Unit 2

Introduction to C#

C# (pronounced "C-sharp") is a popular, modern, and versatile programming language developed by Microsoft. It was introduced in the early 2000s and has since become a fundamental part of the Microsoft technology stack. C# is widely used for developing a wide range of applications, including desktop software, web applications, mobile apps, games, and more. Here's an introduction to C#:

History

C# was developed by Microsoft and first released in 2000 as part of the .NET framework. It was designed to be a modern, object-oriented, and type-safe language for building Windows applications. Over the years, it has evolved and expanded its capabilities to support various application types and platforms.

Key Features and Overview

Object-Oriented: C# is an object-oriented language, which means it allows you to model your software using objects and classes, promoting code reuse and maintainability.

- **Type-Safe**

C# enforces strong type checking at compile-time, reducing the likelihood of runtime errors and improving code reliability.

- **Garbage Collection**

It has automatic memory management through a garbage collector, which simplifies memory management and reduces the risk of memory leaks.

- **Platform Independence**

C# can be used to develop applications for various platforms, including Windows, web, mobile (via Xamarin), and even cross-platform solutions with .NET Core (now .NET 5 and later).

- **Rich Standard Library**

C# comes with a rich class library, known as the .NET Framework (or .NET Core/.NET 5+), which provides a wide range of pre-built functions and classes for common programming tasks.

- **Asynchronous Programming**

It has native support for asynchronous programming using the async and await keywords, making it well-suited for developing responsive and scalable applications.

- **Language Integration**

C# seamlessly integrates with other Microsoft technologies like ASP.NET for web development and Xamarin for cross-platform mobile development.

- **Development Environments**

Visual Studio: Microsoft's flagship integrated development environment (IDE), Visual Studio, is commonly used for C# development. Visual Studio provides a comprehensive set of tools for C# developers.

- **Visual Studio Code**

Many C# developers also use Visual Studio Code, a lightweight, cross-platform code editor with robust C# support, extensions, and debugging capabilities.

Sample C# Code

```
using System;

class Program

{

   static void Main()

   {

      Console.WriteLine("Hello, C#!");

   }

}
```

In this simple example, C# is used to print "Hello, C#!" to the console.

- **Community and Ecosystem**

C# has a vibrant developer community and a rich ecosystem of libraries, frameworks, and tools, making it suitable for a wide range of applications, from enterprise software to game development.

- **Learning Resources**

If you're interested in learning C#, there are numerous online tutorials, courses, and documentation available to help you get started.

Data Types

C# is a statically-typed language, which means that variables must have a defined data type at compile-time. C# provides a variety of built-in data types to store different kinds of values. Here are some of the fundamental data types in C#:

Numeric Data Types

Int - Represents signed 32-bit integers. Example: int myNumber = 42;

Long - Represents signed 64-bit integers. Example: long myLongNumber = 1234567890L;

Float - Represents single-precision floating-point numbers. Example: float myFloat = 3.14f;

Double - Represents double-precision floating-point numbers. Example: double myDouble = 3.14159265359;

Decimal - Represents decimal numbers with higher precision, often used for financial calculations. Example: decimal myDecimal = 123.45m;

- **Boolean Data Type**

Bool - Represents a Boolean value, which can be either true or false. Example: bool isTrue = true;

- **Character Data Type**

Char - Represents a single Unicode character. Example: char myChar = 'A';

- **String Data Type**

String - Represents a sequence of characters. Example: string myString = "Hello, C#!";

- **Enumeration (Enum) Data Type**

Enum - Represents a set of named integer constants.

Example:

```
enum DaysOfWeek
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
```

Sunday

}

- **Date and Time Data Types**

DateTime - Represents a date and time. Example: DateTime currentDate = DateTime.Now;

TimeSpan - Represents a duration of time. Example: TimeSpan timeInterval = TimeSpan.FromHours(2);

- **Reference Types**

Object - Represents a reference to an instance of a class or a value type. It's the root type of the C# type hierarchy.

Dynamic - Represents a type whose behavior is determined at runtime. It allows for dynamic typing.

Null - Represents a reference that does not refer to any object. It's often used to indicate the absence of a value.

- **Arrays**

Arrays allow you to store collections of elements of the same data type. Example: int[] numbers = { 1, 2, 3, 4, 5 };

- **User-Defined Types**

You can define your own custom data types using classes and structs in C#. These types can have fields and properties to store and manipulate data.

- **Nullable Value Types**

You can make value types nullable by appending ? to their type names. For example, int? nullableInt = null; allows the variable to store an integer value or null.

These are some of the basic data types in C#. Depending on your application's requirements, you can choose the appropriate data type to store and manipulate data efficiently and accurately. Additionally, C# allows you to create custom data types by defining classes and structs, which gives you great flexibility in designing your data structures.

Expressions

In C#, an expression is a combination of values, operators, variables, and method calls that can be evaluated to produce a single value. Expressions can be used in a variety of contexts, such as assigning values to variables, making decisions in control structures, performing mathematical calculations, and more. Here are some common types of expressions in C#:

- **Arithmetic Expressions**

Arithmetic expressions involve mathematical operations like addition, subtraction, multiplication, and division.

For example:

```
int result = 5 + 3;       // Addition

int difference = 10 - 4;  // Subtraction

int product = 6 * 2;      // Multiplication

int quotient = 16 / 4;    // Division
```

- **Relational Expressions**

Relational expressions compare values and return a Boolean result (true or false). Common relational operators include

```
bool isEqual = 10 == 5;   // Equal to

bool isNotEqual = 7 != 3;  // Not equal to

bool isGreaterThan = 8 > 3;   // Greater than

bool isLessThan = 4 < 9;      // Less than
```

- **Logical Expressions**

Logical expressions use logical operators to combine or modify Boolean values. Common logical operators include && (logical AND), || (logical OR), and ! (logical NOT).

For example:

```
bool condition1 = true;

bool condition2 = false;

bool andResult = condition1 && condition2;  // Logical AND

bool orResult = condition1 || condition2;    // Logical OR

bool notResult = !condition1;              // Logical NOT
```

- **Conditional Expressions (Ternary Operator)**

The ternary conditional operator ? : allows you to create compact conditional expressions. For example:

```
int x = 10;

int y = 5;

int result = (x > y) ? x : y;  // If x is greater than y, result = x; otherwise, result = y;
```

- **Method Call Expressions**

Expressions can also include method calls that return values.

For example:

```
int CalculateSum(int a, int b)
{
   return a + b;
}
int sum = CalculateSum(3, 4);  // Calling a method and assigning its result to a variable
```

- **Assignment Expressions**

Assignment expressions are used to assign values to variables. For example:

```
int value = 42;  // Assignment expression
```

- **Array and Index Expressions**

You can use expressions to access elements of an array or collection using indexers. For example:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int element = numbers[2];  // Accessing an array element
```

- **Lambda Expressions**

Lambda expressions are used to create inline anonymous methods or functions. They are often used with delegates or functional interfaces in C#.

These are just a few examples of expressions in C#. Expressions are fundamental to the language, and understanding how they work is crucial for writing effective C# code. Depending on the context, C# allows you to use expressions to manipulate data, control program flow, and perform a wide range of tasks.

Declarations and Statements in C#

In C#, declarations are used to define variables, methods, classes, structures, interfaces, enums, delegates, and other program elements. Declarations provide information about the name, data type, and scope of the program elements you are defining. Here are some common types of declarations in C#:

- **Variable Declarations:**

Variables store data values. When declaring a variable, you specify its data type and optionally assign an initial value.

```
int age;            // Declaration of an integer variable
double price = 29.99; // Declaration with initialization
```

- **Method Declarations:**

Methods define reusable blocks of code. They specify the method's return type, name, and parameters.

```
int Add(int x, int y)  // Method declaration
{
   return x + y;
}
```

- **Class Declarations:**

Classes are used to define objects with properties and methods.

```
public class Person  // Class declaration
{
   public string Name { get; set; }
   public int Age { get; set; }
}
```

- **Struct Declarations:**

Structs are similar to classes but are value types.

```
public struct Point  // Struct declaration
{
   public int X { get; set; }
   public int Y { get; set; }
}
```

- **Interface Declarations:**

Interfaces define a contract that implementing classes must adhere to.

```
public interface ILogger  // Interface declaration
{
   void Log(string message);
}
```

- **Enum Declarations:**

Enums define a set of named constants.

```
public enum DaysOfWeek  // Enum declaration
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}
```

- **Delegate Declarations:**

Delegates are used to define method signatures that can be assigned to functions at runtime.

```
public delegate int MathOperation(int a, int b);  // Delegate declaration
```

- **Event Declarations:**

Events are used to define and raise events in C#.

```
public event EventHandler ButtonClicked;  // Event declaration
```

- **Namespace Declarations:**

Namespaces are used to organize code into logical groups.

```
namespace MyNamespace  // Namespace declaration
{
   // Code goes here
}
```

- **Constant and Readonly Field Declarations:**

Constants and readonly fields are used to define values that cannot be changed after declaration.

```
const int MaxValue = 100;    // Constant declaration
readonly int MinValue = 0;   // Readonly field declaration
```

- **Property Declarations:**

Properties provide controlled access to class fields.

```
public int Age { get; set; }  // Property declaration
```

- **Event Handler Method Declarations:**

Event handlers define methods that respond to events.

```
private void Button_Click(object sender, EventArgs e)  // Event handler method declaration
{
   // Event handling code
}
```

These are some of the most common types of declarations in C#. Depending on your program's structure and requirements, you will use various declarations to define and organize your code effectively. Declarations provide the foundation for building C# programs and specifying the structure and behaviour of your application's elements.

Classes & Structs

In C#, classes and structs are both used to define custom data types, but they have some key differences in terms of their behavior and usage. Here's an overview of classes and structs in C#:

Classes

- **Reference Types:**

Classes are reference types, which means when you create an instance of a class, you are working with a reference to the object, not the object itself. Multiple references can point to the same object in memory.

- **Heap Allocation:**

Class objects are allocated on the heap, which is a managed memory area. They have a longer lifespan and are subject to garbage collection when no longer referenced.

- **Default Constructor:**

If you don't explicitly define a constructor, classes have a default constructor that initializes object properties to default values (null for reference types, zero for value types).

- **Inheritance:**

Classes support inheritance, allowing you to create hierarchies of related classes using the extends keyword. This enables code reuse and polymorphism.

- **Polymorphism:**

Classes can take advantage of polymorphism, allowing you to use base class references to refer to derived class objects.

- **Use for Complex Objects:**

Classes are typically used for defining complex data structures, objects, and behaviours, such as modelling real-world entities or implementing various application components.

Example of a class

```
public class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
   public void SayHello()
   {
      Console.WriteLine($"Hello, my name is {Name} and I'm {Age} years old.");
   }
}
```

Structs:

Value Types: Structs are value types, which means when you create an instance of a struct, you are working with the actual object itself, not a reference. Each variable holds its own copy of the struct.

Stack Allocation: Struct objects are allocated on the stack, which is a memory region used for local variables and has a shorter lifespan than the heap. They are not subject to garbage collection.

Parameterless Constructor: Structs do not have a parameterless constructor provided by default. You need to initialize all of their fields in the constructor.

No Inheritance: Structs do not support inheritance or polymorphism. They cannot be derived from or serve as base classes.

Use for Lightweight Data: Structs are typically used for lightweight data structures that represent small, logically related pieces of data, such as coordinates, colors, or simple point-like objects.

Example of a struct:

```
public struct Point
{
   public int X { get; }
   public int Y { get; }

   public Point(int x, int y)
   {
      X = x;
      Y = y;
   }
}
```

In summary, classes and structs in C# serve different purposes and have different characteristics. Classes are used for defining complex objects and behaviours, support inheritance, and are allocated on the heap. Structs are used for lightweight data structures, do not support inheritance, and are allocated on the stack. Choosing between them depends on your specific programming needs and design considerations.

OOPS in C#

Object-Oriented Programming (OOP) is a fundamental programming paradigm used in C# and many other modern programming languages. OOP is based on the concept of objects, which are instances of classes, and it focuses on encapsulation, inheritance, and polymorphism. Here's an overview of OOP principles and how they are implemented in C#:

Classes and Objects

In C#, classes are used to define blueprints for objects. Objects are instances of classes that encapsulate data (attributes or properties) and behaviour (methods).

Example of a class and object in C#

```
public class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
   public void SayHello()
   {
```

```
        Console.WriteLine($"Hello, my name is {Name} and I'm {Age} years old.");
    }
}
// Creating an object of the Person class
Person person1 = new Person();
person1.Name = "Alice";
person1.Age = 30;
person1.SayHello();
```

Encapsulation

Encapsulation is the concept of bundling data (attributes or fields) and the methods (functions) that operate on that data into a single unit, called a class. It restricts direct access to an object's data and allows controlled access through methods.

Access modifiers like public, private, protected, and internal control the visibility and accessibility of class members.

Example

```
public class BankAccount
{
    private decimal balance;
    public void Deposit(decimal amount)
    {
        if (amount > 0)
        {
            balance += amount;
        }
    }
    public decimal GetBalance()
    {
        return balance;
    }
}
```

Inheritance

Inheritance allows you to create a new class (derived or subclass) based on an existing class (base or superclass). The derived class inherits the properties and methods of the base class and can extend or override them.

Example

```
public class Animal
{
   public void Eat()
   {
      Console.WriteLine("The animal is eating.");
   }
}
public class Dog : Animal
{
   public void Bark()
   {
      Console.WriteLine("The dog is barking.");
   }
}
Dog myDog = new Dog();
myDog.Eat();  // Inherited method
myDog.Bark(); // Derived class method
```

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables method overriding and dynamic method binding.

Example:

```
public class Shape
{
   public virtual void Draw()
   {
      Console.WriteLine("Drawing a shape.");
   }
```

```
}
public class Circle : Shape
{
   public override void Draw()
   {
      Console.WriteLine("Drawing a circle.");
   }
}
Shape myShape = new Circle();
myShape.Draw(); // Calls the Circle's overridden method
```

Abstraction

Abstraction is the process of simplifying complex systems by modeling classes based on their essential properties and behaviors, while hiding unnecessary details.

Abstract classes and interfaces are used to define abstractions in C#.

Interfaces

Interfaces define a contract of methods and properties that implementing classes must adhere to. They support multiple inheritance in C#.

Example

```
public interface IShape
{
   void Draw();
   double GetArea();
}
public class Circle : IShape
{
   public void Draw()
   {
      Console.WriteLine("Drawing a circle.");
   }
   public double GetArea()
   {
```

```
        // Calculate and return the area of the circle
    }
}
```

OOP promotes code reusability, maintainability, and modularity by organizing code into classes and objects. C# is a versatile language for implementing OOP principles, making it suitable for building complex and organized software systems.

Constructors & Destructors

In C#, constructors and destructors are special methods used in classes to initialize and clean up objects, respectively. Constructors are used to set up an object's initial state, while destructors are used to release any resources held by an object when it is no longer needed. Here's an overview of constructors and destructors in C#:

Constructors:

Purpose: Constructors are used to initialize the state of an object when it is created. They ensure that the object starts with valid and meaningful values.

Syntax: A constructor has the same name as the class and does not have a return type, not even void. It can have parameters, allowing you to provide initial values during object creation.

Default Constructor: If a class does not define any constructors, C# provides a default parameterless constructor, which initializes fields and properties to their default values (e.g., null for reference types, zero for numeric types).

Overloaded Constructors: You can define multiple constructors within a class with different parameter lists. This allows you to create objects in various ways, providing flexibility in initialization.

Constructor Chaining: Constructors can call other constructors in the same class using the this keyword. This is known as constructor chaining. It helps avoid code duplication when multiple constructors share common initialization logic.

Example:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    // Parameterized constructor
    public Person(string name, int age)
```

```
    {
        Name = name;
        Age = age;
    }
    // Default constructor
    public Person()
    {
        Name = "Unknown";
        Age = 0;
    }
}
```

Destructors (Finalizers):

Purpose: Destructors, also known as finalizers, are used to clean up and release resources held by an object before it is garbage collected. They are called automatically by the garbage collector when an object is no longer reachable.

Syntax: A destructor is defined with the same name as the class but prefixed with a tilde (~). Destructors do not take any parameters and do not return a value. You cannot explicitly call a destructor; they are invoked by the garbage collector.

Resource Cleanup: Destructors are often used to release unmanaged resources, such as file handles, database connections, or memory allocated using native code (e.g., pointers).

Finalize Method: Under the hood, C# destructors are implemented using a method called Finalize. However, you typically don't need to write a destructor explicitly because C# allows you to use the using statement or implement IDisposable for better resource management.

Example (simplified):

```
public class ResourceHolder
{
    private IntPtr resource; // Example unmanaged resource
    // Constructor
    public ResourceHolder()
    {
        resource = /* Initialize unmanaged resource */;
```

```
    }
    // Destructor (Finalizer)
    ~ResourceHolder()
    {
        // Release unmanaged resource
        /* Cleanup code for resource */
    }
}
```

It's important to note that in modern C#, explicit destructors are rarely needed due to the presence of the IDisposable pattern, which allows you to implement resource cleanup more explicitly and efficiently using the Dispose method. Additionally, the C# using statement can help manage resources by ensuring that Dispose is called when an object goes out of scope.

Variables

Variables are containers for storing data values.

In C#, there are different **types** of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

Declaring Variables:

To declare a variable in C#, you specify the data type followed by the variable name.

For example:

```
int age; // Declaring an integer variable named 'age'
```

Initializing Variables:

Variables can be initialized (given an initial value) at the time of declaration or later in the code:

```
int count = 0; // Declaration and initialization
```

age = 30; // Initializing a previously declared variable

Data Types:

C# provides various built-in data types for different kinds of data, including integers (int, long, short), floating-point numbers (float, double, decimal), characters (char), Booleans (bool), and more. You can also use user-defined data types by creating your own classes and structures.

Variable Names:

Variable names must start with a letter (uppercase or lowercase) or an underscore (_). Subsequent characters can include letters, numbers, and underscores. Variable names are case-sensitive, so myVariable and MyVariable are considered different variables.

Scope of Variables:

Variables have a scope, which defines where they are accessible and valid within the code. Local variables are declared within a specific code block (e.g., inside a method) and have limited scope. Class-level variables (fields) are declared within a class and can be accessed throughout the class.

Arrays

In C#, an array is a data structure used to store a collection of elements of the same data type. Arrays are used to group related values together under a single variable name, making it easier to work with and manipulate multiple values as a unit. Here are some key properties and characteristics of arrays in C#:

Declaration and Initialization:

You declare an array by specifying the data type of its elements followed by square brackets [] and the array name. For example:

int[] numbers;

To initialize an array, you can use the new keyword, followed by the data type and the number of elements in the array, enclosed in square brackets:

int[] numbers = new int[5];

Size and Length:

The size of an array is the total number of elements it can hold, which is determined when the array is created. You can access the size using the Length property:

int[] numbers = new int[5];

int size = numbers.Length; // Size is 5

Zero-Based Indexing:

Arrays in C# use zero-based indexing, which means the first element is accessed with index 0, the second element with index 1, and so on.

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

int firstElement = numbers[0]; // Accessing the first element (1)
```

Initialization with Values:

You can initialize an array with values at the time of declaration using an initializer list enclosed in curly braces {}:

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

Multidimensional Arrays:

C# supports multidimensional arrays, including 2D arrays (matrices) and jagged arrays (arrays of arrays).

```
int[,] matrix = new int[3, 3]; // 2D array

int[][] jaggedArray = new int[3][]; // Jagged array
```

Array Methods and Properties:

Arrays provide useful methods and properties to work with elements, such as Length, CopyTo, Sort, Reverse, IndexOf, and ForEach.

Array Bounds:

Accessing an element outside the bounds of an array results in an IndexOutOfRangeException. It's important to ensure that your index is within the valid range of indices.

Arrays are Reference Types:

Arrays are reference types in C#, which means when you assign an array to another variable, you're actually working with a reference to the same array in memory. Modifying the array through one reference affects all references to that array.

Fixed Size:

Arrays in C# have a fixed size, meaning you cannot change the number of elements once the array is created. If you need a dynamic collection, you might consider using other data structures like List<T>.

Arrays are widely used in C# for tasks involving collections of data, such as storing elements in a list, processing data, or performing mathematical operations. Understanding arrays and their properties is fundamental for many programming tasks in C#.

Namespace

In C#, namespaces are used to organize and categorize code into logical groups. They provide a way to prevent naming conflicts between different parts of your code and help improve code maintainability and organization. Here are some key points about namespaces in C#:

Namespace Declaration:

A namespace is declared using the namespace keyword, followed by the namespace name. It typically appears at the beginning of a C# source file.

The following example declares a namespace called MyNamespace:

```
namespace MyNamespace
{
   // Code goes here
}
```

Nested Namespaces:

You can nest namespaces inside other namespaces to create a hierarchical structure for organizing your code. This helps in categorizing related code.

For example:

```
namespace OuterNamespace
{
   namespace InnerNamespace
   {
      // Code goes here
   }
}
```

Using Directives:

The using directive is used to indicate that you want to use types (classes, structs, enums, etc.) from a particular namespace without specifying the full namespace path.

For example:

```
using System; // Using the System namespace
```

Global Namespace:

The global namespace is the default namespace that contains types that are not explicitly placed in a user-defined namespace. It is often used for library code or code that doesn't belong to any specific namespace.

Types defined in the global namespace are automatically available without any using directive.

Avoiding Naming Conflicts:

Namespaces help prevent naming conflicts between types in different parts of your codebase or between your code and external libraries.

For example, the System namespace contains many commonly used types, and using using System; allows you to access them without specifying the full namespace.

Namespace Aliases:

You can create namespace aliases to simplify the use of types from long or complex namespace names.

For example:

using MyAlias = MyNamespace.InnerNamespace;

Methods in C#

In C#, methods are blocks of code that perform specific tasks or operations. They are one of the fundamental building blocks of a C# program and play a crucial role in structuring and organizing your code. Here's an overview of methods in C#:

Method Declaration:

Methods are declared within classes or structs and define their behavior. A method declaration typically includes:

- An access modifier (e.g., public, private, protected, or internal) that specifies the visibility and accessibility of the method.
- A return type that specifies the type of value the method will return (use void if the method doesn't return a value).
- The method name.
- A parameter list (enclosed in parentheses) that specifies the data the method requires to perform its task.
- The method body, enclosed in curly braces {}.

Example of a simple method declaration:

```
public int Add(int a, int b)

{

   return a + b; }
```

Interface in C#

Interfaces in C# are a fundamental concept in object-oriented programming (OOP) that allow you to define a contract for classes to follow. They play a crucial role in achieving polymorphism, code reuse, and separation of concerns within your software.

What is an Interface?

An interface in C# is a blueprint for defining a contract that classes must adhere to. It specifies a set of methods, properties, events, or indexers that implementing classes must provide. Think of an interface as a set of rules that a class must follow. By adhering to these rules, classes can ensure they have certain behaviors or capabilities.

Declaring an Interface:

To declare an interface, you use the interface keyword followed by the interface's name. Inside the interface, you define the method signatures, properties, events, or indexers that classes implementing the interface must provide. Here's a simple example:

```
public interface IShape
{
   double GetArea();
   void Display();
}
```

In this example, IShape is an interface that requires any implementing class to have two methods: GetArea and Display. These methods don't have implementations within the interface; they only define the method signatures.

Implementing an Interface:

To implement an interface, a class uses the implements (C# 8.0 and later) or : (C# 7.0 and earlier) keyword, followed by the interface name. The implementing class must provide concrete implementations for all the members defined in the interface. Let's implement the IShape interface:

```
public class Circle : IShape
{
   private double radius;

   public Circle(double r)
   {
      radius = r;
```

```
    }
    public double GetArea()
    {
        return Math.PI * radius * radius;
    }
    public void Display()
    {
        Console.WriteLine($"This is a circle with radius {radius}");
    }
}
```

In this example, the Circle class implements the IShape interface. It provides concrete implementations for the GetArea and Display methods as required by the interface. This ensures that any Circle object adheres to the IShape contract.

Delegates in C#

Delegates in C# are a versatile and fundamental feature that act as function pointers, allowing you to reference and invoke methods dynamically. They play a vital role in achieving decoupling and flexibility in code, making it possible to pass methods as parameters, store them in variables, and invoke them at runtime. Delegates are commonly used for event handling, callback mechanisms, LINQ queries, and asynchronous programming, contributing to the power and extensibility of C# applications by enabling the creation of flexible and pluggable systems.

```
using System;
// Define a delegate named MathOperation that takes two integers and returns an integer.
delegate int MathOperation(int a, int b);
class Calculator
{
    // Define methods that match the delegate signature.
    public static int Add(int x, int y)
    {
        return x + y;
    }
    public static int Subtract(int x, int y)
```

```
    {
        return x - y;
    }
}
class Program
{
    static void Main()
    {
        // Create delegate instances and associate them with methods.
        MathOperation addDelegate = Calculator.Add;
        MathOperation subtractDelegate = Calculator.Subtract;
        // Use the delegates to perform operations.
        int result1 = addDelegate(10, 5);
        int result2 = subtractDelegate(10, 5);
        // Display the results.
        Console.WriteLine("Result of addition: " + result1);
        Console.WriteLine("Result of subtraction: " + result2);
        // You can also invoke delegates using the () operator.
        int result3 = addDelegate.Invoke(20, 8);
        int result4 = subtractDelegate.Invoke(20, 8);
        Console.WriteLine("Result of addition (Invoke): " + result3);
        Console.WriteLine("Result of subtraction (Invoke): " + result4);
    }
}
```

Boxing & Unboxing

Boxing and unboxing are two operations in C# that involve the conversion between value types (such as integers, floats, and structs) and reference types (such as objects). These operations are necessary when you want to store value types in containers that expect reference types, like collections or when passing value types as objects to methods that expect reference types.

Boxing:

Boxing is the process of converting a value type to a reference type (usually an object). When you box a value type, a new object is created on the heap to hold a copy of the value, and the reference to this object is returned. Here's an example of boxing:

```
int num = 42;         // A value type (integer)

object boxedNum = num; // Boxing: Convert int to object
```

In this example, the integer 42 is boxed into an object called boxedNum. The value of num is copied into a new heap-allocated object, and boxedNum refers to that object.

Unboxing:

Unboxing is the reverse process of converting a reference type (usually an object) back to a value type. It's important to note that unboxing can only be performed if the reference type actually contains a value of the expected value type. Here's an example of unboxing:

```
int unboxedNum = (int)boxedNum; // Unboxing: Convert object back to int
```

In this example, the boxedNum object is unboxed back into an int variable called unboxedNum. The runtime checks if the object can be unboxed into an int, and if the types don't match, an InvalidCastException will be thrown.

Boxing and Unboxing Considerations:

Performance: Boxing and unboxing can have performance implications, as they involve memory allocation and type-checking. These operations are relatively slow compared to direct operations on value types.

Type Safety: Unboxing can lead to runtime errors if the unboxing type does not match the boxed type. Therefore, it's crucial to ensure that the types are compatible to avoid exceptions.

Generics: In C#, generics provide a way to work with value types without boxing. When you use generic collections like List<T> or Dictionary<TKey, TValue>, the items are not boxed when working with value types.

Value Types vs. Reference Types: Understanding the distinction between value types (e.g., structs and primitive types) and reference types (e.g., classes) is essential when dealing with boxing and unboxing. Value types are typically stored on the stack, while reference types are stored on the heap.

Here's an example that highlights the importance of type safety when unboxing:

```
object boxedValue = 42; // Boxing: Store an integer as an object

// Attempt to unbox as a different type (string)

try

{
```

```
    string unboxedString = (string)boxedValue; // InvalidCastException
}
catch (InvalidCastException ex)
{
    Console.WriteLine("InvalidCastException: " + ex.Message);
}
```
