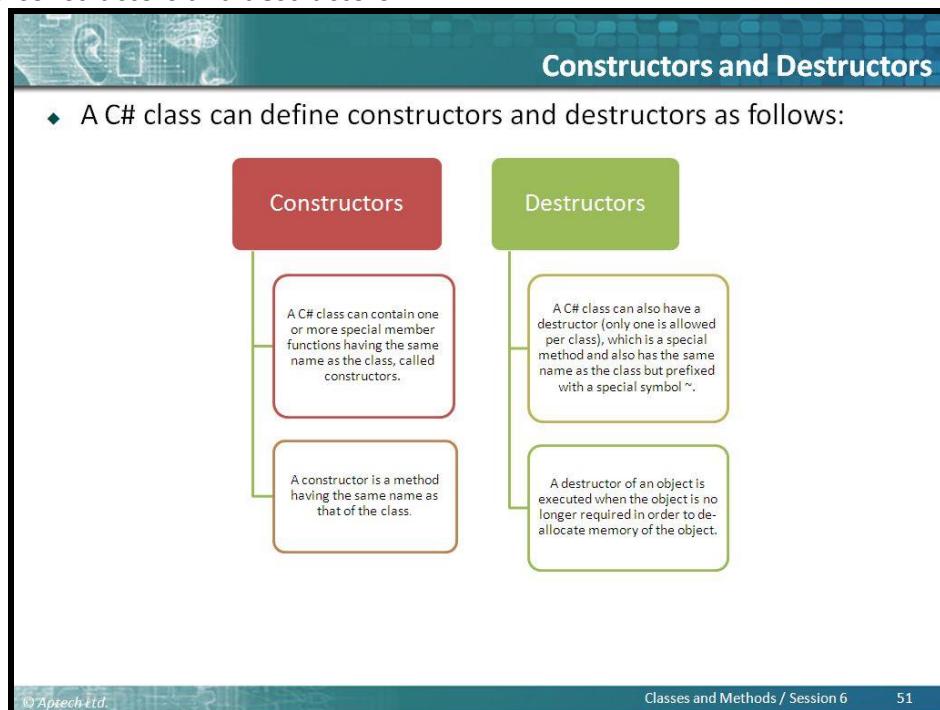
In slide 50, explain the code that uses the `this` keyword to refer to the `_length` and `_breadth` fields of the current instance of the class `Dimension`.

Explain to the students that in the code, 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.

Then, show the figure that displays the use of **this** keyword.

Slide 51

Understand constructors and destructors.



In slide 51, explain about constructors and destructors to the students.

Tell the students that 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.

Tell them that 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.

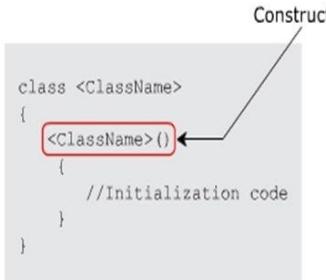
Tell the students that we can have many constructors for a single class but only one destructor per class is allowed. The reason for having many constructors for a class is to allow multiple ways of instantiating objects of that class. Different user may have different requirements and he may want to pass different parameters at different time. This can be possible by having multiple constructors. Since the destructor is called when the object is destroyed so we cannot have many destructors for a class.

Slides 52 to 55

Understand constructors in detail.


Constructors 1-4

- ◆ 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.
- ◆ The following figure shows the constructor declaration:



```
class <ClassName>
{
    <ClassName>()
    {
        //Initialization code
    }
}
```


Constructors 2-4

- ◆ It is possible to specify the accessibility level of constructors within an application by using access modifiers such as:
 - ◆ **public**: Specifies that the constructor will be called whenever a class is instantiated. This instantiation can be done from anywhere and from any assembly.
 - ◆ **private**: Specifies that this constructor cannot be invoked by an instance of a class.
 - ◆ **protected**: Specifies that the base class will initialize on its own whenever its derived classes are created. Here, the class object can only be created in the derived classes.
 - ◆ **internal**: Specifies that the constructor has its access limited to the current assembly. It cannot be accessed outside the assembly.
- ◆ The following code creates a class **Circle** with a **private** constructor:

Snippet

```
using System;
public class Circle
{
    private Circle()
    {
    }
}
class CircleDetails
{
    public static void Main(string[] args)
    {
        Circle objCircle = new Circle();
    }
}
```


Classes and Methods / Session 6 **52**


Classes and Methods / Session 6 **53**

Constructors 3-4

- ◆ In the code:
 - ❖ 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.
- ◆ The following figure shows the output for creating a class **Circle** with a private constructor:

Output

```
C:\WINDOWS\system32\cmd.exe
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

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

© Aptech Ltd. Classes and Methods / Session 6 54

Constructors 4-4

- ◆ The following code used to initialize the values of **_empName**, **_empAge**, and **_deptName** with the help of a constructor:

Snippet

```
using System;

class Employees
{
    string _empName;
    int _empAge;
    string _deptName;
    Employees(string name, intnum)
    {
        _empName = name;
        _empAge = num;
        _deptName = "Research & Development";
    }
    static void Main(string[] args)
    {
        Employees objEmp = new Employees("John", 10);
        Console.WriteLine(objEmp._deptName);
    }
}
```

- ◆ In the code:
 - ❖ 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.

© Aptech Ltd. Classes and Methods / Session 6 55

Use slide 52 to tell the students that a class can contain multiple variables whose declaration and initialization becomes difficult to track if they are done within different blocks. Likewise, there may be other startup operations that need to be performed in an application such as opening a file and so forth. To simplify these tasks, a constructor is used.

Explain them that 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.

Then show the figure that displays the constructor declaration.

In slide 53, tell the students that 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**. Explain these access modifiers as follows:

- **public**: Specifies that the constructor will be called whenever a class is instantiated. This instantiation can be done from anywhere and from any assembly.
- **private**: Specifies that this constructor cannot be invoked by an instance of a class.
- **protected**: Specifies that the base class will initialize on its own whenever its derived classes are created. Here, the class object can only be created in the derived classes.
- **internal**: Specifies that the constructor has its access limited to the current assembly. It cannot be accessed outside the assembly.

Then, show the code that creates a class **Circle** with a private constructor.

Use slide 54 to tell the students that in the code, 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.

Explain them that the 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, mention that the private constructors are only used if a class contains only static data members. This is because static members are invoked using the class name.

Show the output of the code for creating a class **Circle** with a private constructor to the students.

Use slide 55 to show the code used to initialize the values of **_empName**, **_empAge**, and **_deptName** with the help of a constructor. Then, explain the code.

Tell the students that in the code, 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.

With this slide you have finished explaining constructors to the students.

Slide 56

Understand default and static constructors.

Default Constructors

- C# creates a default constructor for a class if no constructor is specified within the class.
- The default constructor automatically initializes all the numeric data type instance variables of the class to zero.
- If you define a constructor in the class, the default constructor is no longer used.

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.
- 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.

Classes and Methods / Session 6 56

In slide 56, explain default and static constructors.

Tell the students that C# creates a default constructor for a class if no constructor is specified within the class. The default constructor automatically initializes all the numeric data type instance variables of the class to zero. If you define a constructor in the class, the default constructor is no longer used.

Tell them that 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. Mention that the students can have only one static constructor in the class. The static keyword is used to declare a constructor as static.

Explain them that 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.

To demonstrate further, you can create two static constructors, compile the code and show the error that is displayed by the user.

Slides 57 and 58

Understand static constructors.

Static Constructors 1-2

- The following figure illustrates the syntax for a static constructor:

```
class <ClassName>
{
    static <ClassName>()
    {
        //Initialization code
    }
}
```

Static Constructor

- The following code shows how static constructors are created and invoked.

Snippet

```
using System;

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();
}
```

© Aptech Ltd.

Classes and Methods / Session 6

57

Static Constructors 2-2

- In the code:

- 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
```

© Aptech Ltd.

Classes and Methods / Session 6

58

In slide 57, show the figure that illustrates the syntax for a static constructor.

Then, show the code that displays how static constructors are created and invoked.

Use slide 58 to explain the students that 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.

Show the output of the code to the students.

Slides 59 to 61

Understand constructor overloading.


Constructor Overloading 1-3

- ◆ Declaring more than one constructor in a class is called constructor overloading.
- ◆ The process of overloading constructors is similar to overloading methods where every constructor has a signature similar to that of a method.
- ◆ Multiple constructors in a class can be declared wherein each constructor will have different signatures.
- ◆ Constructor overloading is used when different objects of the class might want to use different initialized values.
- ◆ Overloaded constructors reduce the task of assigning different values to member variables each time when needed by different objects of the class.


Constructor Overloading 2-3

- ◆ The following code demonstrates the use of constructor overloading:

Snippet

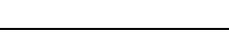
```

using System;
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());
}
}

```


Classes and Methods / Session 6
59


Classes and Methods / Session 6
60

Constructor Overloading 3-3

- ◆ In the code:
 - ❖ 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
```

In slide 59, tell the students that the concept of declaring more than one constructor in a class is called constructor overloading. The process of overloading constructors is similar to overloading methods. Every constructor has a signature similar to that of a method.

Mention that the students can declare multiple constructors in a class wherein each constructor will have different signatures. Constructor overloading is used when different objects of the class might want to use different initialized values. Overloaded constructors reduce the task of assigning different values to member variables each time when needed by different objects of the class.

In slide 60, show the code that demonstrates the use of constructor overloading.

In slide 61, tell the students that 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, mention that the multiplication operation is performed on these variables and the area values are displayed as the output.

Show the output of the code to the students.

Slides 62 to 64

Understand destructors.

Destructors 1-3

- ◆ A destructor is a special method which has the same name as the class but starts with the character ~ before the class name and immediately de-allocate memory of objects that are no longer required. Following are the features of destructors:
 - ◆ Destructors cannot be overloaded or inherited.
 - ◆ Destructors cannot be explicitly invoked.
 - ◆ Destructors cannot specify access modifiers and cannot take parameters.
- ◆ The following code demonstrates the use of destructors:

Snippet

```
using System;

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;
    }
}
```

© Aptech Ltd. Classes and Methods / Session 6 62

Destructors 2-3

```
}

~Employee()
using System;

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");
    }
    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);
    }
}
```

© Aptech Ltd. Classes and Methods / Session 6 63

The slide has a blue header bar with the title 'Destructors 3-3'. The main content area contains the following bullet points:

- ◆ In the code:
 - ❖ The destructor `~Employee` is created having the same name as that of the class and the constructor.
 - ❖ The destructor is automatically called when the object `objEmp` is no longer needed to be used.
 - ❖ However, you have no control on when the destructor is going to be executed.

At the bottom left is the copyright notice '© Aptech Ltd.' and at the bottom right are the page numbers 'Classes and Methods / Session 6' and '64'.

In slide 62, explain destructors.

Explain to the students that 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.

Mention that the students can define only one destructor in a class.

Tell the students that apart from this, destructors have some more features as follows:

- Destructors cannot be overloaded or inherited.
- Destructors cannot be explicitly invoked.
- Destructors cannot specify access modifiers and cannot take parameters.

Explain the code that demonstrates the use of destructors. In slide 63, the class Employee has been created with constructor taking 4 parameters as input, as the object of the class is created constructor will be get initialized.

In slide 64, tell the students that the destructor `~Employee()` is created having the same name as that of the class and the constructor.

The destructor is automatically called when the object `objEmp` is no longer needed to be used.

However, when this will happen cannot be determined and hence, you have no control on when the destructor is going to be executed.

In-Class Question:

After you finish explaining destructors, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



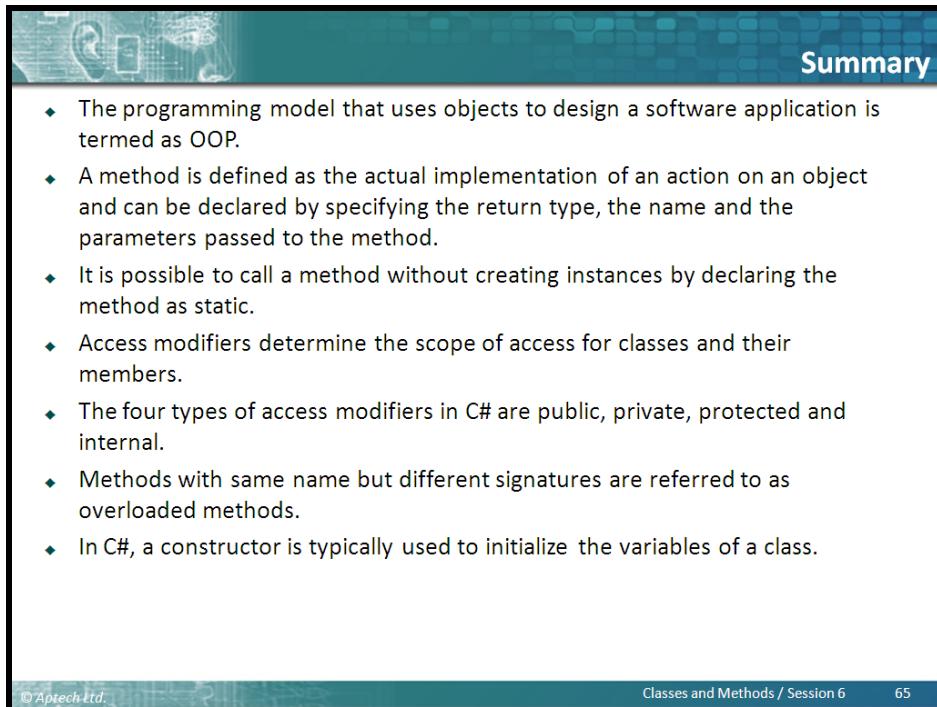
Explain Destructors.

Answer:

A destructor is a special method which has the same name as the class but starts with the character `~` before the class name and immediately de-allocate memory of objects that are no longer required.

Slide 65

Summarize the session.



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.

© Aptech Ltd. Classes and Methods / Session 6 65

In slide 65, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- 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.

6.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. What are classes, how objects are defined in classes?
2. What is OOPS concept/ what are the features oops?

Session 7 - Inheritance and Polymorphism

7.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- 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.

Here, you can ask students the key topics they can recall from previous session. Ask them to briefly explain classes and objects in C#. You can also ask them to explain methods and method overloading.

Furthermore, ask them to explain access modifiers, constructors and destructors.

Prepare a question or two which will be a key point to relate the current session objectives.

7.1.1 Objectives

By the end of this session, the learners will be able to:

- Define and describe inheritance
- Explain method overriding
- Define and describe sealed classes
- Explain polymorphism

7.1.2 Teaching Skills

To teach this session successfully, you must know about inheritance in C#. You should be aware of method overriding and sealed classes. You should also know how to describe polymorphism.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here during the In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to inheritance and polymorphism. They will learn about method overriding and sealed classes.

7.2 In-Class Explanations

Slide 3

Understand inheritance.


Definition of Inheritance 1-2

- ◆ The similarity in physical features of a child to that of its parents is due to the child having inherited these features from its parents.

- ◆ Similarly, in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.

- ◆ The class from which the new class is created is known as the base class and the created class is known as the derived class.

- ◆ The process of creating a new class by extending some features of an existing class is known as inheritance.



Inheritance in Real World

```

graph TD
    SeniorClass[SeniorClass] --> JuniorClass[JuniorClass]
  
```

© Aptech Ltd. | Building Applications Using C# / Session 7 | 3

Use slide 3 to explain that a programmer does not always need to create a class in a C# application from scratch.

Tell that at times, the programmer can create a new class by extending the features of an existing class.

Also, explain that the process of creating a new class by extending some features of an existing class is known as inheritance.

Tell the students that when we extend from any existing class, we need to keep in mind which members we can override and access and which ones we cannot. Normally the public and protected members and functions of the base class are available to derived class but the private members and functions cannot be accessed directly. Tell them that we will learn about this in detail in the next sections to come.

Slide 4

Understand the definition of inheritance.

Definition of Inheritance 2-2

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.
- ◆ The following figure illustrates an example of inheritance:

```

graph TD
    Vehicle[Vehicle] -- "Is a Kind of" --> TwoWheeler[Two-Wheeler Vehicle]
    Vehicle -- "Is a Kind of" --> FourWheeler[Four-Wheeler Vehicle]
    TwoWheeler --> Bike[Bike]
    TwoWheeler --> Bicycle[Bicycle]
    FourWheeler --> Bus[Bus]
    FourWheeler --> Truck[Truck]
  
```

© Aptech Ltd. Building Applications Using C# / Session 7 4

In slide 4, explain to the students that 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.

Tell the students that in C#, inheritance allows you to create a class by deriving the common attributes and methods of an existing class.

Also, tell that 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.

Give an example. Tell that, 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.

You can refer to the figure in slide 4 that illustrates this example.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is inheritance?

Answer:

The process of creating a new class by extending some features of an existing class is known as inheritance.

Slides 5 to 7

Understand purpose of inheritance.



Purpose 1-3

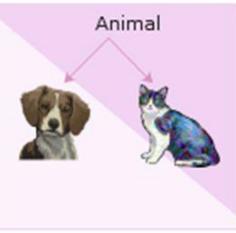
- ◆ The purpose of inheritance is to reuse common methods and attributes among classes without recreating them.
- ◆ Reusability of a code enables you to use the same code in different applications with little or no changes.

Example

- ◆ Consider a class named **Animal** which defines attributes and behavior for animals.
- ◆ If a new class named **Cat** has to be created, it can be done based on **Animal** because a cat is also an animal.
- ◆ Thus, you can reuse the code from the previously-defined class.



Animal Class



Animal

Dog Cat

© Aptech Ltd.
Building Applications Using C# / Session 7
5



Purpose 2-3

- ◆ Apart from reusability, inheritance is widely used for:

Generalization

Specialization

Extension

© Aptech Ltd.
Building Applications Using C# / Session 7
6

Purpose 3-3

- The following figure displays a real-world example demonstrating the purpose of inheritance:

The diagram illustrates inheritance. At the top, a pink rounded rectangle labeled "Generalized" contains the word "Vehicle". A downward-pointing arrow originates from "Vehicle" and points to a larger circle below it. This circle is labeled "Specialized" and contains four distinct icons: a truck, a blue car, a bicycle, and a motorcycle. This visualizes how a single general class ("Vehicle") can serve as the base for more specific classes ("Truck", "Car", "Bike", "Motorcycle").

Use slide 5 to explain the purpose of inheritance is to reuse common methods and attributes among classes without recreating them.

Tell them that reusability of a code enables you to use the same code in different applications with little or no changes.

Give an example. Tell them to consider a class named **Animal** which defines attributes and behavior for animals. If a new class named **Cat** has to be created, it can be done based on **Animal** because cat is also an animal. In such a case, you can reuse the code from the previously-defined class.

Tell the students about inheritance and its advantage in OOPS. Tell them that inheritance is one of the features of OOPS programming that saves our time.

Use slide 6 to tell that inheritance is widely used for generalization, specialization, and extension.

Explain that inheritance implements generalization by creating base classes.

Give an example. Tell them to 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.

Tell that inheritance implements specialization by creating derived classes. Give an example. Tell that the derived classes such as **Bike**, **Bicycle**, **Bus**, and **Truck** are specialized by implementing only specific methods from its generalized base class **Vehicle**.

Tell that inheritance extends the functionalities of a derived class by creating more methods and attributes that are not present in the base class. It provides additional features to the existing derived class without modifying the existing code.

You can refer to the figure in slide 7 that displays a real-world example demonstrating the purpose of inheritance.

Explain to the students about the Vehicle class and its various sub-classes or inherited classes. Show them some common properties and specific properties of all types of vehicles as shown in the figure. Also tell them that all the common properties will be grouped into Vehicle class and specific properties will be specified into derived classes such as Car, Truck, Bike, or Bicycle.

Additional Information

Refer the following links for more information on inheritance:

<http://www.csharp-station.com/Tutorial/CSharp/lesson08>

<http://www.codeproject.com/Articles/1445/Introduction-to-inheritance-polymorphism-in-C>

Slides 8 to 10

Understand multi-level hierarchy.

Multi-level Hierarchy 1-3

- Inheritance allows the programmer to build hierarchies that can contain multiple levels of inheritance.

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.
- The following figure depicts multi-level hierarchy of related classes:

```

graph TD
    Animal[Animal] --> Mammal[Mammal]
    Mammal --> Dog[Dog]
  
```

Multi-level Hierarchy 2-3

- The following code demonstrates multiple levels of inheritance:

Snippet

```

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();
}
  
```

Multi-level Hierarchy 3-3

- In the code, the `Main()` method of the class `Dog` invokes the methods of the class `Animal`, `Mammal`, and `Dog`.

Output

- Every animal eats something.
- Mammals give birth to young ones.
- Dog Barks.

Use slide 8 to explain the students that the inheritance allows the programmer to build hierarchies that can contain multiple levels of inheritance.

Give an example. Ask them to 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.

You can refer to the figure in slide 8 that depicts multi-level hierarchy of related classes.

In slide 9, tell the students that the code demonstrates multiple levels of inheritance.

Use slide 10 to explain the code. Tell that in the code, the `Main()` method of the class `Dog` invokes the methods of the class `Animal`, `Mammal`, and `Dog`.

Explain to the students the output shown in slide 10.

Tell the students how the inheritance works. In the given slide, it is clear that the `Dog` class is based on `Mammal` class which is again based on class `Animal`. When the object of `Dog` class is created it holds all three methods such as `Eat()`, `Features()`, and `Noise()`. Tell them how the `Dog` class is having multi-level inheritance.

Tell them that when the object of `Dog` class is created by activating `Main()` static method, it initializes the `Mammal` class which in turn initializes the class `Animal`. Thus, the object of `Dog` class `objDog` holds all three methods.

Additional Information

Refer the following links for more information on polymorphism:

<http://www.dotnet-tricks.com/Tutorial/oops/JaI0211013-Understanding-Inheritance-and-Different-Types-of-Inheritance.html>

<http://codekingdom.blogspot.in/2009/03/multilevel-inheritance-in-c.html>

Slides 11 to 13

Understand implementing inheritance.

Implementing Inheritance 1-3

- To derive a class from another class in C#, insert a colon after the name of the derived class followed by the name of the base class.
- The derived class can now inherit all non-private methods and attributes of the base class.
- The following syntax is used to inherit a class in C#:

Syntax

```
<DerivedClassName>:<BaseClassName>
```

where,

- DerivedClassName: Is the name of the newly created child class.
- BaseClassName: Is the name of the parent class from which the current class is inherited.

Implementing Inheritance 2-3

- The following syntax is used to invoke a method of the base class:

Syntax

```
<objectName>.<MethodName>;
```

where,

- objectName: Is the object of the base class.
- MethodName: Is the name of the method of the base class.

- The following code demonstrates how to derive a class from another existing class and inherit methods from the base class:

Snippet

```
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();
    }
}
```

Implementing Inheritance 3-3

- ◆ In the code:
 - ❖ 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.
```

Use slide 11 to explain the syntax to derive a class from another class. It is quite simple in C#. Explain that a colon needs to be inserted 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.

Explain the syntax used to inherit a class in C# as given on the slide. Use slide 12 to explain that the syntax is used to invoke a method of the base class.

Tell that the code demonstrates how to derive a class from another existing class and inherit methods from the base class. Use slide 13 to explain the code and the output. Tell that in the code, the class **Animal** consists of two methods, **Eat()** and **DoSomething()**. Then, tell that 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**. Also, tell that 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.

Mention that 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. Tell the students that here **Animal** class is the base class that consists of **Eat()** and **DoSomething()** methods which are common methods of **Animal** class. Now, if any other class is created or derived from **Animal** class will automatically inherit these two methods and will allow the user to add some more variables or methods in the derived class. This is achieved through the OOPS feature called inheritance.

Slides 14 to 16

Understand `protected` access modifier.

protected Access Modifier 1-3

- 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.
- The following figure displays an example of using the `protected` access modifier:

```

graph TD
    BC[Base Class: Animal] -- "Protected Variables" --> PV[• Protected Variables]
    BC -- "Protected Methods" --> PM[• Protected Methods]
    BC -- "Inherits and Can Access" --> DC[Derived Class: Cat]
    BC -- "Cannot Inherit or Access" --> CH[Class: Human]
    DC -- "Protected Variables of Animal Class ✓" --> PV
    DC -- "Protected Methods of Animal Class ✓" --> PM
    CH -- "Protected Variables of Animal Class X" --> PV
    CH -- "Protected Methods of Animal Class X" --> PM
  
```

©Aptech Ltd.

Building Applications Using C# / Session 7

14

protected Access Modifier 2-3

- The following syntax declares a `protected` variable:

Syntax

`protected<data_type><VariableName>;`

where,

- `data_type`: Is the data type of the data member.
- `VariableName`: Is the name of the variable.

- The following syntax declares a `protected` method:

Syntax

`protected<return_type><MethodName>(argument_list);`

where,

- `return_type`: Is the type of value the method will return.
- `MethodName`: Is the name of the method.
- `argument_list`: Is the list of parameters.

©Aptech Ltd.

Building Applications Using C# / Session 7

15

protected Access Modifier 3-3

- The following code demonstrates the use of the `protected` access modifier:

Snippet

```
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 the code:
 - Two variables are created in the class `Animal` with the `protected` keyword.
 - The class `Cat` is inherited from the class `Animal`.
 - The instance of the class `Cat` is created that is referring the two variables defined in the class `Animal` using the dot(.) operator.
 - The `protected` access modifier allows the variables declared in the class `Animal` to be accessed by the derived class `Cat`.

Output

```
Cat loves to eat Mouse.
The Cat loves to laze around.
```

© Aptech Ltd. Building Applications Using C# / Session 7 16

In slide 14, explain that the `protected` access modifier protects the data members that are declared using this modifier.

Tell that 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.

You can refer to the figure in slide 14 that displays an example of using the `protected` access modifier.

Use slide 15 to explain that the syntax as given on the slide.

In slide 16, explain that the code demonstrates use of the `protected` access modifier.

Then, explain the code and the output. Tell that in the code, two variables are created in the class `Animal` with the `protected` keyword.

Also, tell that the class `Cat` is inherited from the class `Animal`. The instance of the class `Cat` is created that is referring the two variables defined in the class `Animal` using the dot(.) operator. The `protected` access modifier allows the variables declared in the class `Animal` to be accessed by the derived class `Cat`.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is a `protected` access modifier?

Answer:

The protected access modifier protects the data members that are declared using this modifier. It is specified using the **protected** keyword.

Slides 17 to 19

Understand **base** keyword.

base Keyword 1-3

- ◆ The **base** keyword allows you to do the following:
 - Access the variables and methods of the base class from the derived class.
 - Re-declare the methods and variables defined in the base class.
 - Invoke the derived class data members.
 - Access the base class members using the **base** keyword.

base Keyword 2-3

- ◆ The following syntax shows the use of the **base** keyword:

Syntax
<pre>class <ClassName> { <accessmodifier><returntype><BaseMethod>() } class <ClassName1>:<ClassName> { base.<BaseMethod>; }</pre>

where,

- ◆ **<ClassName>**: Is the name of the base class.
- ◆ **<accessmodifier>**: Specifies the scope of the class or method.
- ◆ **<returntype>**: Specifies the type of data the method will return.
- ◆ **<BaseMethod>**: Is the base class method.
- ◆ **<ClassName1>**: Is the name of the derived class.
- ◆ **base**: Is a keyword used to access the base class members.

base Keyword 3-3

- The following figure displays an example of using the `base` keyword:

```
class Animal
{
    public void Eat() {} ←
}
class Dog : Animal
{
    → public void Eat() {} ←
    public static Main(string args[])
    {
        Dog objDog = new Dog();
        objDog.Eat();
        base.Eat(); ←
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 7 19

In slide 17, explain that the `base` keyword accesses the variables and methods of the base class from the derived class.

Tell that when to inherit a class, the methods and variables defined in the base class can be re-declared in the derived class.

Also, tell that when methods or access variables are invoked, the derived class data members are invoked and not data members of the base class.

Mention that in such situations, access the base class members using the `base` keyword.

In slide 18, explain the syntax used to specify the `base` keyword as given on the slide.

Use slide 19 to refer to figure that displays an example of using the `base` keyword.

Slides 20 and 21

Understand the `new` keyword.

The slide has a blue header bar with the title "new Keyword 1-2". Below the header, there is a bulleted list:

- ◆ The `new` keyword can either be used as an operator or as a modifier in C#.

Two circular callouts provide detailed information:

- new Operator**: Instantiates a class by creating its object which invokes the constructor of the class.
- Modifier**: Hides the methods or variables of the base class that are inherited in the derived class.

Below the callouts, there is another bulleted list:

- ◆ This allows you to redefine the inherited methods or variables in the derived class.
- ◆ Since redefining the base class members in the derived class results in base class members being hidden, the only way you can access these is by using the `base` keyword.

© Aptech Ltd.

Building Applications Using C# / Session 7

20

The slide has a blue header bar with the title "new Keyword 2-2". Below the header, there is a bulleted list:

- ◆ The following syntax shows the use of the `new` modifier:

A **Syntax** box contains the following code snippet:

```
<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.
- ◆ `<returntype>`: 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.

- ◆ The following code creates an object using the `new` operator:

A **Snippet** box contains the following code snippet:

```
EmployeesobjEmp=newEmployees();
```

- ◆ Here, the code creates an instance called `objEmp` of the class `Employees` and invokes its constructor.

© Aptech Ltd.

Building Applications Using C# / Session 7

21

Use slide 20 to explain that the `new` keyword can either be used as an operator or as a modifier in C#.

Tell that the `new operator` is used to instantiate a class by creating its object that 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.

Also, tell that this allows redefining of the inherited methods or variables in the derived class.

Explain that since redefining the base class members in the derived class results in base class members being hidden, the only way you can access these is by using the `base` keyword.

In slide 21, explain that the syntax shows the use of the `new` modifier.

Tell that the code creates an object using the `new` operator.

Mention that the code creates an instance called `objEmp` of the class `Employees` and invokes its constructor.

Slides 22 and 23

Understand how to use the `new` keyword.

The following code demonstrates the use of the `new` modifier to redefine the inherited methods in the `base` class:

Snippet

```
class Employees
{
    int _empId=1;
    string _empName="JamesAnderson";
    int _age=25;
    public void Display()
    {
        Console.WriteLine("EmployeeID:"+_empId);
        Console.WriteLine("EmployeeName:"+_empName);
    }
}
class Department:Employees
{
    int _deptId=501;
    string _deptName="Sales";
    new void Display()
    {
        base.Display();
        Console.WriteLine("DepartmentID:"+_deptId);
        Console.WriteLine("DepartmentName:"+_deptName);
    }
    static void Main(string[] args)
    {
        Department objDepartment=new Department();
        objDepartment.Display();
    }
}
```

- ◆ In the code:
 - ◆ The class `Employees` declares a method called `Display()`.
 - ◆ This method is inherited in the derived class `Department` and is preceded by the `new` keyword.
 - ◆ The `new` keyword hides the inherited method `Display()` that was defined in the base class, thereby executing the `Display()` method of the derived class when a call is made to it.
 - ◆ However, the `base` keyword allows you to access the base class members.
 - ◆ Therefore, the statements in the `Display()` method of the derived class and the base class are executed, and, finally, the employee ID, employee name, department ID, and department name are displayed in the console window.

Output

```
Employee ID: 1
Employee Name: James Anderson
Department ID: 501
Department Name: Sales
```

Use slides 22 and 23 to explain the code and the output. Explain that in the code, the class **Employees** declares a method called **Display()**. This method is inherited in the derived class **Department** and is preceded by the **new** keyword.

Then, tell that 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.

Also, tell that the **base** keyword allows 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.

Slides 24 to 26

Understand constructor inheritance.

Constructor Inheritance 1-3

- ◆ In C#, you can:
 - Invoke the base class constructor by either instantiating the derived class or the base class.
 - Invoke the constructor of the base class followed by the constructor of the derived class.
 - Invoke the base class constructor by using the base keyword in the derived class constructor declaration.
 - Pass parameters to the constructor.
- ◆ However, C# cannot inherit constructors similar to how you inherit methods.
- ◆ The following figure displays an example of constructor inheritance:

```

graph TD
    subgraph Base_Class_Animal [Base Class: Animal]
        direction TB
        A[Invoked First] --> B[Animal()]
        B --> C[{}]
        C --> D[{}]
        D --> E[base keyword]
        E --> F[Derived Class: Cat]
        F --> G[Cat()]
        G --> H[{}]
        H --> I[public void Create()]
        I --> J[Cat objCat = new Cat();]
        J --> K[{}]
        K --> L[{}]
    end
    subgraph Derived_Class_Cat [Derived Class: Cat]
        direction TB
        M[Invoked Second] --> N[Cat()]
        N --> O[{}]
        O --> P[public void Create()]
        P --> Q[Cat objCat = new Cat();]
        Q --> R[{}]
        R --> S[{}]
    end
  
```

The diagram shows two classes: Animal and Cat. The Animal class has a constructor that is invoked first. The Cat class has a constructor that is invoked second. The Cat class also contains a Create() method that creates an instance of the Animal class.

© Aptech Ltd. Building Applications Using C# / Session 7 24

Constructor Inheritance 2-3

- ◆ The following code explicitly invokes the base class constructor using the base keyword:

Snippet

```

class Animal
{
    public Animal()
    {
        Console.WriteLine("Animal constructor without parameters");
    }
    public Animal(Stringname)
    {
        Console.WriteLine("Animal constructor with a string parameter");
    }
}
class Canine:Animal
{
    //base() takes a string value called "Lion"
    public Canine():base("Lion")
    {
        Console.WriteLine("DerivedCanine");
    }
}
class Details
{
    static void Main(String[] args)
    {
        Canine objCanine=new Canine();
    }
}
  
```

© Aptech Ltd. Building Applications Using C# / Session 7 25

Constructor Inheritance 3-3

- ◆ In the code:
 - ❖ 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

In slide 24, explain that in C#, constructors cannot be inherited.

Tell that the base class constructor can be invoked 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.

Also, mention that the base class constructor can be invoked by using the **base** keyword in the derived class constructor declaration. The **base** keyword allows you to pass parameters to the constructor.

You can refer to the figure in slide 24 that displays an example of constructor inheritance.

Use slide 25 to tell that the code explicitly invokes the base class constructor using the **base** keyword.

In slide 26, explain the code and the output. Tell that in the code, 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**.

Then tell that 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.

Also, mention that in the class **Details**, when the derived class constructor is invoked, it will in turn invoke the parameterized constructor of the base class.

Tell the students that the class `Animal` consists of two constructors. This is an example of overloaded constructors. In this case the constructor without any parameters is known as default constructor. If the object is created without any parameter then this default constructor will be activated. If the user passes a string parameter at the time of creation of the object then automatically the constructor with parameter will be called.

Slides 27 to 29

Understand invoking parameterized base class constructors.

The slide has a blue decorative header bar with icons of a smartphone, laptop, and gear. The title 'Invoking Parameterized Base Class Constructors 1-3' is centered in white text. The main content area is white with a black border. At the bottom, there is a footer bar with the text '© Aptech Ltd.', 'Building Applications Using C# / Session 7', and '27'.

- ◆ 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.

Invoking Parameterized Base Class Constructors 2-3

- The following code demonstrates how parameterized constructors are invoked in a multi-level hierarchy:

Snippet

```
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("CastIron")
    {
        _part=part;
        Console.WriteLine("Part:\t\t"+_part);
    }
    static void Main(string[] args)
    {
        Automobiles objAutomobiles=new Automobiles("Chassies");
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 7 28

Invoking Parameterized Base Class Constructors 3-3

- In the code:
 - 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: CastIron
- Part: Chassies

© Aptech Ltd. Building Applications Using C# / Session 7 29

Use slide 27 to explain that the derived class constructor can explicitly invoke the base class constructor by using the **base** keyword.

Tell that if a base class constructor has a parameter, the **base** keyword is followed by the value of the type specified in the constructor declaration.

Also, mention that if there are no parameters, the `base` keyword is followed by a pair of parentheses.

In slide 28 explain the code that demonstrates how parameterized constructors are invoked in a multi-level hierarchy. Use slide 29 to explain the code and the output. Tell that in the code, the **Automobiles** class inherits the **SteelCompany** class. The **SteelCompany** class inherits the **Metals** class.

Then, tell that 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.

Also, tell that the constructor of the **Automobiles** class is invoked.

Tell the students, this is also an example of multi-level inheritance. Here, the class **Automobiles** is derived from class **SteelCompany** which in turn is derived from the base class **Metals**.

Slide 30

Understand method overriding.

The diagram shows two classes: **Hardware** and **Monitor**. The **Hardware** class has methods `TurnOn()` and `TurnOff()`. The **Monitor** class also has methods `TurnOn()` and `TurnOff()`. An arrow from `TurnOn()` in **Hardware** to `TurnOn(string ans)` in **Monitor** is labeled "This is Not an Override due to Different Signatures". An arrow from `TurnOff()` in **Hardware** to `TurnOff()` in **Monitor** is labeled "This is an Override Feature".

Method Overriding

- ◆ Method overriding:
 - ❖ Is a feature that allows the derived class to override or redefine the methods of the base class which changes the body of the method that was declared in the base class.
 - ❖ Allows the same method with the same name and signature declared in the base class to be reused in the derived class to define a new behavior.
 - ❖ Ensures reusability while inheriting classes.
 - ❖ Is implemented in the derived class from the base class is known as the Overridden Base Method.
- ◆ The following figure depicts method overriding:

© Aptech Ltd. Building Applications Using C# / Session 7 30

In slide 30, tell that the method overriding is a feature that allows the derived class to override or redefine the methods of the base class.

Then, tell that 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.

You can refer to the figure in slide 30 that displays method overriding.

Additional Information

Refer the following links for more information on method overriding:

<http://www.codeproject.com/Articles/18734/Method-Overriding-in-C>

<http://www.dotnet-tricks.com/Tutorial/csharp/U33Y020413-Understanding-virtual,-override-and-new-keyword-in-C>

Slide 31

Understand `virtual` and `override` keywords.

virtual and override Keywords 1-5

- ◆ You can override a base class method in the derived class using appropriate C# keywords such as:
 - To override a particular method of the base class in the derived class, you need to declare the method in the base class using the `virtual` keyword.
 - A method declared using the `virtual` keyword is referred to as a virtual method.
 - In the derived class, you need to declare the inherited virtual method using the `override` keyword.
 - In the derived class, you need to declare the inherited virtual method using the `override` keyword which 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.

© Aptech Ltd. Building Applications Using C# / Session 7 31

Use slide 31 to explain that to override a base class method in the derived class, appropriate C# keywords such as `virtual` and `override` must be used.

Tell that to override a particular method of the base class in the derived class, 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.

Also, tell that in the derived class, 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.

Slides 32 and 33

Understand using the `virtual` keyword.

virtual and override Keywords 2-5

- The following is the syntax for declaring a virtual method using the `virtual` keyword:

Syntax

```
<access_modifier>virtual<return_type><MethodName>(<parameter-list>);
```

where,

- `access_modifier`: Is the access modifier of the method, which can be `private`, `public`, `protected`, or `internal`.
- `virtual`: Is a keyword used to declare a method in the base class that can be overridden by the derived class.
- `return_type`: Is the type of value the method will return.
- `MethodName`: Is the name of the virtual method.
- `parameter-list`: Is the parameter list of the method; it is optional.

© Aptech Ltd.

Building Applications Using C# / Session 7 32

virtual and override Keywords 3-5

- The following is the syntax for overriding a method using the `override` keyword:

Syntax

```
<accessmodifier>override<returntype><MethodName>(<parameters-list>)
```

where,

- `override`: Is the keyword used to override a method in the derived class.

© Aptech Ltd.

Building Applications Using C# / Session 7 33

In slide 32, explain the syntax for declaring a virtual method using the `virtual` keyword as given on the slide. Use slide 33 to explain using the `override` keyword where, `override` is the keyword used to override a method in the derived class.

Slides 34 and 35

Understand **virtual** and **override** keywords in the base and derived classes.

virtual and override Keywords 4-5

- The following code demonstrates the application of the **virtual** and **override** keywords in the base and derived classes respectively:

Snippet

```
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();
    }
}
```

virtual and override Keywords 5-5

- In the code:
 - 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
```

Use slide 34 to tell that the code demonstrates the application of the **virtual** and **override** keywords in the base and derived classes.

In slide 35, explain the code and the output.

Tell that in the code, the class **Animal** consists of two methods, the **Eat()** method with the **virtual** keyword and the **DoSomething()** method with the **protected** keyword.

Also, tell that 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.

Mention that 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.

Tips:

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.

Slides 36 to 38

Understand calling the base class method.

Calling the Base Class Method 1-3

- ◆ Method overriding allows the derived class to redefine the methods of the base class.
- ◆ It allows the base class methods to access the new method but not the original base class method.
- ◆ To execute the base class method as well as the derived class method, you can create an instance of the base class.
- ◆ It allows you to access the base class method, and an instance of the derived class, to access the derived class method.

© Aptech Ltd. Building Applications Using C# / Session 7 36

Calling the Base Class Method 2-3

- The following code demonstrates how to access a base class method:

Snippet

```

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();
    }
}

```

© Aptech Ltd. Building Applications Using C# / Session 7 37

Calling the Base Class Method 3-3

- In the code:
 - 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

```

© Aptech Ltd. Building Applications Using C# / Session 7 38

Use slide 36 to explain that the method overriding allows the derived class to redefine the methods of the base class.

Explain that redefining the base class methods helps to access the new method but not the original base class method.

Tell that to execute both the base class method and the derived class method, create an instance of the base class to access the base class method and an instance of the derived class, to access the derived class method.

Use slide 37 to demonstrate how to access a base class method. Use slide 38 to explain the code and the output. Tell that in the code, the class **Student** consists of a virtual method called **PrintDetails()**.

Tell that 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**.

Also, mention that 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()**.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is method overriding?

Answer:

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.

Slide 39

Understand sealed classes.

The slide has a decorative header with a blue and green pixelated pattern. The title 'Sealed Classes 1-3' is centered in a white box. Below the title, there is a bulleted list of points about sealed classes. A 'Syntax' section shows the code for declaring a sealed class, followed by a note about where the keyword is used.

Sealed Classes 1-3

- ◆ A sealed class is a class that prevents inheritance. The features of a sealed class are as follows:
 - A sealed class can be declared by preceding the class keyword with the sealed keyword.
 - The sealed keyword prevents a class from being inherited by any other class.
 - The sealed class cannot be a base class as it cannot be inherited by any other class. If a class tries to derive a sealed class, the C# compiler generates an error.
- ◆ The following syntax is used to declare a sealed class:

Syntax

```
sealed class<ClassName>
{
    //body of the class
}
```

where,

- * sealed: Is a keyword used to prevent a class from being inherited.
- * ClassName: Is the name of the class that needs to be sealed.

In slide 39, explain that a sealed class is a class that prevents inheritance.

Tell that a sealed class can be declared by preceding the `class` keyword with the `sealed` keyword.

Then, tell that 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.

Explain that the syntax is used to declare a sealed class as given on the slide.

Slides 40 and 41

Understand the use of a sealed class.

Sealed Classes 2-3

- The following code demonstrates the use of a sealed class in C# which will generate a compiler error:

Snippet

```
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 the code:
 - 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**.

Sealed Classes 3-3

- The class **Pen** tries to inherit the sealed class **Product**, the C# compiler generates an error, as shown in the following figure:

Error List			
Description		File	Line
File	Line	Column	Project
Goods.cs	26	11	ConsoleApplication1
1 'ConsoleApplication1.Pen': cannot derive from sealed type 'ConsoleApplication1.Product'			

Use slide 40 to explain the code that demonstrates the use of a sealed class in C# which will generate a compiler error. Use slide 41 to explain the code and the output. Tell that 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**.

Also, tell that when the class **Pen** tries to inherit the sealed class **Product**, the C# compiler generates an error. You can refer to the figure in slide 41.

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.

Slide 42

Understand purpose of sealed classes.

Purpose of Sealed Classes

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

© Aptech Ltd. Building Applications Using C# / Session 7 42

Use slide 42 to give an example.

Ask the students to consider a class named **SystemInformation** that consists of critical methods that affect the working of the operating system.

Tell that you might not want any third party to inherit the class **SystemInformation** and override its methods, thus, causing security and copyright issues.

Mention that you can declare the **SystemInformation** class as sealed to prevent any change in its variables and methods.

Slide 43

Understand guidelines to use sealed classes.

The slide has a teal header bar with the title 'Guidelines' on the right. The main content area contains a bulleted list under a diamond icon. At the bottom, there's a footer bar with the Aptech logo, the slide title, and the page number 43.

◆ Sealed classes are restricted classes that cannot be inherited where the list depicts the conditions in which a class can be marked as sealed:

- ◆ If overriding the methods of a class might result in unexpected functioning of the class.
- ◆ When you want to prevent any third party from modifying your class.

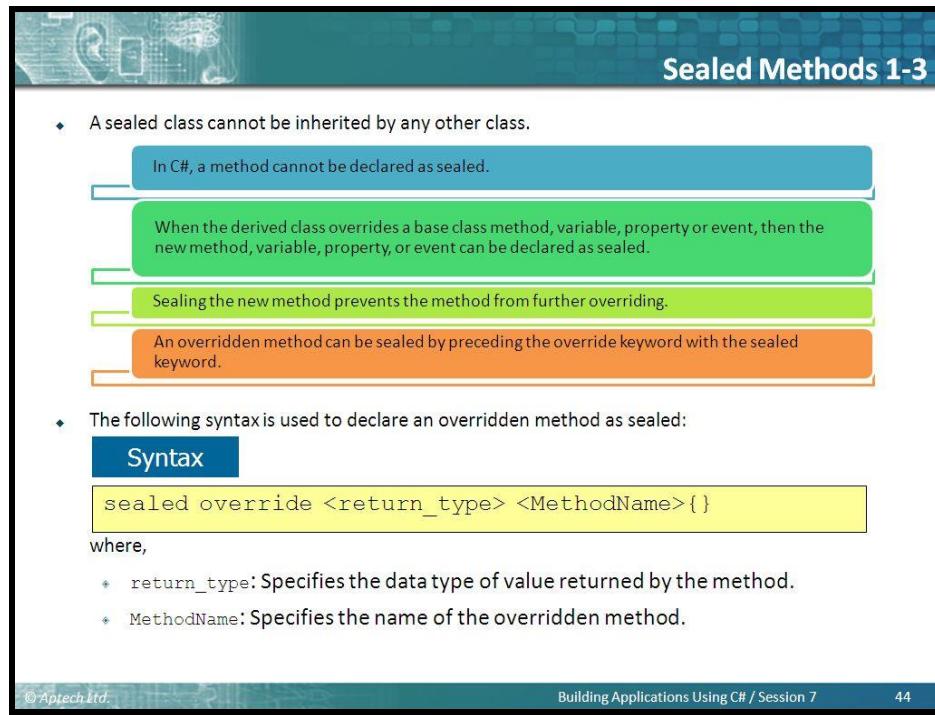
Use slide 43 to explain that sealed classes are restricted classes that cannot be inherited. Tell that the list depicts the conditions in which a class can be marked as sealed.

Mention the conditions.

- If overriding the methods of a class might result in unexpected functioning of the class
- When you want to prevent any third party from modifying your class

Slides 44 to 46

Understand sealed methods.



Sealed Methods 1-3

- ◆ A sealed class cannot be inherited by any other class.
 - In C#, a method cannot be declared as sealed.
 - When the derived class overrides a base class method, variable, property or event, then the new method, variable, property, or event can be declared as sealed.
 - Sealing the new method prevents the method from further overriding.
 - An overridden method can be sealed by preceding the override keyword with the sealed keyword.
- ◆ The following syntax is used to declare an overridden method as sealed:

Syntax

```
sealed override <return_type> <MethodName>{ }
```

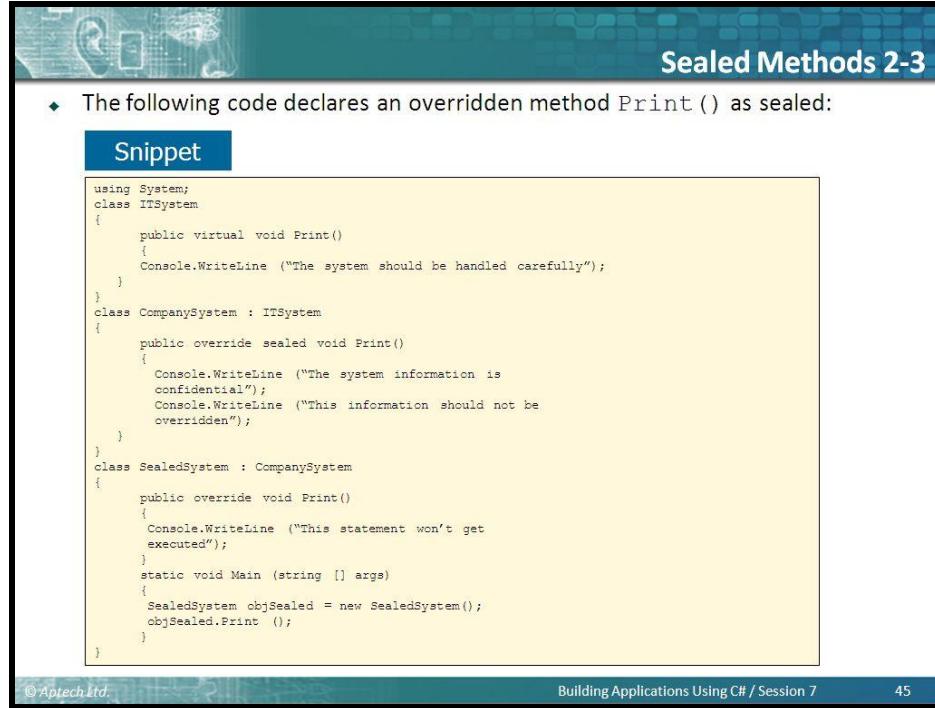
where,

 - * **return_type:** Specifies the data type of value returned by the method.
 - * **MethodName:** Specifies the name of the overridden method.

© Aptech Ltd.

Building Applications Using C# / Session 7

44



Sealed Methods 2-3

- ◆ The following code declares an overridden method `Print()` as sealed:

Snippet

```
using System;
class ITSystem
{
    public virtual void Print()
    {
        Console.WriteLine ("The system should be handled carefully");
    }
}
class CompanySystem : ITSystem
{
    public override sealed void Print()
    {
        Console.WriteLine ("The system information is
confidential");
        Console.WriteLine ("This information should not be
overridden");
    }
}
class SealedSystem : CompanySystem
{
    public override void Print()
    {
        Console.WriteLine ("This statement won't get
executed");
    }
    static void Main (string [] args)
    {
        SealedSystem objSealed = new SealedSystem();
        objSealed.Print ();
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 7

45

Sealed Methods 3-3

- ◆ In the code:
 - ❖ 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 the following figure:

Error List

Description	File	Line	Column	Project
1 'ConsoleApplication1.SealedSystem.Print()': cannot override inherited member 'ConsoleApplication1.CompanySystem.Print()' because it is sealed	SealedSystem.cs	26	22	ConsoleApplication1

In slide 44, explain that a sealed class cannot be inherited by any other class. In C#, a method cannot be declared as sealed.

Tell that 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.

In slide 45, explain that the code declares an overridden method **Print()** as sealed.

Use slide 46 to explain the code and the output. Tell that in the code, the class **ITSystem** consists of a virtual function **Print()**.

Then, tell that 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.

Mention that the class **SealedSystem** is inherited from the class **CompanySystem**. When the class **SealedSystem** overrides the sealed method **Print()**, the C# compiler generates an error.

You can refer to the figure in slide 46.

Slide 47

Understand polymorphism.

The slide has a blue header with the word 'Polymorphism'. The main content area contains the following text:

- ◆ Polymorphism is the ability of an entity to behave differently in different situations.
- ◆ Polymorphism is derived from two Greek words, namely **Poly** and **Morphos**, meaning forms. Polymorphism means existing in multiple forms.
- ◆ The following methods in a class have the same name but different signatures that perform the same basic operation but in different ways:
 - ◆ Area (float radius)
 - ◆ Area (float base, float height)
- ◆ These methods calculate the area of the circle and triangle taking different parameters and using different formulae.

Example

- ◆ Polymorphism allows methods to function differently based on the parameters and their data types. The following figure displays the polymorphism:

```
Shape
SetDimensions(int length,
               int breadth)
SetDimensions(int length,
               int breadth,
               int height)
```

} Overloaded Methods –
Way of Implementing
Polymorphism

© Aptech Ltd. Building Applications Using C# / Session 7 47

Use slide 47 to explain that polymorphism is derived from two Greek words, namely **Poly** and **Morphos**.

Explain that 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.

Give an example. Ask the students to consider the following two methods in a class having the same name but different signatures performing the same basic operation but in different ways.

Tell the two ways,

- **Area (float radius)**
- **Area (float base, float height)**

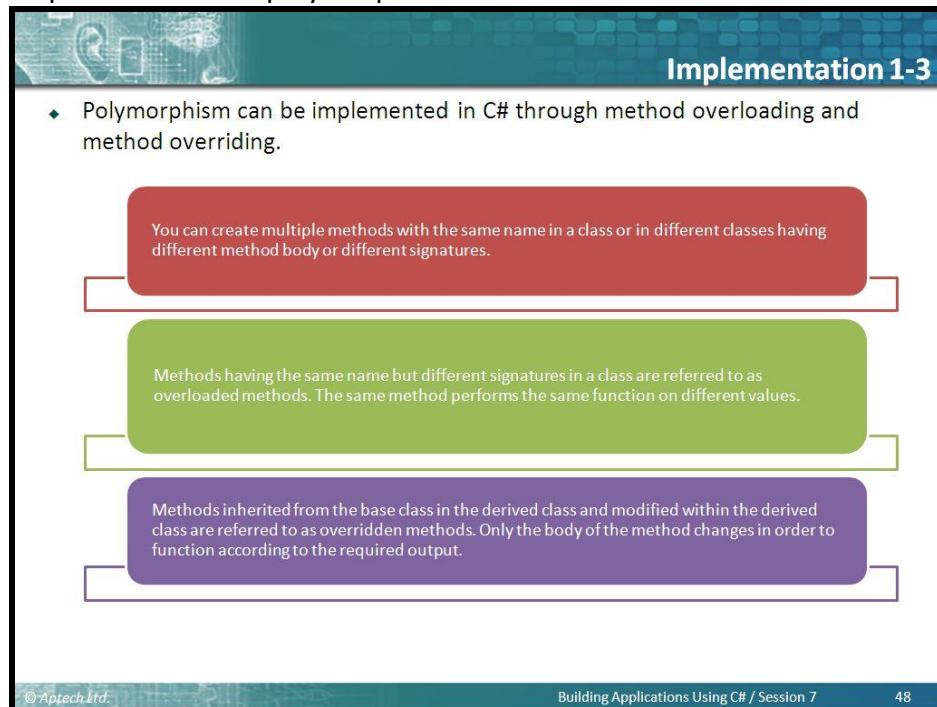
Explain that the two methods calculate the area of the circle and triangle taking different parameters and using different formulae.

Mention that polymorphism allows methods to function differently based on the parameters and their data types.

You can refer to the figure in slide 47 that displays the polymorphism.

Slides 48 to 50

Understand implementation of polymorphism.



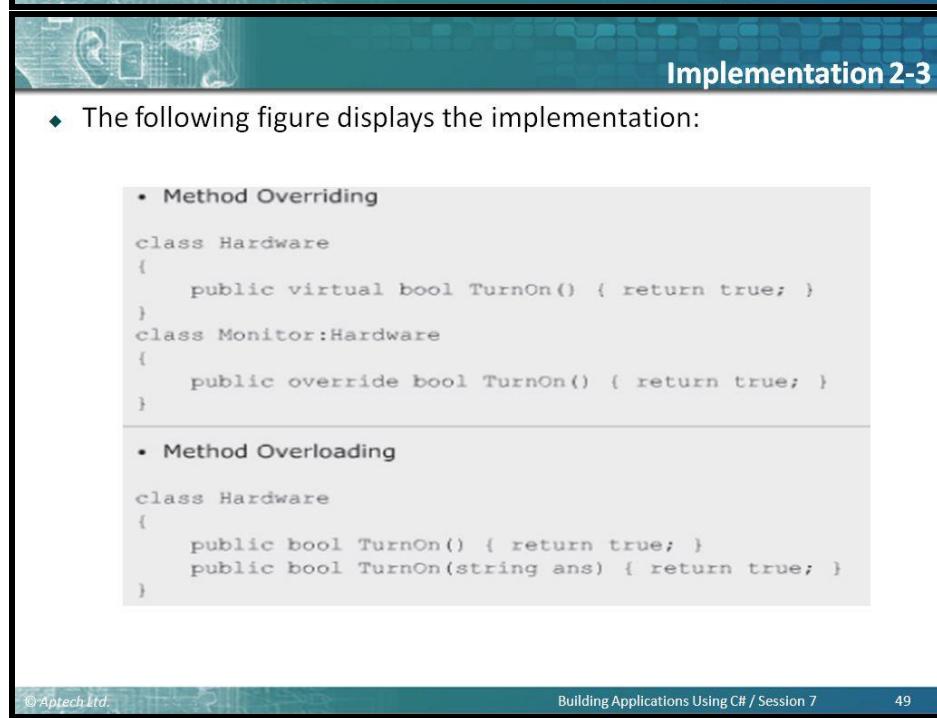
Implementation 1-3

- ◆ Polymorphism can be implemented in C# through method overloading and method overriding.

You can create multiple methods with the same name in a class or in different classes having different method body or different signatures.

Methods having the same name but different signatures in a class are referred to as overloaded methods. 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. Only the body of the method changes in order to function according to the required output.



Implementation 2-3

- ◆ The following figure displays the implementation:

- Method Overriding

```
class Hardware
{
    public virtual bool TurnOn() { return true; }
}
class Monitor:Hardware
{
    public override bool TurnOn() { return true; }
}
```

- Method Overloading

```
class Hardware
{
    public bool TurnOn() { return true; }
    public bool TurnOn(string ans) { return true; }
}
```

Implementation 3-3

- The following code demonstrates the use of method overloading feature:

Snippet

```
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 the code:
 - 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
```

© Aptech Ltd. Building Applications Using C# / Session 7 50

In slide 48, explain that polymorphism can be implemented in C# through method overloading and method overriding. Tell that multiple methods can be created with the same name in a class or in different classes having different method body or different signatures. Then, mention that 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. Also, tell that 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.

Use slide 49 to refer to figure that displays the implementation.

Use slide 50 to explain that the code demonstrates the use of method overloading feature.

Explain the code and the output. Tell that 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.

Additional Information

Refer the following links for more information on polymorphism:

<http://msdn.microsoft.com/en-us/library/ms173152.aspx>

<http://www.codeproject.com/Articles/602141/Polymorphism-in-NET>

Slides 51 to 53

Understand compile-time and run-time polymorphism.

Compile-time and Run-time Polymorphism 1-3

- ◆ Polymorphism can be broadly classified into the following categories:
 - ❖ Compile-time polymorphism
 - ❖ Run-time polymorphism
- ◆ The following table differentiates between compile-time and run-time polymorphism:

Compile-time Polymorphism	Run-time Polymorphism
Is implemented through method overloading .	Is implemented through method overriding .
Is executed at the compile-time since the compiler knows which method to execute depending on the number of parameters and their data types.	Is executed at run-time since the compiler does not know the method to be executed, whether it is the base class method that will be called or the derived class method.
Is referred to as static polymorphism.	Is referred to as dynamic polymorphism.

Compile-time and Run-time Polymorphism 2-3

- ◆ The following code demonstrates the implementation of run-time polymorphism:

Snippet

```
using System;
class Circle
{
    protected const double PI = 3.14;
    protected double Radius = 14.9;
    public virtual double Area()
    {
        return PI * Radius * Radius;
    }
}
class Cone : Circle
{
    protected double Side = 10.2;
    public override double Area()
    {
        return PI * Radius * Side;
    }
    static void Main(string[] args)
    {
        Circle objRunOne = new Circle();
        Console.WriteLine("Area is: " + objRunOne.Area());
        Circle objRunTwo = new Cone();
        Console.WriteLine("Area is: " + objRunTwo.Area());
    }
}
```

Compile-time and Run-time Polymorphism 3-3

- ◆ In the code:
 - ◆ 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
```

©Aptech Ltd. Building Applications Using C# / Session 7 53

In slide 51, explain that polymorphism can be broadly classified into compile-time polymorphism and run-time polymorphism.

You can refer to table from slide 51 that differentiates between compile-time and run-time polymorphism.

In slide 52, tell the students that the code demonstrates the implementation of run-time polymorphism.

Use slide 53 to explain the code. Tell that in the code, 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 ()**.

Tell that 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.

Also, mention that in this case, the **Area ()** method of **Cone** will be called even though the reference created was that of **Circle**.

Slide 54

Summarize the session.



Summary

- ◆ Inheritance allows you to create a new class from another class, thereby inheriting its common properties and methods.
- ◆ Inheritance can be implemented by writing the derived class name followed by a colon and the name of the base class.
- ◆ Method overriding is a process of redefining the base class methods in the derived class.
- ◆ Methods can be overridden by using a combination of virtual and override keywords within the base and derived classes respectively.
- ◆ Sealed classes are classes that cannot be inherited by other classes.
- ◆ You can declare a sealed class in C# by using the `sealed` keyword.
- ◆ Polymorphism is the ability of an entity to exist in two forms that are compile-time polymorphism and run-time polymorphism.

© Aptech Ltd. Building Applications Using C# / Session 7 54

In slide 54, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- Inheritance allows you to create a new class from another class, thereby inheriting its common properties and methods.
- Inheritance can be implemented by writing the derived class name followed by a colon and the name of the base class.
- Method overriding is a process of redefining the base class methods in the derived class.
- Methods can be overridden by using a combination of virtual and override keywords within the base and derived classes respectively.
- Sealed classes are classes that cannot be inherited by other classes.
- You can declare a sealed class in C# by using the `sealed` keyword.
- Polymorphism is the ability of an entity to exist in two forms that are compile-time polymorphism and run-time polymorphism.

7.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. What are abstract methods?
2. What is an object reference?
3. What are dynamic methods?

Session 8 - Abstract Classes and Interfaces

8.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the previous session for a review. Prepare the background knowledge/summary to be discussed with students in the class. The summary of the previous session is as follows:

- Inheritance allows you to create a new class from another class, thereby inheriting its common properties and methods.
- Inheritance can be implemented by writing the derived class name followed by a colon and the name of the base class.
- Method overriding is a process of redefining the base class methods in the derived class.
- Methods can be overridden by using a combination of virtual and override keywords within the base and derived classes respectively.
- Sealed classes are classes that cannot be inherited by other classes.
- You can declare a sealed class in C# by using the sealed keyword.
- Polymorphism is the ability of an entity to exist in two forms that are compile-time polymorphism and run-time polymorphism.

Familiarize yourself with the topics of this session in-depth.

8.1.1 Objectives

By the end of this session, the learners will be able to:

- Define and describe abstract classes
- Explain Interfaces
- Compare abstract classes and interfaces

8.1.2 Teaching Skills

To teach this session successfully, you must know the terms abstract classes and interfaces. You should be aware of various aspects regarding abstract classes and interfaces.

You should teach the concepts in the theory class using the concepts, tables, and codes provided.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

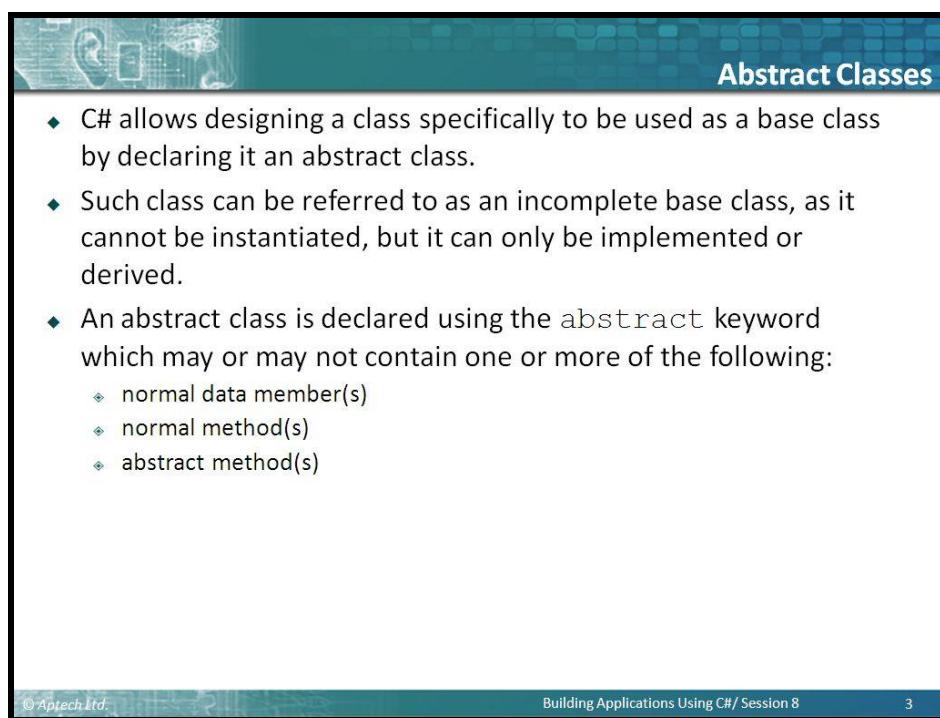
Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced the concept of abstract classes.

This session will also discuss comparison of interfaces and abstract classes.

8.2 In-Class Explanations

Slide 3

Understand the concept of abstract classes.



The slide has a teal header bar with the title "Abstract Classes". The main content area contains a bulleted list explaining abstract classes in C#. The footer bar includes the copyright notice "© Aptech Ltd.", the slide title "Abstract Classes", and the page number "3".

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 declared using the `abstract` keyword which may or may not contain one or more of the following:
 - ◆ normal data member(s)
 - ◆ normal method(s)
 - ◆ abstract method(s)

© Aptech Ltd. Abstract Classes 3

Use slide 3 to explain abstract classes to the students.

Tell the students that C# allows designing a class specifically to be used as a base class by declaring it an abstract class. Tell them that such class can be referred to as an incomplete base class, as it cannot be instantiated, but it can only be implemented or derived.

Explain to the students that an abstract class is a class declared using the `abstract` keyword and has the following characteristics: it may or may not contain normal data member(s), normal method(s), and abstract method(s).

Slides 4 and 5

Understand the purpose of abstract classes.

The slide has a teal header bar with the title "Purpose 1-2". The main content area contains a bulleted list of six points explaining why abstract classes are useful for modeling animals like dogs, cats, lions, and tigers. At the bottom, there is a footer bar with the copyright information "© Aptech Ltd." and the slide number "4".

Purpose 1-2

- ◆ 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, habitat nor do they make similar sounds.
- ◆ Hence, the **Eat()**, **Habitat()**, and **AnimalSound()** methods need to be different for different animals even though they inherit the same base class.
- ◆ These differences can be incorporated using abstract classes.

© Aptech Ltd. Building Applications Using C#/ Session 8 4

Purpose 2-2

- The following figure displays an example of abstract class and subclasses:

```

class Animal {
    Eat()
    {
        "Every animal eats grass"
    }

    AnimalSound()
    {
        "Hiss"
    }
}

class Dog : Animal {
    Eat()
    {
        "Every animal eats grass"
    }

    AnimalSound()
    {
        "Barks"
    }
}

class Cat : Animal {
    Eat()
    {
        "Every animal eats grass"
    }

    AnimalSound()
    {
        "Hiss"
    }
}

class Tiger : Animal {
    Eat()
    {
        "Every animal eats grass"
    }

    AnimalSound()
    {
        "Hiss"
    }
}

```

The diagram shows the **Animal** class at the top, which defines two methods: **Eat()** and **AnimalSound()**. Three arrows point down to three subclasses: **Dog**, **Cat**, and **Tiger**. Each subclass has its own implementation of the **Eat()** and **AnimalSound()** methods, with specific values like "Barks" or "Hiss". Red X marks are placed over the original method definitions in the **Animal** class, indicating they are overridden by the subclasses.

© Aptech Ltd. Building Applications Using C#/ Session 8 5

Use slide 4 to understand the purpose of explaining abstract classes. Then, use slide 5 to explain the students about purpose of abstract classes.

Tell the students to consider the base class **Animal**, that defines methods such as **Eat()**, **Habitat()**, and **AnimalSound()**. Tell them that the **Animal** class is inherited by different subclasses such as **Dog**, **Cat**, **Lion**, and **Tiger**.

Mention that the dogs, cats, lions, and tigers neither share the same food nor habitat and nor do they make similar sounds. Hence, the **Eat()**, **Habitat()**, and **AnimalSound()** methods need to be different for different animals even though they inherit the same base class.

Tell the students that these differences can be incorporated using abstract classes.

Tell the students that when we declare an abstract class it must be declared with access modifier 'abstract' and each function/method in that class must be declared with 'abstract' keyword.

When abstract class is derived, the derived class must use keyword 'override' to redefine the methods/functions that are declared as 'abstract' in the abstract class.

Additional Information

Refer the following links for more information on abstract classes:

<http://msdn.microsoft.com/en-us/library/ms173150.aspx>

<http://www.codeproject.com/Articles/6118/All-about-abstract-classes>

Slide 6

Understand the definition of abstract class.

Definition 1-4

- An abstract class can implement methods that are similar for all the subclasses.

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 will not be compiled.

Since the abstract class contains at least one method without a body, the class cannot be instantiated using the `new` keyword.

© Aptech Ltd. Building Applications Using C# / Session 8 6

Using slide 6, tell the students that an abstract class can implement methods that are similar for all the subclasses.

Tell them that it can declare methods that are different for different subclasses without implementing them and these methods are referred to as abstract methods.

Explain to the students that a class that is defined using the `abstract` keyword and that contains at least one method is not implemented in the class. Such a class is referred to as an abstract class.

Mention that in the absence of the `abstract` keyword, the class cannot be compiled and since the abstract class contains at least one method without a body, the class cannot be instantiated using the `new` keyword.

Additional Information

Refer the following links for more information on abstract classes:

<http://msdn.microsoft.com/en-us/library/ms173150.aspx>

<http://www.codeproject.com/Articles/6118/All-about-abstract-classes>

Slide 7

Understand the contents of abstract class.

Definition 2-4

- The following figure displays the contents of an abstract class:

Abstract Class

```
Method_1 ()  
{  
    // Body  
}  
  
AbstractMethod_1 (); ← Abstract Methods  
AbstractMethod_2 (); ←
```

© Aptech Ltd. Building Applications Using C#/ Session 8 7

In slide 7, explain that the methods which are defined inside abstract class are abstract methods only. In the example shown on the slide, AbstractMethod_1() and AbstractMethod_2() are abstract methods.

Give some examples of abstract classes in real time. Tell them that we cannot create a direct instance of an abstract class because in real world also we do not find any instance of the abstract classes.

We use it just for our understanding or grouping the classes having similar functionalities into one class that is known as an abstract class.

Give them examples of abstract classes such as Tree, Vehicle, and Animal. Explain to them how these classes are abstract classes.

Slide 8

Understand the syntax used for declaring an abstract class.

The slide has a blue header bar with the title "Definition 3-4". Below the header, there is a bulleted list: "◆ The following syntax is used for declaring an abstract class:". Underneath this, there is a yellow box containing the C# syntax for an abstract class:

```
public abstract class <ClassName>
{
    <access_modifier> abstract <return_type>
        <MethodName>(<argument_list>);
}
```

Below the code, the word "where," is followed by two bullet points:

- ◆ **abstract**: Specifies that the declared class is abstract.
- ◆ **ClassName**: Specifies the name of the class.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C#/ Session 8", and the number "8".

In slide 8, display the syntax used for declaring an abstract class and tell the students that in the syntax, **abstract** specifies that the declared class is abstract and the **ClassName** specifies the name of the class.

Slide 9

Understand the code for abstract class.

The slide is titled "Definition 4-4". It contains a bullet point: "The following code declares an abstract class **Animal**:". Below this is a "Snippet" box containing the following C# code:

```
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();
}
```

Below the code, another bullet point says: "In the code:" followed by two sub-points:

- ◆ 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()**.

At the bottom left is the copyright notice "© Aptech Ltd.", at the bottom center is "Building Applications Using C#/ Session 8", and at the bottom right is the number "9".

Tell the students that in the code shown in slide 9, the abstract class **Animal** is created using the **abstract** keyword. Tell them that in the code, the **Animal** class implements the non-abstract method, **Eat ()**, as well as declares two abstract methods, **AnimalSound()** and **Habitat()**.

It is not mandatory for the abstract class to contain only abstract methods. An abstract class cannot be sealed.

Slides 10 to 13

Understand the implementation process of the abstract class.

Implementation 1-4

- The subclass inheriting the abstract class has to override and implement the abstract methods with the same name and arguments.
- On failing to implement, the subclass cannot be instantiated as the C# compiler considers it as abstract.
- The following figure displays an example of inheriting an abstract class:

```

class Diagram {
    class AbstractClass {
        Method_1 () {
            // Body
        }
        AbstractMethod_1 () {
            // Body
        }
        AbstractMethod_2 () {
            // Body
        }
    }
    class InheritingClass : AbstractClass {
        Method_1 () {
            // Body
        }
        AbstractMethod_1 () {
            // Body
        }
        AbstractMethod_2 () {
            // Body
        }
    }
}
  
```

© Aptech Ltd. Building Applications Using C#/ Session 8 10

Implementation 2-4

- The following syntax is used to implement an abstract class:

Syntax

```

class <ClassName> : <AbstractClassName>
{
    // class members;
}
  
```

where,

AbstractClassName: Specifies the name of the inherited abstract class.

© Aptech Ltd. Building Applications Using C#/ Session 8 11

Implementation 3-4

- The following code declares and implements an abstract class:

Snippet

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

    public abstract void AnimalSound();
}

class Lion : Animal
{
    public override void AnimalSound()
    {
        Console.WriteLine("Lion roars");
    }

    static void Main(string[] args)
    {
        Lion objLion = new Lion();
        objLion.AnimalSound();
        objLion.Eat();
    }
}
```

© Aptech Ltd. Building Applications Using C#/ Session 8 12

Implementation 4-4

Output

```
Lion roars
Every animal eats food in order to survive
```

- In the code:
 - 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.

© Aptech Ltd. Building Applications Using C#/ Session 8 13

In slide 10, introduce the implementation process of the abstract class to the students.

Tell the students that an abstract class can be implemented in a way similar to implementing a normal base class. Tell them that the subclass inheriting the abstract class has to override and implement the abstract methods. In addition, explain that the subclass can implement the methods implemented in the abstract class with the same name and arguments.

Mention that if the subclass fails to implement the abstract methods, the subclass cannot be instantiated as the C# compiler considers it as abstract.

Then, refer to the figure in slide 10 to display an example of inheriting an abstract class to the students. Explain the figure to the students.

In slide 11, explain the syntax used to implement an abstract class to the students and tell them that `AbstractClassName` specifies the name of the inherited abstract class.

In slide 12, explain the code that declares and implements an abstract class to the students.

In slide 13, explain the output of the code to the students. Then, explain the working of the code.

Tell them that in the code, the abstract class `Animal` is declared, and the class `Lion` inherits the abstract class `Animal`. Tell them that since the `Animal` class declares an abstract method called `AnimalSound()`, the `Lion` class overrides the method `AnimalSound()` using the `override` keyword and implements it. Mention that the `Main()` method of the `Lion` class then invokes the methods `AnimalSound()` and `Eat()` using the dot (.) operator.

In-Class Question:

After you finish explaining the implementation process of the abstract class, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Why Abstract class is referenced as an incomplete base class?

Answer:

Abstract class is referred as an incomplete base class because it cannot be instantiated, but it can only be implemented or derived.

Slides 14 and 15

Understand the implementation of abstract base class using IntelliSense.

Implement Abstract Base Class Using IntelliSense 1-2

- IntelliSense provides access to member variables, functions, and methods of an object or a class. Thus, it helps the programmer to easily develop the software by reducing the amount of input typed in, since IntelliSense performs the required typing. IntelliSense can be used to implement system-defined abstract classes.
- The steps performed to implement an abstract class using IntelliSense are as follows:

1. Place Cursor	• Place the cursor after the class IntelliSenseDemo statement.
2. Type the following : TimeZone	• The class declaration becomes class IntelliSenseDemo : TimeZone .
3. Click Smart Tag	• 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.

Implement Abstract Base Class Using IntelliSense 2-2

- The following code demonstrates the way the methods of the abstract class **TimeZone** are invoked automatically by IntelliSense:

Snippet

```
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."); }
    }
}
```

In slide 14, explain how abstract base classes can be implemented using IntelliSense.

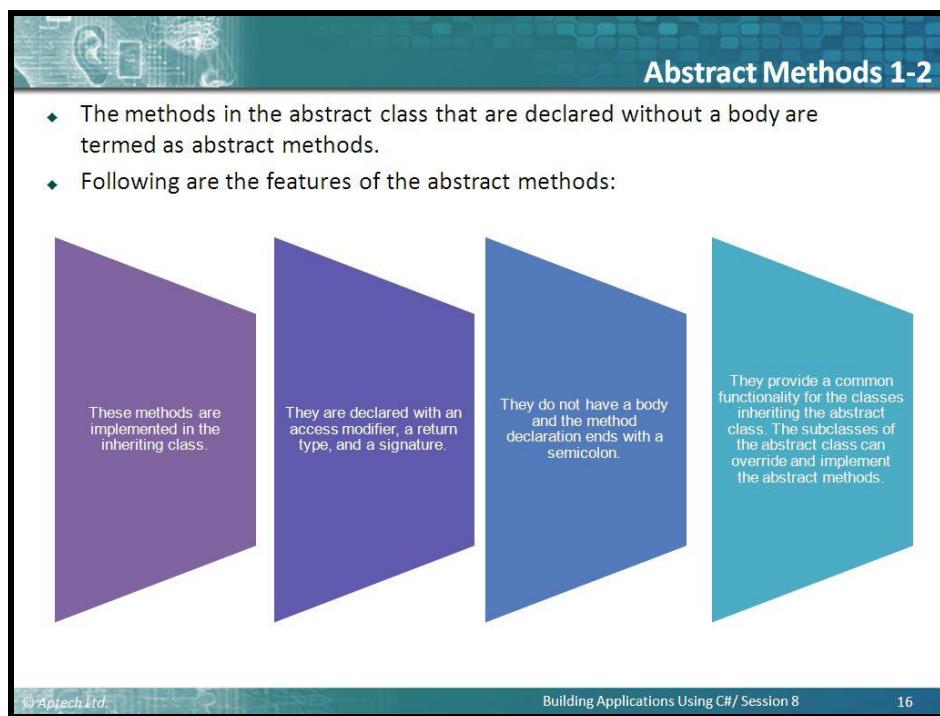
Tell the students that IntelliSense provides access to member variables, functions, and methods of an object or a class. Tell them that it also helps the programmer to easily develop the software by reducing the amount of input typed in, since IntelliSense performs the required typing. Mention that the IntelliSense can be used to implement system-defined abstract classes.

Then, explain the steps that are performed to implement an abstract class using IntelliSense to the students. Explain the code that demonstrates the way the methods of the abstract class `TimeZone` are invoked automatically by IntelliSense.

In slide 15, explain that with the help of IntelliSense features provided, the class `IntelliSenseDemo` is derived from `TimeZone` which will automatically invoke all the operations throughout the code.

Slides 16 and 17

Understand abstract methods.



The slide is titled "Abstract Methods 1-2". It contains a bulleted list of two items:

- The methods in the abstract class that are declared without a body are termed as abstract methods.
- Following are the features of the abstract methods:

Four colored callout boxes provide details:

- Purple box: These methods are implemented in the inheriting class.
- Dark purple box: They are declared with an access modifier, a return type, and a signature.
- Blue box: They do not have a body and the method declaration ends with a semicolon.
- Cyan box: They provide a common functionality for the classes inheriting the abstract class. The subclasses of the abstract class can override and implement the abstract methods.

At the bottom left is the copyright notice "© Aptech Ltd." and at the bottom right are the page numbers "Building Applications Using C# / Session 8" and "16".

Abstract Methods 2-2

- The following figure displays an example of abstract methods:

Abstract Class

```

Method_1 () ← Implemented Method

{
    // Body
}

AbstractMethod_1 (); ← Abstract Methods
AbstractMethod_2 (); ← (Not Implemented,
                        Having No Body)

```

© Aptech Ltd.

Building Applications Using C#/ Session 8 17

In slide 16, explain abstract methods to the students.

Tell the students that the methods in the abstract class are declared without a body are termed as abstract methods. Tell them that these methods are implemented in the inheriting class.

Explain to the students that an abstract method is declared with an access modifier, a return type, and a signature similar to a regular method. Tell them that this method does not have a body and the method declaration ends with a semicolon.

Then, mention to the students that the 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.

Refer slide 17 to display an example of abstract methods. Explain this figure to the students.

Explain to the students that the idea of having an abstract class or abstract method is that, by having it, we can generalize the classes. By declaring a class as abstract, we leave it to the user of the class to decide the implementation of such methods. The user can provide the actual definition to these abstract methods. Also tell the students that all the abstract methods must be public as they will be accessed and overridden in the derived classes. If you declare the abstract method as private, the compiler will throw an error.

Slide 18

Understand multiple inheritance through interfaces.

The slide has a decorative header with a blue and green gradient and a subtle circuit board pattern. The title 'Multiple Inheritance through Interfaces' is centered in a white box. Below the title is a bulleted list:

- ◆ A subclass in C# cannot inherit two or more base classes because C# does not support multiple inheritance.
- ◆ To overcome this drawback, interfaces were introduced.
- ◆ A class in C# can implement multiple interfaces.

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 8' in the center, and the number '18' on the right.

In slide 18, explain to the students multiple inheritance through interfaces.

Tell the students that a subclass in C# cannot inherit two or more base classes as C# does not support multiple inheritance. Tell them that to overcome this drawback, interfaces were introduced and a class in C# can implement multiple interfaces.

Tell the students that C# does not support multiple inheritance. It only supports multi-level inheritance. Sometimes, we may have a situation where we need multiple inheritance. For this we can make use of interfaces.

Additional Information

Refer the following links for more information on interfaces:

<http://msdn.microsoft.com/en-us/library/ms173156.aspx>
<http://www.codeproject.com/Articles/18743/Interfaces-in-C-For-Beginners>

Slide 19

Understand the purpose of the multiple inheritance through interfaces.

Purpose of Interfaces

- ◆ Consider a class **Dog** that needs to inherit features of **Canine** and **Animal** classes.
- ◆ The **Dog** class cannot inherit methods of both these classes as C# does not support multiple inheritance.
- ◆ However, if **Canine** and **Animal** are declared as interfaces, the class **Dog** can implement methods from both the interfaces.
- ◆ The following figure displays an example of subclasses with interfaces in C#:

The diagram illustrates two inheritance scenarios. On the left, under 'Cannot Implement Multiple Classes', a 'Dog' class is shown derived from both a 'Canine Class' (with a mouth icon) and an 'Animal Class' (with paw prints). A red diagonal line through the 'Dog' class indicates that it cannot inherit from both. On the right, under 'Can Implement Multiple Interfaces', the same 'Dog' class is shown derived from a 'Canine Interface' (with a mouth icon) and an 'Animal Interface' (with paw prints). This configuration is labeled as a 'Derived Class'. The title 'Purpose of Interfaces' at the top right suggests that using interfaces allows the 'Dog' class to implement methods from both interfaces.

© Aptech Ltd. Building Applications Using C# / Session 8 19

In slide 19, explain to the students that the figure displays an example of multiple inheritance using interfaces.

Tell the students to consider a class **Dog** that needs to inherit features of **Canine** and **Animal** classes. Tell them that the **Dog** class cannot inherit methods of both these classes as C# does not support multiple inheritance. Explain to the students that if **Canine** and **Animal** are declared as interfaces, the class **Dog** can implement methods from both the interfaces.

Additional Information

Refer the following links for more information on interfaces:

<http://msdn.microsoft.com/en-us/library/ms173156.aspx>
<http://www.codeproject.com/Articles/18743/Interfaces-in-C-For-Beginners>

Slides 20 and 21

Understand interfaces.

Interfaces 1-2

- ◆ An interface contains only abstract members that cannot implement any method.
- ◆ An interface cannot be instantiated but 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.
- ◆ The following figure displays an example of an interface:

Animal Abstract Class <pre> Eat() { "Every animal eats food"; } Habitat(); AnimalSound();</pre>	IAnimalInterface <pre>Eat(); //No Body Habitat(); AnimalSound();</pre>
---	---

© Aptech Ltd. Building Applications Using C#/ Session 8 20

Interfaces 2-2

- ◆ The following syntax is used to declare an interface:

Syntax <pre>interface <InterfaceName> { //interface members }</pre>
--

where,

- ◆ `interface`: Declares an interface.
- ◆ `InterfaceName`: Is the name of the interface.

- ◆ The following code declares an interface **IAnimal**:

Snippet <pre>interface IAnimal { void AnimalType(); }</pre>
--

- ◆ In the code:

- ◆ The interface `IAnimal` is declared which contains an abstract method `AnimalType()`.

© Aptech Ltd. Building Applications Using C#/ Session 8 21

In slide 20, explain interfaces to the students. Explain to the students that an interface contains only abstract members. Tell them that unlike an abstract class, an interface cannot implement any method. Tell them that similar to an abstract class, an interface cannot be instantiated.

Mention that an interface can only be inherited by classes or other interfaces.

Explain to the students that an interface is declared using the keyword `interface`. In C#, by default, all members declared in an interface have `public` as the access modifier.

Then, refer to the figure in slide 20 that displays an example of an interface to the students.

In slide 21, explain the syntax that is used to declare an interface and tell the students that `interface` declares an interface and `InterfaceName` is the name of the interface.

Then, explain a code that declares an interface `IAnimal` and tell them that the interface `IAnimal` is declared and the interface declares an abstract method `AnimalType()`.

Interfaces cannot contain constants, data fields, constructors, destructors, and static members.

Now, you can compare the abstract class and interface and tell the students the main difference between the abstract class and the interface. An abstract class can have fields, constants, abstract as well as non-abstract functions.

Additional Information

Refer the following links for more information on interfaces:

<http://msdn.microsoft.com/en-us/library/ms173156.aspx>
<http://www.codeproject.com/Articles/18743/Interfaces-in-C-For-Beginners>

Slide 22

Understand the implementation of an interface.

Implementing an Interface 1-4

- ◆ An interface is implemented by a class in a way similar to inheriting a class.
- ◆ When implementing an interface in a class, 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.
- ◆ The following figure displays the implementation of an interface:

```

classDiagram
    IAmableInterface {
        AnimalSound();
    }
    Dog {
        AnimalSound() {
            "The dog barks."
        }
    }
    Lion {
        AnimalSound() {
            "The Lion Roars."
        }
    }
    Tiger {
        AnimalSound() {
            "The tiger makes the sound \"grrrr\"."
        }
    }
    IAmableInterface <|-- Dog
    IAmableInterface <|-- Lion
    IAmableInterface <|-- Tiger
  
```

IAmableInterface

AnimalSound();

Dog

Lion

Tiger

© Aptech Ltd.

Building Applications Using C# / Session 8 22

In slide 22, explain to the students that an interface is implemented by a class in a way similar to inheriting a class. Tell them that when implementing an interface in a class, they need to implement all the abstract methods declared in the interface. Also, tell them that if all the methods are not implemented, the class cannot be compiled. Mention that the methods implemented in the class should be declared with the same name and signature as defined in the interface.

You can refer to the figure in slide 22 to display the implementation of an interface.

Slide 23

Understand the implementation of an interface.

The slide has a blue header bar with the title "Implementing an Interface 2-4". Below the header, there is a bulleted list: "The following syntax is used to implement an interface:". A yellow callout box labeled "Syntax" contains the following code:

```
class <ClassName> : <InterfaceName>
{
    //Implement the interface methods.
    //Define class members.
}
```

Below the code, the text "where," is followed by another bullet point: "InterfaceName: Specifies the name of the interface."

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 8", and the number "23".

In slide 23, explain the syntax that is used to implement an interface and tell the students that InterfaceName specifies the name of the interface in the syntax.

Slide 24

Understand the implementation of an interface.

The slide has a teal header bar with the title "Implementing an Interface 3-4". Below the header, there is a list item followed by two sections: "Snippet" and "Output".

◆ The following code declares an interface **IAnimal** and implements it in the class **Dog**:

Snippet

```
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();
    }
}
```

Output

```
Dog
Can be housed with human beings
```

© Aptech Ltd. Building Applications Using C# / Session 8 24

In slide 24, explain the code that declares an interface **IAnimal** and implements it in the class **Dog**. Then, display the output of the code to the students.

Slide 25

Understand the implementation of an interface.

The slide has a teal header bar with the title "Implementing an Interface 4-4". The main content area contains a bulleted list:

- ◆ In the code:
 - ◆ The code 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.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 8", and the number "25".

In slide 25, explain the code. Tell the students that in the code, the code creates an interface **IAnimal** that declares the method **Habitat()**. Tell them that the class **Dog** implements the interface **IAnimal** and its method **Habitat()**. Then, mention to the students that 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.

With this slide, you will finish explaining the implementation of an interface.

Slides 26 to 28

Understand interfaces and multiple inheritance.

Interfaces and Multiple Inheritance 1-3

- Multiple interfaces can be implemented in a single class which provides the functionality of multiple inheritance.
- You can implement multiple interfaces by placing commas between the interface names while implementing them in a class.
- A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- The `override` keyword is not used while implementing abstract methods of an interface.
- The following figure displays the concept of multiple inheritance using interfaces:

```

graph TD
    I1[Interface 1<br/>//Methods Declaration] --> IC[Inheriting Class<br/>//Methods Implementation]
    I2[Interface 2<br/>//Methods Declaration] --> IC
  
```

© Aptech Ltd. Building Applications Using C#/ Session 8 26

Interfaces and Multiple Inheritance 2-3

- The following syntax is used to implement multiple interfaces:

Syntax

```
class <ClassName> : <Interface1>, <Interface2>
{
    //Implement the interface methods
}
```
- where,
 - `Interface1`: Specifies the name of the first interface.
 - `Interface2`: Specifies the name of the second interface.

© Aptech Ltd. Building Applications Using C#/ Session 8 27

Interfaces and Multiple Inheritance 3-3

- The following code declares and implements multiple interfaces:

Snippet

```

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();
    }
}

```

Output

```

The Crocodile eats flesh
The Crocodile can swim four times faster than an
Olympic swimmer

```

© Aptech Ltd. Building Applications Using C#/ Session 8 28

In slide 26, explain the interfaces and multiple Inheritance to the students. Tell them that the multiple interfaces can be implemented in a single class and this implementation provides the functionality of multiple inheritance. Tell them that the students can implement multiple interfaces by placing commas between the interface names while implementing them in a class.

Then, Explain to the students that a class implementing multiple interfaces has to implement all abstract methods declared in the interfaces. Mention that the `override` keyword is not used while implementing abstract methods of an interface.

You can refer to the figure in slide 26 to display the concept of multiple inheritance using interfaces.

In slide 27, explain the syntax that is used to implement multiple interfaces and tell them that `Interface1` specifies the name of the first interface and `Interface2` specifies the name of the second interface.

Additional Information

Refer the following links for more information on multiple inheritance with interfaces:

<http://www.codeproject.com/Articles/23628/Multiple-Inheritance-With-Interfaces>

In slide 28, explain the code that declares and implements multiple interfaces to the students and show the output.

Tell the students that in the code, the interfaces **ITerrestrialAnimal** and **IMarineAnimal** are declared. The two interfaces declare methods **Eat()** and **Swim()**.

C# allows you to inherit a base class and implement more than one interface at the same time.

Slides 29 and 30

Understand explicit interface implementation.

The slide has a blue header bar with the title 'Explicit Interface Implementation 1-5'. Below the title is a bulleted list of rules for explicit interface implementation. At the bottom left is a section titled 'Example' containing another bulleted list of scenarios. The footer contains copyright information and page numbers.

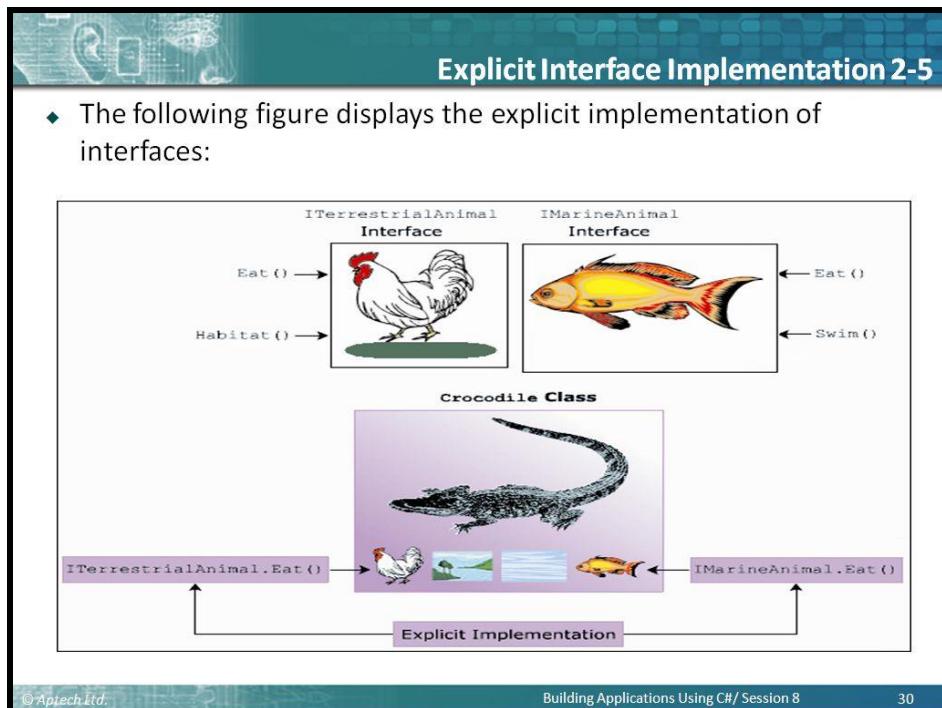
Explicit Interface Implementation 1-5

- ◆ A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- ◆ 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.

Example

- ◆ Consider the interfaces **ITerrestrialAnimal** and **IMarineAnimal**. The interface **ITerrestrialAnimal** declares methods **Eat()** and **Habitat()**.
- ◆ The interface **IMarineAnimal** declares methods **Eat()** and **Swim()**.
- ◆ The class **Crocodile** implementing the two interfaces has to explicitly implement the method **Eat()** from both interfaces by specifying the interface name before the method name.
- ◆ While explicitly implementing an interface, you cannot mention modifiers such as **abstract**, **virtual**, **override**, or **new**.

© Aptech Ltd. Building Applications Using C#/ Session 8 29



In slides 29 and 30, explain about explicit interface implementation to the students.

Tell the students that a class has to explicitly implement multiple interfaces if these interfaces have methods with identical names. Tell them that 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.

Give an example here. Tell the students to consider the interfaces **ITerrestrialAnimal** and **IMarineAnimal**. Tell them that the interface **ITerrestrialAnimal** declares methods **Eat()** and **Habitat()** and the interface **IMarineAnimal** declares methods **Eat()** and **Swim()**. Mention that 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.

Then, tell the students that they cannot mention modifiers such as abstract, virtual, override, or new while explicitly implementing an interface.

Refer to the figure in slide 30 to display the explicit implementation of interface.

Slide 31

Understand the syntax used to explicitly implement interfaces.

The slide has a blue header bar with the title "Explicit Interface Implementation 3-5". Below the title, there is a bulleted list of points. The first point is "The following syntax is used to explicitly implement interfaces:" followed by a "Syntax" box containing code. The second point is "where," followed by three descriptive bullet points. At the bottom of the slide, there is a footer bar with the copyright information "© Aptech Ltd.", the session name "Building Applications Using C# / Session 8", and the page number "31".

◆ The following syntax is used to explicitly implement interfaces:

Syntax

```
class <ClassName> : <Interface1>, <Interface2>
{
    <access modifier> Interface1.Method();
    //statements;
}
<access modifier> Interface2.Method();
//statements;
}
```

◆ where,

- ◆ **Interface1**: Specifies the first interface implemented.
- ◆ **Interface2**: Specifies the second interface implemented.
- ◆ **Method ()**: Specifies the same method name declared in the two interfaces.

In slide 31, explain the syntax that is used to explicitly implement interfaces to the students and tell them that **Interface1** specifies the first interface implemented, **Interface2** specifies the second interface implemented, and **Method ()** specifies the same method name declared in the two interfaces.

Slides 32 and 33

Understand the code that demonstrates the use of implementing interfaces explicitly.

Explicit Interface Implementation 4-5

- The following code demonstrates the use of implementing interfaces explicitly:

Snippet

```

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);
    }
}

```

Explicit Interface Implementation 5-5

Output

```

Crocodile eats other animals
Crocodile eats fish and marine animals

```

- In the code:
 - 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.

In slide 32, tell the students that here is the code that demonstrates the use of implementing interfaces explicitly.

Explain to the students how the explicit reference is created when we have same methods in the base interfaces which are implemented in the current class. If you have a look at the code, you can notice that both the interfaces, `ITerrestrialAnimal` and `IMarineAnimal` have a common method known as `Eat()`. If you want to implement these methods, then you will have two copies of the same method. In this case, to explicitly differentiate both from each other, we use `ITerrestrialAnimal.Eat()` and `IMarineAnimal.Eat()`.

In slide 33, explain the output of the code to the students and then tell them that in the code, the class `Crocodile` explicitly implements the method `Eat()` of the two interfaces, `ITerrestrialAnimal` and `IMarineAnimal`. Mention that the method `Eat()` is called by creating a reference of the two interfaces and then calling the method.

Slides 34 and 35

Understand interface inheritance with an example.

The slide has a blue header bar with the title "Interface Inheritance 1-5". Below the header, there is a bulleted list of points about interface inheritance. A section titled "Example" contains a detailed list of requirements for implementing multiple interfaces.

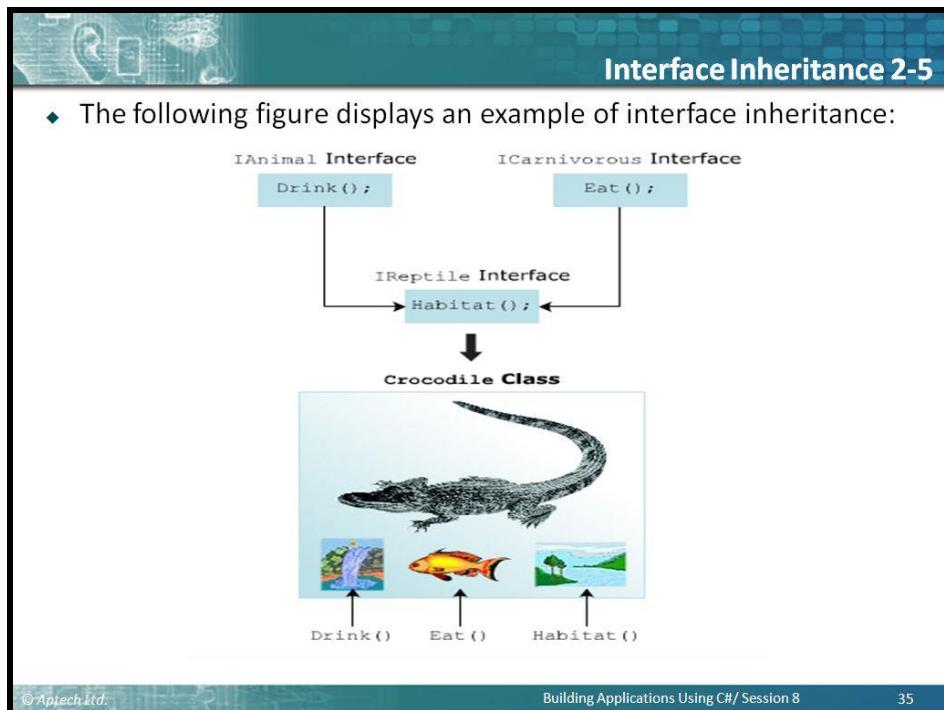
Interface Inheritance 1-5

- ◆ An interface can inherit multiple interfaces but cannot implement them. The implementation has to be done by a class.

Example

- ◆ 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.

© Aptech Ltd. Building Applications Using C# / Session 8 34



Using slide 34, tell the students that an interface can inherit multiple interfaces but cannot implement them. The implementation has to be done by a class.

Give them an example. Tell the students to consider three interfaces, **IAnimal**, **ICarnivorous** and **IReptile**. Then, tell them that the interface **IAnimal** declares methods defining general behavior of all animals. Also, Explain to the students that the interface **ICarnivorous** declares methods defining the general eating habits of carnivorous animals and the interface **IReptile** inherits interfaces **IAnimal** and **ICarnivorous**.

Tell them that these interfaces cannot be implemented by the interface **IReptile** as interfaces cannot implement methods. Mention that 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.

You can refer slide 35 to display and explain an example of interface inheritance.

Slide 36

Understand the syntax used to inherit an interface.

The slide has a blue header bar with the title "Interface Inheritance 3-5". Below the title, there is a bulleted list of points:

- ◆ The following syntax is used to inherit an interface:
 - Syntax
- ◆ where,
 - ◆ InterfaceName: Specifies the name of the interface that inherits another interface.
 - ◆ Inherited_InterfaceName: Specifies the name of the inherited interface.

Below the slide content, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C#/ Session 8", and "36".

In slide 36, explain the syntax that is used to inherit an interface and tell the students that InterfaceName specifies the name of the interface that inherits another interface and Inherited_InterfaceName: Specifies the name of the inherited interface.

Slide 37

Understand the code to declare interfaces inherited by other interfaces.

Interface Inheritance 4-5

- The following code declares interfaces that are inherited by other interfaces:

Snippet

```

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();
    }
}
```

Output

```
Crocodile
Can stay in Water and Land
Eats Flesh
Drinks fresh water
```

© Aptech Ltd. Building Applications Using C# / Session 8 37

In slide 37, explain the code that declares interfaces that are inherited by other interfaces. Then, explain the output of the code.

Additional Information

Refer the following links for more information on inheritance with interfaces:

<http://msdn.microsoft.com/en-us/library/aa664593%28v=vs.71%29.aspx>

<http://msdn.microsoft.com/en-us/library/ms173156.aspx>

Slide 38

Understand the code.

The slide has a blue header bar with the title "Interface Inheritance 5-5". The main content area contains a bulleted list:

- ◆ In the code:
 - ◆ Three interfaces, **IAnimal**, **ICarnivorous**, and **IReptile**, are declared.
 - ◆ The three interfaces declare methods **Drink()**, **Eat()**, and **Habitat()** respectively.
 - ◆ The **IReptile** interface inherits the **IAnimal** and **ICarnivorous** interfaces.
 - ◆ The class **Crocodile** implements the interface **IReptile**, its declared method **Habitat()** and the inherited methods **Eat()** and **Drink()** of the **ICarnivorous** and **IAnimal** interfaces.

At the bottom of the slide, there is footer text: "© Aptech Ltd.", "Building Applications Using C# / Session 8", and "38".

Tell the students that in the code shown in slide 38, three interfaces, **IAnimal**, **ICarnivorous**, and **IReptile**, are declared. Tell them that the three interfaces declare methods **Drink()**, **Eat()**, and **Habitat()** respectively. Then, tell them that the **IReptile** interface inherits the **IAnimal** and **ICarnivorous** interfaces.

Mention that the class **Crocodile** implements the interface **IReptile**, its declared method **Habitat()** and the inherited methods **Eat()** and **Drink()** of the **ICarnivorous** and **IAnimal** interfaces.

Slides 39 and 40

Understand interface re-implementation.

Interface Re-implementation 1-2

- ◆ 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.
- ◆ The following code demonstrates the purpose of re-implementation of an interface:

Snippet

```
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);
}
```

© Aptech Ltd. Building Applications Using C# / Session 8 39

Interface Re-implementation 2-2

- ◆ In the code:
 - ◆ 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 method in the derived class.
- ◆ The following figure displays the output of re-implementation of an interface:

© Aptech Ltd. Building Applications Using C# / Session 8 40

In slide 39, explain the interface re-implementation to the students.

Tell the students that a class can re-implement an interface. Tell them that 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.

Then, explain the code that demonstrates the purpose of re-implementation of an interface.

In slide 40, explain the code that demonstrates the purpose of re-implementation of an interface.

Tell the students that in the code, the interface **IMath** declares the method **Area()**. Tell them that the class **Circle** implements the interface **IMath** and declares a virtual method **Area()** that calculates the area of a circle.

Explain to the students that the class **Sphere** inherits the class **Circle** and overrides the base class method **Area()** to calculate the area of the sphere. Mention that the **base** keyword calls the base class method **Area()**, thereby allowing the use of base class method in the derived class.

Refer to the figure in slide 40 to display the output of re-implementation of an interface.

Slide 41

Understand the `is` and `as` operators in interfaces.

The diagram is titled "The `is` and `as` Operators in Interfaces 1-5". It features two main boxes: a red box for the `is Operator` and a green box for the `as Operator`. The `is Operator` box contains the text: "Checks the compatibility between two types or classes and returns a boolean value based on the check operation performed." The `as Operator` box contains the text: "Returns null if the two types or classes are not compatible with each other." A blue banner at the bottom of the slide reads "© Aptech Ltd." on the left, "Building Applications Using C#/ Session 8" in the center, and "41" on the right.

The `is` and `as` Operators in Interfaces 1-5

- ◆ The `is` and `as` operators in C# when used with interfaces, verify whether the specified interface is implemented or not.

is Operator

Checks the compatibility between two types or classes and returns a boolean value based on the check operation performed.

as Operator

Returns null if the two types or classes are not compatible with each other.

© Aptech Ltd. Building Applications Using C#/ Session 8 41

In slide 41, explain the `is` and `as` operators used with interfaces.

Tell the students that the `is` and `as` operators in C# verify whether the specified interface is implemented or not. Tell them that the `is` operator is used to check the compatibility between two types or classes and returns a boolean value based on the check operation performed.

On the other hand, mention that the `as` operator returns `null` if the two types or classes are not compatible with each other.

Slides 42 and 43

Understand the code that demonstrates an interface with the `is` operator.

The `is` and `as` Operators in Interfaces 2-5

- The following code demonstrates an interface with the `is` operator:

Snippet

```
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");
        }
    }
}
```

© Aptech Ltd. Building Applications Using C#/ Session 8 42

The `is` and `as` Operators in Interfaces 3-5

- In the code:
 - 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`.
- The following figure displays the output of the example using `is` operator:

© Aptech Ltd. Building Applications Using C#/ Session 8 43

Using slide 42, tell the students that in the code, an interface **ICalculate** declares a method **Area()**. Tell them that the class **Rectangle** implements the interface **ICalculate** and it consists of a parameterized constructor that assigns the dimension values of the rectangle.

Explain to the students that the **Area()** method calculates the area of the rectangle and the **Main()** method creates an instance of the class **Rectangle**.

Mention that the **is** operator is used within the **if-else** construct to check whether the class **Rectangle** implements the methods declared in the interface **ICalculate**.

Then, refer to the figure in slide 43 that displays the output of the example using **is** operator.

Slides 44 and 45

Understand the code that demonstrates an interface with the **as** operator.

The is and as Operators in Interfaces 4-5

- The code demonstrates an interface with the **as** operator:

Snippet

```

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");
        }
    }
}

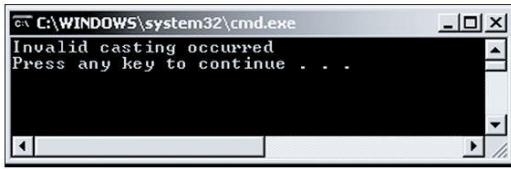
```

©Aptech Ltd. Building Applications Using C#/ Session 8 44

The `is` and `as` Operators in Interfaces 5-5

- ◆ In the code:
 - ◆ The interface `ISet` declares a method `AcceptDetails` with two parameters and the interface `IGet` declares a method `Display()`.
 - ◆ The class `Employee` implements the interface `ISet` and implements the method declared within `ISet`.
 - ◆ The `Main()` method creates an instance of the class `Employee`.
 - ◆ An attempt is made to retrieve an instance of `IGet` interface checks whether the class `Employee` implements the methods defined in the interface.
 - ◆ Since the `as` operator returns null, the code displays the specified error message.

- ◆ The following figure displays the output of the example that uses the `as` operator:



© Aptech Ltd. Building Applications Using C#/ Session 8 45

Using slide 44, tell the students that in the code, the interface `ISet` declares a method `AcceptDetails` with two parameters and the interface `IGet` declares a method `Display()`. Then, tell them that the class `Employee` implements the interface `ISet` and implements the method declared within `ISet`. Tell them that the `Main()` method creates an instance of the class `Employee`.

Explain to the students that an attempt is made to retrieve an instance of `IGet` interface checks whether the class `Employee` implements the methods defined in the interface. Mention that since the `as` operator returns null, the code displays the specified error message.

Then, refer to the figure in slide 45 that displays the output of the example that uses the `as` operator.

With this slide, you will finish explaining the `is` and `as` operators in interfaces.

Slides 46 and 47

Understand comparison of abstract classes and interfaces.

Abstract Classes and Interfaces 1-2

- ◆ 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.
- ◆ The similarities between abstract classes and interfaces are as follows:

Neither an abstract class nor an interface can be instantiated.

Both, abstract classes as well as interfaces, contain abstract methods.

Abstract methods of both, the abstract class as well as the interface, are implemented by the inheriting subclass.

Both, abstract classes as well as interfaces, can inherit multiple interfaces.

© Aptech Ltd. Building Applications Using C#/ Session 8 46

Abstract Classes and Interfaces 2-2

- ◆ The following figure displays the similarities between abstract class and interface:

```

graph LR
    AC[Abstract Class] -- "Cannot be Instantiated" --> IC[Interface]
    AC -- "Abstract Methods" --> IC
    Eat(Eat();) --> AC
    Sound(Sound();) --> IC
  
```

The diagram illustrates the similarities between an Abstract Class and an Interface. Both are represented by orange boxes. An arrow labeled "Cannot be Instantiated" points from the Abstract Class to the Interface. Another arrow labeled "Abstract Methods" points from the Abstract Class to the Interface. Below the boxes, there are method signatures: "Eat();" associated with the Abstract Class and "Sound();" associated with the Interface.

© Aptech Ltd. Building Applications Using C#/ Session 8 47

In slide 46, explain the comparison of abstract classes and interfaces to the students.

Tell the students that abstract classes and interfaces both declare methods without implementing them. Explain that although both abstract classes and interfaces share similar characteristics, they serve different purposes in a C# application. Then, explain the similarities between abstract classes and interfaces to the students. Tell them that neither an abstract class nor an interface can be instantiated. Tell that both, abstract classes as well as interfaces contain abstract methods.

Mention that abstract methods of both, the abstract class as well as the interface, are implemented by the inheriting subclass. Tell the students that both, abstract classes as well as interfaces, can inherit multiple interfaces.

Refer to the figure on slide 47 that depicts the similarities between abstract classes and interfaces.

In-Class Question:

After you finish explaining abstract classes and interfaces, you can ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



When does interface re-implementation occur?

Answer:

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.

Slide 48

Understand the differences between abstract classes and interfaces.

Differences Between an Abstract Class and an Interface

- ◆ 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 as shown in the following table:

Abstract Classes	Interfaces
An abstract class can inherit a class and multiple interfaces.	An interface can inherit multiple interfaces but cannot inherit a class.
An abstract class can have methods with a body.	An interface cannot have methods with a body.
An abstract class method is implemented using the <code>override</code> keyword.	An interface method is implemented without using the <code>override</code> keyword.
An abstract class is a better option when you need to implement common methods and declare common abstract methods.	An interface is a better option when you need to declare only abstract methods.
An abstract class can declare constructors and destructors.	An interface cannot declare constructors or destructors.

© Aptech Ltd. Building Applications Using C#/ Session 8 48

Tell the students that abstract classes and interfaces are similar because both contain abstract methods that are implemented by the inheriting class.

Then, refer table in slide 48 to show differences between abstract classes and interfaces. Explain the table to the students.

Slide 49

Understand the recommendations for using abstract classes and interfaces.

Recommendations for Using Abstract Classes and Interfaces

- ◆ 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.
- ◆ Following are the guidelines to decide when to use an interface and when to use an abstract class:

Interface	<ul style="list-style-type: none"> • 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. • Unlike abstract classes, interfaces cannot be changed once they are created. • A new interface needs to be created to create a new version of the existing interface.
Abstract class	<ul style="list-style-type: none"> • If a programmer wants to create different methods that are useful for different types of objects, it is recommended to create an interface. • 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.

© Aptech Ltd.

Building Applications Using C# / Session 8 49

In slide 49, explain the recommendations for using abstract classes and interfaces.

Tell the students that abstract classes and interfaces both contain abstract methods that are implemented by the inheriting class. Then, tell them that an abstract class can inherit another class whereas an interface cannot inherit a class. Explain to the students that abstract classes and interfaces have certain similarities as well as certain differences.

Explain to the students that there are some guidelines to decide when to use an interface and when to use an abstract class as follows:

- Explain to the students that if a programmer wants to create reusable programs and maintain multiple versions of these programs, it is recommended to create an abstract class. Tell them that abstract classes help to maintain the version of the programs in a simple manner because by updating the base class, all inheriting classes are automatically updated with the required change. Mention that unlike abstract classes, interfaces cannot be changed once they are created. A new interface needs to be created to create a new version of the existing interface.
- Explain to the students that if a programmer wants to create different methods that are useful for different types of objects, it is recommended to create an interface because abstract classes are widely created only for related objects. Tell them that there must exist a relationship between the abstract class and the classes that inherit the abstract class. On the other hand, mention that interfaces are suitable for implementing similar functionalities in dissimilar classes.

Slide 50

Summarize the session.



Summary

- ◆ An abstract class can be referred to as an incomplete base class and can implement methods that are similar for all the subclasses.
- ◆ IntelliSense provides access to member variables, functions, and methods of an object or a class.
- ◆ When implementing an interface in a class, you need to implement all the abstract methods declared in the interface.
- ◆ A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- ◆ A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- ◆ Re-implementation occurs when the method declared in the interface is implemented in a class using the `virtual` keyword and this virtual method is then overridden in the derived class.
- ◆ The `is` operator is used to check the compatibility between two types or classes and as operator returns `null` if the two types or classes are not compatible with each other.

© Aptech Ltd. Building Applications Using C# / Session 8 50

In slide 50, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- An abstract class can be referred to as an incomplete base class and can implement methods that are similar for all the subclasses.
- IntelliSense provides access to member variables, functions, and methods of an object or a class.
- When implementing an interface in a class, you need to implement all the abstract methods declared in the interface.
- A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- Re-implementation occurs when the method declared in the interface is implemented in a class using the `virtual` keyword and this virtual method is then overridden in the derived class.
- The `is` operator is used to check the compatibility between two types or classes and as operator returns `null` if the two types or classes are not compatible with each other.

8.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a question to students to search additional information, such as:

1. Can Abstract classes also define abstract methods? How?
2. What is the difference between Abstract classes and Sealed Classes?
3. What are constituents of interfaces?
4. What is the difference between interface and abstract class?

Session 9 - Properties and Indexers

9.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- An abstract class can be referred to as an incomplete base class and can implement methods that are similar for all the subclasses.
- IntelliSense provides access to member variables, functions, and methods of an object or a class.
- When implementing an interface in a class, you need to implement all the abstract methods declared in the interface.
- A class implementing multiple interfaces has to implement all abstract methods declared in the interfaces.
- A class has to explicitly implement multiple interfaces if these interfaces have methods with identical names.
- Re-implementation occurs when the method declared in the interface is implemented in a class using the `virtual` keyword and this virtual method is then overridden in the derived class.
- The `is` operator is used to check the compatibility between two types or classes and as operator returns `null` if the two types or classes are not compatible with each other.

Here, you can ask students the key topics they can recall from previous session. Ask them to briefly explain abstract classes in C# and also explain interfaces. Prepare a question or two which will be a key point to relate the current session objectives.

9.1.1 Objectives

By the end of this session, the learners will be able to:

- Define properties in C#
- Explain properties, fields, and methods
- Explain indexers

9.1.2 Teaching Skills

To teach this session successfully, you must know about properties in C#. You should be aware of properties, fields, and methods. You should also be familiar with indexers in C#.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here for the In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to properties. They will learn about fields and methods. This session will also discuss indexers in C#.

9.2 In-Class Explanations

Slide 3

Introduce properties in C#.

The slide has a blue header bar with the title 'Properties in C#' and decorative icons. The main content area has a white background with a black border. It contains two bulleted lists under the heading 'Properties in C#' and a section titled 'Example'.

Properties in C#

- ◆ Access modifiers such as public, private, protected, and internal control accessibility of fields and methods in C#.
 - ◆ public fields: accessible by other classes
 - ◆ private fields: accessible only by the class in which they are declared
- ◆ **Properties** in C# allow you to set and retrieve values of fields declared with any access modifier in a secured manner.

Example

- ◆ Consider fields that store names and IDs of employees.
- ◆ You can create properties for these fields to ensure accuracy and validity of values stored in them.

© Aptech Ltd. Building Applications Using C# / Session 9 3

Use slide 3 to explain that access modifiers such as public, private, protected, and internal are used to control the accessibility of fields and methods in C#.

Tell the students that the public fields are accessible by other classes, but private fields are accessible only by the class in which they are declared.

Mention that C# uses a feature called as properties that sets and retrieves values of fields declared with any access modifier in a secured manner. This is because properties validates values before assigning them to fields.

Give an example here. Tell the students that consider fields that store names and IDs of employees. By creating properties for these fields, ensures accuracy and validity of values stored in them.

Slide 4

Understand the applications of properties.

The slide has a teal header bar with the title 'Applications of Properties'. Below the header, there are two bullet points:

- ◆ Properties:
 - ❖ allow to protect a field in the class by reading and writing to the field through a property declaration.
 - ❖ allow to access private fields, which would otherwise be inaccessible.
 - ❖ can validate values before allowing you to change them and also perform specified actions on those changes.
 - ❖ ensure security of private data.
 - ❖ support abstraction and encapsulation by exposing only necessary actions and hiding their implementation.
- ◆ The following syntax is used to declare a property in C#.

A blue box labeled 'Syntax' contains the following code snippet:

```
<access_modifier><return_type><PropertyName>
{
    //body of the property
}
```

Below the code, the text 'where,' is followed by three bullet points:

- ❖ 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.

At the bottom left is the copyright notice '© Aptech Ltd.' and at the bottom right is the page number '4'.

In slide 4, explain to the students that properties protect a field in the class by reading and writing to the field through a property declaration. Additionally, properties help to access private fields, which would otherwise be inaccessible.

Tell them that properties can validate values before changing them and also perform specified actions on those changes. Therefore, properties ensure security of private data.

Tell them that properties support abstraction and encapsulation by exposing only necessary actions and hiding their implementation.

Also, explain to them about the syntax used to declare a property in C# as given on the slide.

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.

Additional Information

Refer the following links for more information on properties:

<http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx>

<http://csharpindepth.com/articles/chapter8/propertiesmatter.aspx>

Slides 5 to 8

Understand the get and set accessors.

get and set Accessors 1-3

- Property accessors allow you to read and assign a value to a field by implementing **get** and **set** accessors as follows:

The get accessor	<ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> 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.

Syntax

```
<access_modifier><return_type>PropertyName>
{
    get
    {
        // return value
    }
    set
    {
        // assign value
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 9

5

get and set Accessors 2-3

- The following code demonstrates the use of the **get** and **set** accessors.

Snippet

```
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);
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 9

6

get and set Accessors 3-3

- ◆ In the code:
 - ❖ The class **SalaryDetails** creates a private variable **_empName** and declares a property called **EmployeeName**.
 - ❖ The instance of the **SalaryDetails** class, **objSal**, invokes the property **EmployeeName** using the dot (.) operator to initialize the value of employee name. This invokes the **set** accessor, where the **value** keyword assigns the value to **_empName**.
 - ❖ The code displays the employee name by invoking the property name.
 - ❖ This invokes the **get** accessor, which returns the assigned employee name. This invokes the **set** accessor, where the **value** keyword assigns the value to **_empName**.

Output

- ❖ Employee Name: Patrick Johnson

© Aptech Ltd. Building Applications Using C# / Session 9 7

Categories of Properties 1-10

- ◆ Properties are broadly divided into three categories:

```

graph TD
    Properties[Properties] --- ReadOnly[Read-only Property]
    Properties --- WriteOnly[Write-only Property]
    Properties ---ReadWrite[Read-Write Property]
  
```

© Aptech Ltd. Building Applications Using C# / Session 9 8

Using slide 5, you will explain the students that the property accessors read and assign a value to a field by implementing two special methods. These methods are referred to as the **get** and **set** accessors. Tell them in detail about 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.

Explain the syntax used to declare the accessors of a property.

Use slide 6 to show the students the code that demonstrates the use of the get and set accessors. In slide 7, explain the code. Tell the students that the class `SalaryDetails` creates a private variable `_empName` and declares a property called `EmployeeName`. The instance of the `SalaryDetails` class, `objSal`, invokes the property `EmployeeName` using the dot (.) operator to initialize the value of employee name. This invokes the set accessor, where the `value` keyword assigns the value to `_empName`.

The code displays the employee name, by invoking the property name. This invokes the get accessor, which returns the assigned employee name.

It is mandatory to always end the get accessor with a `return` statement.

Use the figure on slide 8 to explain the students that properties are broadly divided into three categories, read-only, write-only, and read-write properties. Read-Only property will allow user to read the value from the object. Write-Only property allows setting the value of that property. Read-Write property allows the user to get as well as set the property.

Slide 9

Understand read-only property.

Categories of Properties 2-10

- Read-Only Property:**
 - The read-only property allows you to retrieve the value of a private field. To create a read-only property, you should define the `get` accessor.
 - The following syntax creates a read-only property.

Syntax
<pre><access_modifier><return_type><PropertyName>{ get { // return value } }</pre>

© Aptech Ltd. Building Applications Using C# / Session 9 9

In slide 9, tell the students that the read-only property retrieves the value of a private field. To create a read-only property, define the `get` accessor. Then, explain the syntax that creates a read-only property.

Additional Information

Refer the following links for more information on properties:

<http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx>

<http://csharpindepth.com/articles/chapter8/propertiesmatter.aspx>

Slides 10 and 11

Understand how to create a read-only property.

Categories of Properties 3-10

- The following code demonstrates how to create a read-only property.

Snippet

```
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);
    }
}
```

Categories of Properties 4-10

- In the code:
 - 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

In slide 10, show the code to the students that demonstrates how to create a read-only property.

Use slide 11 to explain the code and the output. Tell the students that in the code:

- The **Books** class creates two read-only properties that return 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.

Additional Information

Refer the following links for more information on properties:

<http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx>

<http://csharpindepth.com/articles/chapter8/propertiesmatter.aspx>

Slide 12

Understand the write-only property.

The slide has a blue header bar with the title "Categories of Properties 5-10". Below the header, there is a section titled "◆ Write-Only Property:" with three bullet points:

- ◆ The write-only property allows you to change the value of a private field.
- ◆ To create a write-only property, you should define the `set` accessor.
- ◆ The following syntax creates a write-only property.

 A yellow box labeled "Syntax" contains the following C# code:


```
<access_modifier><return_type><PropertyName>
{
    set
    {
        // assign value
    }
}
```

In slide 12, tell the students that the write-only property changes the value of a private field.

Define the `set` accessor to create a write-only property.

Also, explain the syntax that creates a write-only property.

Additional Information

Refer the following links for more information on properties:

<http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx>

<http://csharpindepth.com/articles/chapter8/propertiesmatter.aspx>

Slides 13 and 14

Understand the code to create a write-only property.

Categories of Properties 6-10

◆ The following code demonstrates how to create a write-only property.

Snippet

```
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();
    }
}
```

Categories of Properties 7-10

◆ In the code:

- ◆ 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

In slide 13, explain the code to the students that creates a write-only property. Use slide 14 to explain the code and the output. Mention that in the code, 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.

Slide 15

Understand the read-write property.

The slide has a blue header bar with the title "Categories of Properties 8-10". Below the header, there is a section titled "◆ Read-Write Property:" with two bullet points. The first bullet point explains that the read-write property allows setting and retrieving the value of a private field by defining set and get accessors. The second bullet point states that the following syntax creates a read-write property. A "Syntax" button is present, and below it is a code block:

```
<access_modifier><return type><PropertyName>
{
    get
    {
        // return value
    }
    set
    {
        // assign value
    }
}
```

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 9", and "15".

In slide 15, tell the students that the read-write property sets and retrieves the value of a private field. Define the set and get accessors to create a read-write property. Then, explain the syntax that creates a read-write property.

Slides 16 and 17

Understand how to create a read-write property.

Categories of Properties 9-10

- The following code demonstrates how to create a read-write property.

Snippet

```

using System;
class Product {
    string _productName;
    int _productID;
    float _price;
    public Product(string name, intval) {
        _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();
    }
}

```

© Aptech Ltd. Building Applications Using C# / Session 9 16

Categories of Properties 10-10

- In the code:
 - The class **Product** creates a read-write property **Price** that assigns and retrieves the price of the product based on the if statement.
 - The **Goods** class defines the **Main ()** method that creates an instance of the class **Product** by passing the values as parameters that are name and ID of the product.
 - The **Display()** method of the class **Product** is invoked that displays the name, ID, and price of the product.

Output

- Product Name: Hard Disk
- Product ID: 101
- Price: 345.25\$

- Properties can be further classified as static, abstract, and boolean properties.

© Aptech Ltd. Building Applications Using C# / Session 9 17

In slide 16, show the code that demonstrates how to create a read-write property to the students. Use slide 17 to explain the code and output. Mention that in the code, the class **Product** creates a read-write property **Price** that assigns and retrieves the price of the product based on the if statement. The **Goods** class defines the **Main()** method that creates an instance of the class **Product** by passing the values as parameters that are name

and ID of the product. The **Display()** method of the class **Product** is invoked that displays the name, ID, and price of the product.

Then, tell them that properties can be further classified as static, abstract, and boolean properties.

The **Product** class creates read-write property **Price**. This property allows the user to read the value as well to set the value of this variable. The **Goods** class creates an object of **Products** class. The **name** and **ID** properties of the **Product** class are assigned a value and displayed using **Display()** method.

Slides 18 to 20

Understand static properties.

Static Properties 1-3

- ◆ The static property is:
 - ❖ declared by using the `static` keyword.
 - ❖ accessed using the class name and thus, belongs to the class rather than just an instance of the class.
 - ❖ by a programmer without creating an instance of the class.
 - ❖ used to access and manipulate static fields of a class in a safe manner.
- ◆ The following code demonstrates a class with a static property.

Snippet

```
using System;
class University
{
    private static string _department;
    private static string _universityName;
    public static string Department
    {
        get
        {
            return _department;
        }
        set
        {
            _department = value;
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 9 18

Static Properties 2-3

Snippet

```

        }
        public static string UniversityName
        {
            get { return _universityName; }
            set { _universityName = value; }
        }
    }
    class Physics
    {
        static void Main(string[] args)
        {
            University.UniversityName = "University of Maryland";
            University.Department = "Physics";
            Console.WriteLine("University Name: " + University.UniversityName);
            Console.WriteLine("Department name: " + University.Department);
        }
    }
}

```

Building Applications Using C# / Session 9 19

Static Properties 3-3

In the code:

- ◆ 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

Building Applications Using C# / Session 9 20

Use slides 18 and 19 to show the code that demonstrates a class with a static property. In slide 18, tell the students that 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. Mention that a static property is used to access and manipulate static fields of a class in a safe manner. The **Physics** class sets the values of properties **UniversityName** and **Department** without creating an instance of the class **University**. This is possible because

both these properties are declared as static. Thus, static properties can be used to directly initialize the values of variables.

In slide 20, tell the students that in the code 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.

Then explain that 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.

Explain the output of the code.

Additional Information

Refer the following links for more information on static properties:

<http://msdn.microsoft.com/en-us/library/aa664451%28v=vs.71%29.aspx>

<http://www.c-sharpcorner.com/UploadFile/rajeshvs/PropertiesInCS11122005001040AM/PropertiesInCS.aspx>

Slides 21 to 23

Understand abstract properties.

Abstract Properties 1-3

Abstract properties:

- ◆ Declared by using the **abstract** keyword.
- ◆ Contain only the declaration of the property without the body of the **get** and **set** accessors (which do not contain any statements and can be implemented in the derived class).
- ◆ Are only allowed in an abstract class.
- ◆ Are used:
 - ◆ when it is required to secure data within multiple fields of the derived class of the abstract class.
 - ◆ to avoid redefining properties by reusing the existing properties.
- ◆ The following code demonstrates a class that uses an abstract property.

Snippet

```
using System;
public abstract class Figure {
    public abstract float DimensionOne {
        set;
    }
    public abstract float DimensionTwo {
        set; }
}
class Rectangle : Figure {
    float dimensionOne;
```

Abstract Properties 2-3

Snippet

```

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) {
}

```

© Aptech Ltd.

Building Applications Using C# / Session 9

22

Abstract Properties 3-3

Snippet

```

Rectangle objRectangle = new Rectangle();
objRectangle.DimensionOne = 20;
objRectangle.DimensionTwo = 4.233F;
Console.WriteLine("Area of Rectangle: " + objRectangle.Area());
}
}

```

◆ In the code:

- ❖ 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

```

© Aptech Ltd.

Building Applications Using C# / Session 9

23

In slide 21, explain abstract properties. Tell them that:

- Abstract properties are declared by using the **abstract** keyword.
- They contain the declaration of the property without the body of the get and set accessors. The get and set accessors do not contain any statements. These accessors can be implemented in the derived class.
- They are allowed only in an abstract class.
- They are used to secure data within multiple fields of the derived class of the abstract class and to avoid redefining properties by reusing the existing properties.

Use slides 22 and 23 slide to explain the code that demonstrates a class that uses an abstract property.

Explain the code and the output. Mention that in the code:

- 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.

Explain to the students that the abstract class in the given slide no 23 declares a class called **Figure** which has two properties **DimensionOne** and **DimensionTwo**. The values in **DimensionOne** and **DimensionTwo** are set using properties. The method **Area()** calculates the **Area()** of the given rectangle. Finally the method **Area()** is invoked through **Main()** that give the output 'Area of Rectangle : 84.66'.

Additional Information

Refer the following link for more information on abstract properties:

<http://msdn.microsoft.com/en-us/library/yd3z1377.aspx>

Slide 24

Understand boolean properties.

The slide has a teal header bar with the title 'Boolean Properties'. The main content area contains three bullet points about boolean properties. To the right is a small graphic showing 'True' with a checked checkbox and 'False' with an empty checkbox. The footer bar includes the copyright notice '@Aptech Ltd.', the slide title 'Building Applications Using C# / Session 9', and the page number '24'.

- ◆ 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 boolean property, a programmer needs to be sure that the get accessor returns the boolean value.

True	False
<input checked="" type="checkbox"/>	<input type="checkbox"/>

In slide 24, explain to the students that a boolean property is declared by specifying the data type of the property as `bool`.

Mention that unlike other properties, the boolean property produces only true or false values. While working with boolean property, a programmer needs to be sure that the get accessor returns the boolean value.

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 `static` class.

Slides 25 to 27

Understand implementing properties.

Implementing Inheritance 1-3

- ◆ Properties can be inherited just like other members of the class.
- ◆ The base class properties are inherited by the derived class.
- ◆ The following code demonstrates how properties can be inherited.

Snippet

```
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 + "$");
    }
}
```

Implementing Inheritance 2-3

Snippet

```
    }
}
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 + "$");
    }
}
```

Implementing Inheritance 3-3

- ◆ In the code:
 - ◆ 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.
 - ◆ 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\$

Use slides 25, 26, and 27 to explain the code that demonstrates how properties can be inherited.

In slides 25 and 26, explain to the students that properties can be inherited just like other members of the class. This means the base class properties are inherited by the derived class. In slide 27, explain the code and the output.

Mention that in the code, 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**.

Then, tell that 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.

Explain to the students that class **Employee** is having three properties, **name**, **ID** and **salary**.

The class **SalaryDetails** is derived from the **Employee** class and is inherits all the public properties and methods of **Employee** class. The variables **_empID**, **_empName**, and **salary** are private members of the class **Employee** and **EmpID**, **EmpName**, and **Salary** are the public properties of the class **Employee**. The values in the three variables is shown through properties **EmpID**, **EmpName**, and **Salary**.

Slides 28 to 30

Understand auto-implemented properties.

Auto-Implemented Properties 1-3

- ◆ C# provides an alternative syntax to declare properties where a programmer can specify a property in a class without explicitly providing the get and set accessors.
- ◆ Such properties are called auto-implemented properties and results in more concise and easy-to-understand programs.
- ◆ For an auto-implemented property, the compiler automatically creates:
 - ◊ a private field to store the property variable.
 - ◊ the corresponding get and set accessors.
- ◆ The following syntax creates an auto-implemented property.

Syntax

```
public string Name { get; set; }
```

- ◆ The following code uses auto-implemented properties.

Snippet

```
class Employee
{
    public string Name { get; set; }
    public int Age { get; set; }
```

© Aptech Ltd.

Building Applications Using C# / Session 9

28

Auto-Implemented Properties 2-3

Snippet

```
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);
}
```

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

Output

```
Name: John Doe, Age: 24, Designation: Sales Person
```

© Aptech Ltd.

Building Applications Using C# / Session 9

29



Auto-Implemented Properties 3-3

- ◆ Like normal properties, auto-implemented properties can be declared to be read-only and write-only.
- ◆ The following code declares read-only and write-only properties.

Snippet

```
public float Age { get; private set; }
public int Salary { private get; set; }
```

In the code:

- ◆ 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.

© Aptech Ltd.

Building Applications Using C# / Session 9 30

In slide 28, tell the students that 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.

Explain that such properties are called as auto-implemented properties and results in more concise and easy-to-understand programs. For an auto-implemented property, the compiler automatically creates a private field to store the property variable. In addition, the compiler automatically creates the corresponding `get` and `set` accessors.

Explain that the syntax: `public string Name { get; set; }` creates an auto-implemented property.

Use slides 29 and 30 to demonstrate how the code uses auto-implemented properties. Explain the code. The code declares three auto-implemented properties: **Name**, **Age**, and **Designation**. The `Main()` method first sets the values of the properties, and then retrieves the values and writes to the console.

Explain the output of the code:

`Name : John Doe, Age : 24, Designation : Sales Person`

Then, tell the students that similar to normal properties, auto-implemented properties can be declared to be read-only and write-only.

Then, show the code that declares read-only and write-only properties and explain the code. In the code, 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.

Unlike normal properties, auto-implemented properties cannot be assigned a default value at the time of declarations. Any assignment of value to an auto implemented property must be done in the constructor. In addition, auto-implemented properties cannot provide additional functionalities, such as data validations in either of the accessors. Such properties are only meant for simple storage of values.

Additional Information

Refer the following links for more information on auto-implemented properties:

<http://msdn.microsoft.com/en-us/library/bb384054.aspx>

<http://www.blackwasp.co.uk/CSharpAutomaticProperties.aspx>

Slide 31

Understand object initializers.

Object Initializers 1-2

- ◆ In C#, programmers 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.

© Aptech Ltd. Building Applications Using C# / Session 9 31

In slide 31, tell the students that 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.

Mention that 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.

Slide 32

Understand how to initialize an **Employee** object.

Object Initializers 2-2

- The following code uses object initializers to initialize an **Employee** object.

Snippet

```
class Employee {
    public string Name { get; set; }
    public int Age { get; set; }
    public string Designation { get; set; }
    static void Main (string [] args) {
        Employee empl = new Employee {
            Name = "John Doe",
            Age = 24,
            Designation = "Sales Person"
        };
        Console.WriteLine("Name: {0}, Age: {1}, Designation: {2}", empl.Name,
        empl.Age, empl.Designation);
    }
}
```

- The code 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
```

© Aptech Ltd. Building Applications Using C# / Session 9 32

In slide 32, explain to the students that the code uses object initializers to initialize an **Employee** object.

Tell them that the code creates the output three auto-implemented properties in an **Employee** class. And the **Main ()** method uses an object initializer to create an **Employee** object initialized with values of its properties.

Explain the output of the code:

Name: John Doe, Age: 24, Designation: Sales Person

Slides 33 and 34

Understand implementing polymorphism.

Implementing Polymorphism 1-2

- ◆ Properties can implement polymorphism by overriding the base class properties in the derived class.
- ◆ However, properties cannot be overloaded.
- ◆ The following code demonstrates the implementation of polymorphism by overriding the base class properties.

Snippet

```
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";
```

© Aptech Ltd.

Building Applications Using C# / Session 9

33

Implementing Polymorphism 2-2

Snippet

```
Ferrari objFerrari = new Ferrari();
objFerrari.CarType = "Sports Car";
Console.WriteLine("Car Type: " + objCar.CarType);
Console.WriteLine("Ferrari Car Type: " + objFerrari.CarType);
}
```

- ◆ 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 cartype.
- ◆ 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

© Aptech Ltd.

Building Applications Using C# / Session 9

34

Use slides 33 and 34 to explain the code that demonstrates the implementation of polymorphism by overriding the base class properties. In slide 33, tell the students that the properties can implement polymorphism by overriding the base class properties in the derived class. However, properties cannot be overloaded.

Explain the code. In the code, 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.

Explain the output of the code:

Car Type: Utility Vehicle

Ferrari Car Type: Sports Car

In-Class Question:

After you finish explaining the properties, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are the three categories of properties?

Answer:

The three categories of properties are read-only, write-only, and read-write properties.

Slide 35

Understand properties, fields, and methods.

Properties, Fields, and Methods 1-2

- ◆ 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.
- ◆ Properties are similar to fields as both contain values that can be accessed. However, there are certain differences between them.
- ◆ The following table shows 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 ref and out keywords.	Fields are variables that can use the ref and out 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 get and set 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.

© Aptech Ltd.

Building Applications Using C# / Session 9 35

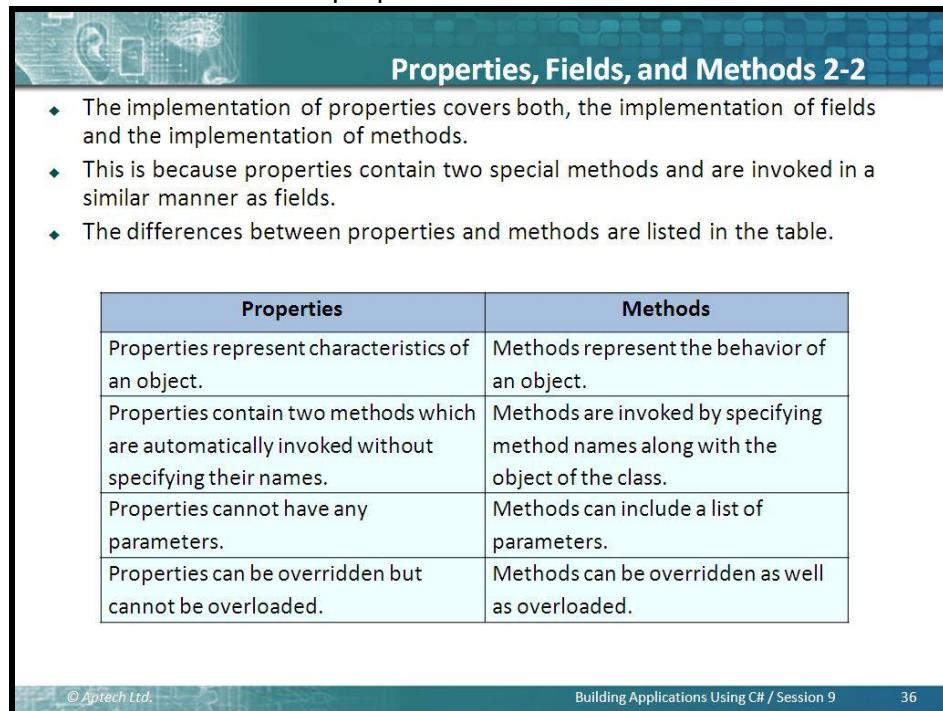
Use slide 35 to explain the students that a class in a C# program can contain a mix of properties, fields, and methods, each serving a different purpose in the class. It is important to understand the differences between them in order to use them effectively in the class.

Mention that properties are similar to fields as both contain values that can be accessed. However, there are certain differences between them.

Then, explain the table that shows the differences between properties and fields.

Slide 36

Understand the difference between properties and methods.



The slide has a title "Properties, Fields, and Methods 2-2" at the top. Below the title is a bulleted list of three points:

- ◆ The implementation of properties covers both, the implementation of fields and the implementation of methods.
- ◆ This is because properties contain two special methods and are invoked in a similar manner as fields.
- ◆ The differences between properties and methods are listed in the table.

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.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 9", and "36".

In slide 36, explain the students that the implementation of properties covers both, the implementation of fields and the implementation of methods. This is because properties contain two special methods and are invoked in a similar manner as fields.

Explain the table that shows the differences between properties and methods.

With this slide, you will finish explaining properties, fields, and methods.

Slides 37 and 38

Understand indexers.

Indexers 1-2

- ◆ 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.

Example

- ◆ Consider a high school teacher who wants to go through the records of a particular student to check the student's progress.
- ◆ Calling the appropriate methods every time to set and get a particular record makes the task tedious.
- ◆ Creating an indexer for student ID:
 - ◊ makes the task of accessing the record much easier as indexers use index position of the student ID to locate the student record.

Indexer	Student_Details	
StudentID	StudentID	StudentName
S001	S003	John
S002	S002	Smith
S003	S004	Albert
S004	S001	Rosa

© Aptech Ltd. Building Applications Using C# / Session 9 37

Indexers 2-2

- ◆ Indexers:
 - ◊ are data members that allow you to access data within objects in a way that is similar to accessing arrays.
 - ◊ provide faster access to the data within an object as they help in indexing the data.
 - ◊ allows you to use the index of an object to access the values within the object.
 - ◊ are also known as smart arrays in C#.
- ◆ In arrays, you use the index position of an object to access its value.
- ◆ The implementation of indexers is similar to properties, except that the declaration of an indexer can contain parameters.
- ◆ Indexers allow you to index a class, struct, or an interface.

© Aptech Ltd. Building Applications Using C# / Session 9 38

Use slide 37 to explain that 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.

Give an example here. Tell them to consider a high school teacher who wants to go through the records of a particular student to check the students' progress. If the teacher calls the appropriate methods every time to set and get a particular record, the task becomes a little tedious. On the other hand, if the teacher creates an indexer for student ID, it makes the task of accessing the record much easier. This is because indexers use index position of the student ID to locate the student record.

Tell them that the figure displays the index position of indexers for this example.

Use slide 38 to explain that the indexers are data members that 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 index position of an object is used to access its value. Similarly, an indexer uses the index of an object to access the values within the object.

Mention that 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.

Additional Information

Refer the following link for more information on indexers:

<http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx>

Slide 39

Understand declaration of indexers.

Declaration of Indexers 1-2

- ◆ 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 the identifier of the index.
 - ❖ The open and close curly braces, which contain the declaration of the `set` and `get` accessors.
- ◆ The following syntax creates an indexer.

Syntax

```
<access_modifier><return_type> this [<parameter>]
{
  get { // return value }
  set { // assign value }
}
```

©Aptech Ltd.

Building Applications Using C# / Session 9

39

Using slide 39, mention that 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.

Tell them that the syntax given on the slide 39 creates an indexer and explain the syntax.

Slide 40

Understand the use of indexers.

Declaration of Indexers 2-2

- The following code demonstrates the use of indexers.

Snippet

```
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");
        }
    }
}
```

- The class **EmployeeDetails** creates an indexer that takes a parameter of type **int**.
- The instance of the class, **objEmp**, is assigned values at each index position.
- The **set** accessor is invoked for each index position.
- The **for** loop iterates for two times and displays values assigned at each index position using the **get** accessor.

Output

```
Employee Names : Jack Anderson Kate Jones
```

© Aptech Ltd. Building Applications Using C# / Session 9 40

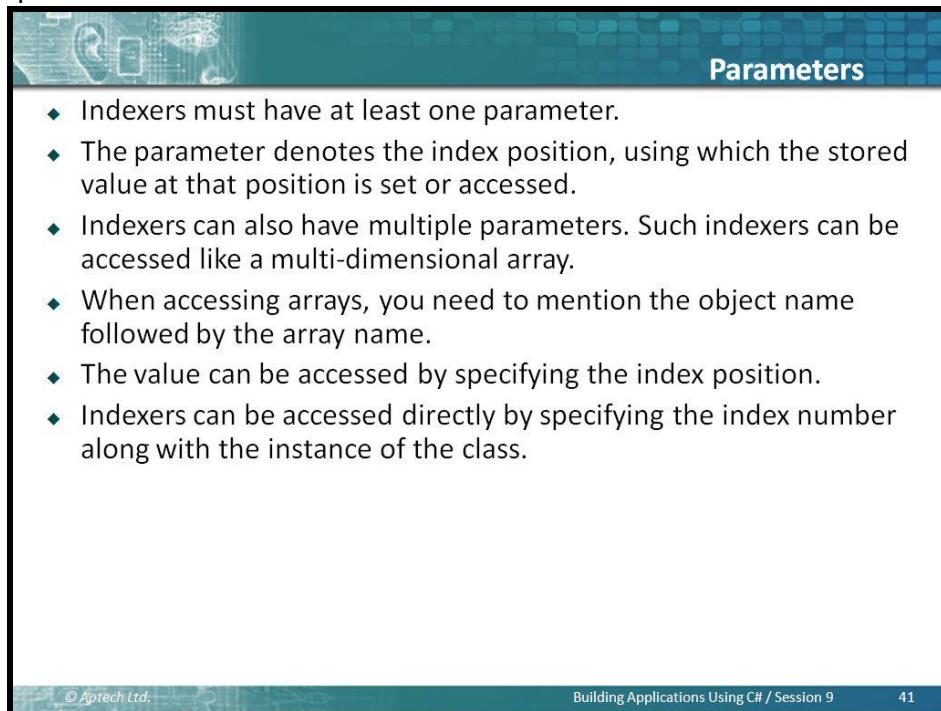
In slide 40, explain to the students how code demonstrates the use of indexers.

Explain the code. Tell that in the code, the class **EmployeeDetails** creates an indexer that takes a parameter of type **int**. The instance of the class, **objEmp**, is assigned values at each index position. The **set** accessor is invoked for each index position. The **for** loop iterates for two times and displays values assigned at each index position using the **get** accessor.

Explain the output of the code.

Slide 41

Understand parameters.



The slide has a teal header bar with the title 'Parameters'. At the bottom left is a yellow lightbulb icon with radiating lines. The main content area contains a bulleted list of six points about indexers:

- ◆ 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.
- ◆ Indexers can also have multiple parameters. Such indexers can be accessed like a multi-dimensional array.
- ◆ When accessing arrays, you need to mention the object name followed by the array name.
- ◆ The value can be accessed by specifying the index position.
- ◆ Indexers can be accessed directly by specifying the index number along with the instance of the class.

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 9' in the center, and the page number '41' on the right.

In the slide 41, explain to the students that 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.

Tell that when accessing arrays, 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.

In-Class Question:

Ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are indexers?

Answer:

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.

Slides 42 and 43

Understand implementing inheritance.

Implementing Inheritance 1-2

- ◆ Indexers can be inherited like other members of the class.
- ◆ It means the base class indexers can be inherited by the derived class.
- ◆ The following code demonstrates the implementation of inheritance with indexers.

Snippet

```
using System;
class Numbers
{
    private int[] num = new int[3];
    public int this[int index]
    {
        get { return num [index]; }
        set { num [index] = value; }
    }
}
class EvenNumbers : Numbers {
    public static void Main()
    {
        EvenNumbers objEven = new EvenNumbers();
        objEven[0] = 0;
        objEven[1] = 2;
        objEven[2] = 4;
        for(int i=0; i<3; i++)
        {
            Console.WriteLine(objEven[i]);
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 9 42

Implementing Inheritance 2-2

- ◆ In the code:
 - ◆ 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.
 - ◆ By inheriting, an indexer in the base class can be reused in the derived class.

Output

- ◆ 0
- ◆ 2
- ◆ 4

© Aptech Ltd. Building Applications Using C# / Session 9 43

Use slide 42 to explain that indexers can be inherited like other members of the class. This means that the base class indexers can be inherited by the derived class.

Explain the code that demonstrates the implementation of inheritance with indexers.

Use slide 43 to explain the code. Tell them that in the code 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. Explain the output of the code:

0
2
4

Slides 44 and 45

Understand implementing polymorphism.

Implementing Polymorphism Using Indexers 1-3

- ◆ Indexers can implement polymorphism by overriding the base class indexers or by overloading indexers.
- ◆ A particular class can include more than one indexer having different signatures. This feature of polymorphism is called **overloading**.
- ◆ Thus, polymorphism allows the indexer to function with different data types of C# and generate customized output.
- ◆ The following code demonstrates the implementation of polymorphism with indexers by overriding the base class indexers.

Snippet

```
class EvenNumbers : Numbers
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] {
```

© Aptech Ltd. Building Applications Using C# / Session 9 44

The screenshot shows a presentation slide with a blue header bar. The title 'Implementing Polymorphism Using Indexers 2-3' is centered in the header. Below the header is a section titled 'Snippet' in a dark blue box. The snippet contains the following C# code:

```
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");
}
}
```

At the bottom of the slide, there is a footer bar with the text '©Aptech Ltd.' on the left, 'Building Applications Using C# / Session 9' in the center, and the number '45' on the right.

In slide 44, tell the students that 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.

Explain to them that 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. Use slides 44 and 45 to explain how to demonstrate the implementation of polymorphism with indexers by overriding the base class indexers.

Slide 46

Implementing Polymorphism Using Indexers 3-3

- ◆ In the code:
 - ◆ The class **Student** declares an array variable and a virtual indexer.
 - ◆ The class **Result** inherits the class **Student** and overrides the virtual indexer.
 - ◆ The **Main()** method declares an instance of the base class **Student** and the derived class **Result**.
 - ◆ When the instance of the class **Student** is assigned values at each index position, the **set** accessor of the class **Student** is invoked.
 - ◆ When the instance of the class **Result** is assigned values at each index position, the **set** accessor of the class **Result** is invoked.
 - ◆ This overrides the base class indexer.
 - ◆ The **set** accessor of the derived class **Result** invokes the base class indexer by using the **base** keyword.
 - ◆ The **for** loop displays values at each index position by invoking the **get** accessors of the appropriate classes.

Output

- ◆ Peter First class
- ◆ Patrick Pass class

© Aptech Ltd. Building Applications Using C# / Session 9 46

Use slide 46 to explain the code and the output. Tell that in the code, 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.

Then, tell that 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.

Explain the output:

Peter First class
Patrick Pass class

In-Class Question:

Ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is implementing polymorphism?

Answer:

Implementing polymorphism allows the derived class indexers to override the base class indexers. Also, a particular class can include more than one indexer having different signatures.

Slides 47 to 49

Understand multiple parameters in indexers.

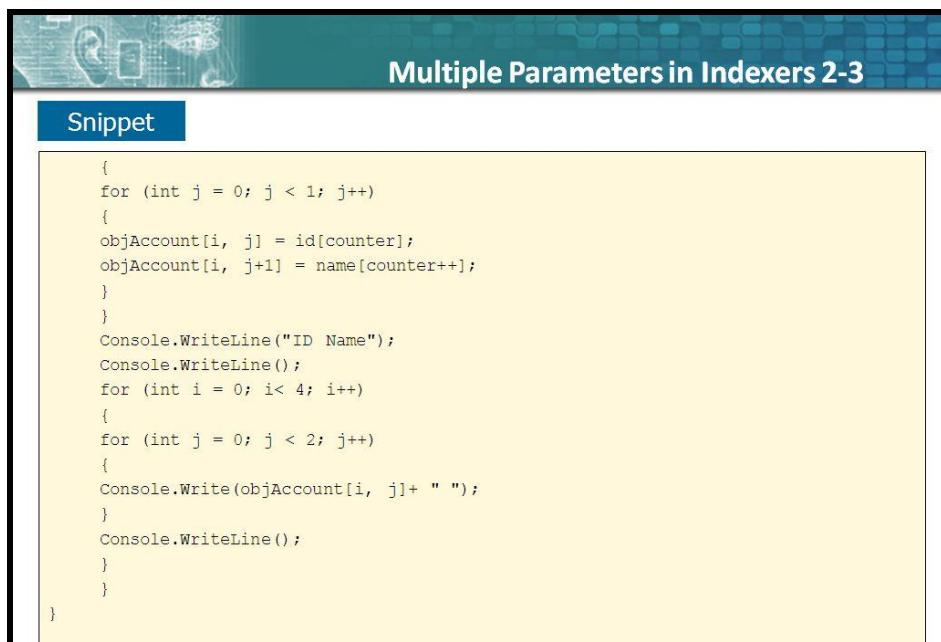

Multiple Parameters in Indexers 1-3

- ◆ Indexers must be declared with at least one parameter within the square bracket notation ([]).
- ◆ 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.
- ◆ The following code demonstrates how multiple parameters can be passed to an indexer.

Snippet

```
using System;
class Account {
    string[,] accountDetails = new string[4, 2];
    public string this[int pos, int column] {
        get { return (accountDetails[pos, column]); }
        set { accountDetails[pos, column] = value; }
    }
    static void Main(string[] args)
    {
        Account objAccount = new Account();
        string[] id = new string[3] { "1001", "1002", "1003" };
        string[] name = new string[3] { "John", "Peter", "Patrick" };
        int counter = 0;
        for (int i = 0; i < 3; i++)
            objAccount[i, 0] = id[i];
        for (int i = 0; i < 3; i++)
            objAccount[0, i] = name[i];
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 9 47



Multiple Parameters in Indexers 2-3

Snippet

```
{  
    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();  
    }  
}
```

© Aptech Ltd.

Building Applications Using C# / Session 9

48

Multiple Parameters in Indexers 3-3

- ◆ In the code:

- ◆ 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.

Output

- ◆ ID Name
- ◆ 1001 John
- ◆ 1002 Peter
- ◆ 1003 Patrick

© Aptech Ltd.

Building Applications Using C# / Session 9

49

Use slide 47 to explain the students that 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.

Mention that 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.

Use slides 47 and 48 to demonstrate how multiple parameters can be passed to an indexer.

Use slide 49 to explain the code. Tell that in the code, 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.

Explain the output of the code:

```
ID Name
1001 John
1002 Peter
1003 Patrick
```

Slide 50

Understand indexers in interfaces.

Indexers in Interfaces 1-3

- ◆ Indexers can also be declared in interfaces.
- ◆ 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.

@Aptech Ltd.

Building Applications Using C# / Session 9 50

In slide 50, explain that indexers can also be declared in interfaces. However, the accessors of indexers declared in interfaces differ from the indexers declared within a class.

Then, tell that 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. This enforces reusability and provides the flexibility to customize indexers.

Slides 51 and 52

Understand the implementation of interface indexers.

Indexers in Interfaces 2-3

- The following code demonstrates the implementation of interface indexers.

Snippet

```

using System;
public interface IDetails {
    string this[int index]
    { get; set; }
}
class Students :IDetails {
    string [] studentName = new string[3];
    int[] studentID = new int[3];
    public string this[int index] {
        get { return studentName[index]; }
        set { studentName[index] = value; }
    }
    static void Main(string[] args) {
        Students objStudent = new Students();
        objStudent[0] = "James";
        objStudent[1] = "Wilson";
        objStudent[2] = "Patrick";
        Console.WriteLine("Student Names");
        Console.WriteLine();
        for (int i = 0; i < 3; i++) {
            Console.WriteLine(objStudent[i]);
        }
    }
}

```

Indexers in Interfaces 3-3

- In the code:
 - 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.

In slide 51, show the code to the students that demonstrates the implementation of interface indexers

Use slide 52 to explain the code. Tell that in the code, 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.

Tell them that this invokes the `set` accessor. The `for` loop displays the output by invoking the `get` accessor of the indexer.

Slide 53

Understand difference between properties and indexers.



Difference between Properties and Indexers

- ◆ Indexers are syntactically similar to properties.
- ◆ However, there are certain differences between them.
- ◆ The following table lists the differences between properties and indexers.

Properties	Indexers
Properties are assigned a unique name in their declaration.	Indexers cannot be assigned a name and use the <code>this</code> keyword in their declaration.
Properties are invoked using the specified name.	Indexers are invoked through an index of the created instance.
Properties can be declared as <code>static</code> .	Indexers can never be declared as <code>static</code> .
Properties are always declared without parameters.	Indexers are declared with at least one parameter.
Properties cannot be overloaded.	Indexers can be overloaded.
Overridden properties are accessed using the syntax <code>base.Prop</code> , where <code>Prop</code> is the name of the property.	Overridden indexers are accessed using the syntax <code>base[indExp]</code> , where <code>indExp</code> is the list of parameters separated by commas.

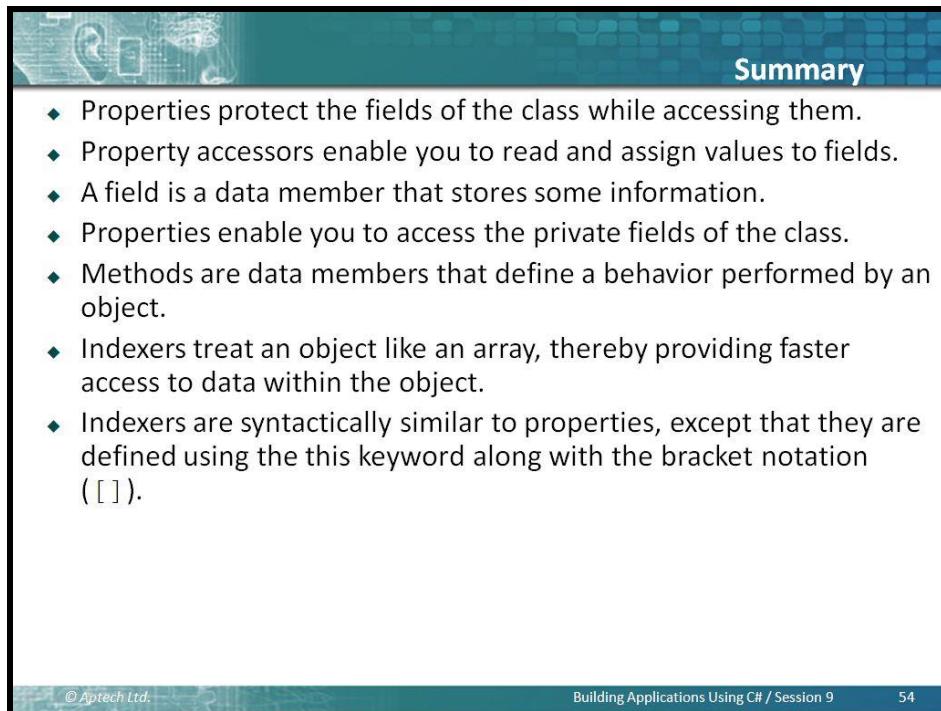
© Aptech Ltd. Building Applications Using C# / Session 9 53

In slide 53, explain the code to the students that indexers are syntactically similar to properties. However, there are certain differences between them.

Explain the table that lists the differences between properties and indexers.

Slide 54

Summarize the session.



The slide features a teal header bar with the word "Summary" in white. Below the header is a white content area containing a bulleted list of nine points about properties and indexers. At the bottom of the slide is a teal footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 9", and the number "54".

- ◆ Properties protect the fields of the class while accessing them.
- ◆ Property accessors enable you to read and assign values to fields.
- ◆ A field is a data member that stores some information.
- ◆ Properties enable you to access the private fields of the class.
- ◆ Methods are data members that define a behavior performed by an object.
- ◆ Indexers treat an object like an array, thereby providing faster access to data within the object.
- ◆ Indexers are syntactically similar to properties, except that they are defined using the `this` keyword along with the bracket notation (`[]`).

In slide 54, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- Properties protect the fields of the class while accessing them.
- Property accessors enable you to read and assign values to fields.
- A field is a data member that stores some information.
- Properties enable you to access the private fields of the class.
- Methods are data members that define a behavior performed by an object.
- Indexers treat an object like an array, thereby providing faster access to data within the object.
- Indexers are syntactically similar to properties, except that, they are defined using the `this` keyword along with the bracket notation (`[]`).

In-Class Questions

4. How can indexers be overloaded?
5. Explain static and non-static methods in C#?
6. List the different kinds of parameters?

9.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a question to students to search additional information, such as:

1. What are the namespaces?
2. What are Exceptions?
3. How to define custom Exceptions?

Session 10 - Namespaces

10.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- Define properties in C#
- Explain properties, fields, and methods
- Explain indexers

Here, you can ask students the key topics they can recall from previous session. Ask them to briefly explain properties in C#. You can also ask them to explain fields, methods, and indexers. Prepare a question or two which will be a key point to relate the current session objectives

10.1.1 Objectives

By the end of this session, the learners will be able to:

- Define and describe namespaces
- Explain nested namespaces

10.1.2 Teaching Skills

To teach this session successfully, you must know about namespaces in C#. You should be aware of nested namespaces.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here during the In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to namespaces. They will learn about nested namespaces.

10.2 In-Class Explanations

Slide 3

Understand namespaces in C#.

The slide has a teal header bar with the title 'Namespaces'. Below the header, there is a large text area containing two code snippets. The first snippet is for the 'Samsung' namespace, which contains a 'Television' class and a 'WalkMan' class. The second snippet is for the 'Sony' namespace, which also contains a 'Television' class and a 'Walkman' class. Both snippets include ellipses (...) between the classes. A blue arrow points from the word 'Namespace' in the text area to the start of the 'Samsung' code block. Another blue arrow points from the same word to the start of the 'Sony' code block.

```

Namespace
namespace Samsung
{
    class Television
    {
        ...
    }
    class WalkMan
    {
        ...
    }
}

Namespace
namespace Sony
{
    class Television
    {
        ...
    }
    class Walkman
    {
        ...
    }
}

```

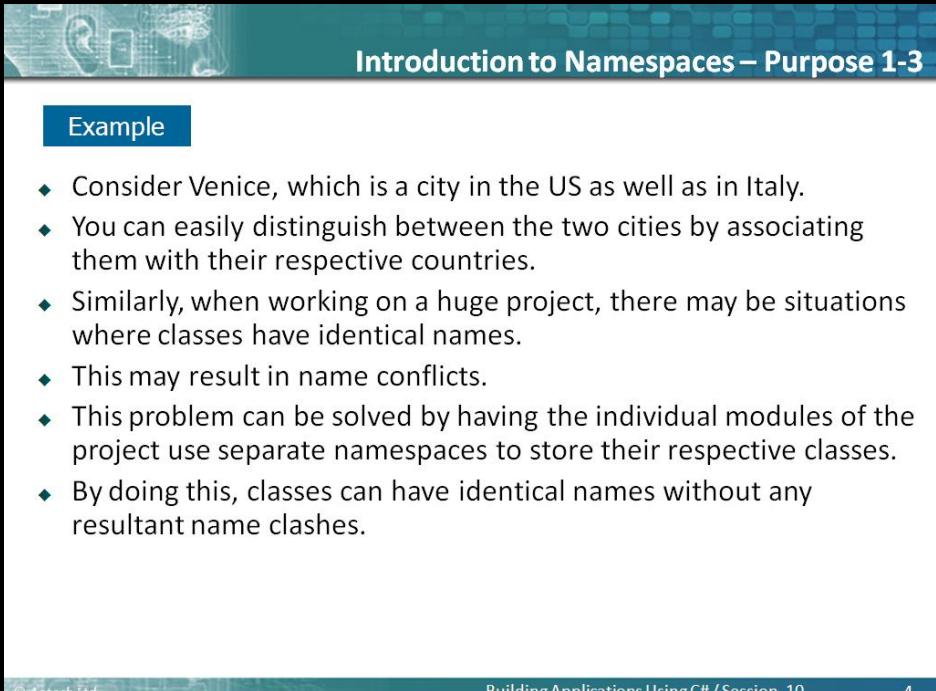
Building Applications Using C# / Session 10 3

Use slide 3 to explain that a namespace in C# is used to group classes logically and prevent name clashes between classes with identical names. Namespaces reduce any complexities when the same class is required in another application.

Tell the students that the classes having similar functionality are put together so that it will be easier for the user to search and identify the required classes through its namespace. Give an example. Tell that classes having SQL functionality are put together in the namespace called System.Data.SqlClient, and classes that are providing functionality for networking are put together in System.Net.

Slides 4 to 6

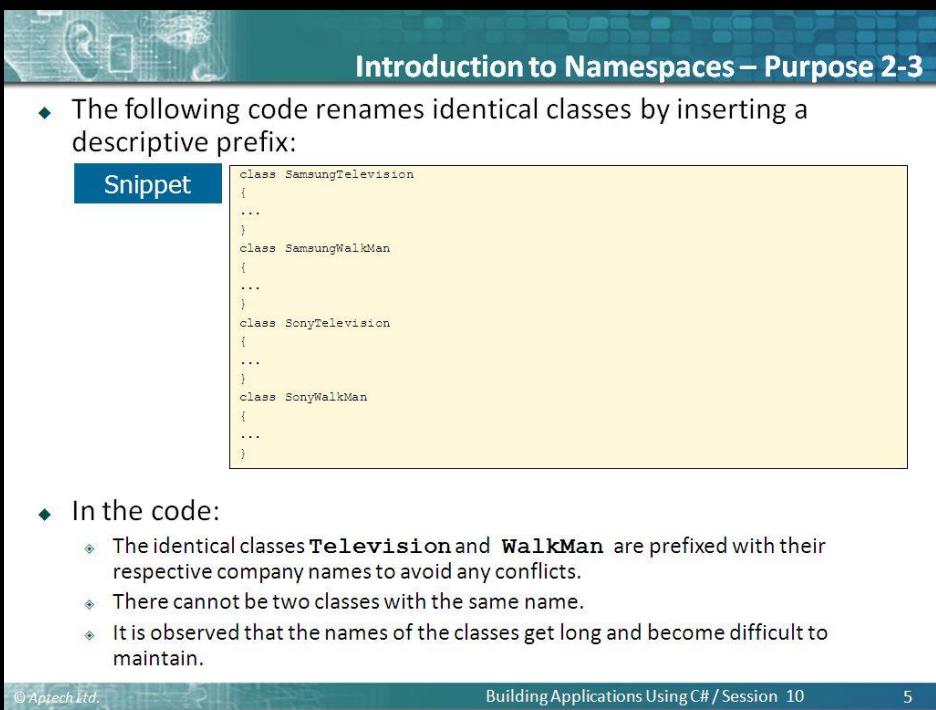
Understand the purpose of namespace.



Introduction to Namespaces – Purpose 1-3

Example

- ◆ Consider Venice, which is a city in the US as well as in Italy.
- ◆ You can easily distinguish between the two cities by associating them with their respective countries.
- ◆ Similarly, when working on a huge project, there may be situations where classes have identical names.
- ◆ This may result in name conflicts.
- ◆ This problem can be solved by having the individual modules of the project use separate namespaces to store their respective classes.
- ◆ By doing this, classes can have identical names without any resultant name clashes.



Introduction to Namespaces – Purpose 2-3

Snippet

```

class SamsungTelevision
{
...
}
class SamsungWalkMan
{
...
}
class SonyTelevision
{
...
}
class SonyWalkMan
{
...
}

```

- ◆ The following code renames identical classes by inserting a descriptive prefix:
- ◆ In the code:
 - ◆ The identical classes **Television** and **WalkMan** are prefixed with their respective company names to avoid any conflicts.
 - ◆ There cannot be two classes with the same name.
 - ◆ It is observed that the names of the classes get long and become difficult to maintain.

Introduction to Namespaces – Purpose 3-3

- The following code demonstrates a solution to overcome this, by using namespaces:

Snippet

```

namespace Samsung
{
    class Television
    {
        ...
    }
    class WalkMan
    {
        ...
    }
}

namespace Sony
{
    class Television
    {
        ...
    }
    class Walkman
    {
        ...
    }
}

```

- In the code:
 - 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.

In slide 4, explain to the students with an example. Tell them to consider Venice, which is a city in the U.S. as well as in Italy. You can easily distinguish between the two cities by associating them with their respective countries.

Explain them that similarly, when working on a huge project, there may be situations where classes have identical names. This may result in name conflicts. This problem can be solved by having the individual modules of the project use separate namespaces to store their respective classes. By doing this, classes can have identical names without any resultant name clashes.

Using slide 5, you will explain the students the code to rename identical classes by inserting a descriptive prefix. Tell them that in the code, 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.

Also mention that when this is done, it is observed that the names of the classes get long and become difficult to maintain. The names of namespaces also indicate which .dll file is included within the current compilation. For example, if we add,

`using System;`

in any program, then the compilation will automatically include a DLL file known as `System.dll`. In this case, all the functionalities of `System.dll` library are automatically added in the current project and it is available to the current application for use.

Use slide 6 to tell the students the code that demonstrates the use of namespaces. Explain the code. Tell the students that in the code, each of the identical classes is placed in their respective namespaces, which denote respective company names. Also tell them that it can be observed that this is a neater, better organized, and more structured way to handle naming conflicts.

Additional Information

Refer the following links for more information on namespaces:

<http://msdn.microsoft.com/en-us/library/0d941h9d.aspx>

<http://www.dotnetperls.com/namespace>

Slide 7

Understand using namespaces.

Using Namespaces

- ◆ C# allows you to specify a unique identifier for each namespace.
- ◆ This identifier helps you to access the classes within the namespace.
- ◆ Apart from classes, the following data structures can be declared in a namespace:

 Interface	<ul style="list-style-type: none"> • An interface is a reference type that contains declarations of the events, indexers, methods, and properties. • Interfaces are inherited by classes and structures and all the declarations are implemented in these classes and structures.
 Structure	<ul style="list-style-type: none"> • A structure is a value type that can hold values of different data types. • It can include fields, methods, constants, constructors, properties, indexers, operators, and other structures.
 Enumeration	<ul style="list-style-type: none"> • An enumeration is a value type that consists of a list of named constants. • This list of named constants is known as the enumerator list.
 Delegate	<ul style="list-style-type: none"> • 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.

© Aptech Ltd. Building Applications Using C# / Session 10 7

In slide 7, tell the students that C# specifies a unique identifier for each namespace that helps to access the classes within the namespace.

Explain that the data structures can be declared in a namespace.

Tell the students that an interface is a reference type that contains declarations of the events, indexers, methods, and properties. Interfaces are inherited by classes and structures and all the declarations are implemented in these classes and structures.

Explain that a structure is a value type that can hold values of different data types. It can include fields, methods, constants, constructors, properties, indexers, operators, and other structures.

Explain that an enumeration is a value type that consists of a list of named constants. This list of named constants is known as the enumerator list. Tell the students that 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.

Slide 8

Understand the characteristics and benefits.

Characteristics and Benefits

- ◆ A namespace groups common and related classes, structures, or interfaces, which support OOP concepts of encapsulation and abstraction.
- ◆ A namespace:
 - ◆ Provides a hierarchical structure that helps to identify the logic for grouping the classes.
 - ◆ Allows you to add more classes, structures, enumerations, delegates, and interfaces once the namespace is declared.
 - ◆ Includes classes with names that are unique within the namespace.
- ◆ A namespace provides the following benefits:
 - ◆ A namespace allows you to use multiple classes with same names by creating them in different namespaces.
 - ◆ It makes the system modular.

© Aptech Ltd. Building Applications Using C# / Session 10 8

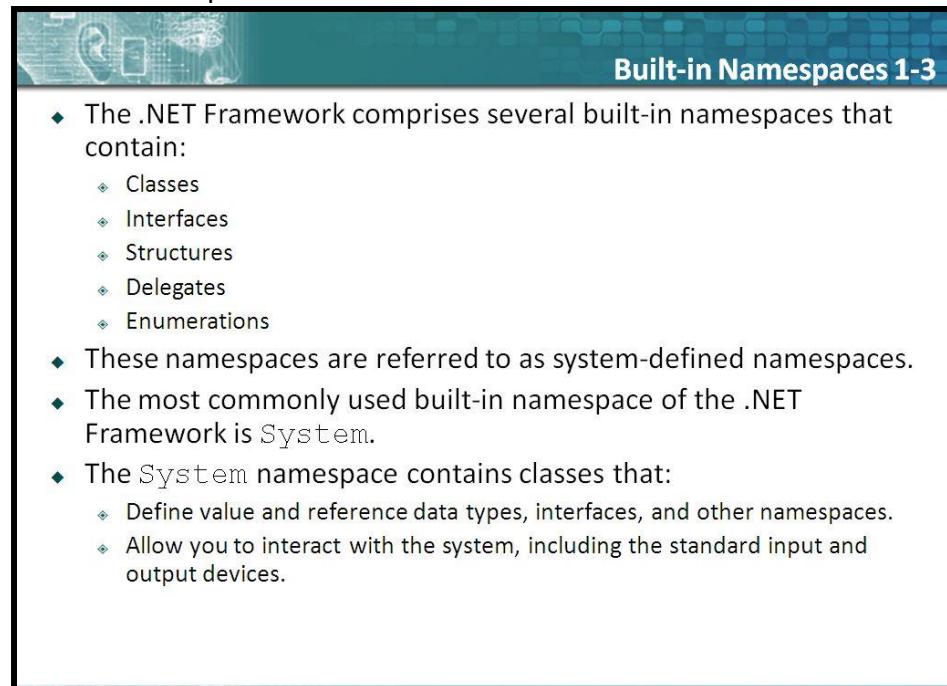
Use slide 8 to explain the students that a namespace groups common and related classes, structures, or interfaces, which support OOP concepts of encapsulation and abstraction.

Tell the students about the characteristics of a namespace. Tell them that it provides a hierarchical structure that helps to identify the logic for grouping the classes. It allows you to add more classes, structures, enumerations, delegates, and interfaces once the namespace is declared. It includes classes with names that are unique within the namespace. Mention the benefits provided by a namespace.

Tell the students that it allows you to use multiple classes with same names by creating them in different namespaces and makes the system modular.

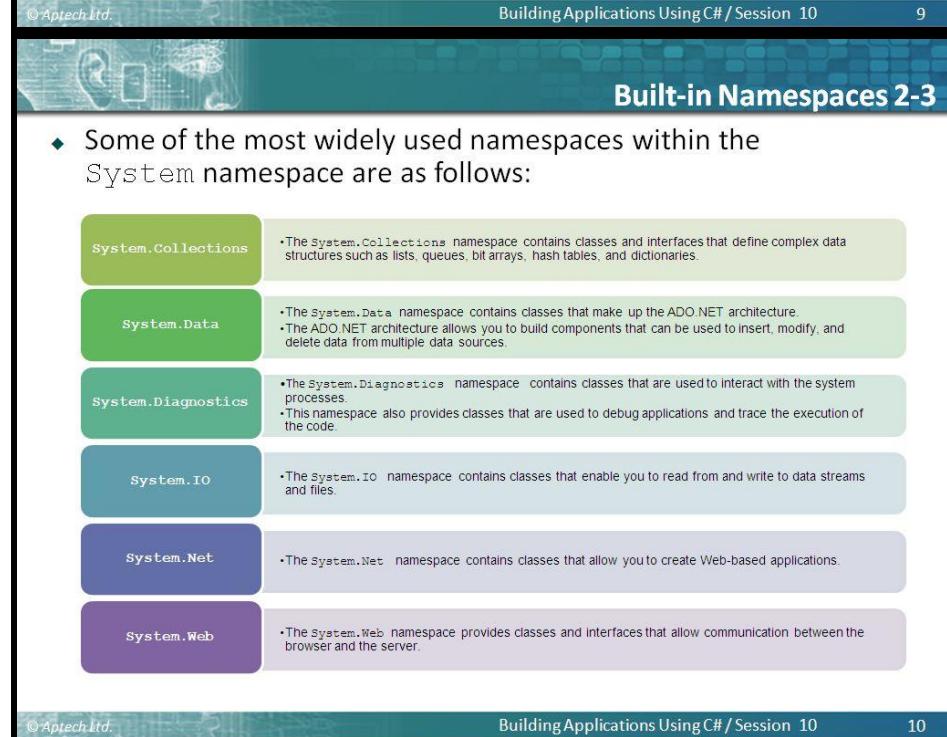
Slides 9 to 11

Understand built-in namespaces.



Built-in Namespaces 1-3

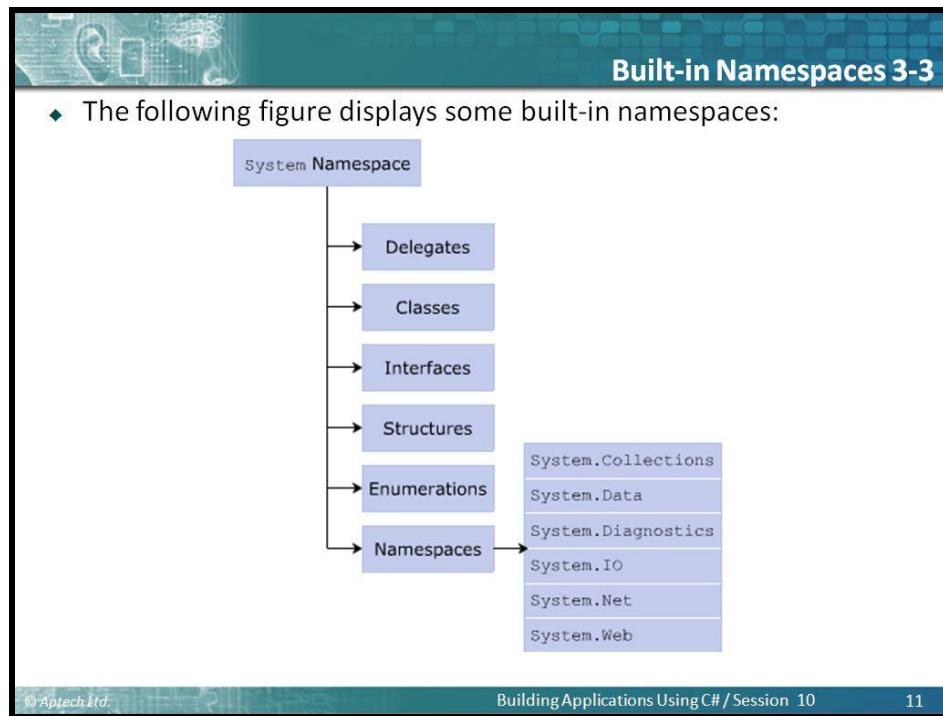
- ◆ The .NET Framework comprises several built-in namespaces that contain:
 - ◆ Classes
 - ◆ Interfaces
 - ◆ Structures
 - ◆ Delegates
 - ◆ Enumerations
- ◆ These namespaces are referred to as system-defined namespaces.
- ◆ The most commonly used built-in namespace of the .NET Framework is `System`.
- ◆ The `System` namespace contains classes that:
 - ◆ Define value and reference data types, interfaces, and other namespaces.
 - ◆ Allow you to interact with the system, including the standard input and output devices.



Built-in Namespaces 2-3

- ◆ Some of the most widely used namespaces within the `System` namespace are as follows:

<code>System.Collections</code>	• The <code>System.Collections</code> namespace contains classes and interfaces that define complex data structures such as lists, queues, bit arrays, hash tables, and dictionaries.
<code>System.Data</code>	• The <code>System.Data</code> namespace contains classes that make up the ADO.NET architecture. • The ADO.NET architecture allows you to build components that can be used to insert, modify, and delete data from multiple data sources.
<code>System.Diagnostics</code>	• The <code>System.Diagnostics</code> namespace contains classes that are used to interact with the system processes. • This namespace also provides classes that are used to debug applications and trace the execution of the code.
<code>System.IO</code>	• The <code>System.IO</code> namespace contains classes that enable you to read from and write to data streams and files.
<code>System.Net</code>	• The <code>System.Net</code> namespace contains classes that allow you to create Web-based applications.
<code>System.Web</code>	• The <code>System.Web</code> namespace provides classes and interfaces that allow communication between the browser and the server.



In slide 9, tell the students that the .NET Framework comprises several built-in namespaces that contain classes, interfaces, structures, delegates, and enumerations.

Explain that these namespaces are referred to as system-defined namespaces. The most commonly used built-in namespace of the .NET Framework is **System**. Mention that the **System** namespace contains classes that define value and reference data types, interfaces, and other namespaces.

Also, mention that it contains classes that allow you to interact with the system, including the standard input and output devices.

In slide 10, explain some of the most widely used namespaces within the **System** namespace. In slide 11, explain that the **System.Collections** namespace contains classes and interfaces that define complex data structures such as lists, queues, bit arrays, hash tables, and dictionaries. Also, tell that the **System.Data** namespace contains classes that make up the ADO.NET architecture. The ADO.NET architecture allows you to build components that can be used to insert, modify, and delete data from multiple data sources.

Explain that the **System.Diagnostic** namespace contains classes that are used to interact with the system processes. This namespace also provides classes that are used to debug applications and trace the execution of the code. Tell the students that the **System.IO** namespace contains classes that enable you to read from and write to data streams and files.

Then tell that the **System.Net** namespace contains classes that allow you to create Web-based applications.

Tell the students that the **System.Web** namespace provides classes and interfaces that allow communication between the browser and the server.

You can refer to the figure in slide 11 to explain some built-in namespaces.

Tell the students that many in-built namespaces are provided to us by Microsoft. These in-built namespaces are automatically installed in our system when the whole .NET framework is installed. Any developer can make use of these in-built libraries or namespaces and develop their own classes.

Additional Information

Refer the following links for more information on the System namespace:

<http://msdn.microsoft.com/en-us/library/system%28v=vs.110%29.aspx>

<http://www.dotnetperls.com/system>

Slide 12

Understand using the System namespace.

Using the System Namespace 1-6

- ◆ The System namespace is imported by default in the .NET Framework.
- ◆ It appears as the first line of the program along with the using keyword.
- ◆ For referring to classes within a built-in namespace, you need to explicitly refer to the required classes.
- ◆ It 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.
- ◆ You can refer to classes within the namespaces in the same manner without the using keyword.
- ◆ However, this results in redundancy because you need to mention the whole declaration every time you refer to the class in the code.

In slide 12, explain 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, 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. Also, refer to classes within the namespaces in the same manner without the using keyword. However, this results in redundancy because the whole declaration needs to be mentioned every time you refer to the class in the code.

Slide 13

Understand the approaches of using the System namespace.

The slide has a teal header bar with the title "Using the System Namespace 2-6". Below the header, there is a bulleted list: "◆ The two approaches of referencing the System namespace are:". Two code snippets are shown in boxes:

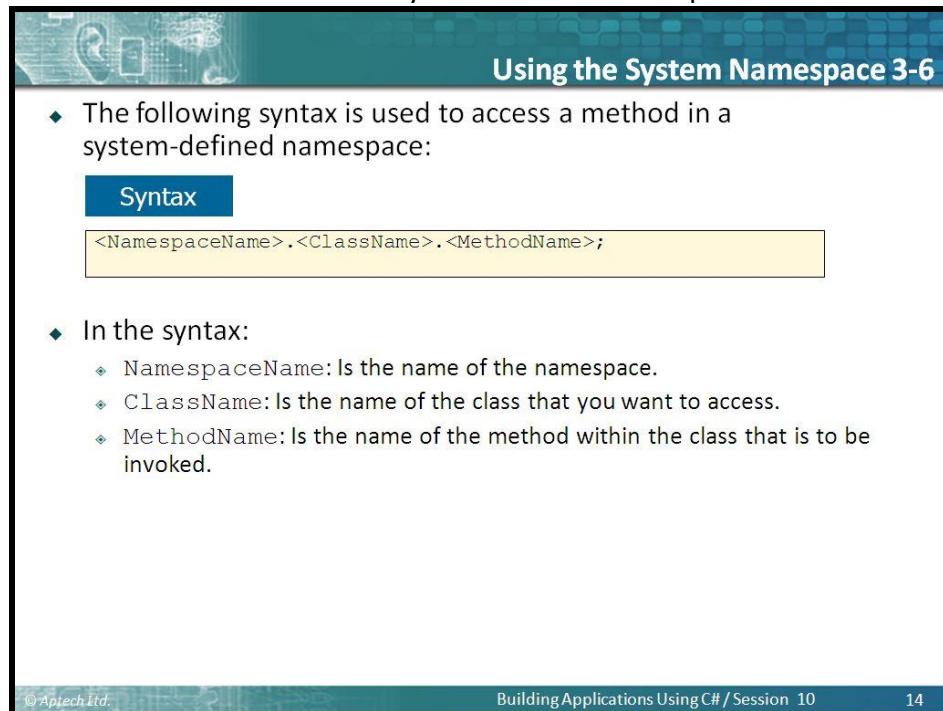
Class 1	Class 2
<pre>using System;</pre>	<pre>System.Console.Read(); System.Console.Read(); ... System.Console.Write();</pre>

Below the boxes, the labels "Efficient Programming" and "Inefficient Programming" are centered under their respective code snippets. At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 10", and "13".

In slide 13, tell the students that there are two approaches of using the System namespace. Explain to them by referring to the figure. Tell them that the two approaches of using the System namespace are, efficient programming and inefficient programming. Tell that though both are technically valid, the first approach is more recommended.

Slide 14

Understand how to access a method in a system-defined namespace.



Using the System Namespace 3-6

- The following syntax is used to access a method in a system-defined namespace:

Syntax

```
<NamespaceName>.<ClassName>.<MethodName>;
```

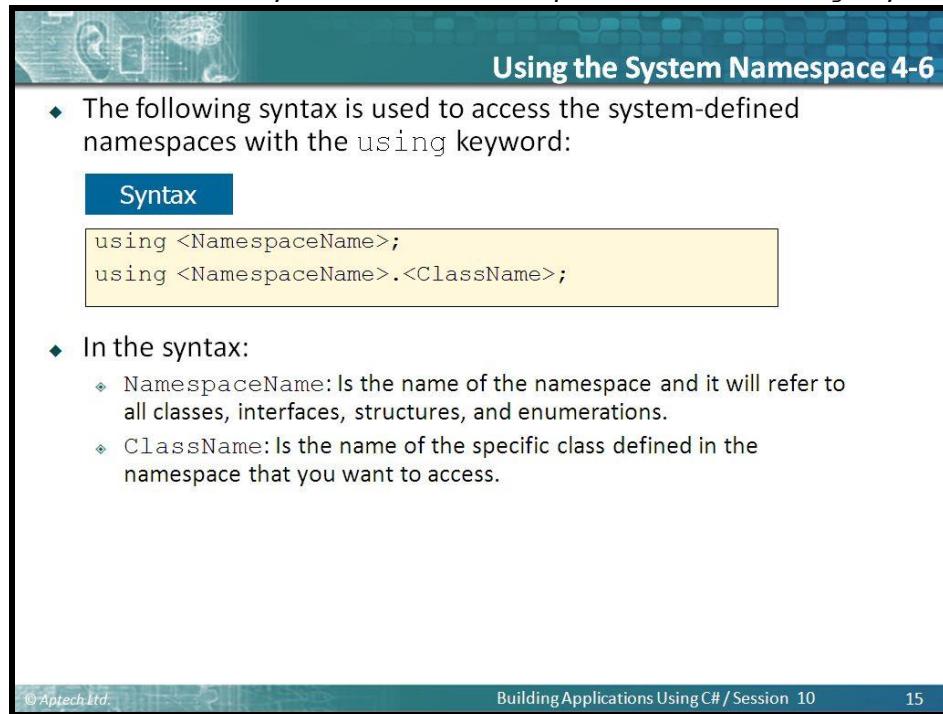
- In the syntax:
 - NamespaceName: Is the name of the namespace.
 - ClassName: Is the name of the class that you want to access.
 - MethodName: Is the name of the method within the class that is to be invoked.

© Aptech Ltd. Building Applications Using C# / Session 10 14

In slide 14, explain the syntax to the students. Tell them that the syntax is used to access a method in a system-defined namespace. Explain the syntax as given on the slide.

Slide 15

Understand how to access the system-defined namespaces with the `using` keyword.



Using the System Namespace 4-6

- The following syntax is used to access the system-defined namespaces with the `using` keyword:

Syntax

```
using <NamespaceName>;
using <NamespaceName>.<ClassName>;
```

- In the syntax:
 - NamespaceName: Is the name of the namespace and it will refer to all classes, interfaces, structures, and enumerations.
 - ClassName: Is the name of the specific class defined in the namespace that you want to access.

© Aptech Ltd. Building Applications Using C# / Session 10 15

Use slide 15 to tell how to access the system-defined namespaces with the `using` keyword.

Then, explain the syntax as given on the slide.

Slide 16

Understand the use of the `using` keyword with namespaces.

The slide has a blue header bar with the title "Using the System Namespace 5-6". Below the title, there is a bulleted list of points:

- ◆ The following code demonstrates the use of the `using` keyword with namespaces:

A "Snippet" box contains the following C# code:

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

Below the snippet, another bulleted list continues:

- ◆ In the code:
 - ◆ 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.

A "Output" box shows the output of the code execution:

```
Hello World
```

At the bottom of the slide, there is footer text: "©Aptech Ltd.", "Building Applications Using C# / Session 10", and "16".

In slide 16, tell the code that demonstrates the use of `using` keyword while using namespaces. Explain the code and the output. Mention that 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.

Slide 17

Understand the `Console` class of the `System` namespace.

Using the System Namespace 6-6

- The following code refers to the `Console` class of the `System` namespace multiple times:

Snippet

```
class World
{
    static void Main(string[] args)
    {
        System.Console.WriteLine("Hello World");
        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#
```

©Aptech Ltd.

Building Applications Using C# / Session 10 17

Use slide 17 to refer to the `Console` class of the `System` namespace multiple times. Then, explain the code and output. Tell them that in the code, the class is not imported but the `System` namespace members are used along with the statements.

Tips:

The `using` keyword is used to globally declare the namespace to be used in the C# file. This feature directly uses the classes within the namespace in the program.

Slides 18 and 19

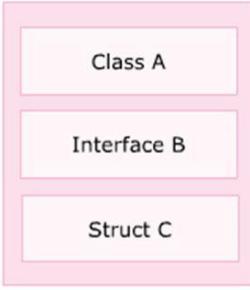
Understand custom namespaces.

Custom Namespaces 1-9

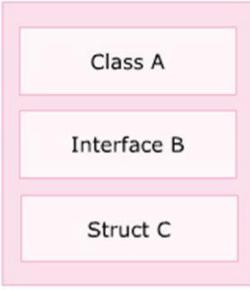
- ◆ C# allows you to create namespaces with appropriate names to organize structures, classes, interfaces, delegates, and enumerations that can be used across different C# applications.
- ◆ When using a custom namespace, you need not worry about name clashes with classes, interfaces, and so on in other namespaces.
- ◆ Custom namespaces:
 - ◆ Enable you to control the scope of a class by deciding the appropriate namespace for the class.
 - ◆ Declared using the `namespace` keyword and is accessed with the `using` keyword similar to any built-in namespace.

Custom Namespaces 2-9

- ◆ The following figure displays a general example of using custom namespaces:



CustomNamespace 1



CustomNamespace 2

© Aptech Ltd.

Building Applications Using C# / Session 10

19

In slide 18, explain to the students that C# creates 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.

Tell them that the custom namespaces control the scope of a class by deciding the appropriate namespace for the class. A custom namespace is declared using the `namespace` keyword and

©Aptech Limited

is accessed with the `using` keyword similar to any built-in namespace. You can refer to the figure in slide 19 to explain a general example of using custom namespaces.

The figure in slide 19 shows an example of custom namespaces that contains class A, interface B, and struct C. Both the namespace `CustomNamespace1` and `CustomNamespace2` contains the class, interface, and struct that have same names. In such cases, if anybody is using these class in a program and if both the namespaces are added. Then, it will be easier and clear to the compiler if the names of the classes, interfaces, and structs are referred by using namespaces.

Give an example as follows:

```
using CustomNamespace1.A;
using CustomNamespace2.A;
```

Additional Information

Refer the following links for more information on custom namespaces:

<http://msdn.microsoft.com/en-us/library/dfb3cx8s.aspx>

<http://msdn.microsoft.com/en-us/library/z2kcy19k.aspx>

Slide 20

Understand how to declare a custom namespace.

Custom Namespaces 3-9

- The following syntax is used to declare a custom namespace:

Syntax

```
namespace <NamespaceName>
{
    //type-declarations;
}
```

- In the syntax:
 - NamespaceName: Is the name of the custom namespace.
 - type-declarations: Are the different types that can be declared. It can be a class, interface, struct, enum, delegate, or another namespace.

©Aptech Ltd. Building Applications Using C# / Session 10 20

Use slide 20 to explain the syntax to declare a custom namespace.

Slide 21

Understand the code that creates a custom namespace named **Department**.

The slide has a teal header bar with the title "Custom Namespaces 4-9". Below the header, there is a bulleted list of points. A blue button labeled "Snippet" is present, and a code block follows. Another blue button labeled "Output" is present, and the output text follows. The footer contains copyright information and page numbers.

◆ The following code creates a custom namespace named **Department**:

Snippet

```
namespace Department
{
    class Sales
    {
        static void Main(string [] args)
        {
            System.Console.WriteLine("You have created a custom namespace
named Department");
        }
    }
}
```

◆ In the code:

- ◆ **Department** is declared as the custom namespace.
- ◆ The class **Sales** is declared within this namespace.

Output

```
You have created a custom namespace named Department
```

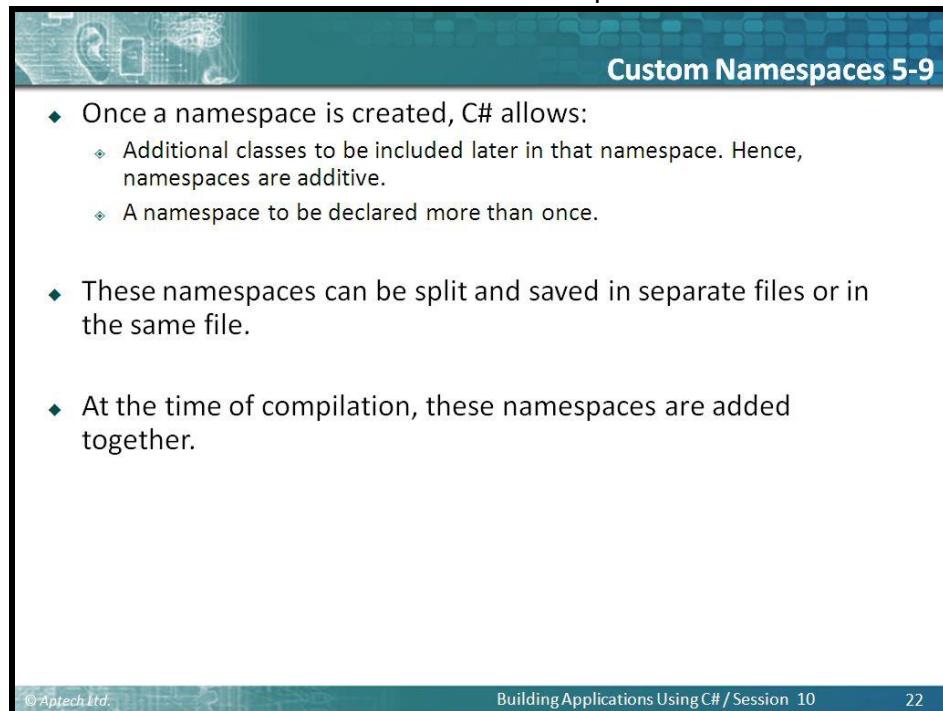
© Aptech Ltd. Building Applications Using C# / Session 10 21

In slide 21, tell the students that the code creates a custom namespace named **Department**. In the code, **Department** is declared as the custom namespace. The class **Sales** is declared within this namespace.

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.

Slide 22

Understand how to include additional classes in a namespace.



The slide has a teal header bar with the title "Custom Namespaces 5-9". The main content area contains a bulleted list of four points. At the bottom of the slide, there is a footer bar with the copyright information "© Aptech Ltd." on the left, the slide title "Building Applications Using C# / Session 10" in the center, and the page number "22" on the right.

- ◆ Once a namespace is created, C# allows:
 - ❖ Additional classes to be included later in that namespace. Hence, namespaces are additive.
 - ❖ 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.

In slide 22, explain that C# allows additional classes to be included later in that namespace. Namespaces are additive.

Tell them that they can be declared more than once.

Now, explain that 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.

Slides 23 to 26

Understand namespace split over multiple files.

Custom Namespaces 6-9

- The namespace split over multiple files is illustrated in the next three code snippets:

Snippet

```
// 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);
        }
    }
}
```

Custom Namespaces 7-9

Snippet

```
//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);
        }
    }
}
```

Custom Namespaces 8-9

Snippet

```
//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();
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 10 25

Custom Namespaces 9-9

- ◆ In the three code snippets:
 - ◆ 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.
 - ◆ 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**.
- ◆ The following figure shows the output of the application containing the three files:

Output

© Aptech Ltd. Building Applications Using C# / Session 10 26

Use slides 23 to 26 to show and explain codes that demonstrate how a namespace can be split over multiple files.

Tell the students that 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.

Mention that 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**.

Slides 27 and 28

Understand the guidelines for creating custom namespaces.

Guidelines for Creating Custom Namespaces 1-2

- ◆ While designing a large framework for a project, it is required to create namespaces to group the types into the appropriate namespaces such that the identical types do not collide.
- ◆ Therefore, the following guidelines must be considered for creating custom namespaces:
 - ◆ All similar elements such as classes and interfaces must be created into a single namespace. This will form a logical grouping of similar types and any programmer will be easily able to search for similar classes.
 - ◆ Creating deep hierarchies that are difficult to browse must be avoided.
 - ◆ Creating too many namespaces must be avoided for simplicity.

Guidelines for Creating Custom Namespaces 2-2

- ◆ Nested namespaces must contain types that depend on the namespace within which it is declared.

Example

- ◆ The classes in the **Country**.**States**.**Cities** namespace will depend on the classes in the namespace **Country**.**States**.

Example

- ◆ If a user has created a class called **US** in the **States** namespace, only the cities residing in U.S. can appear as classes in the **Cities** namespace.

In slide 27, explain that while designing a large framework for a project, it is required to create namespaces to group the types into the appropriate namespaces such that the identical types do not collide.

Explain the following guidelines that must be considered for creating custom namespaces:

- All similar elements such as classes and interfaces must be created into a single namespace. This will form a logical grouping of similar types and any programmer will be easily able to search for similar classes.
- Creating deep hierarchies that are difficult to browse must be avoided.
- Creating too many namespaces must be avoided for simplicity.

In slide 28, explain that the nested namespaces must contain types that depend on the namespace within which it is declared.

Explain the following example to the students. Tell them that the classes in the **Country**.**States**.**Cities** namespace will depend on the classes in the namespace **Country**.**States**.

Give an example. Tell that if a user has created a class called **US** in the **States** namespace, only the cities residing in U.S. can appear as classes in the **Cities** namespace.

Slides 29 and 30

Understand access modifiers for namespaces.

Access Modifiers for Namespaces 1-2

- ◆ Namespaces are always implicitly public.
- ◆ You cannot apply access modifiers such as `public`, `protected`, `private`, or `internal` to namespaces.
- ◆ The namespace is accessible by any other namespace or class that exists outside the accessed namespace.
- ◆ The access scope of a namespace cannot be restricted.

© Aptech Ltd. Building Applications Using C# / Session 10 29



Access Modifiers for Namespaces 2-2

- If any of the access modifiers is specified in the namespace declaration, the compiler generates an error.
- The following code attempts to declare the namespace as public:

Snippet

```
using System;
public namespace Products
{
    class Computers
    {
        static void Main(string [] args)
        {
            Console.WriteLine("This class provides information
about Computers");
        }
    }
}
```

- The code generates the error, 'A namespace declaration cannot have modifiers or attributes'.

In slide 29, tell the students that namespaces are always implicitly public.

Access modifiers such as public, protected, private, or internal cannot be applied to namespaces. This is because a namespace is accessible by any other namespace or class that exists outside the accessed namespace. Therefore, the access scope of a namespace cannot be restricted.

Use slide 30 to explain that if any of the access modifiers is specified in the namespace declaration, the compiler generates an error.

Tell that the code declares the namespace as public and generates the error, 'A namespace declaration cannot have modifiers or attributes'.

Slides 31 and 32

Understand unqualified naming.

Unqualified Naming 1-2

- ◆ A class defined within a namespace is accessed only by its name.
- ◆ To access this class, you just have to specify the name of that class.
- ◆ This form of specifying a class is known as Unqualified naming.
- ◆ The use of unqualified naming results in short names and can be implemented by the `using` keyword.
- ◆ This makes the program simple and readable.

Unqualified Naming 2-2

- ◆ The following code displays the student's name, ID, subject, and marks scored using an unqualified name:

Snippet

```
using System;
using Students;
namespace Students
{
    class StudentDetails
    {
        string _studName = "Alexander";
        int _studID = 30;
        public StudentDetails()
        {
            Console.WriteLine("Student Name: " + _studName);
            Console.WriteLine("Student ID: " + _studID);
        }
    }
}
namespace Examination
{
    class ScoreReport
    {
        public string Subject = "Science";
        public int Marks = 60;
        static void Main(string[] args)
        {
            StudentDetails objStudents = new StudentDetails();
            ScoreReport objReport = new ScoreReport();
            objReport.Subject = "Subject: " + objReport.Subject;
            Console.WriteLine("Marks: " + objReport.Marks);
        }
    }
}
```

- ◆ In the code, the class `ScoreReport` uses the class `StudentDetails` defined in the namespace `Examination`. The class is accessed by its name.

Use slide 31 to explain that a class defined within a namespace is accessed only by its name. Then, tell them that to access this class, you just have to specify the name of that class. This form of specifying a class is known as Unqualified naming.

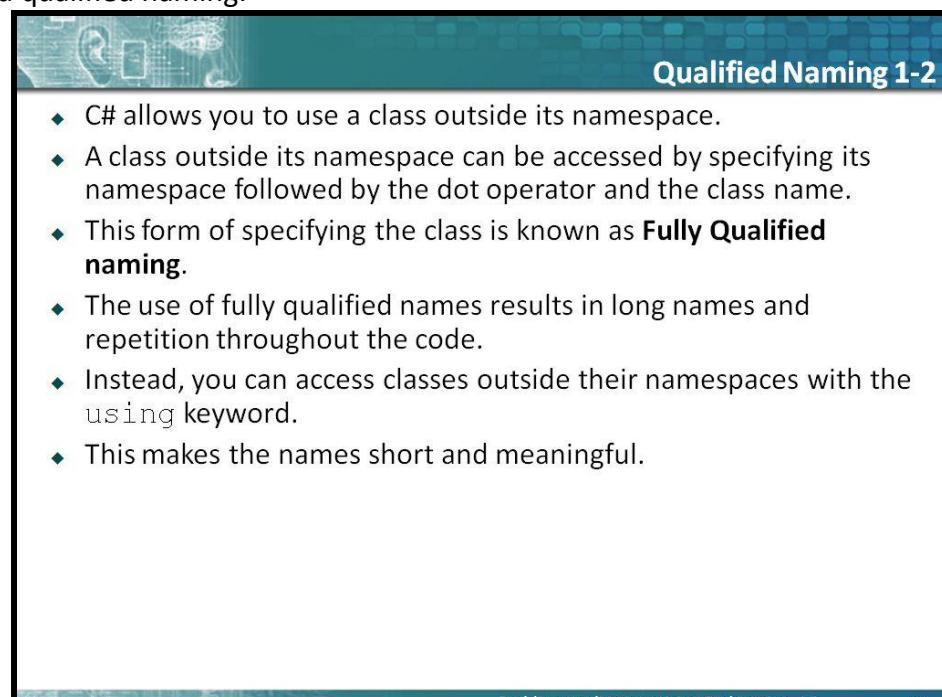
Now, explain the use of unqualified naming results in short names and can be implemented by the `using` keyword. This makes the program simple and readable.

In slide 32, explain that the code displays the student's name, ID, subject, and marks scored using an unqualified name.

Tell that in the code, the class **ScoreReport** uses the class **StudentDetails** defined in the namespace **Examination**. The class is accessed by its name.

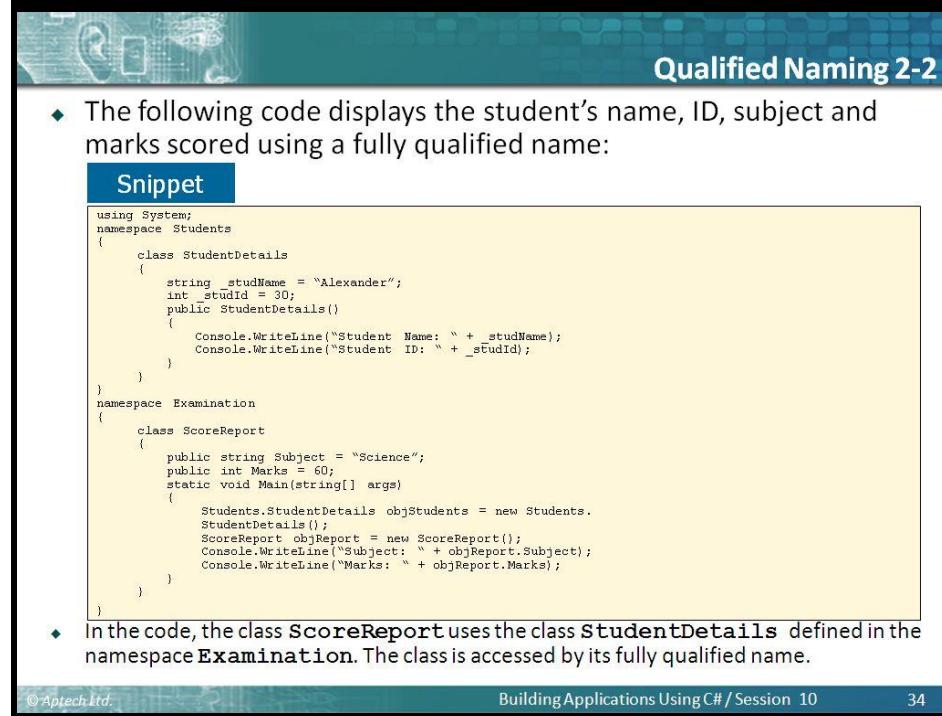
Slides 33 and 34

Understand qualified naming.



Qualified Naming 1-2

- ◆ C# allows you to use a class outside its namespace.
- ◆ A class outside its namespace can be accessed by specifying its namespace followed by the 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, you can access classes outside their namespaces with the `using` keyword.
- ◆ This makes the names short and meaningful.



Qualified Naming 2-2

- ◆ The following code displays the student's name, ID, subject and marks scored using a fully qualified name:

Snippet

```

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

In slide 33, tell the students that C# allows the use of 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**.

Mention that the use of fully qualified names results in long names and repetition throughout the code. Classes can be accessed outside their namespaces with the `using` keyword. This makes the names short and meaningful.

In slide 34, explain to the students that the class `ScoreReport` uses the class `StudentDetails` defined in the namespace `Examination`. The class is accessed by its fully qualified name.

Additional Information

Refer the following link for more information on qualified naming:

<http://msdn.microsoft.com/en-us/library/dfb3cx8s.aspx>

Slide 35

Understand naming conventions for namespaces.

The slide has a blue header bar with the title 'Naming Conventions for Namespaces'. Below the title is a bulleted list of guidelines for naming namespaces. At the bottom of the slide, there is footer text and a page number.

- ◆ When namespaces are being created to handle projects of an organization, it is recommended that namespaces are prefixed with the company name followed by the technology name, the feature, and the design of the brand.
- ◆ Namespaces for projects of an organization can be created as follows:
`CompanyName.TechnologyName [.Feature] [.Design]`
- ◆ Naming conventions that should be followed for creating namespaces are:
 - ◆ Use Pascal case for naming the namespaces.
 - ◆ Use periods to separate the logical components.
 - ◆ Use plural names for namespaces wherever applicable.
 - ◆ Ensure that a namespace and a class do not have same names.
 - ◆ Ensure that the name of a namespace is not identical to the name of the assembly.

©Aptech Ltd. Building Applications Using C# / Session 10 35

In slide 35, tell the students that when namespaces are being created to handle projects of an organization, it is recommended that namespaces are prefixed with the company name followed by the technology name, the feature, and the design of the brand.

Explain that namespaces for projects of an organization can be created as follows:

`CompanyName.TechnologyName [.Feature] [.Design]`

Mention the naming conventions to be followed for creating namespaces. The conventions are use Pascal case for naming the namespaces, use periods to separate the logical components, use plural names for namespaces wherever applicable, ensure that a namespace and a class do

not have same names, and lastly, ensure that the name of a namespace is not identical to the name of the assembly.

With this slide, you will finish explaining namespaces in C#.

Additional Information

Refer the following link for more information on naming conventions for namespaces:

<http://msdn.microsoft.com/en-us/library/893ke618%28v=vs.71%29.aspx>

In-Class Question:

After you finish explaining the namespaces, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which is the most commonly used built-in namespace?

Answer:

The most commonly used built-in namespace of the .NET Framework is System.

Slides 36 and 37

Understand nested namespaces.

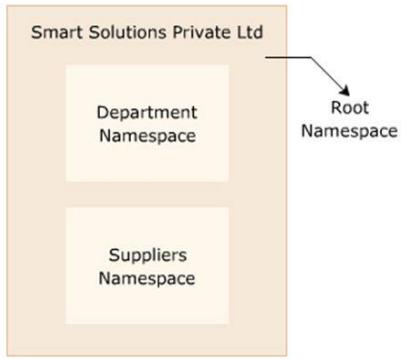

Nested Namespaces 1-2

- ◆ C# allows you to define namespaces within a namespace.
- ◆ This arrangement of namespaces is referred to as nested namespaces.
- ◆ For an organization running multiple projects, nested namespaces is useful.
- ◆ The root namespace can be given the name of the organization and nested namespaces can be given the names of individual projects or modules.
- ◆ This allows the developers to store commonly used classes in appropriate namespaces and use them for all other similar programs.

© Aptech Ltd.
Building Applications Using C# / Session 10
36


Nested Namespaces 2-2

- ◆ The following figure illustrates the concept of nested namespaces:



Nested Namespaces for a Company

© Aptech Ltd.
Building Applications Using C# / Session 10
37

Use slide 36 to explain nested namespace.

Tell that C# defines namespaces within a namespace. This arrangement of namespaces is referred to as nested namespaces.

Also, mention that for an organization running multiple projects, nested namespaces is useful. The root namespace can be given the name of the organization and nested namespaces can be given the names of individual projects or modules.

Tell that this allows the developers to store commonly used classes in appropriate namespaces and use them for all other similar programs.

You can refer to figure in slide 37 that explain the concept of nested namespace.

Additional Information

Refer the following link for more information on nested namespaces:

<http://msdn.microsoft.com/en-us/library/dfb3cx8s.aspx>

Slides 38 to 40

Understand the implementing nested namespaces.

The slide has a blue header bar with the title "Implementing Nested Namespaces 1-3". Below the title is a bulleted list of three items:

- ◆ C# allows you to create a hierarchy of namespaces by creating namespaces within namespaces.
- ◆ Such nesting of namespaces is done by enclosing one namespace declaration inside the declaration of the other namespace.
- ◆ The following syntax can be used to create nested namespaces:

A yellow box labeled "Syntax" contains the following C# code snippet:

```
namespace <NamespaceName>
{
    namespace <NamespaceName>
    {
        ...
    }
    namespace <NamespaceName>
    {
        ...
    }
}
```

At the bottom left of the slide is the copyright notice "© Aptech Ltd.", at the bottom center is "Building Applications Using C# / Session 10", and at the bottom right is the page number "38".

Implementing Nested Namespaces 2-3

- The following code creates nested namespaces:

Snippet

```
namespace Contact
{
    public class Employees
    {
        public int EmpID;
    }
    namespace Salary
    {
        public class SalaryDetails
        {
            public double EmpSalary;
        }
    }
}
```

- In the code:
 - Contact** is declared as a custom namespace that contains the class **Employees** and another namespace **Salary**.
 - The **Salary** namespace contains the class **SalaryDetails**.

Implementing Nested Namespaces 3-3

- The following code displays the salary of an employee using the nested namespace that was created in the previous code:

Snippet

```
using System;
class EmployeeDetails
{
    static void Main(string [] args)
    {
        Contact.Salary.SalaryDetails objSal = new
        Contact.Salary.SalaryDetails();
        objSal.EmpSalary = 1000.50;
        Console.WriteLine("Salary: " + objSal.EmpSalary);
    }
}
```

Output

```
Salary: 1000.5
```

- In the **EmployeeDetails** class of the code:
 - The object of the **SalaryDetails** class is created using the namespaces in which the classes are declared.
 - The value of the variable **EmpSalary** of the **SalaryDetails** class is initialized to **1000.5** and the salary amount is displayed as the output.

In slide 38, explain to the students that the C# allows you to create a hierarchy of namespaces by creating namespaces within namespaces. Such nesting of namespaces is done by enclosing one namespace declaration inside the declaration of the other namespace.

Explain the syntax to create nested namespaces. Use slide 39 to explain how to create nested namespaces. Explain that in the code, **Contact** is declared as a custom namespace that contains the class **Employees** and another namespace **Salary**. The **Salary** namespace contains the class **SalaryDetails**.

In slide 40, explain another code and the output that displays the salary of an employee using the nested namespace.

Mention that in the code, the object of the **SalaryDetails** class is created using the namespaces in which the classes are declared.

The value of the variable **EmpSalary** of the **SalaryDetails** class is initialized to 1000.5 and the salary amount is displayed as the output.

Slides 41 and 42

Understand namespace aliases.

The slide has a blue header bar with the title 'Namespace Aliases 1-5'. The main content area contains a bulleted list of points about namespace aliases:

- ◆ Aliases:
 - ❖ Are temporary alternate names that refer to the same entity.
 - ❖ Are also useful when a program contains many nested namespace declarations and you would like to distinguish as to which class belongs to which namespace.
 - ❖ Would make the code more readable for other programmers and would make it easier to maintain.
- ◆ The namespace referred to with the `using` keyword refers to all the classes within the namespace.
- ◆ However, sometimes you might want to access only one class from a particular namespace.
- ◆ You can use an alias name to refer to the required class and to prevent the use of fully qualified names.

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 10' in the center, and the number '41' on the right.

Namespace Aliases 2-5

- ◆ The following figure displays an example of using namespace aliases:

Referring to the Figures Namespace

```
graph LR; N1[Namespace1] -- "Referring to the Figures Namespace" --> C1[Class1]; N1 <pre>namespace Figures{};</pre> C1 <pre>//Declared an alias for using the Figures namespace.<br/>//Program to find the area</pre>
```

© Aptech Ltd.

Building Applications Using C# / Session 10

42

In slide 41, explain that 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.

Also tell that to access only one class from a particular namespace, refer to the required class and to prevent the use of fully qualified names.

Mention that aliases are also useful when a program contains many nested namespace declarations and you would like to distinguish as to which class belongs to which namespace.

The alias would make the code more readable for other programmers and would make it easier to maintain.

You can refer to the figure in slide 42 to explain an example of using namespace aliases.

Slide 43

Understand creating a namespace alias.

The slide has a blue header bar with the title "Namespace Aliases 3-5". Below the header, there is a bulleted list of points:

- ◆ The following syntax is used for creating a namespace alias:
Syntax

```
using<aliasName> = <NamespaceName>;
```
- ◆ In the code:
 - ◆ aliasName: Is the user-defined name assigned to the namespace.

At the bottom of the slide, there is a footer bar with the text "©Aptech Ltd.", "Building Applications Using C# / Session 10", and the number "43".

In the slide 43, explain the syntax that is used for creating a namespace alias where, aliasName: Is the user-defined name assigned to the namespace.

Slides 44 and 45

Understand how to create an alias for a custom namespace.

Namespace Aliases 4-5

- The following creates a custom namespace called **Bank.Accounts.EmployeeDetails**:

Snippet

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

- The following code displays the name of an employee using the aliases of the **System.Console** and **Bank.Accounts.EmployeeDetails** namespaces:

Snippet

```
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);
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 10 44

Namespace Aliases 5-5

Output

```
Employee Name: Peter
```

- In the code:
 - 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.

© Aptech Ltd. Building Applications Using C# / Session 10 45

Use slide 44 to show the code that creates a custom namespace called **Bank.Accounts.EmployeeDetails**.

Also explain the code and the output that displays the name of an employee using the aliases of the **System.Console** and **Bank.Accounts.EmployeeDetails** namespaces.

Use slide 45 to explain the code and the output.

Tell that 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.

Slides 46 to 48

Understand external aliasing.


External Aliasing 1-3

Example

- ◆ Consider a large organization working on multiple projects.
- ◆ The programmers working on two different projects may use the same class name belonging to the same namespace and store them in different assemblies.
- ◆ The assemblies created can also contain similar method names.
- ◆ When these assemblies are used in a single program and a common method from these assemblies is invoked, compilation error is generated.
- ◆ The compilation error occurs since the same method resides in both the assemblies.
- ◆ This is called an ambiguous reference and it can be resolved by implementing external aliasing.

© Aptech Ltd.
Building Applications Using C# / Session 10
46


External Aliasing 2-3

External aliasing in C#:

- ◆ Allows the users to define assembly qualified namespace aliases.
- ◆ Can be implemented using the **extern** keyword.

◆ This is illustrated in the following code:

Snippet

```

extern alias LibraryOne;
extern alias LibraryTwo;
using System;
class Companies
{
    static void Main (string [] args)
    {
        LibraryOne::Employees.Display();
        LibraryTwo::Employees.Display();
    }
}

```

© Aptech Ltd.
Building Applications Using C# / Session 10
47



External Aliasing 3-3

- ◆ In the code:
 - ◆ The **Companies** class references to two different assemblies in which the **Employees** class is created with the **Display()** method.
 - ◆ The external aliases for the two assemblies are defined as **LibraryOne** and **LibraryTwo**.
 - ◆ During the compilation of this code, the aliases must be mapped to the path of the respective assemblies through the compiler options.
 - ◆ For example, you may write:

```
/reference: LibraryOne =One.dll
/reference: LibraryTwo =two.dll
```

In slide 46, explain the students by giving an example. Tell them to consider a large organization working on multiple projects. The programmers working on two different projects may use the same class name belonging to the same namespace and store them in different assemblies.

Tell that the assemblies created can also contain similar method names. When these assemblies are used in a single program and a common method from these assemblies is invoked, compilation error is generated. Also tell that the compilation error occurs since the same method resides in both the assemblies. This is called an ambiguous reference and it can be resolved by implementing external aliasing.

Understand external aliasing using the `extern` keyword.

In slide 47, tell the students that external aliasing in C# allows the users to define assembly qualified namespace aliases. It can be implemented using the `extern` keyword.

Explain the code to illustrate external aliasing using the `extern` keyword.

Use slide 48 to explain the code.

In the code, the **Companies** class references to two different assemblies in which the **Employees** class is created with the **Display()** method. The external aliases for the two assemblies are defined as **LibraryOne** and **LibraryTwo**. During the compilation of this code, the aliases must be mapped to the path of the respective assemblies through the compiler options.

For example:

```
/reference: LibraryOne =One.dll
/reference: LibraryTwo =two.dll
```

In-Class Question:

Ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are aliases?

Answer:

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.

Slides 49 and 50

Understand namespace alias qualifier.

Namespace Alias Qualifier 1-10

- ◆ There are some situations wherein the alias provided to a namespace matches with the name of an existing namespace.
- ◆ Then, the compiler generates an error while executing the program that references to that namespace.
- ◆ This is illustrated in the following code:

Snippet

```
//The following program is saved in the Automobile.cs file  
// under the Automotive project.  
  
using System;  
using System.Collections.Generic;  
using System.Text;  
using Utility_Vehicle.Car;  
using Utility_Vehicle = Automotive.Vehicle.Jeep;  
  
namespace Automotive
```

The slide title is "Namespace Alias Qualifier 2-10". The code shown is:

```
{  
    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();  
                }  
            }  
        }  
    }  
}
```

At the bottom of the slide, there is footer text: "©Aptech Ltd.", "Building Applications Using C# / Session 10", and "50".

Use slides 49 and 50 to explain the code to illustrate namespace alias qualifier. In slide 49, explain that there are some situations wherein the alias provided to a namespace matches with the name of an existing namespace. Then, the compiler generates an error while executing the program that references to that namespace.

Slides 51 and 52

Understand how to store the program.

Namespace Alias Qualifier 3-10

- Another project is created under the **Automotive** project to store the program shown in the following code:

Snippet

```
//The following program is saved in the Category.cs file under the
//Utility_Vehicle project created under the Automotive project.
using System;
using System.Collections.Generic;
using System.Text;

namespace Utility_Vehicle
{
    namespace Car
    {
        class Category
        {
            string _category;
            public Category()
            {
                _category = "Luxury Vehicle";
            }
            public void Display()
            {
                Console.WriteLine("Car Category: " +
                    _category);
            }
        }
    }
}
```

Namespace Alias Qualifier 4-10

- Both these files are compiled using the following syntax:

Syntax

```
csc /out:<FileName>.exe <FullPath of File1>.cs <Full Path
of File2>.cs
```

- In the syntax:
 - FileName:** Is the name of the single .exe file that will be generated.
 - FullPath of File 1:** Is the complete path where the first file is located.
 - FullPath of File 2:** Is the complete path where the second file is located.

In slide 51, explain a project under the **Automotive** project to store the program.

Use slide 52 to explain that both these files are compiled using the syntax given on the slide.

Slides 53 and 54

Understand the outcome of the compiled code.

Namespace Alias Qualifier 5-10

- The following figure shows the outcome of the compiled code:

Output

©Aptech Ltd. Building Applications Using C# / Session 10 53

Namespace Alias Qualifier 6-10

- 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:

Syntax

```
<LeftOperand> :: <RightOperand>
```

- In the syntax:
 - LeftOperand:** Is a namespace alias, an extern, or a global identifier.
 - RightOperand:** Is the type.

©Aptech Ltd. Building Applications Using C# / Session 10 54

In slide 53, explain the outcome of the compiled code.

Mention that in both the code, the namespaces **Jeep** and **Car** include the class **Category**. An alias **Utility_Vehicle** is provided to the namespace **Automotive.Jeep**. This alias matches with the name of the other namespace in which the namespace **Car** is nested. To

compile both the programs, the `csc` command is used which uses the complete path to refer to the `Category.cs` file. In the `Automobile.cs` file, the alias name `Utility_Vehicle` is the same as the namespace which is being referred by the file. Due to this name conflict, the compiler generates an error.

Use slide 54 to tell them that this problem of ambiguous name references can be resolved by using the namespace alias qualifier.

Explain that 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.

Tell the students that there are chances of having ambiguous names in our program. In such cases we can resolve it using namespace qualifier as given in the slide.

Slides 55 and 56

Understand the correct code.

Namespace Alias Qualifier 7-10

- The correct code for this is given as follows:

Snippet

```
using System;
using System.Collections.Generic;
using System.Text;
using Utility_Vehicle.Car;
using Utility_Vehicle = Automotive.Vehicle.Jeep;

namespace Automotive
{
    namespace Vehicle
    {
        namespace Jeep
        {
            class Category
            {
                string _category;
                public Category()
                {
                    _category = "Multi Utility Vehicle";
                }
                public void Display()
                {
                    Console.WriteLine("Jeep Category: " +
                        _category);
                }
            }
        }
    }
}
```

Namespace Alias Qualifier 8-10

```
class Automobile
{
    static void Main(string[] args)
    {
        Category objCat = new Category();
        objCat.Display();
        Utility_Vehicle::Category objCategory = new
        Utility_Vehicle::Category();
        objCategory.Display();
    }
}
```

Output

```
D:\C#\Automotive\Automotive>cc /out:Auto.exe Automobile.cs D:\C#\Automotive\Utility_Vehicle.cs
Microsoft (R) Visual C++ Compiler version 4.0.30319.17929
Copyright (C) Microsoft Corporation. All rights reserved.

D:\C#\Automotive\Automotive>Auto
Category Category: Multi Utility Vehicle
D:\C#\Automotive\Automotive>
```

- In the code:
 - 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.

In slide 55, tell the correct code and the output to the students.

Use slides 55 and 56 to explain that 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.

Slides 57 and 58

Understand difference between properties and indexers.

Namespace Alias Qualifier 9-10

- ◆ The alias qualifier is used when a user needs to define the class **System** in the custom namespace.
- ◆ C# does not allow users to create a class named **System**.
- ◆ If a class named **System** is created and a constant variable named **Console** is declared within this class, the compiler will generate an error on using the **System.Console** class.
- ◆ This is because C# includes the system defined namespace called **System**.
- ◆ This problem can be resolved by using namespace alias qualifier with its left operand defined as global as shown in the following code:

Snippet

```
using System;
namespace ITCompany
{
    class System
```

©Aptech Ltd.

Building Applications Using C# / Session 10

57

Namespace Alias Qualifier 10-10

```
{
    const string Console = "Console";
    public static string WriteLine()
    {
        return "WriteLine method of my System class";
    }
    static void Main(string[] args)
    {
        global::System.Console.WriteLine (WriteLine());
    }
}
```

Output

WriteLine method of my System class

- ◆ In the code:
 - ◆ The namespace alias qualifier is used in the class **System**.
 - ◆ The left operand of the namespace alias qualifier is specified as global using the **global** keyword.
 - ◆ Therefore, the search for the right operand starts at the global namespace.

©Aptech Ltd.

Building Applications Using C# / Session 10

58

In slide 57, explain that another situation where a namespace alias qualifier is used when a user needs to define the class **System** in the custom namespace. C# does not allow users to create a class named **System**. If a class named **System** is created and a constant variable named **Console** is declared within this class, the compiler will generate an error on using the **System.Console** class. This is because C# includes the system defined namespace called

System. This problem can be resolved by using namespace alias qualifier with its left operand defined as global.

Use slide 58 to explain that in the code, the namespace alias qualifier is used in the class **System**. The left operand of the namespace alias qualifier is specified as global using the **global** keyword. Therefore, the search for the right operand starts at the global namespace.

Slide 59

Summarize the session.

Summary

- ◆ A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names.
- ◆ The System namespace is imported by default in the .NET Framework.
- ◆ Custom namespaces enable you to control the scope of a class by deciding the appropriate namespace for the class.
- ◆ You cannot apply access modifiers such as public, protected, private, or internal to namespaces.
- ◆ A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- ◆ C# supports nested namespaces that allows you to define namespaces within a namespace.
- ◆ External aliasing in C# allows the users to define assembly qualified namespace aliases.

In slide 59, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names.
- The System namespace is imported by default in the .NET Framework.
- Custom namespaces enable you to control the scope of a class by deciding the appropriate namespace for the class.
- You cannot apply access modifiers such as public, protected, private, or internal to namespaces.
- A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- C# supports nested namespaces that allows you to define namespaces within a namespace.
- External aliasing in C# allows the users to define assembly qualified namespace aliases.

10.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information on namespaces and using directive, such as:

1. What are namespace members?
2. How to implement the using directive?
3. How to use alias directive.

Session 11 - Exception Handling

11.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the previous session for a review. Prepare the background knowledge/summary to be discussed with students in the class. The summary of the previous session is as follows:

- A namespace in C# is used to group classes logically and prevent name clashes between classes with identical names.
- The System namespace is imported by default in the .NET Framework.
- Custom namespaces enable you to control the scope of a class by deciding the appropriate namespace for the class.
- You cannot apply access modifiers such as public, protected, private, or internal to namespaces.
- A class outside its namespace can be accessed by specifying its namespace followed by the dot operator and the class name.
- C# supports nested namespaces that allows you to define namespaces within a namespace.
- External aliasing in C# allows the users to define assembly qualified namespace aliases.

Familiarize yourself with the topics of this session in-depth.

11.1.1 Objectives

By the end of this session, the learners will be able to:

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

11.1.2 Teaching Skills

To teach this session successfully, you must know about the exceptions. You should be aware of the basic concepts of throwing and catching exceptions.

You should teach the concepts in the theory class using the concepts, tables, and codes provided.

For teaching in the class, you are expected to use slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced the concept of exceptions. They will learn about the process of throwing and catching exceptions. This session will also discuss the concepts of nested `try`, multiple `catch` blocks, and custom exceptions.

11.2 In-Class Explanations**Slides 3 and 4**

Understand the concept of exceptions and exception handling.

Purpose 1-2

- Exceptions are abnormal events that prevent a certain task from being completed successfully.

©Aptech Ltd.

Building Applications Using C# / Session 11 3

Purpose 2-2

Example

- ◆ Consider a vehicle that halts abruptly due to some problem in the engine.
- ◆ Until this problem is sorted out, the vehicle may not move ahead.
- ◆ Similarly, in C#, exceptions disrupt the normal flow of the program.

Exception occurs

Application terminates abnormally

© Aptech Ltd.

Building Applications Using C# / Session 11

4

Use slide 3 to discuss the purpose of the exceptional handling.

Tell the students that exceptions are run-time errors that may cause a program to be abruptly terminated. Tell them that the exception handling is a process of handling these run-time errors. Handling an exception refers to the action to be taken when an error occurs in order to save the program from being prematurely terminated.

Also tell that, if a user (programmer) do not provide a mechanism to handle these problems, the .NET run time environment provide a default mechanism, which terminates the program execution.

Explain them that the exceptions are abnormal events that prevent a certain task from being completed successfully.

You can refer to the figure in slide 3 to explain the concept of exception.

You can refer to the figure in slide 4 to explain the example.

Tell the students to 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.

Slide 5

Understand exceptions in C#.

The slide has a teal header bar with the title "Exceptions in C#" and decorative icons of a computer monitor, keyboard, and mobile phone. The main content area contains a bulleted list and an illustration. The list describes a scenario where a C# application fails due to lack of permissions and suggests using exception handling to prevent such abrupt termination. Below the list is a cartoon illustration of a man in a white shirt and tie sitting at a wooden desk, looking at a laptop screen. A thought bubble above his head contains the text "Use Exception Handling". The laptop screen shows some code or error messages. At the bottom of the slide, there is footer text: "© Aptech Ltd.", "Building Applications Using C# / Session 11", and the number "5".

- ◆ Consider a C# application that is currently being executed.
- ◆ Assume that at some point of time, the CLR discovers that it does not have the read permission to read a particular file.
- ◆ The CLR immediately stops further processing of the program and terminates its execution abruptly.
- ◆ To avoid this from happening, you can use the exception handling features of C#.

In slide 5, tell the students to consider a C# application that is currently being executed. Tell them to assume that at some point of time, the CLR discovers that it does not have the read permission to read a particular file. The CLR immediately stops further processing of the program and terminates its execution abruptly. Mention that to avoid this from happening, you can use the exception handling features of C#.

Additional Information

Refer the following links for more information on exceptions in C#:

<http://msdn.microsoft.com/en-us/library/ms173160.aspx>

<http://www.codeproject.com/Articles/125470/Exception-Handling-for-C-Beginners>

Slide 6

Understand the types of exceptions in C#.

Types of Exceptions in C#

- ◆ C# can handle different types of exceptions using exception handling statements.
- ◆ It allows you to handle basically two kinds of exceptions. These are as follows:
 - ◆ **System-level Exceptions:** These are the exceptions thrown by the system that are thrown by the CLR.
For example, exceptions thrown due to failure in database connection or network connection are system-level exceptions.
 - ◆ **Application-level Exceptions:** These are thrown by user-created applications.
For example, exceptions thrown due to arithmetic operations or referencing any null object are application-level exceptions.
- ◆ The following figure displays the types of exceptions in C#:

```

graph TD
    EC[Exception Class] --> SSE[System.SystemException]
    EC --> AE[System.ApplicationException]
    SSE --> CLR[Exceptions thrown by the Common-Language Runtime (CLR)]
    AE --> APP[Exceptions thrown by Applications]
  
```

© Aptech Ltd.

Building Applications Using C# / Session 11 6

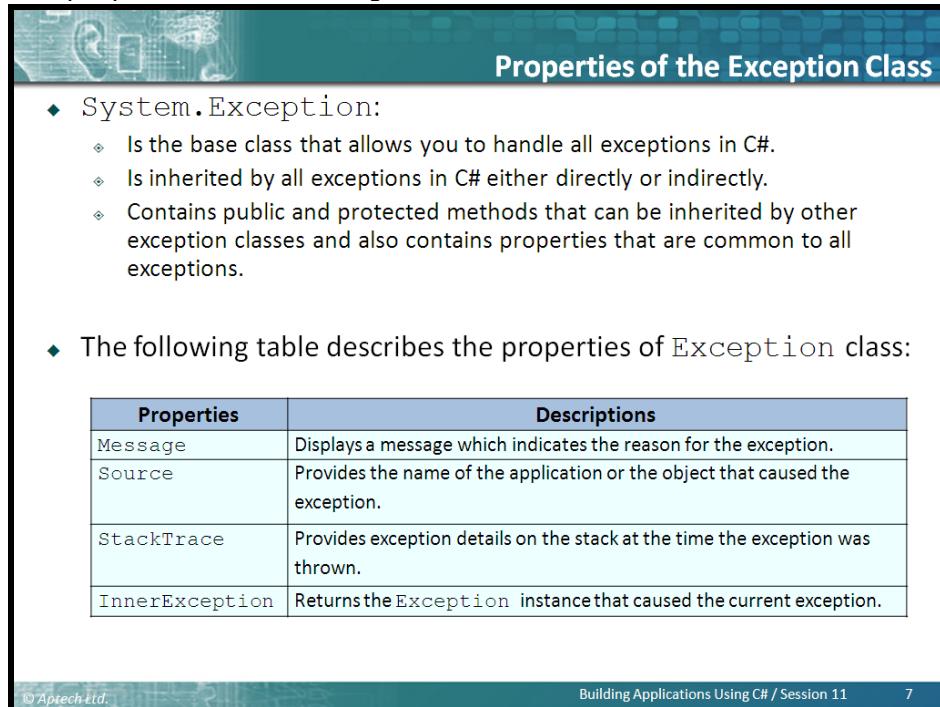
In slide 6, explain the types of exceptions in C# to the students.

Tell the students that C# can handle different types of exceptions using exception handling statements. Exceptions are system defined errors handled by end user with the help of exception handling mechanism. Show figure from slide 6 and tell them that C# allows you to handle basically two kinds of exceptions: System-level Exceptions and Application-level Exceptions. Explain the system-level exceptions by saying that the system-level exceptions are the exceptions thrown by the system. These exceptions are thrown by the CLR. For example, exceptions thrown due to failure in database connection or network connection are system-level exceptions. Explain the application-level exceptions by saying that application-level exceptions are thrown by user-created applications. For example, exceptions thrown due to arithmetic operations or referencing any null object are application-level exceptions.

You can refer to the figure in slide 6 to explain the two kinds of exceptions.

Slide 7

Understand the properties of the Exception class.



Properties of the Exception Class

- ◆ **System.Exception:**
 - ❖ Is the base class that allows you to handle all exceptions in C#.
 - ❖ Is inherited by all exceptions in C# either directly or indirectly.
 - ❖ Contains public and protected methods that can be inherited by other exception classes and also contains properties that are common to all exceptions.
- ◆ The following table describes the properties of Exception class:

Properties	Descriptions
Message	Displays a message which indicates the reason for the exception.
Source	Provides the name of the application or the object that caused the exception.
StackTrace	Provides exception details on the stack at the time the exception was thrown.
InnerException	Returns the <code>Exception</code> instance that caused the current exception.

Use slide 7 to explain properties of Exception class.

Tell the students that the `System.Exception` class is the base class that allows the user to handle all exceptions in C#. This means that all exceptions in C# inherit the `System.Exception` class either directly or indirectly.

Explain that the `System.Exception` class contains public and protected methods that can be inherited by other exception classes. In addition, mention that the `System.Exception` class contains properties that are common to all exceptions.

Then, explain the table that describes the properties of `Exception` class. Explain the properties to the students as shown in the table.

Additional Information

Refer the following link for more information on `Exception` class:

<http://msdn.microsoft.com/en-us/library/system.exception.aspx>

Slide 8

Understand the commonly used Exception classes.

Commonly Used Exception Classes

- ◆ The `System.Exception` class has a number of derived exception classes to handle different types of exceptions.
- ◆ The hierarchy shown in the following figure displays the different exception classes in the `System` namespace:

```

graph TD
    SE[System.Exception] --> SSE[System.SystemException]
    SE --> AE[System.ApplicationException]
    SSE --> AE
    SSE --> SSE1[System.ArgumentException]
    SSE --> SSE2[System.ArithmaticException]
    SSE --> SSE3[System.Data.DataException]
    SSE --> SSE4[System.FormatException]
    SSE --> SSE5[System.IO.IOException]
    SSE --> SSE6[System.IndexOutOfRangeException]
    SSE --> SSE7[System.OutOfMemoryException]
    SSE --> SSE8[System.StackOverflowException]
    SSE --> SSE9[System.NullReferenceException]
    SSE --> SSE10[System.ArrayTypeMismatchException]
    SSE --> SSE11[System.InvalidCastException]
    AE --> ATE[System.Reflection.TargetException]
    ATE --> ANE[System.ArgumentNullException]
    ATE --> DZE[System.DivideByZeroException]
    ATE --> OE[System.OverflowException]
  
```

In slide 8, tell the students that the `System.Exception` class has a number of derived exception classes to handle different types of exceptions.

You can refer to the figure in slide 8 to display the different exception classes in the `System` namespace.

Additional Information

Refer the following link for more information on different exception classes in the `System` namespace:

<http://msdn.microsoft.com/en-us/library/system.exception.aspx>

Slides 9 and 10

Understand the Exception classes.

Exception Classes 1-2	
Exceptions	Descriptions
<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.ArrayTypeMismatchException</code>	This exception is thrown when an attempt is made to store data in an array whose type is incompatible with the type of the array.
<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.
<code>System.InvalidCastException</code>	This exception is thrown when an explicit conversion from the base type or interface type to another type fails.
<code>System.ArgumentNullException</code>	This exception is thrown when a null reference is passed to an argument of a method that does not accept null values.

Exception Classes 2-2	
Exceptions	Descriptions
<code>System.NullReferenceException</code>	This exception is thrown when you try to assign a value to a null object.
<code>System.OutOfMemoryException</code>	This exception is thrown when there is not enough memory to allocate to an object.
<code>System.OverflowException</code>	This exception is thrown when the result of an arithmetic, casting, or conversion operation is too large to be stored in the destination object or variable.
<code>System.StackOverflowException</code>	This exception is thrown when the stack runs out of space due to having too many pending method calls.
<code>System.Data.DataException</code>	This exception is thrown when errors are generated while using the ADO.NET components.
<code>System.FormatException</code>	This exception is thrown when the format of an argument does not match the format of the parameter data type of the invoked method.
<code>System.IO.IOException</code>	This exception is thrown when any I/O error occurs while accessing information using streams, files, and directories.

In slide 9, explain the students that the `System` namespace contains the different exception classes that C# provides. The type of exception to be handled depends on the specified exception class. Then, explain the table that lists some of the commonly used exception classes. Explain the exception classes in the table as follows:

- `System.ArithmetricException` is thrown for problems that occur due to arithmetic or casting and conversion operations.
- `System.ArgumentException` is thrown when one of the arguments does not match the parameter specifications of the invoked method.
- `System.ArrayTypeMismatchException` is thrown when an attempt is made to store data in an array whose type is incompatible with the type of the array.
- `System.DivideByZeroException` is thrown when an attempt is made to divide a numeric value by zero.
- `System.IndexOutOfRangeException` 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 or attempts to access an index of an array that does not exist.
- `System.InvalidCastException` is thrown when an explicit conversion from the base type or interface type to another type fails.
- `System.ArgumentNullException` is thrown when a null reference is passed to an argument of a method that does not accept null values.

In slide 10, explain a table that displays some more exception classes that are most commonly used as follows:

- `System.NullReferenceException` is thrown when you try to assign a value to a null object.
- `System.OutOfMemoryException` is thrown when there is not enough memory to allocate to an object.
- `System.OverflowException` is thrown when the result of an arithmetic, casting or conversion operation is too large to be stored in the destination object or variable.
- `System.StackOverflowException` is thrown when the stack runs out of space due to having too many pending method calls.
- `System.Data.DataException` is thrown when errors are generated while using the ADO.NET components.
- `System.FormatException` is thrown when the format of an argument does not match the format of the parameter data type of the invoked method.
- `System.IO.IOException` is thrown when any I/O error occurs while accessing information using streams, files, and directories.

Slides 11 and 12

Understand `InvalidCastException` class.

InvalidCastException Class 1-2

- ◆ The `InvalidCastException` exception is thrown when an explicit conversion from a base type to another type fails.
- ◆ The following code demonstrates the `InvalidCastException` exception:

```
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);
        }
    }
}
```

Snippet

© Aptech Ltd.

Building Applications Using C# / Session 11 11

InvalidCastException Class 2-2

- ◆ In the code:
 - ◆ 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#.
- ◆ The figure displays the exception that is generated when the program is executed.

C:\WINDOWS\system32\cmd.exe

```
Message : Specified cast is not valid.
Error : System.InvalidCastException: Specified cast is not valid.
       at Project.InvalidCastError.Main(String[] args) in D:\Source Code\Project\Project\InvalidCastError.cs:line 16
Press any key to continue . . .
```

© Aptech Ltd.

Building Applications Using C# / Session 11 12

In slide 11, tell the students that the `InvalidCastException` exception is thrown when an explicit conversion from a base type to another type fails.

`InvalidCastException` is generated by the runtime when a statement tries to cast one reference type to a reference type that is not compatible.

Then, explain the code that demonstrates the `InvalidCastException` exception to the students.

Use slide 12 to explain the code as follows to the students:

Tell that in the code, 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#.

You can refer to the figure in slide 12 to explain the output of the code and tell the students that when they execute the program, it displays the exception is generated.

Additional Information

Refer the following link for more information on `InvalidCastException`:

<http://msdn.microsoft.com/en-us/library/system.invalidcastexception.aspx>

Slides 13 and 14

Understand `ArrayTypeMismatchException` class.

ArrayTypeMismatchException Class 1-2

- ◆ The `ArrayTypeMismatchException` exception is thrown when the data type of the value being stored is incompatible with the data type of the array.
- ◆ The following code demonstrates the `ArrayTypeMismatchException` exception:

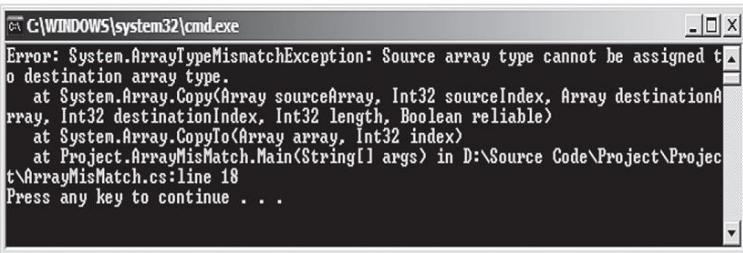
Snippet

```
using System;
class ArrayMisMatch
{
    static void Main(string[] args)
    {
        string[] names = { "James", "Jack", "Peter" };
        int[] id = { 10, 11, 12 };
        double[] salary = { 1000, 2000, 3000 };
        float[] bonus = new float[3];

        try
        {
            salary.CopyTo(bonus, 0);
        }
        catch (ArrayTypeMismatchException objType)
        {
            Console.WriteLine("Error: " + objType);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error: " + objEx);
        }
    }
}
```

ArrayTypeMismatchException Class 2-2

- ◆ In the code:
 - ◆ The **salary** array is of double data type and the **bonus** array is of float data type.
 - ◆ When copying the values stored in **salary** array to **bonus** array using the **CopyTo()** method, data type mismatch occurs.
- ◆ The following table displays the exception that is generated when the program is executed:



```
C:\WINDOWS\system32\cmd.exe
Error: System.ArrayTypeMismatchException: Source array type cannot be assigned to destination array type.
  at System.Array.CopyTo(Array sourceArray, Int32 sourceIndex, Array destinationA
rray, Int32 destinationIndex, Int32 length, Boolean reliable)
  at System.Array.CopyTo(Array array, Int32 index)
  at Project.ArrayMisMatch.Main(String[] args) in D:\Source Code\Project\Projec
t\ArrayMisMatch.cs:line 18
Press any key to continue . . .
```

In slide 13, tell the students that the **ArrayTypeMismatchException** exception is thrown when the data type of the value being stored is incompatible with the data type of the array, i.e. exception will be thrown when trying to add an element of wrong type in an array.

Explain the code demonstrates the **ArrayTypeMismatchException** exception.

Use slide 14 to tell the students that in the code, the **salary** array is of **double** data type and the **bonus** array is of **float** data type. When copying the values stored in **salary** array to **bonus** array using the **CopyTo()** method, data type mismatch occurs.

Then, you can refer to the figure in slide 14 to explain the output of the code that displays the exception that is generated when the program is executed.

Additional Information

Refer the following links for more information on **ArrayTypeMismatchException**:

<http://msdn.microsoft.com/en-us/library/system.arraytypemismatchexception%28v=vs.110%29.aspx>

<https://coderwall.com/p/rsxulw>

Slides 15 and 16

Understand NullReferenceException class.

NullReferenceException Class 1-2

- The `NullReferenceException` exception is thrown when an attempt is made to operate on a null reference as shown in the following code:

```
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);
        }
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 11

15

NullReferenceException Class 2-2

- In the code:
 - A class `Employee` is defined which contains the details of an employee. Two instances of class `Employee` are created with the second instance referencing the first.
 - The first instance is de-referenced using `null`. If the first instance is used to print the values of the `Employee` class, a `NullReferenceException` exception is thrown, which is caught by the catch block.
- The figure displays the exception that is generated when the program is executed.

```
C:\WINDOWS\system32\cmd.exe
Error: System.NullReferenceException: Object reference not set to an instance of
an object.
  at Project.Employee.Main<String[] args> in D:\Source Code\Project\Project\Emp
loyee.cs:line 26
Press any key to continue . . .
```

© Aptech Ltd.

Building Applications Using C# / Session 11

16

In slide 15, tell the students that the `NullReferenceException` exception is thrown when an attempt is made to operate on a null reference.

Explain the code that demonstrates the `NullReferenceException` exception to the students.

Use slide 16 to tell the students that in the code, a class **Employee** is defined which contains the details of an employee. Two instances of class **Employee** are created with the second instance referencing the first. The first instance is de-referenced using `null`. If the first instance is used to print the values of the **Employee** class, a `NullReferenceException` exception is thrown, which is caught by the `catch` block.

Then, after explaining the code, refer to the figure in slide 16 to show the output of the code that displays the exception that is generated when the program is executed.

Additional Information

Refer the following links for more information on exceptions in C#:

<http://msdn.microsoft.com/en-us/library/system.nullreferenceexception.aspx>

<http://codebetter.com/raymondlewallen/2005/06/23/system-nullreferenceexception-object-reference-not-set-to-an-instance-of-an-object-3-common-causes-in-vb-net/>

Slide 17

Understand `System.Exception` class.

The slide has a blue header bar with the title "System.Exception Class 1-5". Below the title is a bulleted list:

- `System.Exception` is the base class that handles all exceptions and contains public and protected methods that can be inherited by other exception classes.
- The following table lists the public methods of the `System.Exception` class and their corresponding description:

Method	Description
<code>Equals</code>	Determines whether objects are equal
<code>GetBaseException</code>	Returns a type of Exception class when overridden in a derived class
<code>GetHashCode</code>	Returns a hash function for a particular type
<code>GetObjectData</code>	Stores information about serializing or deserializing a particular object with information about the exception when overridden in the inheriting class
<code>GetType</code>	Retrieves the type of the current instance
<code>ToString</code>	Returns a string representation of the thrown exception

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 11", and "17".

In slide 17, tell the students, `System.Exception` is the base class that handles all exceptions that occurs during program execution. It contains public and protected methods that can be inherited by other exception classes.

Tell that when an exception is thrown, it is handled by the executing application or provided exception handler.

Explain the table that lists the public methods of the `System.Exception` class and their corresponding description.

Explain the table as given on the slide.

Slide 18

Understand public methods of the System.Exception class.

System.Exception Class 2-5

- The following code demonstrates some public methods of the System.Exception class:

Snippet

```
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 the code:
 - 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.

In slide 18, explain the code that demonstrates some public methods of the System.Exception class to the students. Then, explain the code.

Tell the students that in the code, 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.

Slide 19

Understand the protected methods and the commonly used public properties in the class.

System.Exception Class 3-5

- The following table lists the protected methods of the class and their corresponding descriptions:

Name	Description
Finalize	Allows objects to perform cleanup operations before they are reclaimed by the garbage collector
MemberwiseClone	Creates a copy of the current object

- In the following table lists the commonly used public properties in the class:

Public Property	Description
HelpLink	Retrieves or sets a link to the help file associated with the exception
InnerException	Retrieves the reference of the object of type Exception that caused the thrown exception
Message	Retrieves the description of the current exception
Source	Retrieves or sets the name of the application or the object that resulted in the error
StackTrace	Retrieves a string of the frames on the stack when an exception is thrown
TargetSite	Retrieves the method that throws the exception

In slide 19, explain a table that lists the protected methods of the System.Exception class, Finalize method allows objects to perform cleanup operations before they are reclaimed by the garbage collector.

- MemberwiseClone method creates a copy of the current object.

Then, explain the table that lists the commonly used public properties in the class,

- HelpLink property retrieves or sets a link to the help file associated with the exception.
- InnerException property retrieves the reference of the object of type Exception that caused the thrown exception.
- Message property retrieves the description of the current exception.
- Source property retrieves or sets the name of the application or the object that resulted in the error.
- StackTrace property retrieves a string of the frames on the stack when an exception is thrown.
- TargetSite property retrieves the method that throws the exception.

Slide 20

Understand the code demonstrating public properties of the `System.Exception` class.

The slide has a blue header bar with the title "System.Exception Class 4-5". Below the title is a bulleted list: "The code demonstrates the use of some of the public properties of the System.Exception class." A blue button labeled "Snippet" is positioned to the left of the code block. The code itself is a C# program that multiplies two byte variables and handles an overflow exception by printing its properties.

```
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);
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 11 20

In slide 20, explain the code that demonstrates the use of some of the public properties of the `System.Exception` class.

Slide 21

Understand the code and its output.

The slide has a teal header bar with the title "System.Exception Class 5-5". Below the title is a list of bullet points:

- ◆ In the code:
 - ◆ 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.
- ◆ The following figure displays use of some of the public properties of the `System.Exception` class:

Below the list is a screenshot of a Windows command-line window (cmd.exe) showing the output of an exception. The window title is "C:\WINDOWS\system32\cmd.exe". The output text is:

```
Message : Arithmetic operation resulted in an overflow.
Source  : Project
TargetSite : Void Main(System.String[])
StackTrace : at Project.Exception_Handling.ExceptionProperties.Main()
args> in D:\Source Code\Project\Project\Exception Handling\ExceptionP
s:line 17
Press any key to continue . . .
```

In slide 21, tell the students that in the code, 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.

Then, refer to the figure in slide 21 to explain the output that displays use of some of the public properties of the `System.Exception` class.

In-Class Question:

After you finish explaining the `System.Exception` Class, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which are the two types of exceptions can handled by C#?

Answer:

The system-level exceptions and the application-level exceptions are the two types of exceptions that can be handled by C#.

Slide 22

Understand the concepts of throwing and catching exceptions.

Throwing and Catching Exceptions

- ◆ An exception is an error that occurs during program execution.
- ◆ An exception arises when an operation cannot be completed normally. In such situations, the system throws an error.

The diagram illustrates the concept of exceptions. It shows a road labeled "C# Code" leading to a T-junction. One path from the junction is labeled "No Exception" and leads to a straight road. The other path is labeled "Exception occurs" and leads to a dead end where a car is shown crashing, with an arrow pointing back towards the junction labeled "or terminates".

- ◆ The error is handled through the process of **exception handling**.

© Aptech Ltd. Building Applications Using C# / Session 11 22

In slide 22, explain the students that an exception is an error that occurs during program execution.

You can refer to the figure in slide 22 to explain the concepts.

Tell them that an exception arises when an operation cannot be completed normally. Mention that in such situations, the system throws an error. The error is handled through the process of exception handling.

Additional Information

Refer the following links for more information on exceptions in C#:

<http://msdn.microsoft.com/en-us/library/ms173160.aspx>

<http://www.codeproject.com/Articles/125470/Exception-Handling-for-C-Beginners>

Slide 23

Understand the purpose of exception handlers.

Purpose of Exception Handlers

Example

- ◆ Consider a group of boys playing throwball.
- ◆ If any of the boys fail to catch the ball when thrown, the game is terminated.
- ◆ Thus, the game goes on till the boys are successful in catching the ball on time.
- ◆ Similarly, in C#, exceptions that occur while working on a particular program need to be caught by exception handlers.
- ◆ If the program does not contain the appropriate exception handler, then the program might be terminated.



© Aptech Ltd. Building Applications Using C# / Session 11 23

In slide 23, refer to figure on the slide to explain the concept through the example.

Tell the students to consider a group of boys playing throw ball wherein one boy throws a ball and the other boys catch the ball. Tell them that 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.

Explain them that similarly, in C#, exceptions that occur while working on a particular program need to be caught by exception handlers. Mention that if the program does not contain the appropriate exception handler, then the program might be terminated.

Slides 24 to 26

Understand catching exceptions.

Catching Exceptions 1-3

- Exception handling is implemented using the `try-catch` construct in C# that consists of the following two blocks:

The try block

- It encloses statements that might generate exceptions.
- When these exceptions are thrown, the required actions are performed using the `catch` block.

The catch block

- It consists of the appropriate error-handlers that handle exceptions.
- If the `catch` block does not contain any parameter, it can catch any type of exception.
- The `catch` block follows the `try` block and may or may not contain parameters.
- If the `catch` block contains a parameter, it catches the type of exception specified by the parameter.

© Aptech Ltd.

Building Applications Using C# / Session 11 24

Catching Exceptions 2-3

- The following syntax is used for handling errors using the `try` and `catch` blocks:

Syntax
<pre>try { // program code } catch[(<ExceptionClass><objException>)] { // error handling code }</pre>

- where,
 - try: Specifies that the block encloses statements that may throw exceptions.
 - program code: Are statements that may generate exceptions.
 - catch: Specifies that the block encloses statements that catch exceptions and carry out the appropriate actions.
 - ExceptionClass: Is the name of exception class. It is optional.
 - objException: Is an instance of the particular exception class. It can be omitted if the ExceptionClass is omitted.

© Aptech Ltd.

Building Applications Using C# / Session 11 25

Catching Exceptions 3-3

◆ The code demonstrates the use of `try-catch` blocks.

```
using System;
class DivisionError
{
    static void Main(string[] args)
    {
        int numOne = 133;
        int numTwo = 0;
        int result;
        try
        {
            result = numOne / numTwo;
        }
        catch (DivideByZeroException objDivide)
        {
            Console.WriteLine("Exception caught: " + objDivide);
        }
    }
}
```

◆ In the code:

- ◆ The `Main()` method of the class `DivisionError` declares three variables.
- ◆ 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)
```

© Aptech Ltd. Building Applications Using C# / Session 11 26

In slide 24, explain that the exception handling is implemented using the `try-catch` construct in C#. This construct consists of two blocks, the `try` block and the `catch` block. Tell them that 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. Tell the students that the `catch` block follows the `try` block and may or may not contain parameters. Tell them that if the `catch` block does not contain any parameter, it can catch any type of exception. Mention that if the `catch` block contains a parameter, it catches the type of exception specified by the parameter.

In slide 25, explain the syntax used for handling errors using the `try` and `catch` blocks to the students and explain it to them.

The point in a program at which an exception occurs is called a throw point. If an exception occurs in a `try` block, the `try` block expires (terminates immediately) and program control transfers to the first catch handler following the `try` block. The CLR searches for the first catch handler that can process the type of exception that occurred. The CLR locates the matching catch by comparing the thrown exception's type to each catch's exception-parameter type until the CLR finds a match. A match occurs if the types are identical or if the thrown exception's type is a derived class of the exception-parameter type.

In slide 26, explain the code that demonstrates the use of `try-catch` blocks to the students. Then, explain the code.

Tell the students that in the code, the `Main()` method of the class `DivisionError` declares three variables, two of which are initialized. Tell them that the `try` block contains the code to divide `numOne` by `numTwo` and store the output in the `result` variable. Mention that 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.

Then, explain the output of the code to the students.

The catch block is only executed if the try block throws an exception.

Additional Information

Refer the following links for more information on catching exceptions in C#:

<http://msdn.microsoft.com/en-us/library/xtd0s8kd%28v=vs.110%29.aspx>

Slides 27 and 28

Explain general catch blocks.

General catch Block 1-2

- Following are the features of a general catch block:
 - It can handle all types of exceptions.
 - You can create a catch block with the base class Exception that are referred to as general catch blocks.
 - A general catch block can handle all types of exceptions.

However, the type of exception that the catch block handles depends on the specified exception class.

However, one disadvantage of the general catch block is that there is no instance of the exception and thus, you cannot know what appropriate action must be performed for handling the exception.

© Aptech Ltd. Building Applications Using C# / Session 11 27

General catch Block 2-2

- The following code demonstrates the way in which a general catch block is declared:

Snippet

```

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 the code:
 - A string array called `names` is initialized. 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 declared with `Exception` and this catch block can handle any type of exception.

Output

```

Error: System.IndexOutOfRangeException: Index was outside the bounds of the
array at
Project _New.Exception_Handling.Students.Main(String[] args) in
D:\Exception Handling\Students.cs:line 17

```

In slide 27, explain the features of a general catch block to the students.

Tell the students that 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. Tell them that sometimes they might not know the type of exception the program might throw. In such a case, the students can create a catch block with the base class `Exception`. Such catch blocks are referred to as general catch blocks.

Explain the students that 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, they cannot know what appropriate action must be performed for handling the exception.

In slide 28, explain the code that demonstrates the way in which a general catch block is declared to the students. Then, explain the code.

Tell the students that in the code, 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.

Explain them that the array can store only three values, so this will cause an exception. Mention that 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.

Then explain the output of the code to the students.

Slides 29 to 31

Understand the `throw` statement.

The throw Statement 1-3

- The `throw` statement in C# allows you to programmatically throw exceptions using the `throw` keyword.
- When you `throw` an exception using the `throw` keyword, the exception is handled by the `catch` block as shown in the following syntax:

Syntax

```
throw exceptionObject;
```

- where,
 - `throw`: Specifies that the exception is thrown programmatically.
 - `exceptionObject`: Is an instance of a particular exception class.

The throw Statement 2-3

- The following code demonstrates the use of the `throw` statement:

Snippet

```
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 the code:
 - The class `Employee` declares a `static` method called `ThrowException` that takes a `string` parameter called `name`.
 - If the value of `name` is `null`, the C# compiler throws the exception, which is caught by the instance of the `ArgumentNullException` class.
 - In the `try` block, the value of `name` is `null` and the control of the program goes back to the method `ThrowException`, where the exception is thrown for referencing a null value.
 - The `catch` block consists of the error handler, which is executed when the exception is thrown.

The throw Statement 3-3

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
```

©Aptech Ltd.

Building Applications Using C# / Session 11 31

In slide 29, introduce the `throw` statement to the students.

Tell the students that the `throw` statement in C# allows to programmatically throw exceptions using the `throw` keyword. Tell them that the `throw` statement takes an instance of the particular exception class as a parameter. However, mention that if the instance does not refer to the valid exception class, the C# compiler generates an error.

Tell the students that when they throw an exception using the `throw` keyword, the exception is handled by the `catch` block.

Then, explain a syntax that is used to throw an exception programmatically and tell the students that in the syntax, `throw` specifies that the exception is thrown programmatically and the `exceptionObject` is an instance of a particular exception class.

In slide 30, explain the code that demonstrates the use of the `throw` statement to the students. Then, explain the code.

Explain the students that the class `Employee` declares a `static` method called `ThrowException` that takes a `string` parameter called `name`. Tell them that if the value of `name` is `null`, the C# compiler throws the exception, which is caught by the instance of the `ArgumentNullException` class. Mention that in the `try` block, the value of `name` is `null` and the control of the program goes back to the method `ThrowException`, where the exception is thrown for referencing a `null` value. The `catch` block consists of the error handler, which is executed when the exception is thrown.

Use slide 31 to explain the output of the code.

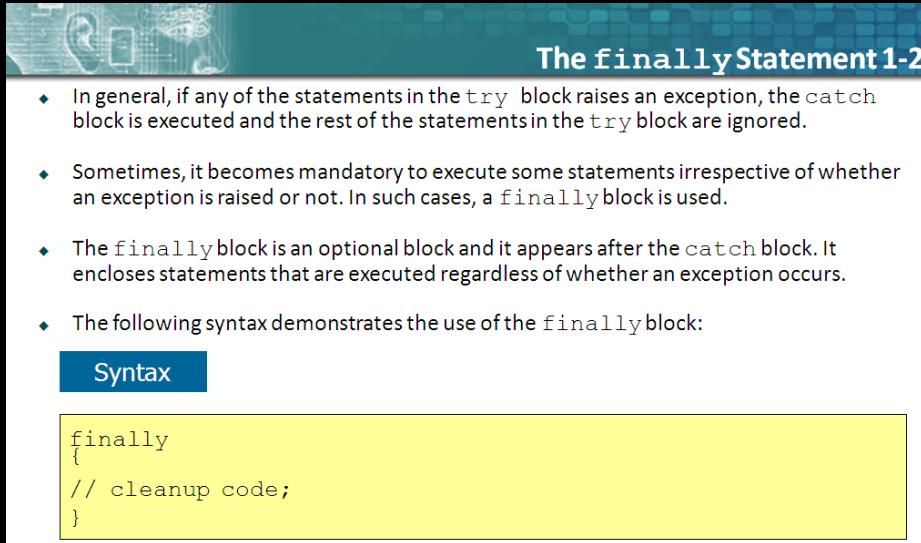
Additional Information

Refer the following link for more information on the throw statement:

<http://msdn.microsoft.com/en-us/library/xhcbs8fz%28v=vs.110%29.aspx>

Slides 32 and 33

Explain the `finally` statement.



The finally Statement 1-2

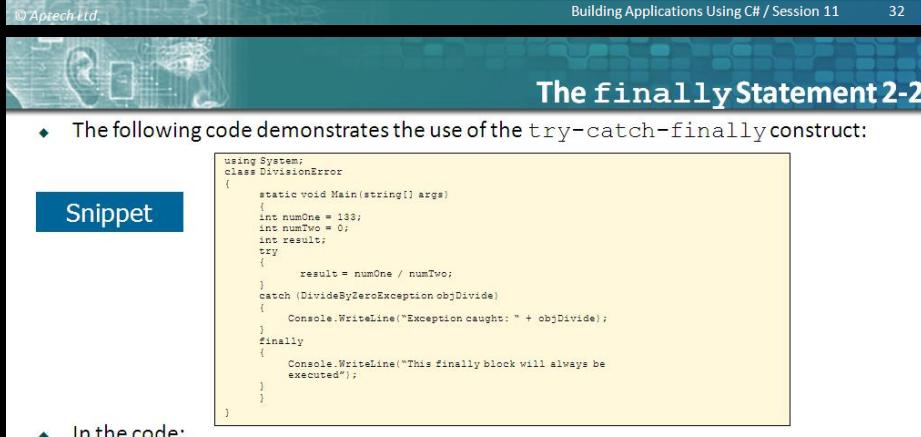
- In general, if any of the statements in the `try` block raises an exception, the `catch` block is executed and the rest of the statements in the `try` block are ignored.
- Sometimes, it becomes mandatory to execute some statements irrespective of whether an exception is raised or not. In such cases, a `finally` block is used.
- The `finally` block is an optional block and it appears after the `catch` block. It encloses statements that are executed regardless of whether an exception occurs.
- The following syntax demonstrates the use of the `finally` block:

Syntax

```
finally
{
    // cleanup code;
}
```

- where,
 - `finally`: Specifies that the statements in the block have to be executed irrespective of whether or not an exception is raised.

© Aptech Ltd. Building Applications Using C# / Session 11 32



The finally Statement 2-2

- The following code demonstrates the use of the `try-catch-finally` construct:

Snippet

```
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 the code:
 - 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
```

© Aptech Ltd. Building Applications Using C# / Session 11 33

In slide 32, tell the students that 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.

Tell them that 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.

Mention that 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.

Finally block will be executed irrespective of an exception is raised or not. Use a finally block to release resources, for example to close any streams or files that were opened in the `try` block.

Then, explain the syntax that demonstrates the use of the `finally` block to the students and tell them that the word `finally` specifies that the statements in the block have to be executed irrespective of whether or not an exception is raised.

In slide 33, explain the code that demonstrates the use of the `try-catch-finally` construct to the students.

Tell the students that in the code, the `Main()` method of the class `DivisionError` declares and initializes two variables. Tell them that 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. Mention that the `finally` block is executed at the end even though an exception is thrown by the `try` block.

Then, explain the output of the code to the students.

Slide 34

Describe the performance of the program hinders by exceptions.

Performance 1-7

- ◆ Exceptions hinder the performance of a program.
- ◆ There are two design patterns that help in minimizing the hindrance caused due to exceptions. These are:
 - ◆ **Tester-Doer Pattern:**
 - A call that might throw exceptions is divided into two parts, tester and doer, using the tester-doer pattern.
 - The tester part will perform a test on the condition that might cause the doer part to throw an exception.
 - This test is then placed just before the code that throws an exception.



©Aptech Ltd. Building Applications Using C# / Session 11 34

In slide 34, tell the students that the exceptions hinder the performance of a program. Tell them that the two design patterns that help in minimizing the hindrance caused due to exceptions are **Tester-Doer Pattern** and **TryParse Pattern**.

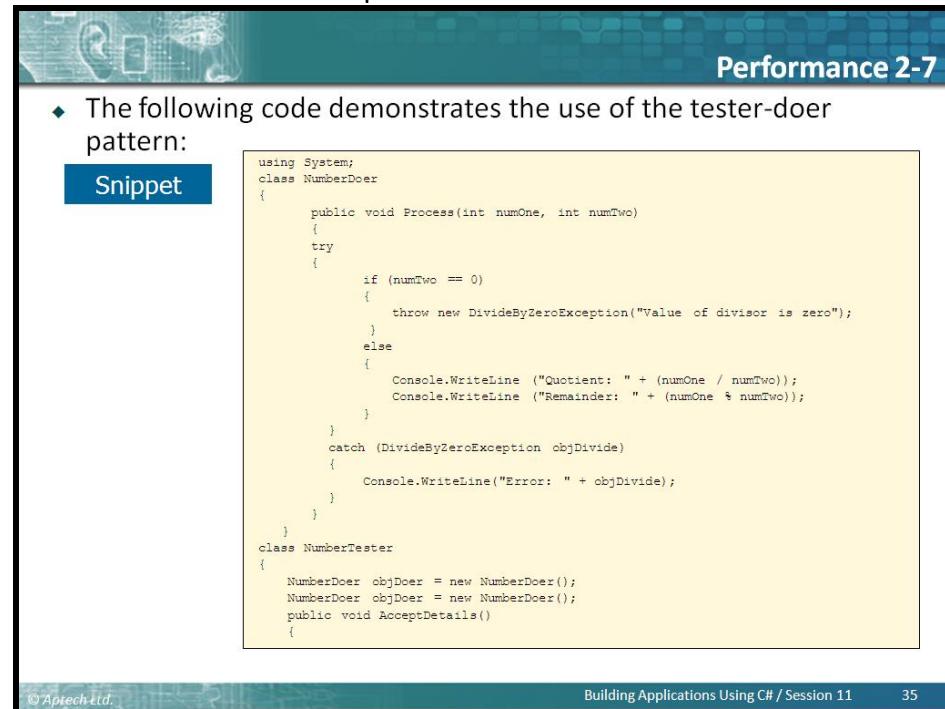
Explain the Tester-Doer Pattern.

Tell the students that a call that might throw exceptions is divided into two parts, tester and doer, using the tester-doer pattern.

Tell them that the tester part will perform a test on the condition that might cause the doer part to throw an exception. This test is then placed just before the code that throws an exception.

Slide 35

Understand the use of the tester-doer pattern.



The slide has a teal header bar with a small icon on the left and the text "Performance 2-7" on the right. Below the header is a white content area. On the left side of the content area, there is a blue button-like shape containing the word "Snippet". To the right of this, a blue diamond-shaped bullet point indicates the following text describes the use of the tester-doer pattern. The main content is a block of C# code:

```
using System;
class NumberDoer
{
    public void Process(int numOne, int numTwo)
    {
        try
        {
            if (numTwo == 0)
            {
                throw new DivideByZeroException("Value of divisor is zero");
            }
            else
            {
                Console.WriteLine ("Quotient: " + (numOne / numTwo));
                Console.WriteLine ("Remainder: " + (numOne % numTwo));
            }
        }
        catch (DivideByZeroException objDivide)
        {
            Console.WriteLine("Error: " + objDivide);
        }
    }
}
class NumberTester
{
    NumberDoer objDoer = new NumberDoer();
    NumberDoer objDoer = new NumberDoer();
    public void AcceptDetails()
    {
```

In slide 35, explain the code that demonstrates the use of the tester-doer pattern to the students.

Slide 36

Understand the use of the tester-doer pattern.

The screenshot shows a presentation slide with a blue header bar containing the text "Performance 3-7". Below the header is a yellow rectangular area containing C# code. At the bottom of the slide is a dark footer bar with the text "Aptech Ltd.", "Building Applications Using C# / Session 11", and the number "36".

```
int dividend = 0;
int divisor = 0;
Console.WriteLine("Enter the value of dividend");
try {
    dividend = Convert.ToInt32(Console.ReadLine());
}
catch (FormatException objFormat) {
    Console.WriteLine("Error: " + objFormat);
}
Console.WriteLine("Enter the value of divisor");
try {
    divisor = Convert.ToInt32(Console.ReadLine());
}
catch (FormatException objFormat) {
    Console.WriteLine("Error: " + objFormat);
}
if ((dividend > 0) || (divisor > 0))
{
    objDoer.Process(dividend, divisor);
}
else
{
    Console.WriteLine("Invalid input");
}
}
static void Main(string[] args)
{
    NumberTester objTester = new NumberTester();
    objTester.AcceptDetails();
}
```

Explain the further part of the code as shown in slide 36.

Slide 37

Understand the code that demonstrates the use of the tester-doer pattern.

Performance 4-7

- ◆ In the code:
 - ◆ Two classes, **NumberDoer** and **NumberTester**, are defined. In the **NumberTester** class, if the value entered is not of **int** data type, an exception is thrown.
 - ◆ Next, a test is performed to check if either of the values of dividend and divisor is greater than 0. If the result is true, the values are passed to the **Process ()** method of the **NumberDoer** class. In the **NumberDoer** class, another test is performed to check whether the value of the divisor is equal to 0.
 - ◆ If the condition is true, then the **DivideByZero** exception is thrown and caught by the **catch** block.
- ◆ The following figure displays the use of tester-doer pattern:

```
C:\WINDOWS\system32\cmd.exe
Enter the value of dividend
5
Enter the value of divisor
0
Error: System.DivideByZeroException: Value of divisor is zero
  at Project.NumberDoer.Process<Int32 numOne, Int32 numTwo> in D:\Source Code\Project\Project\NumberTester.cs:line 15
Press any key to continue . . .
```

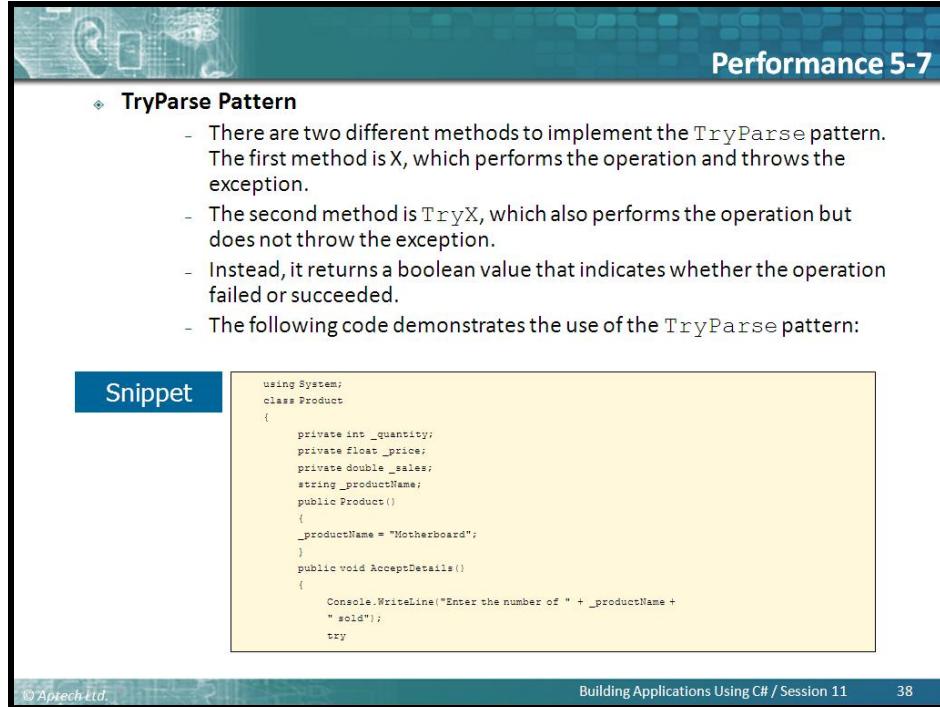
Use slide 37 to explain the code that demonstrates the use of the tester-doer pattern.

Tell the students that in the code, two classes, **NumberDoer** and **NumberTester**, are defined in the code. Tell them that in the **NumberTester** class, if the value entered is not of **int** data type, an exception is thrown. Next, a test is performed to check if either of the values of dividend and divisor is greater than 0. Tell them that if the result is true, the values are passed to the **Process ()** method of the **NumberDoer** class. Mention that in the **NumberDoer** class, another test is performed to check whether the value of the divisor is equal to 0. If the condition is true, then the **DivideByZero** exception is thrown and caught by the **catch** block.

Then, refer to the figure in slide 37 to display the output of the code that demonstrates the use of tester-doer pattern.

Slide 38

Understand TryParse Pattern.



The slide has a blue header bar with a gear icon and the text "Performance 5-7". The main content area has a teal header "TryParse Pattern". Below it is a bulleted list of four points. A "Snippet" box contains C# code for a Product class. The footer bar is teal with the text "© Aptech Ltd." and "Building Applications Using C# / Session 11 38".

❖ TryParse Pattern

- There are two different methods to implement the TryParse pattern. The first method is X, which performs the operation and throws the exception.
- The second method is TryX, which also performs the operation but does not throw the exception.
- Instead, it returns a boolean value that indicates whether the operation failed or succeeded.
- The following code demonstrates the use of the TryParse pattern:

Snippet

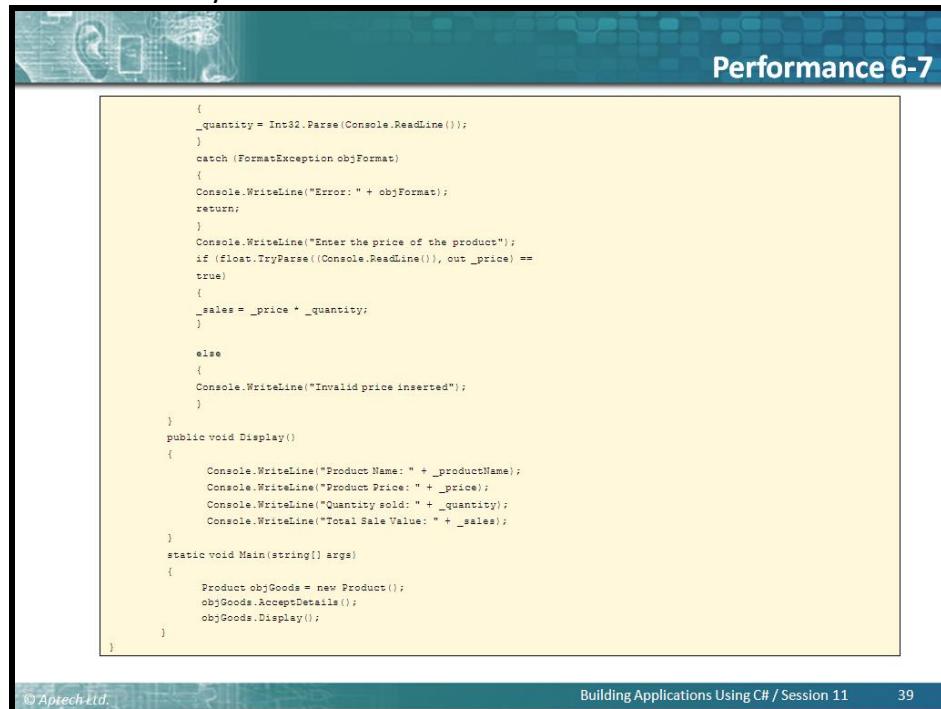
```
using System;
class Product
{
    private int _quantity;
    private float _price;
    private double _sales;
    string _productName;
    public Product()
    {
        _productName = "Motherboard";
    }
    public void AcceptDetails()
    {
        Console.WriteLine("Enter the number of " + _productName +
        " sold");
        try
```

In slide 38, tell the students that there are two different methods to implement the TryParse pattern. Tell them that the first method is X, which performs the operation and throws the exception. Then, tell that the second method is TryX, which also performs the operation but does not throw the exception. Instead, it returns a boolean value that indicates whether the operation failed or succeeded.

Then, explain the code that demonstrates the use of the TryParse pattern to the students.

Slide 39

Understand the code for TryParse Pattern.



The slide has a blue header bar with the text "Performance 6-7". The main content area is yellow. At the bottom, there is a green footer bar with the text "© Aptech Ltd." on the left, "Building Applications Using C# / Session 11" in the center, and the number "39" on the right.

```
        {
            _quantity = Int32.Parse(Console.ReadLine());
        }
        catch (FormatException objFormat)
        {
            Console.WriteLine("Error: " + objFormat);
            return;
        }
        Console.WriteLine("Enter the price of the product");
        if (float.TryParse((Console.ReadLine()), out _price) == true)
        {
            _sales = _price * _quantity;
        }
        else
        {
            Console.WriteLine("Invalid price inserted");
        }
    }
    public void Display()
    {
        Console.WriteLine("Product Name: " + _productName);
        Console.WriteLine("Product Price: " + _price);
        Console.WriteLine("Quantity sold: " + _quantity);
        Console.WriteLine("Total Sale Value: " + _sales);
    }
    static void Main(string[] args)
    {
        Product objGoods = new Product();
        objGoods.AcceptDetails();
        objGoods.Display();
    }
}
```

In slide 39, explain the further part of the code used to implement TryParse pattern.

Slide 40

Understand the code and its output.

Performance 7-7

- ◆ In the code:
 - ❖ A class **Product** is defined in which the total product sales is calculated.
 - ❖ The user-defined function **AcceptDetails** accepts the number of products sold and the price of each product. The number of products sold is converted to **int** data type using the **Parse()** method of the **Int32** structure.
 - ❖ This method throws an exception if the value entered is not of the **int** data type. Hence, this method is included in the **try...catch** block.
 - ❖ When accepting the price of the product, the **TryParse** pattern is used to verify whether the value entered is of the correct data type.
 - ❖ If the value returned by the **TryParse()** method is true, the sale price is calculated.
- ◆ The following figure displays the output of the code when the user enters a value which is not of **int** data type:

Use slide 40 to understand the code provided in slide 39.

Tell the students that in the code, a class **Product** is defined in which the total product sales is calculated. Tell them that the user-defined function **AcceptDetails** accepts the number of products sold and the price of each product. The number of products sold is converted to **int** data type using the **Parse()** method of the **Int32** structure. Tell them that this method throws an exception if the value entered is not of the **int** data type. Hence, this method is included in the **try...catch** block. When accepting the price of the product, the **TryParse** pattern is used to verify whether the value entered is of the correct data type. Mention that if the value returned by the **TryParse()** method is true, the sale price is calculated.

Then, refer to the figure in slide 40 to explain the output of the code when the user enters a value which is not of **int** data type.

Slide 41

Understand nested `try` and multiple `catch` blocks.

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.



© Aptech Ltd. Building Applications Using C# / Session 11 41

In slide 41, tell the students that exception handling code can be nested in a program. Tell them that in nested exception handling, a `try` block can enclose another `try-catch` block. In addition, mention that 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. The various types of `try` blocks that can exist are as follows:

- With only one `catch` block.
- With multiple `catch` block.
- With only a `finally` block.
- With one `catch` block and a `finally` block.
- With one or more `catch` blocks and a `finally` block.

Slide 42

Understand the features of nested `try` blocks.

Nested try Blocks 1-4

- ◆ Following are the features of the nested `try` block:

The nested `try` block consists of multiple `try-catch` constructs that 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.

In slide 42, explain the students that the nested `try` block consists of multiple `try-catch` constructs. Tell them that 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.

Then, tell the students to consider an outer `try` block containing a nested `try-catch-finally` construct. Tell them 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.

Slide 43

Understand the creation of nested try...catch blocks.

The slide has a blue header bar with icons of a computer monitor, keyboard, and mobile phone. The title 'Nested try Blocks 2-4' is in white text on the right side of the header. Below the header is a yellow rectangular box containing C# code. To the left of the code box is a blue button labeled 'Syntax'. The code itself is a nested try...catch block structure:

```
try
{
    // outer try block
    try
    {
        // inner try block
    }
    catch
    {
        // inner catch block
    }
    // this is optional
    finally
    {
        // inner finally block
    }
}
catch
{
    // outer catch block
}
// this is optional
finally
{
    // outer finally block
}
```

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 11' in the center, and the number '43' on the right.

In slide 43, explain the syntax used to create nested try...catch blocks to the students.

Slide 44

Understand the code for nested `try` blocks.

The slide has a blue header bar with a decorative background featuring icons of a computer monitor, keyboard, and mobile phone. The title 'Nested try Blocks 3-4' is centered in white text. Below the title, a bulleted list states: 'The following code demonstrates the use of nested `try` blocks:'. A blue button labeled 'Snippet' is positioned above the code block. The code itself is presented in a syntax-highlighted text area:

```
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 (ArithmetcException objMaths)
        {
            Console.WriteLine("Divide by 0 " + objMaths);
        }
        names[2] = "Smith";
    }
    catch (IndexOutOfRangeException objIndex)
    {
        Console.WriteLine("Wrong number of arguments supplied " + objIndex);
    }
}
```

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 11' in the center, and the page number '44' on the right.

Using slide 44, explain the code that demonstrates the use of nested `try` blocks to the students.

Slide 45

Understand the code.

Nested try Blocks 4-4

- ◆ In the code:
 - ◆ The array variable called `names` of type `string` is initialized to have two values.
 - ◆ The outer `try` block consists of another `try-catch` construct.
 - ◆ The inner `try` block divides two numbers. As an attempt is made to divide the number by zero, the inner `try` block throws an exception, which is handled by the inner `catch` block.
 - ◆ In addition, in the outer `try` block, there is a statement referencing a third array element whereas, the array can store only two values. So, the outer `try` block also throws an exception, which is handled by the outer `catch` block.

Output

```
This is the outer try block
Divide by 0 System.DivideByZeroException: Attempted to divide by zero.
   at Product.Main(String[] args) in
c:\ConsoleApplication1\Program.cs:line 52
Wrong number of arguments supplied System.IndexOutOfRangeException: Index
was outside the bounds of the array.

   at Product.Main(String[] args) in
c:\ConsoleApplication1\Program.cs:line 58
```

Use slide 45 to make students understand about the concept of nested try catch blocks.

Tell the students that in the code, the array variable called `names` of type `string` is initialized to have two values. Tell them that the outer `try` block consists of another `try-catch` construct. The inner `try` block divides two numbers.

Then, explain that 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, mention that 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.

Then, explain the output of the code to the students.

Slide 46

Understand the multiple catch blocks.

Multiple catch Blocks 1-3

- ◆ A `try` block can throw multiple types of exceptions, which need to be handled by the `catch` block. C# allows you to define multiple `catch` blocks to handle the different types of exceptions that might be raised by the `try` block. Depending on the type of exception thrown by the `try` block, the appropriate `catch` block (if present) is executed.
- ◆ However, if the compiler does not find the appropriate `catch` block, then the general `catch` block is executed.
- ◆ Once the `catch` block is executed, the program control is passed to the `finally` block (if any) and then the control terminates the `try-catch-finally` construct.
- ◆ The following syntax is used for defining multiple `catch` blocks:

Syntax

```
try
{
    // program code
}
catch (<ExceptionClass><objException>)
{
    // statements for handling the exception
}
catch (<ExceptionClass1><objException>)
{
    // statements for handling the exception
}
...

```

© Aptech Ltd. Building Applications Using C# / Session 11 46

In slide 46, tell the students that a `try` block can throw multiple types of exceptions, which need to be handled by the `catch` block. Tell them that C# allows you to define multiple `catch` blocks to handle the different types of exceptions that might be raised by the `try` block. Tell them that depending on the type of exception thrown by the `try` block, the appropriate `catch` block (if present) is executed.

Mention that if the compiler does not find the appropriate `catch` block, then the general `catch` block is executed.

Tell the students that 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.

Then, explain the syntax used for defining multiple `catch` blocks to the students.

Slide 47

Understand code to implement multiple `catch` blocks.

The slide has a blue header bar with the title "Multiple catch Blocks 2-3". Below the title is a bulleted list: "The following code demonstrates the use of multiple catch blocks:". A blue button labeled "Snippet" is positioned to the left of the code block. The code itself is a C# Main method:

```
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);
}
```

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 11", and the page number "47".

In slide 47, explain the code that demonstrates the use of multiple `catch` blocks.

Slide 48

Understand code to implement multiple `catch` blocks.

The screenshot shows a slide titled "Multiple catch Blocks 3-3". The content is a bulleted list under the heading "In the code:":

- ◆ In the code:
 - ◆ 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.
 - ◆ 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.

At the bottom of the slide, there is footer text: "© Aptech Ltd.", "Building Applications Using C# / Session 11", and "48".

In slide 48, tell the students that in the code, the array, `names`, is initialized to two element values, and two integer variables are declared and initialized. Tell them that as there is a reference to a third array element, an exception of type `IndexOutOfRangeException` is thrown and the second `catch` block is executed. Tell them that 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.

Explain to the students that 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. Tell them that 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.

Slide 49

Understand the `System.TypeInitializationException` class.

System.TypeInitializationException Class 1-2

- The `System.TypeInitializationException` class is used to handle exceptions that are thrown when an instance of a class attempts to invoke the static constructor of the class as shown in the following code:

Snippet

```
using System;
class TypeInitError
{
    static TypeInitError()
    {
        throw new ApplicationException("This program throws
TypeInitializationException error.");
    }
    static void Main(string[] args)
    {
        try
        {
            TypeInitError objType = new TypeInitError();
        }
        catch (TypeInitializationException objEx)
        {
            Console.WriteLine("Error : {0}",objEx);
        }
        catch (Exception objEx)
        {
            Console.WriteLine("Error : {0}", objEx);
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 11 49

In slide 49, tell the students that the `System.TypeInitializationException` class is used to handle exceptions that are thrown when an instance of a class attempts to invoke the static constructor of the class.

When a class initializer fails to initialize a type, a `TypeInitializationException` is created and passed a reference to the exception thrown by the type's class initializer.

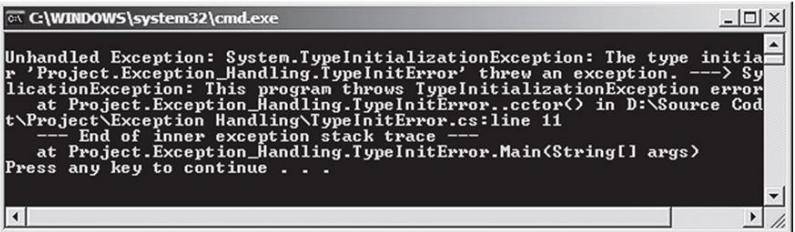
Then, explain the code that demonstrates the use of the `System.TypeInitializationException` class.

Slide 50

Understand the code and explain the output.

System.TypeInitializationException Class 2-2

- ◆ In the code:
 - ❖ A static constructor **TypeInitError** is defined. Here, only the static constructor is defined and when the instance of the class, **objType** attempts to invoke the constructor, an exception is thrown.
 - ❖ This invocation of the static constructor is not allowed and the instance of the **TypeInitializationException** class is used to display the error message in the catch block.
- ◆ The figure displays the **System.TypeInitializationException** generated at runtime:



© Aptech Ltd. Building Applications Using C# / Session 11 50

In slide 50, tell the students that a static constructor **TypeInitError** is defined. Here, only the static constructor is defined and when the instance of the class, **objType** attempts to invoke the constructor, an exception is thrown. This invocation of the static constructor is not allowed and the instance of the **TypeInitializationException** class is used to display the error message in the **catch** block.

Refer to the figure on the slide to display the **System.TypeInitializationException** generated at runtime.

In-Class Question:

After you finish explaining the `System.TypeInitializationException` class, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which are the two design patterns that help in minimizing the hindrance caused due to exceptions?

Answer:

The two design patterns that help in minimizing the hindrance caused due to exceptions are `Tester-Doer` Pattern and `TryParse` Pattern.

Slide 51

Understand custom exceptions.

Custom Exceptions

- ◆ Following are the features of custom exceptions:

They are user-defined exceptions that allow users to recognize the cause of unexpected events in specific programs and display custom messages.

Allows you to simplify and improve the process of error-handling in a program.

Even though C# provides you with a rich set of exception classes, they do not address certain application-specific and system-specific constraints. To handle such constraints, you can define custom exceptions.

Custom exceptions can be created to handle exceptions that are thrown by the CLR as well as those that are thrown by user applications.

© Aptech Ltd. Building Applications Using C# / Session 11 51

In slide 51, explain the features of custom exception to the students.

Tell the students that the custom exceptions are user-defined exceptions that allow users to recognize the cause of unexpected events in specific programs and display custom messages. Tell them that the custom exceptions allow user to simplify and improve the process of error-handling in a program.

Explain that even though C# provides a rich set of exception classes, they do not address certain application-specific and system-specific constraints. To handle such constraints, you can define custom exceptions.

Mention that the custom exceptions can be created to handle exceptions that are thrown by the CLR as well as those that are thrown by user applications.

Additional Information

Refer the following links for more information on custom exceptions in C#:

<http://www.codeproject.com/Tips/90646/Custom-exceptions-in-C-NET>

<http://msdn.microsoft.com/en-us/library/87cdya3t%28v=vs.110%29.aspx>

Slides 52 and 53

Understand how to implement the custom exceptions.

Implementing Custom Exceptions 1-2

- ◆ Custom exceptions can be created by deriving from the `Exception` class, the `SystemException` class, or the `ApplicationException` class.
- ◆ However, to define a custom exception, you first need to define a new class and then derive it from the `Exception`, `SystemException`, or `ApplicationException` class. Once you have created a custom exception, you can reuse it in any C# application.
- ◆ The following demonstrates the implementation of custom exceptions:

Snippet

```
public class CustomMessage : Exception
{
    public CustomMessage (string message) : base(message) {
    }
}
public class CustomException{
    static void Main(string[] args) {
        try
        {
            throw new CustomMessage ("This illustrates creation and catching of custom exception");
        }
        catch(CustomMessage objCustom)
        {
            Console.WriteLine(objCustom.Message);
        }
    }
}
```

©Aptech Ltd. Building Applications Using C# / Session 11 52

Implementing Custom Exceptions 2-2

- ◆ In the code:
 - ❖ The class **CustomMessage** is created, which is derived from the class **Exception**.
 - ❖ The constructor of the class **CustomMessage** is created and it takes a **string** parameter.
 - ❖ This constructor, in turn, calls the constructor of the base class, **Exception**. The class **CustomException** consists of the **Main()** method.
 - ❖ The **try** block of the class **CustomException** throws the custom exception that passes a **string** value.
 - ❖ The **catch** block catches the exception thrown by the **try** block and prints the message.

Output

This illustrates creation and catching of custom exception

In slide 52, explain to the students that the custom exceptions can be created by deriving from the **Exception** class, the **SystemException** class, or the **ApplicationException** class.

Tell them that the **SystemException** class handles exceptions thrown by the CLR. The **ApplicationException** class handles exceptions thrown by the user-created applications. Mention that to define a custom exception, first the user need to define a new class and then derive it from the **Exception**, **SystemException**, or **ApplicationException** class. Once you have created a custom exception, you can reuse it in any C# application.

Then, explain the code that demonstrates the implementation of custom exceptions.

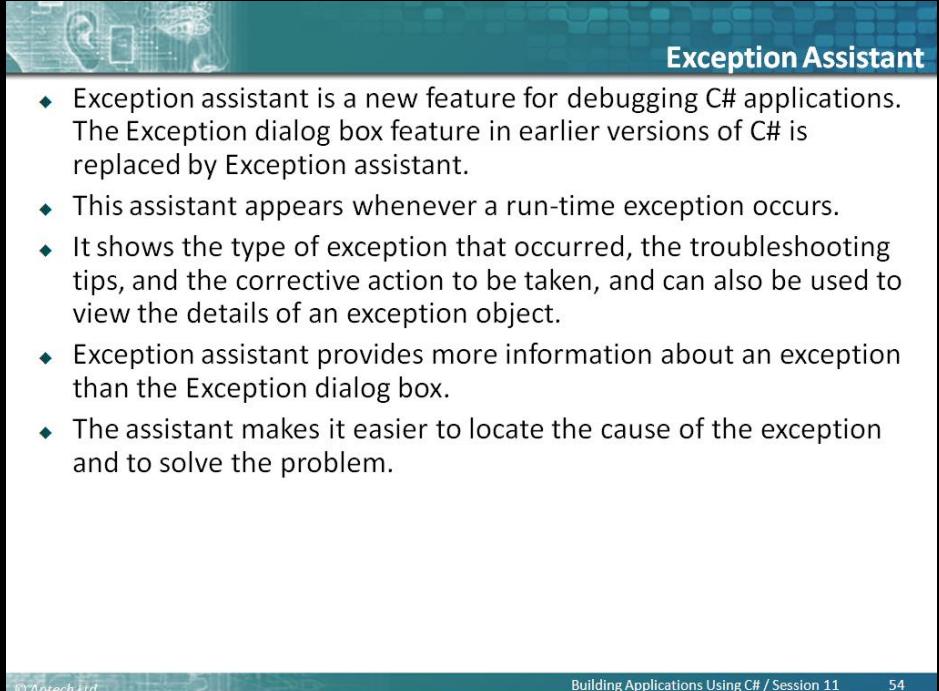
In slide 53, tell the students that in the class **CustomMessage** is created, which is derived from the class **Exception**. Tell them that the constructor of the class **CustomMessage** is created and it takes a **string** parameter. This constructor, in turn, calls the constructor of the base class, **Exception**.

Explain that the class **CustomException** consists of the **Main()** method. The **try** block of the class **CustomException** throws the custom exception that passes a **string** value. Mention that the **catch** block catches the exception thrown by the **try** block and prints the message.

Then, explain the output of the code to the students.

Slide 54

Describe exception assistant.



Exception Assistant

- ◆ Exception assistant is a new feature for debugging C# applications. The Exception dialog box feature in earlier versions of C# is replaced by Exception assistant.
- ◆ This assistant appears whenever a run-time exception occurs.
- ◆ It shows the type of exception that occurred, the troubleshooting tips, and the corrective action to be taken, and can also be used to view the details of an exception object.
- ◆ Exception assistant provides more information about an exception than the Exception dialog box.
- ◆ The assistant makes it easier to locate the cause of the exception and to solve the problem.

© Aptech Ltd. Building Applications Using C# / Session 11 54

In slide 54, tell the students that exception assistant is a new feature for debugging C# applications. The Exception dialog box feature in earlier versions of C# is replaced by Exception assistant.

Exception Assistant provides detailed information about an exception that the exception dialog box which helps to resolve an exception.

Tell them that this assistant appears whenever a run-time exception occurs. It shows the type of exception that occurred, the troubleshooting tips, and the corrective action to be taken, and can also be used to view the details of an exception object.

Additional Information

Refer the following link for more information on Exception Assistant in C#:

<http://msdn.microsoft.com/en-us/library/197c1fsc.aspx>

Slide 55

Summarize the session.

The slide features a decorative header with icons related to software development and a blue footer bar. The main content area is titled "Summary" and contains a bulleted list of points about exception handling in C#.

Summary

- ◆ Exceptions are errors that encountered at run-time.
- ◆ Exception-handling allows you to handle methods that are expected to generate exceptions.
- ◆ The `try` block should enclose statements that may generate exceptions while the `catch` block should catch these exceptions.
- ◆ The `finally` block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the `try` block.
- ◆ Nested `try` blocks allow you to have a `try-catch-finally` construct within a `try` block.
- ◆ Multiple `catch` blocks can be implemented when a `try` block throws multiple types of exceptions.
- ◆ Custom exceptions are user-defined exceptions that allow users to handle system and application-specific runtime errors.

© Aptech Ltd. Building Applications Using C# / Session 11 55

In slide 55, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- Exceptions are errors that encountered at run-time.
- Exception-handling allows you to handle methods that are expected to generate exceptions.
- The `try` block should enclose statements that may generate exceptions while the `catch` block should catch these exceptions.
- The `finally` block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the `try` block.
- Nested `try` blocks allow you to have a `try-catch-finally` construct within a `try` block.
- Multiple `catch` blocks can be implemented when a `try` block throws multiple types of exceptions.
- Custom exceptions are user-defined exceptions that allow users to handle system and application-specific runtime errors.

11.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a question to students to search additional information, such as:

1. What are some other ways to trap the exceptions and improve the performance of the program?

Session 12 - Events, Delegates, and Collections

12.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- Exceptions are errors that encountered at run-time.
- Exception-handling allows you to handle methods that are expected to generate exceptions.
- The `try` block should enclose statements that may generate exceptions while the `catch` block should catch these exceptions.
- The `finally` block is meant to enclose statements that need to be executed irrespective of whether or not an exception is thrown by the `try` block.

Nested `try` blocks allow you to have a `try-catch-finally` construct within a `try` block. Here, you can ask students the key topics they can recall from previous session. Ask them to briefly explain exceptions and custom exceptions in C#. You can also ask them to explain the process of throwing and catching exceptions. Furthermore, ask them to explain nested try and multiple catch blocks. Prepare a question or two which will be a key point to relate the current session objectives.

12.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain delegates
- Explain events
- Define and describe collections

12.1.2 Teaching Skills

To teach this session successfully, you must know about delegates in C#. You should be aware of events and collections.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here during the In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to events, delegates, and collections. They will learn about collections.

12.2 In-Class Explanations

Slide 3

Understand delegates.

The slide has a blue header bar with the title 'Delegates'. Below the header, there is a bullet point: 'Following are the features of delegates:' followed by six colored boxes containing text. The boxes are arranged in two rows of three. The first row contains three boxes: a red box, a green box, and a purple box. The second row contains three boxes: a blue box, an orange box, and a red box. At the bottom left is the Aptech logo, and at the bottom right are the slide details: 'Building Applications Using C# / Session 12' and the number '3'.

- ◆ Following are the features of delegates:

In the .NET Framework, a delegate points to one or more methods. Once you instantiate the delegate, the corresponding methods invoke.	Delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names.	Using delegates, you can call any method, which is identified only at run-time.
A delegate is like having a general method name that points to various methods at different times and invokes the required method at run-time.	In C#, invoking a delegate will execute the referenced method at run-time.	To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.

Use slide 3 to explain that in the .NET Framework, a delegate points to one or more methods. On instantiating the delegate, the corresponding methods invoke.

Also, tell that delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names. Using delegates, you can call any method, which is identified only at run-time. A method of any accessible class or struct which matches the delegate type can be assigned to delegate.

Mention that a delegate is like having a general method name that points to various methods at different times and invokes the required method at run-time. In C#, invoking a delegate will execute the referenced method at run-time.

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

Additional Information

- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.

- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods do not need to match the delegate signature exactly.

Slide 4

Understand delegates in C#.

The slide has a blue header bar with the title 'Delegates in C#' and decorative icons of a smartphone and a laptop. The main content area contains a bulleted list of features and characteristics of 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.
- ◆ Following are the features of delegates in C# that distinguish them from normal methods:
 - ◆ 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.

In slide 4, explain to the students with an example.

Tell them to 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.

Mention that 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.

Explain that delegates in C# have some features that distinguish them from normal methods. Explain the features to them.

Tell them that 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.

Tell the students that a delegate can invoke multiple methods simultaneously. This is known as multicasting. Then, tell that a delegate can encapsulate static methods.

Explain that 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.

Additional Information

Refer the following links for more information on delegates:

<http://msdn.microsoft.com/en-us/library/aa288459%28v=vs.71%29.aspx>

<http://www.codeproject.com/Articles/71154/C-Delegates-A-Practical-Example>

In-Class Question:

You will ask the students a few In-Class questions. This will help you in reviewing their understanding of the topic.



What are delegates?

Answer:

A delegate points to one or more methods. On instantiating the delegate, the corresponding methods invoke.



What is the syntax used for declaring a delegate?

Answer:

```
<access_modifier> delegate <return type>
DelegateName ([list_of_parameteres]);
```

Slide 5

Understand declaring delegates.

The slide is titled "Declaring Delegates". It contains the following points:

- 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.
- The following figure displays an example of declaring delegates:

Valid Delegate Declaration

```
public delegate int Calculation(int numOne, int numTwo); ✓
```

Invalid Delegate Declaration

```
public delegate Calculation(int numOne, int numTwo)
{
}
```

The following syntax is used to declare a delegate:

Syntax

```
<access_modifier> delegate <return_type> DelegateName([list_of_parameters]);
```

where,

- `access_modifier`: Specifies the scope of access for the delegate. If declared outside the class, the scope will always be `public`.
- `return_type`: Specifies the data type of the value that is returned by the method.
- `DelegateName`: Specifies the name of the delegate.
- `list_of_parameters`: Specifies the data types and names of parameters to be passed to the method.

The following code declares the delegate `Calculation` with the return type and the parameter types as integer:

Snippet

```
public delegate int Calculation(int numOne, int numTwo);
```

© Aptech Ltd. Building Applications Using C# / Session 12 5

Using slide 5, explain the students the delegates in C# are declared using the `delegate` keyword followed by the return type, a delegate name and the parameters of the referenced method.

Tell the students that 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.

You can refer to the figure in slide 5 to explain an example of declaring delegates.

Explain the syntax that is used to declare a delegate 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.

Explain the code that declares the delegate `Calculation` with the return type and the parameter types as integer.

Tips:

If the delegate is declared outside the class, it cannot declare another delegate with the same name in that namespace.

Slide 6

Understand instantiating delegates.

Instantiating Delegates 1-2

- The next step after declaring the delegate is to instantiate the delegate and associate it with the required method by creating an object of the delegate.
- Like all other objects, an object of a delegate is created using the `new` keyword.
- This object takes the name of the method as a parameter and this method has a signature similar to that of the delegate.
- The created object is used to invoke the associated method at run-time.
- The following figure displays an example of instantiating delegates:

```
Calculation objCalculation = new Calculation(Addition);
```

↑ ↑ ↑
 Delegate Name Object Name Referenced Method

- The following syntax is used to instantiate a delegate:

Syntax

```
<DelegateName><objName> = new <DelegateName>(<MethodName>);
```

- where,
 - `DelegateName`: Specifies the name of the delegate .
 - `objName`: Specifies the name of the delegate object.
 - `MethodName`: Specifies the name of the method to be referenced by the delegate object.

© Aptech Ltd. Building Applications Using C# / Session 12 6

Use slide 6 to tell the students the next step after declaring the delegate is to instantiate the delegate and associate it with the required method. Here, create an object of the delegate, which is also known as instantiating delegate.

Explain that like all other objects, an object of a delegate is created using the `new` keyword. This object takes the name of the method as a parameter and this method has a signature similar to that of the delegate. The created object is used to invoke the associated method at run-time.

A delegate refers to either an instance method or a static method.

You can refer slide 6 to explain an example of instantiating delegates.

Then, explain the syntax that is used to instantiate a delegate 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.

Slide 7

Understand how to declare a delegate.

Instantiating Delegates 2-2

- The following code declares a delegate **Calculation** outside the class **Mathematics** and instantiates it in the class:

Snippet

```
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 the code:
 - 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

In slide 7, tell the students the code that declares a delegate **Calculation** outside the class **Mathematics** and instantiates it in the class.

Then, explain the code and the output.

Tell the students that in the code, the delegate called **Calculation** is declared outside the class **Mathematics**.

Mention that 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**.

Slide 8

Understand using delegates.

Using Delegates 1-2

- ◆ A delegate can be declared either before creating the class (having the method to be referenced) or can be defined within the class.
- ◆ The following are the four steps to implement delegates in C#:

```

graph TD
    A[Declare a delegate.] --> B[Create the method to be referenced by the delegate.]
    B --> C[Instantiate the delegate.]
    C --> D[Call the method using the object of the delegate.]
  
```

- ◆ Each of these step is demonstrated with an example shown in the following figure:

```

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);
    }
}
  
```

©Aptech Ltd. Building Applications Using C# / Session 12 8

Use slide 8 to explain the students that a delegate can be declared either before creating the class (having the method to be referenced) or can be defined within the class.

Explain the four steps to implement delegates in C#.

- Declare a delegate.
- Create the method to be referenced by the delegate.
- Instantiate the delegate.
- Call the method using the object of the delegate.

Mention that each of these steps is demonstrated with an example.

Simple steps to follow for delegate declaration:

- Delegate declaration
- Delegate method definition
- Delegate instantiation
- Delegate invocation

You can refer to the figure in slide 8.

Slide 9

Understand using delegates.

The slide has a blue header bar with the title "Using Delegates 2-2". Below the title is a code block. A yellow box highlights the anonymous method definition. A brace on the right side of the yellow box is labeled "Anonymous Method".

```

void Action()
{
    System.Threading.Thread objThread = new
    System.Threading.Thread
    (delegate()
    {
        Console.Write("Testing... ");
        Console.WriteLine("Threads.");
    });
    objThread.Start();
}

```

Anonymous Method

©Aptech Ltd. Building Applications Using C# / Session 12 9

Use slide 9 to explain that an anonymous method is an inline block of code that can be passed as a delegate parameter. Using anonymous methods helps to avoid creating named methods. You can refer to the figure in slide 9 that displays an example of using anonymous methods.

Additional Information

Refer the following links for more information on anonymous methods with delegates:

<http://msdn.microsoft.com/en-us/library/98dc08ac.aspx>

<http://codebetter.com/karlseguin/2008/11/27/back-to-basics-delegates-anonymous-methods-and-lambda-expressions/>

Slide 10

Understand delegate-event model.

The slide has a blue header bar with the title 'Delegate-Event Model'. Below the header, there is a bulleted list of points about the delegate-event model. A blue box labeled 'Example' contains another bulleted list of scenarios. At the bottom of the slide, there is a footer bar with the copyright information '© Aptech Ltd.' and the page number '10'.

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 by using the medium, every time the source generates an event, the event is notified to the registered listeners.

Example

- ◆ 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 they take methods that need to be invoked when events occur which are referred to as the event handlers.

© Aptech Ltd. Building Applications Using C# / Session 12 10

In slide 10, explain that 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.

Tell the students that this model consists of an event source, which is the console window in case of console-based applications. Then, explain that the listeners receive the events from the event source. Also, inform that a medium that gives the necessary protocol by which every event is communicated.

Mention that 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.

Give them an example for this. Tell them that, 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.

Explain that 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.

Tell that 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.

Also mention that delegates can be used to handle events. As parameters, they take methods that need to be invoked when events occur. These methods are referred to as the event handlers.

Slides 11 and 12

Understand multiple delegates.

Multiple Delegates 1-2

- 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.
- The following code demonstrates the use of multiple delegates by creating two delegates `CalculateArea` and `CalculateVolume` that have their return types and parameter types as double:

Snippet

```
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));
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 12 11

Multiple Delegates 2-2

- ◆ In the code:
 - ◊ 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.
- ◆ The following figure shows the use of multiple delegates:

© Aptech Ltd.

Building Applications Using C# / Session 12 12

Use slide 11 to explain that 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.

Tell the students that the code demonstrates the use of multiple delegates by creating two delegates **CalculateArea** and **CalculateVolume** that have their return types and parameter types as double.

In slide 12, explain the code.

Tell that in the code, 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.

You can refer the figure in slide 12 to demonstrate the use of multiple delegates.

Slides 13 to 15

Understand multicast delegates.

Multicast Delegates 1-3

- ◆ A single delegate can encapsulate the references of multiple methods at a time to hold a number of method references.
- ◆ Such delegates are termed as ‘Multicast Delegates’ that maintain a list of methods (invocation list) that will be automatically called when the delegate is invoked.
- ◆ Multicast delegates in C# are sub-types of the `System.MulticastDelegate` class. Multicast delegates are defined in the same way as simple delegates, however, the return type of multicast delegates can only be `void`.
- ◆ If any other return type is specified, a run-time exception will occur 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 resulting 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.



Multicast Delegates 2-3

- ◆ The following code creates a multicast delegate `Maths`. This delegate encapsulates the reference to the methods `Addition`, `Subtraction`, `Multiplication`, and `Division`:

Snippet

```
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);
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 12

14

©Aptech Limited

Multicast Delegates 3-3

```

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 the code:
 - + 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.
- ◆ The following figure shows the creation of a multicast delegate:

In slide 13, tell the students that a single delegate can encapsulate the references of multiple methods at a time.

Tell the students that 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.

One of the important properties of delegates is that multiple objects can be assigned to one delegate instance by using `+` operator.

In case of multicast delegates, the delegates should be of the same type, and delegates will be invoked in the same order.

Mention that 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`.

Tell that if any other return type is specified, a run-time exception will occur. This is because if the delegate returns a value, the return value of the last method in the invocation list of the delegate will become the return type of the delegate. This will result in inappropriate results. Hence, the return type is always `void`.

Explain that to add methods into the invocation list of a multicast delegate, the user can use the `‘+’` or the `‘+=’` assignment operator.

Explain that, 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.

Multicast delegate example:

```

mydelegate m1=new mydelegate (t.method1);
mydelegate m2=new mydelegate (t.method2);
mydelegate m3=m1+m2; //delegate multicasting

```

```
m3() //multicasted delegate invocation
```

In slides 14 and 15, explain the code that creates a multicast delegate **Maths**. This delegate encapsulates the reference to the methods **Addition**, **Subtraction**, **Multiplication**, and **Division**.

Use slide 15 to explain the code.

Explain that in the code, 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.

You can refer to the figure in slide 15 that shows the creation of a multicast delegate.

Slide 16

Understand `System.Delegate` Class.

System.Delegate Class 1-5

- 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.
- The following table lists the constructors defined in the `Delegate` class:

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

- The following table lists the properties defined in the `Delegate` class:

Property	Description
<code>Method</code>	Retrieves the referenced method
<code>Target</code>	Retrieves the object of the class in which the delegate invokes the referenced method

©Aptech Ltd. Building Applications Using C# / Session 12 16

In slide 16, tell the students that the `Delegate` class present in `System` namespace is a built-in class defined to create delegates in C#. All delegates in C# implicitly inherit from the `Delegate` class.

Explain that 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.

You can refer constructors table in slide 16 that lists the constructors defined in the `Delegate` class.

You can also refer property table in slide 16 that lists the properties defined in the `Delegate` class.

Slides 17 and 18

Understand properties and methods of the built-in Delegate class.

System.Delegate Class 2-5

- The following table lists some of the methods defined in the Delegate class:

Method	Description
Clone	Makes a copy of the current delegate
Combine	Merges the invocation lists of the multicast delegates
CreateDelegate	Declares and initializes a delegate
DynamicInvoke	Calls the referenced method at run-time
GetInvocationList	Retrieves the invocation list of the current delegate

- The following code demonstrates the use of some of the properties and methods of the built-in Delegate class:

Snippet

```

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 + " ");
        }
    }
}

```

System.Delegate Class 3-5

```

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");
    }
}

```

Use slide 17 to refer to table that lists some of the methods defined in the Delegate class. Also, using slides 17 and 18, explain the code that demonstrates properties and methods of the Delegate class.

Slide 19

Understand properties and methods of the built-in Delegate class.

The slide has a blue header bar with the title "System.Delegate Class 4-5". Below the header is a yellow rectangular area containing C# code. At the bottom of the slide is a dark footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 12", and "19".

```

    }
    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 the code:
 - ◆ 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.

In slide 19, explain the further part of the code to the students and tell them what is being done using the code. Explain to the students that in the code, the delegate **Messenger** is instantiated in the **Start ()** method. Tell that 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.

Slide 20

Understand properties and methods of the built-in Delegate class.

System.Delegate Class 5-5

- ◆ 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.
- ◆ The following figure 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 . .

```

© Aptech Ltd. Building Applications Using C# / Session 12 20

Use slide 20 to tell that 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.

Explain to the students that 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.

You can refer to the figure in slide 20 for output of the code.

Clone and Combine, two methods of `Delegate` class are different in behavior, as `copy` method creates copy of current delegate as a single delegate where `combine` merges the multiple invocation list of multicast delegates.

Slide 21

Understand events.

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).
- ◆ The following figure depicts the concept of events:

In slide 21, give the students an example for this. Tell them to 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.

Explain that 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).

Also, tell the students that in C#, events allow an object (source of the event) to notify other objects (subscribers) about the event (a change having occurred).

You can refer slide 21 that depicts the concept of events.

Slide 22

Understand features of events.

Features

- ◆ An event is a user-generated or system-generated action that enables the required objects to notify other objects or classes to handle the event. Events in C# have the following features:
 - They can be declared in classes and interfaces.
 - They can be declared as abstract or sealed.
 - They can be declared as virtual.
 - They are implemented using delegates.
- ◆ Events can be used to perform customized actions that are not already supported by C#.
- ◆ Events are widely used in creating GUI based applications, where events such as, selecting an item from a list and closing a window are tracked.

© Aptech Ltd. Building Applications Using C# / Session 12 22

In slide 22, explain that 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.

Explain the features of events:

- 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.

Mention that events can be used to perform customized actions that are not already supported by C#.

Also, tell that 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.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is an event?

Answer:

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.

Slide 23

Understand creating and using events.



Creating and Using Events

- ◆ Following are the four steps for implementing events in C#:

Define a public delegate for the event.	Create the event using the delegate.
Subscribe to listen and handle the event.	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.

© Aptech Ltd.

Building Applications Using C# / Session 12 23

Use slide 23 to explain the four steps for implementing events in C#.

- Define a public delegate for the event.
- Create the event using the delegate.
- Subscribe to listen and handle the event.
- Raise the event.

Explain that 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.

Slides 24 and 25

Understand declaring events.

Declaring Events 1-2

- 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.
- The following figure 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>;
```

- 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 which is known as subscribing to an event.

Declaring Events 2-2

- To unsubscribe from an event, use the **- = subtraction assignment** operator.
- The following syntax is used to create a method in the receiver class:

Syntax

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

- where,
 - * `objectName`: Is the object of the class in which the event handler is defined.
- The following code associates the event handler to the declared event:

```
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 the code:
 - * 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`.

Use slide 24 to explain that an event declaration consists of two steps, creating a delegate and creating the event. A delegate is declared using the `delegate` keyword.

Tell the students that 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.

Also, tell that this declaration associates the event with the delegate. Refer slide 24 that displays the syntax for declaring delegates and events.

In slide 25, explain that 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.

Explain that to unsubscribe from an event, use the **- = subtraction assignment** operator.

Explain the syntax that is used to create a method in the receiver class where, **objectName:** Is the object of the class in which the event handler is defined.

Explain the code that associates the event handler to the declared event.

Tell the students that in the code, 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**.

Slides 26 and 27

Understand raising events.

Raising Events 1-2

- ◆ An event is raised to notify all the objects that have subscribed to the event. Events are either raised by the user or the system.
- ◆ Once an event is generated, all the associated event handlers are executed. The delegate calls all the handlers that have been added to the event.
- ◆ However, before raising an event, it is important for you to create handlers and thus, make sure that the event is associated to the appropriate event handlers.
- ◆ If the event is not associated to any event handler, the declared event is considered to be **null**.
- ◆ The following figure 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

©Aptech Ltd. Building Applications Using C# / Session 12 26

Raising Events 2-2

- The following code can be used to check a particular condition before raising the event:

Snippet

```
if(condition)
{
    eventMe();
}
```

- In the code:
 - 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.

© Aptech Ltd.

Building Applications Using C# / Session 12 27

Use slide 26 to explain that 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.

Tell the students that whenever some action is generated an event is raised with the respective event handlers.

Explain that the delegate calls all the handlers that have been added to the event. However, before raising an event, it is important to create handlers and thus, make sure that the event is associated to the appropriate event handlers.

Mention that if the event is not associated to any event handler, the declared event is considered to be **null**.

You can refer slide 26 that displays an example that shows raising events.

In slide 27, tell the students that a code can be used to check a particular condition before raising the event. Explain the code. In the code, if the checked condition is satisfied, the event **eventMe** is raised.

Mention that 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.

Slides 28 and 29

Understand events and inheritance.

Events and Inheritance 1-2

- 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. The following code illustrates how an event can be indirectly invoked:

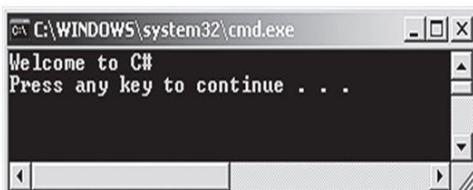
Snippet

```
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();
    }
}
```

Events and Inheritance 2-2

- In the code:
 - 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**.
 - The following figure shows the outcome of invoking the event:



In slide 28, explain that 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.

Also, tell the students that 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.

Tell that the code illustrates how an event can be indirectly invoked.

Explain the code. Tell the students that in the code, 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**.

Tell that 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.

Explain that 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**. You can refer to the figure in slide 29 that shows the outcome of invoking the event.

With this slide, you will finish explaining events.

Tips:

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.

Slide 30

Understand collections.

The slide features a decorative header with a blue gradient and a digital circuit pattern. The title 'Collections' is centered in a white box. Below the title is a bulleted list of three points. A table follows, comparing 'Array' and 'Collection' across four categories. The footer contains copyright information and navigation icons.

- ◆ A collection is a set of related data that may not necessarily belong to the same data type that can be set or modified dynamically at run-time.
- ◆ Accessing collections is similar to accessing arrays, where elements are accessed by their index numbers. However, there are differences between arrays and collections in C#.
- ◆ The following table lists the differences between arrays and collections:

Array	Collection
Cannot be resized at run-time.	Can be resized at run-time.
The individual elements are of the same data type.	The individual elements can be of different data types.
Do not contain any methods for operations on elements.	Contain methods for operations on elements.

© Aptech Ltd.

Building Applications Using C# / Session 12 30

Use slide 30 to explain that a collection is a set of related data that may not necessarily belong to the same data type.

Tell the students that it can be set or modified dynamically at run-time.

Accessing collections is similar to accessing arrays, where elements are accessed by their index numbers. However, there are differences between arrays and collections in C#.

You can refer to the table in slide 30 that lists the differences between arrays and collections.

Slide 31

Understand System.Collections namespace.

The slide has a blue header bar with the title 'System.Collections Namespace 1-3'. Below the title is a bulleted list of three points. At the bottom is a table listing various classes and interfaces with their descriptions.

- ◆ The System.Collections namespace in C# allows you to construct and manipulate a collection of objects that includes elements of different data types. The System.Collections namespace defines various collections such as dynamic arrays, lists, and dictionaries.
- ◆ The System.Collections namespace consists of classes and interfaces that define the different collections.
- ◆ The following table 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

© Aptech Ltd. Building Applications Using C# / Session 12 31

Use slide 31 to explain that the System.Collections namespace in C# allows constructs and manipulates a collection of objects.

Mention that this collection can include elements of different data types. The System.Collections namespace defines various collections such as dynamic arrays, lists, and dictionaries.

Also, tell that the System.Collections namespace consists of classes and interfaces that define the different collections.

You can refer to the table in slide 31 that lists the commonly used classes and interfaces in the System.Collections namespace.

In-Class Question:

After you finish explaining the namespaces, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is a collection?

Answer:

A collection is a set of related data that may not necessarily belong to the same data type.

Slides 32 and 33

Understand commonly used classes and interfaces of the `System.Collections` namespace.

System.Collections Namespace 2-3

- ◆ The following code demonstrates the use of the commonly used classes and interfaces of the `System.Collections` namespace:

Snippet

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

class Employee : DictionaryBase
{
    public void Add(int id, string name)
    {
        Dictionary.Add(id, name);
    }
    public void OnRemove(int id)
    {
        Console.WriteLine("You are going to delete record
containing ID: " + id);
        Dictionary.Remove(id);
    }
    public void GetDetails()
    {
        IDictionaryEnumerator objEnumerate =
        Dictionary.GetEnumerator();
        while (objEnumerate.MoveNext())
        {
            Console.WriteLine(objEnumerate.Key.ToString() +
"\t\t" +
objEnumerate.Value);
        }
    }
    static void Main(string[] args)
    {
```

© Aptech Ltd. Building Applications Using C# / Session 12 32

System.Collections Namespace 3-3

```

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 the code:
 - * 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.
- ◆ The following figure displays the output of the example:

©Aptech Ltd. Building Applications Using C# / Session 12 33

In slide 32, explain that the code demonstrates the use of the commonly used classes and interfaces of the **System.Collections** namespace.

Understand commonly used classes and interfaces of the **System.Collections** namespace.

Use slide 33 to explain the code.

Explain to the students that in the code, 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.

Tell that 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.

Mention that 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.

You can refer slide 33 that displays the output of the example.

Slide 34

Understand System.Collections.Generic namespace.

System.Collections.Generic Namespace

Example

- ◆ Consider an online application form used by students to register for an examination conducted by a university.
- ◆ The application form can be used to apply for examination of any course offered by the university.
- ◆ Similarly, in C#, generics allow you to define data structures that consist of functionalities which can be implemented for any data type.
- ◆ Thus, generics allow you to reuse a code for different data types.
- ◆ To create generics, you should use the built-in classes of the System.Collections.Generic namespace. These classes ensure type-checking.
- ◆ To create generics, you should use the built-in classes of the System.Collections.Generic namespace.
- ◆ These classes ensure type-safety, which is a feature of C# that ensures a value is treated as the type with which it is declared.

© Aptech Ltd. Building Applications Using C# / Session 12 34

In slide 34, give the students an example for understanding generics. Tell the students to consider an online application form used by students to register for an examination conducted by a university.

Tell that the application form can be used to apply for examination of any course offered by the university.

Also, mention that in C#, generics define data structures that consist of functionalities which can be implemented for any data type. Thus, generics reuse a code for different data types.

To create generics, 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.

Tips:

The System.Collections.Generic namespace is similar to the System.Collections namespace as both create collections. However, generic collections are type-safe.

Slide 35

Understand classes and interfaces in the System.Collections.Generic namespace.

Classes and Interfaces 1-5

- ◆ The System.Collections.Generic namespace consists of classes and interfaces that define the different generic collections.
 - ❖ Classes:
 - The System.Collections.Generic namespace consists of classes that allow you to create type-safe collections.
 - The following table lists the commonly used classes in the System.Collections.Generic namespace:

Class	Description
List<T>	Provides a generic collection of items that can be dynamically resized
Stack<T>	Provides a generic collection that follows the LIFO principle, which means that the last item inserted in the collection will be removed first
Queue<T>	Provides a generic collection that follows the FIFO principle, which means that the first item inserted in the collection will be removed first
Dictionary<K, V>	Provides a generic collection of keys and values
SortedDictionary<K, V>	Provides a generic collection of sorted key and value pairs that consist of items sorted according to their key
LinkedList<T>	Implements the doubly linked list by storing elements in it

© Aptech Ltd. Building Applications Using C# / Session 12 35

In slide 35, tell the students that the System.Collections.Generic namespace consists of classes and interfaces that define the different generic collections.

Explain that in Classes, the System.Collections.Generic namespace consists of classes that create type-safe collections.

You can refer to the table in slide 35 that lists the commonly used classes in the System.Collections.Generic namespace.

Slide 36

Understand interfaces and structures in the System.Collections.Generic namespace.

Classes and Interfaces 2-5

- ♦ **Interfaces and Structures**
 - The System.Collections.Generic namespace consists of interfaces and structures that can be implemented to create type-safe collections.
 - The following table lists some of the commonly used ones:

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

Use slide 36 to explain that the System.Collections.Generic namespace consists of interfaces and structures that can be implemented to create type-safe collections. You can refer to the table in slide 36 that lists some of the commonly used ones.

Slides 37 to 39

Understand commonly used classes, interfaces, and structures of the System.Collection.Generic namespace.

The following code demonstrates the use of the commonly used classes, interfaces, and structures of the System.Collection.Generic namespace:

Snippet

```
using System;
using System.Collections;
using System.Collections.Generic;
class Student : IEnumerable
{
    LinkedList<string> objList = new LinkedList<string>();
    public void StudentDetails()
    {
        objList.AddFirst("James");
        objList.AddFirst("John");
        objList.AddFirst("Patrick");
        objList.AddFirst("Peter");
        objList.AddFirst("James");
        Console.WriteLine("Number of elements stored in the list: " +
            + objList.Count);
    }
    public void Display(string name)
    {
        LinkedListNode<string> objNode;
        int count = 0;
        for (objNode = objList.First; objNode != null; objNode =
        objNode.Next)
        {
            if (objNode.Value.Equals(name))
            {
                count++;
            }
        }
        Console.WriteLine("The value " + name + " appears " + count +
" times in the list");
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 12 37

Classes and Interfaces 4-5

```

        }
        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 the code:
 - ◆ 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.

© Aptech Ltd. Building Applications Using C# / Session 12 38

Classes and Interfaces 5-5

- ◆ The following figure displays the **System.Collection.Generic** namespace example:

© Aptech Ltd. Building Applications Using C# / Session 12 39

Use slide 37 to show the code for demonstrating the **System.Collection.Generic** namespace.

In slide 38, explain that in the code, 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.

Tell that 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.

Also, mention that 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.

Use slide 39 to refer to the figure that displays the output of the code.

Slide 40

Understand ArrayList class.

ArrayList Class 1-5

- ◆ Following are the features of ArrayList class:

The ArrayList class is a variable-length array, that can dynamically increase or decrease in size. Unlike the Array class, this class can store elements of different data types.

The ArrayList class allows you to specify the size of the collection, during program execution and also allows you to define the capacity that specifies the number of elements an array list can contain.

However, the default capacity of an ArrayList class is 16. If the number of elements in the list reaches the specified capacity, the capacity of the list gets doubled automatically. It can accept null values and can also include duplicate elements.

The ArrayList class allows you to add, modify, and delete any type of element in the list even at run-time.

The elements in the ArrayList can be accessed by using the index position. While working with the ArrayList class, you need not bother about freeing up the memory.

The ArrayList class consists of different methods and properties that are used to add and manipulate the elements of the list.

© Aptech Ltd.

Building Applications Using C# / Session 12 40

In slide 40, explain that 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 specifies the size of the collection, during program execution.

Tell that the ArrayList class defines 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. Also, mention that the ArrayList class adds, modifies, and deletes 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, do not bother about freeing up the memory. The ArrayList class consists of different methods and properties. These methods and properties are used to add and manipulate the elements of the list.

Slide 41

Understand methods and properties.

ArrayList Class 2-5

- ❖ **Methods**
 - The methods of the `ArrayList` class allow you to perform actions such as adding, removing, and copying elements in the list.
 - The following table displays the commonly used methods of the `ArrayList` class:

Method	Description
<code>Add</code>	Adds an element at the end of the list
<code>Remove</code>	Removes the specified element that has occurred for the first time in the list
<code>RemoveAt</code>	Removes the element present at the specified index position in the list
<code>Insert</code>	Inserts an element into the list at the specified index position
<code>Contains</code>	Determines the existence of a particular element in the list
<code>IndexOf</code>	Returns the index position of an element occurring for the first time in the list
<code>Reverse</code>	Reverses the values stored in the <code>ArrayList</code>
<code>Sort</code>	Rearranges the elements in an ascending order
- ❖ **Properties**
 - The properties of the `ArrayList` class allow you to count or retrieve the elements in the list. The following table displays the commonly used properties of the `ArrayList` class:

Property	Description
<code>Capacity</code>	Specifies the number of elements the list can contain
<code>Count</code>	Determines the number of elements present in the list
<code>Item</code>	Retrieves or sets value at the specified position

© Aptech Ltd. Building Applications Using C# / Session 12 41

In slide 41, explain that the methods of the `ArrayList` class performs actions such as adding, removing, and copying elements in the list.

You can refer to the table in slide 41 that displays the commonly used methods of the `ArrayList` class.

Also, tell that the properties of the `ArrayList` class counts or retrieves the elements in the list.

You can refer properties table in slide 41 that displays the commonly used properties of the `ArrayList` class.

Slide 42

Understand the use of the methods and properties of the `ArrayList` class.

ArrayList Class 3-5

- The following code demonstrates the use of the methods and properties of the `ArrayList` class:

```
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 the code:
 - 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
```

© Aptech Ltd. Building Applications Using C# / Session 12 42

In slide 42, explain the code that demonstrates the use of the methods and properties of the `ArrayList` class.

Explain the code and the output.

Tell the students that in the code, 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.

Tell that 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.

When you try referencing an element at a position greater than the list's size, the C# compiler generates an error.

Slide 43

Understand use of methods of the ArrayList class.

The slide has a decorative header with a blue gradient and a subtle circuit board pattern. The title 'ArrayList Class 4-5' is centered in white text. The main content area contains a C# code snippet:

```
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("Fixed Size : " +
        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.Reverse();
        Console.WriteLine("\nList of customers after
reversing:");
        foreach (string names in objCustomers)
        {
```

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 12' in the center, and the number '43' on the right.

In slide 43, explain the code that demonstrates the use of methods of the ArrayList class.

Slide 44

Understand use of methods of the `ArrayList` class.

ArrayList Class 5-5

```

        Console.WriteLine("{0}", names);
    }
    objCustomers.Clear();
    Console.WriteLine("Count after removing all elements
: " + objCustomers.Count);
}
}

```

- ◆ In the code:
 - ◆ 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.
- ◆ The following figure displays the use of methods of the `ArrayList` class:

© Aptech Ltd.

Building Applications Using C# / Session 12 44

In slide 44, explain the code.

Tell the students that in the code, 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.

Tell that the `Sort()` method then displays the values in the sorted order. The `FixedSize()` property checks whether the array is of a fixed size.

Also tell that 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.

Slide 45

Understand the Hashtable class.

Hashtable Class 1-7

- ◆ Consider the reception area of a hotel where you find the keyholder storing a bunch of keys.
- ◆ Each key in the keyholder uniquely identifies a room and thus, each room is uniquely identified by its key.
- ◆ The following figure demonstrates a real-world example of unique keys:

The diagram shows a horizontal row of six keys, each labeled with a room number below it. Above the keys, the word "Keys" is written above a bracket. Below the room numbers, the text "Room Numbers (Values)" is written above another bracket. The keys are labeled 001, 002, 003, 004, 005, and 006 from left to right.

- ◆ Similar to the keyholder, the `Hashtable` class in C# allows you to create collections in the form of keys and values.
- ◆ It generates a hashtable which associates keys with their corresponding values.
- ◆ The `Hashtable` class uses the `hashtable` to retrieve values associated with their unique key.

Use slide 45 to give an example. Tell the students to consider the reception area of a hotel where you find the keyholder storing a bunch of keys. Each key in the keyholder uniquely identifies a room and thus, each room is uniquely identified by its key.

You can refer to figure in slide 45 that demonstrates a real-world example of unique keys. Also tell that similar to the keyholder, the `Hashtable` class in C# helps 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.

Slide 46

Understand the Hashtable class.

Hashtable Class 2-7

- 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 and the code is used to identify the corresponding value of the key.
- The Hashtable object takes the key to search the value, performs a hashing function and generates a hash code for that key.
- When you search for a particular value using the key, the hash code is used as an index to locate the desired record.
- For example, a student name can be used as a key to retrieve the student id and the corresponding residential address. The following figure represents the Hashtable:

Generated Hash Keys (Hash Code)		Employee Records
9862...782		EmpOne Name: John EmpOne Address: Chicago
9862...745		EmpTwo Name: Brett EmpTwo Address: New York
9862...767		EmpThree Name: Steve EmpThree Address: Florida
9862...795		EmpFour Name: Levenda EmpFour Address: California
9862...786		EmpFive Name: Cathy EmpFive Address: New Jersey

Hash Table

Will Retrieve

©Aptech Ltd. Building Applications Using C# / Session 12 46

Use slide 46 to explain that 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.

Tell that the Hashtable object takes the key to search the value, performs a hashing function and generates a hash code for that key. When searching 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.

You can refer to the figure in slide 46 that represents the Hashtable.

Slide 47

Understand methods and properties that are used to add and manipulate the data within the Hashtable.

Hashtable Class 3-7

- ◆ The `Hashtable` class consists of different methods and properties that are used to add and manipulate the data within the hashtable.
- ◆ The methods of the `Hashtable` class allow you to perform certain actions on the data in the hashtable.
- ◆ The following table 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 Hashtable

- ◆ Properties
 - The properties of the `Hashtable` class allow you to access and modify the data in the hashtable.
 - The following figure 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

In slide 47, tell the students that the `Hashtable` class consists of different methods and properties that are used to add and manipulate the data within the hashtable.

Explain that the methods of the `Hashtable` class performs certain actions on the data in the hashtable.

You can refer method's table in slide 47 that displays the commonly used methods of the `Hashtable` class.

Tell that the properties of the `Hashtable` class help to access and modify the data in the hashtable.

You can refer properties table in slide 47 that displays the commonly used properties of the `Hashtable` class.

Slide 48

Understand the code demonstrating methods and properties of the Hashtable class.

The slide has a blue header bar with the title "Hashtable Class 4-7". Below the title is a bulleted list: "The following code demonstrates the use of the methods and properties of the Hashtable class:". A blue button labeled "Snippet" is positioned above the code block. The code itself is a C# program named "HashCollection" with a Main method. It creates a new Hashtable object, adds four key-value pairs ("001": "John", "002": "Peter", "003": "James", "004": "Joe"), prints the count and original keys, changes the value for key "002" to "Patrick", and then prints the modified hashtable. The footer of the slide includes the copyright notice "© Aptech Ltd.", the page number "48", and the slide title "Building Applications Using C# / Session 12".

```
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 slide 48, explain to the students the code that demonstrates the use of the methods and properties of the Hashtable class.

Slide 49

Understand the code demonstrating methods and properties of the Hashtable class.

The screenshot shows a presentation slide with a blue header bar containing the title "Hashtable Class 5-7". Below the header is a section titled "Output" with a dark blue background. The output text is as follows:

```

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

```

Below the output, there is a bulleted list under the heading "In the Code:":

- ◆ In the Code:
 - ◆ 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.

At the bottom of the slide, there is a footer bar with the text "©Aptech Ltd." on the left, "Building Applications Using C# / Session 12" in the center, and the number "49" on the right.

Use slide 49 to explain the code and the output.

Tell that in the code, the `Add()` method inserts the keys and their corresponding values into the instance. The `Count` property displays the number of elements in the hashtable.

Then, tell that 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.

Also, tell that 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`.

Explain that 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.

Slides 50 and 51

Understand code demonstrating methods and properties of the Hashtable class.

Hashtable Class 6-7

- The Code Snippet demonstrates the use of methods and properties of the Hashtable class.

Snippet

```

using System;
using System;
using System.Collections;
class Authors
{
    static void Main(string[] args)
    {
        Hashtable objAuthors = new Hashtable();
        objAuthors.Add("AU01", "John");
        objAuthors.Add("AU04", "Mary");
        objAuthors.Add("AU09", "William");
        objAuthors.Add("AU11", "Rodrick");
        Console.WriteLine("Read-only : " +
        objAuthors.IsReadOnly);
        Console.WriteLine("Count : " + objAuthors.Count);
        IDictionaryEnumerator objCollection =
        objAuthors.GetEnumerator();
        Console.WriteLine("List of authors:\n");
        Console.WriteLine("Author ID \t Name");
        while(objCollection.MoveNext())
        {
            Console.WriteLine(objCollection.Key + "\t" +
            objCollection.Value);
        }
        if(objAuthors.Contains("AU01"))
        {
            Console.WriteLine("\nList contains author with id
            AU01");
        }
        else
        {
            Console.WriteLine("\nList does not contain author
            with id AU01");
        }
    }
}

```

© Aptech Ltd. Building Applications Using C# / Session 12 50

Hashtable Class 7-7

- In the code:
 - 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.
- The following figure displays the Hashtable example:

© Aptech Ltd. Building Applications Using C# / Session 12 51

Use slide 50 to explain another code that demonstrates the use of methods and properties of the `Hashtable` class. In slide 51, explain the code. Tell the students that in the code, 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.

You can refer to the figure on slide 51 that displays the output of the `Hashtable` example.

Slide 52

Understand `SortedList` class.

The slide has a blue header bar with the title "SortedList Class 1-6". The main content area contains a bulleted list of facts about the `SortedList` class:

- ◆ The `SortedList` class represents a collection of key and value pairs where elements are sorted according to the key.
- ◆ By default, the `SortedList` class sorts the elements in ascending order, however, this can be changed if an `IComparable` object is passed to the constructor of the `SortedList` class.
- ◆ These elements are either accessed using the corresponding keys or the index numbers.
- ◆ If you access elements using their keys, the `SortedList` class behaves like a hashtable, whereas if you access elements based on their index number, it behaves like an array.
- ◆ The `SortedList` class consists of different methods and properties that are used to add and manipulate the data in the sorted list.

Below the list, there is a section titled "Methods" with a single bullet point:

- The methods of the `SortedList` class allow you to perform certain actions on the data in the sorted list.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 12", and "52".

Use slide 52 to explain that the `SortedList` class represents a collection of key and value pairs where elements are sorted according to the key.

Tell the students that 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.

Also, mention that if you access elements using their keys, the `SortedList` class behaves like a hashtable, whereas if you access elements based on their index number, it behaves like an array.

Tell that the `SortedList` class consists of different methods and properties that are used to add and manipulate the data in the sorted list.

Explain the methods of the `SortedList` class perform certain actions on the data in the sorted list.

Slide 53

Understand `SortedList` class.

SortedList Class 2-6

- The following table displays the commonly used methods of the `SortedList` class:

Method	Description
<code>Add</code>	Adds an element to the sorted list with the specified key and value
<code>Remove</code>	Removes the element having the specified key from the sorted list
<code>GetKey</code>	Returns the key at the specified index position
<code>GetByIndex</code>	Returns the value at the specified index position
<code>ContainsKey</code>	Checks whether the instance of the <code>SortedList</code> class contains the specified key
<code>ContainsValue</code>	Checks whether the instance of the <code>SortedList</code> class contains the specified value
<code>RemoveAt</code>	Deletes the element at the specified index

- Properties**
 - The properties of the `SortedList` class allow you to access and modify the data in the sorted list.

Property	Description
<code>Capacity</code>	Specifies the number of elements the sorted list can contain
<code>Count</code>	Specifies the number of elements in the sorted list
<code>Item</code>	Returns the value, adds a new value or modifies the existing value for the specified key
<code>Keys</code>	Returns the keys in the sorted list
<code>Values</code>	Returns the values in the sorted list

© Aptech Ltd. Building Applications Using C# / Session 12 53

You can refer to the table in slide 53 that displays the commonly used methods of the `SortedList` class.

Tell them that properties of the `SortedList` helps to access and modify the data in the sorted list.

You can refer to the table in slide 53 that displays the commonly used properties of the `SortedList` class.

Slide 54

Understand the code demonstrating the methods and properties of the `SortedList` class.

SortedList Class 3-6

- The following code demonstrates the use of methods and properties of the `SortedList` class:

Snippet

```

using System;
using System.Collections;

class SortedCollection
{
    static void Main(string[] args)
    {
        SortedList objSortList = new SortedList();
        objSortList.Add("John", "Administration");
        objSortList.Add("Jack", "Human Resources");
        objSortList.Add("Peter", "Finance");
        objSortList.Add("Joel", "Marketing");
        Console.WriteLine("Original values stored in the 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));
        }
    }
}

```

Use slide 54 to explain the code that demonstrates the use of methods and properties of the `SortedList` class.

Slide 55

Understand the code demonstrating the methods and properties of the `SortedList` class.

SortedList Class 4-6

- ◆ In the code:
- ◆ 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

```

© Aptech Ltd. Building Applications Using C# / Session 12 55

In slide 55, explain the code and the output.

Tell that in the code, 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.

Tell that 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.

Mention that the default `Item` property that is invoked using the square bracket notation (`[]`) replaces the values associated with the specified keys, `Peter` and `Jerry`.

Slides 56 and 57

Understand the code that demonstrates the use of methods in the `SortedList` class.

SortedList Class 5-6

- The following code demonstrates the use of methods in the `SortedList` class:

Snippet

```

using System;
using System.Collections;
class Countries
{
    static void Main(string[] args)
    {
        SortedList objCountries = new SortedList();
        objCountries.Add("UK", "United Kingdom");
        objCountries.Add("GER", "Germany");
        objCountries.Add("USA", "United States of America");
        objCountries.Add("AUS", "Australia");
        Console.WriteLine("Read-only : " +
        objCountries.IsReadOnly);
        Console.WriteLine("Count : " + objCountries.Count);
        Console.WriteLine("List of countries:\n");
        Console.WriteLine("Country Code \t Name");
        for (int i = 0; i < objCountries.Count; i++)
        {
            Console.WriteLine(objCountries.GetKey(i) + "\t\t" +
            " " + objCountries.GetByIndex(i));
        }
        objCountries.RemoveAt(1);
        Console.WriteLine("\nList of countries after removing
element at index 1:\n");
        Console.WriteLine("Country Code \t Name");
        for (int i = 0; i < objCountries.Count; i++)
        {
            Console.WriteLine(objCountries.GetKey(i) + "\t\t" +
            " " + objCountries.GetByIndex(i));
        }
    }
}

```

© Aptech Ltd.

Building Applications Using C# / Session 12

56

SortedList Class 6-6

- The following figure displays the `SortedList` class example:

```

C:\WINDOWS\system32\cmd.exe
Read-only : False
Count : 4
List of countries:
Country Code      Name
AUS              Australia
GER              Germany
UK               United Kingdom
USA              United States of America

List of countries after removing element at index 1:
Country Code      Name
AUS              Australia
UK               United Kingdom
USA              United States of America
Press any key to continue . .

```

In the code:

- 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.

© Aptech Ltd.

Building Applications Using C# / Session 12

57

In slide 56, explain the code that demonstrates the use of methods in the `SortedList` class. Tell the students that in the code, the `Add()` method inserts values in the list. The `IsReadOnly()` method checks whether the values in the list can be modified or not.

Also, tell that the `GetByIndex()` method returns the value at the specified index. The `RemoveAt()` method removes the value at the specified index. Explain the output and the code using slide 57.

Slide 58

Understand Dictionary generic class.

Dictionary Generic Class 1-6

- ◆ 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 that consists of a generic collection of elements organized in key and value pairs and maps the keys to their corresponding values.
- ◆ Unlike other collections in the `System.Collections` namespace, it is used to create a collection of a single data type at a time.
- ◆ Every element that you add to the dictionary consists of a value, which is associated with its key and can retrieve a value from the dictionary by using its key.
- ◆ The following syntax declares a `Dictionary` generic class:

Syntax	<code>Dictionary< TKey, TValue ></code>
---------------	---

- ◆ 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.

© Aptech Ltd. Building Applications Using C# / Session 12 58

In slide 58, explain that 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.

Mention that 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.

Tell the students that every element added to the dictionary consists of a value, which is associated with its key. A value from the dictionary can be retrieved by using its key.

Explain the syntax to declare a `Dictionary` generic class as given on the slide.

Tips:

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.

Slide 59

Understand the different methods and properties in the Dictionary generic class.

Dictionary Generic Class 2-6

- The Dictionary generic class consists of different methods and properties that are used to add and manipulate elements in a collection.

Methods

- The methods of the Dictionary generic class allow you to perform certain actions on the data in the collection.
- The following table displays the commonly used methods of the Dictionary generic class:

Method	Description
Add	Adds the specified key and value in the collection
Remove	Removes the value associated with the specified key
ContainsKey	Checks whether the collection contains the specified key
ContainsValue	Checks whether the collection contains the specified value
GetEnumerator	Returns an enumerator that traverses through the Dictionary
GetType	Retrieves the Type of the current instance

Properties

- The properties of the Dictionary generic class allow you to modify the data in the collection.
- The following table 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

© Aptech Ltd. Building Applications Using C# / Session 12 59

Use slide 59 to explain that the Dictionary generic class consists of different methods and properties that are used to add and manipulate elements in a collection.

Explain that the methods of the Dictionary generic class perform certain actions on the data in the collection.

You can refer to the table in slide 59 that displays the commonly used methods of the Dictionary generic class.

Tell that the properties of the Dictionary generic class modify the data in the collection. You can refer to the table in slide 59 that displays the commonly used properties of the Dictionary generic class.

Slide 60

Understand the code demonstrating methods and properties of the Dictionary class.

Dictionary Generic Class 3-6

- The following code demonstrates the use of the methods and properties of the Dictionary class:

Snippet

```

using System;
using System.Collections;
class DictionaryCollection{
    static void Main(string[] args) {
        Dictionary<int, string> objDictionary = new Dictionary<int,
        string>();
        objDictionary.Add(21, "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");
        }
        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 slide 60, explain the code that demonstrates the use of the methods and properties of the Dictionary class.

Slide 61

Understand the code demonstrating methods and properties of the Dictionary class.

Dictionary Generic Class 4-6

- ◆ In the code:
 - ◆ 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
  
```

© Aptech Ltd. Building Applications Using C# / Session 12 61

Use slide 61 to explain the code and the output.

Explain that in the code, 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.

Tell that 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.

Also, mention that 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.

Slide 62

Understand the code that demonstrates the use of methods in Dictionary generic class.

The slide has a blue header bar with the title "Dictionary Generic Class 5-6". Below the title, there is a bullet point: "The following code demonstrates the use of methods in Dictionary generic class:". A "Snippet" button is visible on the left. The main content area contains the following C# code:

```
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\t Values");
        Console.WriteLine("_____");
        IDictionaryEnumerator objDictionaryEnumerator =
        objDictionary.GetEnumerator();
        while (objDictionaryEnumerator.MoveNext ())
        {
            Console.WriteLine(objDictionaryEnumerator.Key.ToString()
            + "\t\t" + objDictionaryEnumerator.Value);
        }
    }
}
```

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 12", and "62".

In slide 62, explain that in the code, the Add () method inserts values into the list. The GetType () method returns the type of the object.

Slide 63

Understand the code that demonstrates the use of methods in Dictionary generic class.

DictionaryGeneric Class 6-6

- ◆ In the code:
 - ◆ The `Add()` method inserts values into the list.
 - ◆ The `GetType()` method returns the type of the object.
- ◆ The following figure displays the use of Dictionary generic class:

The screenshot shows a Windows Command Prompt window titled 'DictionaryGeneric Class 6-6'. The command entered is 'Dictionary'. The output displays the following information:

```
c:\WINDOWS\system32\cmd.exe
Dictionary class contains values of type
System.Collections.Generic.Dictionary`2[System.Int32,System.String]

Keys          Values
201          Gear Box
220          Oil Filter
330          Engine
385          Radiator
393          Steering

Press any key to continue . . .
```

The window has a standard Windows title bar and taskbar at the bottom. The taskbar includes icons for Start, Task View, File Explorer, and others.

In slide 63, explain the code. You can refer to the figure in slide 63 that displays the use of Dictionary generic class.

Slides 64 and 65

Understand collection initializers.

Collection Initializers 1-2

- Collection initializers allow adding elements to the collection classes of the `System.Collections` and `System.Collections.Generic` namespaces that implements the `IEnumerable` interface using element initializers.
- The element initializers can be a simple value, an expression, or an object initializer.
- When a programmer uses collection initializer, the programmer is not required to provide multiple `Add()` methods to add elements to the collection, making the code concise. It is the responsibility of the compiler to provide the `Add()` methods when the program is compiled.
- The following code uses a collection initializer to initialize an `ArrayList` with integers:

Snippet

```
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);
        }
    }
}
```

Output

```
1
2
18
4
5
```

© Aptech Ltd. Building Applications Using C# / Session 12 64

Collection Initializers 2-2

- In the code:
 - The `Main()` method uses collection initializer to create an `ArrayList` object initialized with integer values and expressions that evaluates to integer values.
 - As collection often contains objects, collection initializers accept object initializers to initialize a collection.
 - The following code shows a collection initializer that initializes a generic `Dictionary` object with an integer keys and `Employee` objects:

Snippet

```
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"}}
        };
    }
}
```

- In the code:
 - The `Employee` class consists of 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.

© Aptech Ltd. Building Applications Using C# / Session 12 65

In slide 64, explain that collection initializers allow adding elements to the collection classes of the `System.Collections` and `System.Collections.Generic` namespaces that implements the `IEnumerable` interface using element initializers.

Tell the students that the element initializers can be a simple value, an expression, or an object initializer. When a programmer uses collection initializer, the programmer is not required to provide multiple `Add()` methods to add elements to the collection, making the code concise.

Also, mention that it is the responsibility of the compiler to provide the `Add()` methods when the program is compiled.

In slide 65, explain the code that uses a collection initializer to initialize an `ArrayList` with integers.

Explain that in the code, the `Main()` method uses collection initializer to create an `ArrayList` object initialized with integer values and expressions that evaluates to integer values.

Mention that as collection often contains objects, collection initializers accept object initializers to initialize a collection.

Explain that the code shows a collection initializer that initializes a generic `Dictionary` object with an integer keys and `Employee` objects.

Tell that code 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.

Also, tell that for each element, added to the collection, the innermost pair of braces encloses the object initializer of the `Employee` class.

Additional Information

Refer the following links for more information on collection initializers:

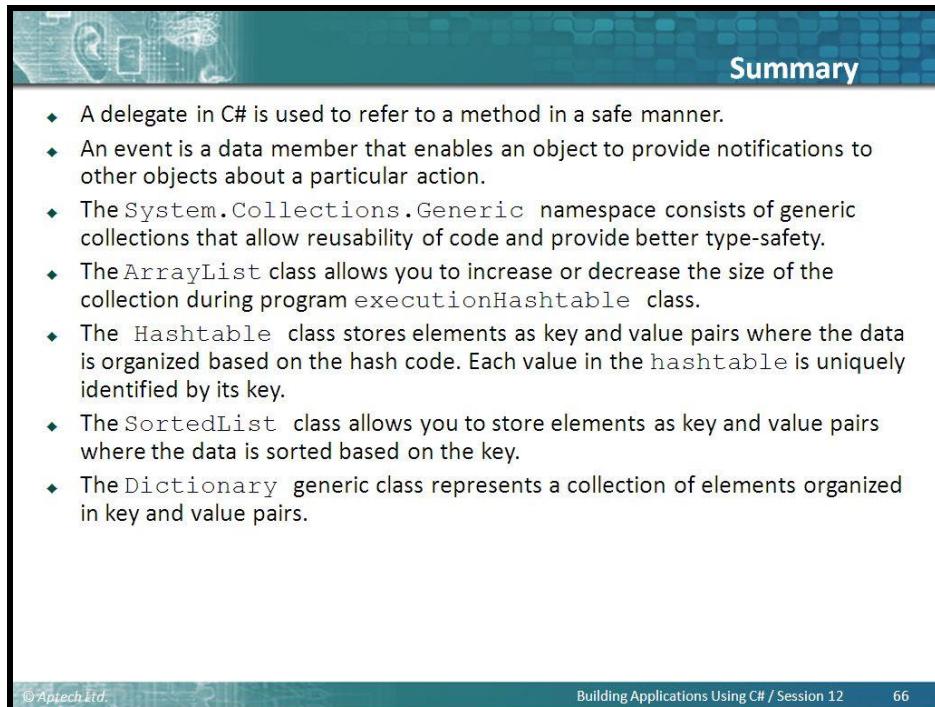
<http://msdn.microsoft.com/en-us/library/bb384062.aspx>

<http://msdn.microsoft.com/en-us/library/bb531208.aspx>

http://msmvps.com/blogs/jon_skeet/archive/2013/02/14/fun-with-object-and-collection-initializers.aspx

Slide 66

Summarize the session.



Summary

- ◆ A delegate in C# is used to refer to a method in a safe manner.
- ◆ An event is a data member that enables an object to provide notifications to other objects about a particular action.
- ◆ The `System.Collections.Generic` namespace consists of generic collections that allow reusability of code and provide better type-safety.
- ◆ The `ArrayList` class allows you to increase or decrease the size of the collection during program execution.
- ◆ The `Hashtable` class stores elements as key and value pairs where the data is organized based on the hash code. Each value in the `Hashtable` is uniquely identified by its key.
- ◆ The `SortedList` class allows you to store elements as key and value pairs where the data is sorted based on the key.
- ◆ The `Dictionary` generic class represents a collection of elements organized in key and value pairs.

In slide 66, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- In C#, delegates can be used to refer a method in a safer 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 allows code reusability.
- The `ArrayList` class increases or decreases 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 stores 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.

12.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions

You can also put a few questions to students to search additional information, such as:

1. How to pass a collection of delegates to run into a method?
2. How to invoke an event?
3. List the kinds of collections.

Session 13 - Generics and Iterators

13.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the previous session for a review. Prepare the background knowledge/summary to be discussed with students in the class. The summary of the previous session is as follows:

- A delegate in C# is used to refer to a method in a safe manner.
- An event is a data member that enables an object to provide notifications to other objects about a particular action.
- The `System.Collections.Generic` namespace consists of generic collections that allow reusability of code and provide better type-safety.
- The `ArrayList` class allows you to increase or decrease the size of the collection during program execution.
- The `Hashtable` class stores elements as key and value pairs where the data is organized based on the hash code. Each value in the hash table is uniquely identified by its key.
- The `SortedList` class allows you to store elements as key and value pairs where the data is sorted based on the key.
- The `Dictionary` generic class represents a collection of elements organized in key and value pairs.

Familiarize yourself with the topics of this session in-depth.

13.1.1 Objectives

By the end of this session, the learners will be able to:

- Define and describe generics
- Explain creating and using generics
- Explain iterators

13.1.2 Teaching Skills

To teach this session successfully, you must know the term Generics and Iterators.

You should be aware of the use of generics and iterators.

You should teach the concepts in the theory class using the concepts, tables, and codes provided.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives.

Show the students slide 2 of the presentation. Tell them that they will be introduced the concept of Generics.

They will also learn about the Iterators.

This session will also discuss creating and using generics.

13.2 In-Class Explanations

Slide 3

Understand the concept of generics.

The slide has a blue header bar with the word 'Generics' on the right. The main content area has a teal background with a grid pattern. It contains two bulleted lists under different sections: 'Example' and 'Consider a C# program that uses an array variable of type Object to store a collection of student names.'

- ◆ Generics are a kind of parameterized data structures that can work with value types as well as reference types.
- ◆ You can define a class, interface, structure, method, or a delegate as a generic type in C#.

Example

- ◆ Consider a C# program that uses an array variable of type Object to store a collection of student names.
- ◆ The names are read from the console as value types and are boxed to enable storing each of them as type Object.
- ◆ In this case, the compiler cannot verify the data stored against its data type as it allows you to cast any value to and from Object.
- ◆ If you enter numeric data, it will be accepted without any verification.
- ◆ To ensure type-safety, C# introduces generics, which has a number of features including the ability to allow you to define generalized type templates based on which the type can be constructed later.

© Aptech Ltd. Generics and Iterators / Session 13 3

In slide 3, introduce the concept of generics to the students. To achieve high level of code reusability in other languages, programmers typically use a feature called generics. Using generics, class templates can be created that support any type. After instantiating that class, the type that want to be used is to be specified, and from that point on, object is "locked in" to the type that has been specified.

Explain to the students that generics are a kind of parameterized data structures that can work with value types as well as reference types. Tell them that they can define a class, interface, structure, method, or a delegate as a generic type in C#.

To understand the generics, give the following example to the students:

Tell them to consider a C# program that uses an array variable of type `Object` to store a collection of student names. Tell them that 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, mention them that the compiler cannot verify the data stored against its data type as it allows you to cast any value to and from `Object`. If you enter numeric data, it will be accepted without any verification.

Mention that to ensure type-safety, C# introduces generics, which has a number of features including the ability to allow you to define generalized type templates based on which the type can be constructed later.

Additional Information

For more information on generics in C#, refer the following links:

<http://msdn.microsoft.com/en-us/library/ms379564%28v=vs.80%29.aspx>
<http://msdn.microsoft.com/en-us/library/512aeb7t.aspx>

Slides 4 to 6

Understand namespaces, classes, and interfaces for generics.

Namespaces, Classes, and Interfaces for Generics 1-3

- There are several namespaces in the .NET Framework that facilitate creation and use of generics which are as follows:

System.Collections.ObjectModel	<ul style="list-style-type: none"> This allows you to create dynamic and read-only generic collections.
System.Collections.Generic	<ul style="list-style-type: none"> The namespace consists of classes and interfaces that allow you to define customized generic collections.

- Classes:**
 - The `System.Collections.Generic` namespace consists of classes that allow you to create type-safe collections.

Namespaces, Classes, and Interfaces for Generics 2-3

- The following table lists some of the widely used classes of the `System.Collections.Generic` namespace:

Class	Descriptions
Comparer	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the <code>IComparer</code> interface
Dictionary.KeyCollection	Consists of keys present in the instance of the <code>Dictionary</code> class
Dictionary.ValueCollection	Consists of values present in the instance of the <code>Dictionary</code> class
EqualityComparer	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the <code>IEqualityComparer</code> interface

Namespaces, Classes, and Interfaces for Generics 3-3

- ◆ **Interfaces**
 - ❖ The `System.Collections.Generic` namespace consists of interfaces that allow you to create type-safe collections.
- ◆ The following table lists some of the widely used interfaces of the `System.Collections.Generic` namespace:

Interface	Descriptions
<code>IComparer</code>	Defines a generic method <code>Compare()</code> that compares values within a collection
<code>IEnumerable</code>	Defines a generic method <code>GetEnumerator()</code> that iterates over a collection
<code>IEqualityComparer</code>	Consists of methods which check for the equality between two objects

In slide 4, tell the students that there are several namespaces in the .NET Framework that facilitate creation and use of generics.

Explain that the `System.Collections.ObjectModel` namespace allows them to create dynamic and read-only generic collections.

Explain that the `System.Collections.Generic` namespace consists of classes and interfaces that allow them to define customized generic collections. Then, explain them about classes in the namespace.

Explain that the `System.Collections.Generic` namespace consists of classes that allow them to create type-safe collections.

As the modules in projects get more complicated, programmers need a means to give better code reusability and customize their existing component-based software.

In slide 5, explain some of the widely used classes of the `System.Collections.Generic` namespace such as `Comparer`, `Dictionary.KeyCollection`, `Dictionary.ValueCollection`, and `EqualityComparer`.

Explain these classes as shown in the table on slide 5.

In slide 6, tell the students that the `System.Collections.Generic` namespace consists of interfaces that allow creating type-safe collections.

Then, refer to the table on slide 6 and explain some of the commonly used interfaces of the `System.Collections.Generic` namespace such as `IComparer`, `IEnumerable`, and `IEqualityComparer`.

With this slide, you will finish explaining the namespaces, classes, and interfaces for generics.

Slide 7

Understand the System.Collections.ObjectModel namespace.

System.Collections.ObjectModel 1-5

- The System.Collections.ObjectModel namespace consists of classes that can be used to create customized generic collections.
- The following table shows the classes contained in the System.Collections.ObjectModel namespace:

Classes	Descriptions
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

Generics and Iterators / Session 13 7

In slide 7, explain the System.Collections.ObjectModel to the students.

Tell them that the System.Collections.ObjectModel namespace consists of classes that can be used to create customized generic collections.

Then, refer to the table on the slide and explain the classes contained in the System.Collections.ObjectModel namespace to the students.

Additional Information

For more information on System.Collections.ObjectModel namespace in C#, refer the following link:

<http://msdn.microsoft.com/en-us/library/system.collections.objectmodel%28v=vs.110%29.aspx>

Slides 8 to 10

Understand the System.Collections.ObjectModel namespace.

System.Collections.ObjectModel 2-5

- The following code demonstrates the use of the `ReadOnlyCollection<T>` class:

Snippet

```
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("Michael");
        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();
```

System.Collections.ObjectModel 3-5

```
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);
    }
}
```

System.Collections.ObjectModel 4-5

- ◆ In the code:
 - ◆ 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.

In slide 8, explain the code demonstrating the use of the **ReadOnlyCollection<T>** class.

In slide 9, explain the further part of the code demonstrating the use of the **ReadOnlyCollection<T>** class.

Explain the students that in the code, the Main() method of the **ReadOnly** class creates an instance of the List class. Tell them that the Add() method inserts elements in the instance of the List class. Tell them that 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.

Describe that 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.

Using slide 10, explain them that the code creates an array variable that is twice the size of the **ReadOnlyCollection** class. Mention that the CopyTo() method copies the elements from the **ReadOnlyCollection** class to the array variable from the fifth position onwards.

Slide 11

Understand the System.Collections.ObjectModel namespace.

◆ The following figure displays the output of the code:

Output

```
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
null
Francis
James
Peter
Baptista
Micheal
Press any key to continue . . .

```

© Aptech Ltd. Generics and Iterators / Session 13 11

Refer to the figure in slide 11 to explain the output of the code to the students.

With this slide, you will finish explaining the System.Collections.ObjectModel.

Slide 12

Understand the features of a generic declaration.

The slide has a teal header bar with the title 'Creating Generic Types'. Below the header, there is a bullet point: 'Following are the features of a generic declaration:' followed by four descriptive boxes arranged in a 2x2 grid. The top-left box is purple and states: 'A generic declaration always accepts a **type parameter**, which is a placeholder for the required data type.' The top-right box is also purple and states: 'The type is specified only when a generic type is referred to or constructed as a type within a program.' The bottom-left box is blue and states: 'The process of creating a generic type begins with a generic type definition containing type parameters that acts like a blueprint.' The bottom-right box is teal and states: '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.' At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.', 'Generics and Iterators / Session 13', and '12'.

- ◆ Following are the features of a generic declaration:

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 that 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.

In slide 12, explain the generic declaration to the students.

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

Then, explain that the process of creating a generic type begins with a generic type definition containing type parameters. The definition acts like a blueprint. Later, mention that 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.

Slide 13

Explain the benefits of creating generics.

Benefits
<ul style="list-style-type: none"> ◆ Generics ensure type-safety at compile-time. ◆ Generics allow you to reuse the code in a safe manner without casting or boxing. ◆ A generic type definition is reusable with different types but can accept values of a single type at a time. ◆ Apart from reusability, there are several other benefits of using generics. These are as follows: <ul style="list-style-type: none"> ❖ Improved performance because of low memory usage as no casting or boxing operation is required to create a generic ❖ Ensured strongly-typed programming model ❖ Reduced run-time errors that may occur due to casting or boxing 

© Aptech Ltd.

Generics and Iterators / Session 13

13

In slide 13, explain the benefits of creating generics to the students.

Tell the students that the generics ensure type-safety at compile-time. Tell them that the generics allow users to reuse the code in a safe manner without casting or boxing. Mention that a generic type definition is reusable with different types but can accept values of a single type at a time.

Explain the students that apart from reusability, there are several other benefits of using generics such as:

- Improved performance because of low memory usage as no casting or boxing operation is required to create a generic
- Ensured strongly-typed programming model
- Reduced run-time errors that may occur due to casting or boxing

Slide 14

Understand generic classes.

Generic Classes 1-5

- ◆ Generic classes define functionalities that can be used for any data type and are declared with a class declaration followed by a **type parameter** enclosed within angular brackets.
- ◆ While declaring a generic class, you can apply some restrictions or constraints to the type parameters by using the **where** keyword. However, applying constraints to the type parameters is optional.
- ◆ Thus, while creating a generic class, you must generalize the data types into the type parameter and optionally decide the constraints to be applied on the type parameter.
- ◆ The following syntax is used for creating a generic class:

Syntax

```
<access modifier> class <ClassName><<type parameter list>> [where <type parameter constraint clause>]
```

- ◆ **where**,
 - ❖ **access_modifier**: Specifies the scope of the generic class. It is optional.
 - ❖ **ClassName**: Is the name of the new generic class to be created.
 - ❖ **<type parameter list>**: Is used as a placeholder for the actual data type.
 - ❖ **type parameter constraint clause**: Is an optional class or an interface applied to the type parameter with the **where** keyword.

In slide 14, explain generic classes to the students.

Tell the students that the generic classes define functionalities that can be used for any data type and they are declared with a class declaration followed by a **type parameter** enclosed within angular brackets.

Tell them that while declaring a generic class, a user can apply some restrictions or constraints to the type parameters by using the **where** keyword. However, applying constraints to the type parameters is optional.

Mention that while creating a generic class, user must generalize the data types into the type parameter and optionally decide the constraints to be applied on the type parameter.

Then, explain the syntax used for creating a generic class.

Tell the students that in the syntax, **<access_modifier>** specifies the scope of the generic class. It is optional and **ClassName** is the name of the new generic class to be created. Tell them that **<type parameter list>** is used as a placeholder for the actual data type and **<type parameter constraint clause>** is an optional class or an interface applied to the type parameter with the **where** keyword.

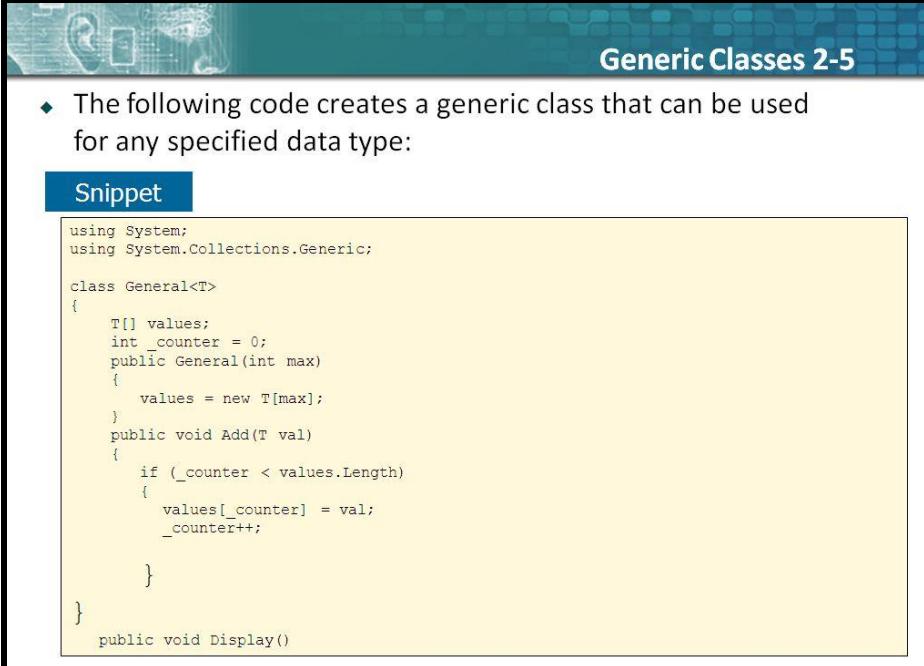
Additional Information

For more information on generic classes, refer the following link:

<http://msdn.microsoft.com/en-us/library/sz6zd40f.aspx>

Slides 15 to 18

Understand generic classes.



The slide is titled "Generic Classes 2-5". It contains a "Snippet" section with the following C# code:

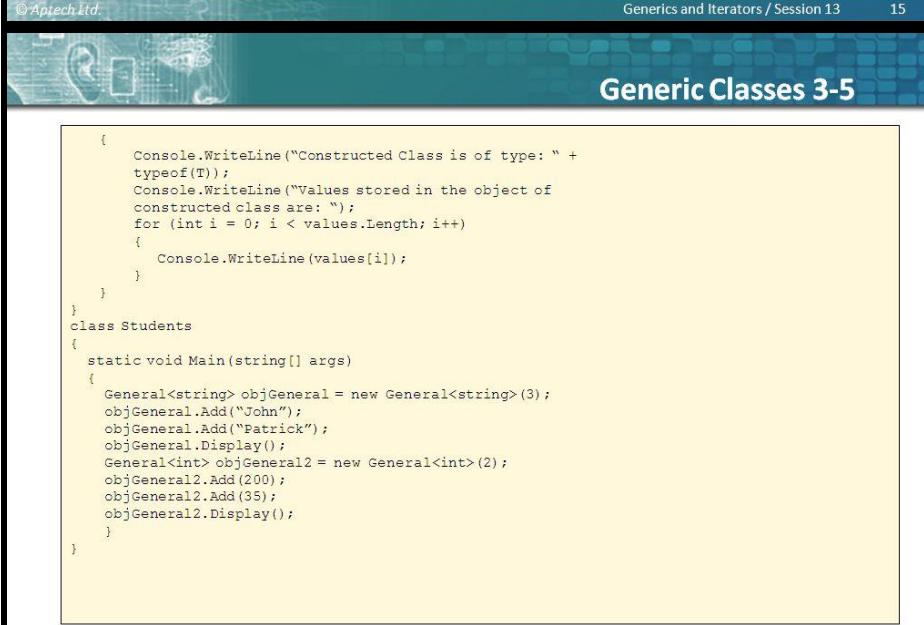
```

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()
    {
        for (int i = 0; i < values.Length; i++)
        {
            Console.WriteLine(values[i]);
        }
    }
}

```

At the bottom of the slide, it says "Generics and Iterators / Session 13" and "15".



The slide is titled "Generic Classes 3-5". It contains the following C# code:

```

{
    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();
    }
}

```

At the bottom of the slide, it says "Generics and Iterators / Session 13" and "16".

Generic Classes 4-5

- ◆ In the code:
 - ◆ A generic class definition for **General** is created that takes a type parameter **T**.
 - ◆ The generic class declares a parameterized constructor with an **int** value.
 - ◆ The **Add()** method takes a parameter of the same type as the generic class.
 - ◆ The method **Display()** displays the value type specified by the type parameter and the values supplied by the user through the object.
 - ◆ The **Main()** method of the class **Students** creates an instance of the generic class **General** by providing the type parameter value as **string** and **total** value to be stored as **3**.
 - ◆ This instance invokes the **Add()** method which takes student names as values.
 - ◆ These student names are displayed by invoking the **Display()** method.
 - ◆ Later, another object is created of a different data type, **int**, based on the same class definition.
 - ◆ The class definition is generic, we need not change the code now, but can reuse the same code for an **int** data type as well.
 - ◆ Thus, using the same generic class definition, we can create two different lists of data.

©Aptech Ltd.

Generics and Iterators / Session 13

17

Generic Classes 5-5

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
```

©Aptech Ltd.

Generics and Iterators / Session 13

18

Using slide 15 to 18, explain the code and output that creates a generic class that can be used for any specified data type to the students.

Explain the students that a generic class definition for **General** is created that takes a type parameter **T**. Tell them that the generic class declares a parameterized constructor with an **int** value.

Tell them that the **Add()** method takes a parameter of the same type as the generic class. The method **Display()** displays the value type specified by the type parameter and the values

supplied by the user through the object. The `Main()` method of the class `Students` creates an instance of the generic class `General` by providing the type parameter value as `string` and `total` values to be stored as 3.

Explain that this instance invokes the `Add()` method which takes student names as values. These student names are displayed by invoking the `Display()` method. Later, explain them that another object is created of a different data type, `int`, based on the same class definition. Mention that the class definition is generic, we need not change the code now, but can reuse the same code for an `int` data type as well. Thus, using the same generic class definition, we can create two different lists of data.

Explain the output shown in slide 18 to the students. With this slide, you will finish explaining generic classes.

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.

In-Class Question:

After you finish explaining generic classes, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are Generics?

Answer:

Generics are a kind of parameterized data structures that can work with value types as well as reference types.

Slide 19

Understand constraints on type parameters.



Constraints on Type Parameters 1-4

- ◆ You can apply constraints on the type parameter while declaring a generic type.
- ◆ A constraint is a restriction imposed on the data type of the type parameter and 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.
- ◆ The following table lists the types of constraints that can be applied to the type parameter:

Constraints	Descriptions
<code>T : struct</code>	Specifies that the type parameter must be of a value type only except the null value
<code>T : class</code>	Specifies that the type parameter must be of a reference type such as a class, interface, or a delegate
<code>T : new()</code>	Specifies that the type parameter must consist of a constructor without any parameter which can be invoked publicly
<code>T : <base class name></code>	Specifies that the type parameter must be the parent class or should inherit from a parent class
<code>T : <interface name></code>	Specifies that the type parameter must be an interface or should inherit an interface

© Aptech Ltd. Generics and Iterators / Session 13 19

In slide 19, explain the constraints on type parameters.

Tell the students that they can apply constraints on the type parameter while declaring a generic type. Tell them that a constraint is a restriction imposed on the data type of the type parameter.

Mention that the constraints are specified using the `where` keyword and 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.

Then, refer to the table on the slide to explain the types of constraints that can be applied to the type parameter.

Slides 20 to 22

Understand the code to create a generic class that uses a class constraint.

Constraints on Type Parameters 2-4

The following code creates a generic class that uses a class constraint:

Snippet

```
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;
        }
    }
}
```

©Aptech Ltd.

Generics and Iterators / Session 13

20

Constraints on Type Parameters 3-4

```
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();
    }
}
```

©Aptech Ltd.

Generics and Iterators / Session 13

21



Constraints on Type Parameters 4-4

In the code:

- ❖ The class **GenericList** is created that takes a type parameter **T**.
- ❖ This type parameter is applied a class constraint, which means the type parameter can only include details of the **Employee** type.
- ❖ The generic class creates an array variable with the type parameter **T**, which means it can include values of type **Employee**.
- ❖ The **Add ()** method consists of a parameter **val**, which will contain the values set in the **Main ()** method.
- ❖ Since, the type parameter should be of the **Employee** type, the constructor is called while setting the values in the **Main ()** method.

Output

```
John, 100
James, 200
Patrich, 300
```

© Aptech Ltd.

Generics and Iterators / Session 13 22

In slide 20 and 21, explain the code used to create a generic class that uses a class constraint. Tell the students that the class **GenericList** is created that takes a type parameter **T**. This type parameter can only include details of the **Employee** type.

Tell them that 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.

Then, describe the output of the code using slide 22.

Explain the following output of the code to the students:

John, 100
 James, 200
 Patrich, 300

Tips:

When you use a certain type as a constraint, the type used as a constraint must have greater scope of accessibility than the generic type that will be using the constraint.

With this slide, you will finish explaining constraints on type parameters.

Slide 23

Understand generic class inheritance.

Inheriting Generic Classes 1-2

- ◆ A generic class can be inherited same as any other non-generic class in C# and can act both as a base class or a derived class.
- ◆ While inheriting a generic class in another generic class, you can use the generic type parameter of the base class instead of passing the data type of the parameter.
- ◆ However, while inheriting a generic class in a non-generic class, you must provide the data type of the parameter instead of the base class generic type parameter.
- ◆ The constraints imposed at the base class level must be included in the derived generic class.
- ◆ The following figure displays a generic class as 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>
{
}

```

© Aptech Ltd. Generics and Iterators / Session 13 23

In slide 23, explain the process of inheriting generic classes to the students.

Tell them that 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.

Explain them that while inheriting a generic class in another generic class, they can use the generic type parameter of the base class instead of passing the data type of the parameter.

Mention that while inheriting a generic class in a non-generic class, the students 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.

Refer to the figure in slide 23 that displays a generic class as base class.

Additional Information

For more information on generic type parameters, refer the following link:

<http://msdn.microsoft.com/en-us/library/0zk36dx2.aspx>

Slide 24

Understand generic class inheritance.

Inheriting Generic Classes 2-2

- The following syntax is used to inherit a generic class from an existing generic class:

Syntax

```
<access_modifier> class <BaseClass><<generic type parameter>>>{}  
<access_modifier> class <DerivedClass> : <BaseClass><<generic type parameter>>>{}
```

where,

- * access_modifier: Specifies the scope of the generic class.
- * BaseClass: Is the generic base class.
- * <generic type parameter>: Is a placeholder for the specified data type.
- * DerivedClass: Is the generic derived class.

- The following syntax is used to inherit a non-generic class from a generic class:

Syntax

```
<access_modifier> class <BaseClass><<generic type parameter>>>{}  
<access_modifier> class <DerivedClass> : <BaseClass><<type parameter value>>>{}
```

where,

- * <type parameter value>: Can be a data type such as int, string, or float.

Using slide 24, explain the syntax that is used to inherit a generic class from an existing generic class.

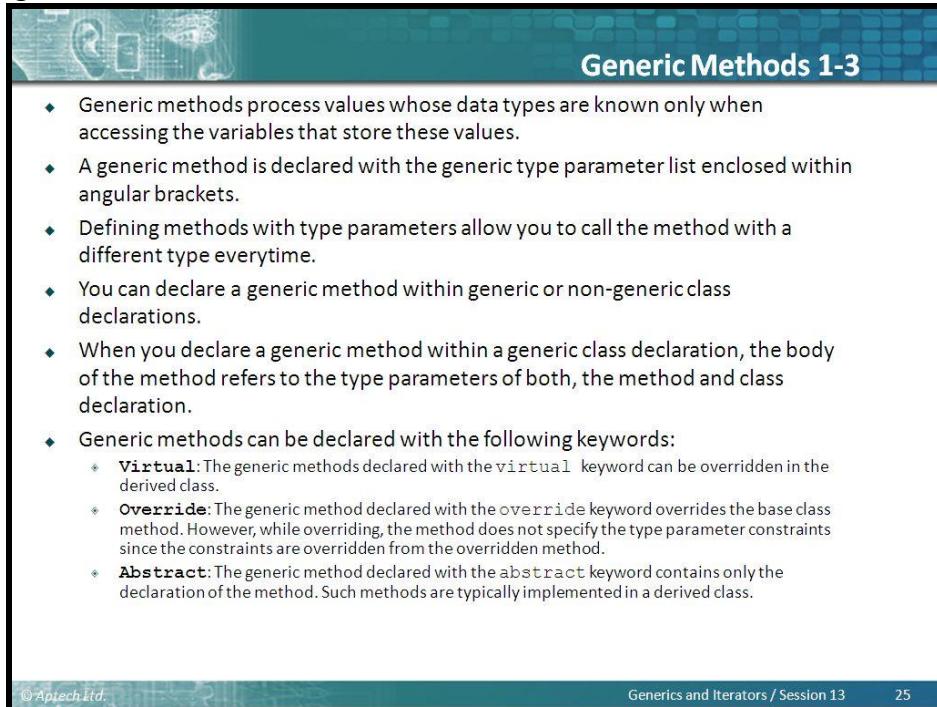
Tell the students that `access_modifier` specifies the scope of the generic class, `BaseClass` is the generic base class, `<generic type parameter>` is a placeholder for the specified data type, and `DerivedClass` is the generic derived class.

Explain the syntax that is used to inherit a non-generic class from a generic class.

Tell the students that in the syntax, `<type parameter value>` can be a data type such as `int`, `string`, or `float`.

Slide 25

Understand generic methods.



Generic Methods 1-3

- ◆ Generic methods process values whose data types are known only when accessing the variables that store these values.
- ◆ A generic method is declared with the generic type parameter list enclosed within angular brackets.
- ◆ Defining methods with type parameters allow you to call the method with a different type everytime.
- ◆ You can declare a generic method within generic or non-generic class declarations.
- ◆ When you declare a generic method within a generic class declaration, the body of the method refers to the type parameters of both, the method and class declaration.
- ◆ Generic methods can be declared with the following keywords:
 - ◆ **Virtual:** The generic methods declared with the `virtual` keyword can be overridden in the derived class.
 - ◆ **Override:** The generic method declared with the `override` keyword overrides the base class method. However, while overriding, the method does not specify the type parameter constraints since the constraints are overridden from the overridden method.
 - ◆ **Abstract:** The generic method declared with the `abstract` keyword contains only the declaration of the method. Such methods are typically implemented in a derived class.

© Aptech Ltd. Generics and Iterators / Session 13 25

In slide 25, explain generic methods to the students.

Tell the students that the generic methods process values whose data types are known only when accessing the variables that store these values. Tell them that a generic method is declared with the generic type parameter list enclosed within angular brackets. Defining methods with type parameters allow user to call the method with a different type every time.

Explain them that they can declare a generic method within generic or non-generic class declarations. Mention that when they declare a generic method within a generic class declaration, the body of the method refers to the type parameters of both, the method and class declaration.

Then tell the students that the Generic methods can be declared with keywords such as `virtual`, `override`, and `abstract`.

Explain the `virtual` keyword. Tell the students that the generic methods declared with the `virtual` keyword can be overridden in the derived class.

Explain the `override` keyword. Tell the students that the generic method declared with the `override` keyword overrides the base class method. However, while overriding, the method does not specify the type parameter constraints since the constraints are overridden from the overridden method.

Explain the `abstract` keyword. Tell the students that the generic method declared with the `abstract` keyword contains only the declaration of the method. Such methods are typically implemented in a derived class.

Additional Information

For more information on generic methods, refer the following link:

<http://msdn.microsoft.com/en-us/library/twcad0zb.aspx>

Slide 26

Understand generic methods.

The slide has a teal header bar with the title "Generic Methods 2-3". Below the header, there is a list of bullet points. The first bullet point is under a "Syntax" heading, which is enclosed in a blue box. The syntax code is shown in a yellow box:
`<access_modifier><return_type><MethodName><<type parameter list>>`

The remaining bullet points are not under any specific heading and are listed below the syntax section:

- ◆ The following syntax is used for declaring a generic method:
- ◆ 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.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Generics and Iterators / Session 13", and the number "26".

In slide 26, refer the syntax that used for declaring a generic method and tell the students that `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, and `<type parameter list>` is used as a placeholder for the actual data type.

Slide 27

Understand generic methods.

Generic Methods 3-3

- The following code creates a generic method within a non-generic class:

Snippet

```
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 the code:
 - 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
```

© Aptech Ltd. Generics and Iterators / Session 13 27

In slide 27, explain the code that creates a generic method within a non-generic class to the students.

Tell the students that in the code, 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**.

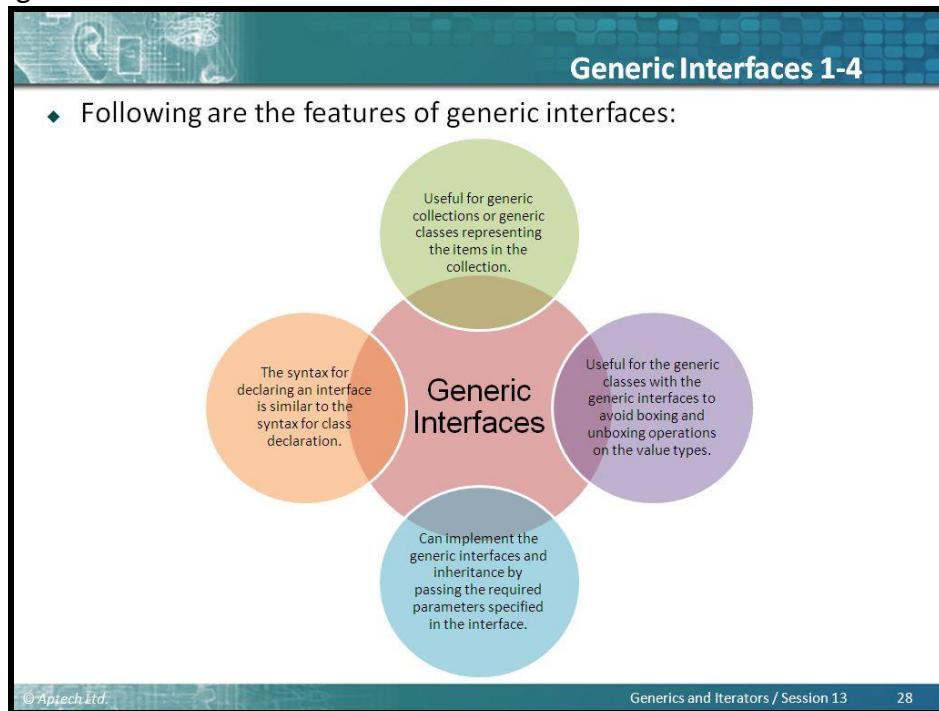
Tell them that the **Swap ()** method creates a variable **temp** of type **T** that is assigned the value within the variable **valOne**.

Mention that 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.

Then, explain the output of the code to the students as given on the slide.

Slide 28

Understand generic interfaces.



In slide 28, explain generic interfaces to the students.

Tell the students that the generic interfaces are useful for generic collections or generic classes representing the items in the collection.

Tell them that they can use the generic classes with the generic interfaces to avoid boxing and unboxing operations on the value types. Explain them that generic classes can implement the generic interfaces by passing the required parameters specified in the interface.

Mention that similar to generic classes, generic interfaces also implement inheritance.

Additional Information

For more information on generic interfaces, refer the following link:

<http://msdn.microsoft.com/en-us/library/kwtft8ak.aspx>

Slide 29

Understand generic interfaces.

The slide has a blue header bar with the title "Generic Interfaces 2-4". Below the header, there is a bulleted list: "◆ The following syntax is used for creating a generic interface:". A yellow callout box labeled "Syntax" contains the interface declaration 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.

© Aptech Ltd. Generics and Iterators / Session 13 29

In slide 29, refer the syntax and tell the students that `access_modifier` specifies the scope of the generic interface and `InterfaceName` is the name of the new generic interface. Explain that the syntax for declaring an interface is similar to the syntax for class declaration.

Tell them that `<type parameter list>` is used as a placeholder for the actual data type and `type parameter constraint clause` is an optional class or an interface applied to the type parameter with the `where` keyword.

Additional Information

For more information on generic interfaces, refer the following link:

<http://msdn.microsoft.com/en-us/library/kwtft8ak.aspx>

Slides 30 and 31

Understand generic interfaces.

Generic Interfaces 3-4

- The following code creates a generic interface that is implemented by the non-generic class:

Snippet

```
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));
    }
}
```

©Aptech Ltd.

Generics and Iterators / Session 13

30

Generic Interfaces 4-4

- In the code:
 - 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
```

©Aptech Ltd.

Generics and Iterators / Session 13

31

Using slides 30 and 31, explain the code that creates a generic interface that is implemented by the non-generic class. Tell the students that in the code, the generic interface **IMaths** takes a type parameter **T** and declares two methods of type **T**. Tell them that 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.

Mention that the `Main()` method creates an instance of the class **Numbers** and displays the addition and subtraction of two numbers.

Then, explain the following output of the code to the students:

Addition of two integer values is: 68

Subtraction of two integer values is: 22

Slide 32

Explain generic interface constraints.

Generic Interface Constraints 1-4

- ◆ You can specify an interface as a constraint on a type parameter to enable the members of the interface within, to use the generic class.
- ◆ In addition, it ensures that only the types that implement the interface are used and also specify multiple interfaces as constraints on a single type parameter.

© Aptech Ltd. Generics and Iterators / Session 13 32

In slide 32, explain the generic interface constraints to the students.

Tell the students that they can specify an interface as a constraint on a type parameter. Tell them that this enables them 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.

Mention that the students can also specify multiple interfaces as constraints on a single type parameter.

All the classes and methods from generics combine reusability, type safety and efficiency.

It is recommended that all applications that target the .NET Framework 2.0 and later use the new generic collection classes instead of the older non-generic counterparts such as `ArrayList`.

Slides 33 and 34

Explain generic interface constraints.

Generic Interface Constraints 2-4

- The following code creates a generic interface that is used as a constraint on a generic class:

Snippet

```
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);
    }
}
```

Generic Interface Constraints 3-4

```
class GenericList<T> where T : IDetails
{
    T[] _values = new T[3];
    int _counter = 0;
    public void Add(T val)
    {
        _values[_counter] = val;
        _counter++;
    }
    public void Display()
    {
        for (int i = 0; i < 3; i++)
        {
            _values[i].GetDetails();
        }
    }
}
class InterfaceConstraintDemo
{
    static void Main(string[] args)
    {
        GenericList<Student> objList = new GenericList<Student>();
        objList.Add(new Student("Wilson", 120));
        objList.Add(new Student("Jack", 130));
        objList.Add(new Student("Peter", 140));
        objList.Display();
    }
}
```

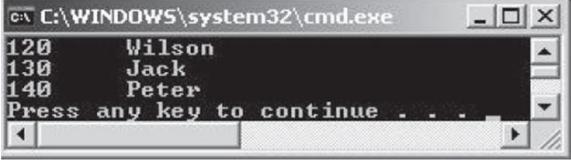
In slides 33 and 34, explain the code that creates a generic interface that is used as a constraint on a generic class.

Slide 35

Explain generic interface constraints.

Generic Interface Constraints 4-4

- ◆ In the code:
 - ❖ An interface **IDetails** declares a method **GetDetails ()**.
 - ❖ The class **Student** implements the interface **IDetails**.
 - ❖ The class **GenericList** is created that takes a type parameter **T**.
 - ❖ This type parameter is applied an interface constraint, which means the type parameter can only include details of the **IDetails type**.
 - ❖ The **Main ()** method creates an instance of the class **GenericList** by passing the type parameter value as **Student**, since the class **Student** implements the interface **IDetails**.
- ◆ The following figure shows the output of the code to create a generic interface:



©Aptech Ltd. Generics and Iterators / Session 13 35

Tell the students that an interface **IDetails** declares a method **GetDetails ()**. Tell them that the class **Student** implements the interface **IDetails** and the class **GenericList** is created that takes a type parameter **T**. Tell them that this type parameter is applied an interface constraint, which means the type parameter can only include details of the **IDetails** type.

Mention that 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**.

Then, refer to the figure in slide 35 to display the output of the code where a generic interface is created.

Slide 36

Understand generic delegates.



Generic Delegates 1-4

- ◆ Delegates are reference types that encapsulate a reference to a method that has a signature and a return type.
- ◆ Following are the features of a generic delegate:
 - ◆ Delegates can also be declared as generic.
 - ◆ It can be used to refer to multiple methods in a class with different types of parameters.
 - ◆ The number of parameters of the delegate and the referenced methods must be the same.
 - ◆ The type parameter list is specified after the delegate's name in the syntax.

© Aptech Ltd. Generics and Iterators / Session 13 36

In slide 36, explain about generic delegates to the students.

Tell the students that delegates are reference types that encapsulate a reference to a method that has a signature and a return type. Similar to classes, interfaces, structures, methods, and delegates can also be declared as generic.

Explain them that 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.

Mention that 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.

Slide 37

Understand the syntax used to declare a generic delegate.

The slide has a blue header bar with the title "Generic Delegates 2-4". Below the header, there is a bulleted list of points. The first point is "The following syntax is used to declare a generic delegate:" followed by a "Syntax" button. A code snippet is shown in a yellow box: `delegate <return_type><DelegateName><type parameter list>(<argument_list>);`. The second point is "where," followed by four sub-points: "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.", and "argument_list: Specifies the parameter within the delegate." At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd." and "Generics and Iterators / Session 13" on the left, and "37" on the right.

In slide 37, refer the syntax on the slide and tell the students that in the syntax, `return_type` determines the type of value the delegate will return, `DelegateName` is the name of the generic delegate. Tell them that `type parameter list` is used as a placeholder for the actual data type and `argument_list` specifies the parameter within the delegate.

Additional Information

For more information on generic delegates, refer the following link:

<http://msdn.microsoft.com/en-us/library/sx2bwtw7.aspx>

Slides 38 and 39

Understand the code that declares a generic delegate.

Generic Delegates 3-4

- The following code declares a generic delegate:

Snippet

```
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));
    }
}
```

©Aptech Ltd.

Generics and Iterators / Session 13

38

Generic Delegates 4-4

- In the code:
 - 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.
- The following figure shows the output of the code to declare a generic delegate:

©Aptech Ltd.

Generics and Iterators / Session 13

39

Using slide 38, explain the code. Tell the students that in the code, a generic delegate is declared in the **Numbers** class. Tell them that 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.

Explain them that 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. Mention that 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.

Then, refer to the figure in slide 39 that displays the output of the code.

Slide 40

Understand overloading methods using type parameters.

Overloading Methods Using Type Parameters 1-3

- ◆ 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.



© Aptech Ltd. Generics and Iterators / Session 13 40

In slide 40, explain overloading methods using type parameters to the students.

Tell the students that the methods of a generic class that take generic type parameters can be overloaded. Tell them that the programmer can overload the methods that use type parameters by changing the type or the number of parameters. However, mention that the type difference is not based on the generic type parameter, but is based on the data type of the parameter passed.

Slides 41 and 42

Understand overloading methods using type parameters.

Overloading Methods Using Type Parameters 2-3

- The following code demonstrates how to overload methods that use type parameters:

Snippet

```
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();
    }
}
```

© Aptech Ltd.

Generics and Iterators / Session 13

41

Overloading Methods Using Type Parameters 3-3

- In the code:
 - 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.
- The following figure shows the output of the code to overload methods using type parameters:

© Aptech Ltd.

Generics and Iterators / Session 13

42

Explain the code in slide 41 that demonstrates how to overload methods that use type parameters. Tell the students that in the code, the **General** class has two overloaded methods with different type parameters. Tell them that 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.

Explain them that 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`.

Mention that 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.

Then, refer to the figure in slide 42 to show the output of the code that displays the overload methods using type parameters.

Slides 43 to 45

Explain overriding virtual methods in generic class.

Overriding Virtual Methods in Generic Class 1-3

- ◆ Methods in generic classes can be overridden same as 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 as shown in the following code:

Snippet

```
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;
    }
}
```

©Aptech Ltd. All Rights Reserved

Generics and Iterators / Session 13 43

Overriding Virtual Methods in Generic Class 2-3

```

        }
    }
}

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\tSex");
        Console.WriteLine (objStudent.GetValue());
    }
}

```

Overriding Virtual Methods in Generic Class 3-3

- ◆ In the code:
 - ◆ The **GeneralList** class consists of a constructor that assigns the name of the student.
 - ◆ The **GetValue()** method of the **GeneralList** class is overridden in the **Student** class.
 - ◆ The constructor of the **Student** class invokes the **base** class constructor by using the **base** keyword and assigns the gender of the specified student.
 - ◆ The **GetValue()** method of the derived class returns the sex of the student.
 - ◆ The name of the student is invoked 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 following figure shows the output of the code to override virtual methods for generic class:



In slide 43, explain overriding virtual methods in generic class to the students.

Tell them that methods in generic classes can be overridden like the method in any non-generic class. Explain them that 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.

In slide 44, explain that the generic type **T** is overriding by getting the values from student method which is derived from base class. In slide 45, tell the students that in the code, the **GeneralList** class consists of a constructor that assigns the name of the student.

Tell them that the **GetValue()** method of the **GeneralList** class is overridden in the **Student** class. Tell that the constructor of the **Student** class invokes the base class constructor by using the **base** keyword and assigns the gender of the specified student.

Explain them that the **GetValue()** method of the derived class returns the sex of the student. The name of the student is invoked by using the **base** keyword to call the **GetValue()** method of the base class.

Mention that 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.

Then, refer to the figure in slide 45 to display the output of the code that displays the overridden virtual methods for generic class.

With this slide, you will finish explaining overriding virtual methods in generic class.

Slide 46

Understand iterators.

Iterators

- ◆ Consider a scenario where a person is trying to memorize a book of 100 pages.
- ◆ To finish the task, the person has to iterate through each of these 100 pages.
- ◆ Similar to this person who iterates through the pages, an iterator in C# is used to traverse through a list of values or a collection.
- ◆ It is a block of code that uses the `foreach` loop to refer to a collection of values in a sequential manner.
- ◆ For example, consider a collection of values that needs to be sorted.
- ◆ To implement the logic manually, a programmer can iterate through each value sequentially using iterators to compare the values.
- ◆ An iterator is not a data member but is a way of accessing the member.
- ◆ It can be a method, a `get` accessor, or an operator that allows you to navigate through the values in a collection.
- ◆ Iterators specify the way, values are generated, when the `foreach` statement accesses the elements within a collection.
- ◆ They keep track of the elements in the collection, so that you can retrieve these values if required.
- ◆ Consider an array variable consisting of 6 elements, where the iterator can return all the elements within an array one by one.
- ◆ The following figure illustrates these analogies:

Example

© Aptech Ltd.

Generics and Iterators / Session 13 46

In slide 46, introduce iterators to the students.

Give them an example. Tell them to consider the following scenario:

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. Tell them that 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. Mention that it is a block of code that uses the `foreach` loop to refer to a collection of values in a sequential manner.

Also, tell them to consider the following scenario:

Tell them to consider a collection of values that needs to be sorted. To implement the logic manually, a programmer can iterate through each value sequentially using iterators to compare the values.

Refer to the figure in slide 46 to illustrate these analogies.

Then, tell the students that 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 them to navigate through the values in a collection.

Iterators are the ways with the help of which one can iterate through the collections such as `ArrayList` or `List`.

Explain them that the iterators specify the way, values are generated, when the `foreach` statement accesses the elements within a collection. They keep track of the elements in the collection, so that you can retrieve these values if required.

Also, tell the students to consider the following scenario:

Tell them to consider an array variable consisting of 6 elements, where the iterator can return all the elements within an array one by one.

Additional Information

For more information on iterators, refer the following link:

<http://msdn.microsoft.com/en-us/library/dscyy5s0.aspx>

Slide 47

Understand the benefits of iterators.

	Benefits
	<ul style="list-style-type: none"> ◆ For a class that behaves like a collection, it is preferable to use iterators to iterate through the values of the collection with the <code>foreach</code> statement. ◆ By doing this, one can get the following benefits: <ul style="list-style-type: none"> ◆ Iterators provide a simplified and faster way of iterating through the values of a collection. ◆ Iterators reduce the complexity of providing an enumerator for a collection. ◆ Iterators can return large number of values. ◆ Iterators can be used to evaluate and return only those values that are needed. ◆ Iterators can return values without consuming memory by referring each value in the list.

Explain using slide 47 that 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.

Tell them that by doing this, one can get following benefits:

Tell the students that iterators provide a simplified and faster way of iterating through the values of a collection and also reduce the complexity of providing an enumerator for a collection. Explain them that the iterators can return large number of values and can be used to evaluate and return only those values that are needed. Mention that the iterators can return values without consuming memory by referring each value in the list.

Slide 48

Understand the implementation of iterators.



Implementation 1-3

- ◆ 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.



© Aptech Ltd.

Generics and Iterators / Session 13 48

In slide 48, explain the implementation of the iterators to the students.

Tell the students that the iterators can be created by implementing the `GetEnumerator()` method that returns a reference of the `IEnumerator` interface.

Explain them that the iterator block uses the `yield` keyword to provide values to the instance of the enumerator or to terminate the iteration. Tell them that the `yield return` statement returns the values, while the `yield break` statement ends the iteration process. Mention that 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.

The `yield return` statement is used by Iterator method to return each element one at a time. When `yield` statement is reached code is remembered, next time when iterator function is called, execution is restarted.

Slides 49 and 50

Understand the implementation of iterators.

Implementation 2-3

- The following code demonstrates the use of iterators to iterate through the values of a collection:

Snippet

```
using System;
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);
        }
    }
}
```

©Aptech Ltd.

Generics and Iterators / Session 13

49

Implementation 3-3

- In the code:
 - 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.
- The following code displays the use of iterators:

©Aptech Ltd.

Generics and Iterators / Session 13

50

Understand the code that demonstrates the use of iterators to iterate through the values of a collection. Tell the students that the class **Department** implements the interface **IEnumerable**. Tell them that the class **Department** consists of an array variable that stores the department names and a **GetEnumerator()** method, that contains the **for** loop.

Using slide 49, explain them that 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.

Mention that the `Main()` method creates an instance of the class **Department** and contains a `foreach` loop that displays the department names.

Refer to the figure in slide 50 to show the output of the code that displays the use of iterators. 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.

Slides 51 to 53

Understand generic iterators.

Generic Iterators 1-3

- ◆ 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.
- ◆ The following code demonstrates how to create a generic iterator to iterate through values of any type:

Snippet

```
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;
        }
    }
}
```

Generic Iterators 2-3

```

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();
    }
}

```

- The following figure shows the output of the code to create a generic iterator:

© Aptech Ltd. Generics and Iterators / Session 13 52

Generic Iterators 3-3

- In the code:
 - The generic class, **GenericDepartment**, is created with the generic type parameter T.
 - The class declares an array variable and consists of a parameterized constructor that assigns values to this array variable.
 - In the generic class, **GenericDepartment**, the `GetEnumerator()` method returns a generic type of the `IEnumerator` interface.
 - This method returns elements stored in the array variable, using the `yield` statement.
 - In the **GenericIterator** class, an instance of the **GenericDepartment** class is created that refers to the different department names within the array.
 - Another object of the **GenericDepartment** class is created, that refers to the different department IDs within the array.

© Aptech Ltd. Generics and Iterators / Session 13 53

In slides 51 and 52, explain generic iterators to the students.

Tell the students that C# allows programmers to create generic iterators. Tell them that the generic iterators are created by returning an object of the generic `IEnumerable<T>` or `IEnumerable<T>` interface. They are used to iterate through values of any value type.

In slide 53, explain the code that demonstrates how to create a generic iterator to iterate through values of any type and refer to the figure in slide 52 to show the output of the code to create a generic iterator.

Tell the students that in the code, the generic class, **GenericDepartment**, is created with the generic type parameter **T**. Tell them that the class declares an array variable and consists of a parameterized constructor that assigns values to this array variable.

Explain them that 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.

Mention that in the **GenericIterator** class, an instance of the **GenericDepartment** class is created that refers to the different department names within the array. Another object of the **GenericDepartment** class is created, that refers to the different department IDs within the array.

Slide 54

Explain the implementation of named iterators.

Implementing Named Iterators 1-2

- ◆ Another way of creating iterators is by creating a method, whose return type is the **IEnumerable** interface.
- ◆ This is called a **named iterator**. Named iterators can accept parameters that can be used to manage the starting and end points of a **foreach** loop.
- ◆ This flexible technique allows you to fetch the required values from the collection.
- ◆ The following syntax creates a named iterator:

Syntax

```
<access_modifier> IEnumerable <IteratorName>
  (<parameter list>) {}
```

- ◆ where,
 - ◆ **access_modifier**: Specifies the scope of the named iterator.
 - ◆ **IteratorName**: Is the name of the iterator method.
 - ◆ **parameter list**: Defines zero or more parameters to be passed to the iterator method.

In slide 54, explain the implementation of named iterators to the students.

Tell the students that another way of creating iterators is by creating a method, whose return type is the **IEnumerable** interface. This is called a **named iterator**.

Tell them that the Named iterators can accept parameters that can be used to manage the starting and end points of a **foreach** loop. Mention that this flexible technique allows user to fetch the required values from the collection.

Then, refer the syntax that creates a named iterator and tell the students that in the syntax, **access_modifier** specifies the scope of the named iterator, **IteratorName** is the name

of the iterator method, and parameter list defines zero or more parameters to be passed to the iterator method.

Slide 55

Explain the implementation of named iterators.

Implementing Named Iterators 2-2

- The following code demonstrates how to create a named iterator:

Snippet

```
using System;
class NamedIterators{
    string[] cars = { "Ferrari", "Mercedes", "BMW", "Toyota", "Nissan"};
    public IEnumerable GetCarNames() {
        for (int i = 0; i < cars.Length; i++) {
            yield return cars[i];
        }
    }
    static void Main(string[] args) {
        NamedIterators objIterator = new NamedIterators();
        foreach (string str in objIterator.GetCarNames()) {
            Console.WriteLine(str);
        }
    }
}
```

Output

```
Ferrari
Mercedes
BMW
Toyota
Nissan
```

- In the code:
 - 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.

© Aptech Ltd. Generics and Iterators / Session 13 55

Using slide 55, tell the student that in the code, the **NamedIterators** class consists of an array variable and a method **GetCarNames ()**, whose return type is **IEnumerable**. Tell them that the **for** loop iterates through the values within the array variable.

Mention that 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.

In-Class Question:

After you finish explaining the Implementing Name Iterators, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are the benefits of using iterators?

Answer:

Following are the benefits of iterators:

- Iterators provide a simplified and faster way of iterating through the values of a collection.
- Iterators reduce the complexity of providing an enumerator for a collection.
- Iterators can return large number of values.
- Iterators can be used to evaluate and return only those values that are needed.
- Iterators can return values without consuming memory by referring each value in the list.

Slide 56

Summarize the session.

Q
Summary

- ◆ Generics are data structures that allow you to reuse a code for different types such as classes or interfaces.
- ◆ Generics provide several benefits such as type-safety and better performance.
- ◆ Generic types can be declared by using the type parameter, which is a placeholder for a particular type.
- ◆ Generic classes can be created by the class declaration followed by the type parameter list enclosed in the angular brackets and application of constraints (optional) on the type parameters.
- ◆ An iterator is a block of code that returns sequentially ordered values of the same type.
- ◆ One of the ways to create iterators is by using the GetEnumerator() method of the IEnumerable or IEnumerator interface.
- ◆ The yield keyword provides values to the enumerator object or to signal the end of the iteration.

© Aptech Ltd.
Generics and Iterators / Session 13 56

In slide 56, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- Generics are data structures that allow you to reuse a code for different types such as classes or interfaces.
- Generics provide several benefits such as type-safety and better performance.
- Generic types can be declared by using the type parameter, which is a placeholder for a particular type.
- Generic classes can be created by the class declaration followed by the type parameter list enclosed in the angular brackets and application of constraints (optional) on the type parameters.
- An iterator is a block of code that returns sequentially ordered values of the same type.
- One of the ways to create iterators is by using the `GetEnumerator()` method of the `IEnumerable` or `IEnumerator` interface.
- The `yield` keyword provides values to the enumerator object or to signal the end of the iteration.

13.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a question to students to search additional information, such as:

1. What are some other examples of generics? Give an example.
2. What is meant by overriding virtual methods in generic class?

Session 14 - Advanced Methods and Types

14.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- Generics are data structures that allow you to reuse a code for different types such as classes or interfaces.
- Generics provide several benefits such as type-safety and better performance.
- Generic types can be declared by using the type parameter, which is a placeholder for a particular type.
- Generic classes can be created by the class declaration followed by the type parameter list enclosed in the angular brackets and application of constraints (optional) on the type parameters.
- An iterator is a block of code that returns sequentially ordered values of the same type.
- One of the ways to create iterators is by using the `GetEnumerator()` method of the `IEnumerable` or `IEnumerator` interface.
- The `yield` keyword provides values to the enumerator object or to signal the end of the iteration.

Here, you can ask students the key topics they can recall from previous session. Ask them to briefly explain generics in C#. You can also ask them to explain the process of creating and using generics. Furthermore, ask them to explain iterators. Prepare a question or two which will be a key point to relate the current session objectives.

14.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe anonymous methods
- Define extension methods
- Explain anonymous types
- Explain partial types
- Explain nullable types

14.1.2 Teaching Skills

To teach this session successfully, you must know about anonymous and extension methods in C#. You should be aware of anonymous, partial, and nullable types.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here during the In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to advanced methods and types.

14.2 In-Class Explanations

Slide 3

Understand anonymous methods.

The slide has a teal header bar with the title "Anonymous Methods". The main content area contains a bulleted list of points about anonymous methods, followed by a code snippet. A yellow box highlights a portion of the code, and a callout bubble labeled "Anonymous Method" points to it.

- ◆ An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.
- ◆ Delegates can invoke one or more named methods that are included while declaring the delegates.
- ◆ Prior to anonymous methods, if you wanted to pass a small block of code to a delegate, you always had to create a method and then pass it to the delegate.
- ◆ With the introduction of anonymous methods, you can pass an inline block of code to a delegate without actually creating a method.
- ◆ The following code 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();
}

```

© Aptech Ltd. Building Applications Using C# / Session 14 3

Use slide 3 to explain that an anonymous method is an inline nameless block of code that can be passed as a delegate parameter.

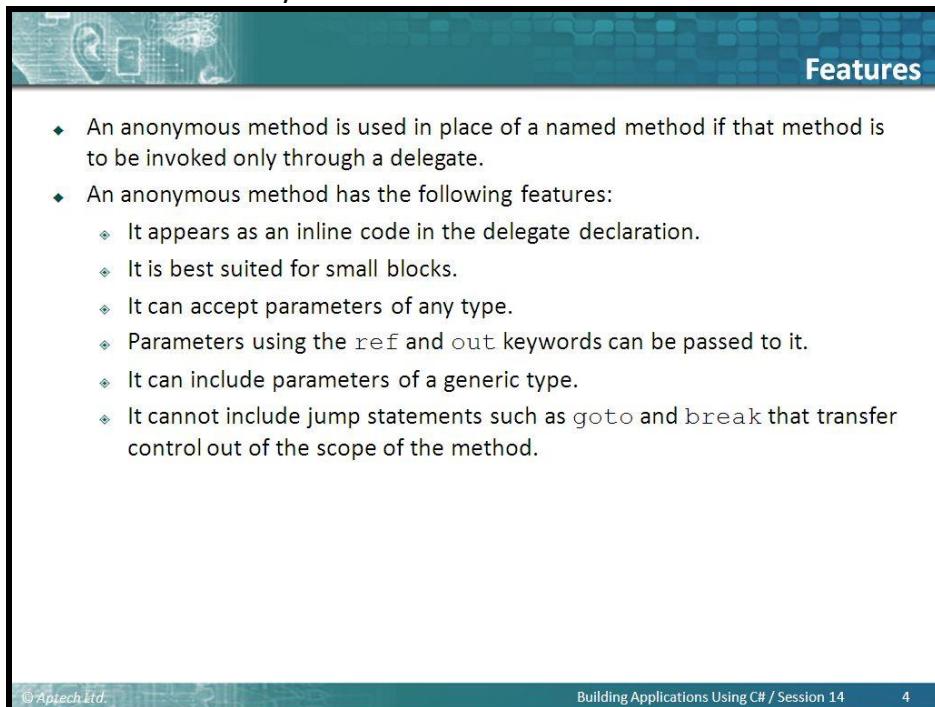
Tell that delegates can invoke one or more named methods that are included while declaring the delegates. Anonymous methods are the best way to implement inline codes.

Explain that prior to anonymous methods, to pass a small block of code to a delegate, create a method and then pass it to the delegate. With the introduction of anonymous methods, pass an inline block of code to a delegate without actually creating a method.

You can refer to figure in slide 3 that displays an example of anonymous method.

Slide 4

Understand the features of anonymous methods.



The slide has a blue header bar with the word 'Features' on the right. The main content area is white with a black border. At the bottom, there's a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 14' in the center, and the number '4' on the right.

- ◆ An anonymous method is used in place of a named method if that method is to be invoked only through a delegate.
- ◆ An anonymous method has the following features:
 - ◆ It appears as an inline code in the delegate declaration.
 - ◆ It is best suited for small blocks.
 - ◆ It can accept parameters of any type.
 - ◆ Parameters using the `ref` and `out` keywords can be passed to it.
 - ◆ It can include parameters of a generic type.
 - ◆ It cannot include jump statements such as `goto` and `break` that transfer control out of the scope of the method.

In slide 4, explain to the students that an anonymous method is used in place of a named method if that method is to be invoked only through a delegate.

Mention the features of anonymous method. Explain that it appears as an inline code in the delegate declaration. Then, tell that it is best suited for small blocks, as having large blocks of anonymous code spanning several pages can make debugging difficult. Explain that an anonymous method can accept parameters of any type. Mention that parameters using the `ref` and `out` keywords can be passed to it. Also, tell that it can include parameters of a generic type. Lastly, tell that it cannot include jump statements such as `goto` and `break` that transfer control out of the scope of the method.

Additional Information

For more information on anonymous methods, refer the following link:

<http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx>

Slides 5 to 7

Understand creating anonymous methods.

Creating Anonymous Methods 1-3

- ◆ An anonymous method is created when you instantiate or reference a delegate with a block of unnamed code.

- ◆ Following points need to be noted while creating anonymous methods:
 - ❖ When a `delegate` keyword is used inside a method body, it must be followed by an anonymous method body.
 - ❖ The method is defined as a set of statements within curly braces while creating an object of a delegate.
 - ❖ Anonymous methods are not given any return type.
 - ❖ Anonymous methods are not prefixed with access modifiers.

Creating Anonymous Methods 2-3

- ◆ The following figure and snippet display the syntax and code for anonymous methods respectively:

```
// Create a delegate instance
<access modifier> delegate <return type>
<DelegateName> (parameters);

// Instantiate the delegate using an anonymous method

<DelegateName> <objDelegate> = new <DelegateName>
(parameters)
{ /* ... */};
```

```
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();
    }
}
```

Snippet

©Aptech Ltd.

Building Applications Using C# / Session 14

6

©Aptech Limited

Creating Anonymous Methods 3-3

- ◆ In the code:
 - ❖ 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

Using slide 5, explain the students that an anonymous method is created on instantiating or referencing a delegate with a block of unnamed code.

Tell the points that are to be noted while creating anonymous methods.

Explain that when a `delegate` keyword is used inside a method body, it must be followed by an anonymous method body.

Then, tell that the method is defined as a set of statements within curly braces while creating an object of a delegate. Also, tell that anonymous methods are not given any return type and are not prefixed with access modifiers.

Anonymous methods enable you to omit the parameter list. This means that an anonymous method can be converted to delegates with a variety of signatures.

Creating anonymous methods is essentially a way to pass a code block as a delegate parameter.

Use slide 6 to refer to the figure that displays the syntax of anonymous methods. Use slide 7 to explain the code and output that creates an anonymous method.

Mention that in the code, 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.

Slide 8

Understand referencing multiple anonymous methods.

Referencing Multiple Anonymous Methods 1-2

- ◆ C# allows you to create and instantiate a delegate that can reference multiple anonymous methods.
- ◆ This is done using the `+=` operator.
- ◆ The `+=` operator is used to add additional references to either named or anonymous methods after instantiating the delegate.
- ◆ The following code shows how one delegate instance can reference several anonymous methods:

Snippet

```
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");
        };
    }
}
```

Use slide 8 to explain the students that C# creates and instantiates 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.

Tell that the code shows how one delegate instance can reference several anonymous methods.

Slide 9

Understand referencing multiple anonymous methods.

The slide has a blue header bar with the title "Referencing Multiple Anonymous Methods 2-2". Below the header is a yellow rectangular box containing C# code. The code creates a delegate, adds two anonymous methods to it, and then calls the delegate. The output section shows the console output corresponding to the code execution.

```
//delegate instantiated with another anonymous method
// reference
objDisp += delegate()
{
    Console.WriteLine("This illustrates another anonymous
        method with the same delegate instance");
};

objDisp();
```

Output

This illustrates one anonymous method
This illustrates another anonymous method with the
same delegate instance

©Aptech Ltd.

Building Applications Using C# / Session 14

9

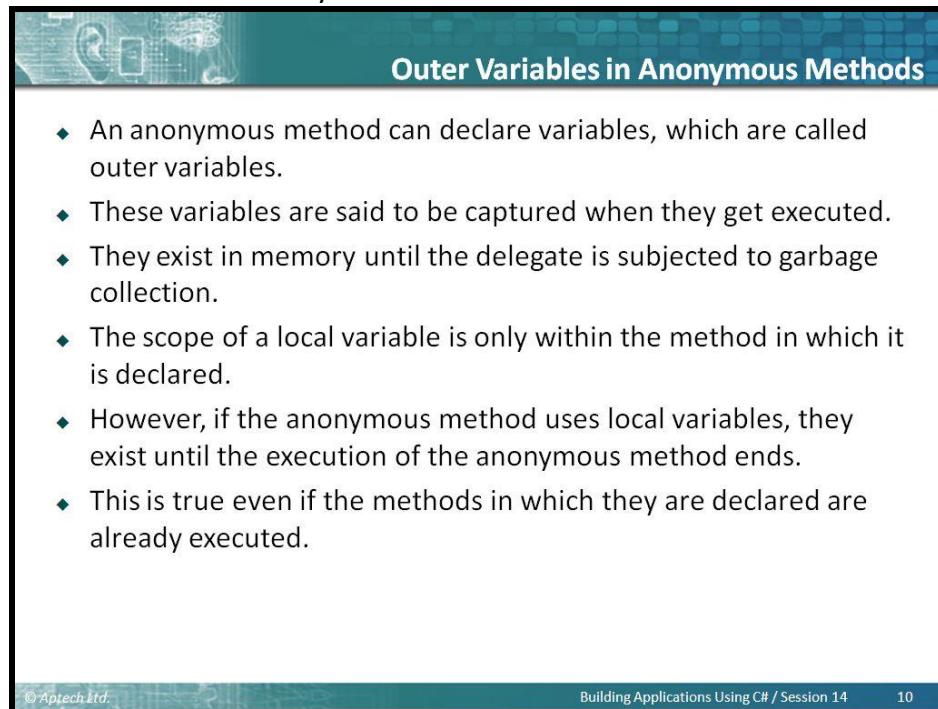
Use slide 9 to explain the code and output.

In the code, an anonymous method is created during the delegate instantiation and another anonymous method is created and referenced by the delegate using the `+=` operator.

With the help of anonymous methods, coding overhead can be reduced in instantiating delegates because there is no need to create a separate method.

Slide 10

Understand outer variables in anonymous methods.



Outer Variables in Anonymous Methods

- ◆ An anonymous method can declare variables, which are called outer variables.
- ◆ These variables are said to be captured when they get executed.
- ◆ They exist in memory until the delegate is subjected to garbage collection.
- ◆ The scope of a local variable is only within the method in which it is declared.
- ◆ However, if the anonymous method uses local variables, they exist until the execution of the anonymous method ends.
- ◆ This is true even if the methods in which they are declared are already executed.

In slide 10, explain that an anonymous method can declare variables, which are called outer variables. These variables are said to be captured when they get executed.

Tell that they exist in memory until the delegate is subjected to garbage collection. The scope of a local variable is only within the method in which it is declared.

Also, tell that if the anonymous method uses local variables, they exist until the execution of the anonymous method ends. This is true even if the methods in which they are declared are already executed.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is an anonymous method?

Answer:

An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.

Slides 11 and 12

Understand passing parameters.

Passing Parameters 1-2

- ◆ 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 parentheses.
- ◆ The block of code within the anonymous method can access these specified parameters just like any normal method.
- ◆ You can pass the parameter values to the anonymous method while invoking the delegate.
- ◆ The following code demonstrates how parameters are passed to anonymous methods:

Snippet

```
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);
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 14 11

Passing Parameters 2-2

- ◆ In the code:
 - ◆ 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

© Aptech Ltd. Building Applications Using C# / Session 14 12

Use slide 11 to explain that 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.

Mention that these parameters are specified within parenthesis. The block of code within the anonymous method can access these specified parameters just like any normal method.

Also, tell that the parameter values can be passed to the anonymous method while invoking the delegate. Tell that the code that demonstrates how parameters are passed to anonymous methods.

Use slide 12 to explain the code and output. In the code, a delegate **Display** is created. Mention that the 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.

If the anonymous method definition does not include any arguments, you can use a pair of empty parenthesis in the declaration of the delegate. The anonymous methods without a parameter list cannot be used with delegates that specify out the parameters.

With this slide, you will finish explaining anonymous methods.

Slide 13

Understand extension methods.

Extension Methods 1-7

- ◆ Extension methods allow you to extend an existing type with new functionality without directly modifying those types.
- ◆ Extension methods are static methods that have to be declared in a static class.
- ◆ You can declare an extension method by specifying the first parameter with the `this` keyword.
- ◆ The first parameter in this method identifies the type of objects in which the method can be called.
- ◆ The object that you use to invoke the method is automatically passed as the first parameter.

Syntax

```
static return-type MethodName (this type obj, param-list)
```

where:

- ◆ `return-type`: the data type of the return value
- ◆ `MethodName`: the extension method name
- ◆ `type`: the data type of the object
- ◆ `param-list`: the list of parameters (optional)

© Aptech Ltd.

Building Applications Using C# / Session 14

13

In slide 13, tell the students that extension methods allow you to extend an existing type with new functionality without directly modifying those types.

Then, tell that extension methods are static methods that have to be declared in a static class. Also, tell that an extension method can be declared by specifying the first parameter with `this` keyword.

Mention that the first parameter in this method identifies the type of objects in which the method can be called. The object that you use to invoke the method is automatically passed as the first parameter. Explain the syntax as given on the slide.

Additional Information

For more information on extension methods, refer the following link:

<http://msdn.microsoft.com/en-us/library/bb383977.aspx>

Slide 14

Understand extension methods.

The slide has a teal header bar with the title "Extension Methods 2-7". Below the header is a yellow rectangular area containing a C# code snippet. The snippet is enclosed in a blue box labeled "Snippet". The code defines a static class "ExtensionExample" with a public static method "FirstLetterLower" that takes a string "result" and returns a new string where the first character is converted to lowercase. The code uses a character array to achieve this.

```
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;
    }
}
```

Explain how to create an extension method for a string which converts first character of the string to lowercase. In slide 14, tell that the code creates an extension method for a string and converts the first character of the string to lowercase.

Slide 15

Understand extension methods.

The slide has a teal header bar with the title "Extension Methods 3-7". The main content area contains a code snippet:

```
class Program
{
    public static void Main(string[] args)
    {
        string country = "Great Britain";
        // Calling the extension method
        Console.WriteLine(country.FirstLetterLower());
    }
}
```

Below the code, there is a bulleted list:

- ◆ In the code:
 - ◆ 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.

At the bottom of the slide, there is footer text: "© Aptech Ltd.", "Building Applications Using C# / Session 14", and "15".

Use slide 15 to explain the code.

In the code, an extension method named **FirstLetterLower** is defined with one parameter that is preceded with `this` keyword.

Mention that 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.

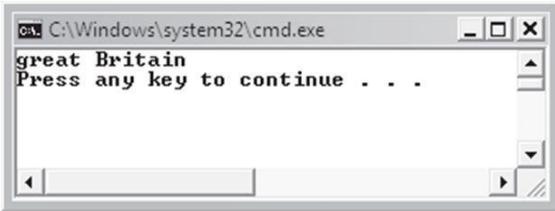
In slide 16, explain the output of the code.

Slide 16

Understand extension methods.

Extension Methods 4-7

- ◆ The following figure depicts the output:



- ◆ The advantages of extension methods are as follows:
 - ◆ You can extend the functionality of the existing type without modification. This will avoid the problems of breaking source code in existing applications.
 - ◆ You can add additional methods to standard interfaces without physically altering the existing class libraries.

© Aptech Ltd. Building Applications Using C# / Session 14 16

You can refer to the figure in slide 16 to depict the output of the code.

Also, mention the advantages of extension methods as follows:

Tell that the functionality of the existing type can be extended without modification. This will avoid the problems of breaking source code in existing applications.

Also mention that additional methods can be added to standard interfaces without physically altering the existing class libraries.

An anonymous method is an inline nameless block of code that can be passed as a delegate parameter.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What are extension methods?

Answer:

Extension methods allow you to extend an existing type with new functionality without directly modifying those types. They are static methods that have to be declared in a static class.

Slide 17

Understand extension methods.

The slide has a teal header bar with the title "Extension Methods 5-7". Below the header is a blue box labeled "Snippet". The code within the snippet is as follows:

```
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;
    }
}
```

At the bottom of the slide, there is a footer bar with the text "©Aptech Ltd.", "Building Applications Using C# / Session 14", and "17".

In slide 17, tell that the code 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.

Slide 18

Understand extension methods.

Extension Methods 6-7

```

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 the code:
 - ❖ 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)

```

Use slide 18 to explain the code.

In the code, the extension method **RemoveDuplicate()** is declared and returns a generic List when invoked.

Mention that the method accepts a generic List<T> as the first argument:

```
public static List<T> RemoveDuplicate<T>(this List<T> allCities)
```

Slide 19

Understand extension methods.

The slide has a teal header bar with the title "Extension Methods 7-7". Below the header, there is a bulleted list of points:

- ◆ The following lines of code iterate through each value in the collection, remove the duplicate values, and store the unique values in the List, **finalCities**:

```
foreach (var eachCity in allCities)
if (!finalCities.Contains(eachCity))
finalCities.Add(eachCity);
```
- ◆ The following figure displays the output:

A blue button labeled "Output" is visible. Below it is a screenshot of a Windows command prompt window titled "cmd C:\Windows\system32\cmd.exe". The window shows the following text:
Seoul
Beijing
Berlin
Istanbul
Paris
Press any key to continue . . .

At the bottom of the slide, there is footer text: "©Aptech Ltd.", "Building Applications Using C# / Session 14", and "19".

Use slide 19 to explain how the code iterates through each value in the collection, remove the duplicate values, and store the unique values in the List, **finalCities**.

You can refer to the figure 14.4 that displays the output of code.

Tips:

Even though the extension method is declared as static, you can still call it using an object.

With this slide, you will finish explaining extension method.

Slide 20

Understand anonymous types.

The slide has a blue header bar with the title 'Anonymous Types 1-8'. Below the header, there is a bulleted list under the heading 'Anonymous type:' and a code snippet section under 'Syntax'.

- ◆ Anonymous type:
 - ◆ Is basically a class with no name and is not explicitly defined in code.
 - ◆ Uses object initializers to initialize properties and fields. Since it has no name, you need to declare an implicitly typed variable to refer to it.

Syntax

```
new { identifierA = valueA, identifierB =  
      valueB, ..... }
```

where,

- ◆ identifierA, identifierB, ...: Identifiers that will be translated into read-only properties that are initialized with values

At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 14' in the center, and the number '20' on the right.

Use slide 20 to explain that an anonymous type is basically a class with no name and is not explicitly defined in code.

Tell that an anonymous type uses object initializers to initialize properties and fields.

Then, tell that since it has no name, an implicitly typed variable can be declared to refer to it.

Explain the syntax as given on the slide.

Additional Information

For more information on anonymous type, refer the following link:

<http://msdn.microsoft.com/en-us/library/bb397696.aspx>

Slide 21

Understand the use of anonymous types.

Anonymous Types 2-8

- The following code demonstrates the use of anonymous types:

Snippet

```
using System;
/// <summary>
/// Class AnonymousTypeExample to demonstrate anonymous type
/// </summary>
class AnonymousTypeExample
{
    public static void Main(string[] args)
    {
        // Anonymous Type with three properties.
        var stock = new { Name = "Michigan Enterprises", Code = 1301,
                         Price = 35056.75 };
        Console.WriteLine("Stock Name: " + stock.Name);
        Console.WriteLine("Stock Code: " + stock.Code);
        Console.WriteLine("Stock Price: " + stock.Price);
    }
}
```

- Consider the following line of code:

```
var stock = new { Name = "Michigan Enterprises", Code =
1301, Price = 35056.75 };
```

- The compiler creates an anonymous type with all the properties that is inferred from object initializer.
- In this case, the type will have properties **Name**, **Code**, and **Price**.

In slide 21, explain that the code demonstrates the use of anonymous types.

Tell that in the code, 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**.

Tell that the compiler automatically generates the get and set methods, as well as the corresponding private variables to hold these properties.

Also, mention that 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.

Slide 22

Understand the use of anonymous types.

Anonymous Types 3-8

- ◆ 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.
- ◆ The following figure displays output:

The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The window contains the following text:
Stock Name: Michigan Enterprises
Stock Code: 1301
Stock Price: 35056.75
Press any key to continue . . .

At the bottom of the slide, there is footer text: ©Aptech Ltd., Building Applications Using C# / Session 14, and the number 22.

You can refer to figure in slide 22 that displays the output of the code.

Slide 23

Understand the use of anonymous types.

Anonymous Types 4-8

- ◆ When an anonymous type is created, the C# compiler carries out the following tasks:
 - ◆ Interprets the type
 - ◆ Generates a new class
 - ◆ Use the new class to instantiate a new object
 - ◆ Assigns the object with the required parameters
- ◆ The compiler internally creates a class with the respective properties when code is compiled.
- ◆ In this program, the class might look like the one that is shown in code.

In slide 23, tell the students that when an anonymous type is created, the C# compiler carries out certain tasks. Tell the students that they will now understand the tasks that are carried out when an anonymous type is created. Mention the tasks. Tell them that the type is interpreted. Then, a new class is generated. Tell that the new class is used to instantiate a new object. Also, tell that it assigns the object with the required parameters. Mention that the compiler internally creates a class with the respective properties when the code is compiled.

Slide 24

Understand the use of anonymous types.

The slide has a teal header bar with the title "Anonymous Types 5-8". Below the header is a yellow rectangular box containing a C# class definition. The class is named `_NO_NAME_` and contains three properties: `_Name`, `_Code`, and `_Price`. Each property has a corresponding `get` and `set` accessor. The code is enclosed in a `Snippet` box. At the bottom of the slide, there is a footer bar with the text "©Aptech Ltd.", "Building Applications Using C# / Session 14", and the number "24".

```
class _NO_NAME_
{
    private string _Name;
    private int _Code;
    private double _Price;
    public string Name
    {
        get { return _Name; }
        set { _Name = value; }
    }
    public int Code
    {
        get { return _Code; }
        set { _Code = value; }
    }
    public double Price
    {
        get { return _Price; }
        set { _Price = value; }
    }
}
```

In slide 24, tell that in this program, the class might look like the one that is shown in the code. The C# compiler generates the anonymous type at compile time, not at runtime.

Slide 25

Understand the use of anonymous types.

The slide has a blue header bar with the title "Anonymous Types 6-8". Below the header is a yellow box containing a C# code snippet. To the left of the code box is a blue button labeled "Snippet". Above the code box is a bulleted list describing the purpose of the code.

◆ The following code demonstrates passing an instance of the anonymous type to a method and displaying the details:

```
using System;
using System.Reflection;
/// <summary>
/// Class Employee to demonstrate anonymous type.
/// </summary>
public class Employee
{
    public void DisplayDetails(object emp)
    {
        String fName = "";
        String lName = "";
        int age = 0;
        PropertyInfo[] attrs = emp.GetType().GetProperties();
        foreach (PropertyInfo attr in attrs)
        {
            switch (attr.Name)
            {
                case "FirstName":
                    fName = attr.GetValue(emp, null).ToString();
                    break;
                case "LastName":
                    lName = attr.GetValue(emp, null).ToString();
                    break;
                case "Age":
                    age = (int)attr.GetValue(emp, null);
                    break;
            }
        }
    }
}
```

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 14", and the number "25".

Use slide 25 to explain the code that demonstrates passing an instance of the anonymous type to a method and displaying the details.

Slide 26

Understand the use of anonymous types.

Anonymous Types 7-8

```

        Console.WriteLine("Name: {0} {1}, Age: {2}", fName, lName,
                        age);
    }
}
class AnonymousExample
{
    public static void Main(string[] args)
    {
        Employee david = new Employee();
        // Creating the anonymous type instance and passing it
        // to a method.
        david.DisplayDetails(new { FirstName = "David", LastName =
            "Blake", Age = 30 });
    }
}

```

In the code:

- ❖ It creates an instance of the anonymous type with three properties, **FirstName**, **LastName**, and **Age** with values David, Blake, and 30 respectively.
- ❖ This instance is then passed to the method, **DisplayDetails()** .
- ❖ In **DisplayDetails()** method, the instance that was passed as parameter is stored in the object, **emp**.

In slide 26, explain the code. Tell that the code creates an instance of the anonymous type with three properties, **FirstName**, **LastName**, and **Age** with values David, Blake, and 30 respectively. This instance is then passed to the method, **DisplayDetails()**.

Mention that in **DisplayDetails()** method, the instance that was passed as parameter is stored in the object, **emp**. Then, the code uses reflection to query the object's properties. The **GetType()** method retrieves the type of the current instance, **emp** and

GetProperties() method retrieves the properties of the object, **emp**.

Also, tell that the details are then stored in the **PropertyInfo** collection, **attr**. Finally, the details are extracted through the **GetValue()** method of **PropertyInfo** class. If this program did not make use of an anonymous type, a lot more code would have been required to produce the same output.

Slide 27

Understand the use of anonymous types.

Anonymous Types 8-8

- ◆ Then, the code uses reflection to query the object's properties.
- ◆ The **GetType()** method retrieves the type of the current instance, **emp** and **GetProperties()** method retrieves the properties of the object, **emp**.
- ◆ The details are then stored in the **PropertyInfo** collection, **attr**. Finally, the details are extracted through the **GetValue()** method of the **PropertyInfo** class.
- ◆ If this program did not make use of an anonymous type, a lot more code would have been required to produce the same output.

◆ The following figure displays the output:

©Aptech Ltd. Building Applications Using C# / Session 14 27

You can refer to the figure in the slide that displays the output of code.

With this slide, you will finish explaining anonymous types.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is an anonymous type?

Answer:

An anonymous type is basically a class with no name and is not explicitly defined in code and uses object initializers to initialize properties and fields.

Slide 28

Understand partial types.

The slide features a blue header bar with the title "Partial Types" on the right. Below the header is a white content area with a blue sidebar on the left containing the word "Example". The main content area contains a bulleted list of points. At the bottom of the slide is a blue footer bar with the copyright information "© Aptech Ltd." on the left, the session title "Building Applications Using C# / Session 14" in the center, and the slide number "28" on the right.

Example

- ◆ Assume that a large organization has its IT department spread over two locations, Melbourne and Sydney.
- ◆ The overall functioning takes place through consolidated data gathered from both the locations.
- ◆ The customer of the organization would see it as a whole entity, whereas, in reality, it would be composed of multiple units.
- ◆ Now, think of a very large C# class or structure with lots of member definitions.
- ◆ You can split the data members of the class or structure and store them in different files.
- ◆ These members can be combined into a single unit while executing the program.
- ◆ This can be done by creating partial types.

Use slide 28 to give an example.

Tell the students to assume that a large organization has its IT department spread over two locations, Melbourne and Sydney.

Tell that 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.

Then, ask them to think of a very large C# class or structure with lots of member definitions. You can split the data members of the class or structure and store them in different files. These members can be combined into a single unit while executing the program. This can be done by creating partial types.

Slide 29

Understand features of partial types.

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.
- ◆ The following figure displays an example of a partial type:

File 1	File 2
<pre>partial struct Sample { <MethodOne>; }</pre>	<pre>partial struct Sample { <MethodTwo>; }</pre>

In slide 29, tell that the partial types feature facilitates the definition of classes, structures, and interfaces over multiple files. Partial types provide various benefits.

Tell that partial types separate the generator code from the application code. Tell that they help in easier development and maintenance of the code. Then, tell that they make the debugging process easier.

Also, tell that they prevent programmers from accidentally modifying the existing code.

You can refer to the figure in slide 29 that displays an example of a partial type.

Partial type definitions are used to allow for the definition of a class, struct, or interface to be split into multiple files.

Additional Information

For more information on partial types, refer the following links:

<http://msdn.microsoft.com/en-us/library/wa80x488.aspx>

<http://msdn.microsoft.com/en-us/library/wbx7zzdd.aspx>

<http://www.codeproject.com/Articles/11858/Introducing-C-Partial-Types>

Slide 30

Understand merged elements during compilation.



Merged Elements during Compilation 1-4

- ◆ The members of partial classes, partial structures, or partial interfaces declared and stored at different locations are combined together at the time of compilation.
- ◆ These members can include:
 - ◆ XML comments
 - ◆ Interfaces
 - ◆ Generic-type parameters
 - ◆ Class variables
 - ◆ Local variables
 - ◆ Methods
 - ◆ Properties
- ◆ A partial type can be compiled at the Developer Command Prompt for VS2012. The command to compile a partial type is:
`csc /out:<FileName>.exe <CSharpFileNameOne>.cs <CSharpFileNameTwo>.cs`
where,
FileName: Is the user specified name of the .exe file.
CSharpFileNameOne: Is the name of the first file where a partial type is defined.
CSharpFileNameTwo: Is the name of the second file where a partial type is defined.

© Aptech Ltd.

Building Applications Using C# / Session 14

30

In slide 30, explain that the members of partial classes, partial structures, or partial interfaces declared and stored at different locations are combined together at the time of compilation.

Tell that these members include XML comments, interfaces, generic-type parameters, class variables, local variables, methods, and properties.

Also, tell that a partial type can be compiled at the Developer Command Prompt for VS2012. Explain the syntax as given on the slide.

Slides 31 to 33

Understand merged elements during compilation.

Merged Elements during Compilation 2-4

- You can directly run the .exe file to see the required output. This is demonstrated in the following code:

Snippet

```

using System;
using System.Collections.Generic;
using System.Text;
//Stored in StudentDetails.cs file
namespace School
{
    public partial class StudentDetails
    {
        int _rollNo;
        string _studName;
        public StudentDetails(int number, string name)
        {
            _rollNo = number;
            _studName = name;
        }
    }
}
using System;
using System;
using System.Collections.Generic;
using System.Text;
//Stored in Students.cs file
namespace School
{
    public partial class StudentDetails
    {
        public void Display()
        {
    
```

Merged Elements during Compilation 3-4

```

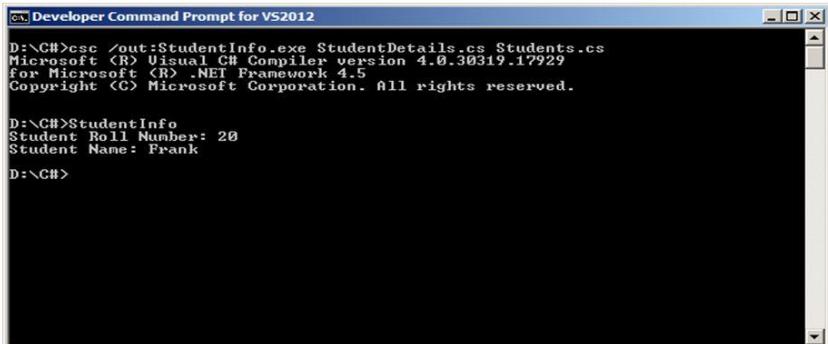
Console.WriteLine("Student Roll Number: " + _rollNo);
Console.WriteLine("Student Name: " + _studName);

}
}
public class Students
{
static void Main(string[] args)
{
    StudentDetails objStudents = new StudentDetails(20,
    "Frank");
    objStudents.Display();
}
}
    
```

- In the code:
 - The partial class **StudentDetails** exists in two different files.
 - When both these files are compiled at the Visual Studio 2005 Command Prompt, an .exe file is created which merges the **StudentDetails** class from both the files.
 - On executing the exe file at the command prompt, the student's roll number and name are displayed as output.

Merged Elements during Compilation 4-4

- The following code shows how to compile and execute the **StudentDetails.cs** and **Students.cs** files created in the examples using Developer Command Prompt for VS2012:



```
D:\>csc /out:StudentInfo.exe StudentDetails.cs Students.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

D:\>StudentInfo
Student Roll Number: 20
Student Name: Frank

D:\>
```

© Aptech Ltd.

Building Applications Using C# / Session 14 33

In slides 31 and 32, tell students how to directly run the .exe file to see the required output. Explain the code. In the code, the partial class **StudentDetails** exists in two different files. When both these files are compiled at the Visual Studio 2005 Command Prompt, an .exe file is created which merges the **StudentDetails** class from both the files.

Mention that on executing the .exe file at the command prompt, the student's roll number and name are displayed as output.

Use slide 33 to refer to the figure that shows how to compile and execute the **StudentDetails.cs** and **Students.cs** files created in the examples using Developer Command Prompt for VS2012.

Slide 34

Understand the rules for partial types.



Rules for Partial Types

- ◆ There are certain rules for creating and working with partial types.
- ◆ These rules must be followed, without which a user might not be able to create partial types successfully.
- ◆ The rules are as follows:
 - ◆ The partial-type definitions must include the `partial` keyword in each file.
 - ◆ The `partial` keyword must always follow the `class`, `struct`, or `interface` keywords.
 - ◆ The partial-type definitions of the same type must be saved in the same assembly.
 - ◆ Generic types can be defined as partial. Here, the type parameters and its order must be the same in all the declarations.
- ◆ The partial-type definitions can contain certain C# keywords which must exist in the declaration in different files. These keywords are as follows:
 - ◆ `public`
 - ◆ `private`
 - ◆ `protected`
 - ◆ `internal`
 - ◆ `abstract`
 - ◆ `sealed`
 - ◆ `new`

© Aptech Ltd. Building Applications Using C# / Session 14 34

In slide 34, explain that there are certain rules for creating and working with partial types. These rules must be followed, without which a user might not be able to create partial types successfully.

Explain the rules. Tell that the partial-type definitions must include the `partial` keyword in each file. Then, tell that the `partial` keyword must always follow the `class`, `struct`, or `interface` keywords.

Also, tell that the partial-type definitions of the same type must be saved in the same assembly. Mention that the generic types can be defined as partial. Here, the type parameters and its order must be the same in all the declarations.

Explain that the partial-type definitions can contain certain C# keywords which must exist in the declaration in different files.

Explain the keywords such as `public`, `private`, `protected`, `internal`, `abstract`, `sealed`, or `new`.

Slide 35

Understand implementing partial types.

The slide has a blue header bar with the title "Implementing Partial Types 1-4". The main content area contains a bulleted list of six points:

- ◆ 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.
- ◆ Each of the partial parts of the code must have the same access modifier.

At the bottom left of the slide is a small watermark that says "© Aptech Ltd.". At the bottom right, there is a footer bar with the text "Building Applications Using C# / Session 14" and the number "35".

Use slide 35 to explain that 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.

Tell that the type names of all the constituent parts of a partial code are prefixed with the `partial` keyword.

Give an example here. Tell that 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.

Slide 36

Understand implementing partial types.

The following syntax is used to split the definition of a class, a struct, or an interface:

Syntax
<code>[<access_modifier>] [<keyword>] partial <type> <Identifier></code>

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 an interface.

In slide 36, explain the syntax that is used to split the definition of a class, a struct, or an interface.

The keyword `partial` indicates that the other parts of `class`, `struct`, or `interface` can be defined in the namespace. Keyword `partial` must be used for all the parts.

If any part is declared `abstract`, then the whole type is considered `abstract`. If any part is declared `sealed`, then the whole type is considered `sealed`. If any part declares a base type, then the whole type inherits that class.

Slide 37

Understand implementing partial types.

The slide has a blue header bar with icons for search, refresh, and help. The title 'Implementing Partial Types 3-4' is in white text on the right. Below the title is a bulleted list: 'The following figure creates an interface with two partial interface definitions:'. A 'Snippet' button is highlighted in blue. The code block contains C# code for 'MathsDemo.cs' and 'MathsDemo2.cs'.

```
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("Addition of two numbers: " +
            objCalculate.Addition(numOne, numTwo));
        Console.WriteLine("Subtraction of two numbers: " +
            objCalculate.Subtraction(numOne, numTwo));
    }
}
```

At the bottom left is a small watermark '© Aptech Ltd.' and at the bottom right are the page numbers 'Building Applications Using C# / Session 14' and '37'.

In slide 37, tell the students that the code creates an interface with two partial interface definitions.

Slide 38

Understand implementing partial types.

Implementing Partial Types 4-4

- ◆ In the code:
 - ◆ 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
```

© Aptech Ltd. Building Applications Using C# / Session 14 38

Use slide 38 to explain the code and the output.

Tell that in the code, 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**.

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.

Slides 39 to 41

Understand partial classes.

Partial Classes 1-3

- ◆ A class is one of the types in C# that supports partial definitions.
- ◆ Classes can be defined over multiple locations to store different members such as variables, methods, and so on.
- ◆ Although the definition of the class is split into different parts stored under different names, all these sections of the definition are combined during compilation to create a single class.
- ◆ You can create partial classes to store private members in one file and public members in another file.
- ◆ More importantly, multiple developers can work on separate sections of a single class simultaneously if the class itself is spread over separate files.
- ◆ The following code creates two partial classes that display the name and roll number of a student:

Snippet

```
using System;
//Program: StudentDetails.cs
public partial class StudentDetails
{
    public void Display()
    {
        Console.WriteLine("Student Roll Number: " + _rollNo);
        Console.WriteLine("Student Name: " + _studName);
    }
}
```

Partial Classes 2-3

```
{
    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();
    }
}
```

Partial Classes 3-3

- ◆ In the code:
 - ❖ The class **StudentDetails** has its definition spread over two files, **StudentDetails.cs** and **StudentDetails2.cs**.
 - ❖ **StudentDetails.cs** contains the part of the class that contains the **Display()** method.
 - ❖ **StudentDetails2.cs** contains the remaining part of the class that includes the constructor.
 - ❖ The class **Students** creates an instance of the class **StudentDetails** and invokes the method **Display**.
 - ❖ The output displays the roll number and the name of the student.

Output

```
Student Roll Number: 20
Student Name: Frank
```

Use slide 39 to explain that 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.

Tell that 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.

Tell that partial classes can be created to store private members in one file and public members in another file. More importantly, multiple developers can work on separate sections of a single class simultaneously, if the class itself is spread over separate files.

In slide 40, tell that the code creates two partial classes that display the name and roll number of a student.

Use slide 41 to explain the that code creates two partial classes that display the name and roll number of a student.

Tell that in the code, the class **StudentDetails** has its definition spread over two files, **StudentDetails.cs** and **StudentDetails2.cs**. **StudentDetails.cs** contains the part of the class that contains the **Display()** method.

Tell that the **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**.

Mention that the output displays the roll number and the name of the student.

Slide 42

Understand partial methods.



Partial Methods 1-7

- ◆ 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.

© Aptech Ltd. Building Applications Using C# / Session 14 42

Use slide 42 to give an example.

Tell them to 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**.

Then, tell that the partial class **Shape** contains the definition of **Create()** in **Shape.cs**.

Mention that 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.

Slide 43

Understand how to create and use partial methods.

The slide has a decorative header with a blue gradient and a subtle circuit board pattern. The title 'Partial Methods 2-7' is centered in a white box. Below the title is a bulleted list of four items. A blue button labeled 'Snippet' is positioned above a code block. The code block contains C# code for a partial class 'Shape' with a single method 'Create'. At the bottom of the slide, there is a footer bar with the text '©Aptech Ltd.' on the left, 'Building Applications Using C# / Session 14' in the center, and the number '43' on the right.

Partial Methods 2-7

- ◆ 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.
- ◆ The following code illustrates how to create and use partial methods.
- ◆ The code contains only the signature and another code contains the implementation:

Snippet

```
using System;
namespace PartialTest
{
    /// <summary>
    /// Class Shape is a partial class and defines a partial method.
    /// </summary>
    public partial class Shape
    {
        partial void Create();
    }
}
```

In slide 43, explain that 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. Explain that the code contains only the signature.

Slides 44 and 45

Understand how to create and use partial methods.

Partial Methods 3-7

```
using System;
namespace PartialTest
{
    /// <summary>
    /// Class Shape is a partial class and contains the implementation
    /// of a partial method.
    /// </summary>
    public partial class Shape
    {
        partial void Create()
        {
            Console.WriteLine("Creating Shape");
        }
        public void Test()
        {
            Create();
        }
    }
    class Program
    {
        static void Main(String[] args)
        {
            Shape s = new Shape();
            s.Test();
        }
    }
}
```

Partial Methods 4-7

- ◆ By separating the definition and implementation into two files, it is possible that two developers can work on them or even use a code-generator tool to create the definition of the method.
- ◆ Also, it is upto the developer whether to implement the partial method or not.
- ◆ It is also valid to have both the signature and implementation of **Create ()** in the same part of **Shape**.

Using slide 44, tell that the code illustrates how to create and use partial methods. The code contains the implementation for the partial methods. Use slide 45 to explain that by separating the definition and implementation into two files, it is possible that two developers can work on them or even use a code-generator tool to create the definition of the method.

Also, tell that it is up to the developer whether to implement the partial method or not. It is also valid to have both the signature and implementation of **Create()** in the same part of **Shape**.

Slides 46 and 47

Understand how to define and implement a method in a single file.

Partial Methods 5-7

- The following figure demonstrates how you can define and implement a method in a single file:

Snippet

```
namespace PartialTest
{
    /// <summary>
    /// Class Shape is a partial class and contains the definition and
    /// implementation of a partial method.
    /// </summary>
    public partial class Shape {
        partial void Create();
        . .
        partial void Create() {
            Console.WriteLine("Creating Shape");
        }
        public void Test() {
            Create();
        }
    }
    class Program {
        static void Main(String[] args) {
            Shape s = new Shape();
            s.Test();
        }
    }
}
```

Partial Methods 6-7

- It is possible to have only the signature of **Create()** in one part of **Shape** and no implementation of **Create()** anywhere.
- In that case, the compiler removes all references to **Create()**, including any method calls.
- A partial method must always include the **partial** keyword.
- Partial methods can be defined only within a partial class or type.
- If the class containing the definition or implementation of a partial method does not have the **partial** keyword, then a compile-time error would be raised.

In slide 46, explain the code that demonstrates how to define and implement a method in a single file. In slide 47, explain that it is possible to have only the signature of `Create()` in one part of `Shape` and no implementation of `Create()` anywhere.

Tell, that the compiler removes all references to `Create()`, including any method calls.

Also, tell that a partial method must always include the `partial` keyword.

Then, mention that partial methods can be defined only within a partial class or type. If the class containing the definition or implementation of a partial method does not have the `partial` keyword, then a compile-time error would be raised.

Slide 48

Understand the restrictions when working with partial methods.

The screenshot shows a presentation slide with a teal header bar. The title 'Partial Methods 7-7' is centered in the header. The main content area contains a bulleted list of restrictions:

- ◆ Some of the restrictions when working with partial methods are as follows:
 - ◆ The `partial` keyword is a must when defining or implementing a partial method
 - ◆ Partial methods must return `void`
 - ◆ They are implicitly `private`
 - ◆ Partial methods can return `ref` but not `out`
 - ◆ Partial methods cannot have any access modifier such as `public`, `private`, and so forth, or keywords such as `virtual`, `abstract`, `sealed`, or so forth
 - ◆ Partial methods are useful when you have part of the code auto-generated by a tool or IDE and want to customize the other parts of the code.

At the bottom of the slide, there is footer text: '© Aptech Ltd.', 'Building Applications Using C# / Session 14', and '48'.

Use slide 48 to tell some of the restrictions when working with partial methods.

Tell that the `partial` keyword is a must when defining or implementing a partial method.

Then, tell that the partial methods must return `void`.

Also tell that they are implicitly `private` and that partial methods can return `ref` but not `out`.

Mention that partial methods cannot have any access modifier such as `public`, `private`, and so forth, or keywords such as `virtual`, `abstract`, `sealed`, or so forth.

Explain that the partial methods are useful when you have part of the code auto-generated by a tool or IDE and want to customize the other parts of the code.

Slide 49

Understand using partial types.

Using Partial Types

- ◆ A large project in an organization involves creation of multiple structures, classes, and interfaces.
- ◆ If these types are stored in a single file, their modification and maintenance becomes very difficult.
- ◆ In addition, multiple programmers working on the project cannot use the file at the same time for modification.
- ◆ Thus, partial types can be used to split a type over separate files, allowing the programmers to work on them simultaneously.
- ◆ Partial types are also used with the code generator in Visual Studio 2012.

© Aptech Ltd. Building Applications Using C# / Session 14 49

In slide 49, tell that a large project in an organization involves creation of multiple structures, classes, and interfaces.

Explain that if these types are stored in a single file, their modification and maintenance becomes very difficult. In addition, multiple programmers working on the project cannot use the file at the same time for modification.

Tell that partial types can be used to split a type over separate files, allowing the programmers to work on them simultaneously.

Also, mention that partial types are also used with the code generator in Visual Studio 2012. You can add the auto-generated code into your file without recreation of the source file. You can use partial types for both these codes.

When a GUI-based project is created using the Windows Forms technology in Visual Studio 2012, a partial class is automatically created to hold the form design.

Slide 50

Understand inheriting partial classes.

The slide has a teal header bar with the title 'Inheriting Partial Classes 1-3'. Below the title is a bulleted list of nine items. At the bottom left is a 'Snippet' button, and at the bottom right is a code block.

Inheriting Partial Classes 1-3

- ◆ You can add the auto-generated code into your file without recreation of the source file.
- ◆ You can use partial types for both these codes.
- ◆ A partial class can be inherited just like any other class in C#.
- ◆ It can contain virtual methods defined in different files which can be overridden in its derived classes.
- ◆ In addition, a partial class can be declared as an abstract class using the `abstract` keyword.
- ◆ Abstract partial classes can be inherited.
- ◆ The following code demonstrate how to inherit a partial class:

Snippet

```
//The following code is stored in Geometry.cs file
using System;
abstract partial class Geometry
{
    public abstract double Area(double val);
}
```

© Aptech Ltd. Building Applications Using C# / Session 14 50

In slide 50, tell the students that a partial class can be inherited just like any other class in C#. Explain that it can contain virtual methods defined in different files which can be overridden in its derived classes.

Also, tell that a partial class can be declared as an abstract class using the `abstract` keyword. Abstract partial classes can be inherited.

Slides 51 and 52

Understand inheriting partial classes.

```
//The following code is stored in Cube.cs file
using System;
abstract partial class Geometry
{
    public virtual void Volume(double val)
    {
    }
}
class Cube : Geometry
{
    public override double Area (double side)
    {
        return 6 * (side * side);
    }
    public override void Volume(double side)
    {
        Console.WriteLine("Volume of cube: " + (side * side));
    }
    static void Main(string[] args)
    {
        double number = 20.56;
        Cube objCube = new Cube();
        Console.WriteLine ("Area of Cube: " +
        objCube.Area(number));
        objCube.Volume(number);
    }
}
```

©Aptech Ltd.

Building Applications Using C# / Session 14

51

◆ In the code:

- ◆ The abstract partial class **Geometry** is defined across two C# files.
- ◆ It defines an abstract method called **Area()** and a virtual method called **Volume()**.
- ◆ Both these methods are inherited in the derived class called **Cube**.

Output

```
Area of Cube: 2536.2816
Volume of cube: 422.7136
```

©Aptech Ltd.

Building Applications Using C# / Session 14

52

In slide 51, explain the code that demonstrates how to inherit partial classes.

Use slide 52 to explain the codes. Tell that in the code, the abstract partial class **Geometry** is defined across two C# files. It defines an abstract method called **Area ()** and a virtual method called **Volume ()**.

Tell that both these methods are inherited in the derived class called **Cube**.

With this slide, you will finish explaining partial types.

Slide 53

Understand nullable types.

The slide has a blue header bar with the title 'Nullable Types'. The main content area contains a bulleted list of 17 points explaining nullable types in C#. At the bottom of the slide, there is a footer bar with the text '© Aptech Ltd.' on the left, 'Building Applications Using C# / Session 14' in the center, and the number '53' on the right.

- ◆ 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.
- ◆ 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.

Use slide 53 to explain that 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.

Tell that the 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.

Also, mention that 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.

Explain that 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.

Then, explain that 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.

Slide 54

Understand the characteristics of nullable types.

The slide features a decorative header with a blue gradient and a subtle circuit board pattern. The word "Characteristics" is centered in white text. The main content area is white with a black border. At the bottom, there's a footer bar with the Aptech logo, the text "Building Applications Using C# / Session 14", and the number "54".

Characteristics

- ◆ Nullable types in C# have the following characteristics:
 - ◆ They represent a value type that can be assigned a null value.
 - ◆ They allow values to be assigned in the same way as that of the normal value types.
 - ◆ They return the assigned or default values for nullable types.
 - ◆ When a nullable type is being assigned to a non-nullable type and the assigned or default value has to be applied, the ?? operator is used.

© Aptech Ltd. Building Applications Using C# / Session 14 54

Use slide 54 to explain the characteristics of nullable types in C#.

Tell that they represent a value type that can be assigned a null value.

Then, tell that they allow values to be assigned in the same way as that of the normal value types.

Mention that they return the assigned or default values for nullable types.

Also, explain that 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.

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.

A nullable type can represent the correct range of values for its underlying value type, plus an additional null value.

Slide 55

Understand implementing nullable types.



Implementing Nullable Types 1-3

- ◆ 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.

© Aptech Ltd. Building Applications Using C# / Session 14 55

Use slide 55 to explain that a nullable type can include any range of values that is valid for the data type to which the nullable type belongs.

Give an example here. Tell 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.

Explain that in the HasValue property, the 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.

Explain that in 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.

Slides 56 and 57

Understand implementing nullable types.

Implementing Nullable Types 2-3

- The following code displays the employee's name, ID, and role using the nullable types:

Snippet

```
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");
        }
    }
}
```

Implementing Nullable Types 3-3

- In the code:
 - 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
```

In slide 56, tell the code that displays the employee's name, ID, and role using the nullable types. Use slide 57 to tell that in the code, **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**.

Also, tell that `role` is defined as a nullable character with `null` value. The output displays the role of the employee as `null`.

Additional Information

For more information on nullable types, refer the following links:

<http://msdn.microsoft.com/en-us/library/1t3y8s4s.aspx>

<http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx>

<http://www.codeproject.com/Articles/275471/Nullable-Types-in-Csharp-Net>

Slides 58 and 59

Understand nullable types in expressions.

Nullable Types in Expressions 1-2

- ◆ C# allows you to use nullable types in expressions that can result in a `null` value.
- ◆ Thus, an expression can contain both, nullable types and non-nullable types.
- ◆ An expression consisting of both, the nullable and non-nullable types, results in the value `null`.
- ◆ The following code demonstrates the use of nullable types in expressions:

Snippet

```
using System;
class Numbers
{
    static void Main (string[] args)
    {
        System.Nullable<int> numOne = 10;
        System.Nullable<int> numTwo = null;
        System.Nullable<int> result = numOne + numTwo;
        if (result.HasValue == true)
        {
            Console.WriteLine("Result: " + result);
        }
        else
        {
            Console.WriteLine("Result: null");
        }
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 14 58

Nullable Types in Expressions 2-2

- ◆ In the code:
 - ❖ **numOne** and **numTwo** are declared as integer variables and initialized to values 10 and null respectively.
 - ❖ In addition, **result** is declared as an integer variable and initialized to a value which is the sum of **numOne** and **numTwo**.
 - ❖ The result of this sum is a null value and this is indicated in the output.

Output

```
Result: null
```

© Aptech Ltd. Building Applications Using C# / Session 14 59

Use slide 58 to explain that C# allows you to use nullable types in expressions that can result in a null value.

Also, tell that an expression can contain both, nullable types and non-nullable types. An expression consisting of both, the nullable and non-nullable types, results in the value `null`. Then, tell that the code demonstrates the use of nullable types in expressions.

Use slide 59 to explain that in the code, **numOne** and **numTwo** are declared as integer variables and initialized to values 10 and null respectively.

Also, tell that **result** is declared as an integer variable and initialized to a value which is the sum of **numOne** and **numTwo**.

Mention that the result of this sum is a null value and this is indicated in the output.

Slide 60

Understand the ?? operator.

The ?? Operator 1-2

- ◆ A nullable type can either have a defined value or the value can be undefined.
- ◆ If a nullable type contains a null value and you assign this nullable type to a non-nullable type, the compiler generates an exception called `System.InvalidOperationException`.
- ◆ To avoid this problem, you can specify a default value for the nullable type that can be assigned to a non-nullable type using the ?? operator.
- ◆ If the nullable type contains a null value, the ?? operator returns the default value.
- ◆ The following code demonstrates the use of ?? operator:

Snippet

```
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);
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 14 60

In slide 60, explain that a nullable type can either have a defined value or the value can be undefined.

Tell that if a nullable type contains a null value and you assign this nullable type to a non-nullable type, the compiler generates an exception called `System.InvalidOperationException`.

Explain that to avoid this problem, you can specify a default value for the nullable type that can be assigned to a non-nullable type using the ?? operator. If the nullable type contains a null value, the ?? operator returns the default value.

Tell that the code demonstrates the use of ?? operator.

Slide 61

Understand the ?? operator.

The slide has a teal header bar with the title 'The ?? Operator 2-2'. Below the header is a white content area. On the left, there is a blue button labeled 'Output'. To the right of the button, the text 'Value: 100.2' and 'Market Value: 0' is displayed. At the bottom of the slide, there is a dark footer bar with the text '©Aptech Ltd.' on the left, 'Building Applications Using C# / Session 14' in the center, and the number '61' on the right.

- ◆ In the code:
 - ❖ 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

©Aptech Ltd. Building Applications Using C# / Session 14 61

Use slide 61 to explain the code. Tell that in the code, 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.

Also, mention that when it is assigned to **marketValue**, a ?? operator has been used. This will assign **marketValue** the default value of 0.0.

Slide 62

Understand converting nullable types.

Converting Nullable Types 1-4

- ◆ C# allows any value type to be converted into nullable type or a nullable type into a value type.
- ◆ C# supports two types of conversions on nullable types:
 - ◆ Implicit conversion
 - ◆ Explicit conversion
- ◆ The storing of a value type into a nullable type is referred to as implicit conversion.
- ◆ A variable to be declared as nullable type can be set to null using the `null` keyword.
- ◆ This is illustrated in the following code:

Snippet

```
using System;
class ImplicitConversion
{
    static void Main(string[] args)
    {
        int? numOne = null;
        if (numOne.HasValue == true)
        {
            Console.WriteLine("Value of numOne before
conversion: " + numOne);
        }
        else
        {
            Console.WriteLine("Value of numOne: null");
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 14 62

Use slide 62 to explain that 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 such as, implicit conversion and explicit conversion.

Tell that the storing of a value type into a nullable type is referred to as implicit conversion. Explain that a variable to be declared as nullable type can be set to null using the `null` keyword.

Tell the code that illustrates converting nullable types.

Slide 63

Understand converting nullable types.

The slide has a teal header bar with the title "Converting Nullable Types 2-4". Below the header is a yellow box containing C# code. The code declares a nullable integer variable `numOne`, initializes it to 20, and then prints its value after implicit conversion. A blue button labeled "Output" is shown below the code, followed by the printed output.

```
numOne = 20;
Console.WriteLine("Value of numOne after implicit
conversion: " + numOne);
}
```

Output

```
Value of numOne: null
Value of numOne after implicit conversion: 20
```

©Aptech Ltd. Building Applications Using C# / Session 14 63

Use slide 63 to explain the code. Tell that in the code, the variable `numOne` is declared as nullable.

Tell that the `HasValue` property is being used to check whether the variable is of a null type.

Then, tell that `numOne` is assigned the value 20, which is of `int` type stored in a nullable type. This is implicit conversion.

Slides 64 and 65

Understand converting nullable types.

Converting Nullable Types 3-4

- ◆ The conversion of a nullable type to a value type is referred to as explicit conversion.
- ◆ This is illustrated in the following code:

Snippet

```
using System;
class ExplicitConversion
{
    static void Main(string[] args)
    {
        int? numOne = null;
        int numTwo = 20;
        int? resultOne = numOne + numTwo;
        if (resultOne.HasValue == true)
        {
            Console.WriteLine("Value of resultOne before conversion: " +
+ resultOne);
        }
        else
        {
            Console.WriteLine("Value of resultOne: null");
        }
        numOne = 10;
        int result = (int)(numOne + numTwo);
        Console.WriteLine("Value of result after implicit conversion: " + result);
    }
}
```

Converting Nullable Types 4-4

- ◆ In the code:
 - ◆ The **numOne** and **resultOne** variables are declared as null.
 - ◆ The **HasValue** property is being used to check whether the **resultOne** variable is of a null type.
 - ◆ Then, **numOne** is assigned the value 10, which is of **int** type stored in a nullable type.
 - ◆ The values in both the variables are added and the result is stored in the **result** variable of **int** type.
 - ◆ This is explicit conversion.

Output

```
Value of resultOne: null
Value of resultTwo after explicit conversion: 30
```

Use slide 64 to tell the code that illustrates the conversion of a nullable type to a value type is referred to as explicit conversion.

In slide 65, explain the code. Tell that in the code, the **numOne** and **resultOne** variables are declared as null. Tell that the **HasValue** property is being used to check whether the **resultOne** variable is of a null type. Then, tell that **numOne** is assigned the value 10, which is of **int** type stored in a nullable type. Also, mention that the values in both the variables are added and the result is stored in the **result** variable of **int** type. This is explicit conversion.

Slides 66 and 67

Understand boxing nullable types.

Boxing Nullable Types 1-2

- An instance of the **Object** type can be created as a nullable type that can hold both null and non-null values.
- The instance can be boxed only if it holds a non-null value and the **HasValue** property returns true.
- In this case, only the data type of the nullable variable is converted to type **Object**.
- While boxing, if the **HasValue** property returns false, the object is assigned a null value.
- The following code demonstrates how to box nullable types:

Snippet

```
using System;
class Boxing
{
    static void Main(string[] args)
    {
        int? number = null;
        object objOne = number;
        if (objOne != null)
        {
            Console.WriteLine("Value of object one: " +
                objOne);
        }
        else
        {
            Console.WriteLine("Value of object one: null");
        }
    }
}
```

Boxing Nullable Types 2-2

```

double? value = 10.26;
object objTwo = value;
if (objTwo != null)
{
    Console.WriteLine("Value of object two: " +
        objTwo);
}
else
{
    Console.WriteLine("Value of object two: null");
}
}

```

- ◆ In the code:
 - ◆ The **number** variable declared as nullable is boxed and its value is stored in **objOne** as null.
 - ◆ The **value** variable declared as nullable is boxed and its value is stored in **objTwo** as 10.26.

Output

```

Value of object one: null
Value of object two: 10.26

```

© Aptech Ltd.

Building Applications Using C# / Session 14

67

Use slide 66 to tell that an instance of the `object` type can be created as a nullable type that can hold both null and non-null values.

Tell that the instance can be boxed only if it holds a non-null value and the `HasValue` property returns true.

Mention that in this case, only the data type of the nullable variable is converted to type `object`. While boxing, if the `HasValue` property returns false, the object is assigned a null value.

Tell that the code demonstrates how to box nullable types.

In slide 67, explain the code. Tell that in the code, the `number` variable declared as nullable is boxed and its value is stored in `objOne` as null.

Tell that the `value` variable declared as nullable is boxed and its value is stored in `objTwo` as 10.26.

Slide 68

Summarize the session.

Summary

- ◆ Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- ◆ Extension methods allow you to extend different types with additional static methods.
- ◆ You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.
- ◆ Partial types allow you to split the definitions of classes, structs, and interfaces to store them in different C# files.
- ◆ You can define partial types using the partial keyword.
- ◆ Nullable types allow you to assign null values to the value types.
- ◆ Nullable types provide two public read-only properties, HasValue and Value.

In slide 68, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- Extension methods allow you to extend different types with additional static methods.
- You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.
- Partial types allow you to split the definitions of classes, structs, and interfaces to store them in different C# files.
- You can define partial types using the partial keyword.
- Nullable types allow you to assign null values to the value types.
- Nullable types provide two public read-only properties, HasValue and Value.

14.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. What are the various elements of a method?
2. Give an example or a scenario where anonymous type can be used.
3. What are partial types attributes?

Session 15 - Advanced Concepts in C#

15.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- Anonymous methods allow you to pass a block of unnamed code as a parameter to a delegate.
- Extension methods allow you to extend different types with additional static methods.
- You can create an instance of a class without having to write code for the class beforehand by using a new feature called anonymous types.
- Partial types allow you to split the definitions of classes, structs, and interfaces to store them in different C# files.
- You can define partial types using the partial keyword.
- Nullable types allow you to assign null values to the value types.
- Nullable types provide two public read-only properties, HasValue and Value.

Here, you can ask students the key topics they can recall from previous session. Ask them to briefly explain anonymous methods in C#. You can also ask them to explain extension methods. Furthermore, ask them to explain anonymous and partial types. Prepare a question or two which will be a key point to relate the current session objectives.

15.1.1 Objectives

By the end of this session, the learners will be able to:

- Describe system-defined generic delegates
- Define lambda expressions
- Explain query expressions
- Describe Windows Communication Framework (WCF)
- Explain parallel programming
- Explain dynamic programming

15.1.2 Teaching Skills

To teach this session successfully, you must know about system-defined generic delegates in C#. You should be aware of lambda and query expressions. You should also know how to describe WCF. Furthermore, you should be aware of parallel and dynamic programming.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here for the In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced to advanced concepts of C#. They will learn about parallel and dynamic programming which includes,

- Lambda expressions
- Query expressions
- Windows Communication Framework (WCF)

15.2 In-Class Explanations

Slides 3 to 5

Understand system-defined generic delegates.

System-Defined Generic Delegates 1-3

- A delegate is a reference to a method. Consider an example to understand this.

Example

- You create a delegate `CalculateValue` to point to a method that takes a `string` parameter and returns an `int`.
- You need not specify the method name at the time of creating the delegate.
- At some later stage in your code, you can instantiate the delegate by assigning it a method name.
- The .NET Framework and C# have a set of predefined generic delegates that take a number of parameters of specific types and return values of another type.
- The advantage of these predefined generic delegates is that they are ready for reuse with minimal coding.

System-Defined Generic Delegates 2-3

- Following are the commonly used predefined generic delegates:

<code>Func<TResult>() Delegate</code>	<code>Func<T, TResult>(T arg) Delegate</code>	<code>Func<T1, T2, TResult>(T1 arg1, T2 arg2) Delegate</code>
• It represents a method having zero parameters and returns a value of type <code>TResult</code> .	• It represents a method having one parameter of type <code>T</code> and returns a value of type <code>TResult</code> .	• It represents a method having two parameters of type <code>T1</code> and <code>T2</code> respectively and returns a value of type <code>TResult</code> .

- The following code determines the length of a given word or phrase:

Syntax

```

/// <summary>
/// Class WordLength determines the length of a given word or phrase
/// </summary>
public class WordLength{
    public static void Main() {
        // Instantiate delegate to reference Count method
        Func<string, int> count = Count;
        string location = "Netherlands";
        // Use delegate instance to call Count method
        Console.WriteLine("The number of characters in the input is:
{0} ",count(location).ToString());
    }
    private static int Count(string inputString) {
        return inputString.Length;
    }
}

```

System-Defined Generic Delegates 3-3

- ◆ The code:
 - ❖ Makes use of the `Func<T, TResult> (T arg)` predefined generic delegate that takes one parameter and returns a result.

- ◆ The following figure shows the use of a predefined generic delegate:

Building Applications Using C# / Session 15 5

Use slide 3 to explain that a delegate is a reference to a method. Give them an example to understand this. Tell them that a delegate **CalculateValue** needs to be created to point to a method that takes a `string` parameter and returns an `int`.

Tell them that the method name need not be specified at the time of creating the delegate. Tell them that at some later stage in your code, instantiate the delegate by assigning it a method name. Explain that the .NET Framework and C# have a set of predefined generic delegates that take a number of parameters of specific types and return values of another type. The advantage of these predefined generic delegates is that they are ready for reuse with minimal coding.

In slide 4, explain to the students some of the commonly used predefined generic delegates. Tell them that `Func<TResult> () Delegate` represents a method having zero parameters and returns a value of type `TResult`. Then, tell them that `Func<T, TResult> (T arg) Delegate` represents a method having one parameter of type `T` and returns a value of type `TResult`.

Also, tell them that `Func<T1, T2, TResult> (T1 arg1, T2 arg2) Delegate` represents a method having two parameters of type `T1` and `T2` respectively and returns a value of type `TResult`. Refer to the figure in slide 4 that shows the use of a predefined generic delegate. Mention that the code determines the number of characters of a given word or phrase. Use slide 5 to tell that the code makes use of the `Func<T, TResult> (T arg)` predefined generic delegate that takes one parameter and returns a result.

Using generic delegates, it is possible to concise delegate type means there is no need to define the delegate statement. These delegates are the `Func<T>`, `Action<T>`, and `Predicate<T>` delegates and defined in the `System` namespace.

Additional Information

For more information on system-defined delegates, refer the following links:

<http://www.informit.com/articles/article.aspx?p=1570280&seqNum=3>

<http://msdn.microsoft.com/en-us/library/bb549151%28v=vs.110%29.aspx>

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



List the commonly used predefined generic delegates.

Answer:

The commonly used predefined generic delegates are:

- Func<TResult> () Delegate
- Func<T, TResult> (T arg) Delegate
- Func<T1, T2, TResult> (T1 arg1, T2 arg2) Delegate

Slides 6 to 8

Understand lambda expressions.

Lambda Expressions 1-3

- ◆ 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.

© Aptech Ltd.

Building Applications Using C# / Session 15

6

Lambda Expressions 2-3

- ◆ The following syntax shows a lambda expression that 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

Example

- ◆ The following code is a lambda expression:
`word => word.Length;`
- ◆ Consider a complete example to illustrate the use of lambda expressions.
- ◆ Assume that you want to calculate the square of an integer number.
- ◆ You can use a method **Square()** and pass the method name as a parameter to the `Console.WriteLine()` method.

© Aptech Ltd.

Building Applications Using C# / Session 15

7

Lambda Expressions 3-3

- The following code uses a lambda expression to calculate the square of an integer number:

Snippet

```
class Program
{
    delegate int ProcessNumber(int input);
    static void Main(string[] args)
    {
        ProcessNumber del = input => input * input;
        Console.WriteLine(del(5));
    }
}
```

- In the code:
 - 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.
 - The following figure shows an example that returns the square of an integer number:

Building Applications Using C# / Session 15 8

Use slide 6 to tell that a method associated with a delegate is never invoked by itself; instead, it is only invoked through the delegate.

Also, tell that sometimes, it can be very cumbersome to create a separate method just so that it can be invoked through the delegate.

Explain that 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.

Mention that a lambda expression is an anonymous expression that can contain expressions and statements and enables to simplify development through inline coding. Tell that 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.

In slide 7, explain the syntax showing a lambda expression that can be used wherever a delegate or anonymous method is expected. Give an example such that the code is a lambda expression. Consider a complete example to illustrate the use of lambda expressions. Assume that you want to calculate the square of an integer number. You can use a method **Square()** and pass the method name as a parameter to the `Console.WriteLine()` method.

If required local functions can be passed as arguments or returned as the value of function calls by lambda expressions. Lambda expressions are particularly helpful for writing LINQ query expressions.

Use slide 8 to explain the students that the code uses a lambda expression to calculate the square of an integer number.

Explain the code. Tell them that 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.

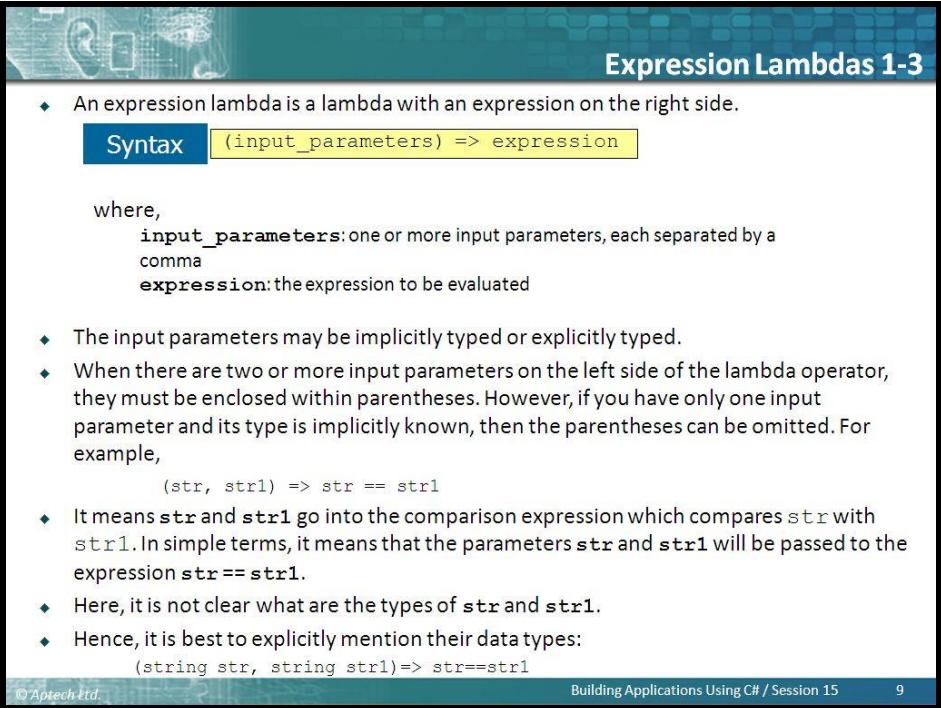
Tell them to refer to the figure in slide 8 that demonstrates an example that returns the square of an integer number. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator =>, and you put the expression or statement block on the other side.

Additional Information

For more information on lambda expressions, refer the following link:
<http://msdn.microsoft.com/en-us/library/bb397687%28v=vs.110%29.aspx>

Slides 9 to 11

Understand expression lambdas.



Expression Lambdas 1-3

- An expression lambda is a lambda with an expression on the right side.

Syntax

```
(input_parameters) => expression
```

where,

- input_parameters:** one or more input parameters, each separated by a comma
- expression:** the expression to be evaluated

- The input parameters may be implicitly typed or explicitly typed.
- When there are two or more input parameters on the left side of the lambda operator, they must be enclosed within parentheses. However, if you have only one input parameter and its type is implicitly known, then the parentheses can be omitted. For example,

```
(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
```

© Aptech Ltd. Building Applications Using C# / Session 15 9

Expression Lambdas 2-3

- ◆ To use a lambda expression:
 - ◆ Declare a delegate type which is compatible with the lambda expression.
 - ◆ Then, create an instance of the delegate and assign the lambda expression to it. After this, you will invoke the delegate instance with parameters, if any.
 - ◆ This will result in the lambda expression being executed. The value of the expression will be the result returned by the lambda.
- ◆ The following code demonstrates expression lambdas:

Snippet

```

/// <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"));
    }
}

```

Building Applications Using C# / Session 15 10

Expression Lambdas 3-3

- ◆ In the code:
 - ◆ 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.
- ◆ The following figure displays the output of using expression lambdas:

Output

In slide 9, explain that an expression lambda is a lambda with an expression on the right side. Tell that the input parameters may be implicitly typed or explicitly typed. Mention that when there are two or more input parameters on the left side of the lambda operator, they must be enclosed within parentheses. Then, tell that if there is no parameter at all, a pair of empty parentheses must be used. Also, tell that if there is only one input parameter and its type is implicitly known, then the parentheses can be omitted.

For example, consider the lambda expression, `str, str1) => str == str1`

Mention that it means **str** and **str1** go into the comparison expression which compares **str** with **str1**. The parameters **str** and **str1** will be passed to the expression **str == str1**. Explain that in this case, it is not clear what are the types of **str** and **str1**. Therefore, it is best to explicitly mention their data types.

In slide 10, explain that to use a lambda expression, declare a delegate type which is compatible with the lambda expression. Tell them to create an instance of the delegate and assign the lambda expression to it.

Then, tell them to 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. Tell that the code demonstrates expression lambdas.

Use slide 11 to explain the code and the corresponding output of the code.

Tell that 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.

Mention that 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. You can refer to the figure in slide 11 that displays the output of using expression lambdas.

Additional Information

For more information on expression lambdas, refer the following link:

<http://msdn.microsoft.com/en-us/library/bb397687%28v=vs.110%29.aspx>

Slide 12

Understand statement lambdas.

Statement Lambdas

- A statement lambda is a lambda with one or more statements. It can include loops, if statements, and so forth.

Syntax

```
(input_parameters) => {statement;}
```

- where,
 - input_parameters**: one or more input parameters, each separated by a comma
 - statement**: a statement body containing one or more statements
 - Optionally, you can specify a return statement to get the result of a lambda.
- The following code demonstrates a statement lambda expression:

Snippet

```
/// <summary>
/// Class WordLength determines the length of a given word or phrase
/// </summary>
public class WordLength{
    // Declare a delegate that has no return value but accepts a string
    delegate void GetLength(string s);
    public static void Main() {
        // Here, the body of the lambda comprises two entire statements
        GetLength len = name => { int n =
            name.Length;Console.WriteLine(n.ToString()); };
        // Invoke the delegate with a string
        len("Mississippi");
    }
}
```

Output

```
C:\Windows\system32\cmd.exe
10
Press any key to continue . . .
```

© Aptech Ltd. Building Applications Using C# / Session 15 12

Use slide 12 to explain that a statement lambda is a lambda with one or more statements. It can include loops, if statements, and so forth.

Tell that a return statement needs to be specified to get the result of a lambda. Explain that the code demonstrates a statement lambda expression. You can refer to the figure in slide 12 that displays the statement lambda.

Additional Information

For more information on statement lambdas, refer the following link:
<http://msdn.microsoft.com/en-us/library/bb397687%28v=vs.110%29.aspx>

Slides 13 and 14

Understand lambdas with standard query operators.

Lambdas with Standard Query Operators 1-2

- ◆ Lambda expressions can also be used with standard query operators.
- ◆ The following table lists the standard query operators:

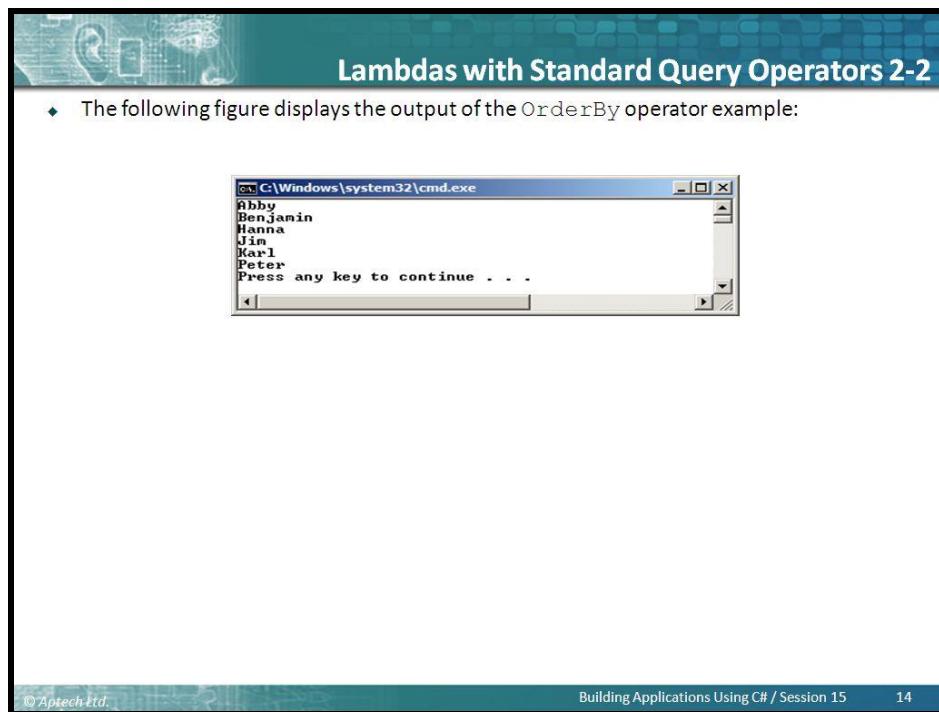
Operator	Description
Sum	Calculates sum of the elements in the expression
Count	Counts the number of elements in the expression
OrderBy	Sorts the elements in the expression
Contains	Determines if a given value is present in the expression

- ◆ The following shows how to use the `OrderBy` operator with the lambda operator to sort a list of names:

Snippet

```
/// <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);
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 15 13



In slide 13, tell the students that lambda expressions can also be used with standard query operators. You can refer to table in slide 13 that lists the standard query operators.

Tell that the code shows how to use the `OrderBy` operator with the lambda operator to sort a list of names.

Use slide 14 to refer to the figure that displays the output of the `OrderBy` operator example. With this slide, you will finish explaining system-defined lambda expressions.

Additional Information

For more information on lambda expressions, refer the following link:

<http://msdn.microsoft.com/en-us/library/bb397687%28v=vs.110%29.aspx>

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is a lambda expression?

Answer:

A lambda expression is an anonymous expression that can contain expressions and statements and enables to simplify development through inline coding.

Slides 15 and 16

Understand query expressions.

Query Expressions 1-3

- ◆ 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 that simplifies working with data present in various formats in different data sources. LINQ provides a consistent model to work with such data.
- ◆ 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.
- ◆ The following code 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':

Snippet

```
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);
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 15

15

Query Expressions 2-3

- ◆ The following displays the query expression example:

Output



```
C:\Windows\system32\cmd.exe
Pearl
Mel
Jill
Karl
Press any key to continue . . .
```

- ◆ 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 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.

© Aptech Ltd.

Building Applications Using C# / Session 15

16

Use slide 15 to explain that 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.

Tell that 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.

Mention that it simplifies working with data present in various formats in different data sources. LINQ provides a consistent model to work with such data.

Then, tell that 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.

Also, tell that the code shows a simple example of a query expression.

Tell that a collection of strings representing names is created and then, a query expression is constructed to retrieve only those names that end with 'l'.

Use slide 16 to refer to the figure that displays the query expression example. Tell that whenever a compiler encounters a query expression, internally it converts it into a method call, using extension methods. Tell that the code demonstrates an expression that is converted appropriately. Explain that though this line is more compact, the SQL way of writing the query expression is more readable and easier to understand.

Additional Information

For more information on lambda expressions in queries, refer the following link:
<http://msdn.microsoft.com/en-us/library/bb397675.aspx>

Slide 17

Understand commonly used query keywords seen in query expressions.


Query Expressions 3-3

- ◆ Some of the commonly used query keywords seen in query expressions are listed in the following table:

Clause	Description
<code>from</code>	Used to indicate a data source and a range variable
<code>where</code>	Used to filter source elements based on one or more boolean expressions that may be separated by the operators <code>&&</code> or <code> </code>
<code>select</code>	Used to indicates how the elements in the returned sequence will look like when the query is executed
<code>group</code>	Used to group query results based on a specified key value
<code>orderby</code>	Used to sort query results in ascending or descending order
<code>ascending</code>	Used in an orderby clause to represent ascending order of sort
<code>descending</code>	Used in an orderby clause to represent descending order of sort

© Aptech Ltd.

Building Applications Using C# / Session 15

17

In slide 17, you can refer to table that lists some of the commonly used query keywords seen in query expressions.

Different query expressions are used more precisely to give specific criteria, condition, to sort out the data in ascending or descending order or group the results as per the specific requirement.

With this slide, you will finish explaining query expressions.

Slide 18

Understand accessing databases using the Entity Framework.

Accessing Databases Using the Entity Framework

- ◆ To persist and retrieve data, an application first needs to connect with the data store.
- ◆ The application must also ensure that the definitions and relationships of classes or objects are mapped with the database tables and table relationships.
- ◆ Finally, the application needs to provide the data access code to persist and retrieve data.
- ◆ These operations can be achieved using ADO.NET, which is a set of libraries that allows an application to interact with data sources.
- ◆ 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 and 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.

.NET Framework

```

graph TD
    ParallelLINQ[Parallel LINQ] --- LINQ[LINQ]
    LINQ --- ADO.NETEntityFramework[ADO.NET Entity Framework]
    ADO.NETEntityFramework --- TaskParallelLibrary[Task Parallel Library]
    ADO.NETEntityFramework --- WPF[WPF]
    ADO.NETEntityFramework --- WCF[WCF]
    ADO.NETEntityFramework --- WF[WF]
    ADO.NETEntityFramework --- CardSpace[CardSpace]
    WPF --- WinForms[WinForms]
    WCF --- ASPNET[ASP.NET]
    WCF --- ADDONET[ADO.NET]
    WinForms --- BaseFrameworkClasses[Base Framework Classes]
    ASPNET --- DLR[DLR]
    ADDONET --- CLR[CLR]
    CLR --- CLS[CLS]
    CLR --- CTS[CTS]
  
```

© Aptech Ltd. Building Applications Using C# / Session 15 18

In slide 18, explain that most of the C# applications need to persist and retrieve data that might be stored in some data source, such as a relational database, XML file, or spreadsheet.

Tell that in an application, data is usually represented in the form of classes and objects. However, in a database, data is stored in the form of tables and views. Therefore, an application in order to persist and retrieve data first needs to connect with the data store.

Mention that the application must also ensure that the definitions and relationships of classes or objects are mapped with the database tables and table relationships.

Also, tell that the application needs to provide the data access code to persist and retrieve data.

All these operations can be achieved using ADO.NET, which is a set of libraries that allows an application to interact with data sources. However, in enterprise data-centric applications, using ADO.NET results in development complexity, which subsequently increases development time and cost. In addition, as the data access code of an application increases, the application becomes difficult to maintain and often leads to performance overhead.

Explain that 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.

Slide 19

Understand the Entity Data Model.

The Entity Data Model

- ◆ The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- ◆ EDM allows a programmer to handle data access logic by programming against entities without having to worry about the structure of the underlying data store and how to connect with it.

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.
- ◆ The figure shows the role of EDM in the Entity Framework architecture:

```

graph TD
    CA[C# Application] <--> EF[Entity Framework]
    subgraph EF [Entity Framework]
        direction TB
        EDM[Entity Data Model (EDM)]
    end
    ADO[ADO.NET] --> D[Database]
    ADO <--> EF
  
```

© Aptech Ltd.

Building Applications Using C# / Session 15 19

Use slide 19 to explain that 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.

Also, tell that 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.

Give an example such that, 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.

You can refer to the figure in slide 19 that shows the role of EDM in the Entity Framework architecture.

In-Class Question:

You will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is an EDM?

Answer:

EDM is a conceptual model that describes the entities and the associations they participate in an application that 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.

Additional Information

The Entity Data Model (EDM) is a set of concepts that describe the structure of data, regardless of its stored form.

Slide 20

Understand development approaches.

Development Approaches

- Entity Framework eliminates the need to write most of the data-access code that otherwise need to be written and uses different approaches to manage data related to an application which are as follows:

The database-first approach <ul style="list-style-type: none"> The Entity Framework creates the data model containing all the classes and properties corresponding to the existing database objects, such as tables and columns.
The model-first approach <ul style="list-style-type: none"> The Entity Framework creates database objects based on the model that a programmer creates to represent the entities and their relationships in the application.
The code-first approach <ul style="list-style-type: none"> The Entity Framework creates database objects based on custom classes that a programmer creates to represent the entities and their relationships in the application.

Use slide 20 to explain that Entity Framework eliminates the need to write most of the data-access code that otherwise need to be written. It uses different approaches to manage data related to an application.

Explain the various development approaches.

Tell that in the database-first approach, the Entity Framework creates the data model containing all the classes and properties corresponding to the existing database objects, such as tables and columns.

Then, tell that in the model-first approach, the Entity Framework creates database objects based on the model that a programmer creates to represent the entities and their relationships in the application.

Also, tell that in the code-first approach, the Entity Framework creates database objects based on custom classes that a programmer creates to represent the entities and their relationships in the application.

Slide 21

Understand the process of creating an Entity Data Model.

Creating an Entity Data Model 1-4

- ◆ Visual Studio 2012 provides support for creating and using EDM in C# application.
- ◆ Programmers can use the Entity Data Model wizard to create a model in an application.
- ◆ After creating a model, programmer can add entities to the model and define their relationship using the Entity Framework designer.
- ◆ The information of the model is stored in an .edmx file.
- ◆ Based on the model, programmer can use Visual Studio 2012 to automatically generate the database objects corresponding to the entities.
- ◆ Finally, the programmer can use LINQ queries against the entities to retrieve and update data in the underlying database.
- ◆ To create an entity data model and generate the database object, a programmer needs to perform the following steps:
 - Open Visual Studio 2012.
 - Create a Console Application project, named **EDMDemo**.
 - Right-click **EDMDemo** in the Solution Explorer window, and select Add → New Item. The Add New Item – **EDMDemo** dialog box is displayed.

In slide 21, explain that Visual Studio 2012 provides support for creating and using EDM in C# application. Programmers can use the Entity Data Model wizard to create a model in an application.

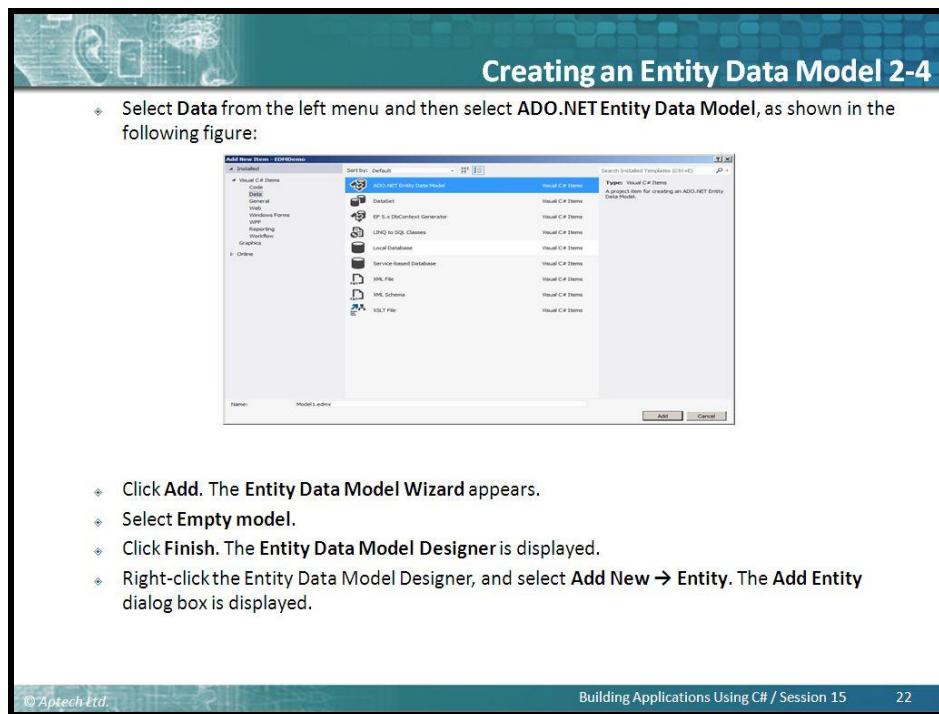
Tell that after creating a model, programmer can add entities to the model and define their relationship using the Entity Framework designer. The information of the model is stored in an .edmx file. Based on the model, programmer can use Visual Studio 2012 to automatically generate the database objects corresponding to the entities.

Also, tell that the programmer can use LINQ queries against the entities to retrieve and update data in the underlying database.

To create an entity data model and generate the database object, a programmer needs to perform some steps. Explain the steps as given on the slide.

Slide 22

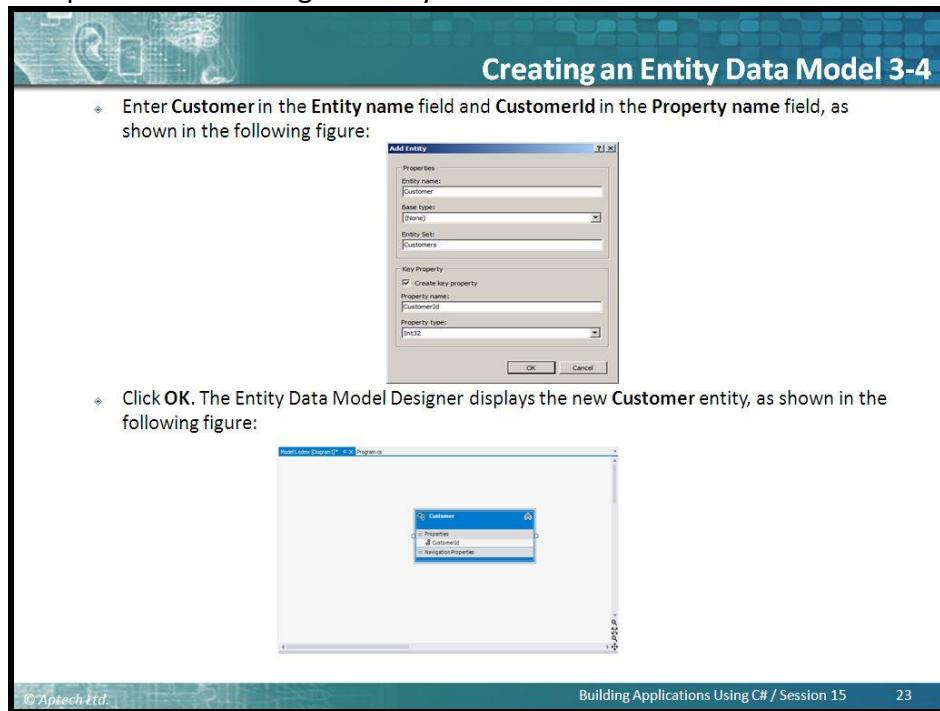
Understand the process of creating an Entity Data Model.



Use slide 22 to refer to the figure and explain the further steps. These steps will help to create an empty data model and add a new entity.

Slide 23

Understand the process of creating an Entity Data Model.



Use slide 23 to refer to the figures and explain the next series of steps that will help to create an entity named **Customer**. Tell the students that once they are familiar with the process of creating a new entity, they can create other entities in a similar manner.

Slide 24

Understand the process of creating an Entity Data Model.

The screenshot shows a slide titled "Creating an Entity Data Model 4-4". The slide content is as follows:

Creating an Entity Data Model 4-4

- Right-click the **Customer** entity and select **Add New → Scalarproperty**.
- Enter **Name** as the name of the property.
- Similarly, add an **Address** property to the **Customer** entity.
- Add another entity named **Order** with an **OrderId** key property.
- Add a **Cost** property to the **Order** entity.

At the bottom of the slide, there is footer text: "© Aptech Ltd.", "Building Applications Using C# / Session 15", and "24".

Use slide 24 to continue to explain the steps that will help to create properties for the entity named **Customer**.

Slides 25 and 26

Understand defining relationships.

Defining Relationships 1-2

- 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:
 - Right-click the Entity Data Model Designer, and select Add New → Association. The Add Association dialog box is displayed.
 - Ensure that the left-hand End section of the relationship point to Customer with a multiplicity of 1 (One) and the right-hand End section point to Post with a multiplicity of *(Many). Accept the default setting for the other fields, as shown in the following figure:

© Aptech Ltd.

Building Applications Using C# / Session 15

25

Defining Relationships 2-2

- Click OK. The Entity Data Model Designer displays the entities with the defined relationship, as shown in the following figure:

© Aptech Ltd.

Building Applications Using C# / Session 15

26

Use slide 25 to explain that 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. Tell that as a customer can have multiple orders, the Customer entity will have a one-to-many relationship with the Order entity.

Explain the steps to create an association between the Customer and Order entities.

You can refer to the figure in slide 25.

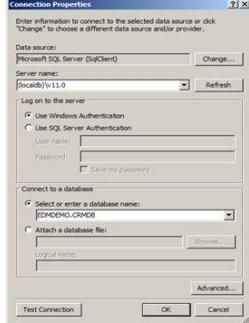
In slide 26, explain the final step and the resulting outcome, that is, the entities and their relationship. You can refer to the figure in slide 26.

Slides 27 and 28

Understand creating database objects.

Creating Database Objects 1-2

- ◆ After designing the model of the application, the programmer needs to generate the database objects based on the model. To generate the database objects:
 - ◆ Right-click the Entity Data Model Designer and select **Generate Database from Model**. The **Generate Database Wizard** dialog box appears.
 - ◆ 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 the following figure:



© Aptech Ltd.

Building Applications Using C# / Session 15

27

Creating Database Objects 2-2

- ◆ Click **OK**. Visual Studio will prompt whether to create a new database.
- ◆ Click **Yes**.
- ◆ Click **Next** in the **Generate Database Wizard** window. Visual Studio generates the scripts to create the database objects.
- ◆ Click **Finish**. Visual Studio opens the file containing the scripts to create the database objects.
- ◆ Right-click the file and select **Execute**. The **Connect to Server** dialog box is displayed.
- ◆ Click **Connect**. Visual Studio creates the database objects.

© Aptech Ltd.

Building Applications Using C# / Session 15

28

Use slide 27 to explain that after designing the model of the application, the programmer needs to generate the database objects based on the model. To generate the database objects tell the steps. You can refer to the figure in slide 27.

Use slide 28 to continue explaining the steps. Tell that,

- Click **OK**, Visual Studio will prompt whether to create a new database.
- Click **Yes**.
- Click **Next** in the **Generate Database Wizard** window. Visual Studio generates the scripts to create the database objects.
- Click **Finish**. Visual Studio opens the file containing the scripts to create the database objects.
- Right-click the file and select **Execute**. The **Connect to Server** dialog box is displayed.
- Click **Connect**. Visual Studio creates the database objects.

Slide 29

Understand using the EDM.

Using the EDM 1-3

- ◆ 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:

Database Context Class	<p>This class extends the <code>DbContext</code> class of the <code>System.Data.Entity</code> namespace to allow a programmer to query and save the data in the database.</p> <p>In the <code>EDMDemo</code> Project, the <code>Model1Container</code> class present in the <code>Model1.Context.cs</code> file is the database context class.</p>	Entity Classes	<p>These classes represent the entities that programmers add and design in the Entity Data Model Designer.</p> <p>In the <code>EDMDemo</code> Project, Customer and Order are the entity classes.</p>
------------------------	--	----------------	---

© Aptech Ltd.

Building Applications Using C# / Session 15 29

Use slide 29 so that when a programmer uses Visual Studio to create an EDM with entities and their relationships, Visual Studio automatically creates several classes.

Explain the important classes that a programmer will use.

Explain that database context 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.

Then, tell that Entity Classes represent the entities that programmers add and design in the Entity Data Model Designer. In the EDMDemo Project, both Customer and Order are the entity classes.

Slides 30 and 31

Understand the Main() method that creates and persists Customer and Order entities.

Using the EDM 2-3

- The following code shows the Main() method that creates and persists Customer and Order entities:

Snippet

```
class Program{
    static void Main(string[] args)
    {
        using (ModelContainer dbContext = new ModelContainer())
        {
            Console.WriteLine("Enter Customer name: ");
            var name = Console.ReadLine();
            Console.WriteLine("Enter Customer Address: ");
            var address = Console.ReadLine();
            Console.WriteLine("Enter Order Cost: ");
            var cost = Console.ReadLine();
            var customer = new Customer { Name=name,Address=address };
            var order = new Order { Cost = cost };
            customer.Orders.Add(order);
            dbContext.Customers.Add(customer);
            dbContext.SaveChanges();
            Console.WriteLine("Customer and Order Information added successfully.");
        }
    }
}
```

Output

```
Enter Customer name: Alex Parker
Enter Customer Address: 10th Park Street, Leo Mount
Enter Order Cost:575
Customer and Order Information added successfully.
```

© Aptech Ltd.

Building Applications Using C# / Session 15

30

Using the EDM 3-3

- The code:
 - Prompts and accept customer and order information from the console.
 - Then, the Customer and Order objects are created and initialized with data. The Order object is added to the Orders property of the Customer object.
 - The Orders property which is of type, `ICollection<Order>` enables adding multiple Order objects to a Customer object, based on the one-to-many relationship that exists between the Customer and Order entities.
 - Then, the database context object of type `ModelContainer` 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.

© Aptech Ltd.

Building Applications Using C# / Session 15

31

In slide 30, tell that the code Main() method creates and persists Customer and Order entities. Use slide 31 to explain the code.

The code prompts and accept customer and order information from the console. Then, the **Customer** and **Order** objects are created and initialized with data.

Tell that 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, tell that 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.

Slide 32

Understand querying data by using LINQ query expressions.

Querying Data by Using LINQ Query Expressions 1-4

- ◆ LINQ provides a consistent programming model to create standard query expression syntax to query different types of data sources.
- ◆ However, different data sources accept queries in different formats. To solve this problem, LINQ provides the various LINQ providers, such as LINQ to Entities, LINQ to SQL, LINQ to Objects, and LINQ to XML.
- ◆ To create and execute queries against the conceptual model of Entity Framework, programmers can use LINQ to Entities. In LINQ to Entities, a programmer creates a query that returns a collection of zero or more typed entities.
- ◆ To create a query, the programmer needs a data source against which the query will execute.
- ◆ An instance of the **ObjectQuery** class represents the data source. In LINQ to entities, a query is stored in a variable. When the query is executed, it is first converted into a command tree representation that is compatible with the Entity Framework.
- ◆ Then, the Entity Framework executes the query against the data source and returns the result.

LINQ providers

LINQ to XML	LINQ to Objects	LINQ to SQL	LINQ to DataSet	LINQ to Entities
-------------	-----------------	-------------	-----------------	------------------	------

Data sources

XML	objects	RDBMS	DataSet	ADO.NET Entity Framework	others
-----	---------	-------	---------	--------------------------	--------

© Aptech Ltd. Building Applications Using C# / Session 15 32

In slide 32 explain that LINQ provides a consistent programming model to create standard query expression syntax to query different types of data sources.

Tell that different data sources accept queries in different formats. To solve this problem, LINQ provides the various LINQ providers, such as LINQ to Entities, LINQ to SQL, LINQ to Objects, and LINQ to XML. To create and execute queries against the conceptual model of Entity Framework, programmers can use LINQ to Entities.

Mention that in LINQ to Entities, a programmer creates a query that returns a collection of zero or more typed entities. To create a query, the programmer needs a data source against which the query will execute.

Also, mention that 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.

Slide 33

Understand how to create and execute a query to retrieve the records of all `Customer` entities along with the associated `Order` entities.

Querying Data by Using LINQ Query Expressions 2-4

◆ The following code creates and executes a query to retrieve the records of all `Customer` entities along with the associated `Order` entities:

Snippet

```
public static void DisplayPropertiesMethodBasedQuery()
{
    public static void DisplayAllCustomers()
    {
        using (Model1Container dbContext = new Model1Container())
        {
            IQueryables<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 the code:

- ◆ 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 `Customer`s 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.

© Aptech Ltd. Building Applications Using C# / Session 15 33

In slide 33, explain the code. In the code, 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 `Customer`s data source, and `c` is the range variable.

Tell that when the query is executed, the range variable acts as a reference to each successive element in the data source.

Also, mention that the `select` clause in the LINQ query specifies the type of the returned elements as an `IQueryable<Customer>` object.

Mention that the `foreach` loops iterate through the results of the query returned as an `IQueryable<Customer>` object to print the customer and order details.

Slide 34

Understand using LINQ to perform various other operations.

Querying Data by Using LINQ Query Expressions 3-4

- In addition to simple data retrieval, programmers can use LINQ to perform various other operations, such as forming projections, filtering data, and sorting data.

- When using LINQ queries, the programmer might only need to retrieve specific properties of an entity from the data store; for example only the **Name** property of the **Customer** entity instances. The programmer can achieve this by forming projections in the `select` clause.

- The following code shows a LINQ query that retrieves only the customer names of the **Customer** entity instances:

Snippet

```
public static void DisplayCustomerNames(){
    using (Model1Container dbContext = new Model1Container()) {
        IQueryables<String> query = from c in dbContext.Customers select c.Name;
        Console.WriteLine("Customer Names:");
        foreach (String custName in query){
            Console.WriteLine(custName);
        }
    }
}
```

- In the code:
 - The `select` method retrieves a sequence of customer names as an `IQueryable<String>` object and the `foreach` loop iterates through the result to print out the names.

Output

Customer Names:
Alex Parker
Peter Milne

© Aptech Ltd.

Building Applications Using C# / Session 15 34

Use slide 34 to explain that in addition to simple data retrieval, programmers can use LINQ to perform various other operations, such as forming projections, filtering data, and sorting data. Tell that for forming projections when using LINQ queries, the programmer might only need to retrieve specific properties of an entity from the data store.

For example, only the **Name** property of the **Customer** entity instances. The programmer can achieve this by forming projections in the `select` clause.

Tell that the code shows a LINQ query that retrieves only the customer names of the **Customer** entity instances.

Then, explain the code and output.

Tell that in the code, 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.

Slide 35

Understand filtering data.

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.

Snippet

```
public static void DisplayCustomerByName() {
    using (ModelContainer dbContext = new ModelContainer()) {
        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);
    }
}
```

Output

```
Customer Information:
Customer ID: 1, Name: Alex Parker, Address: 10th Park Street, Leo Mount
```

© Aptech Ltd. Building Applications Using C# / Session 15 35

In slide 35, explain that the `where` clause in a LINQ query enables filtering data based on a Boolean condition, known as the predicate.

Tell that 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.

Tell the code uses the `where` clause to filter customer records.

Then, explain the code and output.

Tell that the code uses the `where` clause to retrieve information of the customer with the name Alex Parker. The `foreach` statement iterate through the result to print the information of the customer.

Slides 36 and 37

Understand querying data by using LINQ method-based queries.

Querying Data by Using LINQ Method-Based Queries 1-2

- ◆ 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.
- ◆ The following code uses the `Select` method to project the `Name` and `Address` properties of `Customer` entity instances into a sequence of anonymous types:

Snippet

```
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);
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 15 36

Querying Data by Using LINQ Method-Based Queries 2-2

Output

```
Customer Names and Addresses:
Name: Alex Parker, Address: 10th Park Street, Leo Mount
Name: Peter Milne, Address: Lake View Street, Cheros
Mount
```

- ◆ Similarly, you can use the other operators such as `where`, `GroupBy`, `Max`, and so on through method-based queries.

© Aptech Ltd. Building Applications Using C# / Session 15 37

Use slide 36 to explain that the LINQ queries used so far are created using query expression syntax.

Tell that such queries are compiled into method calls to the standard query operators, such as `select`, `where`, and `orderby`.

Explain that 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.

Tell that the code uses the `Select` method to project the `Name` and `Address` properties of `Customer` entity instances into a sequence of anonymous types.

Use slide 37 to explain the output.

Also, tell that other operators can be used such as `Where`, `GroupBy`, `Max`, and so on through method-based queries.

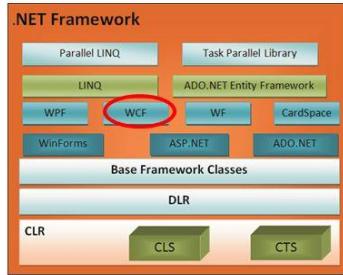
With this slide, you will finish explaining accessing databases using the entity framework.

Slide 38

Understand Web services with windows communication framework (WCF).

Web Services with WCF

- ◆ Windows Communication Foundation (WCF) is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- ◆ SOA is an extension of distributed computing based on the request/response design pattern.
- ◆ SOA allows creating interoperable services that can be accessed from heterogeneous systems.
- ◆ Interoperability enables a service provider to host a service on any hardware or software platform that can be different from the platform on the consumer end.



The diagram illustrates the .NET Framework architecture. At the top is the **.NET Framework**, which contains several layers of components. From top to bottom, the layers are: Parallel LINQ, Task Parallel Library, LINQ, ADO.NET Entity Framework, WPF, WCF (highlighted with a red circle), WF, CardSpace, WinForms, ASP.NET, and ADO.NET. Below these is the **Base Framework Classes** layer, followed by the **DLR** (Dynamic Language Runtime) layer, and finally the **CLR** (Common Language Runtime) layer at the bottom, which contains CLS and CTS.

© Aptech Ltd. Building Applications Using C# / Session 15 38

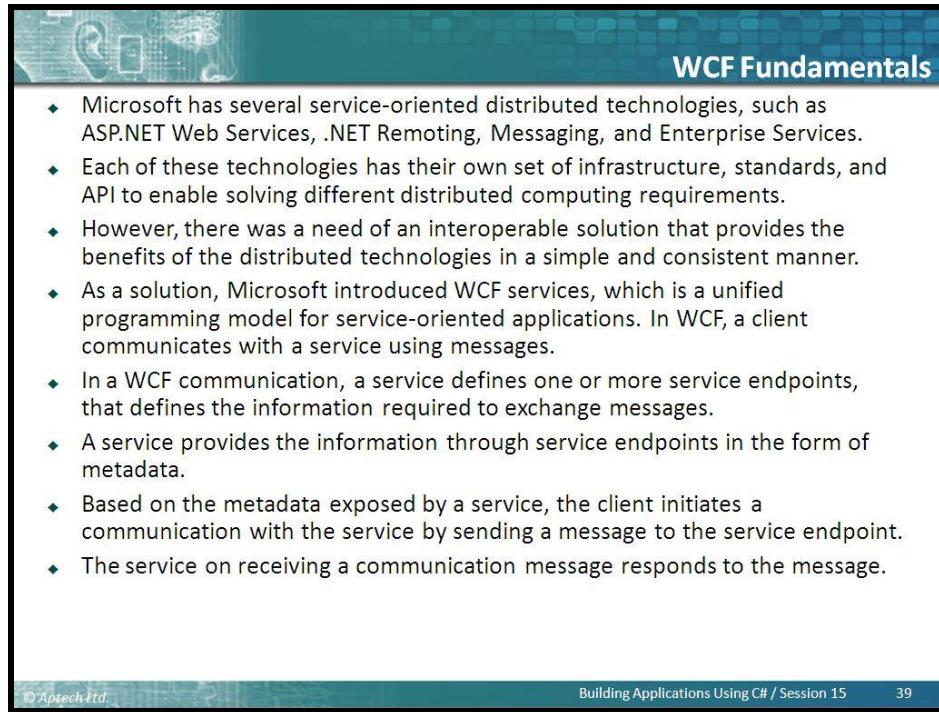
In slide 38, tell the students that WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).

Tell that SOA is an extension of distributed computing based on the request/response design pattern.

Mention that SOA allows creating interoperable services that can be accessed from heterogeneous systems. Interoperability enables a service provider to host a service on any hardware or software platform that can be different from the platform on the consumer end.

Slide 39

Understand WCF fundamentals.



WCF Fundamentals

- ◆ Microsoft has several service-oriented distributed technologies, such as ASP.NET Web Services, .NET Remoting, Messaging, and Enterprise Services.
- ◆ Each of these technologies has their own set of infrastructure, standards, and API to enable solving different distributed computing requirements.
- ◆ However, there was a need of an interoperable solution that provides the benefits of the distributed technologies in a simple and consistent manner.
- ◆ As a solution, Microsoft introduced WCF services, which is a unified programming model for service-oriented applications. In WCF, a client communicates with a service using messages.
- ◆ In a WCF communication, a service defines one or more service endpoints, that defines the information required to exchange messages.
- ◆ A service provides the information through service endpoints in the form of metadata.
- ◆ Based on the metadata exposed by a service, the client initiates a communication with the service by sending a message to the service endpoint.
- ◆ The service on receiving a communication message responds to the message.

© Aptech Ltd. Building Applications Using C# / Session 15 39

In slide 39, explain that Microsoft has several service-oriented distributed technologies, such as ASP.NET Web Services, .NET Remoting, Messaging, and Enterprise Services.

Tell that each of these technologies has their own set of infrastructure, standards, and API to enable solving different distributed computing requirements.

Also, tell that there was a need of an interoperable solution that provides the benefits of the distributed technologies in a simple and consistent manner.

Then, mention that as a solution, Microsoft introduced WCF services, which is a unified programming model for service-oriented applications.

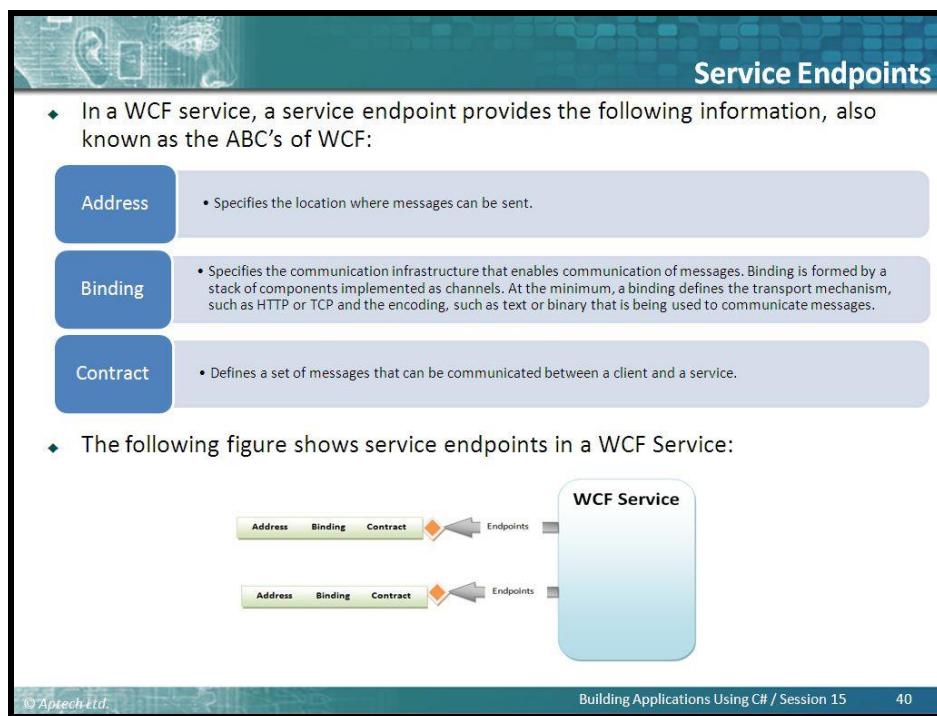
Explain that in WCF, a client communicates with a service using messages. In a WCF communication, a service defines one or more service endpoints, that defines the information required to exchange messages.

Tell that a service provides the information through service endpoints in the form of metadata. Based on the metadata exposed by a service, the client initiates a communication with the service by sending a message to the service endpoint.

Also, mention that the service on receiving a communication message responds to the message.

Slide 40

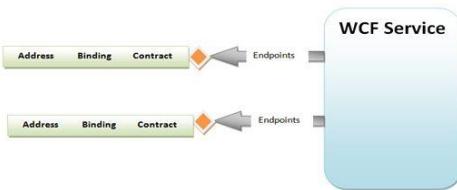
Understand service endpoints.



Service Endpoints

- In a WCF service, a service endpoint provides the following information, also known as the ABC's of WCF:

Address	• Specifies the location where messages can be sent.
Binding	• Specifies the communication infrastructure that enables communication of messages. Binding is formed by a stack of components implemented as channels. At the minimum, a binding defines the transport mechanism, such as HTTP or TCP and the encoding, such as text or binary that is being used to communicate messages.
Contract	• Defines a set of messages that can be communicated between a client and a service.
- The following figure shows service endpoints in a WCF Service:



WCF Service

© Aptech Ltd. Building Applications Using C# / Session 15 40

Use slide 40 to explain that in a WCF service, a service endpoint provides the following information, also known as the ABC's of WCF.

Tell that Address specifies the location where messages can be sent.

Then, tell that Binding specifies the communication infrastructure that enables communication of messages. Binding is formed by a stack of components implemented as channels. At the minimum, a binding defines the transport mechanism, such as HTTP or TCP and the encoding, such as text or binary that is being used to communicate messages.

Also mention that Contract defines a set of messages that can be communicated between a client and a service.

You can refer to the figure in slide 40 that shows service endpoints in a WCF Service.

Slides 41 and 42

Understand WCF contracts.

WCF Contracts 1-4

- WCF operations are based on standard contracts that a WCF service provides that are as follows:
 - Service Contract:**
 - Describes what operations a client can perform on a service.
 - The following code shows a service contract applied to an `IProductService` interface:

Snippet

```
[ServiceContract]
public interface IProductService {
}
```

- The code uses the `ServiceContract` attribute on the `IProductService` interface to specify that the interface will provide one or more operations that client can invoke using WCF.

- Operational Contract:**
 - The following code shows two operational contracts applied to the `IProductService` interface:

Snippet

```
[ServiceContract]
public interface IProductService
{
    [OperationContract]
    String GetProductName (int productId);

    [OperationContract]
    IEnumerable<ProductInformation> GetProductInfo (int productId);
}
```

© Aptech Ltd.

Building Applications Using C# / Session 15

41

WCF Contracts 2-4

- The code:
 - Uses the `OperationContract` attribute on the `GetProductName()` and `GetProductInfo()` methods to specify that these methods can service WCF clients.
 - In the first operational contract applied to the `GetProductName()` method, no additional data contract is required as both the parameter and return types are primitives.
 - However, the second operational contract returns a complex `ProductInfo` type and therefore, must have a data contract defined for it.

Data
Contract
and Data
Members

Specifies how the WCF infrastructure should serialize complex types for transmitting its data between the client and the service. A data contract can be specified by applying the `DataContract` attribute to the complex type, which can be a class, structure, or enumeration. Once a data contract is specified for a type, the `DataMember` attribute must be applied to each member of the type.

© Aptech Ltd.

Building Applications Using C# / Session 15

42

Use slide 41 to explain that WCF operations are based on standard contracts that a WCF service provides. Explain the standard contracts. Tell the students that service contract describes what operations a client can perform on a service. Tell that the code shows a service contract applied to an `IProductService` interface. Then, explain the code.

The code uses the `ServiceContract` attribute on the `IProductService` interface to specify that the interface will provide one or more operations that client can invoke using WCF.

Also, explain operational contract. Tell that the code shows two operational contracts applied to the `IProductService` interface.

Use slide 42 to explain that code uses the `OperationContract` attribute on the `GetProductName()` and `GetProductInfo()` methods to specify that these methods can service WCF clients.

Tell that in the first operational contract applied to the `GetProductName()` method, no additional data contract is required as both the parameter and return types are primitives.

Also, tell that the second operational contract returns a complex `ProductInfo` type and therefore, must have a data contract defined for it.

Then, tell that Data Contract and Data Members specify how the WCF infrastructure should serialize complex types for transmitting its data between the client and the service.

Also, mention that a data contract can be specified by applying the `DataContract` attribute to the complex type, which can be a class, structure, or enumeration. Once a data contract is specified for a type, the `DataMember` attribute must be applied to each member of the type.

Slide 43

Understand the code that uses the `OperationContract` attribute on the `GetProductName()` and `GetProductInfo()` methods.

WCF Contracts 3-4

- The following code shows a data contract applied to the `ProductInformation` class:

Snippet

```
[DataContract]
public class ProductInformation
{
    [DataMember]
    public int ProductId
    {
        get; set;
    }
    [DataMember]
    public String ProductName
    {
        get; set;
    }
    [DataMember]
    public int ProductPrice
    {
        get; set;
    }
}
```

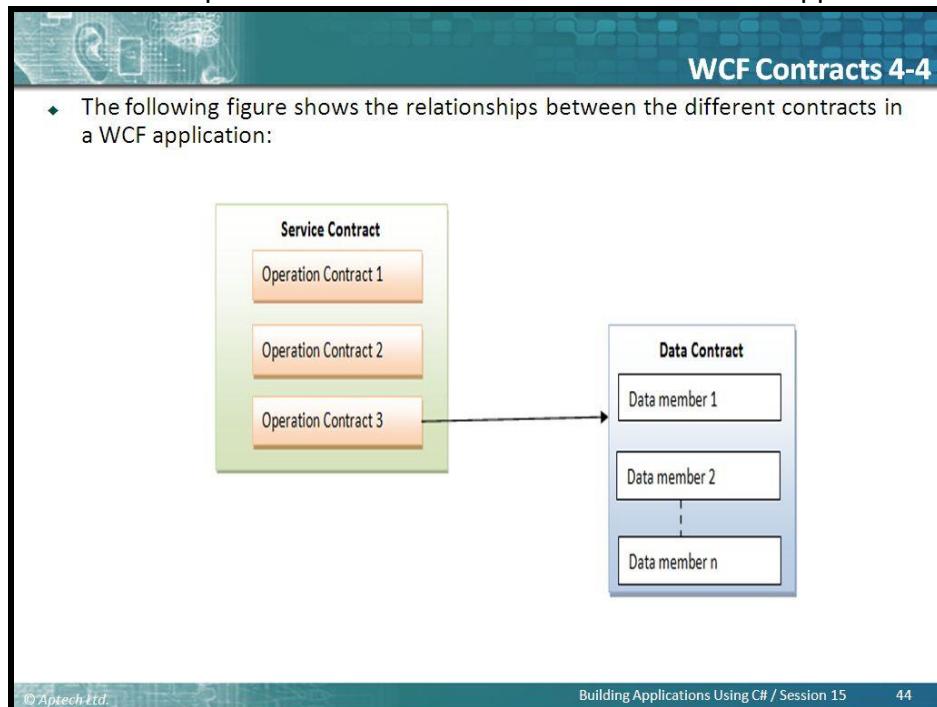
- The code:
 - Applies the `DataContract` attribute to the `ProductInformation` class and the `DataMember` attribute to the `ProductId`, `ProductName`, and `ProductPrice` attributes.

Use slide 43 to tell that the code shows a data contract applied to the `ProductInformation` class.

Explain the code saying that the code applies the `DataContract` attribute to the `ProductInformation` class and the `DataMember` attribute to the `ProductId`, `ProductName`, and `ProductPrice` attributes.

Slide 44

Understand the relationships between the different contracts in a WCF application.



In slide 44, refer to the figure that shows the relationships between the different contracts in a WCF application.

Slides 45 and 46

Understand creating the implementation class.

Creating the Implementation Class 1-2

- ◆ After defining the various contracts in a WCF application, the next step is to create the implementation class. This class implements the interface marked with the `ServiceContract` attribute.
- ◆ The following code shows the `ProductService` class that implements the `IProductService` interface:

```
public class ProductService : IProductService{
    List<ProductInformation> products = new
    List<ProductInformation>() {
        public ProductService() {
            products.Add(new ProductInformation{ ProductId = 001,
                ProductName = "Hard Drive", ProductPrice= 175 });
            products.Add(new ProductInformation { ProductId = 002,
                ProductName = "Keyboard", ProductPrice = 15 });
            products.Add(new ProductInformation { ProductId = 003,
                ProductName = "Mouse", ProductPrice = 15 });
        }

        public string GetProductName(int productId) {
            IEnumerable<string> Product
                = from product in products
                  where product.ProductId == productId
                  select product.ProductName;
            return Product.FirstOrDefault();
        }

        public IEnumerable<ProductInformation> GetProductInfo(int productId) {
            IEnumerable<ProductInformation> Product =
                from product in products
                where product.ProductId == productId
                select product ;
            return Product;
        }
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 15 45

Creating the Implementation Class 2-2

- ◆ The code:
 - ◆ Creates the `ProductService` class that implements the `IProductService` interface marked as the service contract.
 - ◆ The constructor of the `ProductService` class initializes a `List` object with `ProductInformation` objects.
 - ◆ The `ProductService` class implements the `GetProductName()` and `GetProductInfo()` methods marked as operational contract in the `IProductService` interface.
 - ◆ The `GetProductName()` method accepts a product ID and performs a LINQ query on the `List` object to retrieve the name and price of the `ProductInformation` object.
 - ◆ The `GetProductInfo()` method accepts a product ID and performs a LINQ query on the `List` object to retrieve a `ProductInfo` object in an `IEnumerable` object.

© Aptech Ltd. Building Applications Using C# / Session 15 46

In slide 45, explain that after defining the various contracts in a WCF application, the next step is to create the implementation class. This class implements the interface marked with the `ServiceContract` attribute. Tell that the code shows the `ProductService` class that implements the `IProductService` interface.

Use slide 46 to explain that the code creates the **ProductService** class that implements the **IProductService** interface marked as the service contract.

Tell that the constructor of the **ProductService** class initializes a **List** object with **ProductInformation** objects.

Then, tell that the **ProductService** class implements the **GetProductName()** and **GetProductInfo()** methods marked as operational contract in the **IProductService** interface.

Also, tell that the **GetProductName()** method accepts a product ID and performs a LINQ query on the **List** object to retrieve the name and price of the **ProductInformation** object.

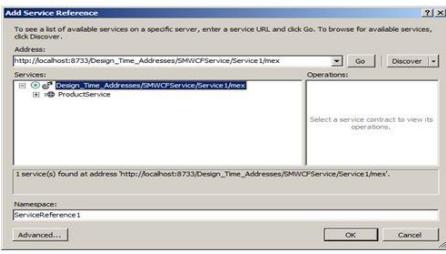
Mention that the **GetProductInfo()** method accepts a product ID and performs a LINQ query on the **List** object to retrieve a **ProductInfo** object in an **IEnumerable** object.

Slides 47 and 48

Understand creating a client to access a WCF service.

Creating a Client to Access a WCF Service 1-2

- ◆ After creating a WCF service, a client application can access the service using a proxy instance.
- ◆ To use the proxy instance and access the service, the client application requires a reference to the service.
- ◆ In the Visual Studio 2012 IDE, you need to perform the following steps to add a service reference to a client project:
 - ◆ In the **Solution Explorer**, right-click the **Service References** node under the project node and select **Add References**. The **Add Service Reference** dialog box is displayed.
 - ◆ Click **Discover**. The **Add Service Reference** dialog box displays the hosted WCF service, as shown in the following figure:



- ◆ Click **OK**. A reference to the WCF service is added to the project.

Creating a Client to Access a WCF Service 2-2

- ◆ The following code shows the Main () method of a client that accesses a WCF service using the proxy instance:

```
ServiceReference1.ProductServiceClient client = new
    ServiceReference1.ProductServiceClient();
Console.WriteLine("Name of product with ID 001");
Console.WriteLine(client.GetProductName(001));

Console.WriteLine("Information of product with ID 002");
ServiceReference1.ProductInformation[] productArr=client.GetProductInfo(002);
foreach(ServiceReference1.ProductInformation product in productArr){
    Console.WriteLine("Product name: {0}, Product price: {1}",
        product.ProductName, product.ProductPrice);
}
Console.ReadLine();
```

- ◆ The code:
 - ❖ Creates a proxy instance of a WCF client of type ServiceReference1.ProductServiceClient.
 - ❖ The proxy instance is then used to invoke the GetProductName () and GetProductInfo () service methods.
 - ❖ The results returned by the service methods are printed to the console.
- ◆ The following figure shows the output of the client application:

Building Applications Using C# / Session 15 48

Use slide 47 to explain that after creating a WCF service, a client application can access the service using a proxy instance.

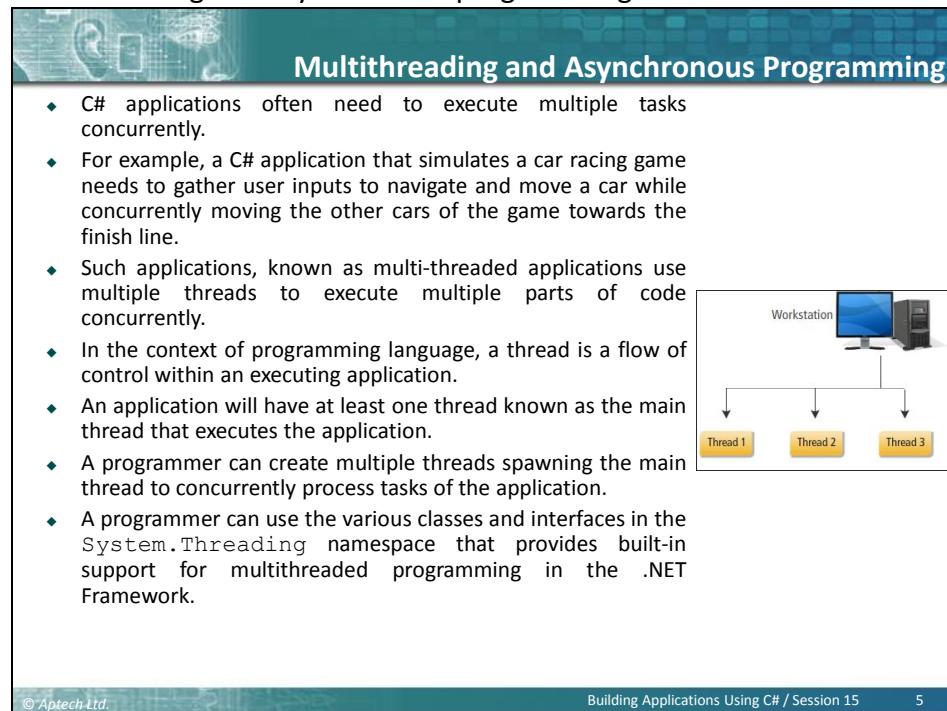
Tell that to use the proxy instance and access the service, the client application requires a reference to the service.

Then, tell the steps used to perform in the Visual Studio 2012 IDE to add a service reference to a client project. Refer to the figure given in slide 47.

Use slide 48 to tell that the code shows the Main () method of a client that accesses a WCF service using the proxy instance. Explain the code. Tell that the code creates a proxy instance of a WCF client of type ServiceReference1.ProductServiceClient. The proxy instance is then used to invoke the GetProductName () and GetProductInfo () service methods. The results returned by the service methods are printed to the console. Mention that the figure in slide 48 shows the output of the client application.

Slide 49

Explain the multithreading and asynchronous programming.



Multithreading and Asynchronous Programming

- ◆ C# applications often need to execute multiple tasks concurrently.
- ◆ For example, a C# application that simulates a car racing game needs to gather user inputs to navigate and move a car while concurrently moving the other cars of the game towards the finish line.
- ◆ Such applications, known as multi-threaded applications use multiple threads to execute multiple parts of code concurrently.
- ◆ In the context of programming language, a thread is a flow of control within an executing application.
- ◆ An application will have at least one thread known as the main thread that executes the application.
- ◆ A programmer can create multiple threads spawning the main thread to concurrently process tasks of the application.
- ◆ A programmer can use the various classes and interfaces in the System.Threading namespace that provides built-in support for multithreaded programming in the .NET Framework.

Workstation

Thread 1 Thread 2 Thread 3

© Aptech Ltd. Building Applications Using C# / Session 15 5

In slide 49, explain that C# applications often need to execute multiple tasks concurrently. Give an example. Tell that a C# application that simulates a car racing game needs to gather user inputs to navigate and move a car while concurrently moving the other cars of the game towards the finish line.

Mention that 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.

Explain that 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.

Also, tell that a programmer can use the various classes and interfaces in the System.Threading namespace that provides built-in support for multithreaded programming in the .NET Framework.

Slides 50 and 51

Understand the Thread class.

The Thread Class 1-2

- ◆ 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.
- ◆ The following code instantiates and starts a new thread:

Snippet

```
ServiceReference1.ProductServiceClient client = new
class ThreadDemo {
public static void Print() {
    while (true)
        Console.WriteLine("1");
}
static void Main (string [] args) {
    Thread newThread = new Thread(new ThreadStart(Print));
    newThread.Start();
    while (true)
        Console.WriteLine("2");
}
}
```

© Aptech Ltd.

Building Applications Using C# / Session 15

50

The Thread Class 2-2

- ◆ In the code:
 - ◆ The **Print()** method uses an infinite while loop to print the value 1 to the console. The **Main()** method instantiates and starts a new thread to execute the **Print()** method.
 - ◆ The **Main()** method then uses an infinite while loop to print the value 2 to the console.
 - ◆ In this program two threads simultaneously executes both the infinite while loops, which results in the output as shown in the following figure:

Output

© Aptech Ltd.

Building Applications Using C# / Session 15

51

Use slide 50 to explain that the Thread class of the System.Threading namespace allows programmers to create and control a thread in a multithreaded application.

Tell that 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.

Also, tell that 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.

Tell that the code instantiates and starts a new thread.

Use slide 51 to explain that the `Print()` method uses an infinite while loop to print the value 1 to the console.

Also, tell that the `Main()` method instantiates and starts a new thread to execute the `Print()` method. The `Main()` method then uses an infinite while loop to print the value 2 to the console. In this program two threads simultaneously executes both the infinite while loops, which results in the output.

You can refer to the figure added in slide 51.

Slide 52

Understand the `ThreadPool` class.

The ThreadPool Class

- ◆ The `System.Threading` namespace provides the `ThreadPool` class to create and share multiple threads as and when required by an application.
- ◆ The `ThreadPool` class represents a thread pool, which is a collection of threads in an application.
- ◆ Based on the request from the application, the thread pool assigns a thread to perform a task.
- ◆ When the thread completes execution, it is put back in the thread pool to be reused for another request.
- ◆ The `ThreadPool` class contains a `QueueUserWorkItem()` method that a programmer can call to execute a method in a thread from the thread pool.
- ◆ This method accepts a `WaitCallback` delegate that accepts `Object` as its parameter.
- ◆ The `WaitCallback` delegate represents the method that needs to execute in a separate thread of the thread pool.

© Aptech Ltd. Building Applications Using C# / Session 15 52

In slide 52, tell the students that the `System.Threading` namespace provides the `ThreadPool` class to create and share multiple threads as and when required by an application.

Then, tell that the `ThreadPool` class represents a thread pool, which is a collection of threads in an application. Based on the request from the application, the thread pool assigns a thread to perform a task. When the thread completes execution, it is put back in the thread pool to be reused for another request.

Also, mention that the `ThreadPool` class contains a `QueueUserWorkItem()` method that a programmer can call to execute a method in a thread from the thread pool. This method accepts a `WaitCallback` delegate that accepts `Object` as its parameter. The `WaitCallback` delegate represents the method that needs to execute in a separate thread of the thread pool.

Slides 53 and 54

Understand synchronizing threads.

Synchronizing Threads 1-2

- ◆ When multiple threads need to share data, their activities needs to be coordinated.
- ◆ This ensures that one thread does not change the data used by the other thread to avoid unpredictable results.
- ◆ For example, consider two threads in a C# program. One thread reads a customer record from a file and the other tries to update the customer record at the same time.
- ◆ In this scenario, the thread that is reading the customer record might not get the updated value as the other thread might be updating the record at that instance.
- ◆ To avoid such situations, C# allows programmers to coordinate and manage the actions of multiple threads at a given time using the following thread synchronization mechanisms.
 - ❖ **Locking using the lock Keyword**
 - Locking is the process that gives control to execute a block of code to one thread at one point of time.
 - The block of code that locking protects is referred to as a critical section. Locking can be implemented using the lock keyword. When using the lock keyword, the programmer needs to pass an object reference that a thread must acquire to execute the critical section.
 - For example, to lock a section of code within an instance method, the reference to the current object can be passed to the lock.

Synchronizing Threads 2-2

- ❖ **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.

Use slide 53 to explain that when multiple threads need to share data, their activities needs to be coordinated. This ensures that one thread does not change the data used by the other thread to avoid unpredictable results. For example, consider two threads in a C# program.

Tell that one thread reads a customer record from a file and the other tries to update the customer record at the same time. In this scenario, the thread that is reading the customer record might not get the updated value as the other thread might be updating the record at that instance. To avoid such situations, C# allows programmers to coordinate and manage the actions of multiple threads at a given time using the following thread synchronization mechanisms.

Tell that 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. Also, tell that locking can be implemented using the lock keyword. When using the lock keyword, the programmer needs to pass an object reference that a thread must acquire to execute the critical section. For example, to lock a section of code within an instance method, the reference to the current object can be passed to the lock.

Use slide 54 to explain synchronization events.

Tell that the locking mechanism used for synchronizing threads is useful for protecting critical sections of code from concurrent thread access.

Then, tell that locking does not allow communication between threads. To enable communication between threads while synchronizing them, C# supports synchronization events.

Also, tell that 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.

Mention that 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.

The `System.Threading` namespace also contains a `ManualResetEvent` class that similar to the `AutoResetEvent` class represents a synchronization event. However, unlike the `AutoResetEvent` class, an object of the `ManualResetEvent` class needs to be manually changed from signaled to un-signaled state by calling its `Reset()` method.

Slide 55

Understand the Task Parallel Library.

The Task Parallel Library

- ◆ Modern computers contain multiple CPUs. In order to take advantage of the processing power that computers with multiple CPUs deliver, a C# application needs to execute tasks in parallel on multiple CPUs. This is known as parallel programming.
- ◆ To make parallel and concurrent programming simpler, the .NET Framework introduced Task Parallel Library (TPL). TPL is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces.

```

graph TD
    PLINQ[PLINQ] --> ParallelAPI[Parallel API]
    ParallelAPI --> TPL[Task Parallel Library (TPL)]
    ParallelAPI --> PLINQ[Parallel LINQ (PLINQ)]
    TPL -.-> Server[Server]
    TPL -.-> Objects[Objects]
    PLINQ --> Databases[Databases]
    PLINQ --> Objects
  
```

Building Applications Using C# / Session 15 55

Use slide 55 to explain that modern computers contain multiple CPUs. In order to take advantage of the processing power that computers with multiple CPUs deliver, a C# application needs to execute tasks in parallel on multiple CPUs. This is known as parallel programming.

Also, mention that 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.

Slides 56 and 57

Understand the Task class.

The Task Class 1-2

- ◆ TPL provides the `Task` class in the `System.Threading.Tasks` namespace. This represents an asynchronous task in a program. Programmers can use this class to invoke a method asynchronously.
- ◆ To create a task, the programmer provides a user delegate that encapsulates the code that the task will execute. The delegate can be a named delegate, such as the `Action` delegate, an anonymous method, or a lambda expression. After creating a `Task`, the programmer calls the `Start()` method to start the task.
- ◆ This method passes the task to the task scheduler that assigns threads to perform the work.
- ◆ To ensure that a task completes before the main thread exits, a programmer can call the `Wait()` method of the `Task` class.
- ◆ To ensure that all the tasks of a program 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.
- ◆ The following code creates and starts two tasks:

Snippet

```
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");
    }
}
```

© Aptech Ltd.

Building Applications Using C# / Session 15

56

The Task Class 2-2

- ◆ In the code:
 - ◆ The `Main()` method creates a `Task`, named `task1` using an `Action` delegate and passing the name of the method to execute asynchronously.
 - ◆ The `Start()` method is called to start the task. The `Run()` method is used to create and start another task, named `task2`. The call to the `Wait()` method ensures that both the tasks complete before the `Main()` method exits.

Output

```
Executed by a Task
Executed by a Task
Exiting main method
```

© Aptech Ltd.

Building Applications Using C# / Session 15

57

Use slide 56 to explain that TPL provides the `Task` class in the `System.Threading.Tasks` namespace. This 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.

Then, tell that after creating a Task, the programmer calls the Start () method to start the task. This method passes the task to the task scheduler that assigns threads to perform the work. To ensure that a task completes before the main thread exits, a programmer can call the Wait () method of the Task class. To ensure that all the tasks of a program completes, the programmer can call the WaitAll () method passing an array of the Tasks objects that have started. Mention that the Task class also provides a Run () method to create and start a task in a single operation. Tell that the code creates and starts two tasks.

Use slide 57 to explain the code and the output. Tell that in the code, 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.

Slide 58

Understand the obtaining results from a task.



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.

© Aptech Ltd.

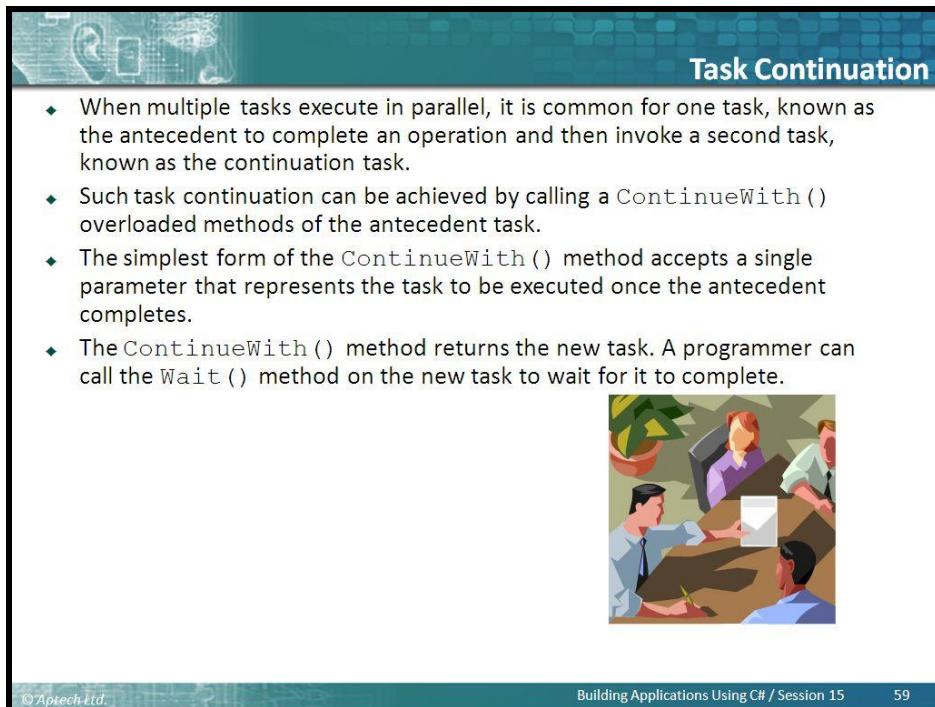
Building Applications Using C# / Session 15 58

In slide 58, tell the students that 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.

Also, tell that 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.

Slide 59

Understand task continuation.



The slide has a teal header bar with the title "Task Continuation". Below the header is a bulleted list of four points explaining task continuation. To the right of the list is a small illustration of four people at a table. At the bottom of the slide is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 15", and "59".

- ◆ When multiple tasks execute in parallel, it is common for one task, known as the antecedent to complete an operation and then invoke a second task, known as the continuation task.
- ◆ Such task continuation can be achieved by calling a `ContinueWith()` overloaded methods of the antecedent task.
- ◆ The simplest form of the `ContinueWith()` method accepts a single parameter that represents the task to be executed once the antecedent completes.
- ◆ The `ContinueWith()` method returns the new task. A programmer can call the `Wait()` method on the new task to wait for it to complete.



© Aptech Ltd. Building Applications Using C# / Session 15 59

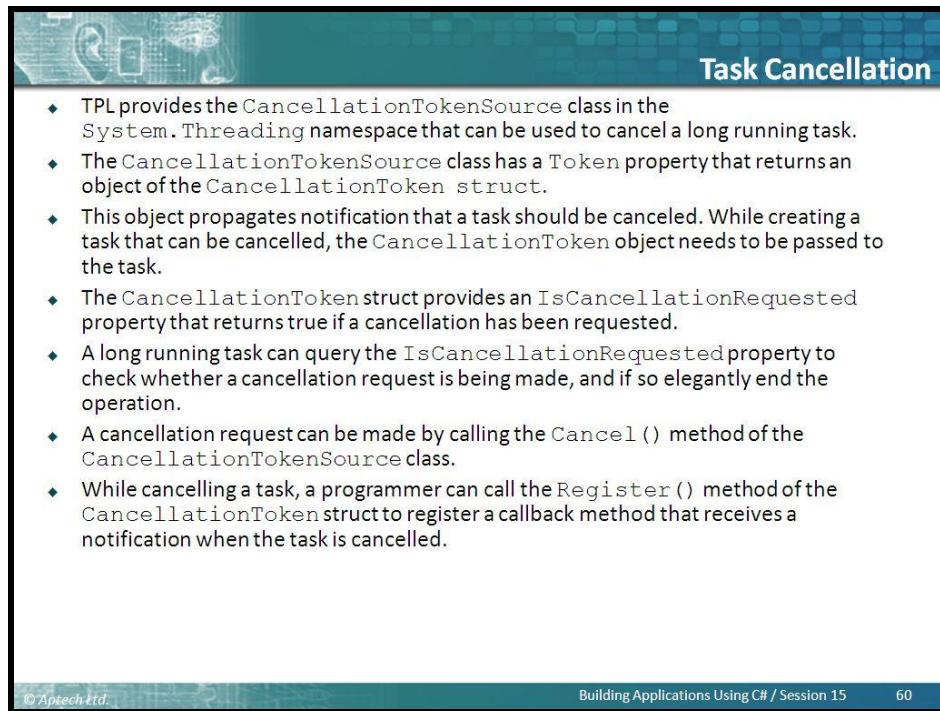
In slide 59, explain that when multiple tasks execute in parallel, it is common for one task, known as the antecedent to complete an operation and then invoke a second task, known as the continuation task.

Tell that 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.

Also, mention that the `ContinueWith()` method returns the new task. A programmer can call the `Wait()` method on the new task to wait for it to complete.

Slide 60

Understand task cancellation.



The slide has a blue header bar with the title "Task Cancellation". The main content area contains a bulleted list of points about the CancellationTokenSource class. At the bottom, there is a footer bar with the copyright information "© Aptech Ltd.", the session title "Building Applications Using C# / Session 15", and the slide number "60".

- ◆ 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 cancelled, the `CancellationToken` object needs to be passed to the task.
- ◆ The `CancellationToken` struct provides an `IsCancellationRequested` property that returns true if a cancellation has been requested.
- ◆ A long running task can query the `IsCancellationRequested` property to check whether a cancellation request is being made, and if so elegantly end the operation.
- ◆ A cancellation request can be made by calling the `Cancel()` method of the `CancellationTokenSource` class.
- ◆ While cancelling 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 cancelled.

Use slide 60 to explain that TPL provides the `CancellationTokenSource` class in the `System.Threading` namespace that can be used to cancel a long running task.

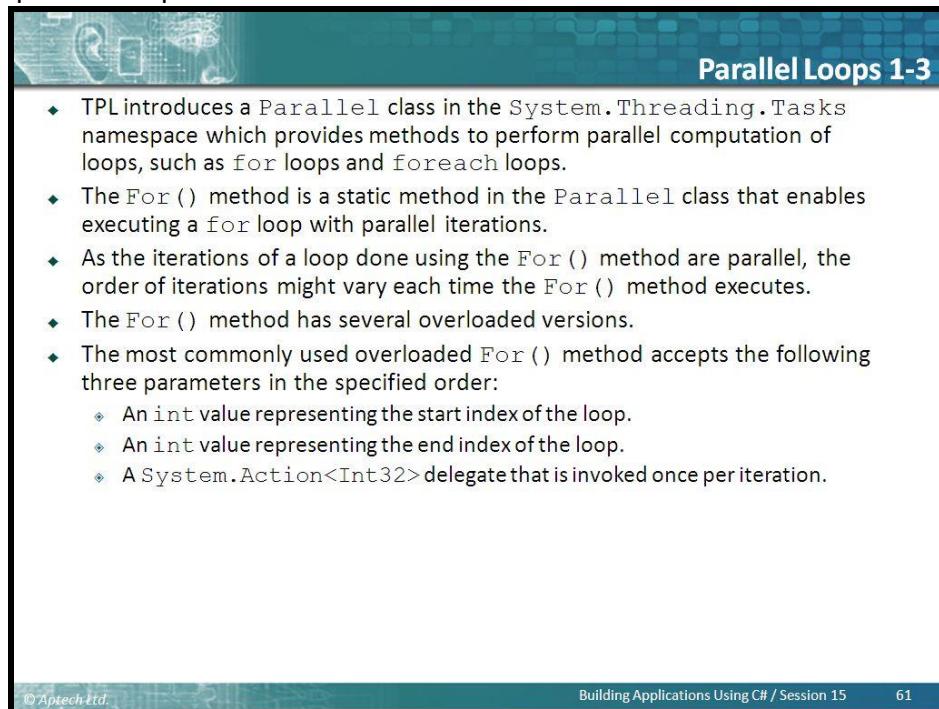
Tell that the `CancellationTokenSource` class has a `Token` property that returns an object of the `CancellationToken` struct. This object propagates notification that a task should be cancelled. While creating a task that can be cancelled, the `CancellationToken` object needs to be passed to the task.

Also, tell that 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.

Mention that while cancelling 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 cancelled.

Slide 61

Understand parallel loops.



Parallel Loops 1-3

- ◆ 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 the following three parameters in the specified order:
 - ◆ An `int` value representing the start index of the loop.
 - ◆ An `int` value representing the end index of the loop.
 - ◆ A `System.Action<Int32>` delegate that is invoked once per iteration.

In slide 61, explain that 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.

Tell that 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.

Explain that the `For()` method has several overloaded versions.

Explain the most commonly used overloaded `For()` method that accepts the three parameters.

- An `int` value representing the start index of the loop.
- An `int` value representing the end index of the loop.
- A `System.Action<Int32>` delegate that is invoked once per iteration.

Slides 62 and 63

Understand the code that uses a traditional for loop and the `Parallel.For()` method.

Parallel Loops 2-3

- The following code uses a traditional for loop and the `Parallel.For()` method:

Snippet

```
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 the code:
 - 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 the figure, 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 the figure multiple threads indicated by the `Thread.CurrentThread.ManagedThreadId` property executes the `for` loop in parallel and the sequence of iteration is unordered.

© Aptech Ltd.

Building Applications Using C# / Session 15

62

Parallel Loops 3-3

- The following figure shows one of the possible outputs of the code:

```
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 2
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 2
i = 9 executed by thread with ID 4
Press any key to continue . . .
```

© Aptech Ltd.

Building Applications Using C# / Session 15

63

In slide 62, explain that the code that uses a traditional for loop and the `Parallel.For()` method. Tell that in the code, 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.

Tell that the `Console.WriteLine()` method prints the results sequentially as a single thread is executing the `for` loop. `Main()` method then performs the same operation using the `Parallel.For()` method. Multiple threads indicated by the `Thread.CurrentThread.ManagedThreadId` property executes the `for` loop in parallel and the sequence of iteration is unordered.

Use slide 63 to refer to the figure that shows one of the possible outputs of code.

Slides 64 and 65

Understand parallel LINQ (PLINQ).

Parallel LINQ (PLINQ) 1-2

- ◆ 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.
- ◆ The following code shows using both a sequential LINQ to Object and PLINQ to query an array:

Snippet

```

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);
}

```

©Aptech Ltd. Building Applications Using C# / Session 15 64

Parallel LINQ (PLINQ) 2-2

- ◆ The code:
 - ◆ 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 `for each` 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
```

© Aptech Ltd. Building Applications Using C# / Session 15 65

Use slide 64 to explain that 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. Tell that 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.

Mention that for parallel access, PLINQ partitions the data source into segments, and then executes each segment through separate threads in parallel. `ParallelEnumerable` class of the `System.Linq` namespace provides methods that implement PLINQ functionality.

Tell that the code shows using both a sequential LINQ to Object and PLINQ to query an array.

Use slide 65 to explain the code and the output.

Tell that the code creates a string array initialized with values.

Then, tell that a sequential LINQ query is used to retrieve the values of the array that are printed to the console in a `for each` loop.

Also, tell that 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. Mention that as the PLINQ query is executed in parallel the order of elements retrieved from the source array is different.

Slide 66

Understand concurrent collections.

The slide has a blue header bar with the title 'Concurrent Collections 1-3'. Below the title is a bulleted list of six points. At the bottom of the slide, there is a table with four rows, each containing a collection class name and its description. The table has a light gray background and is enclosed in a thin black border. The footer of the slide shows the copyright information '© Aptech Ltd.' and the page number '66'.

ConcurrentDictionary<TKey, TValue>	• Is a thread-safe implementation of a dictionary of key-value pairs.
ConcurrentQueue<T>	• Is a thread-safe implementation of a queue.
ConcurrentStack<T>	• Is a thread-safe implementation of a stack.
ConcurrentBag<T>	• Is a thread-safe implementation of an unordered collection of elements.

Use slide 66 to explain that the collection classes of the `System.Collections.Generic` namespace provides improved type safety and performance compared to the collection classes of the `System.Collections` namespace.

Then, tell that the collection classes of the `System.Collections.Generic` namespace are not thread safe. As a result, programmer needs to provide thread synchronization code to ensure integrity of the data stored in the collections. Mention that to address thread safety issues in collections, the .NET Framework provides concurrent collection classes in the `System.Collections.Concurrent` namespace.

Also, tell that these classes being thread safe relieves programmers from providing thread synchronization code when multiple threads simultaneously accesses these collections. Tell the important classes of the `System.Collections.Concurrent` namespace. Tell that the **ConcurrentDictionary<TKey, TValue>** is a thread-safe implementation of a dictionary of key-value pairs. Tell that **ConcurrentQueue<T>** is a thread-safe implementation of a queue.

Also, tell that **ConcurrentStack<T>** is a thread-safe implementation of a stack. Mention that **ConcurrentBag<T>** is a thread-safe implementation of an unordered collection of elements.

Slide 67

Understand how to use multiple threads to add elements to an object of `ConcurrentDictionary<string, int>` class.

The slide has a blue header bar with the title "Concurrent Collections 2-3". Below the header, there is a bullet point followed by a code snippet. The code demonstrates the use of two threads to add 100 integer values to a `ConcurrentDictionary<string, int>`. The `Main` method creates two threads, `thread1` and `thread2`, which both call the `AddToDictionary` method. After the threads have completed, the `Count` property of the dictionary is printed to the console.

```
class CollectionDemo
{
    static ConcurrentDictionary<string, int> dictionary = new
    ConcurrentDictionary<string, int>();
    static void AddToDictionary()
    {
        for (int i = 0; i < 100; i++)
        {
            dictionary.TryAdd(i.ToString(), i);
        }
    }

    static void Main(string[] args)
    {
        Thread thread1 = new Thread(new ThreadStart(AddToDictionary));
        Thread thread2 = new Thread(new ThreadStart(AddToDictionary));
        thread1.Start();
        thread2.Start();
        thread1.Join();
        thread2.Join();
        Console.WriteLine("Total elements in dictionary: {0}",
dictionary.Count());
    }
}
```

© Aptech Ltd. Building Applications Using C# / Session 15 67

In slide 67, explain the code that uses multiple threads to add elements to an object of `ConcurrentDictionary<string, int>` class.

Slide 68

Understand how to use multiple threads to add elements to an object of `ConcurrentDictionary<string, int>` class.

The slide has a title bar 'Concurrent Collections 3-3'. The main content area contains a bulleted list under the heading 'The code:':

- ◆ The code:
 - ❖ Calls the `TryAdd()` method of the `ConcurrentDictionary` class to concurrently add elements to a `ConcurrentDictionary<string, int>` object using two separate threads.
 - ❖ The `TryAdd()` method, unlike the `Add()` method of the `Dictionary` class does not throw an exception if a key already exists.
 - ❖ The `TryAdd()` method instead returns false if a key exist and allows the program to exit normally, as shown in the following figure:

Below the list is a screenshot of a Windows Command Prompt window (cmd.exe) showing the output of the program. The output reads:

```
C:\Windows\system32\cmd.exe
Total elements in dictionary: 100
Press any key to continue . . .
```

At the bottom of the slide, there is footer text: '© Aptech Ltd.' and 'Building Applications Using C# / Session 15 68'.

Use slide 68 to explain the code and output.

Tell that the code calls the `TryAdd()` method of the `ConcurrentDictionary` class to concurrently add elements to a `ConcurrentDictionary<string, int>` object using two separate threads.

Then, tell the `TryAdd()` method, unlike the `Add()` method of the `Dictionary` class does not throw an exception if a key already exists.

Mention that the `TryAdd()` method instead returns false if a key exist and allows the program to exit normally.

You can refer to the figure added in slide 68.

Slide 69

Understand asynchronous methods.

The slide has a blue header bar with the title 'Asynchronous Methods 1-3'. Below the title is a bulleted list of points about TPL's support for asynchronous programming:

- ◆ TPL provides support for asynchronous programming through two new keywords: `async` and `await`.
- ◆ These keywords can be used to asynchronously invoke long running methods in a program.
- ◆ A method marked with the `async` keyword notifies the compiler that the method will contain at least one `await` keyword.
- ◆ If the compiler finds a method marked as `async` but without an `await` keyword, it reports a compilation error.
- ◆ The `await` keyword is applied to an operation to temporarily stop the execution of the `async` method until the operation completes.
- ◆ In the meantime, control returns to the `async` method's caller. Once the operation marked with `await` completes, execution resumes in the `async` method.
- ◆ A method marked with the `async` keyword can have either one of the following return types:
 - ◆ `void`
 - ◆ `Task`
 - ◆ `Task<TResult>`

At the bottom left is a small logo for 'Aptech Ltd.' and at the bottom right are the page numbers 'Building Applications Using C# / Session 15' and '69'.

Use slide 69 to explain that 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.

Tell that 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.

Mention that 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.

Also, tell that a method marked with the `async` keyword can have either one of the return types:

- `Void`
- `Task`
- `Task<TResult>`

By convention, a method marked with the `async` keyword ends with an `Async` suffix.

Slides 70 and 71

Understand the code shows the use of the `async` and `await` keywords.

Asynchronous Methods 2-3

- The following code shows the use of the `async` and `await` keywords:

Snippet

```

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);
    }
}

```

© Aptech Ltd. Building Applications Using C# / Session 15 70

Asynchronous Methods 3-3

- In the code:
 - 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.
- The following figure shows the output:

© Aptech Ltd. Building Applications Using C# / Session 15 71

Use slide 70 to show that the code shows the use of the `async` and `await` keywords. Use slide 71 to explain that the `AnalyzeData()` method of the `ComplexTask` class creates and starts a new task to execute the `GetResult()` method.

Tell that the `GetResult()` method simulates a long running operation by making the thread sleep for two seconds before returning a random number.

Explain that 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.

Then, tell that 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.

Mention that once the `AnalyzeData()` method returns, execution resumes in the `PerformComputationAsync()` method and the retrieved random number is printed on the console.

You can refer to the figure in slide 71 that shows the output. With this slide, you will finish explaining multithreading and asynchronous programming.

Slide 72

Understand dynamic programming.

Dynamic Programming 1-4

- ◆ C# provides dynamic types to support dynamic programming for interoperability of .NET applications with dynamic languages, such as IronPython and 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 Dynamic Language Runtime (DLR).
- ◆ A programmer using a dynamic type is not required to determine the source of the object's value during application development.
- ◆ However, any error that escapes compilation checks causes a run-time exception.
- ◆ To understand how dynamic types bypass compile type checking, consider the following code:

Snippet

```

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();
    }
}

```

©Aptech Ltd. Building Applications Using C# / Session 15 72

In slide 72, explain that C# provides dynamic types to support dynamic programming for interoperability of .NET applications with dynamic languages, such as IronPython and COM APIs such as the Office Automation APIs.

Tell that 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 Dynamic Language Runtime (DLR). Also, tell that a programmer using a dynamic type is not required to determine the source of the object's value during application development. However, any error that escapes compilation checks causes a run-time exception.

Tell that the code helps to understand how dynamic types bypasses compile type checking.

Slide 73

Understand dynamic programming.

Dynamic Programming 2-4

- ◆ In the code:
 - ❖ The **DemoClass** class has a single **Operation ()** method that accepts a **String** parameter.
 - ❖ The **Main ()** method in the **DynamicDemo** class creates a dynamic type and assigns a **DemoClass** object to it.
 - ❖ The dynamic type then calls the **Operation ()** method without passing any parameter.
 - ❖ However, the program compiles without any error as the compiler on encountering the **dynamic** keyword does not performs any type checking.
 - ❖ However, on executing the program, a runtime exception will be thrown, as shown in the following figure:

© Aptech Ltd. Building Applications Using C# / Session 15 73

Use slide 73 to explain the code.

Tell that in the code 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.

Mention that the dynamic type then calls the **Operation ()** method without passing any parameter.

Also, tell that the program compiles without any error as the compiler on encountering the **dynamic** keyword does not performs any type checking.

You can refer to the figure given on the slide that demonstrates that on executing the program, a runtime exception will be thrown.

Slides 74 and 75

Understand `dynamic` keyword.

Dynamic Programming 3-4

- The `dynamic` keyword can also be applied to fields, properties, method parameters, and return types.
- The following code shows how the `dynamic` keyword can be applied to methods to make them reusable in a program:

Snippet

```

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("Hello World");
        dynamic dynaVar3 = DynaMethod(12.5);
        Console.WriteLine("\nReturned dynamic values:\n{0} \n{1} \n{2}", dynaVar1,
        dynaVar2, dynaVar3);
    }
}

```

© Aptech Ltd. Building Applications Using C# / Session 15 74

Dynamic Programming 4-4

- In the code, the `DynaMethod()` method accepts a dynamic type as a parameter. Inside the `DynaMethod()` method, the `is` keyword is used in an if-else-if-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

```

© Aptech Ltd. Building Applications Using C# / Session 15 75

Use slide 74 to explain that the `dynamic` keyword can also be applied to fields, properties, method parameters, and return types.

Tell that the code shows how the `dynamic` keyword can be applied to methods to make them reusable in a program.

Use slide 75 to explain the code and the output.

Tell that in the code, the `DynaMethod()` method accepts a dynamic type as a parameter.

Inside the `DynaMethod()` method, the `is` keyword is used in an `if-else if-else` construct to check for the parameter type and accordingly returns a value.

Mention that 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.

Slide 76

Summarize the session.

Summary

- ◆ System-defined generic delegates take a number of parameters of specific types and return values of another type.
- ◆ A lambda expression is an inline expression or statement block having a compact syntax and can be used in place of a delegate or anonymous method.
- ◆ A query expression is a query that is written in query syntax using clauses such as from, select, and so forth.
- ◆ The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- ◆ WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- ◆ Various classes and interfaces in the `System.Threading` namespace provide built-in support for multithreaded programming in the .NET Framework.
- ◆ To make parallel and concurrent programming simpler, the .NET Framework introduced TPL, which is a set of public types and APIs in the `System.Threading` and `System.Threading.Tasks` namespaces.

© Aptech Ltd. Building Applications Using C# / Session 15 76

In slide 76, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session. Explain each of the following points in brief. Tell them that:

- System-defined generic delegates take a number of parameters of specific types and return values of another type.
- A lambda expression is an inline expression or statement block having a compact syntax and can be used in place of a delegate or anonymous method.

- A query expression is a query that is written in query syntax using clauses such as from, select, and so forth.
- The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- Various classes and interfaces in the `System.Threading` namespace provide built-in support for multithreaded programming in the .NET Framework.
- To make parallel and concurrent programming simpler, the .NET Framework introduced TPL, which is a set of public types and APIs in the `System.Threading` and `System.Threading.Tasks` namespaces.

15.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. What are pre-processor directives?
2. What are event accessors?
3. What are implicit and explicit conversions?

Session 16 - Encrypting and Decrypting Data

16.1 Pre-Class Activities

Before you commence the session, you should revisit the topics of the previous session for a brief review. The summary of the previous session is as follows:

- System-defined generic delegates take a number of parameters of specific types and return values of another type.
- A lambda expression is an inline expression or statement block having a compact syntax and can be used in place of a delegate or anonymous method.
- A query expression is a query that is written in query syntax using clauses such as from, select, and so forth.
- The Entity Framework is an implementation of the Entity Data Model (EDM), which is a conceptual model that describes the entities and the associations they participate in an application.
- WCF is a framework for creating loosely-coupled distributed application based on Service Oriented Architecture (SOA).
- Various classes and interfaces in the `System.Threading` namespace provide built-in support for multithreaded programming in the .NET Framework.
- To make parallel and concurrent programming simpler, the .NET Framework introduced TPL, which is a set of public types and APIs in the `System.Threading` and `System.Threading.Tasks` namespaces.

Here, you can ask students the key topics they can recall from previous session. Ask them to briefly explain anonymous methods in C#. You can also ask them to explain extension methods. Furthermore, ask them to explain anonymous and partial types. Prepare a question or two which will be a key point to relate the current session objectives.

16.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain symmetric encryption.
- Explain asymmetric encryption.
- List the various types in the `System.Security.Cryptography` namespace that supports symmetric and asymmetric encryptions.

16.1.2 Teaching Skills

To teach this session successfully, you must know about symmetric and asymmetric encryption. You should be aware of the various types in the `System.Security.Cryptography` namespace that supports symmetric and asymmetric encryptions.

You should teach the concepts in the theory class using the concepts, tables, and codes provided.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order given here during In-Class activities.

Overview of the Session:

Give the students a brief overview of the current session in the form of session objectives. Show the students slide 2 of the presentation. Tell them that they will be introduced the symmetric and asymmetric encryption. They will learn about the various types in the System.Security.Cryptography namespace that supports symmetric and asymmetric encryptions.

16.2 In-Class Explanations

Slides 3 and 4

Understand cryptography and encryption.

Introducing Cryptography and Encryption 1-2

- ◆ All organizations need to handle sensitive data.
- ◆ This data can be either present in storage or may be exchanged between different entities within and outside the organization over a network.

Example

- ◆ Steve is the CEO of a company.
- ◆ Steve while travelling needs to access performance appraisal reports of the top management of the company.
- ◆ These reports contain data that are confidential and are stored in the company's server.
- ◆ Such data is often prone to misuse either intentionally with malicious intent or unintentionally.
- ◆ To avoid such misuse, there should be some security mechanism that can ensure confidentiality and integrity of data.



© Aptech Ltd.
Building Applications Using C# / Session 16 3

Introducing Cryptography and Encryption 2-2

- ◆ Cryptography is a security mechanism to ensure data confidentiality and integrity.
- ◆ The commonly used ways to secure sensitive data is through encryption.





© Aptech Ltd.
Building Applications Using C# / Session 16 4

In slide 3, explain cryptography and encryption to the students.

 ©Aptech Limited

You can begin with an analogy or example to illustrate the need for security and cryptography. Bring out the need for securing data and explain what can happen to data that is unsecured using some examples.

Tell the students that all organizations need to handle sensitive data. Tell them that this data can be either present in storage or may be exchanged between different entities within and outside the organization over a network.

Give them an example to understand the concepts. Tell the students that Steve is the CEO of a company. Steve while travelling needs to access performance appraisal reports of the top management of the company. Tell them that these reports contain data that are confidential and are stored in the company's server. Such data is often prone to misuse either intentionally with malicious intent or unintentionally.

Mention that to avoid such misuse, there should be some security mechanism that can ensure confidentiality and integrity of data.

Define cryptography as the mechanism or technique that helps to protect data from being viewed by malicious persons, prevents data tampering and helps to check if data has been modified in any way. Cryptography is implemented through standard cryptographic algorithms.

Use slide 4 to explain that cryptography is a security mechanism to ensure data confidentiality and integrity. Then, mention that encryption is one of the commonly used ways to secure sensitive data encryption.

Tell the students that .NET Framework provides implementations of many of these algorithms. Tell the students that many classes are available in

System.Security.Cryptography namespace that manages many details of cryptography including secure encoding and decoding of data, as well as many other operations, such as hashing, random number generation, and message authentication.

Additional Information

For more information on cryptography, refer the following link:

<http://msdn.microsoft.com/en-us/library/92f9ye3s%28v=vs.110%29.aspx>

<http://www.garykessler.net/library/crypto.html>

<http://www.codemag.com/Article/0307051>

Slides 5 and 6

Understand encryption basics.


Encryption Basics 1-2

- ◆ The primary objective of cryptography is to secure data exchanged between entities, each of which can be a person or an application.

Example
- ◆ Consider a scenario:
 - ◆ User A transmits sensitive data to User B.
 - ◆ The transmission needs to be confidential to ensure that even if a third party obtains the data, the data is incomprehensible.
 - ◆ User B before using the data must be sure about the integrity of the data, which means that User B must be sure that the data has not been modified in transit.
 - ◆ User B must be able to authenticate that the data is actually sent by User A.
 - ◆ After sending the data, User A must not be able to deny sending the data to User B.
- ◆ **Encryption:**
 - ◆ Is a cryptographic technique that ensures data confidentiality.
 - ◆ Converts data in plain text to cipher (secretly coded) text.
- ◆ The opposite process of encryption is called decryption, which is a process that converts the encrypted cipher text back to the original plain text.

© Aptech Ltd. Building Applications Using C# / Session 16 5


Encryption Basics 2-2

- ◆ The following figure shows the process of encryption and decryption of a password as an example.

```

graph LR
    A[pass@123] --> B[Encryption]
    B --> C["Q@#123%%^Sdd*"]
    C --> D[Decryption]
    D --> E[pass@123]
  
```

- ◆ In the figure:
 - ◆ The plain text, pass@123 is encrypted to a cipher text.
 - ◆ The cipher text is decrypted back to the original plain text.

© Aptech Ltd. Building Applications Using C# / Session 16 6

Use slide 5 to tell the students that the primary objective of cryptography is to secure data exchanged between entities, each of which can be a person or an application.

To explain the concept, give them an example. Ask them to consider the following scenario.

User A transmits sensitive data to User B. The transmission needs to be confidential to ensure that even if a third party obtains the data, the data is incomprehensible. In addition, User B before using the data must be sure about the integrity of the data, which means that User B must be sure that the data has not been modified in transit. Also, User B must be able to authenticate that the data is actually been send by User A. On the other hand, after sending the data, User A must not be able to deny sending the data to User B.

Explain the students that all the necessary security mechanisms for the data communication can be achieved using cryptographic techniques. Tell the students that encryption is one such cryptographic technique that ensures data confidentiality. Tell them that the encryption converts data in plain text to cipher (secretly coded) text.

Mention them that the opposite process of encryption is called decryption, which is a process that converts the encrypted cipher text back to the original plain text.

You can refer to the figure in slide 6 that shows the process of encryption and decryption of a password as an example. Tell them that the plain text, pass@123 is encrypted to a cipher text and the cipher text is decrypted back to the original plain text.

Additional Information

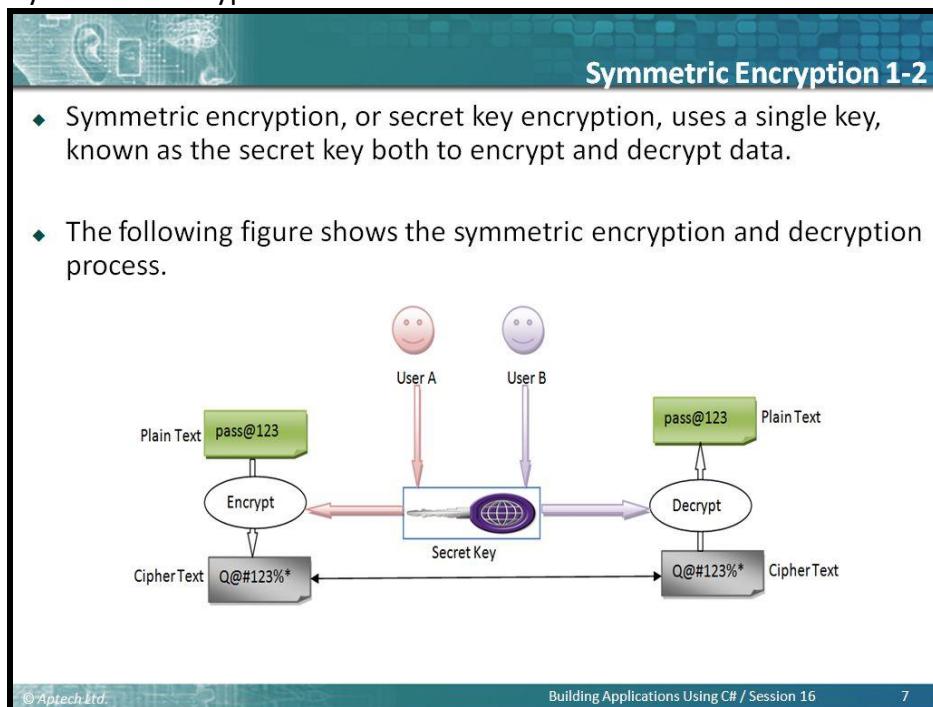
For more information on encryption, refer the following links:

<http://msdn.microsoft.com/en-us/library/92f9ye3s%28v=vs.110%29.aspx>

<http://www.codemag.com/Article/0307051>

Slides 7 and 8

Understand symmetric encryption.





Symmetric Encryption 2-2

- ◆ The following steps outline an example usage of symmetric encryption:
 - ❖ User A uses a secret key to encrypt a plain text to cipher text.
 - ❖ User A shares the cipher text and the secret key with User B.
 - ❖ User B uses the secret key to decrypt the cipher text back to the original plain text.

© Aptech Ltd. Building Applications Using C# / Session 16 8

In slide 7, tell the students that the symmetric encryption is also called the secret key encryption that is used to encrypt and decrypt data.

Symmetric Algorithm class derives a set of classes in which it requires a key and an initialization vector (IV) to perform cryptographic transmissions on data.

Explain the steps to outline an example usage of symmetric encryption. Tell them that a user A uses a secret key to encrypt a plain text to cipher text. Then, tell that user A shares the cipher text and the secret key with User B. Also, tell that user B uses the secret key to decrypt the cipher text back to the original plain text.

You can refer to the figure in slide 8 to understand the symmetric encryption and decryption process.

To make symmetric encryption more secure, an Initialization Vector (IV) can be used with the secret key. An IV is a random number that generates different sequence of encrypted text for identical sequence of text present in the plain text. When you use an IV with a key to symmetrically encrypt data, you will need the same IV and key to decrypt data.

Additional Information

For more information on symmetric encryption, refer the following links:

<http://msdn.microsoft.com/en-us/library/92f9ye3s%28v=vs.110%29.aspx>

<http://www.codemag.com/Article/0307051>

Slides 9 and 10

Understand the asymmetric encryption.

Asymmetric Encryption 1-2

- ◆ Asymmetric encryption uses a pair of public and private key to encrypt and decrypt data.
- ◆ The following figure shows the symmetric encryption and decryption process.

```

graph LR
    PA[User A] --> PKA[Public Key]
    PA --> SKA[Private Key]
    PB[User B] --> PKB[Public Key]
    PKB -- Encrypt --> CT1["Q@#123%*"]
    CT1 --> PT1["pass@123"]
    SKA -- Decrypt --> CT2["Q@#123%*"]
    CT2 --> PT2["pass@123"]
  
```

© Aptech Ltd. Building Applications Using C# / Session 16 9

Asymmetric Encryption 2-2

- ◆ The following steps outline an example usage of asymmetric encryption:
 - ❖ User A generates a public and private key pair.
 - ❖ User A shares the public key with User B.
 - ❖ User B uses the public key to encrypt a plain text to cipher text.
 - ❖ User A uses the private key to decrypt the cipher text back to the original plain text.

© Aptech Ltd. Building Applications Using C# / Session 16 10

In slide 9, mention that asymmetric encryption uses a pair of public and private key to encrypt and decrypt data.

You can refer to the figure in slide 9 to tell that the figure shows the symmetric encryption and decryption process.

Then, tell that asymmetric encryption uses a pair of public and private key to encrypt and decrypt data. Explain the steps to outline an example usage of asymmetric encryption.

Use slide 10 to tell that user A generates a public and private key pair. Tell that user A shares the public key with User B. Mention that user B uses the public key to encrypt a plain text to cipher text. Also, mention that user A uses the private key to decrypt the cipher text back to the original plain text.

With this slide, you will finish explaining cryptography and encryption.

Additional Information

For more information on asymmetric encryption, refer the following link:

<http://msdn.microsoft.com/en-us/library/92f9ye3s%28v=vs.110%29.aspx>

Slide 11

Understand the encryption support in the .NET Framework.

Symmetric Encryption Algorithms 1-6

- ◆ The System.Security.Cryptography namespace provides the SymmetricAlgorithm base class for all symmetric algorithm classes.
- ◆ The derived classes of the SymmetricAlgorithm base class are as follows:

```

graph TD
    RC2[RC2] --- DES[DES]
    DES --- TripleDES[TripleDES]
    TripleDES --- Aes[Aes]
    Aes --- Rijndael[Rijndael]
  
```

© Aptech Ltd. Building Applications Using C# / Session 16 11

In slide 11, tell the students that the .NET Framework provides various types in the System.Security.Cryptography namespace to support symmetric and asymmetric encryptions.

Explain them that the System.Security.Cryptography namespace provides the SymmetricAlgorithm base class for all symmetric algorithm classes.

Tell the students that derived classes of the SymmetricAlgorithm base class are RC2, DES, TripleDES, Aes, and Rijndael.

Additional Information

For more information on asymmetric encryption, refer the following links:

<http://msdn.microsoft.com/en-us/library/system.security.cryptography%28v=vs.110%29.aspx>

<http://www.codeproject.com/Articles/15280/Cryptography-for-the-NET-Framework>

Slide 12

Understand RC2.

The slide has a blue header bar with the title "Symmetric Encryption Algorithms 2-6". Below the header, there is a section titled "RC2" enclosed in a rounded rectangle. A bulleted list provides details about the RC2 algorithm:

- ◆ Is an abstract base class for all classes that implement the RC2 algorithm.
- ◆ Is a proprietary algorithm developed by RSA Data Security, Inc in 1987.
- ◆ Supports key sizes ranging from 40 bits to 128 bits in 8-bit increments for encryption.
- ◆ Was designed for the old generation processors and currently have been replaced by more faster and secure algorithms.
- ◆ Derives the `RC2CryptoServiceProvider` class to provide an implementation of the RC2 algorithm.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 16", and "12".

Use slide 12 to tell the students that RC2 is an abstract base class for all classes that implement the RC2 algorithm.

Tell that this algorithm is a proprietary algorithm developed by RSA Data Security, Inc in 1987. Tell them that the RC2 supports key sizes of from 40 bits to 128 bits in 8-bit increments for encryption.

Mention that the RC2 was designed for the old generation processors and currently have been replaced by faster and secure algorithms.

Explain that the `RC2CryptoServiceProvider` class derives from the RC2 class to provide an implementation of the RC2 algorithm.

Additional Information

For more information on RC2, refer the following link:

<http://msdn.microsoft.com/en-us/library/system.security.cryptography.rc2%28v=vs.110%29.aspx>

Slide 13

Understand DES.

The slide has a decorative header bar with a digital circuit pattern. The title 'Symmetric Encryption Algorithms 3-6' is centered at the top. Below the title, there is a green rounded rectangle containing the word 'DES'. A bulleted list follows, describing the DES algorithm:

- ◆ Is an abstract base class for all classes that implement the Data Encryption Standard (DES) algorithm.
- ◆ Developed by IBM but as of today available as a U.S. Government Federal Information Processing Standard (FIPS 46-3).
- ◆ Works on the data to encrypt as blocks where each block is of 64 bits.
- ◆ Uses a key of 64 bits to perform the encryption. On account of its small key size DES encrypts data faster as compared to other asymmetric algorithms.
- ◆ Is prone to brute force security attacks because of its smaller key size.
- ◆ Derives the `DESCryptoServiceProvider` class to provide an implementation of the DES algorithm.

At the bottom of the slide, there is a footer bar with the text '©Aptech Ltd.' on the left, 'Building Applications Using C# / Session 16' in the center, and the number '13' on the right.

Use slide 13 to explain to the students that the `DES` is an abstract base class for all classes that implement the Data Encryption Standard (DES) algorithm.

Tell them that this algorithm was developed by IBM but as of today available as a U.S. Government Federal Information Processing Standard (FIPS 46-3).

Mention that the DES algorithm works on the data to encrypt as blocks where each block is of 64 bit. To perform the encryption, DES uses a key of 64 bits. Because of its small key size DES encrypts data faster as compared to other asymmetric algorithms.

Also, tell that DES is prone to brute force security attacks because of its smaller key size. The `DESCryptoServiceProvider` class derives from the `DES` class to provide an implementation of the DES algorithm.

Additional Information

For more information on `DES`, refer the following link:

<http://msdn.microsoft.com/en-us/library/system.security.cryptography.des%28v=vs.110%29.aspx>

Slide 14

Understand the TripleDES.

The screenshot shows a slide from a presentation titled "Symmetric Encryption Algorithms 4-6". The main content is a bulleted list under the heading "TripleDES".

TripleDES

- ◆ Is an abstract base class for all the classes that implement the TripleDES algorithm.
- ◆ Is an enhancement to the DES algorithm for the purpose of making the DES algorithm more secured against security threats.
- ◆ Works on 64 bit blocks that is similar to the DES algorithm.
- ◆ Supports key sizes of 128 bits to 192 bits.
- ◆ Derives the `TripleDESCryptoServiceProvider` class to provide an implementation of the TripleDES algorithm.

At the bottom of the slide, there is footer text: "© Aptech Ltd.", "Building Applications Using C# / Session 16", and "14".

In slide 14, explain the TripleDES.

Explain to the students that the `TripleDES` is an abstract base class for all classes that implement the `TripleDES` algorithm.

Tell that this algorithm is an enhancement to the DES algorithm for the purpose of making the DES algorithm more secure against security threats. Similar to the DES algorithm, the `TripleDES` algorithm also works on 64 bit blocks.

Also, mention that the `TripleDES` algorithm supports key sizes of 128 bits to 192 bits. The `TripleDESCryptoServiceProvider` class derives from the `TripleDES` class to provide an implementation of the `TripleDES` algorithm.

Additional Information

For more information on `TripleDES`, refer the following link:

<http://msdn.microsoft.com/en-us/library/system.security.cryptography.tripledes%28v=vs.110%29.aspx>

Slide 15

Understand Aes.

The screenshot shows a slide titled "Symmetric Encryption Algorithms 5-6". A yellow box highlights the word "Aes". Below it is a bulleted list:

- ◆ Is an abstract base class for all classes that implement the Advanced Encryption Standard (AES) algorithm.
- ◆ Is a successor of DES and is currently considered as one of the most secure algorithm.
- ◆ Is more efficient in encrypting large volume of data in the size of several gigabytes.
- ◆ Works on 128-bits blocks of data and key sizes of 128, 192, or 256 bits for encryption.
- ◆ Derives the `AesCryptoServiceProvider` and `AesManaged` classes to provide an implementation of the AES algorithm.

At the bottom left is the copyright notice "© Aptech Ltd.", at the bottom center "Building Applications Using C# / Session 16", and at the bottom right "15".

In slide 15, explain to the students that the `Aes` is an abstract base class for all classes that implement the Advanced Encryption Standard (AES) algorithm.

Tell them that this algorithm is a successor of DES and is currently considered as one of the most secure algorithm. In addition, AES is more efficient in encrypting large volume of data in the size of several gigabytes.

Mention that AES works on 128-bits blocks of data and key sizes of 128, 192, or 256 bits for encryption. The `AesCryptoServiceProvider` and `AesManaged` classes derive from the `Aes` class to provide an implementation of the AES algorithm.

Additional Information

For more information on AES, refer the following links:

<http://www.codeproject.com/Articles/15280/Cryptography-for-the-NET-Framework>

<http://stephenhaunts.com/2013/03/04/cryptography-in-net-advanced-encryption-standard-aes/>

Slide 16

Understand the Rijndael.

The slide has a blue header bar with the title "Symmetric Encryption Algorithms 6-6". Below the header, there is a section titled "Rijndael" enclosed in an orange rounded rectangle. A bulleted list provides details about the Rijndael class:

- ◆ Is an abstract base class for all the classes that implement the Rijndael algorithm.
- ◆ Is a superset of the Aes algorithm.
- ◆ Supports key sizes of 128, 192, or 256 bits similar to the Aes algorithm.
- ◆ Can have block sizes of 128, 192, or 256 bits, unlike Aes, which has a fixed block size of 128 bits.
- ◆ Provides the flexibility to select an appropriate block size based on the volume of data to encrypt by supporting different block sizes.
- ◆ Derives the RijndaelManaged class to provide an implementation of the Rijndael algorithm.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 16", and "16".

Use slide 16 to tell the students that the Rijndael is an abstract base class for all classes that implement the Rijndael algorithm. This algorithm is a superset of the Aes algorithm.

Tell that similar to the Aes algorithm, Rijndael supports key sizes of 128, 192, or 256 bits. However, unlike Aes which has a fixed block size of 128 bits; Rijndael can have block sizes of 128, 192, or 256 bits.

Also, explain that by supporting different block sizes, Rijndael provides the flexibility to select an appropriate block size based on the volume of data to encrypt. The RijndaelManaged class derives from the Rijndael class to provide an implementation of the Rijndael algorithm.

Slide 17

Understand asymmetric encryption algorithm.



Asymmetric Encryption Algorithm 1-2

- ◆ The `System.Security.Cryptography` namespace provides the `AsymmetricAlgorithm` base class for all asymmetric algorithm classes.
- ◆ RSA is an abstract class that derives from the `AsymmetricAlgorithm` class.
- ◆ The RSA class is the base class for all the classes that implement the RSA algorithm.
- ◆ The RSA algorithm was designed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman and till now is the most widely adopted algorithm to perform asymmetric encryption and decryption.
- ◆ This algorithm functions in three steps: key generation, encryption, and decryption.

© Aptech Ltd. Building Applications Using C# / Session 16 17

In slide 17, tell the students that the `System.Security.Cryptography` namespace provides the `AsymmetricAlgorithm` base class for all asymmetric algorithm classes.

Tell the students that the RSA is an abstract class that derives from the `AsymmetricAlgorithm` class and it is the base class for all the classes that implement the RSA algorithm.

Explain that the RSA algorithm was designed in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman and till now is the most widely adopted algorithm to perform asymmetric encryption and decryption.

Mention that this algorithm functions in three steps: key generation, encryption, and decryption.

Slide 18

Understand asymmetric encryption algorithm.



Asymmetric Encryption Algorithm 2-2

- ◆ The RSA algorithm generates a public key as a product of two large prime numbers, along with a public (or encryption) value.
- ◆ The algorithm generates a private key as a product of the same two large prime numbers, along with a private (or decryption) value.
- ◆ The public key is used to perform encryption while the private key is used to perform decryption.
- ◆ In the .NET Framework, the RSACryptoServiceProvider class derives from the RSA class to provide an implementation of the RSA algorithm.

© Aptech Ltd.

Building Applications Using C# / Session 16 18

Use slide 18 to explain the students that the RSA algorithm generates a public key as a product of two large prime numbers, along with a public (or encryption) value.

Tell them that the algorithm generates a private key as a product of the same two large prime numbers, along with a private (or decryption) value. The public key is used to perform encryption while the private key is used to perform decryption.

Mention that in the .NET Framework, the RSACryptoServiceProvider class derives from the RSA class to provide an implementation of the RSA algorithm.

With this slide, you will finish explaining encryption support in the .NET framework.

Slide 19

Understand how to perform symmetric encryption.

The slide has a teal header bar with the title "Performing Symmetric Encryption". Below the title are two bullet points:

- ◆ You need to use one of the symmetric encryption implementation classes of the .NET Framework to perform symmetric encryption.
- ◆ The first step to perform symmetric encryption is to create the symmetric key.

In the center of the slide is a yellow key icon with black outlines and shading.

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 16", and the number "19".

In slide 19, tell students that they need to use one of the symmetric encryption implementation classes of the .NET Framework to perform symmetric encryption.

Tell them that the first step to perform symmetric encryption is to create the symmetric key.

Slides 20 and 21

Understand the generating symmetric keys process.

Generating Symmetric Keys 1-4

- When you use the default constructor of the symmetric encryption classes, such as `RijndaelManaged` and `AesManaged`, a key and IV are automatically generated.
- The generated key and the IV can be accessed as byte arrays using the `Key` and `IV` properties of the encryption class.
- The following code creates a symmetric key and IV using the `RijndaelManaged` class.

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RijndaelManaged symAlgo = new RijndaelManaged();
Console.WriteLine("Generated key: {0}, \nGenerated IV: {1}",
Encoding.Default.GetString(symAlgo.Key), Encoding.Default.GetString(symAlgo.IV));
```

© Aptech Ltd. Building Applications Using C# / Session 16 20

Generating Symmetric Keys 2-4

- The code snippet uses the default constructor of the `RijndaelManaged` class to generate a symmetric key and IV.
- The `Key` and `IV` properties are accessed and printed as strings using the default encoding to the console.
- The following figure shows the output of the code.

Output

```
C:\Windows\system32\cmd.exe
Generated key: EYEcGd!!"7DÉ]+ö= z±ñUp 1æ-L&Z,
Generated IV: ö?rSM"åä!GAjK
Press any key to continue . . .
```

© Aptech Ltd. Building Applications Using C# / Session 16 21

In slide 20, tell the students that when the default constructor of the symmetric encryption classes, such as `RijndaelManaged` and `AesManaged`, are used, a key and IV are automatically generated.

Explain them that the generated key and the IV can be accessed as byte arrays using the `Key` and `IV` properties of the encryption class.

Tell that the code creates a symmetric key and IV using the `RijndaelManaged` class. Use slide 21 to explain the code. Tell that in the code, the default constructor of the `RijndaelManaged` class is used to generate a symmetric key and IV. The `Key` and `IV` properties are accessed and printed as strings using the default encoding to the console. You can refer to the figure in slide 21 that shows the output of the code.

Additional Information

For more information on generating symmetric keys, refer the following link:

<http://msdn.microsoft.com/en-us/library/sb7w85t6%28v=vs.85%29.aspx>

Slides 22 and 23

Understand the symmetric encryption classes.

The screenshot shows a presentation slide with a blue header bar containing icons of a smartphone, laptop, and gear. The main title is "Generating Symmetric Keys 3-4". Below the title, there is a bulleted list and a code snippet.

- ◆ The symmetric encryption classes, such as `RijndaelManaged` also provide the `GenerateKey()` and `GenerateIV()` methods that you can use to generate keys and IVs, as shown in the following code:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RijndaelManaged symAlgo = new RijndaelManaged();
symAlgo.GenerateKey();
symAlgo.GenerateIV();
byte[] generatedKey = symAlgo.Key;
byte[] generatedIV = symAlgo.IV;
Console.WriteLine("Generated key through GenerateKey(): {0}, \nGenerated IV through
GenerateIV(): {1}", Encoding.Default.GetString(generatedKey),
Encoding.Default.GetString(generatedIV));
```

Generating Symmetric Keys 4-4

- ◆ In the code:
 - ❖ An `RijndaelManaged` object is created and the `GenerateKey()` and `GenerateIV()` methods are called to generate a key and an IV.
 - ❖ The Key and the IV properties are then accessed and printed as strings using the default encoding to the console.
- ◆ The following figure shows the output of the code.

Output

C:\Windows\system32\cmd.exe
Generated key through GenerateKey(): =ägyçDSÄµ, iChüAf2%íÑäëìl_02â¤ch!!,
Generated IV through GenerateIV(): JaoEt;owXnciginI
Press any key to continue . . .

© Aptech Ltd.

Building Applications Using C# / Session 16 23

In slide 22, tell that the symmetric encryption classes, such as `RijndaelManaged` also provide the `GenerateKey()` and `GenerateIV()` methods that you can use to generate keys and IVs.

Use slide 23, to explain the code. Tell the students that in the code, an `RijndaelManaged` object is created and the `GenerateKey()` and `GenerateIV()` methods are called to generate a key and an IV.

Mention that the Key and the IV properties are then accessed and printed as strings using the default encoding to the console.

In-Class Question:

After you finish explaining the Symmetric Encryption and Asymmetric Encryption, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



Which are the derived classes of the `SymmetricAlgorithm` base class?

Answer:

The derived classes of the `SymmetricAlgorithm` base class are `RC2`, `DES`, `TripleDES`, `Aes`, and `Rijndael`.

In-Class Question:

After you finish explaining the decrypting data, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



What is Encryption?

Answer:

The Encryption is a cryptographic technique that ensures data confidentiality. It converts data in plain text to cipher (secretly coded) text.

Slides 24 and 25

Understand encrypting data.

The slide has a teal header bar with the title "Encrypting Data 1-5". The main content area contains a bulleted list of five points about symmetric encryption in .NET. At the bottom, there is a footer bar with the copyright information "© Aptech Ltd.", the slide title "Building Applications Using C# / Session 16", and the page number "24".

- ◆ The symmetric encryption classes of the .NET Framework provides the `CreateEncryptor()` method that returns an object of the `ICryptoTransform` interface.
- ◆ The `ICryptoTransform` object is responsible for transforming the data based on the algorithm of the encryption class.
- ◆ Once you have obtained an `ICryptoTransform` object, you can use the `CryptoStream` class to perform encryption.
- ◆ The `CryptoStream` class acts as a wrapper of a stream-derived class, such as `FileStream`, `MemoryStream`, and `NetworkStream`.



Encrypting Data 2-5

- ◆ The `CryptoStream` class acts as a wrapper of a stream-derived class, such as `FileStream`, `MemoryStream`, and `NetworkStream`.
- ◆ A `CryptoStream` object operates in one of the following two modes defined by the `CryptoStreamMode` enumeration:
 - ❖ **Write:**
 - This mode allows write operation on the underlying stream.
 - Use this mode to perform encryption.
 - ❖ **Read:**
 - This mode allows read operation on the underlying stream.
 - Use this mode to perform decryption.

© Aptech Ltd.

Building Applications Using C# / Session 16 25

In slide 24, explain the students the process of encrypting the data. Tell the students that the symmetric encryption classes of the .NET Framework provides the `CreateEncryptor()` method that returns an object of the `ICryptoTransform` interface.

Explain them that the `ICryptoTransform` object is responsible for transforming the data based on the algorithm of the encryption class. Once you have obtained an `ICryptoTransform` object, you can use the `CryptoStream` class to perform encryption.

Mention that the `CryptoStream` class acts as a wrapper of a stream-derived class, such as `FileStream`, `MemoryStream`, and `NetworkStream`.

Use slide 25 to explain that a `CryptoStream` object operates in the two modes defined by the `CryptoStreamMode` enumeration are `Write` and `Read`.

Tell them that the `Write` mode allows write operation on the underlying stream and this mode is used to perform encryption.

Mention that the `Read` mode allows read operation on the underlying stream. Mention that this mode is used to perform decryption.

Slides 26 to 28

Understand creating a `CryptoStream` object and its three parameters.

Encrypting Data 3-5

- ◆ You can create a `CryptoStream` object by calling the constructor that accepts the following three parameters:
 - ◆ The underlying stream object
 - ◆ The `ICryptoTransform` object
 - ◆ The mode defined by the `CryptoStreamMode` enumeration
- ◆ After creating the `CryptoStream` object you can call the `Write()` method to write the encrypted data to the underlying stream.
- ◆ The following code encrypts data using the `RijndaelManaged` class and writes the encrypted data to a file.

Snippet

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
    static void EncryptData(String plainText, RijndaelManaged algo)
    {
        byte[] plaindataArray = ASCIIEncoding.ASCII.GetBytes(plainText);
```

© Aptech Ltd. Building Applications Using C# / Session 16 26

Encrypting Data 4-5

```
ICryptoTransform transform=algo.CreateEncryptor();
using (var fileStream = new FileStream("D:\\CipherText.txt",
    FileMode.OpenOrCreate, FileAccess.Write))
{
    using (var cryptoStream = new CryptoStream(fileStream, transform,
        CryptoStreamMode.Write))
    {
        cryptoStream.Write(plaindataArray, 0,
            plaindataArray.GetLength(0));
        Console.WriteLine("Encrypted data written to:
D:\\CipherText.txt");
    }
}
static void Main()
{
    RijndaelManaged symAlgo = new RijndaelManaged();
    Console.WriteLine("Enter data to encrypt.");
    string dataToEncrypt = Console.ReadLine();
    EncryptData(dataToEncrypt, symAlgo);
}
```

© Aptech Ltd. Building Applications Using C# / Session 16 27

Encrypting Data 5-5

- ◆ In the code:
 - ❖ The `Main()` method creates a `RijndaelManaged` object and passes it along with the data to encrypt to the `EncryptData()` method.
 - ❖ The call to the `CreateEncryptor()` method creates the `ICryptoTransform` object in the `EncryptData()` method.
 - ❖ Then, a `FileStream` object is created to write the encrypted text to the `CipherText.txt` file.
 - ❖ The `CryptoStream` object is created and its `Write()` method is called.
- ◆ The following figure shows the output of the code.

Output

In slide 26, tell the students that a `CryptoStream` object can be created by calling the constructor that accepts the three parameters.

Mention the three parameters. They are, the underlying stream object, the `ICryptoTransform` object, and the mode defined by the `CryptoStreamMode` enumeration.

Tell them that after creating the `CryptoStream` object, call the `Write()` method to write the encrypted data to the underlying stream.

In slides 26 and 27, show the code that encrypts data using the `RijndaelManaged` class and writes the encrypted data to a file.

Use slide 28 to explain the code and the output.

Tell that in the code, the `Main()` method creates a `RijndaelManaged` object and passes it along with the data to encrypt to the `EncryptData()` method. In the `EncryptData()` method, the call to the `CreateEncryptor()` method creates the `ICryptoTransform` object. Then, a `FileStream` object is created to write the encrypted text to the `CipherText.txt` file. Next, the `CryptoStream` object is created and its `Write()` method is called.

You can refer to the figure in slide 28 that shows the output of the code. With slide 28, you will finish explaining performing symmetric encryption.

Slide 29

Understand decrypting data.

Decrypting Data 1-4

- ◆ To decrypt data, you need to:

- Step 1**
 - Use the same symmetric encryption class, key, and IV used for encrypting the data.
- Step 2**
 - Call the `CreateDecryptor()` method to obtain a `ICryptoTransform` object that will perform the transformation.
- Step 3**
 - Create the `CryptoStream` object in Read mode and initialize a `StreamReader` object with the `CryptoStream` object.
- Step 4**
 - Call the `ReadToEnd()` method of the `StreamReader` that returns the decrypted text as a string.

- ◆ The code creates a program that performs both encryption and decryption.

© Aptech Ltd. Building Applications Using C# / Session 16 29

In slide 29, tell the students that to decrypt data they need to use the same symmetric encryption class, key, and IV used for encrypting the data.

Then, tell them to call the `CreateDecryptor()` method to obtain a `ICryptoTransform` object that will perform the transformation.

After that, tell them that they need to create the `CryptoStream` object in Read mode and initialize a `StreamReader` object with the `CryptoStream` object.

Mention that the students need to call the `ReadToEnd()` method of the `StreamReader` that returns the decrypted text as a string at the end.

Explain to students that the code creates a program that performs both encryption and decryption.

Slides 30 to 32

Understand the code that creates a program that performs both encryption and decryption.

Decrypting Data 2-4

Snippet

```

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
class SymmetricEncryptionDemo
{
    static void EncryptData(String plainText, RijndaelManaged algo)
    {
        byte[] plaindataArray = ASCIIEncoding.ASCII.GetBytes(plainText);

        ICryptoTransform transform = algo.CreateEncryptor();
        using (var fileStream = new FileStream("D:\\CipherText.txt",
        FileMode.OpenOrCreate, FileAccess.Write))
        {
            using (var cryptoStream = new CryptoStream(fileStream, transform,
            CryptoStreamMode.Write))
            {
                cryptoStream.Write(plaindataArray, 0, plaindataArray.GetLength(0));
                Console.WriteLine("Encrypted data written to: D:\\CipherText.txt");
            }
        }
    }
    static void DecryptData(RijndaelManaged algo)
    {
    }

```

© Aptech Ltd. Building Applications Using C# / Session 16 30

Decrypting Data 3-4

```

ICryptoTransform transform = algo.CreateDecryptor();
        using (var fileStream = new FileStream("D:\\CipherText.txt", FileMode.Open,
        FileAccess.Read))
        {
            using (CryptoStream cryptoStream = new CryptoStream(fileStream,
            transform, CryptoStreamMode.Read))
            {

                using (var streamReader = new StreamReader(cryptoStream))
                {
                    string decryptedData = streamReader.ReadToEnd();
                    Console.WriteLine("Decrypted data: \n{0}", decryptedData);

                }
            }
        }
    static void Main()
    {
        RijndaelManaged symAlgo = new RijndaelManaged();
        Console.WriteLine("Enter data to encrypt.");
        string dataToEncrypt = Console.ReadLine();
        EncryptData(dataToEncrypt, symAlgo);
        DecryptData(symAlgo);
    }
}

```

© Aptech Ltd. Building Applications Using C# / Session 16 31

Decrypting Data 4-4

- ◆ In the code:
 - ◆ The Main () method creates a `RijndaelManaged` object and passes it along with the data to encrypt the `EncryptData()` method.
 - ◆ The encrypted data is saved to the `CipherText.txt` file.
 - ◆ The Main () method calls the `DecryptData()` method passing the same `RijndaelManaged` object created for encryption.
 - ◆ The `DecryptData()` method creates the `ICryptoTransform` object and uses a `FileStream` object to read the encrypted data from the file.
 - ◆ The `CryptoStream` object is created in the Read mode initialized with the `FileStream` and `ICryptoTransform` objects.
 - ◆ A `StreamReader` object is created by passing the `CryptoStream` object to the constructor.
 - ◆ The `ReadToEnd()` method of the `StreamReader` object is called.
- ◆ The decrypted text returned by the `ReadToEnd()` method is printed to the console, as shown in the following figure:

Output

©Aptech Ltd.

Building Applications Using C# / Session 16

32

In slide 30 and 31, tell that the code creates a program that performs both encryption and decryption to the students.

Use slide 32 to explain the code. Tell that in the code, the `Main()` method creates a `RijndaelManaged` object and passes it along with the data to encrypt the `EncryptData()` method. The encrypted data is saved to the `CipherText.txt` file.

Tell that the `Main()` method calls the `DecryptData()` method passing the same `RijndaelManaged` object created for encryption. The `DecryptData()` method creates the `ICryptoTransform` object and uses a `FileStream` object to read the encrypted data from the file.

Then, tell that the `CryptoStream` object is created in the Read mode initialized with the `FileStream` and `ICryptoTransform` objects.

Next, a `StreamReader` object is created by passing the `CryptoStream` object to the constructor. Finally, the `ReadToEnd()` method of the `StreamReader` object is called. The decrypted text returned by the `ReadToEnd()` method is printed to the console.

You can refer to the figure in slide 31.

Additional Information

For more information on decrypting data, refer the following link:

<http://msdn.microsoft.com/en-us/library/ms229740%28v=vs.85%29.aspx>

Slide 33

Understand performing asymmetric encryption.

Performing Asymmetric Encryption

- ◆ You can use the RSACryptoServiceProvider class of the System.Security.Cryptography namespace to perform asymmetric encryption.

© Aptech Ltd. Building Applications Using C# / Session 16 33

In slide 33, mention that the students can use the RSACryptoServiceProvider class of the System.Security.Cryptography namespace to perform asymmetric encryption.

Additional Information

For more information on RSACryptoServiceProvider class, refer the following link:

<http://msdn.microsoft.com/en-us/library/system.security.cryptography.rsacryptoserviceprovider.aspx>

Slide 34

Understand the process of generating asymmetric keys.

Generating Asymmetric Keys 1-2

- ◆ When you call the default constructor of the `RSACryptoServiceProvider` class, a new public/private key pair is automatically generated.
- ◆ After you create a new instance of the class, you can export the key information by using one of the following methods:
 - ◆ `ToXMLString()`
 - Returns an XML representation of the key information.
 - ◆ `ExportParameters()`
 - Returns an `RSAParameters` structure that holds the key information.
 - Accept a Boolean value by both the `ToXMLString()` and `ExportParameters()` methods.
 - A `false` value indicates that the method should return only the public key information while a `true` value indicates that the method should return information of both the public and private keys.

© Aptech Ltd. Building Applications Using C# / Session 16 34

Use slide 34 to tell the students that when the default constructor of the `RSACryptoServiceProvider` class is called, a new public/private key pair is automatically generated.

Tell them that after creating a new instance of the class, you can export the key information by using `ToXMLString()` method or `ExportParameters()` method.

Explain them that the `ToXMLString()` method returns an XML representation of the key information.

Tell them that the `ExportParameters()` method returns an `RSAParameters` structure that holds the key information.

Tell them that both the methods accept a Boolean value by both the `ToXMLString()` and `ExportParameters()` methods.

Mention that a `false` value indicates that the method should return only the public key information while a `true` value indicates that the method should return information of both the public and private keys.

Slide 35

Understand the codes that show how to create and initialize an RSACryptoServiceProvider object.

The slide has a blue header bar with the title "Generating Asymmetric Keys 2-2". Below the header, there are two bullet points:

- ◆ The following code shows how to create and initialize an RSACryptoServiceProvider object and then export the public key in XML format:
Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
}
```
- ◆ The following code shows how to create and initialize an RSACryptoServiceProvider object and then export both the public and private keys as an RSAParameters structure:
Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
```

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 16", and "35".

Use slide 35 to explain the code that shows how to create and initialize an RSACryptoServiceProvider object and then export the public key in XML format.

Then explain the code that shows how to create and initialize an RSACryptoServiceProvider object and then export both the public and private keys as an RSAParameters structure.

Slide 36

Understand the key containers.



Key Containers 1-3

- ◆ Private keys used to decrypt data in asymmetric encryption should be stored using a secured mechanism.
- ◆ To achieve this, you can use a key container that is a logical structure to securely store asymmetric keys.
- ◆ The .NET Framework provides the `CspParameters` class to create a key container and to add and remove keys to and from the container.
- ◆ To create a key container for an `RSACryptoServiceProvider` object:
 - ◆ Use the default constructor of the `CspParameters` class to create a key container instance.
 - ◆ Set the container name using the `KeyContainerName` property of the `CspParameters` class.
 - ◆ Pass the `CspParameters` object to the constructor while creating the `RSACryptoServiceProvider` object to store the key pair in the key container.

Building Applications Using C# / Session 16 36

Use slide 36 to explain that private keys used to decrypt data in asymmetric encryption should be stored using a secured mechanism.

Tell that to achieve this, use a key container that is a logical structure to securely store asymmetric keys.

Also, tell that the .NET Framework provides the `CspParameters` class to create a key container and to add and remove keys to and from the container.

Mention that to create a key container for an `RSACryptoServiceProvider` object, use the default constructor of the `CspParameters` class to create a key container instance.

Then, set the container name using the `KeyContainerName` property of the `CspParameters` class.

Also, mention that to store the key pair in the key container, pass the `CspParameters` object to the constructor while creating the `RSACryptoServiceProvider` object.

Slide 37

Understand the code that uses a key container to store a key pair and its output.

Key Containers 2-3

- The following code uses a key container to store a key pair:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
...
CspParameters cspParams = new CspParameters(); cspParams.KeyContainerName =
"RSA_CONTAINER"; RSACryptoServiceProvider rSAKeyGenerator =
new RSACryptoServiceProvider(cspParams); Console.WriteLine("RSA key added to the
container,\n\n{0}", rSAKeyGenerator.ToXmlString(true));
```

- The following figure shows the output of the code:

Output

```
Microsoft Windows [Version 10.0.19041]
Copyright (c) 2020 Microsoft Corporation. All rights reserved.

C:\>using System;
using System.Security.Cryptography;
using System.Text;
CspParameters cspParams = new CspParameters(); cspParams.KeyContainerName =
"RSA_CONTAINER"; RSACryptoServiceProvider rSAKeyGenerator =
new RSACryptoServiceProvider(cspParams); Console.WriteLine("RSA key added to the
container,\n\n{0}", rSAKeyGenerator.ToXmlString(true));
Press any key to continue . . .
```

© Aptech Ltd.

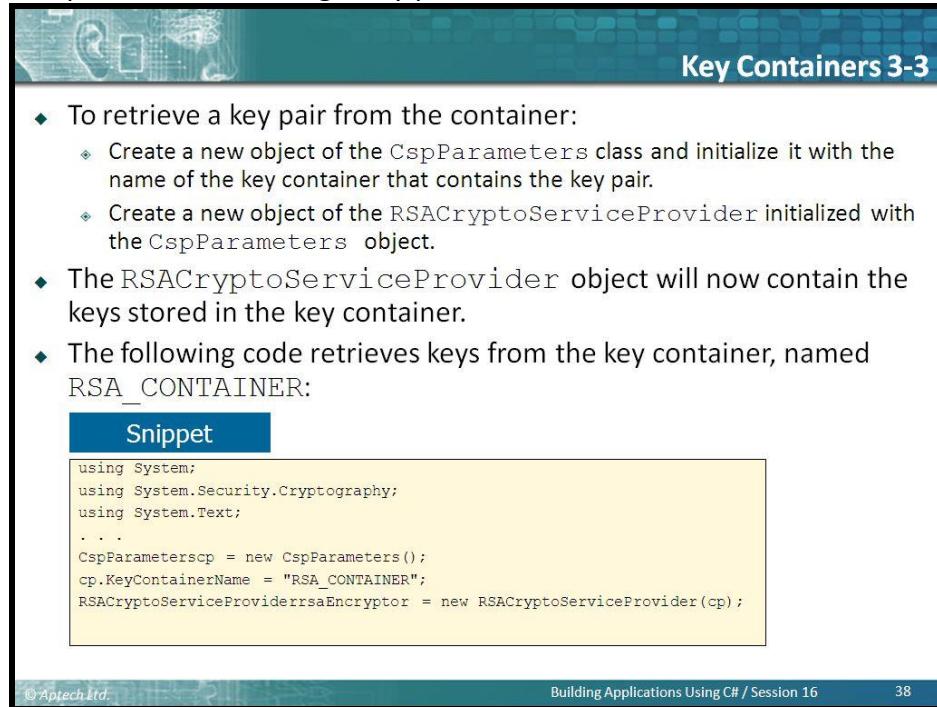
Building Applications Using C# / Session 16

37

Use slide 37 to explain that code that uses a key container to store a key pair and its output to the students.

Slide 38

Understand the process of retrieving a key pair.



The slide has a blue header bar with the title "Key Containers 3-3". Below the header, there is a list of steps to retrieve a key pair from a container, followed by a code snippet in a yellow box, and finally a footer bar with copyright information.

Key Containers 3-3

- ◆ To retrieve a key pair from the container:
 - ◆ Create a new object of the `CspParameters` class and initialize it with the name of the key container that contains the key pair.
 - ◆ Create a new object of the `RSACryptoServiceProvider` initialized with the `CspParameters` object.
- ◆ The `RSACryptoServiceProvider` object will now contain the keys stored in the key container.
- ◆ The following code retrieves keys from the key container, named `RSA_CONTAINER`:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
.

CspParameterscp = new CspParameters();
cp.KeyContainerName = "RSA_CONTAINER";
RSACryptoServiceProviderrsaEncryptor = new RSACryptoServiceProvider(cp);
```

© Aptech Ltd. Building Applications Using C# / Session 16 38

Use slide 38 to tell the students that to retrieve a key pair from the container, they need to create a new object of the `CspParameters` class and initialize it with the name of the key container that contains the key pair. Then, they need to create a new object of the `RSACryptoServiceProvider` initialized with the `CspParameters` object.

Mention that the `RSACryptoServiceProvider` object will now contain the keys stored in the key container. The code retrieves keys from the key container, named `RSA_CONTAINER` to the students.

Slides 39 to 41

Understand encrypting data using C#.

Encrypting Data 1-3



- ◆ To encrypt data, you need to:
 - ❖ Create a new instance of the `RSACryptoServiceProvider` class
 - ❖ Call the `ImportParameters()` method to initialize the instance with the public key information exported to an `RSAParameters` structure.
- ◆ The following code shows how to initialize an `RSACryptoServiceProvider` object with the public key exported to an `RSAParameters` structure:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new
RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(false);
RSACryptoServiceProvider rsaEncryptor= new
RSACryptoServiceProvider();
rsaEncryptor.ImportParameters(rSAKeyInfo);
```

© Aptech Ltd. Building Applications Using C# / Session 16 39

Encrypting Data 2-3

- ◆ If the public key information is exported to XML format, you need to call the `FromXmlString()` method to initialize the `RSACryptoServiceProvider` object with the public key, as shown in the following code:

Snippet

```
using System;
using System.Security.Cryptography;
using System.Text;
...
RSACryptoServiceProvider rSAKeyGenerator = new
RSACryptoServiceProvider();
string publicKey = rSAKeyGenerator.ToXmlString(false);
RSACryptoServiceProvider rsaEncryptor= new RSACryptoServiceProvider();
rsaEncryptor.FromXmlString(publicKey);
```

© Aptech Ltd. Building Applications Using C# / Session 16 40



Encrypting Data 3-3

- ◆ After the RSACryptoServiceProvider object is initialized with the public key, you can encrypt data by calling the Encrypt() method of the RSACryptoServiceProvider class.
- ◆ The Encrypt() method accepts the following two parameters:
 - ◆ byte array of the data to encrypt
 - ◆ A Boolean value (It indicates whether or not to perform encryption using Optimal Asymmetric Encryption Padding (OAEP) padding. A true value uses OAEP padding while a false value uses PKCS#1 v1.5 padding.)
- ◆ The Encrypt() method after performing encryption returns a byte array of the encrypted text, as shown in the code:

Snippet

```
byte[] plainbytes = new UnicodeEncoding().GetBytes("Plain text to encrypt.");
byte[] cipherbytes = rsaEncryptor.Encrypt(plainbytes, true);
```

Use slide 39 to explain that to encrypt data, you need to create a new instance of the RSACryptoServiceProvider class and call the ImportParameters() method to initialize the instance with the public key information exported to an RSAParameters structure.

Tell that the code in slide 40 shows the process to initialize an RSACryptoServiceProvider object with the public key exported to an RSAParameters structure.

Tell the students that if the public key information is exported to XML format, call the FromXmlString() method to initialize the RSACryptoServiceProvider object with the public key. After the RSACryptoServiceProvider object is initialized with the public key, they can encrypt data by calling the Encrypt() method of the RSACryptoServiceProvider class.

Using slide 41, explain that the Encrypt() method accepts the two parameters: A byte array of the data to encrypt and a Boolean value that indicates whether or not to perform encryption using Optimal Asymmetric Encryption Padding (OAEP) padding. A true value uses OAEP padding while a false value uses PKCS#1 v1.5 padding.

In encryption, padding is used with an encryption algorithm to make the encryption stronger. Padding prevents predictability to find patterns that might aid in breaking the encryption. For example, OAEP is a padding scheme often used together with RSA encryption.

Slide 42

Understand decrypting data.



Decrypting Data 1-8

- ◆ To decrypt data, you need to initialize an RSACryptoServiceProvider object using the private key of the key pair whose public key was used for encryption.
- ◆ The following code shows how to initialize an RSACryptoServiceProvider object with the private key exported to an RSAParameters structure:

Snippet

```
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
RSAParameters rSAKeyInfo = rSAKeyGenerator.ExportParameters(true);
RSACryptoServiceProvider rsaDecryptor= new RSACryptoServiceProvider();
rsaDecryptor.ImportParameters(rSAKeyInfo);
```

© Aptech Ltd. Building Applications Using C# / Session 16 42

Use slide 42 to tell the students that to decrypt data, you need to initialize an RSACryptoServiceProvider object using the private key of the key pair whose public key was used for encryption.

Then, tell that the code on the slide displays how to initialize an RSACryptoServiceProvider object with the private key exported to an RSAParameters structure.

Slides 43 and 44

Understand the code that displays how to initialize an RSACryptoServiceProvider object with the private key exported to XML format.

The slide has a blue header bar with the title "Decrypting Data 2-8". The main content area contains a bulleted list and a code snippet box. The list item is: "The following code shows how to initialize an RSACryptoServiceProvider object with the private key exported to XML format:". Below this is a "Snippet" box containing the following C# code:

```
RSACryptoServiceProvider rSAKeyGenerator = new RSACryptoServiceProvider();
string keyPair = rSAKeyGenerator.ToXmlString(true);
RSACryptoServiceProvider rsaDecryptor = new RSACryptoServiceProvider();
rsaDecryptor.FromXmlString(keyPair);
```

At the bottom of the slide, there is a footer bar with the text "© Aptech Ltd.", "Building Applications Using C# / Session 16", and "43".



Decrypting Data 3-8

- ◆ When the RSACryptoServiceProvider object is initialized with the private key, you can decrypt data by calling the Decrypt() method of the RSACryptoServiceProvider class.
- ◆ The Decrypt() method accepts the following two parameters:
 - ◆ byte array of the encrypted data
 - ◆ A Boolean value (It indicates whether or not to perform encryption using OAEP padding. A true value uses OAEP padding while a false value uses PKCS#1 v1.5 padding.)

© Aptech Ltd.

Building Applications Using C# / Session 16

44

In slide 43, tell that the code displays how to initialize an RSACryptoServiceProvider object with the private key exported to XML format.

Use slide 44 to explain that when the RSACryptoServiceProvider object is initialized with the private key, they can decrypt data by calling the Decrypt() method of the RSACryptoServiceProvider class.

Explain that the Decrypt() method accepts the two parameters: byte array of the encrypted data and a Boolean value. Mention that it indicates whether or not to perform encryption using OAEP padding. A true value uses OAEP padding while a false value uses PKCS#1 v1.5 padding.

Slides 45 to 49

Understand the Decrypt() method that returns a byte array of the original data.

Decrypting Data 4-8

- The Decrypt() method returns a byte array of the original data, as shown in the following code:

Snippet

```
byte[] plainbytes = rsaDecryptor.Decrypt(cipherbytes, false);
```

- The following code shows a program that performs asymmetric encryption and decryption:

Snippet

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
classAsymmetricEncryptionDemo
{
```

Decrypting Data 5-8

```
static byte[] EncryptData(string plainText, RSAParameters
rsaParameters)
{
byte[] plainTextArray = new
UnicodeEncoding().GetBytes(plainText);
RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
RSA.ImportParameters(rsaParameters);
byte[] encryptedData = RSA.Encrypt(plainTextArray, true);
return encryptedData;

}
static byte[] DecryptData(byte[] encryptedData, RSAParameters
rsaParameters)
{
RSACryptoServiceProvider RSA = new
RSACryptoServiceProvider();
RSA.ImportParameters(rsaParameters);
byte[] decryptedData = RSA.Decrypt(encryptedData, true);
return decryptedData;
```

Decrypting Data 6-8

```

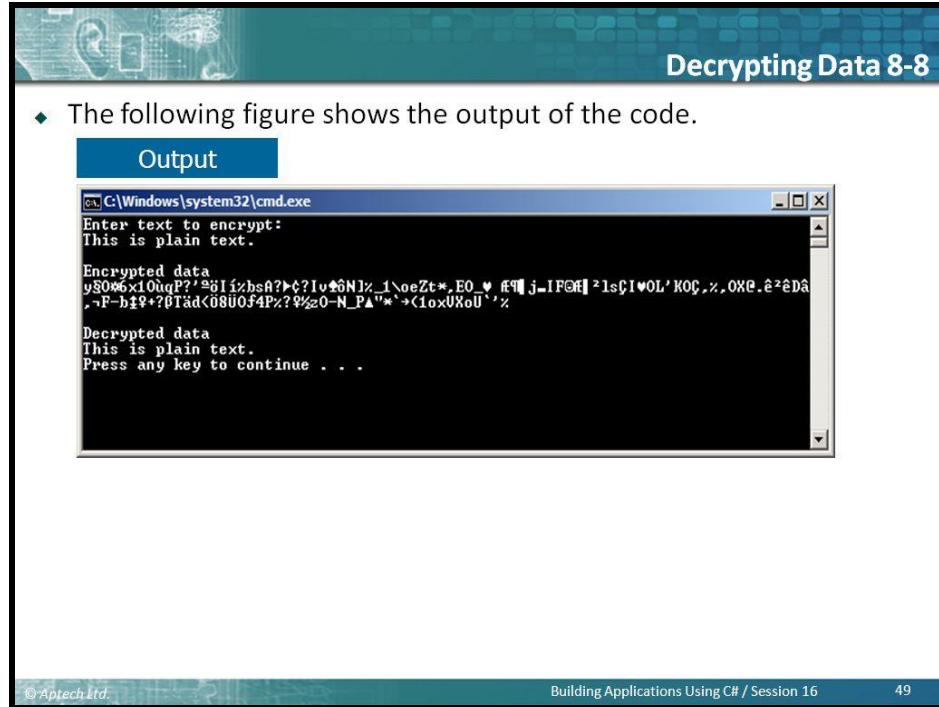
    }
    static void Main(string[] args)
    {
        Console.WriteLine("Enter text to encrypt:");
        String inputText = Console.ReadLine();
        RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
        RSAParametersRSAParam=RSA.ExportParameters(false);
        byte[] encryptedData = EncryptData(inputText, RSAParam);
        stringencryptedString =
        Encoding.Default.GetString(encryptedData);
        Console.WriteLine("\nEncrypted data \n{0}", encryptedString);
        byte[] decryptedData = DecryptData(encryptedData,
                                         RSA.ExportParameters(true));
        StringdecryptedString = new
        UnicodeEncoding().GetString(decryptedData);
        Console.WriteLine("\nDecrypted data \n{0}", decryptedString);
    }
}

```

Decrypting Data 7-8

◆ In the code:

- ◆ The `Main()` method creates a `RSACryptoServiceProvider` object and exports the public key as a `RSAParameters` structure.
- ◆ The `EncryptData()` method is then called passing the user entered plain text and the `RSAParameters` object.
- ◆ The `EncryptData()` method uses the exported public key to encrypt the data and returns the encrypted data as a byte array.
- ◆ The `Main()` method then exports both the public and private key of the `RSACryptoServiceProvider` object into a second `RSAParameters` object.
- ◆ The `DecryptData()` method is called passing the encrypted byte array and the `RSAParameters` object.
- ◆ The `DecryptData()` method performs the decryption and returns the original plain text as a string.



In slides 45, 46, and 47, tell that the code displays the `Decrypt()` method returns a byte array of the original data.

Use slides 48 and 49 to explain the code and the output.

Tell that in the code, the `Main()` method creates a `RSACryptoServiceProvider` object and exports the public key as a `RSAParameters` structure. The `EncryptData()` method is then called passing the user entered plain text and the `RSAParameters` object.

Explain that the `EncryptData()` method uses the exported public key to encrypt the data and returns the encrypted data as a byte array. The `Main()` method then exports both the public and private key of the `RSACryptoServiceProvider` object into a second `RSAParameters` object.

The `DecryptData()` method is called passing the encrypted byte array and the `RSAParameters` object.

Mention that the `DecryptData()` method performs the decryption and returns the original plain text as a string.

In-Class Question:

After you finish explaining the decrypting data, you will ask the students an In-Class question. This will help you in reviewing their understanding of the topic.



How to decrypt data?

Answer:

To decrypt data, you need to initialize an `RSACryptoServiceProvider` object using the private key of the key pair whose public key was used for encryption.

Slide 50

Summarize the session.

Summary

- ◆ Encryption is a security mechanism that converts data in plain text to cipher text.
- ◆ An encryption key is a piece of information or parameter that determines how a particular encryption mechanism works.
- ◆ The .NET Framework provides various types in the `System.Security.Cryptography` namespace to support symmetric and asymmetric encryptions.
- ◆ When you use the default constructor to create an object of the symmetric encryption classes, a key and IV are automatically generated.
- ◆ The `ICryptoTransform` object is responsible for transforming the data based on the algorithm of the encryption class.
- ◆ The `CryptoStream` class acts as a wrapper of a stream-derived class, such as `FileStream`, `MemoryStream`, and `NetworkStream`.
- ◆ When you call the default constructor of the `RSACryptoServiceProvider` and `DSACryptoServiceProvider` classes, a new public/private key pair is automatically generated.

© Aptech Ltd. Building Applications Using C# / Session 16 50

In slide 50, you will summarize the session. You will end the session, with a brief summary of what has been taught in the session. Tell the students pointers of the session. This will be a revision of the current session and it will be related to the next session.

Explain each of the following points in brief. Tell them that:

- Encryption is a security mechanism that converts data in plain text to cipher text.
- An encryption key is a piece of information or parameter that determines how a particular encryption mechanism works.
- The .NET Framework provides various types in the `System.Security.Cryptography` namespace to support symmetric and asymmetric encryptions.
- When you use the default constructor to create an object of the symmetric encryption classes, a key and IV are automatically generated.
- The `ICryptoTransform` object is responsible for transforming the data based on the algorithm of the encryption class.
- The `CryptoStream` class acts as a wrapper of a stream-derived class, such as `FileStream`, `MemoryStream`, and `NetworkStream`.
- When you call the default constructor of the `RSACryptoServiceProvider` and `DSACryptoServiceProvider` classes, a new public/private key pair is automatically generated.

16.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips:

You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information, such as:

1. How to store the value of textbox into database in encrypted format?
2. How can you use C# to encrypt and decrypt strings using a salt key to protect the data?
3. How to encrypt data before sending via QueryString?