

LEARN



IN DEPTH

PROGRAMMING

WITH C#



109, WING A, SHOPPER'S ORBIT,
VISHRANTWADI, PUNE - 411015



7757012051/52

ENOSIS LEARNING

<http://www.enosislearning.com>

Contents

Chapter 1: Introduction to .NET Framework	6
Framework Class Library (FCL)	10
.NET Applications (Assemblies, Metadata, Applications)	11
.NET Applications	12
Chapter 2: C# - Program Structure	14
Creating Hello World Program	14
Compiling and Executing the Program	16
String Type	21
C# Type Conversion Methods	23
Defining Variables	25
Accepting Values from User	26
Defining Constants	28
Chapter 3: C# - Decision Making and loops	30
The ? : Operator	31
Loop Control Statements	33
Chapter 4: C# - Classes and Object	33
Member Functions and Member variables	34
Example of method	36
Access modifiers	37
All types and type members have an accessibility level. The accessibility level controls whether they can be used from other code in your assembly or other assemblies.	37
Defining Methods in C#	38
Example	38
Calling Methods in C#	39
Passing Parameters to a Method	42
C# Constructors	42
Object in C#	45
Chapter 5: C# - Encapsulation	47
Chapter 6: C# - Inheritance	51
Multiple Inheritance in C#	52
Chapter 7: C# - Polymorphism	53
Static Polymorphism	54
Function Overloading	54
Dynamic Polymorphism	55
Chapter 8: C# - Operator Overloading	57
Implementing the Operator Overloading	58
Chapter 9 C# - Nullables	59

The Null Coalescing Operator (??)	61
Chapter 10: C# - Arrays	61
Assigning Values to an Array.....	62
Using the <i>foreach</i> Loop	64
C# Arrays	65
Chapter 11: C# - Strings	66
Creating a String Object	66
Chapter 12: C# - Structures	67
Features of C# Structures.....	69
Class versus Structure	69
Chapter 13: C# - Enums	71
Declaring <i>enum</i> Variable	71
Example.....	71
Chapter 14: C# - Interfaces	72
Declaring Interfaces	72
Example.....	73
Chapter 15 C# - Namespaces	74
Defining a Namespace	75
The <i>using</i> Keyword.....	76
Nested Namespaces.....	77
Chapter 16: C# - Preprocessor Directives	78
Preprocessor Directives in C#	79
Chapter 17: C# - Regular Expressions	79
Constructs for Defining Regular Expressions	80
The Regex Class.....	80
Example 1.....	80
Example 2.....	81
Chapter 18: C# - Exception Handling	82
Syntax.....	83
Exception Classes in C#	83
Handling Exceptions.....	85
Creating User-Defined Exceptions	86
Throwing Objects	86
Chapter 19: C# - File I/O	87
C# I/O Classes.....	87
The FileStream Class	88
Example.....	89
Advanced File Operations in C#	90

Chapter 20: C# - Attributes	91
Specifying an Attribute	91
Predefined Attributes	92
AttributeUsage	92
Conditional	93
Obsolete	94
Creating Custom Attributes	95
Declaring a Custom Attribute	95
Constructing the Custom Attribute	96
Applying the Custom Attribute	97
Chapter 21: C# - Reflection	98
Applications of Reflection	98
Viewing Metadata	99
Anonymous Functions	101
Benefits of Anonymous Functions	101
C# Lambda Expressions	102
C# Anonymous Method	103
Chapter 22: C# - Properties	104
Accessors	104
Example	105
Abstract Properties	106
Chapter 23: C# - Indexers	108
Syntax	108
Use of Indexers	109
Overloaded Indexers	110
Chapter 24: C# - Delegates	112
Declaring Delegates	113
Instantiating Delegates	114
Using Delegates	116
Chapter 25: C# - Events	117
Add/Remove Operation in Events	118
Can we use Events without Delegate?	119
Chapter 26: C# -Collections	122
Why a Collection?	122
Various Collection Classes and Their Usage	124
Chapter 27: C# -Generics	129
Features of Generics	131
Generic Methods	131

Generic Delegates	134
Anonymous Methods in C#	135
Writing an Anonymous Method	135
Example	135
Chapter 28: C#- Unsafe Codes	137
Pointers	138
Retrieving the Data Value Using a Pointer	139
Passing Pointers as Parameters to Methods	139
Accessing Array Elements Using a Pointer	140
Compiling Unsafe Code	141
Chapter 29: C#- Multithreading	141
Thread Life Cycle	142
The Main Thread	143
Properties and Methods of the Thread Class	143
Join	146
Managing Threads	148
Chapter 30: C#- DLL	151
What is a DLL?	151
Advantages of Using DLLs in C#:	151
Creating a DLL in C#:	152
Using a DLL in C#:	153
Chapter 31: C#- Dependency Injection	153
Implementation of Dependency Injection Pattern in C#	154
Tight Coupling and Loose Coupling	155
Tight coupling:	155
Loose coupling:	155
Types of Dependency Injection (DI)	156
Constructor Injection:	156
Property Injection:	157
Method Injection:	157
Constructor Injection in C#	158
Advantages of Dependency Injection	160
Chapter 32: C#- Networking	160
System.Net classes	160
How to send email from C#	161
What is SMTP?	161
SMTP Servers	161
SMTP Authentication	161

How do I send mail using C#?	161
Send Email using Gmail	162
Chapter 34: C#- Design Patterns	163
What are Design Patterns in Software Development?	163
Chapter 35: C# 11-New Features	164
Chapter 36 Introduction to ADO.NET	165
ADO.NET Objects	166
INTERACTING WITH MICROSOFT ACCESS	176
ADO.NET TRANSACTIONS	176
Bulk Insert Update in C# using Stored Procedure	178

Chapter 1: Introduction to .NET Framework

.NET Framework is an environment for developing and executing software applications of different types.

It is a Unified programming model, set of languages, class libraries, infrastructure, components and tools for application development

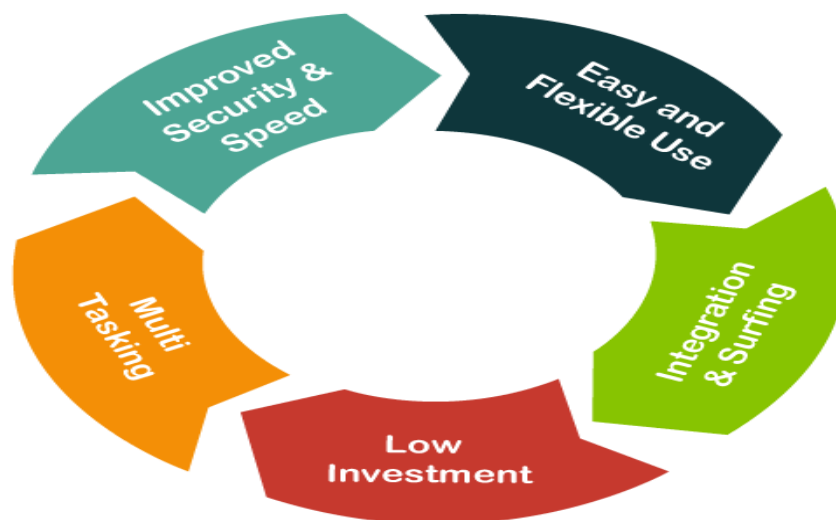


Environment for controlled execution of managed code

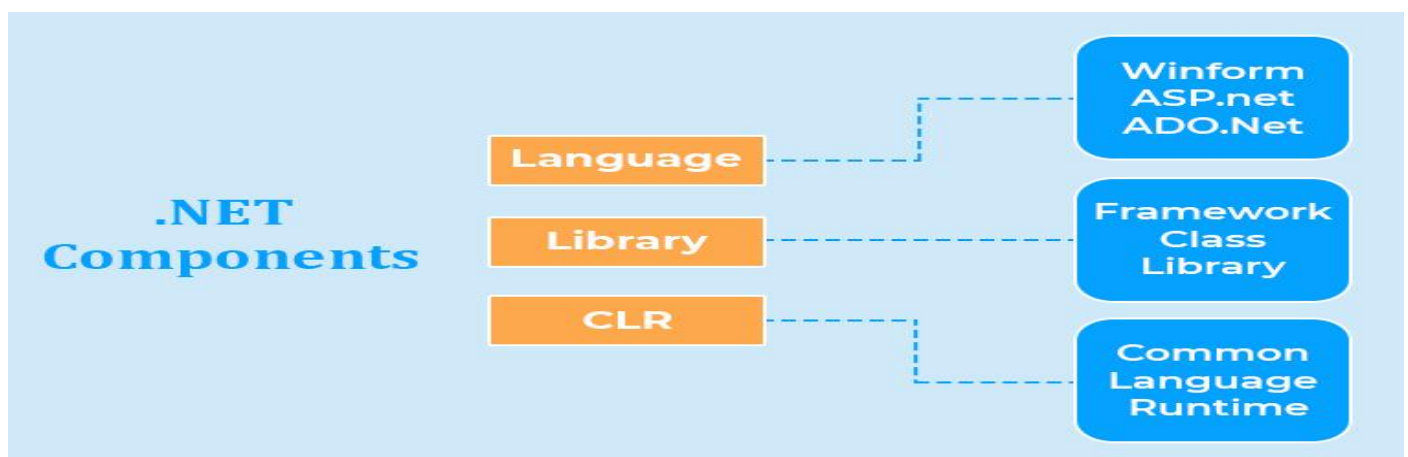
- ◆ It is commonly assumed that

.NET platform == .NET Framework

Characteristics of .NET Framework



.NET Framework Components



The .NET framework has two major components-- The Common Runtime (CLR) and the Class Library

Common Language Runtime (CLR)

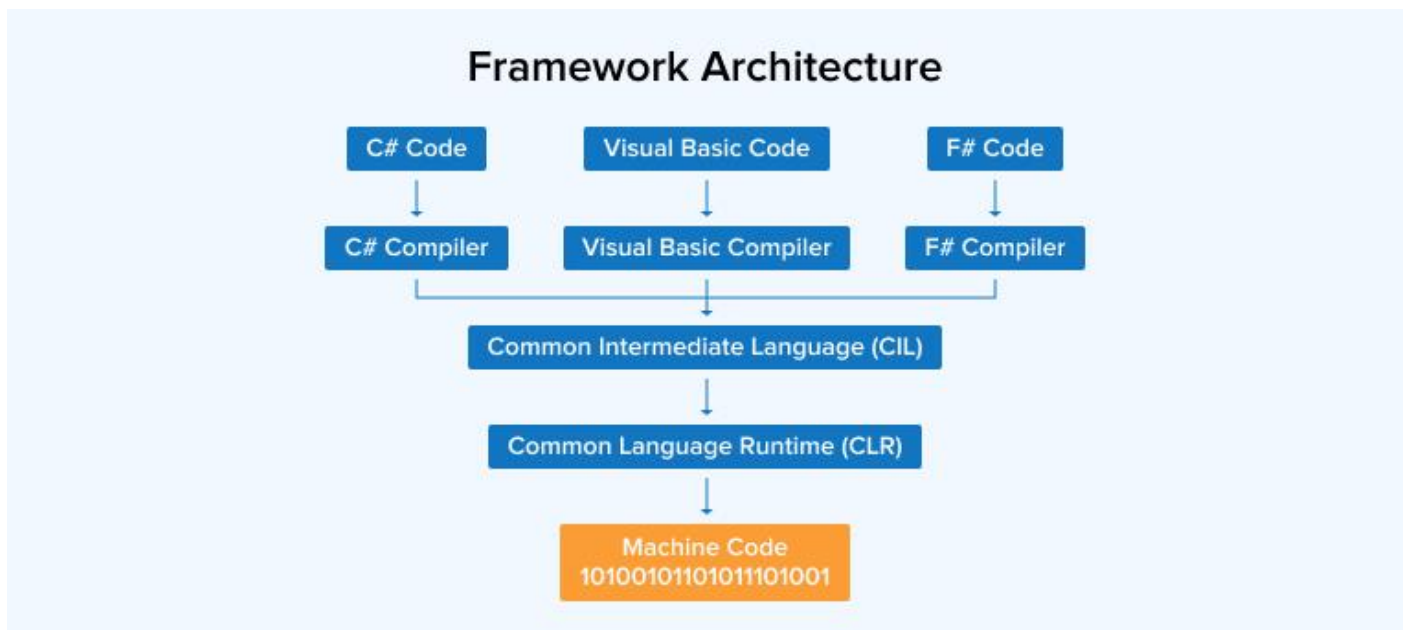
The CLR is the foundation upon which the .NET Framework has been built. The runtime manages code at execution time and provides all the core services such as memory management, thread management and remoting. This capability to manage code at runtime is the distinguishing feature of the CLR. All code that is managed by the CLR is known as managed code while other codes are known as unmanaged code.

Framework Class Library (FCL)

Standard class library for .NET development

Delivers basic functionality for developing: XML, ADO.NET, LINQ, ASP.NET, WPF, WCF, WWF, Silverlight, Web services, Windows Forms etc.

.NET Framework Architecture



- The OS manages the resources, the processes and the users of the machine
- Provides to the applications some services (threads, I/O, GDI+, DirectX, COM, COM+, MSMQ, IIS, WMI,
- CLR is a separate process in the OS
- CLR manages the execution of the .NET code
- Manages the memory, concurrency, security,
- Rich object-oriented library with fundamental classes
- Input-output, collections, text processing, networking, security, multi-threading, ...
- Database access
- ADO.NET, LINQ, LINQ-to-SQL and Entity Framework
- Strong XML support
- Windows Communication Foundation (WCF) and Windows Workflow Foundation (WWF) for the SOA world
- User interface technologies: Web based, Windows GUI, WPF, and Silverlight, mobile.
- Programming language on your flavor!

Common Language Runtime (CLR)

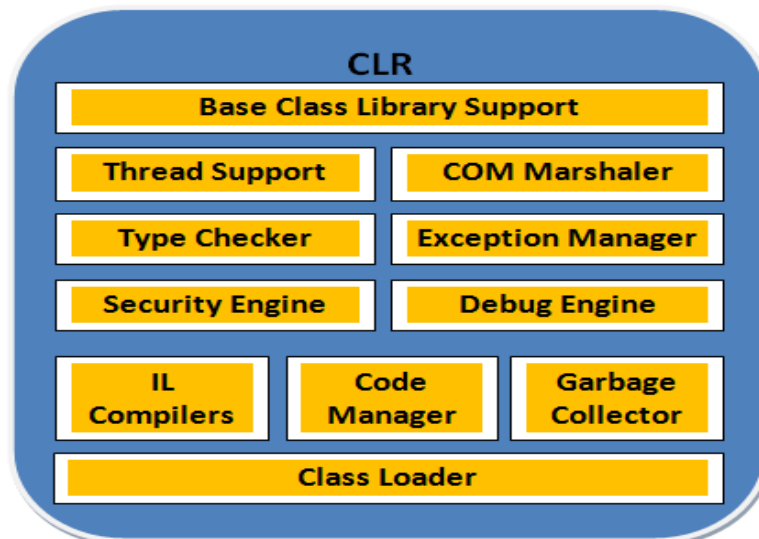
(The Heart of .NET Framework)

- ◆ **Managed execution environment**
Controls the execution of managed .NET programming code
- ◆ **Something like virtual machine**
Like the Java Virtual Machine (JVM)
- ◆ **Not an interpreter**
Compilation on-demand is used
Known as Just In Time (JIT) compilation
- ◆ **Possible compilation in advance (Ngen)**

Responsibilities of CLR

- ◆ Execution of the IL code and the JIT compilation
- ◆ Managing memory and application resources
- ◆ Ensuring type safety
- ◆ Interaction with the OS
- ◆ Managing security
Code access security
Role-based security
- ◆ Managing exceptions
- ◆ Managing concurrency – controlling the parallel execution of application threads
- ◆ Managing application domains and their isolation
- ◆ Interaction with unmanaged code
- ◆ Supporting debug / profile of .NET code

CLR Architecture



Managed Code

- ✓ CLR executed code is called managed code
- ✓ Represents programming code in the low level language MSIL (MS Intermediate Language)
- ✓ Contains metadata

- Description of classes, interfaces, properties, fields, methods, parameters, etc.
- ✓ Programs, written in any .NET language are
 - Compiled to managed code (MSIL)
 - Packaged as assemblies (.exe or .dll files)
- ✓ Object-oriented
- ✓ Secure
- ✓ Reliable
 - Protected from irregular use of types (type-safe)
- ✓ Allows integration between components and data types of different programming languages
- ✓ Portable between different platforms
 - Windows, Linux, Mac OS X, etc.

Unmanaged (Win32) Code

- ✓ No protection of memory and type-safety
 - Reliability problems
 - Safety problems
- ✓ Doesn't contain metadata
 - Needs additional overhead like (e.g. use COM)
- ✓ Compiled to machine-dependent code
 - Need of different versions for different platforms
 - Hard to be ported to other platforms

Memory Management

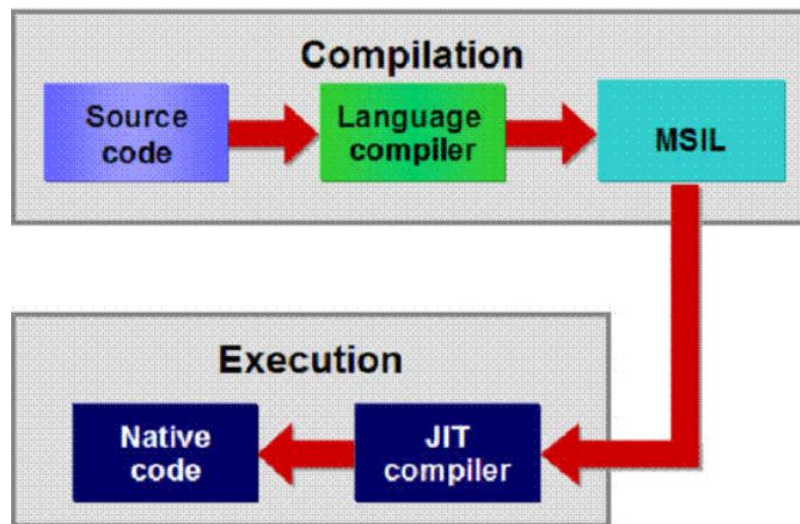
- ◆ CLR manages memory automatically
 - Dynamically loaded objects are stored in the managed heap
 - Unusable objects are automatically cleaned up by the garbage collector
- ◆ Some of the big problems are solved
 - Memory leaks
 - Access to freed or unallocated memory
- ◆ Objects are accessed through a reference

Intermediate Language (MSIL, IL, CIL)

- ◆ Low level language (machine language) for the .NET CLR
- ◆ Has independent set of CPU instructions
 - Loading and storing data, calling methods
 - Arithmetic and logical operations
 - Exception handling
- ◆ MSIL is converted to instructions for the current physical CPU by the JIT compiler

Sample MSIL Code :

Compilation and Execution



Common Type System (CTS)

A number of types are supported by the CLR and are described by the CTS. Both value types are supported—primitive data types and reference types. The primitive data types include Byte, Int16, Double and Boolean while Reference types include arrays, classes and object and string types. Reference types are types that store a reference to the location of their values. The value is stored as part of a defined class and is referenced through a class member on the instance of a class.

Language compilers implement types using their own terminology.

- ◆ CTS defines the CLR supported types of data and the operations over them
- ◆ Ensures data level compatibility between different .NET languages
E.g. string in C# is the same like String in VB.NET and in J#
- ◆ Value types and reference types

All types derive from System.Object

The Runtime enforces code robustness by implementing strict type and code verification infrastructure called Common type System (CTS). The CTS ensures that all managed code is self describing and all Microsoft or third party language compiler generated codes conform to CTS.

Common Language Infrastructure (CLI)

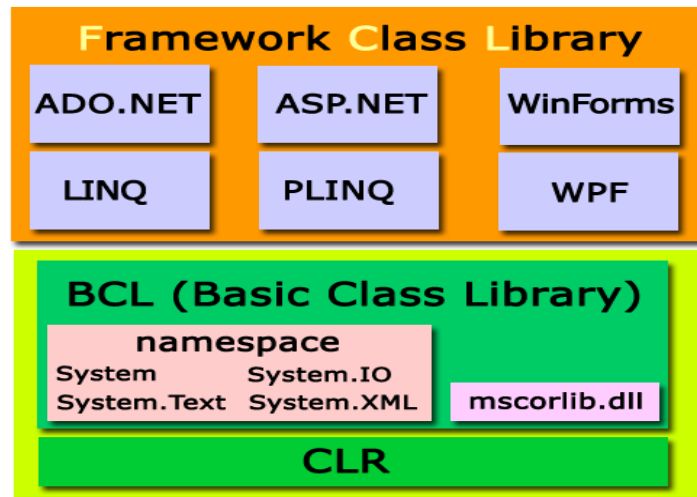
A subset of the .NET framework is the CLI. The CLI includes the functionality of the Common Language Runtime and specifications for the Common Type System, metadata and Intermediate language.

Common Language Specification (CLS)

The CLR supports the CLS which is a subset of it. Additionally the CLR supports a set of rules that language and compiler designers follow.

Framework Class Library (FCL)

- ◆ Framework Class Library is the standard .NET Framework library of out-of-the-box reusable classes and components (APIs)



The Class Library is an object oriented collection of reusable types. Unmanaged components that load CLR into their processes can be hosted by the .NET Framework to initiate the execution of managed code. This creates a software environment that exploits both the managed and unmanaged codes. The .NET Framework also provides a number of runtime hosts and supports third party runtime hosts

Class Library Features:

1. The class library is a collection of reusable types that integrate with the CLR.
2. It is object oriented and provides types from which user defined types can derive functionality. This makes for ease of use and is time saving.
3. It supports a variety of specialized development scenarios such as console application development, Windows GUI applications, ASP.NET Applications, XML Web services.

.NET Applications (Assemblies, Metadata, Applications)

.NET Assemblies

Assembly is a single deployable unit that contains all the information about the classes, structure, and interface. Assemblies store all the information about itself. This information can be called metadata. It is physical grouping of logical units, Namespace can span multiple assemblies.

Assemblies can be private & public

◆ .NET assemblies:

- Self-containing .NET components
- Stored in .DLL and .EXE files
- Contain list of classes, types and resources
- Smallest deployment unit in CLR
- Have unique version number

◆ .NET deployment model

- No version conflicts (forget the "DLL hell")
- Supports side-by-side execution of different versions of the same assembly

Metadata in the Assemblies

Data about data contained in the assembly
Integral part of the assembly
Generated by the .NET languages compiler

Describes all classes, their class members, versions, resources, etc.

Namespace

It is logically group classes. It avoid any naming conflicts between classes which have same name. It allows u to organize your classes so that they can be easily accessed in other application.

Global assembly cache (GAC)

Is where all assemblies resides

If the app. has to be shared among several applications this is in the same comp

.NET Applications

- ◆ Configurable executable .NET units
- ◆ Consist of one or more assemblies
- ◆ Installed by "copy / paste"
No complex registration of components
- ◆ Different applications use different versions of common assemblies
No conflicts due to their "strong name"

Easy installation, un-installation and update

.NET Languages

- ◆ .NET languages by Microsoft
C#, VB.NET, Managed C++, J#, F#, JScript
- ◆ .NET languages by third parties
Object Pascal, Perl, Python, COBOL, Haskell, Oberon, Scheme, Smalltalk...
- ◆ Different languages can be mixed in a single application

Cross-language inheritance of types and exception handling

.NET Main Features

One framework,
multiple languages



Common Language
Runtime (CLR)



Interoperability



Base Class Library



Security



Automatic Resource
Management



Portability













Profiling and
debugging support



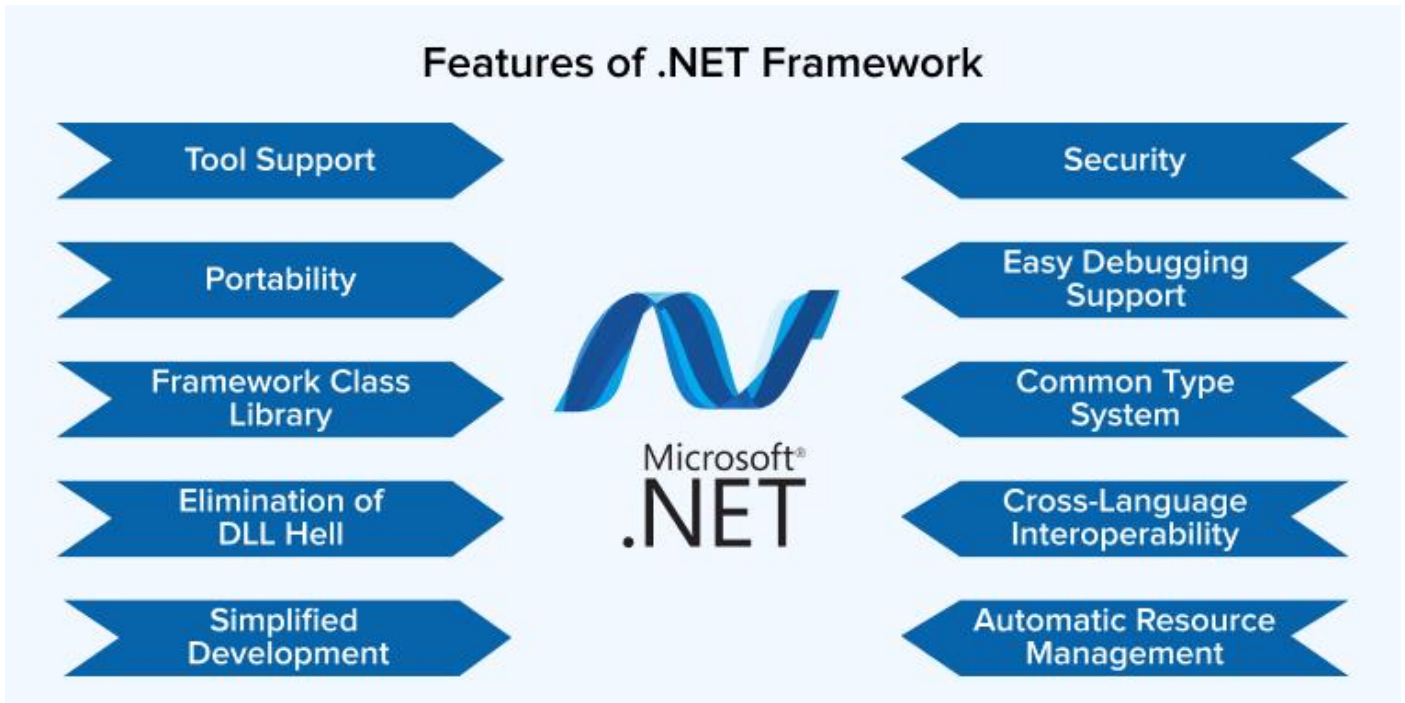
Easy Deployment



Advantages and Limitations of .NET Framework

ADVANTAGES	LIMITATIONS
 Cross-platform design	 Supplier dependence
 Visual Studio IDE	 Issues with object-relational support
 Automated code checking	 License cost
 Advanced UI control	 Stability issues for new releases
 Reliable caching system	 RAM requirement

Features of .NET Framework



C# is a strongly typed object-oriented programming language. C# is open source, simple, modern, flexible, and versatile. In this article, let's learn what C# is, what C# can do, and how C# is different than C++ and other programming languages.

A programming language on computer science is a language that is used to write software programs.

C# is a programming language developed and launched by Microsoft in 2001. C# is a simple, modern, and object-oriented language that provides modern day developers flexibility and features to build software that will not only work today but will be applicable for years in the future.

Key characteristics of C# language include:

1. Modern and easy
2. Fast and open source
3. Cross platform
4. Safe
5. Versatile
6. Evolving

The below table explains the different versions of .Net Framework along with release years.

Version	Year	Features
1.0	1999-2002	Modern, Object Oriented, Simple, Flexible, Typesafe, Managed, Garbage Collection, Cross-platform
2.0	2005	Generics, Anonymous Method, Partial Class, Nullable Type
3.0	2008	LINQ, Lamda Expression, Extension Method, Anonymous Type, Var
4.0	2010	Named and Optional Parameters, Dynamic Binding
5.0	2012	Async Programming
6.0	2015	Compiler-as-a-service (Roslyn), Exception filters, Await in catch/finally blocks, Auto property initializers, Dictionary initializer, Default values for getter-only properties, Expression-bodied members. Null propagator, String interpolation, nameof operator
7.0	2017	Tuples, Out variables, Pattern matching, Deconstruction, Local functions, Digit separators, Binary literals, Ref returns and locals, Generalized async return types, Expression bodied constructors and finalizers, Expression bodied getters and setters, Throw can also be used as expression
7.1	2017	Async main, Default literal expressions, Inferred tuple element names
7.2	2017	Reference semantics with value types, Non-trailing named arguments, Leading underscores in numeric literals, private protected access modifier
7.3	2018	Accessing fixed fields without pinning, Reassigning ref local variables, Using initializers on stackalloc arrays, Using fixed statements with any type that supports a pattern, Using additional generic constraints
8.0	2019	Nullable reference types, Async streams, ranges and indices, default implementation of interface members, recursive patterns, switch expressions, target-type new expressions

Chapter 2: C# - Program Structure

Before we study basic building blocks of the C# programming language, let us look at a bare minimum C# program structure so that we can take it as a reference in upcoming chapters.

Creating Hello World Program

A C# program consists of the following parts –

- Namespace declaration
- A class
- Class methods
- Class attributes
- A Main method
- Statements and Expressions
- Comments

Let us look at a simple code that prints the words "Hello World" –

```
using System;

namespace HelloEnosisApplication {

    class HelloWorld {

        static void Main(string[] args) {
            /* my first program in C# */
            Console.WriteLine("Hello Enosis Learning");
            Console.ReadKey();
        }
    }
}
```

Output: Hello Enosis Learning

Let us look at the various parts of the given program –

- The first line of the program **using System;** - the **using** keyword is used to include the **System** namespace in the program. A program generally has multiple **using** statements.
- The next line has the **namespace** declaration. A **namespace** is a collection of classes. The *HelloWorldApplication* namespace contains the class *HelloWorld*.
- The next line has a **class** declaration, the class *HelloWorld* contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the *HelloWorld* class has only one method **Main**.
- The next line defines the **Main** method, which is the **entry point** for all C# programs. The **Main** method states what the class does when executed.
- The next line */*...*/* is ignored by the compiler and it is put to add **comments** in the program.
- The Main method specifies its behaviour with the statement **Console.WriteLine("Hello World");**
Write Line is a method of the *Console* class defined in the *System* namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

- The last line **Console.ReadKey();** is for the VS.NET Users. This makes the program wait for a key press and it prevents the screen from running and closing quickly when the program is launched from Visual Studio .NET.

It is worth to note the following points –

- C# is case sensitive.
- All statements and expression must end with a semicolon (;).
- The program execution starts at the Main method.
- Unlike Java, program file name could be different from the class name.

Compiling and Executing the Program

If you are using Visual Studio.Net for compiling and executing C# programs, take the following steps –

- Start Visual Studio.
- On the menu bar, choose File -> New -> Project.
- Choose Visual C# from templates, and then choose Windows.
- Choose Console Application.
- Specify a name for your project and click OK button.
- This creates a new project in Solution Explorer.
- Write code in the Code Editor.
- Click the Run button or press F5 key to execute the project. A Command Prompt window appears that contains the line Hello World.



C# - Basic Syntax

C# is an object-oriented programming language. In Object-Oriented Programming methodology, a program consists of various objects that interact with each other by means of actions. The actions that an object may take are called methods. Objects of the same kind are said to have the same type or, are said to be in the same class.

Let us look at implementation of a Bank class and discuss C# basic syntax –

```

using System;

namespace BankApplication {

    class Bank {
        // member variables
        double amtcredit;
        double amtdebit;

        public void AcceptAccountdetails() {
            amtcredit = 4500;
            amtdebit = 3500;
        }

        public double GetDetails() {
            return amtcredit * amtdebit;
        }

        public void DisplayAccountDeatils() {
            Console.WriteLine("amtcredit: {0}", amtcredit);
            Console.WriteLine("amtdebit: {0}", amtdebit);
            Console.WriteLine("BankDetails: {0}", GetDetails ());
        }
    }

    class ExecuteBankDetails {

        static void Main(string[] args) {
            Bank r = new Bank ();
            r. AcceptAccountdetails ();
            r. DisplayAccountDeatils ();
            Console.ReadLine();
        }
    }
}

```

C# Keywords

Keywords are reserved words predefined to the C# compiler. These keywords cannot be used as identifiers. However, if you want to use these keywords as identifiers, you may prefix the keyword with the @ character.

In C#, some identifiers have special meaning in context of code, such as get and set are called contextual keywords.

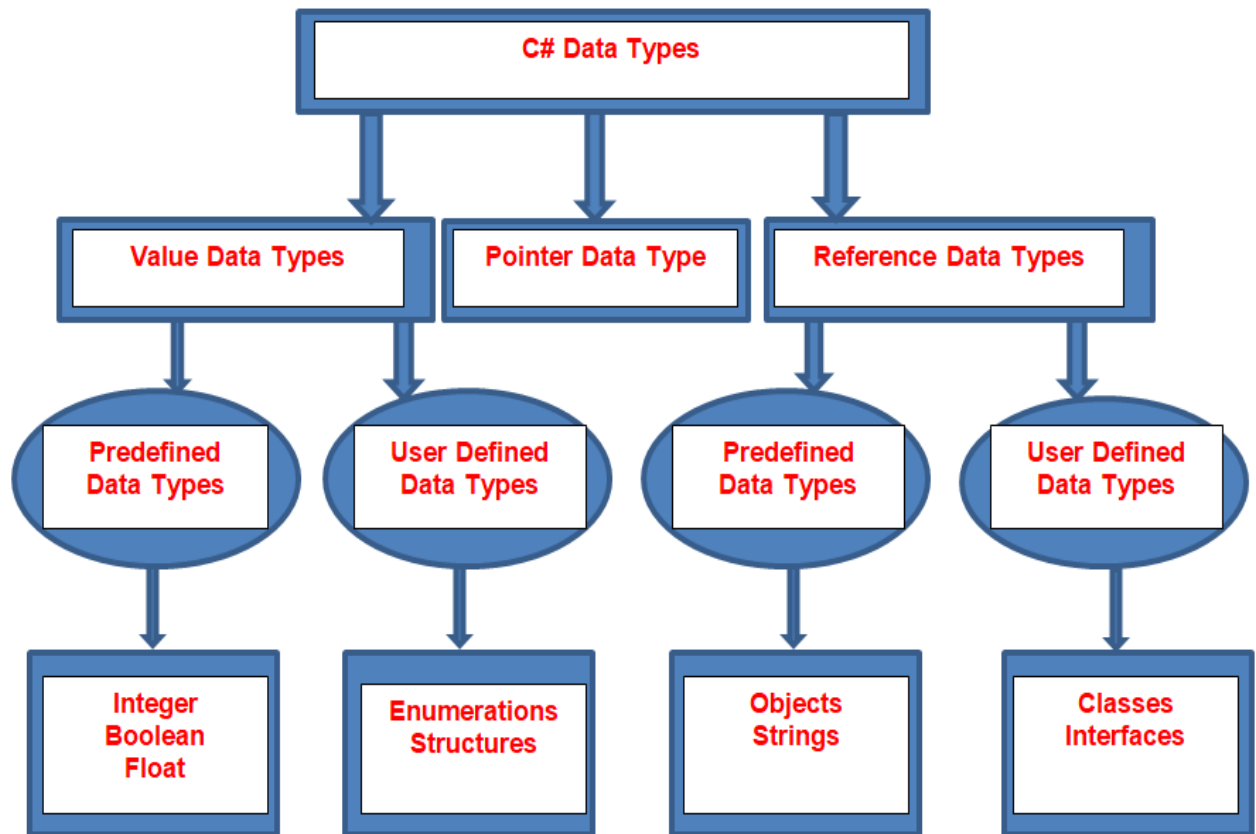
The following table lists the reserved keywords and contextual keywords in C# –

Reserved Keywords						
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	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	Out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile	while					
Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	Join	let	orderby	partial (type)
partial (method)	remove	select	Set			

C# - Data Types

The variables in C#, are categorized into the following types –

- Value types
- Reference types
-



Value Type

Value type variables can be assigned a value directly. They are derived from the class **System.ValueType**.

The value types directly contain data. Some examples are **int**, **char**, and **float**, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an **int** type, the system allocates memory to store the value.

The following table lists the available value types in C# 2022 –

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28} \text{ to } 7.9 \times 10^{28}) / 10^0 \text{ to } 28$	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324} \text{ to } (+/-)1.7 \times 10^{308}$	0.0D

float	32-bit single-precision floating point type	-3.4 x 10 ³⁸ to + 3.4 x 10 ³⁸	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** method. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Following is an example to get the size of *int* type on any machine –

```
using System;
namespace DataTypeApplication {

    class Program {
        static void Main(string[] args)
        {
            Console.WriteLine("Size of int: {0}", sizeof(int));
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

Size of int: 4

Reference Type

The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.

In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the

variables, the other variable automatically reflects this change in value. Example of **built-in** reference types are: **object**, **dynamic**, and **string**.

Object Type

The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
object obj;  
obj = 100; // this is boxing
```

Dynamic Type

You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.

Syntax for declaring a dynamic type is –

```
dynamic <variable_name> = value;  
For example: dynamic d = 20;
```

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example : String str = "Enosis Learning";

A @quoted string literal looks as follows – @"Enosis Learning";

The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

Pointer Type

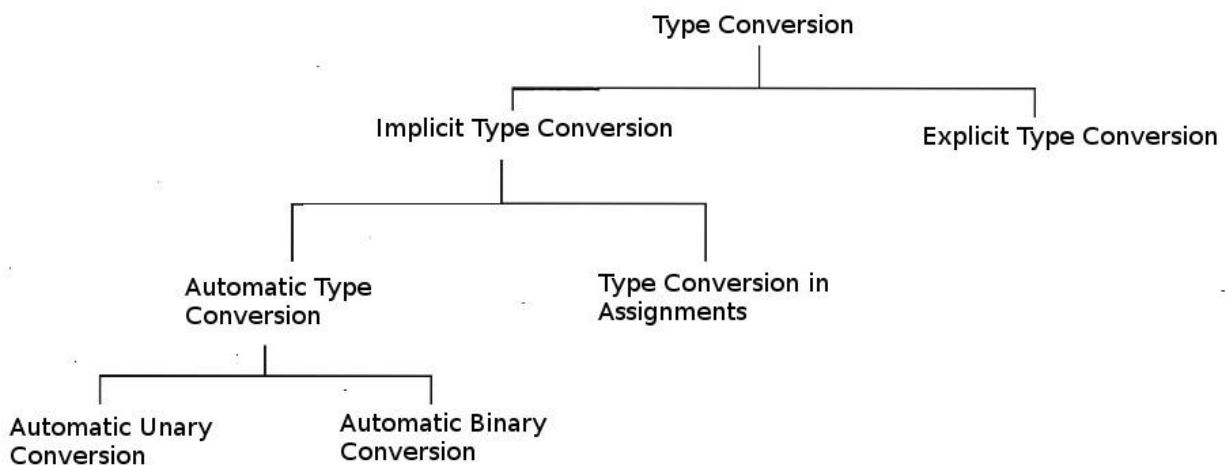
Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.

Syntax for declaring a pointer type is –

```
type* identifier;  
For example,  
char* cptr;  
int* iptr;
```

C# - Type Conversion

Type conversion is converting one type of data to another type. It is also known as Type Casting. In C#, type casting has two forms –



- **Implicit type conversion** – These conversions are performed by C# in a type-safe manner. For example, are conversions from smaller to larger integral types and conversions from derived classes to base classes?
- **Explicit type conversion** – These conversions are done explicitly by users using the pre-defined functions. Explicit conversions require a cast operator.

The following example shows an explicit type conversion –

```
namespace TypeConversionApplication {  
  
    class ExplicitConversion {  
  
        static void Main(string[] args) {
```

```

double d = 5673.74;
int i;
// cast double to int.
i = (int)d;
Console.WriteLine(i);
Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result – 5673

C# Type Conversion Methods

C# provides the following built-in type conversion methods –

Sr.No.	Methods & Description
1	Convert.ToBoolean : Converts a type to a Boolean value, where possible.
2	Convert.ToByte : Converts a type to a byte.
3	Convert.ToChar : Converts a type to a single Unicode character, where possible.
4	Convert.ToDateTime : Converts a type (integer or string type) to date-time structures.
5	Convert.ToDecimal : Converts a floating point or integer type to a decimal type.
6	Convert.ToDouble : Converts a type to a double type.
7	Convert.ToInt16 : Converts a type to a 16-bit integer.
8	Convert.ToInt32 : Converts a type to a 32-bit integer.
9	Convert.ToInt64 : Converts a type to a 64-bit integer.
10	Convert.ToSbyte : Converts a type to a signed byte type.
11	Convert.ToSingle : Converts a type to a small floating point number.
12	Convert.ToString : Converts a type to a string.
13	Convert.ToType : Converts a type to a specified type.
14	Convert.ToUInt16 : Converts a type to an unsigned int type.
15	Convert.ToUInt32 : Converts a type to an unsigned long type.
16	Convert.ToUInt64 : Converts a type to an unsigned big integer.

The following example converts various value types to string type –


```

static void Main(string[] args) {
    int i = 75;
    float f = 53.005f;
    double d = 2345.7652;
    bool b = true;

    Console.WriteLine(i.ToString());
    Console.WriteLine(f.ToString());
    Console.WriteLine(d.ToString());
    Console.WriteLine(b.ToString());
    Console.ReadKey();
}

```

When the above code is compiled and executed, it produces the following result –

```

75
53.005
2345.7652
True

```

BOXING / UNBOXING IN C#

	Boxing	UnBoxing
1	Implicit Conversion.	Explicit Conversion.
2	Convert To Value Type to Reference Type .	Convert To Reference Type to Value Type.
3	Boxing is used to Store Value Types in the Garbage-Collected Heap.	UnBoxing an Interface Type to a Value Type that Implements the Interface.
4	Ex; <pre>int i = 100; object obj = i;</pre>	Ex; <pre>int i = 100; object obj = i; int j = (int)obj;</pre>

C# - Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The basic value types provided in C# can be categorized as –

Type	Example
Integral types	sbyte, byte, short, ushort, int, uint, long, ulong, and char
Floating point types	float and double
Decimal types	Decimal
Boolean types	true or false values, as assigned
Nullable types	Nullable data types

C# also allows defining other value types of variable such as **enum** and reference types of variables such as **class**, which we will cover in subsequent chapters.

Defining Variables

Syntax for variable definition in C# is –

```
<data_type> <variable_list>;
```

Here, data_type must be a valid C# data type including char, int, float, double, or any user-defined data type, and variable_list may consist of one or more identifier names separated by commas.

Some valid variable definitions are shown here –

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

You can initialize a variable at the time of definition as –

```
int i = 100;
```

The following example uses various types of variables –

```
static void Main(string[] args) {
    short a;
    int b ;
    double c;

    /* actual initialization */
    a = 10;
    b = 20;
```

```
c = a + b;  
Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);  
Console.ReadLine();  
}
```

When the above code is compiled and executed, it produces the following result –

a = 10, b = 20, c = 30

Accepting Values from User

The **Console** class in the **System** namespace provides a function **ReadLine()** for accepting input from the user and store it into a variable.

For example,

```
int num;  
num = Convert.ToInt32(Console.ReadLine());
```

The function **Convert.ToInt32 ()** converts the data entered by the user to int data type, because **Console.ReadLine ()** accepts the data in string format.

C# - Constants and Literals

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals : An integer literal can be a decimal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, and there is no prefix id for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals –

```
212      /* Legal */  
215u     /* Legal */  
0xFeeL   /* Legal */
```

Following are other examples of various types of Integer literals –

```
85    /* decimal */
0x4b  /* hexadecimal */
30    /* int */
30u   /* unsigned int */
30l   /* long */
30ul  /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

Here are some examples of floating-point literals –

```
3.14159    /* Legal */
314159E-5F  /* Legal */
510E       /* Illegal: incomplete exponent */
210f       /* Illegal: no decimal or exponent */
.e55       /* Illegal: missing integer or fraction */
```

While representing in decimal form, you must include the decimal point, the exponent, or both; and while representing using exponential form you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

```
static void Main(string[] args) {
    Console.WriteLine("Hello\tWorld\n\n");
    Console.ReadLine();
}
```

Output : Hello World

String Literals

String literals or constants are enclosed in double quotes "" or with @"". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating the parts using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
@"hello dear"
```

Defining Constants

Constants are defined using the **const** keyword. Syntax for defining a constant is –

```
const <data_type> <constant_name> = value;
```

The following program demonstrates defining and using a constant in your program –

```
static void Main(string[] args) {  
    const double pi = 3.14159;  
  
    // constant declaration  
    double r;  
    Console.WriteLine("Enter Radius: ");  
    r = Convert.ToDouble(Console.ReadLine());  
    double areaCircle = pi * r * r;  
    Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);  
    Console.ReadLine();  
}
```

When the above code is compiled and executed, it produces the following result –

Enter Radius:

3

Radius: 3, Area: 28.27431

C# - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C# has rich set of built-in operators and provides the following type of operators

Category	Operators
arithmetic	-, +, *, /, %, ++, --
logical	&&, , !, ^
binary	&, , ^, ~, <<, >>
comparison	==, !=, >, <, >=, <=
assignment	=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=
string concatenation	+
type conversion	(type), as, is, typeof, sizeof
other	., new, (), [], ?:, ??

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc. Operators

Category	Operators
arithmetic	<code>-, +, *, /, %, ++, --</code>
logical	<code>&&, , !, ^</code>
binary	<code>&, , ^, ~, <<, >></code>
comparison	<code>==, !=, >, <, >=, <=</code>
assignment	<code>=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=</code>
string concatenation	<code>+</code>
type conversion	<code>(type), as, is, typeof, sizeof</code>
other	<code>., new, (), [], ?:, ??</code>

Arithmetic

Arithmetic Operator	Name	Example
<code>+</code>	Addition	<code>a + b</code>
<code>-</code>	Subtraction	<code>a - b</code>
<code>*</code>	Multiplication	<code>a * b</code>
<code>/</code>	Division	<code>a / b</code>
<code>%</code>	Modulus	<code>a % b</code>
<code>**</code>	Exponentiation	<code>a ** b</code>
<code>++</code>	Increment Operator	<code>a++</code>
<code>--</code>	Decrement Operator	<code>a--</code>

Operator	Description
== equal to operator.	Checks the equality of the two operand values
!= not equal to operator	Checks equality of two operand values.
< less than operator	Checks whether the value of left operand is lesser than the value of right operand.
> great than operator	Checks whether the value of left operand is greater than the value of right operand.
<= less than or equal to operator	Checks whether the value of left operand is lesser than or equal to the value of right operand
>= greater than or equal to operator	Checks whether the value of left operand is greater than or equal to the value of right operand

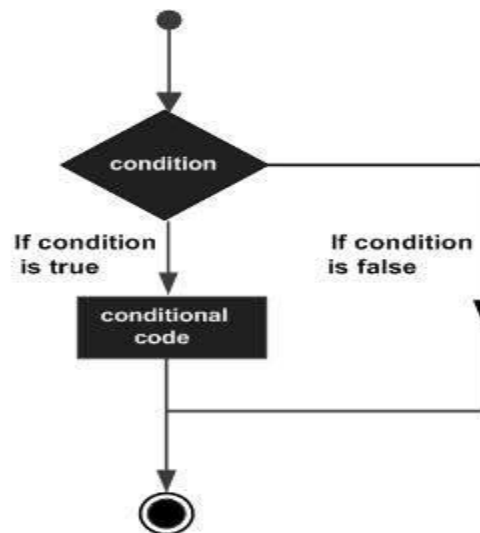
Assignment Operator	Name	Example
=	Assignment Operator	c = a + b
+=	Add AND assignment	a += b is same as a = a + b
-=	Subtract AND assignment	a -= b is same as a = a - b
*=	Multiply AND assignment	a *= b is same as a = a * b
/=	Divide AND assignment	a /= b is same as a = a / b

Operator	Meaning
 	Logical OR
&&	Logical AND
!	Logical NOT
 	Bitwise logical OR
&	Bitwise logical AND
^	Bitwise logical exclusive OR

Chapter 3: C# - Decision Making and loops

Decision making structures requires the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages –



C# provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	if statement An if statement consists of a boolean expression followed by one or more statements.
2	if...else statement An if statement can be followed by an optional else statement , which executes when the boolean expression is false.
3	nested if statements You can use one if or else if statement inside another if or else if statement(s).
4	switch statement A switch statement allows a variable to be tested for equality against a list of values.
5	nested switch statements You can use one switch statement inside another switch statement(s).

The ? : Operator

We have covered **conditional operator ? :** in previous chapter which can be used to replace **if...else** statements. It has the following general form –

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

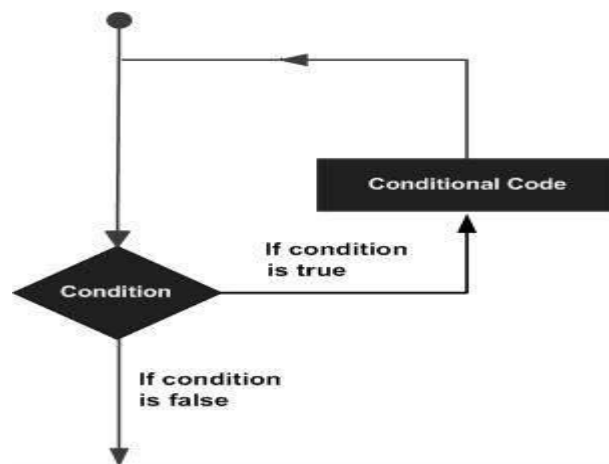
The value of a? expression is determined as follows: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

C# - Loops

There may be a situation, when you need to execute a block of code several number of times. In general, the statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or a group of statements multiple times and following is the general from of a loop statement in most of the programming languages –



C# provides following types of loop to handle looping requirements. Click the following links to check their detail.

Sr.No.	Loop Type & Description
1	while loop It repeats a statement or a group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop It executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop It is similar to a while statement, except that it tests the condition at the end of the loop body
4	nested loops

	You can use one or more loop inside any another while, for or do..while loop.
--	---

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C# provides the following control statements. Click the following links to check their details.

Sr.No.	Control Statement & Description
1	break statement Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but programmers more commonly use the for(;;) construct to signify an infinite loop.

Chapter 4: C# - Classes and Object

When you define a class, you define a blueprint for a data type. This does not actually define any data, but it does define what the class name means. That is, what an object of the class consists of and what operations can be performed on that object. Objects are instances of a class. The methods and variables that constitute a class are called members of the class.

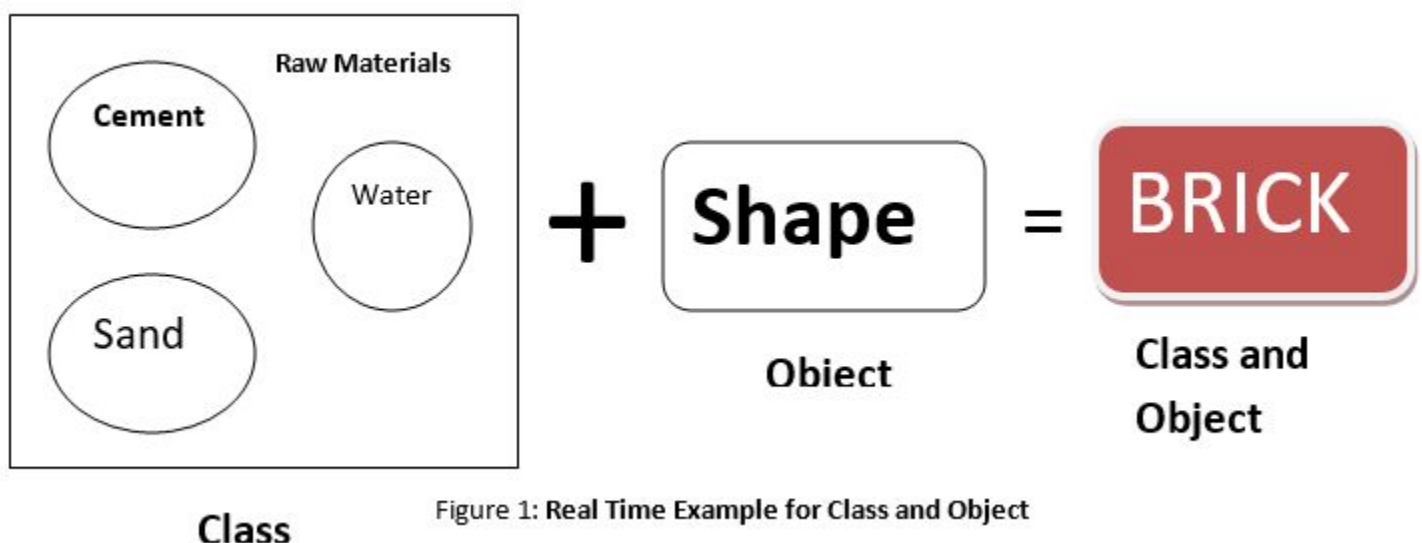


Figure 1: Real Time Example for Class and Object

Defining a Class

A class definition starts with the keyword `class` followed by the class name; and the class body enclosed by a pair of curly braces. Following is the general form of a class definition –

```
<access specifier> class class_name {  
    // member variables  
    <access specifier> <data type> variable1;  
    <access specifier> <data type> variable2;  
    ...  
    <access specifier> <data type> variableN;  
    // member methods  
    <access specifier> <return type> method1(parameter_list) {  
        // method body  
    }  
    <access specifier> <return type> method2(parameter_list) {  
        // method body  
    }  
    ...  
    <access specifier> <return type> methodN(parameter_list) {  
        // method body  
    }  
}
```

Note –

- Access specifiers specify the access rules for the members as well as the class itself. If not mentioned, then the default access specifier for a class type is **internal**. Default access for the members is **private**.
- Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
- To access the class members, you use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

The following example illustrates the concepts discussed so far –

Member Functions and Member variables

A member function of a class is a function that has its definition or its prototype within the class definition similar to any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member variables are the attributes of an object (from design perspective) and they are kept private to implement encapsulation. These variables can only be accessed using the public member functions.

Let us put above concepts to set and get the value of different class members in a class –

```
public class BankAccount
{
    public string Number { get; } //member variables
    public string Owner { get; set; } //member variables
    public decimal Balance { get; } //member variables

    public void MakeDeposit(decimal amount, DateTime date, string note) //member function
    {

    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note) //member function
    {

    }

}
```

Declaration of Method

A method is a block of code that performs a specific task. Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

method to draw the circle

method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

Return type methodname ()

```
{
//method body
}
```

Here,

ReturnType - It specifies what type of value a method returns. For example, if a method has an `int` return type then it returns an `int` value.

If the method does not return a value, its return type is `void`.

MethodName - It is an identifier that is used to refer to the particular method in a program.

Method body - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces `{ }`

Example:

```
void Display()  
  
{  
  
//code block  
  
}
```

Calling a Method

In the above example, we have declared a method named `display()`. Now, to use the method, we need to call it.

Here's how we can call the `display()` method.

//calling

Display();

```
void display() {  
    // code  
}  
....  
display();  
....
```

Example of method

```
namespace Method  
{  
    class Program  
    {  
        // method declaration  
        public void display()  
        {  
            Console.WriteLine("Hello World");  
        }  
  
        static void Main(string[] args)  
        {  
            // create class object  
            Program p1 = new Program();  
  
            //call method  
            p1.display();  
  
            Console.ReadLine();  
        }  
    }  
}
```

```
}  
}  
}
```

Methods Parameters

In C#, we can also create a method that accepts some value. These values are called method parameters. For example,

```
Int addnumber(int a int b)
```

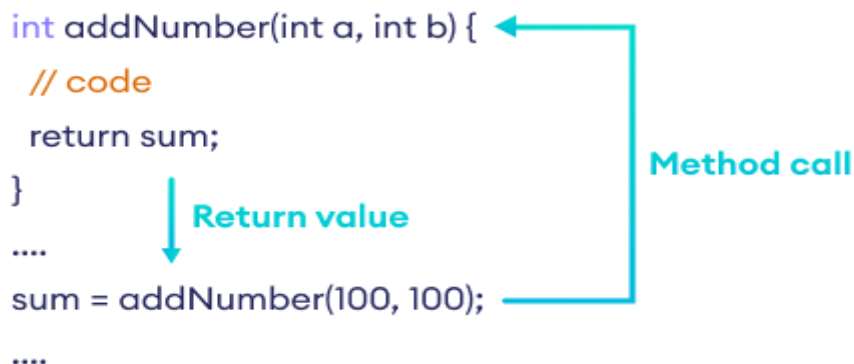
```
{
```

```
//code
```

```
}
```

```
//calling method
```

```
addnumber(100,200)
```



Access modifiers

All types and type members have an accessibility level. The accessibility level controls whether they can be used from other code in your assembly or other assemblies.

- Public: Code in any assembly can access this type or member. The accessibility level of the containing type controls the accessibility level of public members of the type.
- Private: Only code declared in the same class or struct can access this member.
- Protected: Only code in the same class or in a derived class can access this type or member.
- Internal: Only code in the same assembly can access this type or member.

- Protected internal: Only code in the same assembly *or* in a derived class in another assembly can access this type or member.
- Private protected: Only code in the same assembly *and* in the same class or a derived class can access the type or member.

C# - Methods

A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.

To use a method, you need to –

- Define the method
- Call the method

Defining Methods in C#

When you define a method, you basically declare the elements of its structure. The syntax for defining a method in C# is as follows –

```
<Access Specifier> <Return Type> <Method Name>(Parameter List) {
    Method Body
}
```

Following are the various elements of a method –

- **Access Specifier** – This determines the visibility of a variable or a method from another class.
- **Return type** – A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**.
- **Method name** – Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list** – Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method body** – This contains the set of instructions needed to complete the required activity.

Example

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator {
```

```

public int FindMax(int num1, int num2) {
    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
...
}

```

Calling Methods in C#

You can call a method using the name of the method. The following example illustrates this –

```

using System;

namespace CalculatorApplication {

    class NumberManipulator {

        public int FindMax(int num1, int num2) {
            /* local variable declaration */
            int result;

            if (num1 > num2)
                result = num1;
            else
                result = num2;
            return result;
        }

        static void Main(string[] args) {
            /* local variable definition */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();

            //calling the FindMax method

```



```

        ret = n.FindMax(a, b);
        Console.WriteLine("Max value is : {0}", ret );
        Console.ReadLine();
    }
}
}

```

When the above code is compiled and executed, it produces the following result –

Max value is : 200

You can also call public method from other classes by using the instance of the class. For example, the method *FindMax* belongs to the *NumberManipulator* class, you can call it from another class *Test*.

```

using System;

namespace CalculatorApplication {

    class NumberManipulator {

        public int FindMax(int num1, int num2) {
            /* local variable declaration */
            int result;

            if(num1 > num2)
                result = num1;
            else
                result = num2;

            return result;
        }
    }

    class Test {

        static void Main(string[] args) {
            /* local variable definition */
            int a = 100;
            int b = 200;
            int ret;
            NumberManipulator n = new NumberManipulator();

            //calling the FindMax method
            ret = n.FindMax(a, b);
        }
    }
}

```

```

        Console.WriteLine("Max value is : {0}", ret );
        Console.ReadLine();
    }
}
}

```

When the above code is compiled and executed, it produces the following result –

Max value is : 200

Recursive Method Call

A method can call itself. This is known as **recursion**. Following is an example that calculates factorial for a given number using a recursive function –

```

using System;

namespace CalculatorApplication {

    class NumberManipulator {

        public int factorial(int num) {
            /* local variable declaration */
            int result;
            if (num == 1) {
                return 1;
            }
            else {
                result = factorial(num - 1) * num;
                return result;
            }
        }

        static void Main(string[] args) {
            NumberManipulator n = new NumberManipulator();
            //calling the factorial method {0}", n.factorial(6));
            Console.WriteLine("Factorial of 7 is : {0}", n.factorial(7));
            Console.WriteLine("Factorial of 8 is : {0}", n.factorial(8));
            Console.ReadLine();
        }
    }
}

```

When the above code is compiled and executed, it produces the following result –

Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320

Passing Parameters to a Method

When method with parameters is called, you need to pass the parameters to the method. There are three ways that parameters can be passed to a method –

Sr.No.	Mechanism & Description
1	Value parameters This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Reference parameters This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
3	Output parameters This method helps in returning more than one value.

C# Constructors

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor has exactly the same name as that of class and it does not have any return type. Following example explains the concept of constructor –

```
using System;

namespace LineApplication {

    class Line {
        private double length; // Length of a line

        public Line() {
            Console.WriteLine("Object is being created");
        }
    }
}
```

```

public void setLength( double len ) {
    length = len;
}

public double getLength() {
    return length;
}

static void Main(string[] args) {
    Line line = new Line();

    // set line length
    line.setLength(6.0);
    Console.WriteLine("Length of line : {0}", line.getLength());
    Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result –

Object is being created

Length of line : 6

A **default constructor** does not have any parameter but if you need, a constructor can have parameters. Such constructors are called **parameterized constructors**. This technique helps you to assign initial value to an object at the time of its creation as shown in the following example –

```

using System;

namespace LineApplication {

    class Line {
        private double length; // Length of a line

        public Line(double len) { //Parameterized constructor
            Console.WriteLine("Object is being created, length = {0}", len);
            length = len;
        }

        public void setLength( double len ) {
            length = len;
        }
    }
}

```

```

public double getLength() {
    return length;
}

static void Main(string[] args) {
    Line line = new Line(10.0);
    Console.WriteLine("Length of line : {0}", line.getLength());

    // set line length
    line.setLength(6.0);
    Console.WriteLine("Length of line : {0}", line.getLength());
    Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result –

Object is being created, length = 10

Length of line : 10

Length of line : 6

C# Destructors

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope. A **destructor** has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.

Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

Following example explains the concept of destructor –

```

using System;

namespace LineApplication {

    class Line {
        private double length; // Length of a line

        public Line() { // constructor
            Console.WriteLine("Object is being created");
        }

        ~Line() { //destructor

```

```

        Console.WriteLine("Object is being deleted");
    }

    public void setLength( double len ) {
        length = len;
    }

    public double getLength() {
        return length;
    }

    static void Main(string[] args) {
        Line line = new Line();

        // set line length
        line.setLength(6.0);
        Console.WriteLine("Length of line : {0}", line.getLength());
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Object is being created
Length of line : 6
Object is being deleted

```

Object in C#

In C#, an object is a fundamental data type that represents a real-world entity. It combines data (variables) and methods (functions) that operate on the data, encapsulating behavior and state. Objects are instances of classes, defining their structure and behavior, facilitating the principles of object-oriented programming in C#.

An object consists of:

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Declaring Objects

A class is said to be instantiated when an instance of the class is created. The characteristics and actions of the class are shared by all instances. However, each object has a different value for these characteristics, which is the state. Any number of instances can be found within a class.

Objects refers to things present in the real world. For example, a shapes program may have objects such as “triangle”, “square”, “circle”. An online learning system might have objects such as “student”, and “course”.

Example:

```
class Program
{
    static void Main()
    {
        // Initializing an object of the Person class
        Person person = new Person();

        // Setting properties of the object
        person.Name = "Employee";
        person.Age = 25;

        // Displaying object properties
        Console.WriteLine("Name: " + person.Name);
        Console.WriteLine("Age: " + person.Age);
    }
}
```

Example of Objects and Classes in C#

```
using System;

class BankAccount
{
    private string accountNumber;
    private decimal balance;

    public BankAccount(string accountNumber, decimal balance)
    {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public void Deposit(decimal amount)
    {
        balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= balance)
        {
            balance -= amount;
        }
        else
        {
            Console.WriteLine("Insufficient funds.");
        }
    }

    public void PrintBalance()
    {
        Console.WriteLine($"Account Number: {accountNumber}");
        Console.WriteLine($"Balance: {balance:C}");
    }
}

class Program
```

```
{
    static void Main(string[] args)
    {
        BankAccount myAccount = new BankAccount("123456789", 1000);

        myAccount.PrintBalance();

        myAccount.Deposit(500);
        myAccount.PrintBalance();

        myAccount.Withdraw(200);
        myAccount.PrintBalance();

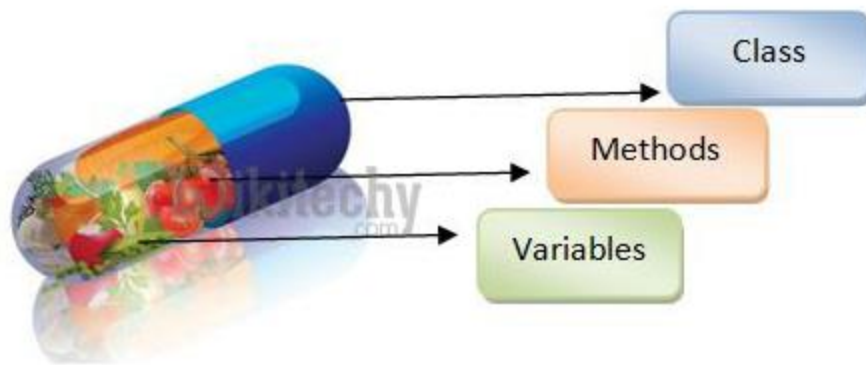
        myAccount.Withdraw(2000); // This will result in "Insufficient funds" message
        myAccount.PrintBalance();
    }
}
```

Difference between Objects and Classes in C#

Objects	Classes
An object is an instance of a class.	A class is a blueprint or a template that defines the structure and behavior of objects
It represents a specific entity or an individual occurrence based on the class definition.	It encapsulates data (attributes/properties) and functions (methods) that operate on that data.
Objects have state (values of attributes/properties) and behavior (methods).	Classes define the common characteristics shared by multiple objects.
They are created using the new keyword, which allocates memory and initializes the object.	They serve as a blueprint for creating objects.
Objects can interact with each other by invoking methods or accessing properties.	Classes are defined using the class keyword.

Chapter 5: C# - Encapsulation

Encapsulation is defined 'as the process of enclosing one or more items within a physical or logical package'. Encapsulation, in object oriented programming methodology, prevents access to implementation details.



Abstraction and encapsulation are related features in object oriented programming. Abstraction allows making relevant information visible and encapsulation enables a programmer to *implement the desired level of abstraction*.

```
using System;

class BankAccountProtected
{
    public void CloseAccount()
    {
        ApplyPenalties();
        CalculateFinalInterest();
        DeleteAccountFromDB();
    }

    protected virtual void ApplyPenalties()
    {
        // deduct from account
    }

    protected virtual void CalculateFinalInterest()
    {
        // add to account
    }

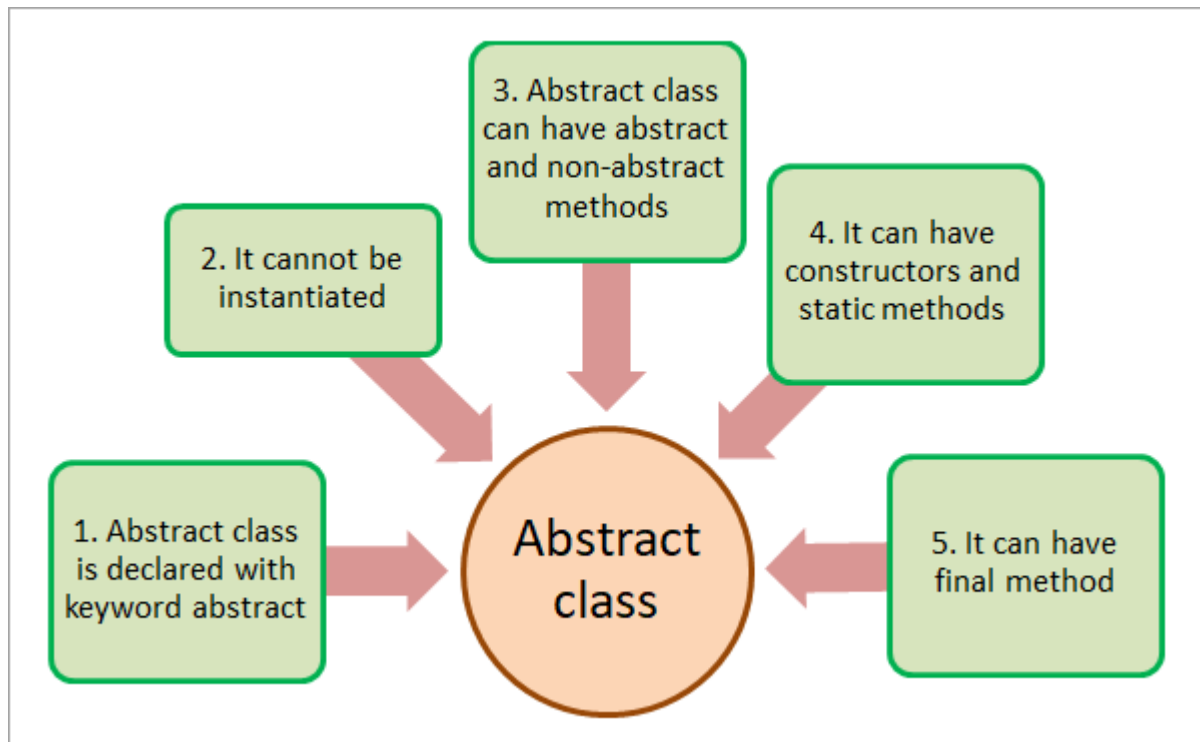
    protected virtual void DeleteAccountFromDB()
    {
        // send notification to data entry personnel
    }
}
```

Derived SavingsAccount Class Using protected Members of its Base Class: SavingsAccount.cs

```
using System;

class SavingsAccount : BankAccountProtected
```

```
{  
    protected override void ApplyPenalties()  
    {  
        Console.WriteLine("Savings Account Applying Penalties");  
    }  
  
    protected override void CalculateFinalInterest()  
    {  
        Console.WriteLine("Savings Account Calculating Final Interest");  
    }  
  
    protected override void DeleteAccountFromDB()  
    {  
        base.DeleteAccountFromDB();  
        Console.WriteLine("Savings Account Deleting Account from DB");  
    }  
}
```



Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
4. Abstraction - Outer layout, used in terms of design. For Example:- Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	4. Encapsulation - Inner layout, used in terms of implementation. For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member. C# supports the following access specifiers –

- Public
- Private
- Protected
- Internal
- Protected internal

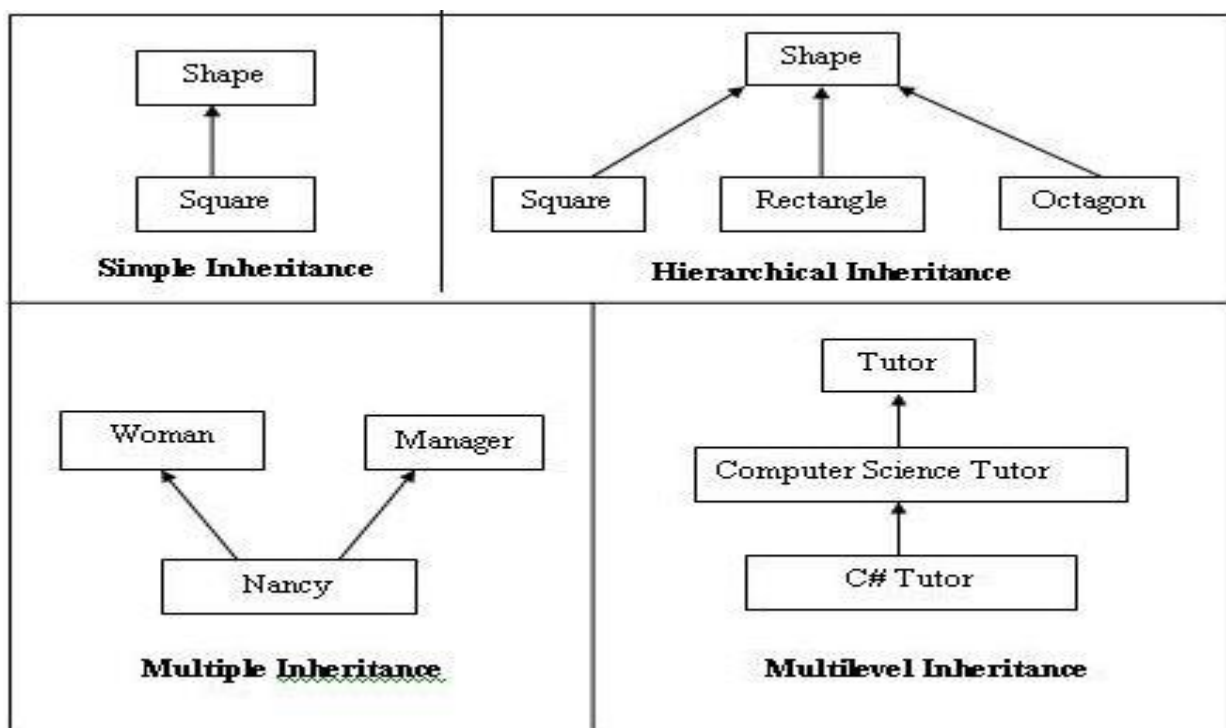
visibility keyword	Containing Classes	Derived Classes	Containing Assembly	Anywhere outside the containing assembly
public	yes	yes	yes	yes
protected internal	yes	yes	yes	no
protected	yes	yes	no	no
private	yes	no	no	no
internal	yes	no	yes	no

Chapter 6: C# - Inheritance

One of the most important concepts in object-oriented programming is inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **IS-A** relationship. For example, mammal **IS A** animal, dog **IS-A** mammal hence dog **IS-A** animal as well, and so on.



Base and Derived Classes

A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

The syntax used in C# for creating derived classes is as follows –

```
<access-specifier> class <base_class> {  
    ...  
}
```

```
class <derived_class> : <base_class> {
```

```
...  
}
```

Example:

```
namespace ConsoleAppsingleInt  
{  
  
    public class Employee//base class  
    {  
        public string name = "Emp1";// feilds of base class  
        public float salary = 40000;  
    }  
    public class Programmer : Employee//derived class programmer  
    {  
        public string Dept = "HR";  
        public float bonus = 10000;//feilds of derived class  
    }  
    class TestInheritance  
    {  
        public static void Main(string[] args)  
        {  
            Programmer p1 = new Programmer();//always create object on derived  
  
            Console.WriteLine("Salary: " + p1.salary);  
            Console.WriteLine("Bonus: " + p1.bonus);  
            Console.WriteLine("Dept: " + p1.Dept);  
            Console.WriteLine("Name: " + p1.name);  
            Console.ReadLine();  
        }  
    }  
}
```

Multiple Inheritance in C#

C# does not support multiple inheritance. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this –

```
namespace ConsoleAppIMultipleIn  
{  
    interface calc1  
    {  
        int add(int a, int b);  
    }  
    interface calc2  
    {  
        int sub(int x, int y);  
    }  
    interface calc3  
    {  
        int mul(int r, int s);  
    }  
    interface calc4  
    {  
        int div(int c, int d);  
    }  
}
```

```

}
class Calculation : calc1, calc2, calc3, calc4 //interface declare above
{
    public int result1;//data member
    public int add(int a, int b)
    {
        return result1 = a + b;
    }
    public int result2;
    public int sub(int x, int y)
    {
        return result2 = x - y;
    }
    public int result3;
    public int mul(int r, int s)
    {
        return result3 = r * s;
    }
    public int result4;
    public int div(int c, int d)
    {
        return result4 = c / d;
    }
}

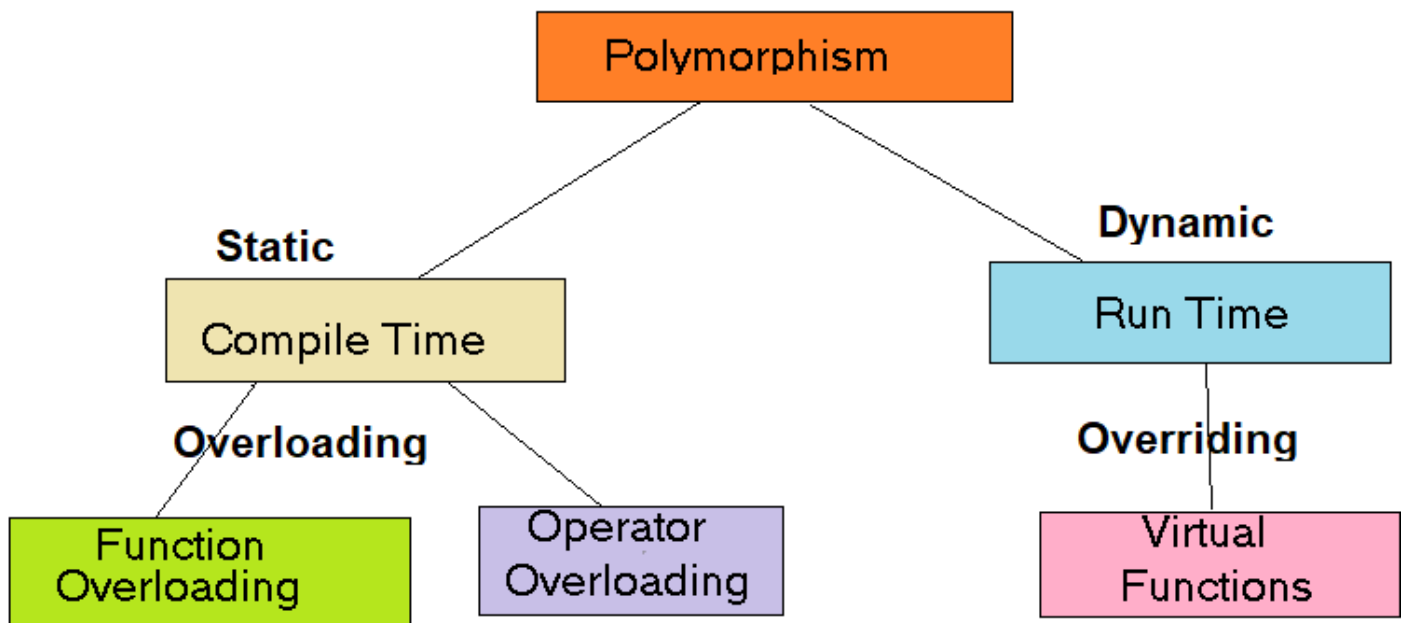
class Program
{
    static void Main(string[] args)
    {
        Calculation c = new Calculation();
        c.add(8, 1);
        c.sub(20, 10);
        c.mul(5, 2);
        c.div(20, 10);
        Console.WriteLine("Multiple Inheritance concept Using Interfaces :\n ");
        Console.WriteLine("Addition: " + c.result1);
        Console.WriteLine("Substraction: " + c.result2);
        Console.WriteLine("Multiplication : " + c.result3);
        Console.WriteLine("Division: " + c.result4);
        Console.ReadLine();
    }
}
}

```

Chapter 7: C# - Polymorphism

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.



Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are –

- Function overloading
- Operator overloading

We discuss operator overloading in next chapter.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

The following example shows using function **print()** to print different data types –

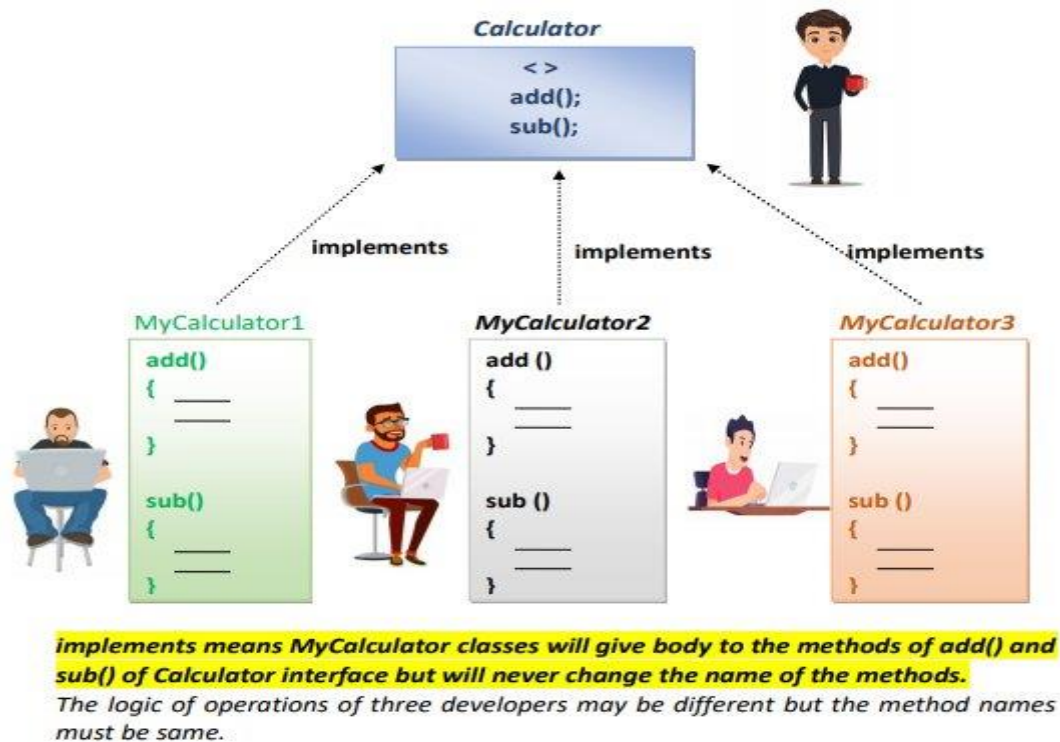
```
using System;
```

When the above code is compiled and executed, it produces the following result –

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.



Here are the rules about abstract classes –

- You cannot create an instance of an abstract class
- You cannot declare an abstract method outside an abstract class
- When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class –

```
namespace Functionoverload
{
    class Printdetails
    {
        void print(int i)
        {
            Console.WriteLine("Printing int: {0}", i);
        }

        void print(double f)
        {
            Console.WriteLine("Printing float: {0}", f);
        }
    }
}
```



```

    }

    void print(string s)
    {
        Console.WriteLine("Printing string: {0}", s);
    }

    static void Main(string[] args)
    {
        Printdetails p = new Printdetails();

        // Call print to print integer
        p.print(5);

        // Call print to print float
        p.print(500.263);

        // Call print to print string
        p.print("Hello C++");
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

Rectangle class area :

Area: 70

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this –

```

namespace Consoleabstractclass
{
    class Program
    {
        abstract class AreaClass
        {
            // declare method
            // 'Area' as abstract
            abstract public int Area();
        }

        // class 'AreaClass' inherit
        // in child class 'Square'
        class Square : AreaClass//child class of area
        {
            int side = 0;

```

```

// constructor
public Square(int n)//constructor with parameter
{
    side = n;
}

// the abstract method
// 'Area' is overridden here
public override int Area()
{
    return side * side;
}
}

class cal
{
    // Main Method
    public static void Main()
    {
        Square s = new Square(6);
        Console.WriteLine("Area = " + s.Area());
        Console.Read();
    }
}
}

```

Chapter 8: C# - Operator Overloading

You can redefine or overload most of the built-in operators available in C#. Thus a programmer can use operators with user-defined types as well. Overloaded operators are functions with special names the keyword **operator** followed by the symbol for the operator being defined. similar to any other function, an overloaded operator has a return type and a parameter list.

For example, go through the following function –

```

public static Box operator+ (Box b, Box c) {
    Box box = new Box();
    box.length = b.length + c.length;
    box.breadth = b.breadth + c.breadth;
    box.height = b.height + c.height;
    return box;
}

```

The above function implements the addition operator (+) for a user-defined class Box. It adds the attributes of two Box objects and returns the resultant Box object.

Implementing the Operator Overloading

The following program shows the complete implementation –

```
using System;

class Matrix
{
    public const int DimSize = 3;
    private double[,] m_matrix = new double[DimSize, DimSize];

    // allow callers to initialize
    public double this[int x, int y]
    {
        get { return m_matrix[x, y]; }
        set { m_matrix[x, y] = value; }
    }

    // let user add matrices
    public static Matrix operator +(Matrix mat1, Matrix mat2)
    {
        Matrix newMatrix = new Matrix();

        for (int x=0; x < DimSize; x++)
            for (int y=0; y < DimSize; y++)
                newMatrix[x, y] = mat1[x, y] + mat2[x, y];

        return newMatrix;
    }
}

class MatrixTest
{
    // used in the InitMatrix method.
    public static Random m_rand = new Random();

    // test Matrix
    static void Main()
    {
        Matrix mat1 = new Matrix();
        Matrix mat2 = new Matrix();

        // init matrices with random values
        InitMatrix(mat1);
        InitMatrix(mat2);

        // print out matrices
        Console.WriteLine("Matrix 1: ");
        PrintMatrix(mat1);

        Console.WriteLine("Matrix 2: ");
        PrintMatrix(mat2);
    }
}
```

```

results      // perform operation and print out

Matrix mat3 = mat1 + mat2;

Console.WriteLine();
Console.WriteLine("Matrix 1 + Matrix
2 = ");

PrintMatrix(mat3);

Console.ReadLine();

}

// initialize matrix with random values
public static void InitMatrix(Matrix mat)
{
    for (int x=0; x < Matrix.DimSize; x++)
        for (int y=0; y < Matrix.DimSize; y++)
            mat[x, y] = m_rand.NextDouble();
}

// print matrix to console
public static void PrintMatrix(Matrix mat)
{
    Console.WriteLine();
    for (int x=0; x < Matrix.DimSize; x++)
    {
        Console.Write("[ ");
        for (int y=0; y < Matrix.DimSize; y++)
        {
            // format the output
            Console.Write("{0,8:#.000000}", mat[x, y]);

            if ((y+1 % 2) < 3)
                Console.Write(", ");
        }
        Console.WriteLine("]");
    }
    Console.WriteLine();
}
}

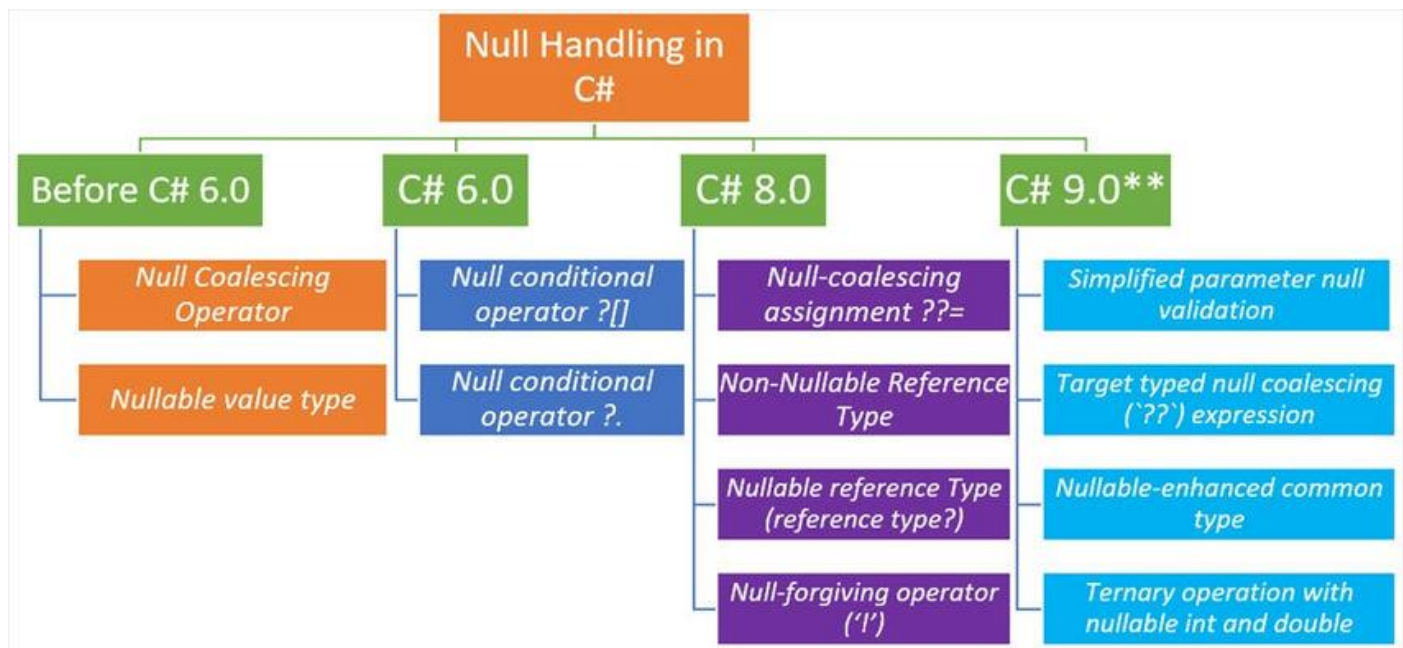
```

Chapter 9 C# - Nullables

C# provides a special data types, the **nullable** types, to which you can assign normal range of values as well as null values.

For example, you can store any value from -2,147,483,648 to 2,147,483,647 or null in a `Nullable<Int32>` variable. Similarly, you can assign true, false, or null in a `Nullable<bool>` variable. Syntax for declaring a **nullable** type is as follows –

< data_type> ? <variable_name> = null;



The following example demonstrates use of nullable data types –

```
using System;

namespace CalculatorApplication {

    class NullablesAtShow {

        static void Main(string[] args) {
            int? num1 = null;
            int? num2 = 45;
            double? num3 = new double?();
            double? num4 = 3.14157;

            bool? boolval = new bool?();

            // display the values
            Console.WriteLine("Nullables at Show: {0}, {1}, {2}, {3}", num1, num2, num3, num4);
            Console.WriteLine("A Nullable boolean value: {0}", boolval);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

Nullables at Show: , 45, , 3.14157

A Nullable boolean value:

The Null Coalescing Operator (??)

The null coalescing operator is used with the nullable value types and reference types. It is used for converting an operand to the type of another nullable (or not) value type operand, where an implicit conversion is possible.

If the value of the first operand is null, then the operator returns the value of the second operand, otherwise it returns the value of the first operand. The following example explains this –

```
using System;

namespace CalculatorApplication {

    class NullablesAtShow {

        static void Main(string[] args) {
            double? num1 = null;
            double? num2 = 3.14157;
            double num3;
            num3 = num1 ?? 5.34;
            Console.WriteLine(" Value of num3: {0}", num3);
            num3 = num2 ?? 5.34;
            Console.WriteLine(" Value of num3: {0}", num3);
            Console.ReadLine();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

Value of num3: 5.34

Value of num3: 3.14157

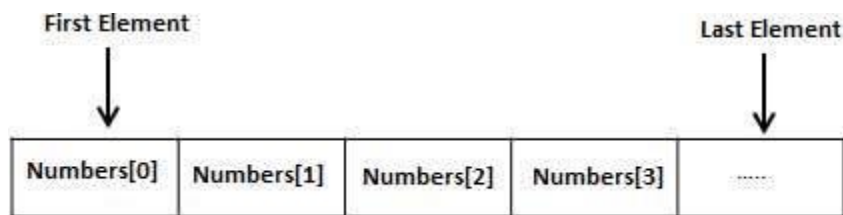
Chapter 10: C# - Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type stored at contiguous memory locations.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ...,

numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C#, you can use the following syntax –

```
datatype[] arrayName;
```

where,

- *datatype* is used to specify the type of elements in the array.
- *[]* specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

For example,

```
double[] balance;
```

Initializing an Array

Declaring an array does not initialize the array in the memory. When the array variable is initialized, you can assign values to the array.

Array is a reference type, so you need to use the **new** keyword to create an instance of the array. For example,

```
double[] balance = new double[10];
```

Assigning Values to an Array

You can assign values to individual array elements, by using the index number, like –

```
double[] balance = new double[10];  
balance[0] = 4500.0;
```

You can assign values to the array at the time of declaration, as shown –

```
double[] balance = { 2340.0, 4523.69, 3421.0};
```

You can also create and initialize an array, as shown –

```
int [] marks = new int[5] { 99, 98, 92, 97, 95};
```

You may also omit the size of the array, as shown –

```
int [] marks = new int[] { 99, 98, 92, 97, 95};
```

You can copy an array variable into another target array variable. In such case, both the target and source point to the same memory location –

```
int [] marks = new int[] { 99, 98, 92, 97, 95};  
int[] score = marks;
```

When you create an array, C# compiler implicitly initializes each array element to a default value depending on the array type. For example, for an int array all elements are initialized to 0.

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example,

```
double salary = balance[9];
```

The following example, demonstrates the above-mentioned concepts declaration, assignment, and accessing arrays –

```
using System;  
  
namespace ArrayApplication {  
  
    class MyArray {  
  
        static void Main(string[] args) {  
            int [] n = new int[10]; /* n is an array of 10 integers */  
            int i,j;  
  
            /* initialize elements of array n */  
            for ( i = 0; i < 10; i++ ) {  
                n[ i ] = i + 100;  
            }  
        }  
    }  
}
```



```

    /* output each array element's value */
    for (j = 0; j < 10; j++ ) {
        Console.WriteLine("Element[{0}] = {1}", j, n[j]);
    }
    Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result –

```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

Using the *foreach* Loop

In the previous example, we used a for loop for accessing each array element. You can also use a **foreach** statement to iterate through an array.

```

using System;

namespace ArrayApplication {

    class MyArray {

        static void Main(string[] args) {
            int [] n = new int[10]; /* n is an array of 10 integers */

            /* initialize elements of array n */
            for ( int i = 0; i < 10; i++ ) {
                n[i] = i + 100;
            }

            /* output each array element's value */
            foreach (int j in n ) {

```

```

        int i = j-100;
        Console.WriteLine("Element[{0}] = {1}", i, j);

    }
    Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result –

```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

C# Arrays

There are following few important concepts related to array which should be clear to a C# programmer –

Sr.No.	Concept & Description
1	Multi-dimensional arrays C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Jagged arrays C# supports multidimensional arrays, which are arrays of arrays.
3	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
4	Param arrays This is used for passing unknown number of parameters to a function.
5	The Array Class Defined in System namespace, it is the base class to all arrays, and provides various properties and methods for working with arrays.

Chapter 11: C# - Strings

In C#, you can use strings as array of characters, However, more common practice is to use the **string** keyword to declare a string variable. The string keyword is an alias for the **System.String** class.

Creating a String Object

You can create string object using one of the following methods –

- By assigning a string literal to a String variable
- By using a String class constructor
- By using the string concatenation operator (+)
- By retrieving a property or calling a method that returns a string
- By calling a formatting method to convert a value or an object to its string representation

The following example demonstrates this –

```
using System;

namespace StringApplication {

    class Program {

        static void Main(string[] args) {
            //from string literal and string concatenation
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            char []letters= { 'H', 'e', 'l', 'l', 'o' };
            string [] sarray={ "Hello", "From", "Enosis Learnings", "Point" };

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //by using string constructor { 'H', 'e', 'l', 'l', 'o' };
```

```

string greetings = new string(letters);
Console.WriteLine("Greetings: {0}", greetings);

//methods returning string { "Hello", "From", "Enosis Learnings", "Point" };
string message = String.Join(" ", sarray);
Console.WriteLine("Message: {0}", message);

//formatting method to convert a value
DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
string chat = String.Format("Message sent at {0:t} on {0:D}", waiting);
Console.WriteLine("Message: {0}", chat);
    }
}
}

```

When the above code is compiled and executed, it produces the following result –

Full Name: RowanAtkinson

Greetings: Hello

Message: Hello From Enosis Learning

Message: Message sent at 5:58 PM on Wednesday, October 10, 2012

Chapter 12: C# - Structures

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member for your program.

For example, here is the way you can declare the Book structure –

```

struct Books {
    public string title;
    public string author;
    public string subject;
}

```

```
public int book_id;
};
```

The following program shows the use of the structure –

```
using System;

struct Books {
    public string title;
    public string author;
    public string subject;
    public int book_id;
};

public class testStructure {

    public static void Main(string[] args) {
        Books Book1; /* Declare Book1 of type Book */
        Books Book2; /* Declare Book2 of type Book */

        /* book 1 specification */
        Book1.title = "C Programming";
        Book1.author = "Nuha Ali";
        Book1.subject = "C Programming Enosis Learning";
        Book1.book_id = 6495407;

        /* book 2 specification */
        Book2.title = "Telecom Billing";
        Book2.author = "Zara Ali";
        Book2.subject = "Telecom Billing Enosis Learning";
        Book2.book_id = 6495700;

        /* print Book1 info */
        Console.WriteLine( "Book 1 title : {0}", Book1.title);
        Console.WriteLine("Book 1 author : {0}", Book1.author);
        Console.WriteLine("Book 1 subject : {0}", Book1.subject);
        Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

        /* print Book2 info */
        Console.WriteLine("Book 2 title : {0}", Book2.title);
        Console.WriteLine("Book 2 author : {0}", Book2.author);
        Console.WriteLine("Book 2 subject : {0}", Book2.subject);
        Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);
    }
}
```

```
Console.ReadKey();  
}  
}
```

When the above code is compiled and executed, it produces the following result –

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Enosis Learning

Book 1 book_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Enosis Learning

Book 2 book_id : 6495700

Features of C# Structures

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features –

- Structures can have methods, fields, indexers, properties, operator methods, and events.
- Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
- Unlike classes, structures cannot inherit other structures or classes.
- Structures cannot be used as a base for other structures or classes.
- A structure can implement one or more interfaces.
- Structure members cannot be specified as abstract, virtual, or protected.
- When you create a struct object using the **New** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
- If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

Class versus Structure

Classes and Structures have the following basic differences –

- classes are reference types and structs are value types
- structures do not support inheritance
- structures cannot have default constructor

In the light of the above discussions, let us rewrite the previous example –

```

using System;

struct Books {
    private string title;
    private string author;
    private string subject;
    private int book_id;

    public void getValues(string t, string a, string s, int id) {
        title = t;
        author = a;
        subject = s;
        book_id = id;
    }

    public void display() {
        Console.WriteLine("Title : {0}", title);
        Console.WriteLine("Author : {0}", author);
        Console.WriteLine("Subject : {0}", subject);
        Console.WriteLine("Book_id :{0}", book_id);
    }
};

public class testStructure {

    public static void Main(string[] args) {
        Books Book1 = new Books(); /* Declare Book1 of type Book */
        Books Book2 = new Books(); /* Declare Book2 of type Book */

        /* book 1 specification */
        Book1.getValues("C Programming",
            "Nuha Ali", "C Programming Enosis Learning",6495407);

        /* book 2 specification */
        Book2.getValues("Telecom Billing",
            "Zara Ali", "Telecom Billing Enosis Learning", 6495700);

        /* print Book1 info */
        Book1.display();

        /* print Book2 info */
        Book2.display();

        Console.ReadKey();
    }
}

```

```
}  
}
```

When the above code is compiled and executed, it produces the following result –

Title : C Programming

Author : Nuha Ali

Subject : C Programming Enosis Learning

Book_id : 6495407

Title : Telecom Billing

Author : Zara Ali

Subject : Telecom Billing Enosis Learning

Book_id : 6495700

Chapter 13: C# - Enums

An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword.

C# enumerations are value data type. In other words, enumeration contains its own values and cannot inherit or cannot pass inheritance.

Declaring *enum* Variable

The general syntax for declaring an enumeration is –

```
enum <enum_name> {  
    enumeration list  
};
```

Where,

- The *enum_name* specifies the enumeration type name.
- The *enumeration list* is a comma-separated list of identifiers.

Each of the symbols in the enumeration list stands for an integer value, one greater than the symbol that precedes it. By default, the value of the first enumeration symbol is 0. For example –

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

Example

The following example demonstrates use of enum variable –


```
using System;

namespace EnumApplication {

    class EnumProgram {
        enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

        static void Main(string[] args) {
            int WeekdayStart = (int)Days.Mon;
            int WeekdayEnd = (int)Days.Fri;
            Console.WriteLine("Monday: {0}", WeekdayStart);
            Console.WriteLine("Friday: {0}", WeekdayEnd);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

```
Monday: 1
Friday: 5
```

Chapter 14: C# - Interfaces

An interface is defined as a syntactical contract that all the classes inheriting the interface should follow. The interface defines the '**what**' part of the syntactical contract and the deriving classes define the '**how**' part of the syntactical contract.

Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

Declaring Interfaces

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration –

```
public interface ITransactions {
```

```
// interface members
void showTransaction();
double getAmount();
}
```

Example

The following example demonstrates implementation of the above interface –

```
namespace InterfaceApplication
{
    public interface ITransactions
    {
        // interface members
        void showTransaction();
        double getAmount();
    }

    public class Transaction : ITransactions
    {
        private string tCode;
        private string date;
        private double amount;

        public Transaction()
        {
            tCode = " ";
            date = " ";
            amount = 0.0;
        }

        public Transaction(string c, string d, double a)
        {
            tCode = c;
            date = d;
            amount = a;
        }

        public double getAmount()
        {
            return amount;
        }

        public void showTransaction()
        {
            Console.WriteLine("Transaction: {0}", tCode);
            Console.WriteLine("Date: {0}", date);
            Console.WriteLine("Amount: {0}", getAmount());
        }
    }

    class Tester
    {
        static void Main(string[] args)
        {
            Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
            Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
        }
    }
}
```

```

t1.showTransaction();
t2.showTransaction();
Console.ReadKey();
}
}
}

```

What is an Interface?

- A factory may require all machines to have a standard "interface" to simplify employee training.



Abstract Class	Interface
An Abstract class doesn't provide full abstraction	Interface does provide full abstraction
Using Abstract we can not achieve multiple inheritance	using an Interface we can achieve multiple inheritance.
We can declare a member field	We can not declare a member field in an Interface
An abstract class can contain access modifiers for the subs, functions, properties	We can not use any access modifier i.e. public , private , protected , internal etc. because within an interface by default everything is public
An abstract class can be defined	An Interface member cannot be defined using the keyword static, virtual, abstract or sealed
A class may inherit only one abstract class.	A class may inherit several interfaces.
An abstract class can provide complete, default code and/or just the details that have to be overridden.	An interface cannot provide any code, just the signature.

Chapter 15 C# - Namespaces

A **namespace** is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace does not conflict with the same class names declared in another.

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows –

```
namespace namespace_name {  
    // code declarations  
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows –

```
namespace_name.item_name;
```

The following program demonstrates use of namespaces –

```
using System;  
  
namespace first_space {  
    class namespace_cl {  
        public void func() {  
            Console.WriteLine("Inside first_space");  
        }  
    }  
}  
  
namespace second_space {  
    class namespace_cl {  
        public void func() {  
            Console.WriteLine("Inside second_space");  
        }  
    }  
}  
  
class TestClass {  
    static void Main(string[] args) {  
        first_space.namespace_cl fc = new first_space.namespace_cl();  
        second_space.namespace_cl sc = new second_space.namespace_cl();  
        fc.func();  
    }  
}
```

```
    sc.func();
    Console.ReadKey();
}
}
```

When the above code is compiled and executed, it produces the following result –

```
Inside first_space
Inside second_space
```

The *using* Keyword

The **using** keyword states that the program is using the names in the given namespace. For example, we are using the **System** namespace in our programs. The class Console is defined there. We just write –

```
Console.WriteLine ("Hello there");
```

We could have written the fully qualified name as –

```
System.Console.WriteLine("Hello there");
```

You can also avoid prepending of namespaces with the **using** namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code –

Let us rewrite our preceding example, with using directive –

```
using System;
using first_space;
using second_space;

namespace first_space {

    class abc {

        public void func() {
            Console.WriteLine("Inside first_space");
        }
    }
}

namespace second_space {
```

```

class efg {

    public void func() {
        Console.WriteLine("Inside second_space");
    }
}

class TestClass {

    static void Main(string[] args) {
        abc fc = new abc();
        efg sc = new efg();
        fc.func();
        sc.func();
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Inside first_space
Inside second_space

```

Nested Namespaces

You can define one namespace inside another namespace as follows –

```

namespace namespace_name1 {

    // code declarations
    namespace namespace_name2 {
        // code declarations
    }
}

```

You can access members of nested namespace by using the dot (.) operator as follows –

```

using System;
using first_space;
using first_space.second_space;

namespace first_space {

```

```

class abc {

    public void func() {
        Console.WriteLine("Inside first_space");
    }
}

namespace second_space {

    class efg {

        public void func() {
            Console.WriteLine("Inside second_space");
        }
    }
}

class TestClass {

    static void Main(string[] args) {
        abc fc = new abc();
        efg sc = new efg();
        fc.func();
        sc.func();
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Inside first_space
Inside second_space

```

Chapter 16: C# - Preprocessor Directives

The preprocessor directives give instruction to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not statements, so they do not end with a semicolon (;).

C# compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In C# the preprocessor directives are used to help in conditional compilation.

Unlike C and C++ directives, they are not used to create macros. A preprocessor directive must be the only instruction on a line.

Preprocessor Directives in C#

The following table lists the preprocessor directives available in C# –

Sr.No.	Preprocessor Directive & Description
1	#define It defines a sequence of characters, called symbol.
2	#undef It allows you to undefine a symbol.
3	#if It allows testing a symbol or symbols to see if they evaluate to true.
4	#else It allows to create a compound conditional directive, along with #if.
5	#elif It allows creating a compound conditional directive.
6	#endif Specifies the end of a conditional directive.
7	#line It lets you modify the compiler's line number and (optionally) the file name output for errors and warnings.
8	#error It allows generating an error from a specific location in your code.
9	#warning It allows generating a level one warning from a specific location in your code.
10	#region It lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.
11	#endregion It marks the end of a #region block.

Chapter 17: C# - Regular Expressions

A **regular expression** is a pattern that could be matched against an input text. The .Net framework provides a regular expression engine that allows such matching. A pattern consists of one or more character literals, operators, or constructs.

Constructs for Defining Regular Expressions

There are various categories of characters, operators, and constructs that lets you to define regular expressions. Click the following links to find these constructs.

- [Character escapes](#)
- [Character classes](#)
- [Anchors](#)
- [Grouping constructs](#)
- [Quantifiers](#)
- [Backreference constructs](#)
- [Alternation constructs](#)
- [Substitutions](#)
- [Miscellaneous constructs](#)

The Regex Class

The Regex class is used for representing a regular expression. It has the following commonly used methods –

Sr.No.	Methods & Description
1	public bool IsMatch(string input) Indicates whether the regular expression specified in the Regex constructor finds a match in a specified input string.
2	public bool IsMatch(string input, int startat) Indicates whether the regular expression specified in the Regex constructor finds a match in the specified input string, beginning at the specified starting position in the string.
3	public static bool IsMatch(string input, string pattern) Indicates whether the specified regular expression finds a match in the specified input string.
4	public MatchCollection Matches(string input) Searches the specified input string for all occurrences of a regular expression.
5	public string Replace(string input, string replacement) In a specified input string, replaces all strings that match a regular expression pattern with a specified replacement string.
6	public string[] Split(string input) Splits an input string into an array of substrings at the positions defined by a regular expression pattern specified in the Regex constructor.

Example 1

The following example matches words that start with 'S' –

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
    class Program
    {
        private static void showMatch(string text, string expr)
        {
            Console.WriteLine("The Expression: " + expr);
            MatchCollection mc = Regex.Matches(text, expr);
            foreach (Match m in mc)
            {
                Console.WriteLine(m);
            }
        }

        static void Main(string[] args)
        {
            string str = "A Thousand Splendid Suns";

            Console.WriteLine("Matching words that start with 'S': ");
            showMatch(str, @"\bS\S*");
            Console.ReadKey();
        }
    }
}
```

Example 2

The following example matches words that start with 'm' and ends with 'e' –

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication {

    class Program {
        private static void showMatch(string text, string expr) {
            Console.WriteLine("The Expression: " + expr);
            MatchCollection mc = Regex.Matches(text, expr);
            foreach (Match m in mc) {
                Console.WriteLine(m);
            }
        }

        static void Main(string[] args) {
            string str = "make maze and manage to measure it";

            Console.WriteLine("Matching words start with 'm' and ends with 'e':");
        }
    }
}
```

```

        showMatch(str, @"\bm\S*e\b");
        Console.ReadKey();
    }
}
}

```

When the above code is compiled and executed, it produces the following result –

Matching words start with 'm' and ends with 'e':

The Expression: `\bm\S*e\b`

make

maze

manage

measure

Example:

```

namespace ReglarEXp
{
    class Program
    {
        static void Main(string[] args)
        {
            Regex r = new Regex(@"^+?\d{0,2}\-?\d{4,5}\-?\d{5,6}");
            //class Regex represents an immutable regular expression.

            string[] str = { "+91-9678967101", "9678967101", "+91-9678-967101", "+91-96789-67101", "+919678967101" };
            //Input strings for Match valid mobile number.
            foreach (string s in str)
            {
                Console.WriteLine("{0} {1} a valid mobile number.", s,
                    r.IsMatch(s) ? "is" : "is not");
                //The IsMatch method is used to validate a string or
                //to ensure that a string conforms to a particular pattern.
            }
        }
    }
}

```

Chapter 18: C# - Exception Handling

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** – A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** – The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** – A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

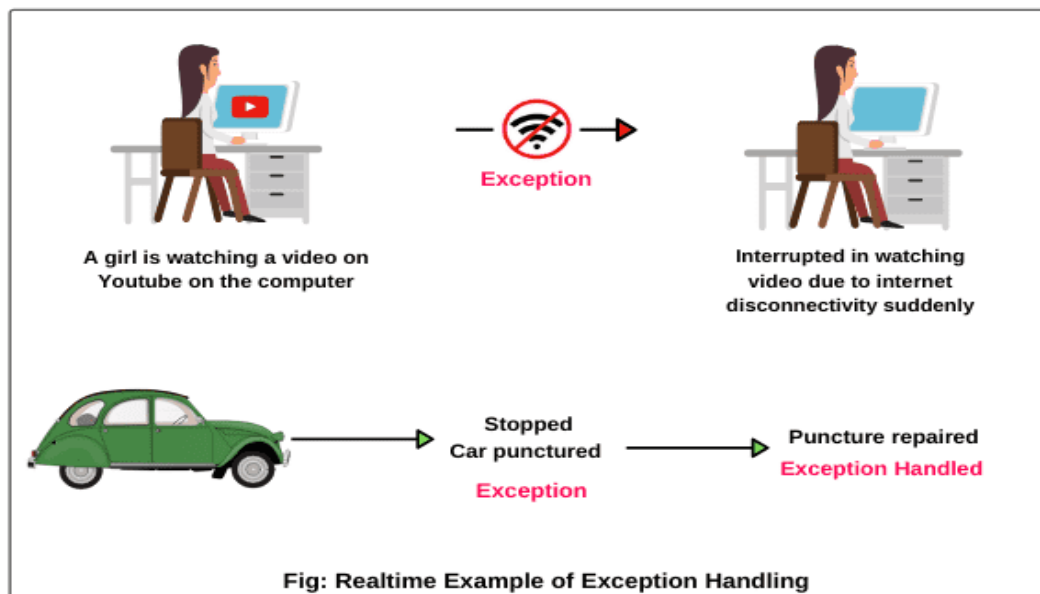
Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

```
try {
    // statements causing exception
} catch( ExceptionName e1 ) {
    // error handling code
} catch( ExceptionName e2 ) {
    // error handling code
} catch( ExceptionName eN ) {
    // error handling code
} finally {
    // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.



The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class –

Sr.No.	Exception Class & Description
1	System.IO.IOException Handles I/O errors.
2	System.IndexOutOfRangeException Handles errors generated when a method refers to an array index out of range.
3	System.ArrayTypeMismatchException Handles errors generated when type is mismatched with the array type.
4	System.NullReferenceException Handles errors generated from referencing a null object.
5	System.DivideByZeroException Handles errors generated from dividing a dividend with zero.
6	System.InvalidCastException Handles errors generated during typecasting.
7	System.OutOfMemoryException Handles errors generated from insufficient free memory.
8	System.StackOverflowException

	Handles errors generated from stack overflow.
--	---

Handling Exceptions

C# provides a structured solution to the exception handling in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch**, and **finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs –

```
using System;

namespace ErrorHandlingApplication {

    class DivNumbers {
        int result;

        DivNumbers() {
            result = 0;
        }

        public void division(int num1, int num2) {
            try {
                result = num1 / num2;
            } catch (DivideByZeroException e) {
                Console.WriteLine("Exception caught: {0}", e);
            } finally {
                Console.WriteLine("Result: {0}", result);
            }
        }

        static void Main(string[] args) {
            DivNumbers d = new DivNumbers();
            d.division(25, 0);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

Exception caught: System.DivideByZeroException: Attempted to divide by zero.

at ...

Result: 0

Creating User-Defined Exceptions

You can also define your own exception. User-defined exception classes are derived from the **Exception** class. The following example demonstrates this –

```
using System;

namespace UserDefinedException {

    class TestTemperature {

        static void Main(string[] args) {
            Temperature temp = new Temperature();
            try {
                temp.showTemp();
            } catch(TemplsZeroException e) {
                Console.WriteLine("TemplsZeroException: {0}", e.Message);
            }
            Console.ReadKey();
        }
    }
}

public class TemplsZeroException: Exception {

    public TemplsZeroException(string message): base(message) {

    }
}

public class Temperature {
    int temperature = 0;

    public void showTemp() {

        if(temperature == 0) {
            throw (new TemplsZeroException("Zero Temperature found"));
        } else {
            Console.WriteLine("Temperature: {0}", temperature);
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

TemplsZeroException: Zero Temperature found

Throwing Objects

You can throw an object if it is either directly or indirectly derived from the **System.Exception** class. You can use a throw statement in the catch block to throw the present object as –

```
Catch(Exception e) {  
    ...  
    Throw e  
}
```

Chapter 19: C# - File I/O

A **file** is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a **stream**.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the **input stream** and the **output stream**. The **input stream** is used for reading data from file (read operation) and the **output stream** is used for writing into the file (write operation).

C# I/O Classes

The System.IO namespace has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file etc.

The following table shows some commonly used non-abstract classes in the System.IO namespace –

Sr.No.	I/O Class & Description
1	BinaryReader Reads primitive data from a binary stream.
2	BinaryWriter Writes primitive data in binary format.
3	BufferedStream A temporary storage for a stream of bytes.
4	Directory Helps in manipulating a directory structure.
5	DirectoryInfo Used for performing operations on directories.
6	DriveInfo Provides information for the drives.

7	File Helps in manipulating files.
8	FileInfo Used for performing operations on files.
9	FileStream Used to read from and write to any location in a file.
10	MemoryStream Used for random access to streamed data stored in memory.
11	Path Performs operations on path information.
12	StreamReader Used for reading characters from a byte stream.
13	StreamWriter Is used for writing characters to a stream.
14	StringReader Is used for reading from a string buffer.
15	StringWriter Is used for writing into a string buffer.

The FileStream Class

The **FileStream** class in the System.IO namespace helps in reading from, writing to and closing files. This class derives from the abstract class Stream.

You need to create a **FileStream** object to create a new file or open an existing file. The syntax for creating a **FileStream** object is as follows –

```
FileStream <object_name> = new FileStream( <file_name>, <FileMode Enumerator>,
    <FileAccess Enumerator>, <FileShare Enumerator>);
```

For example, we create a FileStream object **F** for reading a file named **sample.txt** as shown –

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read,
    FileShare.Read);
```

Sr.No.	Parameter & Description
1	FileMode The FileMode enumerator defines various methods for opening files. The members of the FileMode enumerator are –

	<ul style="list-style-type: none"> • Append – It opens an existing file and puts cursor at the end of file, or creates the file, if the file does not exist. • Create – It creates a new file. • CreateNew – It specifies to the operating system, that it should create a new file. • Open – It opens an existing file. • OpenOrCreate – It specifies to the operating system that it should open a file if it exists, otherwise it should create a new file. • Truncate – It opens an existing file and truncates its size to zero bytes.
2	FileAccess FileAccess enumerators have members: Read , ReadWrite and Write .
3	FileShare FileShare enumerators have the following members – <ul style="list-style-type: none"> • Inheritable – It allows a file handle to pass inheritance to the child processes • None – It declines sharing of the current file • Read – It allows opening the file for readin. • ReadWrite – It allows opening the file for reading and writing • Write – It allows opening the file for writing

Example

The following program demonstrates use of the **FileStream** class –

```
using System;
using System.IO;

namespace FileIOApplication {

    class Program {

        static void Main(string[] args) {
            FileStream F = new FileStream("test.dat", FileMode.OpenOrCreate,
                FileAccess.ReadWrite);

            for (int i = 1; i <= 20; i++) {
                F.WriteByte((byte)i);
            }
            F.Position = 0;
            for (int i = 0; i <= 20; i++) {
                Console.Write(F.ReadByte() + " ");
            }
            F.Close();
            Console.ReadKey();
        }
    }
}
```

```
}
```

When the above code is compiled and executed, it produces the following result –

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

Advanced File Operations in C#

The preceding example provides simple file operations in C#. However, to utilize the immense powers of C# System.IO classes, you need to know the commonly used properties and methods of these classes.

Sr.No.	Topic & Description
1	Reading from and Writing into Text files It involves reading from and writing into text files. The StreamReader and StreamWriter class helps to accomplish it.
2	Reading from and Writing into Binary files It involves reading from and writing into binary files. The BinaryReader and BinaryWriter class helps to accomplish this.
3	Manipulating the Windows file system It gives a C# programmer the ability to browse and locate Windows files and directories.

Example to create Directory:

```
using System;
using System.Collections.Generic;
using System.IO;

namespace DirectoryCreate
{
    internal class Program
    {
        public static void Main()
        {
            // Create directory named Enosis in C drive
            // Using CreateDirectory() method
            //Directory.CreateDirectory("C:\\Enosis");

            //Console.WriteLine(" Directory Created");
            //Console.ReadLine();

            // Check whether the directory named
            // Enosis exists or not
            // Using Exists() method
            if (Directory.Exists("C:/Enosis"))
                Console.WriteLine("Exists");
            else
                Console.WriteLine("Not Exist");
            Console.ReadLine();
        }
    }
}
```

```
}  
}  
}
```

```
using System;  
using System.IO;  
  
namespace ConsoleAppWriteInExistingFile  
{  
    class fileexercise3  
    {  
        public static void Main()  
        {  
            string OriginalFile = @"D:\source.txt";  
            string FinalFile = @"D:\finalnew.txt";  
            try  
            {  
                // Method used to copy an existing file into a new file.  
                File.Copy(OriginalFile, FinalFile);  
            }  
            catch (IOException iox)  
            {  
                // Exception message catching box  
                Console.WriteLine(iox.Message);  
            }  
            Console.WriteLine("Copying of contents from source to destination file is completed");  
        }  
    }  
}
```

Chapter 20: C# - Attributes

An **attribute** is a declarative tag that is used to convey information to runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program. You can add declarative information to a program by using an attribute. A declarative tag is depicted by square ([]) brackets placed above the element it is used for.

Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program. The .Net Framework provides two types of attributes: *the pre-defined* attributes and *custom built* attributes.

Specifying an Attribute

Syntax for specifying an attribute is as follows –

```
[attribute(positional_parameters, name_parameter = value, ...)]  
element
```

Name of the attribute and its values are specified within the square brackets, before the element to which the attribute is applied. Positional parameters specify the essential information and the name parameters specify the optional information.

Predefined Attributes

The .Net Framework provides three pre-defined attributes –

- `AttributeUsage`
- `Conditional`
- `Obsolete`

AttributeUsage

The pre-defined attribute **AttributeUsage** describes how a custom attribute class can be used. It specifies the types of items to which the attribute can be applied.

Syntax for specifying this attribute is as follows –

```
[AttributeUsage (  
    validon,  
    AllowMultiple = allowmultiple,  
    Inherited = inherited  
)]
```

Where,

- The parameter `validon` specifies the language elements on which the attribute can be placed. It is a combination of the value of an enumerator *AttributeTargets*. The default value is *AttributeTargets.All*.
- The parameter *allowmultiple* (optional) provides value for the *AllowMultiple* property of this attribute, a Boolean value. If this is true, the attribute is multiuse. The default is false (single-use).
- The parameter `inherited` (optional) provides value for the *Inherited* property of this attribute, a Boolean value. If it is true, the attribute is inherited by derived classes. The default value is false (not inherited).

For example,

```
[AttributeUsage(AttributeTargets.Class |  
AttributeTargets.Constructor |  
AttributeTargets.Field |  
AttributeTargets.Method |  
AttributeTargets.Property,  
AllowMultiple = true)]
```

Conditional

This predefined attribute marks a conditional method whose execution depends on a specified preprocessing identifier.

It causes conditional compilation of method calls, depending on the specified value such as **Debug** or **Trace**. For example, it displays the values of the variables while debugging a code.

Syntax for specifying this attribute is as follows –

```
[Conditional(  
    conditionalSymbol  
)]
```

For example,

```
[Conditional("DEBUG")]
```

The following example demonstrates the attribute –

```
#define DEBUG  
using System;  
using System.Diagnostics;  
  
public class Myclass {  
    [Conditional("DEBUG")]  
  
    public static void Message(string msg) {  
        Console.WriteLine(msg);  
    }  
}  
  
class Test {  
    static void function1() {  
        Myclass.Message("In Function 1.");  
        function2();  
    }  
  
    static void function2() {  
        Myclass.Message("In Function 2.");  
    }  
  
    public static void Main() {  
        Myclass.Message("In Main function.");  
        function1();  
    }  
}
```

```
    Console.ReadKey();  
}  
}
```

When the above code is compiled and executed, it produces the following result –

In Main function

In Function 1

In Function 2

Obsolete

This predefined attribute marks a program entity that should not be used. It enables you to inform the compiler to discard a particular target element. For example, when a new method is being used in a class and if you still want to retain the old method in the class, you may mark it as obsolete by displaying a message the new method should be used, instead of the old method.

Syntax for specifying this attribute is as follows –

```
[Obsolete (  
    message  
)]
```

```
[Obsolete (  
    message,  
    iserror  
)]
```

Where,

- The parameter *message*, is a string describing the reason why the item is obsolete and what to use instead.
- The parameter *iserror*, is a Boolean value. If its value is true, the compiler should treat the use of the item as an error. Default value is false (compiler generates a warning).

The following program demonstrates this –

```
using System;  
  
public class MyClass {  
    [Obsolete("Don't use OldMethod, use NewMethod instead", true)]  
  
    static void OldMethod() {  
        Console.WriteLine("It is the old method");  
    }  
}
```

```

}

static void NewMethod() {
    Console.WriteLine("It is the new method");
}

public static void Main() {
    OldMethod();
}
}

```

When you try to compile the program, the compiler gives an error message stating –

Don't use OldMethod, use NewMethod instead

Creating Custom Attributes

The .Net Framework allows creation of custom attributes that can be used to store declarative information and can be retrieved at run-time. This information can be related to any target element depending upon the design criteria and application need.

Creating and using custom attributes involve four steps –

- Declaring a custom attribute
- Constructing the custom attribute
- Apply the custom attribute on a target program element
- Accessing Attributes Through Reflection

The Last step involves writing a simple program to read through the metadata to find various notations. Metadata is data about data or information used for describing other data. This program should use reflections for accessing attributes at runtime. This we will discuss in the next chapter.

Declaring a Custom Attribute

A new custom attribute should is derived from the **System.Attribute** class. For example,

```

//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

```



```
public class DeBugInfo : System.Attribute
```

In the preceding code, we have declared a custom attribute named *DeBugInfo*.

Constructing the Custom Attribute

Let us construct a custom attribute named *DeBugInfo*, which stores the information obtained by debugging any program. Let it store the following information –

- The code number for the bug
- Name of the developer who identified the bug
- Date of last review of the code
- A string message for storing the developer's remarks

The *DeBugInfo* class has three private properties for storing the first three information and a public property for storing the message. Hence the bug number, developer's name, and date of review are the positional parameters of the *DeBugInfo* class and the message is an optional or named parameter.

Each attribute must have at least one constructor. The positional parameters should be passed through the constructor. The following code shows the *DeBugInfo* class –

```
//a custom attribute BugFix to be assigned to a class and its members
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute {
    private int bugNo;
    private string developer;
    private string lastReview;
    public string message;

    public DeBugInfo(int bg, string dev, string d) {
        this.bugNo = bg;
        this.developer = dev;
        this.lastReview = d;
    }

    public int BugNo {
        get {
            return bugNo;
        }
    }
}
```

```

    }
}

public string Developer {
    get {
        return developer;
    }
}

public string LastReview {
    get {
        return lastReview;
    }
}

public string Message {
    get {
        return message;
    }
    set {
        message = value;
    }
}
}

```

Applying the Custom Attribute

The attribute is applied by placing it immediately before its target –

```

[Debugger(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[Debugger(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle {
    //member variables
    protected double length;
    protected double width;
    public Rectangle(double l, double w) {
        length = l;
        width = w;
    }
    [Debugger(55, "Zara Ali", "19/10/2012", Message = "Return type mismatch")]

    public double GetArea() {
        return length * width;
    }
}

```

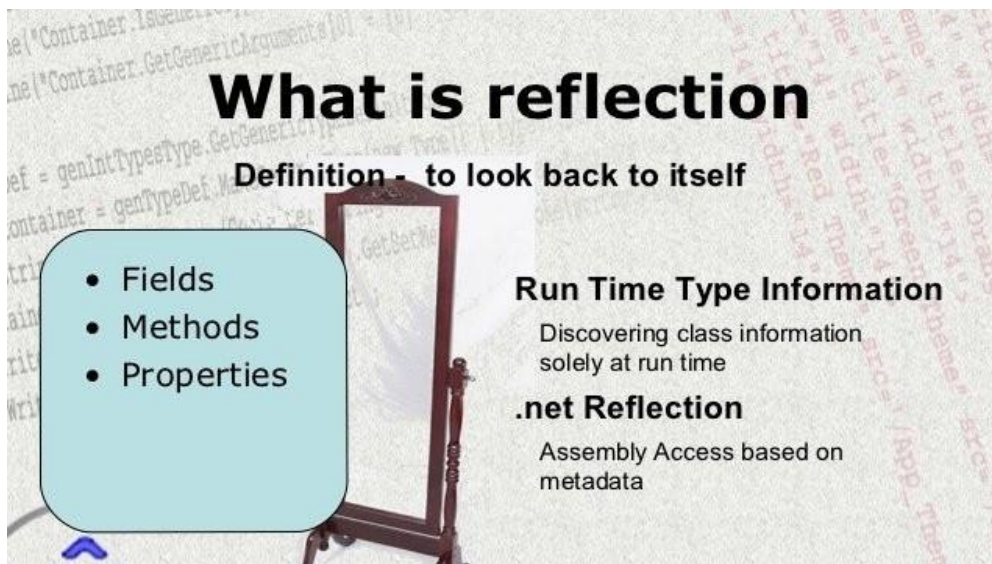
```
[DebugInfo(56, "Zara Ali", "19/10/2012")]
```

```
public void Display() {  
    Console.WriteLine("Length: {0}", length);  
    Console.WriteLine("Width: {0}", width);  
    Console.WriteLine("Area: {0}", GetArea());  
}  
}
```

In the next chapter, we retrieve attribute information using a Reflection class object.

Chapter 21: C# - Reflection

Reflection objects are used for obtaining type information at runtime. The classes that give access to the metadata of a running program are in the **System.Reflection** namespace.



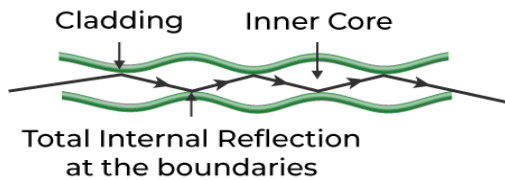
The **System.Reflection** namespace contains classes that allow you to obtain information about the application and to dynamically add types, values, and objects to the application.

Applications of Reflection

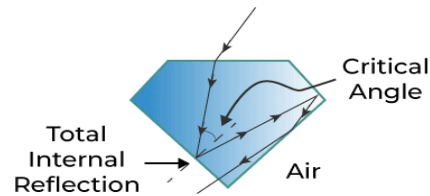
Reflection has the following applications –

- It allows view attribute information at runtime.
- It allows examining various types in an assembly and instantiate these types.
- It allows late binding to methods and properties
- It allows creating new types at runtime and then performs some tasks using those types.

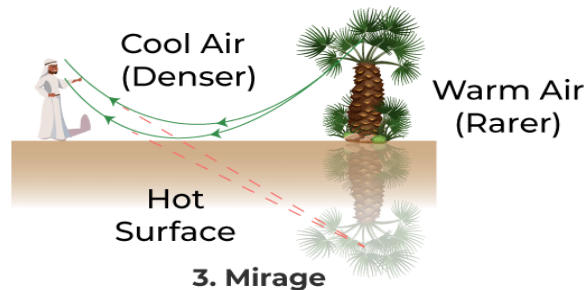
Examples of Total Internal reflection



1. Optical Fibre



2. Diamond



Viewing Metadata

We have mentioned in the preceding chapter that using reflection you can view the attribute information.

The **MemberInfo** object of the **System.Reflection** class needs to be initialized for discovering the attributes associated with a class. To do this, you define an object of the target class, as –

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

The following program demonstrates this –

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute {
    public readonly string Url;

    public string Topic // Topic is a named parameter {
        get {
            return topic;
        }

        set {
            topic = value;
        }
    }

    public HelpAttribute(string url) // url is a positional parameter {
```

```

    this.Url = url;
}
private string topic;
}

[HelpAttribute("Information on the class MyClass")]
class MyClass {

}

namespace AttributeAppl {

    class Program {

        static void Main(string[] args) {
            System.Reflection.MemberInfo info = typeof(MyClass);
            object[] attributes = info.GetCustomAttributes(true);
            for (int i = 0; i < attributes.Length; i++) {
                System.Console.WriteLine(attributes[i]);
            }
            Console.ReadKey();
        }
    }
}

```

Reflection Example: Get Type

```

using System;
public class ReflectionExample
{
    public static void Main()
    {
        int a = 10;
        Type type = a.GetType();
        Console.WriteLine(type);
    }
}

```

Reflection Example: Get Assembly

```

using System;
using System.Reflection;
public class ReflectionExample
{
    public static void Main()
    {
        Type t = typeof(System.String);
    }
}

```

```

        Console.WriteLine(t.Assembly);
    }
}

```

Reflection Example: Print Fields

```

using System;
using System.Reflection;
public class ReflectionExample
{
    public static void Main()
    {
        Type t = typeof(System.String);

        Console.WriteLine("Fields of {0} type...", t);
        FieldInfo[] ci = t.GetFields(BindingFlags.Public | BindingFlags.Static | BindingFlags.NonPublic);
        foreach (FieldInfo f in ci)
        {
            Console.WriteLine(f);
        }
    }
}

```

Anonymous Functions

Anonymous function is a type of function that does not has name. In other words, we can say that a function without name is known as anonymous function.

Benefits of Anonymous Functions

Anonymous functions offer several benefits to C# developers.

- ✓ **Conciseness:** They allow you to write compact and readable code, especially for small and simple operations.
- ✓ **Improved Readability:** When used appropriately, anonymous functions can enhance code readability by keeping related logic close together.
- ✓ **Reduced Boilerplate:** You don't need to create separate methods for small functions, reducing the clutter in your codebase.
- ✓ **Encapsulation:** They help encapsulate logic where it's needed without exposing it to the broader scope.
- ✓ **Functional Programming:** They enable functional programming techniques in C#, facilitating a more declarative and expressive coding style.

In C#, there are two types of anonymous functions:

- Lambda Expressions
- Anonymous Methods

C# Lambda Expressions

Lambda expression is an anonymous function which we can use to create delegates. We can use lambda expression to create local functions that can be passed as an argument. It is also helpful to write LINQ queries.

C# Lambda Expression Syntax

(input-parameters) => expression

```
using System;
namespace LambdaExpressions
{
    class Program
    {
        delegate int Square(int num);
        static void Main(string[] args)
        {
            Square GetSquare = x => x * x;
            int j = GetSquare(5);
            Console.WriteLine("Square: "+j);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Lambda_Expressions {
class Program {
    static void Main(string[] args)
    {
        // List to store numbers
        List<int> numbers = new List<int>() {36, 71, 12,
            15, 29, 18, 27, 17, 9, 34};

        // foreach loop to display the list
        Console.Write("The list : ");
        foreach(var value in numbers)
        {
            Console.Write("{0} ", value);
        }
        Console.WriteLine();

        // Using lambda expression
        // to calculate square of
        // each value in the list
        var square = numbers.Select(x => x * x);
```

```

// foreach loop to display squares
Console.Write("Squares : ");
foreach(var value in square)
{
    Console.Write("{0} ", value);
}
Console.WriteLine();

// Using Lambda expression to
// find all numbers in the list
// divisible by 3
List<int> divBy3 = numbers.FindAll(x => (x % 3) == 0);

// foreach loop to display divBy3
Console.Write("Numbers Divisible by 3 : ");
foreach(var value in divBy3)
{
    Console.Write("{0} ", value);
}
Console.WriteLine();
}
}
}

```

C# Anonymous Method

```

using System;
namespace AnonymousMethods
{
    class Program
    {
        public delegate void AnonymousFun();
        static void Main(string[] args)
        {
            AnonymousFun fun = delegate () {
                Console.WriteLine("This is anonymous function");
            };
            fun();
        }
    }
}

```

```

using System;
using System.Collections.Generic;

namespace AynomusMethod
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> numbers = new List<int> { 5, 2, 8, 1, 3 };

```



```

// Sorting the list using an anonymous method
numbers.Sort(delegate (int x, int y)
{
    return x.CompareTo(y);
});

Console.WriteLine("Sorted numbers:");
foreach (int num in numbers)
{
    Console.Write(num + " ");
}
}
}

```

Chapter 22: C# - Properties

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors** through which the values of the private fields can be read, written or manipulated.

Properties do not name the storage locations. Instead, they have **accessors** that read, write, or compute their values.

For example, let us have a class named Student, with private fields for age, name, and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

Accessors

The **accessor** of a property contains the executable statements that helps in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both. For example –

```

// Declare a Code property of type string:
public string Code {
    get {
        return code;
    }
    set {
        code = value;
    }
}

// Declare a Name property of type string:

```

```

public string Name {
    get {
        return name;
    }
    set {
        name = value;
    }
}

// Declare a Age property of type int:
public int Age {
    get {
        return age;
    }
    set {
        age = value;
    }
}

```

Example

The following example demonstrates use of properties –

```

using System;
namespace Enosis Learningspoint {
    class Student {
        private string code = "N.A";
        private string name = "not known";
        private int age = 0;

        // Declare a Code property of type string:
        public string Code {
            get {
                return code;
            }
            set {
                code = value;
            }
        }

        // Declare a Name property of type string:
        public string Name {
            get {
                return name;
            }
            set {
                name = value;
            }
        }
    }
}

```

```

// Declare a Age property of type int:
public int Age {
    get {
        return age;
    }
    set {
        age = value;
    }
}
public override string ToString() {
    return "Code = " + Code + ", Name = " + Name + ", Age = " + Age;
}
}

class ExampleDemo {
    public static void Main() {

        // Create a new Student object:
        Student s = new Student();

        // Setting code, name and the age of the student
        s.Code = "001";
        s.Name = "Zara";
        s.Age = 9;
        Console.WriteLine("Student Info: {0}", s);

        //let us increase age
        s.Age += 1;
        Console.WriteLine("Student Info: {0}", s);
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10

```

Abstract Properties

An abstract class may have an abstract property, which should be implemented in the derived class. The following program illustrates this –

```

using System;
namespace Enosis Learningspoint {
    public abstract class Person {
        public abstract string Name {

```

```

    get;
    set;
}
public abstract int Age {
    get;
    set;
}
}

class Student : Person {

    private string code = "N.A";
    private string name = "N.A";
    private int age = 0;

    // Declare a Code property of type string:
    public string Code {
        get {
            return code;
        }
        set {
            code = value;
        }
    }

    // Declare a Name property of type string:
    public override string Name {
        get {
            return name;
        }
        set {
            name = value;
        }
    }

    // Declare a Age property of type int:
    public override int Age {
        get {
            return age;
        }
        set {
            age = value;
        }
    }

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

```

```

class ExampleDemo {
    public static void Main() {
        // Create a new Student object:
        Student s = new Student();

        // Setting code, name and the age of the student
        s.Code = "001";
        s.Name = "Zara";
        s.Age = 9;
        Console.WriteLine("Student Info:- {0}", s);

        //let us increase age
        s.Age += 1;
        Console.WriteLine("Student Info:- {0}", s);
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Student Info: Code = 001, Name = Zara, Age = 9
Student Info: Code = 001, Name = Zara, Age = 10

```

Chapter 23: C# - Indexers

An **indexer** allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to a **virtual array**. You can then access the instance of this class using the array access operator ([]).

Syntax

A one dimensional indexer has the following syntax –

```

element-type this[int index] {

    // The get accessor.
    get {
        // return the value specified by index
    }

    // The set accessor.
}

```

```

set {
    // set the value specified by index
}
}

```

Important points to remember on indexers:

- ✓ Indexers are always created with **this** keyword.
- ✓ Parameterized property are called indexer.
- ✓ Indexers are implemented through get and set accessors for the [] operator.
- ✓ ref and out parameter modifiers are not permitted in indexer.
- ✓ The formal parameter list of an indexer corresponds to that of a method and at least one parameter should be specified.
- ✓ Indexer is an instance member so can't be static but property can be static.
- ✓ Indexers are used on group of elements.
- ✓ Indexer is identified by its signature where as a property is identified it's name.
- ✓ Indexers are accessed using indexes where as properties are accessed by names.
- ✓ Indexer can be overloaded.

Use of Indexers

Declaration of behavior of an indexer is to some extent similar to a property. similar to the properties, you use **get** and **set** accessors for defining an indexer. However, properties return or set a specific data member, whereas indexers returns or sets a particular value from the object instance. In other words, it breaks the instance data into smaller parts and indexes each part, gets or sets each part.

Defining a property involves providing a property name. Indexers are not defined with names, but with the **this** keyword, which refers to the object instance. The following example demonstrates the concept –

```

using System;
namespace Indexer_example1
{
    class IndexerClass
    {
        private string[] names = new string[10];
        public string this[int i]
        {
            get
            {
                return names[i];
            }
            set
            {

```

```

        names[i] = value;
    }
}
}
static void Main(string[] args)
{
    IndexerClass Team = new IndexerClass();
    Team[0] = "Rocky";
    Team[1] = "Teena";
    Team[2] = "Ana";
    Team[3] = "Victoria";
    Team[4] = "Yani";
    Team[5] = "Mary";
    Team[6] = "Gomes";
    Team[7] = "Arnold";
    Team[8] = "Mike";
    Team[9] = "Peter";
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(Team[i]);
    }
    Console.ReadKey();
}
}
}

```

Indexers	Properties
Indexers are created with this keyword.	Properties don't require this keyword.
Indexers are identified by signature.	Properties are identified by their names.
Indexers are accessed using indexes.	Properties are accessed by their names.
Indexer are instance member, so can't be static.	Properties can be static as well as instance members.
A get accessor of an indexer has the same formal parameter list as the indexer.	A get accessor of a property has no parameters.
A set accessor of an indexer has the same formal parameter list as the indexer, in addition to the value parameter.	A set accessor of a property contains the implicit value parameter.

Overloaded Indexers

Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not necessary that the indexes have to be integers. C# allows indexes to be of other types, for example, a string.

The following example demonstrates overloaded indexers –

```

using System;

namespace IndexerApplication {

```

```

class IndexedNames {
    private string[] namelist = new string[size];
    static public int size = 10;

    public IndexedNames() {
        for (int i = 0; i < size; i++) {
            namelist[i] = "N. A.";
        }
    }

    public string this[int index] {

        get {
            string tmp;

            if( index >= 0 && index <= size-1 ) {
                tmp = namelist[index];
            } else {
                tmp = "";
            }

            return ( tmp );
        }
        set {
            if( index >= 0 && index <= size-1 ) {
                namelist[index] = value;
            }
        }
    }

    public int this[string name] {

        get {
            int index = 0;

            while(index < size) {
                if (namelist[index] == name) {
                    return index;
                }
                index++;
            }
            return index;
        }

    }

    static void Main(string[] args) {
        IndexedNames names = new IndexedNames();
    }
}

```



```

names[0] = "Zara";
names[1] = "Riz";
names[2] = "Nuha";
names[3] = "Asif";
names[4] = "Davinder";
names[5] = "Sunil";
names[6] = "Rubic";

//using the first indexer with int parameter
for (int i = 0; i < IndexedNames.size; i++) {
    Console.WriteLine(names[i]);
}

//using the second indexer with the string parameter
Console.WriteLine(names["Nuha"]);
Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result –

```

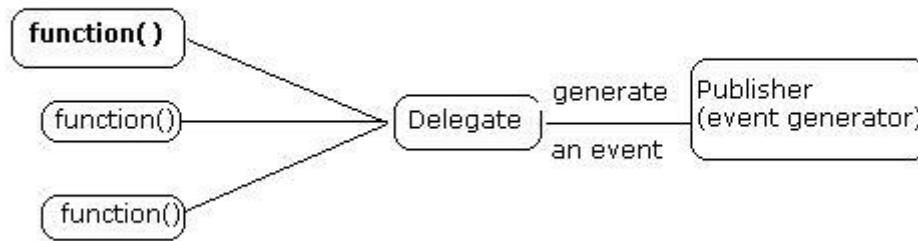
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2

```

Chapter 24: C# - Delegates

C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

Delegate figure



Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which has the same signature as that of the delegate.

For example, consider a delegate –

```
public delegate int MyDelegate (string s);
```

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is –

```
delegate <return type> <delegate-name> <parameter list>
```

Important Points about Delegates:

- ✓ Provides a good way to encapsulate the methods.
- ✓ Delegates are the library class in System namespace.
- ✓ These are the type-safe pointer of any method.
- ✓ Delegates are mainly used in implementing the call-back methods and events.
- ✓ Delegates can be chained together as two or more methods can be called on a single event.
- ✓ It doesn't care about the class of the object that it references.
- ✓ Delegates can also be used in "anonymous methods" invocation



Instantiating Delegates

Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method. For example –

```
public delegate void printString(string s);  
...  
printString ps1 = new printString(WriteToScreen);  
printString ps2 = new printString(WriteToFile);
```

Example:

```
delegate int Calculator(int n); //declaring delegate
```

```
public class DelegateExample
```

```
{  
    static int number = 100;  
    public static int add(int n)  
    {  
        number = number + n;  
        return number;  
    }  
    public static int mul(int n)  
    {  
        number = number * n;  
        return number;  
    }  
    public static int getNumber()  
    {  
        return number;  
    }  
    public static void Main(string[] args)
```

```

{
    Calculator c1 = new Calculator(add);//instantiating delegate
    Calculator c2 = new Calculator(mul);
    c1(20);//calling method using delegate
    Console.WriteLine("After c1 delegate, Number is: " + getNumber());
    c2(3);
    Console.WriteLine("After c2 delegate, Number is: " + getNumber());
}
}

```

Multicasting of a Delegate

Delegate objects can be composed using the "+" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed. The "-" operator can be used to remove a component delegate from a composed delegate.

Using this property of delegates you can create an invocation list of methods that will be called when a delegate is invoked. This is called **multicasting** of a delegate. The following program demonstrates multicasting of a delegate –

```

using System;

delegate int NumberChanger(int n);
namespace DelegateAppl {

    class TestDelegate {
        static int num = 10;

        public static int AddNum(int p) {
            num += p;
            return num;
        }

        public static int MultNum(int q) {
            num *= q;
            return num;
        }

        public static int getNum() {
            return num;
        }

        static void Main(string[] args) {
            //create delegate instances
            NumberChanger nc;
            NumberChanger nc1 = new NumberChanger(AddNum);
            NumberChanger nc2 = new NumberChanger(MultNum);
            nc = nc1;
            nc += nc2;
        }
    }
}

```

```

    //calling multicast
    nc(5);
    Console.WriteLine("Value of Num: {0}", getNum());
    Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result –

Value of Num: 75

Using Delegates

The following example demonstrates the use of delegate. The delegate *printString* can be used to reference method that takes a string as input and returns nothing.

We use this delegate to call two methods, the first prints the string to the console, and the second one prints it to a file –

```

using System;
using System.IO;

namespace DelegateAppl {

    class PrintString {
        static FileStream fs;
        static StreamWriter sw;

        // delegate declaration
        public delegate void printString(string s);

        // this method prints to the console
        public static void WriteToScreen(string str) {
            Console.WriteLine("The String is: {0}", str);
        }

        //this method prints to a file
        public static void WriteToFile(string s) {
            fs = new FileStream("c:\\message.txt",
                FileMode.Append, FileAccess.Write);
            sw = new StreamWriter(fs);
            sw.WriteLine(s);
            sw.Flush();
            sw.Close();
            fs.Close();
        }

        // this method takes the delegate as parameter and uses it to

```

```
// call the methods as required
public static void sendString(printString ps) {
    ps("Hello World");
}

static void Main(string[] args) {
    printString ps1 = new printString(WriteToScreen);
    printString ps2 = new printString(WriteToFile);
    sendString(ps1);
    sendString(ps2);
    Console.ReadKey();
}
}
```

When the above code is compiled and executed, it produces the following result –

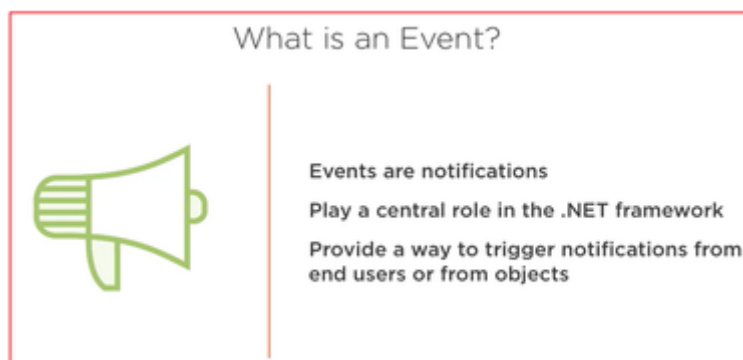
The String is: Hello World

Chapter 25: C# - Events

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

Following are the key points about Event

- ✓ Event Handlers in C# return void and take two parameters.
- ✓ The First parameter of Event - Source of Event means publishing object.
- ✓ The Second parameter of Event - Object derived from EventArgs.
- ✓ The publishers determines when an event is raised and the subscriber determines what action is taken in response.
- ✓ An Event can have so many subscribers.
- ✓ Events are basically used for the single user action like button click.



The syntax for the declaration of an event

```
Public event EventHandler MyEvent;
```

Steps for implementing event: To declare an event inside a class, first, a Delegate type for the event must be declared like below:

```
Public delegate void MyEventHandler (object sender, EventArgs e);
```

Declare an Event: Public event MyEventHandler MyEvent;

Invoking an Event: if (MyEvent! = null) MyEvent (this, e);

Invoking an Event can only be done from within the class that declared the event.

Hooking up Event: MyEventClass.MyEvent += new ChangedEventHandler (MyEventChanged);

Now, detach the event: MyEventClass.MyEvent -= new ChangedEventHandler (MyEventChanged);



Example :Here, an examples of events and response to the events .

Add/Remove Operation in Events

Below is an example of an Event - System.EventHandler *Delegate type*:

Example:

```
public class MyTest
{
    public event EventHandler MyEvent
    {
        add
        {
            Console.WriteLine("add operation");
        }
        remove
        {
            Console.WriteLine("remove operation");
        }
    }
}
```

```

public class Test
{
    public void TestEvent()
    {
        MyTest myTest = new MyTest();
        myTest.MyEvent += myTest_MyEvent;
        myTest.MyEvent -= myTest_MyEvent;
    }
    public void myTest_MyEvent(object sender, EventArgs e)
    {
    }
}
static void Main(string[] args)
{
    Test test = new Test();
    test.TestEvent();
    Console.ReadKey();
}

```

Can we use Events without Delegate?

No, Events use Delegates internally. Events are encapsulation over Delegates. There is already defined Delegate "EventHandler"

Example of Event Using Delegate

```

using System;
namespace Example {
    public delegate string demoDelegate(string str1, string str2);
    class MyEvents {
        event demoDelegate myEvent;
        public MyEvents() {
            this.myEvent += new demoDelegate(this.Display);
        }
        public string Display(string studentname, string subject) {
            return "Student: " + studentname + "\nSubject: " +subject;
        }
        static void Main(string[] args) {
            MyEvents e = new MyEvents();
            string res = e.myEvent("Jack", "Science");
            Console.WriteLine("RESULT...\n"+res);
        }
    }
}

```



```
}
```

The output of the above program is as follows:

RESULT...

Student: Jack

Subject: Science

Declaring and Implementing a Delegate: SimpleDelegate.cs

```
using System;

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

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

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

    // this is the delegate method handler
    public static int CompareFirstNames(object name1, object name2)
    {
        string n1 = ((Name)name1).FirstName;
        string n2 = ((Name)name2).FirstName;

        if (String.Compare(n1, n2) > 0)
        {
            return 1;
        }
        else if (String.Compare(n1, n2) < 0)
        {
            return -1;
        }
        else
        {
            return 0;
        }
    }

    public override string ToString()
    {
        return FirstName + " " + LastName;
    }
}
```

```

}

class SimpleDelegate
{
    Name[] names = new Name[5];

    public SimpleDelegate()
    {
        names[0] = new Name("Joe", "Mayo");
        names[1] = new Name("John", "Hancock");
        names[2] = new Name("Jane", "Doe");
        names[3] = new Name("John", "Doe");
        names[4] = new Name("Jack", "Smith");
    }

    static void Main(string[] args)
    {
        SimpleDelegate sd = new SimpleDelegate();

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

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

        sd.PrintNames();

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

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

        sd.PrintNames();
    }

    // observe the delegate parameter
    public void Sort(Comparer compare)
    {
        object temp;

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

```

```

    }
}

}

}

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

    foreach (Name name in names)
    {
        Console.WriteLine(name.ToString());
    }
}
}

```

We use events for:

- ✓ Decoupling our applications, or rather creating loosely coupled applications
- ✓ Implementing mechanisms for communication between objects
- ✓ Providing a simple yet effective way to extend the application without changing the existing code

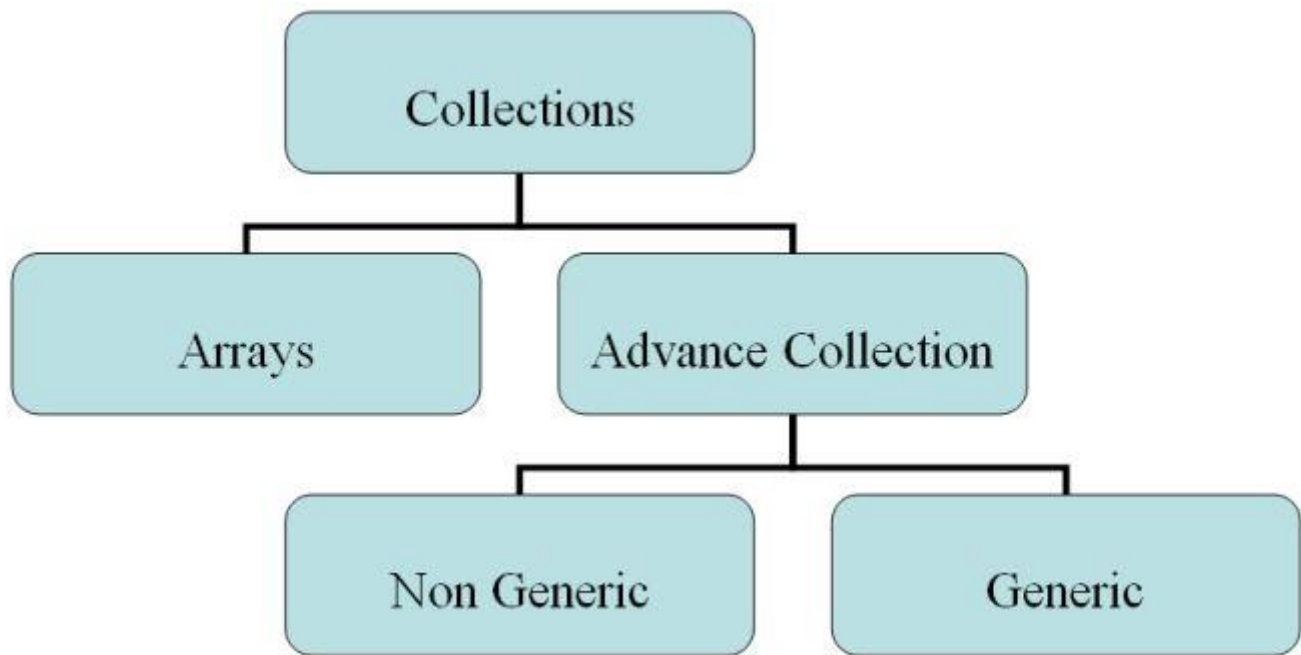
Chapter 26: C# -Collections

Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.

Why a Collection?

- ✓ Array size is fixed and cannot be increased dynamically. However, in actual development scenarios, the same type of objects are needed to be processed and added dynamically, and the size of the unit holding the objects should grow or shrink accordingly. This functionality is provided in Collections.
- ✓ The insertion and deletion of elements in an array are also costly in terms of performance.
- ✓ Also, the way these objects should be stored can be different, there can be a requirement to store the objects sequentially, non-sequentially, sorted order, etc.
- ✓ Collections come in handy as a return type in the implementation of business methods, generally, when data is obtained from a file or database, it can be a single or multiple objects. When one is not sure, it is safe to define return type as a collection since it can hold multiple objects. This works even if a single or no object is returned.
- ✓ A collection is a set of similar types of objects that are grouped together.
- ✓ *System.Collections* namespace contains specialized classes for storing and accessing the data.

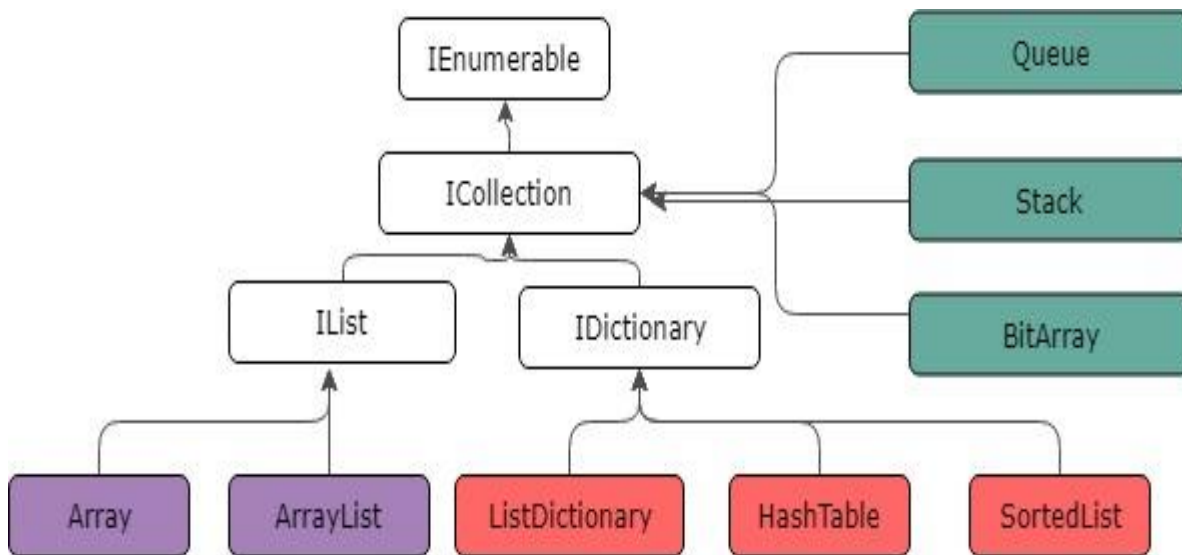
- ✓ Each collection class defined in .NET has a unique feature.



Arrays: Array class is defined in System namespace. Arrays can store any type of data but only one type at a time. The size of the array has to be specified at compilation time. Insertion and deletion reduce performance.

Advanced Collections: Many times we can't give number elements in the list and we need to perform different operations like inserting, deleting, sorting, searching, comparing, and so on. To perform these operations efficiently, the data needs to be organized in a specific way. This gives rise to advanced collections. Advanced collections are found in System.Collections namespace.

Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.



Various Collection Classes and Their Usage

The following are the various commonly used classes of the **System.Collection** namespace. Click the following links to check their detail.

Sr.No.	Class & Description and Useage
1	ArrayList It represents ordered collection of an object that can be indexed individually. It is basically an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an index and the array resizes itself automatically. It also allows dynamic memory allocation, adding, searching and sorting items in the list.
2	Hashtable It uses a key to access the elements in the collection. A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.
3	SortedList It uses a key as well as an index to access the items in a list. A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key , it is a Hashtable. The collection of items is always sorted by the key value.
4	Stack It represents a last-in, first out collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item.
5	Queue

	<p>It represents a first-in, first out collection of object.</p> <p>It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called enqueue and when you remove an item, it is called deque.</p>
6	<p>BitArray</p> <p>It represents an array of the binary representation using the values 1 and 0. It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an integer index, which starts from zero.</p>

Examples

```
using System;
using System.Collections.Generic;
/*The List<T> Generic Collection Class in C# is used to store and fetch elements.
 * It can have duplicate elements. It belongs to System.Collections.Generic namespace.*/

namespace Collection
{
    class Program
    {
        public static void Main(string[] args)
        {
            List<int> integerList = new List<int>();
            integerList.Add(11);
            integerList.Add(22);
            integerList.Add(55);
            integerList.Add(65);
            integerList.Add(10);
            //The following line give you compile time error as the value is string
            //integerList.Add("Hello");
            Console.WriteLine("List of Elements: ");
            foreach (int item in integerList)
            {
                Console.WriteLine($"{item} ");
            }
            Console.ReadKey();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
/* The Generic HashSet<T> Collection Class in C# can be used to store, remove or view elements.
 * It does not allow the addition of duplicate elements.
 * It is suggested to use the HashSet class if you have to store only unique elements.
 * It belongs to System.Collections.Generic namespace.*/

namespace Collection2
{
```

```

class Program
{
    public static void Main(string[] args)
    {
        HashSet<int> integerHashSet = new HashSet<int>();
        integerHashSet.Add(11);
        integerHashSet.Add(22);
        integerHashSet.Add(55);
        integerHashSet.Add(65);
        //Addind Duplicate Elements
        integerHashSet.Add(55);
        //The following line give you compile time error as the value is string
        //integerHashSet.Add("Hello");
        Console.WriteLine("List of Elements: ");
        foreach (int item in integerHashSet)
        {
            Console.WriteLine($"{item} ");
        }
        Console.ReadKey();
    }
}

```

```

using System;
using System.Collections.Generic;
/*The Generic SortedSet<T> Collection Class in C# is used to store, remove or view elements.
 * By default, it stores the elements in ascending order and does not store duplicate elements.
 * It is recommended to use if you have to store unique elements as well as
 * if you want to maintain ascending order.*/
namespace Collection3
{
    class Program
    {
        public static void Main(string[] args)
        {
            SortedSet<int> integerSortedSet = new SortedSet<int>();
            integerSortedSet.Add(11);
            integerSortedSet.Add(66);
            integerSortedSet.Add(55);
            integerSortedSet.Add(88);
            integerSortedSet.Add(22);
            integerSortedSet.Add(77);
            //Addind Duplicate Elements
            integerSortedSet.Add(55);
            //The following line give you compile time error as the value is string
            //integerSortedSet.Add("Hello");
            Console.WriteLine("List of Elements of SortedSet:");
            foreach (int item in integerSortedSet)
            {
                Console.WriteLine($"{item} ");
            }
            Console.ReadKey();
        }
    }
}

```

```

    }
}
}

```

using System;
using System.Collections.Generic;
/*The Generic Queue<T> Collection Class in C# is used to Enqueue and Dequeue elements in FIFO (First In First Out) order.
* The Enqueue operation adds an element in a collection, whereas the
* Dequeue operation is used to remove the firstly added element from the queue collection.
* It can have duplicate elements. The Queue<T> Collection Class belongs to System.Collections.Generic namespace.*/

```

namespace Collection4
{
    class Program
    {
        public static void Main(string[] args)
        {
            Queue<string> countriesQueue = new Queue<string>();
            countriesQueue.Enqueue("India");
            countriesQueue.Enqueue("USA");
            countriesQueue.Enqueue("UK");
            countriesQueue.Enqueue("China");
            countriesQueue.Enqueue("Nepal");
            Console.WriteLine("Queue Elements: ");
            foreach (string country in countriesQueue)
            {
                Console.Write($"{country} ");
            }
            Console.WriteLine("\n\nPeek Element: " + countriesQueue.Peek());
            Console.WriteLine("Element Removed: " + countriesQueue.Dequeue());
            Console.WriteLine("\nQueue Elements: ");
            foreach (string country in countriesQueue)
            {
                Console.Write($"{country} ");
            }
            Console.ReadKey();
        }
    }
}

```

using System;
using System.Collections.Generic;
/*The Generic Stack<T> Collection Class in C# is used to push and pop elements in LIFO (Last in First Out) order.
* The push operation adds an element to a collection, whereas the pop operation is used to remove the most recently added
* element from a collection.
* It can have duplicate elements. */

namespace CollectionStack


```

{
class Program
{
    public static void Main(string[] args)
    {
        Stack<string> countriesStack = new Stack<string>();
        countriesStack.Push("India");
        countriesStack.Push("India");
        countriesStack.Push("USA");
        countriesStack.Push("UK");
        countriesStack.Push("China");
        countriesStack.Push("Nepal");
        Console.WriteLine("Stack Elements: ");
        foreach (string country in countriesStack)
        {
            Console.Write($"{country} ");
        }
        Console.WriteLine("\n\nPeek Element: " + countriesStack.Peek());
        Console.WriteLine("Element Popped: " + countriesStack.Pop());
        Console.WriteLine("\nStack Elements: ");
        foreach (string country in countriesStack)
        {
            Console.Write($"{country} ");
        }
        Console.ReadKey();
    }
}
}

```

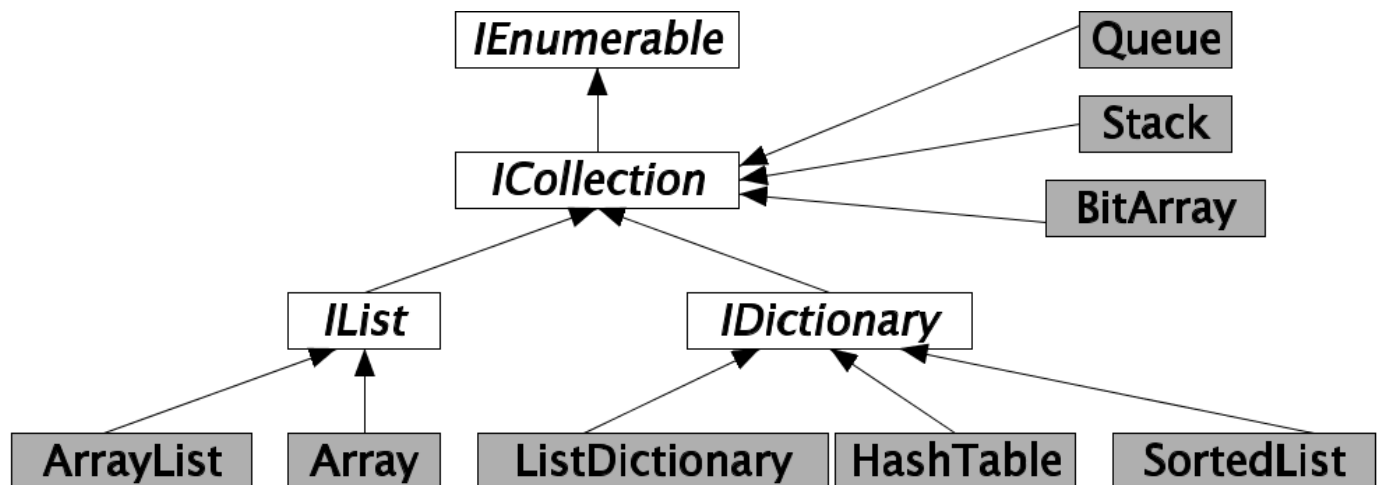
Question:

- In C# Collections, how does the performance of a List compare to a Linked List when it comes to insertion and deletion operations in the middle of the collection? Explain why.
- Explain the differences and use cases between IEnumerable, ICollection, and IList interfaces in C# Collections.
- How can you implement a custom IComparer to be used with C# Collection classes that can have configurable sorting behavior during runtime?
- What are the advantages of using a System.Collections.Concurrent namespace over the regular System.Collections.Generic namespace for collections, especially in multithreading scenarios?
- Explain the concept of a “bucket” in the context of the Dictionary class in C# Collections and how it affects the performance during hash collisions.
- How does the GetHashCode method affect the behavior of HashSet and Dictionary in C# Collections?
- Explain the importance of immutability in relation to C# Collection classes, and provide an example of a scenario where an immutable collection would be beneficial.
- Discuss the performance differences between Array, ArrayList, and List when it comes to adding, deleting, inserting, and looping through elements in C# Collections.
- What are potential issues that can arise when using default comparison methods for complex types in C# Collections, and how would you address them?

- Discuss the differences between a Stack and a Queue in C# Collections, and provide a real-world example of when each type would be most suitable.
- What are the differences between IEnumerable, ICollection, and IList interfaces in C#?

Chapter 27: C# -Generics

Generics allow you to define the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.



You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type. A simple example would help understanding the concept –

```

using System;
using System.Collections.Generic;

namespace GenericApplication {

    public class MyGenericArray<T> {
        private T[] array;

        public MyGenericArray(int size) {
            array = new T[size + 1];
        }

        public T getItem(int index) {
            return array[index];
        }
    }
  
```

```

    public void setItem(int index, T value) {
        array[index] = value;
    }
}

class Tester {
    static void Main(string[] args) {

        //declaring an int array
        MyGenericArray<int> intArray = new MyGenericArray<int>(5);

        //setting values
        for (int c = 0; c < 5; c++) {
            intArray.setItem(c, c*5);
        }

        //retrieving the values
        for (int c = 0; c < 5; c++) {
            Console.Write(intArray.getItem(c) + " ");
        }

        Console.WriteLine();

        //declaring a character array
        MyGenericArray<char> charArray = new MyGenericArray<char>(5);

        //setting values
        for (int c = 0; c < 5; c++) {
            charArray.setItem(c, (char)(c+97));
        }

        //retrieving the values
        for (int c = 0; c < 5; c++) {
            Console.Write(charArray.getItem(c) + " ");
        }
        Console.WriteLine();

        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

0 5 10 15 20

Features of Generics

Generics is a technique that enriches your programs in the following ways –

- ✓ It helps you to maximize code reuse, type safety, and performance.
- ✓ You can create generic collection classes. The .NET Framework class library contains several new generic collection classes in the *System.Collections.Generic* namespace. You may use these generic collection classes instead of the collection classes in the *System.Collections* namespace.
- ✓ You can create your own generic interfaces, classes, methods, events, and delegates.
- ✓ You may create generic classes constrained to enable access to methods on particular data types.
- ✓ You may get information on the types used in a generic data type at run-time by means of reflection.

Advantages of generic collections

- ✓ No boxing and type casts are required, thereby improves the performance of the application.
- ✓ Type errors are detected at compile-time, thus avoids runtime errors.
- ✓ Generic code once is written can be reused by instantiating the class with a specific type.

Generic Methods

In the previous example, we have used a generic class; we can declare a generic method with a type parameter. The following program illustrates the concept –

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

namespace GenericMethodAppl {

    class Program {

        static void Swap<T>(ref T lhs, ref T rhs) {
            T temp;
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }

        static void Main(string[] args) {
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'l';
```

```

    d = 'V';

    //display values before swap:
    Console.WriteLine("Int values before calling swap:");
    Console.WriteLine("a = {0}, b = {1}", a, b);
    Console.WriteLine("Char values before calling swap:");
    Console.WriteLine("c = {0}, d = {1}", c, d);

    //call swap
    Swap<int>(ref a, ref b);
    Swap<char>(ref c, ref d);

    //display values after swap:
    Console.WriteLine("Int values after calling swap:");
    Console.WriteLine("a = {0}, b = {1}", a, b);
    Console.WriteLine("Char values after calling swap:");
    Console.WriteLine("c = {0}, d = {1}", c, d);

    Console.ReadKey();
}
}
}

```

When the above code is compiled and executed, it produces the following result –

```

Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I

```

Introduction to Using Generic Collections with an Example of the List<T> and Dictionary<TKey, TValue> Generic Collections

```

using System;
using System.Collections.Generic;

public class Customer
{
    public Customer(int id, string name)
    {
        ID = id;
        Name = name;
    }
}

```

```

private int m_id;

public int ID
{
    get { return m_id; }
    set { m_id = value; }
}

private string m_name;

public string Name
{
    get { return m_name; }
    set { m_name = value; }
}
}

class Program
{
    static void Main(string[] args)
    {
        List<int> myInts = new List<int>();

        myInts.Add(1);
        myInts.Add(2);
        myInts.Add(3);

        for (int i = 0; i < myInts.Count; i++)
        {
            Console.WriteLine("MyInts: {0}", myInts[i]);
        }

        Dictionary<int, Customer> customers = new Dictionary<int, Customer>();

        Customer cust1 = new Customer(1, "Cust 1");
        Customer cust2 = new Customer(2, "Cust 2");
        Customer cust3 = new Customer(3, "Cust 3");

        customers.Add(cust1.ID, cust1);
        customers.Add(cust2.ID, cust2);
        customers.Add(cust3.ID, cust3);

        foreach (KeyValuePair<int, Customer> custKeyVal in customers)
        {
            Console.WriteLine(
                "Customer ID: {0}, Name: {1}",
                custKeyVal.Key,
                custKeyVal.Value.Name);
        }

        Console.ReadKey();
    }
}

```

Generic Delegates

You can define a generic delegate with type parameters. For example –

```
delegate T NumberChanger<T>(T n);
```

The following example shows use of this delegate –

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

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl {

    class TestDelegate {
        static int num = 10;

        public static int AddNum(int p) {
            num += p;
            return num;
        }

        public static int MultNum(int q) {
            num *= q;
            return num;
        }

        public static int getNum() {
            return num;
        }

        static void Main(string[] args) {
            //create delegate instances
            NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
            NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);

            //calling the methods using the delegate objects
            nc1(25);
            Console.WriteLine("Value of Num: {0}", getNum());
            nc2(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

Value of Num: 35

Value of Num: 175

Anonymous Methods in C#

We discussed that delegates are used to reference any methods that has the same signature as that of the delegate. In other words, you can call a method that can be referenced by a delegate using that delegate object.

Anonymous methods provide a technique to pass a code block as a delegate parameter. Anonymous methods are the methods without a name, just the body.

You need not specify the return type in an anonymous method; it is inferred from the return statement inside the method body.

Writing an Anonymous Method

Anonymous methods are declared with the creation of the delegate instance, with a **delegate** keyword. For example,

```
delegate void NumberChanger(int n);  
...  
NumberChanger nc = delegate(int x) {  
    Console.WriteLine("Anonymous Method: {0}", x);  
};
```

The code block *Console.WriteLine("Anonymous Method: {0}", x);* is the body of the anonymous method.

The delegate could be called both with anonymous methods as well as named methods in the same way, i.e., by passing the method parameters to the delegate object.

For example,

```
nc(10);
```

Example

The following example demonstrates the concept –

```
using System;  
  
delegate void NumberChanger(int n);  
namespace DelegateAppl {
```



```

class TestDelegate {
    static int num = 10;

    public static void AddNum(int p) {
        num += p;
        Console.WriteLine("Named Method: {0}", num);
    }

    public static void MultNum(int q) {
        num *= q;
        Console.WriteLine("Named Method: {0}", num);
    }

    public static int getNum() {
        return num;
    }

    static void Main(string[] args) {
        //create delegate instances using anonymous method
        NumberChanger nc = delegate(int x) {
            Console.WriteLine("Anonymous Method: {0}", x);
        };

        //calling the delegate using the anonymous method
        nc(10);

        //instantiating the delegate using the named methods
        nc = new NumberChanger(AddNum);

        //calling the delegate using the named methods
        nc(5);

        //instantiating the delegate using another named methods
        nc = new NumberChanger(MultNum);

        //calling the delegate using the named methods
        nc(2);
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

Anonymous Method: 10

Named Method: 15

Named Method: 30

```

using System.Windows.Forms;

public partial class Form1 : Form
{
    public Form1()
    {
        Button btnHello = new Button();
        btnHello.Text = "Hello";

        btnHello.Click +=
            delegate
            {
                MessageBox.Show("Hello");
            };

        Controls.Add(btnHello);
    }
}

```

Using Parameters with Anonymous Methods

```

using System;
using System.Windows.Forms;

public partial class Form1 : Form
{
    public Form1()
    {
        Button btnHello = new Button();
        btnHello.Text = "Hello";

        btnHello.Click +=
            delegate
            {
                MessageBox.Show("Hello");
            };

        Button btnGoodBye = new Button();
        btnGoodBye.Text = "Goodbye";
        btnGoodBye.Left = btnHello.Width + 5;
        btnGoodBye.Click +=
            delegate(object sender, EventArgs e)
            {
                string message = (sender as Button).Text;
                MessageBox.Show(message);
            };

        Controls.Add(btnHello);
        Controls.Add(btnGoodBye);
    }
}

```

Chapter 28: C#- Unsafe Codes

C# allows using pointer variables in a function or code block when it is marked by the **unsafe** modifier. The **unsafe code** or the unmanaged code is a code block that uses a **pointer** variable.

Note – To execute the programs mentioned in this chapter at [codingground](#), please set compilation option in Project >> Compile Options >> Compilation Command to
`mcs *.cs -out:main.exe -unsafe`

Pointers

A **pointer** is a variable whose value is the address of another variable i.e., the direct address of the memory location. similar to any variable or constant, you must declare a pointer before you can use it to store any variable address.

The general form of a pointer declaration is –

```
type *var-name;
```

Following are valid pointer declarations –

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The following example illustrates use of pointers in C#, using the unsafe modifier –

```
using System;

namespace UnsafeCodeApplication {

    class Program {

        static unsafe void Main(string[] args) {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ", var);
            Console.WriteLine("Address is: {0}", (int)p);
            Console.ReadKey();
        }
    }
}
```

When the above code was compiled and executed, it produces the following result –

Data is: 20

Address is: 99215364

Instead of declaring an entire method as unsafe, you can also declare a part of the code as unsafe. The example in the following section shows this.

Retrieving the Data Value Using a Pointer

You can retrieve the data stored at the located referenced by the pointer variable, using the **ToString()** method. The following example demonstrates this –

```
using System;

namespace UnsafeCodeApplication {

    class Program {

        public static void Main() {

            unsafe {
                int var = 20;
                int* p = &var;
                Console.WriteLine("Data is: {0} ", var);
                Console.WriteLine("Data is: {0} ", p->ToString());
                Console.WriteLine("Address is: {0} ", (int)p);
            }

            Console.ReadKey();
        }
    }
}
```

When the above code was compiled and executed, it produces the following result –

Data is: 20

Data is: 20

Address is: 77128984

Passing Pointers as Parameters to Methods

You can pass a pointer variable to a method as parameter. The following example illustrates this –

```
using System;

namespace UnsafeCodeApplication {
```

```

class TestPointer {

    public unsafe void swap(int* p, int *q) {
        int temp = *p;
        *p = *q;
        *q = temp;
    }

    public unsafe static void Main() {
        TestPointer p = new TestPointer();
        int var1 = 10;
        int var2 = 20;
        int* x = &var1;
        int* y = &var2;

        Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
        p.swap(x, y);

        Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

Before Swap: var1: 10, var2: 20

After Swap: var1: 20, var2: 10

Accessing Array Elements Using a Pointer

In C#, an array name and a pointer to a data type same as the array data, are not the same variable type. For example, `int *p` and `int[] p`, are not same type. You can increment the pointer variable `p` because it is not fixed in memory but an array address is fixed in memory, and you can't increment that.

Therefore, if you need to access an array data using a pointer variable, as we traditionally do in C, or C++ (please check: [C Pointers](#)), you need to fix the pointer using the **fixed** keyword.

The following example demonstrates this –

```

using System;

namespace UnsafeCodeApplication {

    class TestPointer {

        public unsafe static void Main() {

```

```

int[] list = {10, 100, 200};
fixed(int *ptr = list)

/* let us have array address in pointer */
for ( int i = 0; i < 3; i++) {
    Console.WriteLine("Address of list[{0}]= {1}", i, (int)(ptr + i));
    Console.WriteLine("Value of list[{0}]= {1}", i, *(ptr + i));
}

Console.ReadKey();
}
}
}

```

When the above code was compiled and executed, it produces the following result –

```

Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200

```

Compiling Unsafe Code

For compiling unsafe code, you have to specify the **/unsafe** command-line switch with command-line compiler.

For example, to compile a program named prog1.cs containing unsafe code, from command line, give the command –

```
csc /unsafe prog1.cs
```

If you are using Visual Studio IDE then you need to enable use of unsafe code in the project properties.

To do this –

- Open **project properties** by double clicking the properties node in the Solution Explorer.
- Click on the **Build** tab.
- Select the option "**Allow unsafe code**".

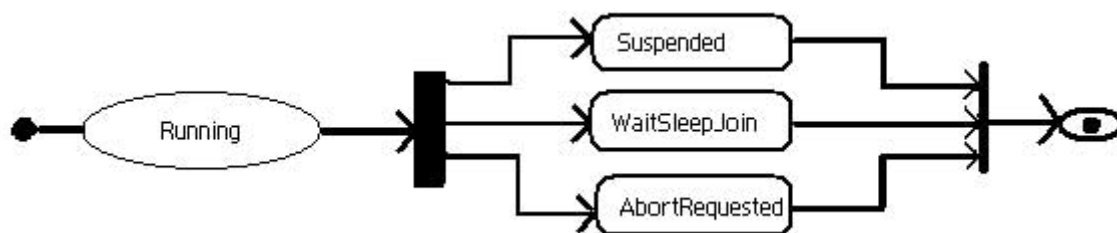
Chapter 29: C#- Multithreading

A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations, then it is

often helpful to set different execution paths or threads, with each thread performing a particular job.



Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.



So far we wrote the programs where a single thread runs as a single process which is the running instance of the application. However, this way the application can perform one job at a time. To make it execute more than one task at a time, it could be divided into smaller threads.

Thread Life Cycle

The life cycle of a thread starts when an object of the `System.Threading.Thread` class is created and ends when the thread is terminated or completes execution.

Following are the various states in the life cycle of a thread –

- **The Unstarted State** – It is the situation when the instance of the thread is created but the `Start` method is not called.
- **The Ready State** – It is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State** – A thread is not executable, when
 - `Sleep` method has been called
 - `Wait` method has been called
 - Blocked by I/O operations

- **The Dead State** – It is the situation when the thread completes execution or is aborted.

The Main Thread

In C#, the **System.Threading.Thread** class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the **main** thread.

When a C# program starts execution, the main thread is automatically created. The threads created using the **Thread** class are called the child threads of the main thread. You can access a thread using the **CurrentThread** property of the Thread class.

The following program demonstrates main thread execution –

```
using System;
using System.Threading;

namespace MultithreadingApplication {

    class MainThreadProgram {

        static void Main(string[] args) {
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";
            Console.WriteLine("This is {0}", th.Name);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

This is MainThread

Properties and Methods of the Thread Class

The following table shows some most commonly used **properties** of the **Thread** class –

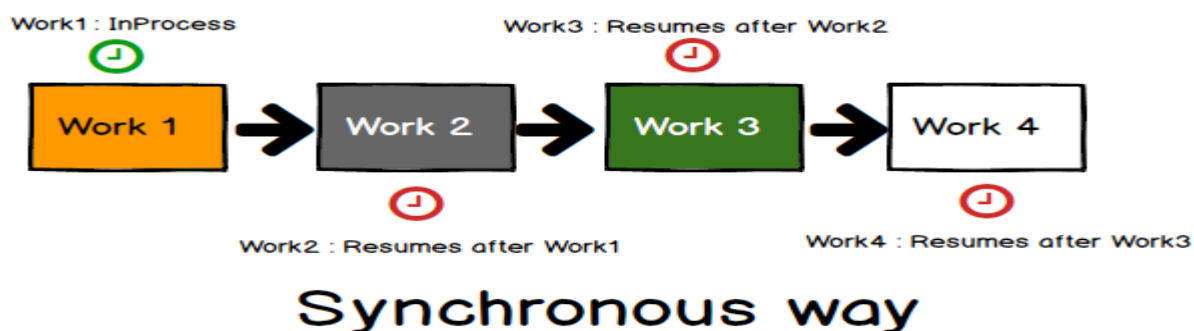
Sr.No.	Property & Description
1	CurrentContext Gets the current context in which the thread is executing.
2	CurrentCulture Gets or sets the culture for the current thread.
3	CurrentPrinciple

	Gets or sets the thread's current principal (for role-based security).
4	CurrentThread Gets the currently running thread.
5	CurrentUICulture Gets or sets the current culture used by the Resource Manager to look up culture-specific resources at run-time.
6	ExecutionContext Gets an ExecutionContext object that contains information about the various contexts of the current thread.
7	IsAlive Gets a value indicating the execution status of the current thread.
8	IsBackground Gets or sets a value indicating whether or not a thread is a background thread.
9	IsThreadPoolThread Gets a value indicating whether or not a thread belongs to the managed thread pool.
10	ManagedThreadId Gets a unique identifier for the current managed thread.
11	Name Gets or sets the name of the thread.
12	Priority Gets or sets a value indicating the scheduling priority of a thread.
	ThreadState Gets a value containing the states of the current thread? <div data-bbox="221 1408 1246 2016" data-label="Diagram"> <pre> graph TD Unstarted -- Start --> Running Running -- "Thread Blocks" --> WaitSleepJoin WaitSleepJoin -- "Thread Unblocks" --> Running Running -- "Thread Ends" --> Stopped Running -- "Abort" --> AbortRequested AbortRequested -- "ResetAbort" --> Running AbortRequested -- "Thread Ends" --> Stopped WaitSleepJoin -- "Abort" --> AbortRequested AbortRequested -. "in theory only!" .-> Aborted </pre> <p>The diagram illustrates the states of a thread and the transitions between them. The states are represented by rounded rectangles: Unstarted, Running, WaitSleepJoin, Stopped, AbortRequested, and Aborted. Transitions are represented by arrows with labels: 'Start' from Unstarted to Running; 'Thread Blocks' from Running to WaitSleepJoin; 'Thread Unblocks' from WaitSleepJoin to Running; 'Thread Ends' from Running to Stopped; 'Abort' from Running to AbortRequested; 'ResetAbort' from AbortRequested back to Running; 'Thread Ends' from AbortRequested to Stopped; 'Abort' from WaitSleepJoin to AbortRequested; and a dashed arrow labeled 'in theory only!' from AbortRequested to Aborted.</p> </div>

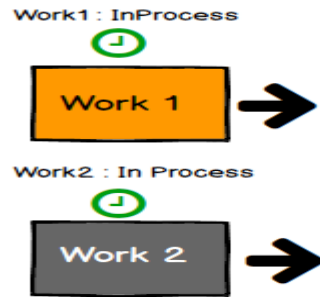
The following table shows some of the most commonly used **methods** of the **Thread** class –

STATE	DESCRIPTION
A thread is created	Unstarted
Another thread calls the Thread. Start method on the new thread, and the call returns. The Start method does not return until the new thread has started running. There is no way to know at what point the new thread will start running, during the call to Start.	Running
The thread calls Sleep	WaitSleepJoin
The thread calls Wait on another object.	WaitSleepJoin
The thread calls Join on another thread.	WaitSleepJoin
Another thread calls Interrupt	Running
Another thread calls Suspend	SuspendRequested
The thread responds to a Suspend request.	Suspended
Another thread calls Resume	Running
Another thread calls Abort	AbortRequested
The thread responds to a Abort request.	Stopped
A thread is terminated.	Stopped
The thread state includes AbortRequested and the thread is now dead, but its state has not yet changed to Stopped.	Aborted
If thread is a background thread	Background

Synchronous way means where work multiple jobs are executed one after the other. Here Work 2 have to wait till Work 1 is completed same way the others as shown in below image



Asynchronous means multiple work has been executed simultaneously like doing multitask at a same time.



Asynchronous way

Synchronization in C# is a mechanism that makes sure only one process or thread accesses the critical section of the program. All the other threads have to wait until the critical section is free before they can enter it.

Some of the advantages of thread synchronization is given as follows:

- ✓ Synchronization can be used to achieve mutual exclusion i.e. only one process or thread accesses the critical section of the program.
- ✓ No thread or process has an infinite waiting time using synchronization.
- ✓ Synchronization is an inherently fair process.
- ✓ All the requests are granted in a specific order using synchronization.
- ✓ Synchronization can be used for resource allocation as well. It makes sure that only one process or thread can use a resource at a time.

Join

1. In thread synchronization, join is a blocking mechanism that pauses the calling thread. This is done till the thread whose join method was called has completed its execution.
2. A program that demonstrates the join method is given as follows:

```
using System;
using System.Threading;
namespace JoinDemo
{
    class Example
    {
        static void Main(string[] args)
        {
            Thread t1 = new Thread(Func1);
            t1.Start();
            Thread t2 = new Thread(Func2);
            t2.Start();
            t1.Join();
        }
    }
}
```

```

        t2.Join();
    }
    private static void Func2(object obj)
    {
        Console.WriteLine("Thread1 is executed");
    }
    private static void Func1(object obj)
    {
        Console.WriteLine("Thread2 is executed");
    }
}

```

Lock is a synchronization method that is used to lock in the current thread so that no other thread can interrupt the execution of the locked thread. After the thread execution is complete, it is unlocked.

A program that demonstrates the lock method is given as follows:
using System;

```

using System.Threading;
namespace LockDemo
{
    class LockDisplay
    {
        public void DisplayNum()
        {
            lock (this)
            {
                for (int i = 1; i <= 5; i++)
                {
                    Thread.Sleep(100);
                    Console.WriteLine("i = {0}", i);
                }
            }
        }
    }
}
class Example
{
    public static void Main(string[] args)
    {
        LockDisplay obj = new LockDisplay();

        Console.WriteLine("Threading using Lock");

        Thread t1 = new Thread(new ThreadStart(obj.DisplayNum));
        Thread t2 = new Thread(new ThreadStart(obj.DisplayNum));
        t1.Start();
    }
}

```

```
        t2.Start();
    }
}
```

Creating Threads

Threads are created by extending the Thread class. The extended Thread class then calls the **Start()** method to begin the child thread execution.

The following program demonstrates the concept –

```
using System;
using System.Threading;

namespace MultithreadingApplication {

    class ThreadCreationProgram {

        public static void CallToChildThread() {
            Console.WriteLine("Child thread starts");
        }

        static void Main(string[] args) {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

```
In Main: Creating the Child thread
Child thread starts
```

Managing Threads

The Thread class provides various methods for managing threads.

The following example demonstrates the use of the **sleep()** method for making a thread pause for a specific period of time.

```
using System;
using System.Threading;

namespace MultithreadingApplication {

    class ThreadCreationProgram {

        public static void CallToChildThread() {
            Console.WriteLine("Child thread starts");

            // the thread is paused for 5000 milliseconds
            int sleepfor = 5000;

            Console.WriteLine("Child Thread Paused for {0} seconds", sleepfor / 1000);
            Thread.Sleep(sleepfor);
            Console.WriteLine("Child thread resumes");
        }

        static void Main(string[] args) {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

Destroying Threads

The **Abort()** method is used for destroying threads.

The runtime aborts the thread by throwing a **ThreadAbortException**. This exception cannot be caught, the control is sent to the *finally* block, if any.

The following program illustrates this –

```
using System;
using System.Threading;
```

```

namespace MultithreadingApplication {

    class ThreadCreationProgram {

        public static void CallToChildThread() {

            try {
                Console.WriteLine("Child thread starts");

                // do some work, like counting to 10
                for (int counter = 0; counter <= 10; counter++) {
                    Thread.Sleep(500);
                    Console.WriteLine(counter);
                }

                Console.WriteLine("Child Thread Completed");
            } catch (ThreadAbortException e) {
                Console.WriteLine("Thread Abort Exception");
            } finally {
                Console.WriteLine("Couldn't catch the Thread Exception");
            }
        }

        static void Main(string[] args) {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();

            //stop the main thread for some time
            Thread.Sleep(2000);

            //now abort the child
            Console.WriteLine("In Main: Aborting the Child thread");

            childThread.Abort();
            Console.ReadKey();
        }
    }
}

```

When the above code is compiled and executed, it produces the following result –

In Main: Creating the Child thread

Child thread starts

0

1

In Main: Aborting the Child thread

Thread Abort Exception

Couldn't catch the Thread Exception

Chapter 30: C#- DLL

Dynamic-Link Library (DLL) is a shared library concept that was introduced in the Microsoft Windows Operating System. **DLLs** are collections of code, data, and resources that can be used by multiple applications simultaneously. They offer several advantages over static libraries, such as reduced memory consumption, faster startup time, and easier maintenance. In this article, we will see what a **DLL** is in C# and how it can be used to build modular and extensible applications.

What is a DLL?

A **DLL** is a binary file that has code and data that can be utilized by multiple applications at the same time. The **DLL** is loaded into the memory of each application that uses it, and the code and data can be accessed by those applications as if they were part of the application's code. This makes **DLLs** a powerful tool for building modular and extensible applications.

A **DLL** can contain any type of code or data that can be used by an application, including functions, classes, variables, and resources. When an application needs to use a **DLL**, it loads the **DLL** into memory and calls the functions or uses the data it contains. Once the application is done with the **DLL**, it can unload it from memory.

In C#, a **DLL** is a compiled assembly that contains **.NET** Framework code. It is created by compiling one or more C# source files into a **DLL** file. The **DLL** file can then be referenced by other C# projects, allowing them to use the code and data that it contains.

Advantages of Using DLLs in C#:

There are several advantages to using **DLLs** in C#:

Reusability:

DLLs allow code to be shared between multiple applications. This can help reduce development time and improve code maintainability.

Modularity:

DLLs allow code to be organized into separate modules, each of which can be loaded and unloaded independently. This can help reduce memory usage and improve application startup time.

Extensibility:

DLLs can be used to add functionality to an application without modifying its existing code. This can be useful for adding plugins or extensions to an application.

Versioning:

DLLs can be versioned, which allows different versions of a **DLL** to be used by different applications. This can help prevent compatibility issues between applications that use different versions of the same **DLL**.

Creating a DLL in C#:

Creating a **DLL** in C# is a straightforward process. Here are the steps involved:

1. Create a new C# project in Visual Studio.
2. Add the code that you want to include in the **DLL** to the project.
3. Build the project to create a **DLL**
4. Reference the **DLL** file from other C# projects that need to use the code that it contains.

Let's walk through these steps in more detail.

Step 1: Create a New C# project

To create a new C# project in Visual Studio, follow these steps:

- Open Visual Studio.
- From the File menu, select New > Project.
- In the New Project dialog box, select C# from the list of project types.
- Select Class Library from the list of the given templates.
- Choose a name and location for your project, and then click Create.

Step 2: Add the Code to the Project

Once you have created the project, you can add the code that you want to include in the **DLL**. This can include functions, classes, variables, and resources. Here is an example of a simple function that can be included in a DLL:

C# Code:

```
namespace MyLibrary
{
    public class MyFunctions
    {
        public static int Add(int a, int b)
        {
            return a + b;
        }
    }
}
```

Step 3: Build the project

Once you have added the code to the project, you can build the project to create a **DLL** file. To do this, follow these steps:

- From the Build menu, select Build Solution.
- Visual Studio will compile the code and create a **DLL** file in the output folder for the project. By default, this folder is located in the bin\Debug or bin\Release folder of the project directory, depending on the build configuration.

Step 4: Reference the DLL From Other C# projects

To use the code that you have included in the **DLL** from other C# projects, you need to reference the **DLL** file. To do this, follow these steps:

- Open the project that needs to use the code from the **DLL**.
- From the Solution Explorer, right-click on the project name and select Add Reference.
- In the Reference Manager dialog box, select Browse and navigate to the location of the **DLL**
- Select the **DLL** file and click Add.

The **DLL** file should now be listed in the References section of the project. You can now use the code that it contains in your project.

Using a DLL in C#:

Once you have created a **DLL** and referenced it from another C# project, you can use the code that it contains in your project. Below is an example of how to use the Add function that we created earlier:

C# Code:

```
using MyLibrary;  
  
int result = MyFunctions.Add(2, 3);  
  
Console.WriteLine(result); // Output: 5
```

DLLs are a powerful tool for building modular and extensible applications in C#. They allow code to be shared between multiple applications, organized into separate modules, and added to an application without modifying its existing code. Creating a **DLL** in C# is a straightforward process that involves creating a new project, adding code to the project, building the project, and referencing the **DLL** file from other C# projects. Once a **DLL** has been created and referenced, its code and data can be accessed by other applications as if they were part of the application's own code.

Chapter 31: C#- Dependency Injection

Dependency Injection (DI) is a **design pattern** used in software development that helps manage the dependencies between objects or classes. The main idea of DI is to **inject (provide)** the required dependencies to a class or object from the outside, rather than letting the class create them internally.

In C#, Dependency Injection is typically implemented using a framework like **Microsoft's Dependency Injection framework**. Following an example of how Dependency Injection can be used in C#:

Implementation of Dependency Injection Pattern in C#

Suppose you have a class called **UserService** that needs to send emails to users when certain events occur. In order to send emails, UserService needs an instance of a **MailService class**. Without DI, UserService might create an instance of MailService internally, like the following:

```
public class UserService
{
    private MailService mailService = new MailService();

    public void SendEmailToUser(User user, string subject, string body)
    {
        mailService.SendEmail(user.EmailAddress, subject, body);
    }
}
```

However, this approach has some drawbacks. First, it creates a **tight coupling** between UserService and MailService. If you ever want to switch to a different email service, you will have to modify the UserService class. Second, it's hard to test UserService in **isolation**, since you can't easily substitute a different implementation of MailService.

With **Dependency Injection**, you can inject an instance of MailService into UserService from the outside. following code shows how that might look:

```
public class UserService
{
    private readonly IMailService mailService;
    public UserService(IMailService mailService)
    {
        this.mailService = mailService;
    }
    public void SendEmailToUser(User user, string subject, string body)
    {
        mailService.SendEmail(user.EmailAddress, subject, body);
    }
}
```

In the above version of the UserService class, the MailService instance is **injected via the constructor**, and it's stored in a private field called mailService. Note that instead of MailService, you are using an interface called IMailService. This is a common practice in **Dependency Injection**, since it makes it easy to substitute different implementations of IMailService as needed.

To use UserService, you need to **create an instance** of MailService, and then pass it to the UserService constructor:

```
IMailService mailService = new MailService();

UserService userService = new UserService(mailService);
```

By using Dependency Injection, you can have decoupled UserService from MailService, which makes it **easier to maintain** and test our code. You can easily swap out MailService with a different email service, or even a mock implementation of IMailService for testing purposes.

Tight Coupling and Loose Coupling

In software design, coupling refers to the **degree of interdependence** between different modules or components of a software system. Tight coupling means that modules or components are **highly interdependent**, while loose coupling means that they are less interdependent.

Tight coupling can make a software system inflexible and **difficult to maintain**, since changes in one module may require changes in many other modules. On the other hand, loose coupling can make a system more flexible and **easier to maintain**, since changes in one module may have less of an impact on other modules.

Following are some examples to illustrate the difference between **tight coupling and loose coupling**:

Tight coupling:

A class that creates an instance of another class internally and calls its methods directly, like the following:

```
public class OrderProcessor {
    private PaymentProcessor paymentProcessor = new PaymentProcessor();
    public void Process(Order order) {
        paymentProcessor.Charge(order.TotalAmount);
        // other code to process the order
    }
}
```

In the above example, the OrderProcessor class is **tightly coupled** to the PaymentProcessor class, since it creates an instance of PaymentProcessor internally and calls its Charge method directly. This makes it difficult to substitute a different payment processor implementation or test the OrderProcessor class in isolation.

Loose coupling:

A class that depends on an **interface** instead of a concrete implementation, and the implementation is passed to the class externally, like the following:

```

public interface IPaymentProcessor {

    void Charge(decimal amount);

}

public class OrderProcessor {

    private readonly IPaymentProcessor paymentProcessor;

    public OrderProcessor(IPaymentProcessor paymentProcessor) {

        this.paymentProcessor = paymentProcessor;

    }

    public void Process(Order order) {

        paymentProcessor.Charge(order.TotalAmount);

        // other code to process the order

    }

}

```

In the above example, the OrderProcessor class is loosely coupled to the payment processor implementation, since it depends on the IPaymentProcessor **interface** instead of a concrete implementation. The implementation is passed to the OrderProcessor class via its constructor, so you can easily substitute a **different implementation** or use a mock implementation for testing purposes.

Types of Dependency Injection (DI)

In C#, there are three common types of Dependency Injection (DI) design patterns:

- Constructor Injection
- Property Injection
- Method Injection

Constructor Injection:

Constructor Injection is the most commonly used **DI pattern in C#**. In Constructor Injection, dependencies are passed to a class through its constructor.

```

public class MyService {
    private readonly IDependency _dependency;
    public MyService(IDependency dependency) {
        _dependency = dependency;
    }
}

```

In the above example, the MyService class takes an instance of **IDependency** as a parameter in its constructor. The dependency is then stored in a private field for later use.

Property Injection:

Property Injection is another type of **DI pattern in C#**. In Property Injection, dependencies are passed to a class through its public properties.

```
public class MyService {  
  
    public IDependency Dependency { get; set; }  
  
}
```

In the above example, the `MyService` class has a public property called `Dependency` of type **IDependency**. The dependency is set by an external entity, typically a DI container, after an instance of `MyService` is created.

Method Injection:

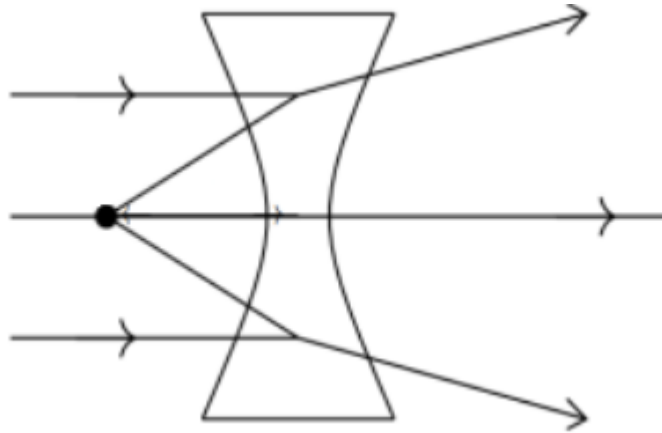
Method Injection is the least common DI pattern in C#. In Method Injection, dependencies are passed to a class through its public methods.

```
public class MyService {  
    public void DoSomething(IDependency dependency) {  
        // use dependency to do something  
    }  
}
```

In the above example, the `MyService` class has a public method called `DoSomething` that takes an instance of **IDependency** as a parameter. The dependency is passed to the method by an external entity, typically a DI container.

It's worth noting that **Constructor Injection** is generally considered the most flexible and maintainable DI pattern, since it allows dependencies to be provided in a consistent manner and makes them immediately available for use. Property Injection and **Method Injection** can be useful in certain scenarios, but they can also make code more difficult to reason about and test, since dependencies may not be immediately available or may change over time.

Constructor Injection in C#



Constructor injection is a type of **dependency injection**, which is a design pattern used to invert the control of object creation and management. Instead of having an object create and manage its own dependencies, the dependencies are injected into the object from the outside.

Constructor injection has a number of benefits, including:

- It makes the code more testable, because dependencies can be easily mocked or substituted for testing purposes.
- It simplifies object creation, because the object's dependencies are clearly defined in its constructor.
- It makes the code more modular, because objects can be easily composed from smaller, reusable dependencies.

Working example of C# Constructor Injection:

```
namespace EmailServiceExample
{
    public interface ILogger
    {
        void Log(string message);
    }
    public class ConsoleLogger : ILogger
    {
        public void Log(string message)
        {
            Console.WriteLine($"[INFO] {message}");
        }
    }
    public interface IConfiguration
    {
        string GetSetting(string key);
    }
    public class AppConfigConfiguration : IConfiguration
    {
        public string GetSetting(string key)
        {

```

```

    // implementation details here
    return "";
}
}
public class EmailService
{
    private readonly ILogger _logger;
    private readonly IConfiguration _configuration;
    public EmailService(ILogger logger, IConfiguration configuration)
    {
        _logger = logger;
        _configuration = configuration;
    }
    public void SendEmail(string toAddress, string subject, string body)
    {
        _logger.Log($"Sending email to {toAddress} with subject '{subject}' and body '{body}'");
        // implementation details here
    }
}
class Program
{
    static void Main(string[] args)
    {
        ILogger logger = new ConsoleLogger();
        IConfiguration configuration = new AppConfigConfiguration();
        EmailService emailService = new EmailService(logger, configuration);
        emailService.SendEmail("example@example.com", "Test Subject", "This is a test email");
        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    }
}
}

```

//Output

[INFO] Sending email to example@example.com with subject 'Test Subject' and body 'This is a test email'

Press any key to exit...

In the above example, EmailService has a constructor that takes two dependencies: an ILogger and an IConfiguration. These dependencies are **injected into the constructor** when an instance of EmailService is created.

In the Main method, create instances of ConsoleLogger, AppConfigConfiguration, and EmailService, passing in the **dependencies to the constructor** of EmailService. Then call the SendEmail method on EmailService and output the result to the console.

```

ILogger logger = new ConsoleLogger();
IConfiguration configuration = new AppConfigConfiguration();
EmailService emailService = new EmailService(logger, configuration);

```


These dependencies could be substituted for other implementations of ILogger and IConfiguration as needed.

Advantages of Dependency Injection

Dependency Injection (DI) is a **design pattern** used in software engineering to help reduce coupling between different modules of an application. Here are some advantages of Dependency Injection:

- **Loose coupling:** Dependency Injection promotes loose coupling between the different components of an application. It makes it easier to modify, test and maintain these components in isolation.
- **Testability:** One of the key benefits of DI is that it makes unit testing easier. By injecting dependencies into a component, it is possible to replace real dependencies with mock objects or test doubles. This allows developers to test components in isolation without having to worry about the behavior of other components.
- **Reusability:** DI promotes the reuse of components across different parts of an application. Since components are not tightly coupled, they can be reused in different contexts, which leads to a more modular and flexible codebase.
- **Encapsulation:** DI can help encapsulate the creation of dependencies in a single place, making it easier to manage them. This can also help to reduce the number of dependencies that need to be explicitly declared and passed around.
- **Flexibility:** By using DI, it is possible to change the behavior of a component by simply changing the dependencies that are injected into it. This makes the code more flexible and adaptable to changing requirements.

Dependency Injection can lead to a **more maintainable**, testable, and flexible codebase, which is easier to modify and extend over time

Chapter 32: C#- Networking

The Microsoft .NET framework offers two namespaces, namely System.Net and System.Net.Sockets, which provide a managed implementation of Internet protocols. These protocols enable applications to send and receive data over the Internet. Network programming in the Windows environment can be achieved using sockets, allowing peer-to-peer Microsoft Windows applications to act as both servers and clients for data exchange.

System.Net classes

The System.Net classes offer functionalities that are similar to the Microsoft WinInet API. They enable communication between classes and other applications using various Internet protocols such as Hypertext Transfer Protocol (HTTP), Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Socket. These protocols facilitate data transfer and communication between different systems. By exploring the chapters ahead, you can gain a deeper understanding of important classes and their practical applications in C# programming.

How to send email from C#

What is SMTP?

SMTP (Simple Mail Transfer Protocol) is a part of the application layer of the TCP/IP protocol. It is an Internet standard for electronic mail (email) transmission. The default TCP port used by SMTP is 25 and the SMTP connections secured by SSL, known as SMTPS, uses the default to port 465.

SMTP Servers



SMTP provides a set of protocol that simplify the communication of email messages between email servers. Most SMTP server names are written in the form "smtp.domain.com" or "mail.domain.com": for example, a Gmail account will refer to smtp.gmail.com. It is usually used with one of two other protocols, POP3 or IMAP, that let the user save messages in a server mailbox and download them periodically from the server.

SMTP Authentication

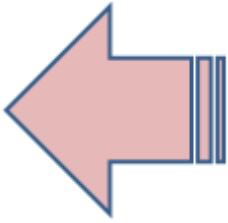
SMTP Authentication, often abbreviated SMTP AUTH, is an extension of the SMTP (Simple Mail Transfer Protocol) whereby an SMTP client may log in using an authentication mechanism chosen among those supported by the SMTP servers.

```
SmtpClient SmtpServer = new SmtpClient("smtp.gmail.com");  
  
SmtpServer.Port = 587;  
  
SmtpServer.Credentials =  
  
new System.Net.NetworkCredential("username", "password");
```

How do I send mail using C#?

The Microsoft .NET framework provides two namespaces, **System.Net** and **System.Net.Sockets** for managed implementation of Internet protocols that applications can use to send or receive data over the Internet. **SMTP protocol** is using for sending email from C#. SMTP stands for Simple Mail Transfer Protocol. C# using System.Net.Mail namespace for sending email. We can instantiate SmtpClient class and assign the Host and Port. The default port using SMTP is 25, but it may vary different Mail Servers.

Send Email using Gmail



The following C# source code shows how to send an email from a Gmail address using SMTP server. The Gmail SMTP server name is **smtp.gmail.com** and the port using send mail is 587 and also using NetworkCredential for password based authentication.

Full Source C#

```
using System;

using System.Windows.Forms;
using System.Net.Mail;
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            try
            {
                MailMessage mail = new MailMessage();
                SmtpClient SmtpServer = new SmtpClient("smtp.gmail.com");
                mail.From = new MailAddress("Enosis@123gamil.com");
                mail.To.Add("to_address");
                mail.Subject = "Test Mail";
                mail.Body = "This is for testing SMTP mail from GMAIL";
                SmtpServer.Port = 587;
                SmtpServer.Credentials = new System.Net.NetworkCredential("username", "password");
                SmtpServer.EnableSsl = true;
                SmtpServer.Send(mail);
                MessageBox.Show("mail Send");
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.ToString());
            }
        }
    }
}
```

Chapter 34: C#- Design Patterns

Design patterns provide general solutions or a flexible way to solve common design problems. This article introduces design patterns and how design patterns are implemented in C# and .NET.

Before starting with design patterns in .NET, let's understand the meaning of design patterns and why they are useful in software architecture and programming.

What are Design Patterns in Software Development?

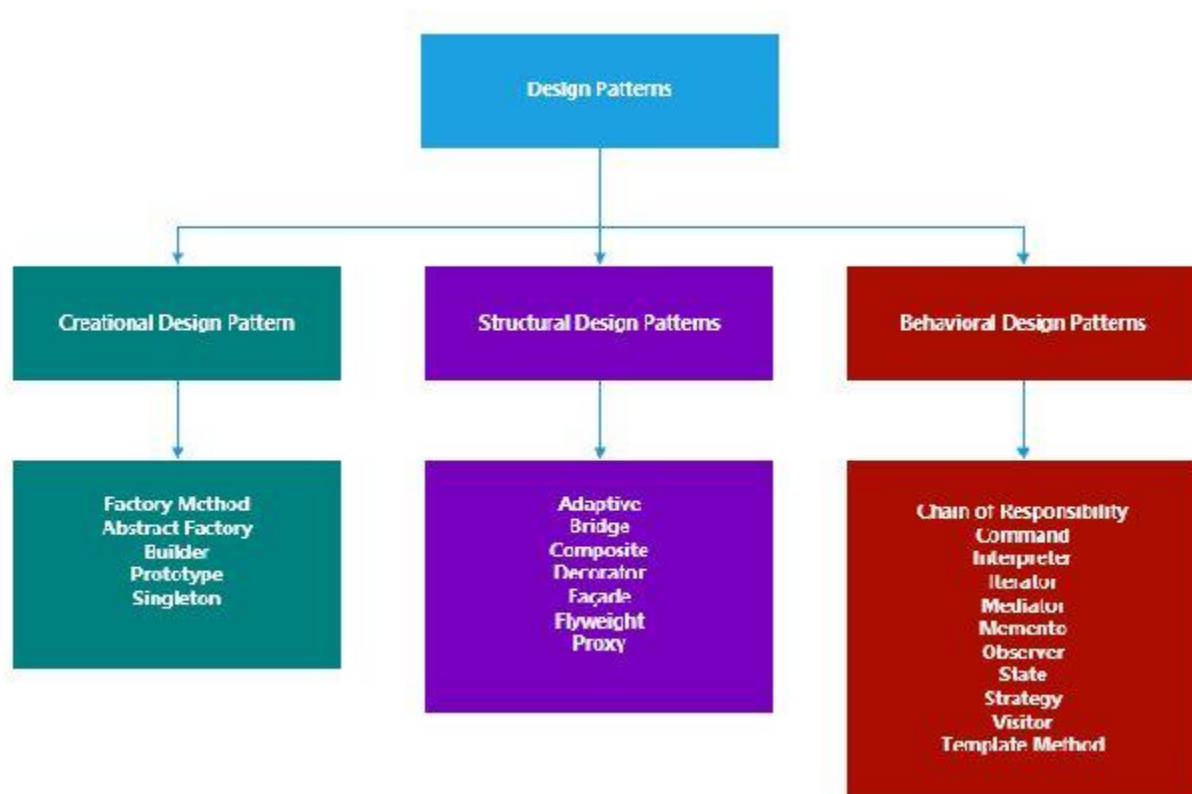
Design Patterns in the object-oriented world are a reusable solution to common software design problems that repeatedly occur in real-world application development. It is a template or description of how to solve problems that can be used in many situations.

"A pattern is a recurring solution to a problem in a context."

"Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice." - Christopher Alexander, A Pattern Language.

Developers use patterns for their specific designs to solve their problems. Pattern choice and usage among various design patterns depend on individual needs and concerns. Design patterns are a very powerful tool for software developers. It is important to understand design patterns rather than memorizing their classes, methods, and properties. Learning how to apply patterns to specific problems is also important to get the desired result. This will require continuous practice using and applying design patterns in software development. First, identify the software design problem, then see how to address these problems using design patterns and determine the best-suited design problem to solve the problem.

There are 23 design patterns, also known as Gang of Four (GoF) design patterns. The Gang of Four is the authors of the book, "Design Patterns: Elements of Reusable Object-Oriented Software". These 23 patterns are grouped into three main categories:



Chapter 35: C# 11-New Features

1. Raw String Literals

A new form of writing arbitrary text involving new lines, white space, single quotes, double quotes, and more special characters. Moreover, your code wouldn't need an escape sequence.

At the start and end of the string, you have to use at least three double quotes (""") characters to define a raw string literal. The string is initiated with a single line consisting of three double quotes. But when three double quotes are in separate lines, the string is terminated.

2. Newlines in string interpolation expressions

By enclosing the text within the {and} character, you are enabled to write string interpolation across multiple lines. The content between these markers will be considered C# code. It can include newlines and other valid C# syntax.

3. List Pattern

Let's say there is a sequence [1, 2, 3]. Now, it will match for both an array and a list of three integers. A List pattern allows you to match the sequences of elements in an array and a list using different patterns like relational, property, type, or constant.

Any single element will match with the discard pattern (_) whereas any sequence of zero or more elements will be matched with a new range pattern (..).

4. Required Members

The addition of a "required" modifier in fields and properties compels the callers and constructors to initialize those values.

5. Auto Default Struct

The initialization of default values of all the struct type fields is guaranteed by the C# 11 compiler as a part of the execution of a constructor. If a constructor does not initialize any auto property or field then they will be automatically initialized through the compiler.

So, the struct might not assign every field in the constructor but they would be compiled and the fields that aren't initialized will be set to their default value. In this case, C#10 and other previous language versions were seen displaying errors.

6. Generic Math Support

The mathematical operations on previous versions of C# were difficult to carry out as they needed many workarounds. But C# 11 offers new generic math interfaces in the System.Numerics namespace. It helps you write generic algorithms with any kind of numerics. It makes your code more readable and efficient.

7. Pattern match Span<char> or ReadOnlySpan<char> on a constant string

Microsoft says that in c# Pattern matching could help you check if any string consists of a constant value. Now, the same logic can be applied in matching the patterns for Span and ReadOnlySpan variables.

8. Extended nameof scope

The *nameof* function is used to generate the name of the member, variable, or type of a string constant. And since it is analyzed and implemented at the time of compilation, it doesn't affect run time in any possible way. Maintaining the argument checking code is one of the utilities of the *nameof* expression.

With the help of the `nameof` operators in the new C#11, specifying the name of method parameters in the method declaration's attribute has become possible. When used in the *nameof* expression, name and type parameters are found in the scope. It also makes adding attributes for nullable analysis easy.

9. UTF-8 string literals

This feature helps create UTF-8 strings. They help you with the HTTP string constants and other similar text protocols. To specify the UTF-8 encoding, you can set the `u8` suffix on the string literal.

10. File local types

This new feature of C#11 helps you source generator authors without the need for naming collisions. You can also scope the visibility of a type to its declared source file utilizing a file access modifier.

11. Improved Method Group Conversion to Delegate

Conversions from method group to delegate are made more simple and secure. If the selected method is not the derived method that receivers call at a runtime then a warning will be issued by the compiler.

There are possibilities that some errors might occur in the runtime in the case when the selected method is not the most derived method called by the receiver in the runtime.

12. Warning Wave 7

Another new feature from C#11 is the Warning Wave 7. It is a set of compiler warnings that are intended to promote safe and clean code. With the help of the project file elements such as `<AnalysisMode>` and `<AnalysisLevel>`, you will be able to control these warnings.

It is recommended that you do not ignore these warnings as they can help you improve the quality and security of the code. A warning-free code is a sign of a healthy codebase.

Chapter 36 Introduction to ADO.NET

- Understand what a data provider is.
- Understand what a connection object is.
- Understand what a command object is.
- Understand what a DataReader object is.
- Understand what a DataSet object is.
- Understand what a DataAdapter object is.

ADO.NET is an object-oriented set of libraries that allows you to interact with data sources. Commonly, the data source is a database, but it could also be a text file, an Excel spreadsheet, or an XML file. For the purposes of this tutorial, we will look at ADO.NET as a way to interact with a data base.

Data Providers

We know that ADO.NET allows us to interact with different types of data sources and different types of databases. However, there isn't a single set of classes that allow you to accomplish this universally. Since different data sources expose different protocols, we need a way to communicate with the right data source using the right protocol. Some older data sources use the ODBC protocol, many newer data sources use the OleDb protocol, and there are more data sources every day that allow you to communicate with them directly through .NET ADO.NET class libraries.

ADO.NET Data Providers are class libraries that allow a common way to interact with specific data sources or protocols. The library APIs have prefixes that indicate which provider they support.

Provider Name	API prefix	Data Source Description
ODBC Data Provider	Odbc	Data Sources with an ODBC interface. Normally older data bases.
OleDb Data Provider	OleDb	Data Sources that expose an OleDb interface, i.e. Access or Excel.
Oracle Data Provider	Oracle	For Oracle Databases.
SQL Data Provider	Sql	For interacting with Microsoft SQL Server.
Borland Data Provider	Bdp	Generic access to many databases such as Interbase, SQL Server, IBM DB2, and Oracle.

An example may help you to understand the meaning of the API prefix. One of the first ADO.NET objects you'll learn about is the connection object, which allows you to establish a connection to a data source. If we were using the OleDb Data Provider to connect to a data source that exposes an OleDb interface, we would use a connection object named OleDbConnection. Similarly, the connection object name would be prefixed with Odbc or Sql for an OdbcConnection object on an Odbc data source or a SqlConnection object on a SQL Server database, respectively. Since we are using MSDE in this tutorial (a scaled down version of SQL Server) all the API objects will have the Sql prefix. i.e. SqlConnection.

ADO.NET Objects

ADO.NET includes many objects you can use to work with data

The SqlConnection Object

To interact with a database, you must have a connection to it. A connection object is used by command objects so they will know which database to execute the command on.

The SqlCommand Object

The process of interacting with a database means that you must specify the actions you want to occur. This is done with a command object. You use a command object to send SQL statements to the database.

The SqlDataReader Object

Many data operations require that you only get a stream of data for reading. The data reader object allows you to obtain the results of a SELECT statement from a command object. For performance reasons, the data returned from a data reader is a fast forward-only stream of data. This means that you can only pull the data from the stream in a sequential manner. This is good for speed, but if you need to manipulate data, then a DataSet is a better object to work with.

The DataSet Object

DataSet objects are in-memory representations of data. They contain multiple DataTable objects, which contain columns and rows, just like normal database tables. You can even define relations between tables to create parent-child relationships. The DataSet is specifically designed to help manage data in memory and to support disconnected operations on data, when such a scenario make sense.

The SqlDataAdapter Object

Sometimes the data you work with is primarily read-only and you rarely need to make changes to the underlying data source. Some situations also call for caching data in memory to minimize the number of database calls for data that does not change. The data adapter makes it easy for you to accomplish these things by helping to manage data in a disconnected mode. The data adapter fills a DataSet object when reading the data and writes in a single batch when persisting changes back to the database. A data adapter contains a reference to the connection object and opens and closes the connection automatically when reading from or writing to the database. Additionally, the data adapter contains command object references for SELECT, INSERT, UPDATE, and DELETE operations on the data. You will have a data adapter defined for each table in a DataSet and it will take care of all communication with the database for you. All you need to do is tell the data adapter when to load from or write to the database.

The SqlConnection Object

- Know what connection objects are used for.
- Learn how to instantiate a SqlConnection object.
- Understand how the SqlConnection object is used in applications.
- Comprehend the importance of effective connection lifetime management.

Creating a SqlConnection Object

A SqlConnection is an object, just like any other C# object. Most of the time, you just declare and instantiate the SqlConnection all at the same time, as shown below:


```
SqlConnection conn = new SqlConnection(
    "Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");
```

The SqlConnection object instantiated above uses a constructor with a single argument of type string. This argument is called a connection string. table 1 describes common parts of a connection string.

table 1. ADO.NET Connection Strings contain certain key/value pairs for specifying how to make a database connection. They include the location, name of the database, and security credentials.

Connection String Parameter Name	Description
Data Source	Identifies the server. Could be local machine, machine domain name, or IP Address.
Initial Catalog	Database name.
Integrated Security	Set to SSPI to make connection with user's Windows login
User ID	Name of user configured in SQL Server.
Password	Password matching SQL Server User ID.

Integrated Security is secure when you are on a single machine doing development. However, you will often want to specify security based on a SQL Server User ID with permissions set specifically for the application you are using. The following shows a connection string, using the User ID and Password parameters:

```
SqlConnection conn = new SqlConnection(
    "Data Source=DatabaseServer;Initial Catalog=Northwind;User ID=YourUserID;Password=YourPassword");
```

Notice how the Data Source is set to DatabaseServer to indicate that you can identify a database located on a different machine, over a LAN, or over the Internet. Additionally, User ID and Password replace the Integrated Security parameter.

Using a SqlConnection

The purpose of creating a SqlConnection object is so you can enable other ADO.NET code to work with a database. Other ADO.NET objects, such as a SqlCommand and a SqlDataAdapter take a connection object as a parameter. The sequence of operations occurring in the lifetime of a SqlConnection are as follows:

1. Instantiate the SqlConnection.
2. Open the connection.
3. Pass the connection to other ADO.NET objects.
4. Perform database operations with the other ADO.NET objects.
5. Close the connection.

We've already seen how to instantiate a SqlConnection. The rest of the steps, opening, passing, using, and closing are shown in Listing 1.

```
using System;
using System.Data;
using System.Data.SqlClient;

/// Demonstrates how to work with SqlConnection objects
class SqlConnectionDemo
{
    static void Main()
    {
        // 1. Instantiate the connection
```

```

SqlConnection conn = new SqlConnection("Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");
SqlDataReader rdr = null;
try
{
    // 2. Open the connection
    conn.Open();
    // 3. Pass the connection to a command object
    SqlCommand cmd = new SqlCommand("select * from Customers", conn);
    // 4. Use the connection
    // get query results
    rdr = cmd.ExecuteReader();
    // print the CustomerID of each record
    while (rdr.Read())
    {
        Console.WriteLine(rdr[0]);
    }
}
finally
{
    // close the reader
    if (rdr != null)
    {
        rdr.Close();
    }
    // 5. Close the connection
    if (conn != null)
    {
        conn.Close();
    }
}
}

```

As shown in Listing 1, you open a connection by calling the *Open()* method of the *SqlConnection* instance, *conn*. Any operations on a connection that was not yet opened will generate an exception. So, you must open the connection before using it.

Before using a *SqlCommand*, you must let the ADO.NET code know which connection it needs. In Listing 1, we set the second parameter to the *SqlCommand* object with the *SqlConnection* object, *conn*. Any operations performed with the *SqlCommand* will use that connection.

The code that uses the connection is a *SqlCommand* object, which performs a query on the Customers table. The result set is returned as a *SqlDataReader* and the *while* loop reads the first column from each row of the result set, which is the CustomerID column. We'll discuss the *SqlCommand* and *SqlDataReader* objects in later lessons. For right now, it is important for you to understand that these objects are using the *SqlConnection* object so they know what database to interact with.

When you are done using the connection object, you must close it. Failure to do so could have serious consequences in the performance and scalability of your application. There are a couple points to be made about how we closed the connection in Listing 1: the *Close()* method is called in a *finally* block and we ensure that the connection is not null before closing it.

Notice that we wrapped the ADO.NET code in a *try/finally* block.

Another precaution you should take when closing connections is to make sure the connection object is not *null*.

This example showed how to use a `SqlConnection` object with a `SqlDataReader`, which required explicitly closing the connection. However, when using a disconnected data model, you don't have to open and close the connection yourself. We'll see how this works in a future lesson when we look at the `SqlDataAdapter` object.

The SqlCommand Object

- Know what a command object is.
- Learn how to use the `ExecuteReader` method to query data.
- Learn how to use the `ExecuteNonQuery` method to insert and delete data.
- Learn how to use the `ExecuteScalar` method to return a single value.

Creating a SqlCommand Object

Similar to other C# objects, you instantiate a `SqlCommand` object via the new instance declaration, as follows:

```
SqlCommand cmd = new SqlCommand("select CategoryName from Categories", conn);
```

The line above is typical for instantiating a `SqlCommand` object. It takes a string parameter that holds the command you want to execute and a reference to a `SqlConnection` object. `SqlCommand` has a few overloads, which you will see in the examples of this tutorial.

Querying Data

When using a SQL select command, you retrieve a data set for viewing. To accomplish this with a `SqlCommand` object, you would use the `ExecuteReader` method, which returns a `SqlDataReader` object. We'll discuss the `SqlDataReader` in a future lesson. The example below shows how to use the `SqlCommand` object to obtain a `SqlDataReader` object:

```
// 1. Instantiate a new command with a query and connection
SqlCommand cmd = new SqlCommand("select CategoryName from Categories", conn);

// 2. Call Execute reader to get query results
SqlDataReader rdr = cmd.ExecuteReader();
```

In the example above, we instantiate a `SqlCommand` object, passing the command string and connection object to the constructor. Then we obtain a `SqlDataReader` object by calling the `ExecuteReader` method of the `SqlCommand` object, `cmd`.

This code is part of the `ReadData` method of Listing 1 in the `Putting it All Together` section later in this lesson.

Inserting Data

To insert data into a database, use the `ExecuteNonQuery` method of the `SqlCommand` object. The following code shows how to insert data into a database table:

```
// prepare command string
string insertString = @"
insert into Categories
(CategoryName, Description)
values ('Miscellaneous', 'Whatever doesn't fit elsewhere');"
```

```
// 1. Instantiate a new command with a query and connection
```

```
SqlCommand cmd = new SqlCommand(insertString, conn);
```

```
// 2. Call ExecuteNonQuery to send command
```

```
cmd.ExecuteNonQuery();
```

The SqlCommand instantiation is just a little different from what you've seen before, but it is basically the same. Instead of a literal string as the first parameter of the SqlCommand constructor, we are using a variable, insertString. The insertString variable is declared just above the SqlCommand declaration.

Notice the two apostrophes (") in the insertString text for the word "doesn't". This is how you escape the apostrophe to get the string to populate column properly.

Another observation to make about the insert command is that we explicitly specified the columns CategoryName and Description.

To execute this command, we simply call the ExecuteNonQuery method on the SqlCommand instance, cmd.

This code is part of the Insertdata method of Listing 1 in the Putting it All Together section later in this lesson.

Deleting Data

You can also delete data using the ExecuteNonQuery method. The following example shows how to delete a record from a database with the ExecuteNonQuery method:

```
// prepare command string
```

```
string deleteString = @"  
delete from Categories  
where CategoryName = 'Other';
```

```
// 1. Instantiate a new command
```

```
SqlCommand cmd = new SqlCommand();
```

```
// 2. Set the CommandText property
```

```
cmd.CommandText = deleteString;
```

```
// 3. Set the Connection property
```

```
cmd.Connection = conn;
```

```
// 4. Call ExecuteNonQuery to send command
```

```
cmd.ExecuteNonQuery();
```

This example uses the SqlCommand constructor with no parameters. Instead, it explicitly sets the CommandText and Connection properties of the SqlCommand object, cmd.

We could have also used either of the two previous SqlCommand constructor overloads, used for the insert or update command, with the same result. This demonstrates that you can change both the command text and the connection object at any time.

The ExecuteNonQuery method call sends the command to the database.

This code is part of the DeleteData method of Listing 1 in the Putting it All Together section later in this lesson.

Getting Single values

Sometimes all you need from a database is a single value, which could be a count, sum, average, or other aggregated value from a data set. Performing an `ExecuteReader` and calculating the result in your code is not the most efficient way to do this. The best choice is to let the database perform the work and return just the single value you need. The following example shows how to do this with the `ExecuteScalar` method:

```
// 1. Instantiate a new command
SqlCommand cmd = new SqlCommand("select count(*) from Categories", conn);

// 2. Call ExecuteNonQuery to send command
int count = (int)cmd.ExecuteScalar();
```

The query in the `SqlCommand` constructor obtains the count of all records from the `Categories` table. This query will only return a single value. The `ExecuteScalar` method in step 2 returns this value. Since the return type of `ExecuteScalar` is type object, we use a cast operator to convert the value to `int`.

This code is part of the `GetNumberOfRecords` method of Listing 1 in the `Putting it All Together` section later in this lesson.

Putting it All Together

For simplicity, we showed snippets of code in previous sections to demonstrate the applicable techniques. It is also useful to have an entire code listing to see how this code is used in a working program. Listing 1 shows all of the code used in this example, along with a driver in the `Main` method to produce formatted output.

Listing 1. SqlConnection Demo

```
using System;
using System.Data;
using System.Data.SqlClient;

/// Demonstrates how to work with SqlCommand objects
class SqlCommandDemo
{
    SqlConnection conn;

    public SqlCommandDemo()
    {
        // Instantiate the connection
        conn = new SqlConnection(
            "Data Source=(local);Initial Catalog=Northwind;Integrated Security=SSPI");
    }

    // call methods that demo SqlCommand capabilities
    static void Main()
    {
        SqlCommandDemo scd = new SqlCommandDemo();

        Console.WriteLine();
        Console.WriteLine("Categories Before Insert");
        Console.WriteLine("-----");

        // use ExecuteReader method
        scd.ReadData();
        // use ExecuteNonQuery method for Insert
        scd.Insertdata();
        Console.WriteLine();
        Console.WriteLine("Categories After Insert");
    }
}
```

```

Console.WriteLine("-----");

scd.ReadData();
// use ExecuteNonQuery method for Update
scd.UpdateData();

Console.WriteLine();
Console.WriteLine("Categories After Update");
Console.WriteLine("-----");

scd.ReadData();
// use ExecuteNonQuery method for Delete
scd.DeleteData();

Console.WriteLine();
Console.WriteLine("Categories After Delete");
Console.WriteLine("-----");

scd.ReadData();

// use ExecuteScalar method
int numberOfRecords = scd.GetNumberOfRecords();

Console.WriteLine();
Console.WriteLine("Number of Records: {0}", numberOfRecords);
}

/// use ExecuteReader method
public void ReadData()
{
    SqlDataReader rdr = null;

    try
    {
        // Open the connection
        conn.Open();
        // 1. Instantiate a new command with a query and connection
        SqlCommand cmd = new SqlCommand("select CategoryName from Categories", conn);
        // 2. Call Execute reader to get query results
        rdr = cmd.ExecuteReader();
        // print the CategoryName of each record
        while (rdr.Read())
        {
            Console.WriteLine(rdr[0]);
        }
    }
    finally
    {
        // close the reader
        if (rdr != null)
        {
            rdr.Close();
        }

        // Close the connection
        if (conn != null)
        {
            conn.Close();
        }
    }
}

/// <summary>
/// use ExecuteNonQuery method for Insert

```

```

///</summary>
public void Insertdata()
{
    try
    {
        // Open the connection
        conn.Open();
        // prepare command string
        string insertString = @"
            insert into Categories
            (CategoryName, Description)
            values ('Miscellaneous', 'Whatever doesn't fit elsewhere');
        // 1. Instantiate a new command with a query and connection
        SqlCommand cmd = new SqlCommand(insertString, conn);

        // 2. Call ExecuteNonQuery to send command
        cmd.ExecuteNonQuery();
    }
    finally
    {
        // Close the connection
        if (conn != null)
        {
            conn.Close();
        }
    }
}

/// use ExecuteNonQuery method for Update
public void UpdateData()
{
    try
    {
        // Open the connection
        conn.Open();
        // prepare command string
        string updateString = @"
            update Categories
            set CategoryName = 'Other'
            where CategoryName = 'Miscellaneous';

        // 1. Instantiate a new command with command text only
        SqlCommand cmd = new SqlCommand(updateString);
        // 2. Set the Connection property
        cmd.Connection = conn;
        // 3. Call ExecuteNonQuery to send command
        cmd.ExecuteNonQuery();
    }
    finally
    {
        // Close the connection
        if (conn != null)
        {
            conn.Close();
        }
    }
}

/// use ExecuteNonQuery method for Delete
public void DeleteData()
{
    try
    {
        // Open the connection
        conn.Open();
        // prepare command string
    }
}

```

```

string deleteString = @"
    delete from Categories
    where CategoryName = 'Other'";

// 1. Instantiate a new command
SqlCommand cmd = new SqlCommand();
// 2. Set the CommandText property
cmd.CommandText = deleteString;
// 3. Set the Connection property
cmd.Connection = conn;
// 4. Call ExecuteNonQuery to send command
cmd.ExecuteNonQuery();
}
finally
{
    // Close the connection
    if (conn != null)
    {
        conn.Close();
    }
}
}
/// use ExecuteScalar method
///<returns>number of records</returns>
public int GetNumberOfRecords()
{
    int count = -1;
    try
    {
        // Open the connection
        conn.Open();
        // 1. Instantiate a new command
        SqlCommand cmd = new SqlCommand("select count(*) from Categories", conn);
        // 2. Call ExecuteScalar to send command
        count = (int)cmd.ExecuteScalar();
    }
    finally
    {
        // Close the connection
        if (conn != null)
        {
            conn.Close();
        }
    }
    return count;
}
}
}

```

In Listing 1, the SqlConnection object is instantiated in the SqlCommandDemo structure. This is okay because the object itself will be cleaned up when the CLR garbage collector executes. What is important is that we close the connection when we are done using it. This program opens the connection in a try block and closes it in a finally block in each method.

The ReadData method displays the contents of the CategoryName column of the Categories table. We use it several times in the Main method to show the current status of the Categories table, which changes after each of the insert, update, and delete commands. Because of this, it is convenient to reuse to show you the effects after each method call.

Summary

A SqlCommand object allows you to query and send commands to a database. It has methods that are specialized for different commands. The ExecuteReader method returns a SqlDataReader object for viewing the results of a select query. For insert, update, and delete SQL commands, you use the ExecuteNonQuery method. If you only need a single aggregate value from a query, the ExecuteScalar is the best choice.

INTERACTING WITH MICROSOFT ACCESS

```
using System. Data;
using System.Data.OleDb;

private void btnInsert_Click(object sender, EventArgs e)
{
    olnConn = new OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;Data Source =
D:\students.mdb;Persist Security Info=False");
    cmd = new OleDbCommand();

    cmd.CommandText = "insert into student values(" + this.txtRollNo.Text + "," + this.txtName.Text + "," + this.txtFatherName.Text + "," + this.txtAddress.Text + ")";

    olnConn.Open();
    cmd.Connection = olnConn;
    int temp = cmd.ExecuteNonQuery();
    if (temp > 0)
    {
        MessageBox.Show("Record Added");
    }
    else
    {
        MessageBox.Show("Record not Added");
    }
    olnConn.Close();
}
```

ADO.NET TRANSACTIONS

Within .NET, transactions are managed with the System.Data.SqlClient.SqlTransaction class. Again, a transaction exists over a SqlConnection object – and thus all the SqlCommand objects you create using that connection. Let's look at a quick example:

```
using System;
using System.Data.SqlClient;

namespace Ado_transaction
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Before Transaction");
                GetAccountsData();

                //Doing the Transaction
            }
        }
    }
}
```

```

        MoneyTransfer();

        //Verifying the Data After Transaction
        Console.WriteLine("After Transaction");

        GetAccountsData();
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Exception Occurred: {ex.Message}");
    }
    Console.ReadKey();
}

private static void MoneyTransfer()
{
    //Store the connection string in a variable
    string ConnectionString = @"Data Source=DELL\CAPGEMINI; Initial Catalog=ADODB;
Integrated Security=True";

    //Creating the connection object
    using (SqlConnection connection = new SqlConnection(ConnectionString))
    {
        //Open the connection
        //The connection needs to be open before we begin a transaction
        connection.Open();

        // Create the transaction object by calling the BeginTransaction method on
        connection object
        SqlTransaction transaction = connection.BeginTransaction();

        try
        {
            // Associate the first update command with the transaction
            SqlCommand cmd = new SqlCommand("UPDATE Accounts SET Balance = Balance - 500
WHERE AccountNumber = 'Account1'", connection, transaction);
            //Execute the First Update Command
            cmd.ExecuteNonQuery();

            // Associate the second update command with the transaction
            cmd = new SqlCommand("UPDATE Accounts SET Balance = Balance + 500 WHERE
AccountNumber = 'Account2'", connection, transaction);
            //Execute the Second Update Command
            cmd.ExecuteNonQuery();

            // If everythingh goes well then commit the transaction
            transaction.Commit();
            Console.WriteLine("Transaction Committed");
        }
        catch (Exception ex)
        {
            // If anything goes wrong, then Rollback the transaction
            transaction.Rollback();
            Console.WriteLine("Transaction Rollback");
        }
    }
}

private static void GetAccountsData()
{
    //Store the connection string in a variable
    string ConnectionString = @"Data Source=DELL\CAPGEMINI; Initial Catalog=ADODB;
Integrated Security=True";

    //Create the connection object
    using (SqlConnection connection = new SqlConnection(ConnectionString))

```

```

        {
            connection.Open();
            SqlCommand cmd = new SqlCommand("Select * from Accounts", connection);
            SqlDataReader sdr = cmd.ExecuteReader();
            while (sdr.Read())
            {
                Console.WriteLine(sdr["AccountNumber"] + ", " + sdr["CustomerName"] + ", " +
+ sdr["Balance"]);
            }
        }
    }
}

```

Bulk Insert Update in C# using Stored Procedure

Step1 : Create Database and Table

create database adonew

```

CREATE TABLE Employee(
    Id INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(50),
    Mobile VARCHAR(50),
)
GO
INSERT INTO Employee VALUES (100, 'Anurag', 'Anurag@dotnettutorial.net', '1234567890')
INSERT INTO Employee VALUES (101, 'Priyanka', 'Priyanka@dotnettutorial.net', '2233445566')
INSERT INTO Employee VALUES (102, 'Preety', 'Preety@dotnettutorial.net', '6655443322')
INSERT INTO Employee VALUES (103, 'Sambit', 'Sambit@dotnettutorial.net', '9876543210')
GO

```

Step 2:

Create table type

--A Table-Type Parameter is a database object in SQL Server and it provides an easy way to pass multiple rows of data from a client application to an SQL Server Database without multiple round trips. That means we can use a Table-Type Parameter to encapsulate rows of data in a client application and send we can send those rows of data to the SQL Server Database in a single request. Once the SQL Server received the data, then the SQL Server process the data

```

CREATE TYPE EmployeeType AS TABLE(
    Id INT NULL,
    Name VARCHAR(100) NULL,
    Email VARCHAR(50) NULL,
    Mobile VARCHAR(50) NULL
)
GO

```

Step 3:

--Stored Procedure using MERGE statement to perform Bulk Insert and Update Operations:

```

CREATE PROCEDURE SP_Bulk_Insert_Update_Employees
    @Employees EmployeeType READONLY
AS
BEGIN
    SET NOCOUNT ON;

    MERGE INTO Employee E1
    USING @Employees E2
    ON E1.Id=E2.Id

```

```

WHEN MATCHED THEN
UPDATE SET
    E1.Name = E2.Name,
    E1.Email = E2.Email,
    E1.Mobile = E2.Mobile
WHEN NOT MATCHED THEN
INSERT VALUES(E2.Id, E2.Name, E2.Email, E2.Mobile);
END

```

Step 4:

C# code

```

using System;
using System.Data;
using System.Data.SqlClient;

namespace BulkOperADO
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                //Storing the connection string in a variable
                string ConnectionString = @"Data Source=DELL\CAPGEMINI; Initial Catalog=adonew;
Integrated Security=True";
                //Creating Data Table
                DataTable EmployeeDataTable = new DataTable("Employees");
                //Add Columns to the Data Table as per the columns defined in the Table Type
                Parameter
                DataColumn Id = new DataColumn("Id");
                EmployeeDataTable.Columns.Add(Id);
                DataColumn Name = new DataColumn("Name");
                EmployeeDataTable.Columns.Add(Name);
                DataColumn Email = new DataColumn("Email");
                EmployeeDataTable.Columns.Add(Email);
                DataColumn Mobile = new DataColumn("Mobile");
                EmployeeDataTable.Columns.Add(Mobile);
                //Adding Multiple Rows into the DataTable
                //Following Rows are going to be updated
                EmployeeDataTable.Rows.Add(101, "ABC", "ABC@enosisis.com", "12345");
                EmployeeDataTable.Rows.Add(102, "PQR", " PQR@enosisis.com ", "11223");
                EmployeeDataTable.Rows.Add(103, "XYZ", " XYZ@enosisis.com ", "23432");
                //Following Rows are going to be Inserted
                EmployeeDataTable.Rows.Add(106, "A", "A@enosisislearning.com ", "12345");
                EmployeeDataTable.Rows.Add(107, "B", " B@enosisislearning.com ", "23456");
                EmployeeDataTable.Rows.Add(108, "C", " C@enosisislearning.com ", "34567");
                EmployeeDataTable.Rows.Add(109, "D", "D@enosisislearning.com ", "45678");
                EmployeeDataTable.Rows.Add(110, "E", " E@enosisislearning.com ", "56789");
                EmployeeDataTable.Rows.Add(111, "F", " F@enosisislearning.com ", "67890");
                //Creating the connection object
                using (SqlConnection connection = new SqlConnection(ConnectionString))
                {
                    //You can pass any stored procedure
                    //As I am using Higher version of SQL Server, so, I am using the Stored
                    Procedure which uses MERGE Function
                    using (SqlCommand cmd = new SqlCommand("SP_Bulk_Insert_Update_Employees",
connection))
                    {
                        //Set the command type as StoredProcedure
                        cmd.CommandType = CommandType.StoredProcedure;
                        //Add the input parameter required by the stored procedure
                        cmd.Parameters.AddWithValue("@Employees", EmployeeDataTable);
                    }
                }
            }
            catch { }
        }
    }
}

```

```

        //Open the connection
        connection.Open();
        //Execute the command
        cmd.ExecuteNonQuery();
    }
}
Console.WriteLine("BULK INSERT UPDATE Successful");
}
catch (Exception ex)
{
    Console.WriteLine($"Exception Occurred: {ex.Message}");
}
Console.ReadKey();
}
}
}

```

