Contents

Use of base keyword

The base keyword in C# is used in object-oriented programming when working with inheritance. It refers to the base class of a derived class and allows the derived class to access members (methods, properties, or constructors) of the base class.

a. Accessing base class field

We can use the base keyword to access the fields of the base class within derived class. It is useful if base and derived classes have the same fields. If derived class doesn't define same field, there is no need to use base keyword. Base class field can be directly accessed by the derived class.

Example:

```
using System;

public class Animal{

   public string color = "white";

}

public class Dog: Animal

{

   string color = "black";

   public void showColor()

   {

      Console.WriteLine(base.color);

      Console.WriteLine(color);

   }


}

public class TestBase

{

   public static void Main()

   {

      Dog d = new Dog();

      d.showColor();

   }

}
```

b. Accessing base class method

By the help of base keyword, we can call the base class method also. It is useful if base and derived classes defines same method. In other words, if method is overridden. If derived class doesn't define same method, there is no need to use base keyword. Base class method can be directly called by the derived class method.

Virtual keyword is used in the method of base class to represent that the method can be overridden in the derived class.

In the derived class override keyword is used to represent that the method has been overridden

Example:

```
using System;
public class Animal{
  public virtual void eat(){
    Console.WriteLine("eating...");
  }
}
public class Dog: Animal
{
  public override void eat()
  {
    base.eat();
    Console.WriteLine("eating bread...");
  }

}
public class TestBase
{
  public static void Main()
  {
    Dog d = new Dog();
    d.eat();
  }
}
```

c. Accessing base class constructor

Whenever we inherit the base class, base class constructor is internally invoked.

Example:

```
using System;
public class Animal{
    public Animal(){
        Console.WriteLine("animal...");
    }
}
public class Dog: Animal
{
    public Dog()
    {
        Console.WriteLine("dog...");
    }
}
public class TestOverriding
{
    public static void Main()
    {
        Dog d = new Dog();

    }
}
```

Complex Example: Demonstrating the implementation of payroll system:

```
using System;
public class Employee
{
    public string Name { get; set; }
    public double Salary { get; set; }
    // Base class constructor
    public Employee(string name, double salary)
    {
        Name = name;
        Salary = salary;
    }
    // Method to display employee details
    public virtual void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Salary: {Salary}");
    }
}
public class Manager : Employee
{
    public double Bonus { get; set; }
    // Derived class constructor using 'base'
    public Manager(string name, double salary, double bonus)
        : base(name, salary) // Calls the base class constructor
    {
        Bonus = bonus;
    }
    // Overriding method and using 'base'
    public override void DisplayInfo()
    {
        base.DisplayInfo(); // Calls the base class version
        Console.WriteLine($"Bonus: {Bonus}");
    }
}
class Program
```

```
{
    static void Main()
    {
        Employee emp = new Employee("Alice", 50000);
        emp.DisplayInfo();
        Console.WriteLine();
        Manager mgr = new Manager("Bob", 70000, 15000);
        mgr.DisplayInfo();
    }
}
```

Method Hiding

In method hiding, the derived class defines a new method with the same name as the base class method. The base class method is not replaced but is hidden in the derived class. Keyword ***new*** is used to hide the base class method.

Example:

```
using System;
public class Vehicle
{
    public void Move() // Base class method
    {
        Console.WriteLine("The vehicle is moving.");
    }
}
public class Car : Vehicle
{
    public new void Move() // Hides the base class method
    {
        Console.WriteLine("The car is driving.");
    }
}
class Program
```

```
{
    public static void Main()
    {
        Vehicle myVehicle = new Vehicle();
        Vehicle myCarAsVehicle = new Car();
        Car myCar = new Car();
        Console.WriteLine("Method Hiding Example:");
        myVehicle.Move();          // Calls the base class method
        myCarAsVehicle.Move();     // Calls the base class method
        myCar.Move();              // Calls the derived class method
    }
}
```

Method Overriding

If derived class defines same method as defined in its base class, it is known as method overriding in C#. It is used to achieve runtime polymorphism. It enables to provide specific implementation of the method which is already provided by its base class.

To perform method overriding in C#, we need to use ***virtual*** keyword with base class method and ***override*** keyword with derived class method.

Example:

```
using System;
public class Vehicle
{
    public virtual void Move() // Base class method marked virtual
    {
        Console.WriteLine("The vehicle is moving.");
    }
}
public class Car : Vehicle
{
    public override void Move() // Overrides the base class method
    {
        Console.WriteLine("The car is driving smoothly.");
    }
}
class Program
{
    public static void Main()
    {
        Vehicle myVehicle = new Vehicle();
        Vehicle myCarAsVehicle = new Car();
        Car myCar = new Car();
        Console.WriteLine("\nMethod Overriding Example:");
        myVehicle.Move();         // Calls the base class method
        myCarAsVehicle.Move();    // Calls the overridden method in the derived class
        myCar.Move();             // Calls the overridden method in the derived class
    }
}
```

Structure

A structure (or struct) in C# is a value type that can be used to group related variables (called fields) under one name. Unlike classes, structures are stored in memory as value types, which means they hold the actual data instead of a reference to it. Structures are commonly used for small, lightweight objects that don't need inheritance or complex behavior.

Helps to make a single variable hold related data of various data types. The ***struct*** keyword is used for creating a structure.

Example 1:

```
using System;
public struct Rectangle
{
    public int width, height;
 }
public class TestStructs
{
    public static void Main()
    {
        Rectangle r = new Rectangle();
        r.width = 4;
        r.height = 5;
        Console.WriteLine("Area of Rectangle is: " + (r.width * r.height));
    }
}
```

Example 2:

```
using System;
public struct Person
{
    // Fields in the struct
    public string Name;
    public int Age;
    // Constructor to initialize the struct
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Method to display details
    public void Display()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
}
class Program
{
    public static void Main()
    {
        // Create an instance of the struct
        Person person1 = new Person("Alice", 30);
        // Access struct fields and methods
        person1.Display();
        // Create another instance using object initializer
        Person person2 = new Person
        {
            Name = "Bob",
            Age = 25
        };
        // Display details of the second person
        person2.Display();
    }
}
```

Enums

Enum in C# is also known as enumeration. It is used to store a set of named constants such as season, days, month, size etc. The enum constants are also known as enumerators. Enum in C# can be declared within or outside class and structs.

Enum constants has default values which starts from 0 and incremented to one by one. But we can change the default value.

Keyword ***enum*** is used.

Example:

```
using System;
enum Day
{
    Sunday,    // 0
    Monday,    // 1
    Tuesday,   // 2
    Wednesday, // 3
    Thursday,  // 4
    Friday,    // 5
    Saturday   // 6
}
class Program
{
    static void Main()
    {
        // Assigning an enum value to a variable
        Day today = Day.Monday;
        // Displaying the enum value
        Console.WriteLine($"Today is: {today}"); // Output: Today is: Monday
        // Casting enum to its underlying integer value
        int dayNumber = (int)today;
        Console.WriteLine($"Numeric value of {today} is: {dayNumber}");
        // Using enum in a switch statement
        switch (today)
        {
            case Day.Monday:
                Console.WriteLine("Start of the workweek!");
                break;
            case Day.Friday:
                Console.WriteLine("End of the workweek!");
                break;
            default:
                Console.WriteLine("It's a regular day.");
                break;
        }
    }
}
```

Abstract Class

An abstract class in C# is a blueprint for other classes. It cannot be directly instantiated, which means we cannot create objects of an abstract class. Instead, it provides a foundation for other classes to build upon.

It's like a template that defines common properties and methods, but it may also contain methods that are incomplete (abstract methods). Derived classes must implement those incomplete methods.

Keyword ***abstract*** is used to define abstract class/

Key Features of Abstract Classes

1. Cannot be instantiated: You cannot create an object of an abstract class.
2. Can have abstract methods: These are methods without implementation (just a declaration).
3. Can have regular methods: These methods have a body (implementation).
4. Used for inheritance: Derived classes inherit from the abstract class and provide specific implementations.

Example:

```
using System;

// Abstract class

abstract class Animal

{

   // Abstract method (must be implemented by derived classes)

   public abstract void MakeSound();


   // Regular method (common to all animals)

   public void Eat()

   {

      Console.WriteLine("This animal is eating.");

   }

}

// Derived class

class Dog : Animal

{

   public override void MakeSound()

   {
        Console.WriteLine("Dog barks: Woof Woof!");
    }
}
// Another derived class
class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Cat meows: Meow Meow!");
    }
}
class Program
{
    static void Main()
    {
        // You cannot create an object of Animal (abstract class)
        // Animal animal = new Animal(); // This will give an error
        // Create objects of derived classes
        Animal dog = new Dog();
        Animal cat = new Cat();
        // Call methods
        dog.MakeSound(); // Output: Dog barks: Woof Woof!
        dog.Eat();       // Output: This animal is eating.
        cat.MakeSound(); // Output: Cat meows: Meow Meow!
        cat.Eat();       // Output: This animal is eating.
    }
}
```

Example for polymorphism for Extensibility(Trying to add new bird as Animal):

```
class Bird : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bird chirps: Chirp Chirp!");
    }
}
class Program
{
    static void Main()
    {
        // Using Animal as reference type
        List<Animal> animals = new List<Animal> { new Dog(), new Cat(), new Bird() };
        foreach (var animal in animals)
        {
            animal.MakeSound(); // Calls correct method based on actual object type
            animal.Eat();       // Calls common method
        }
    }
}
```

Sealed Class

In C#, a sealed class is a class that cannot be inherited. This means no other class can derive from a sealed class. It helps to prevent further modification or extension of that class, ensuring that the class behaves exactly as defined.

Main Points:

- If a class is marked as sealed, it's like putting a "No Entry" sign on it for other classes. No other class can extend (or inherit) from it.
- This is useful when we want to make sure your class cannot be changed or extended, which could potentially break its behavior.

Example:

```
// This is a sealed class
public sealed class Car
{
    public string Model { get; set; }
    public void StartEngine()
    {
        Console.WriteLine("The engine has started.");
    }
}
// This will cause an error because you cannot inherit from a sealed class
// public class ElectricCar : Car
// {
//     public void ChargeBattery()
//     {
//         Console.WriteLine("Charging the battery.");
//     }
// }
class Program
{
    static void Main()
    {
        Car myCar = new Car();
        myCar.StartEngine();  // This works fine.
    }
}
```

Interface

An interface in C# is like a contract that defines a set of methods, properties, events, or indexers that a class or struct must implement. It only provides the declaration of members, not their implementation. Think of it as a blueprint.

- No Implementation: Interfaces cannot have method bodies. They only have method signatures (declarations).
- Multiple Inheritance: A class can implement multiple interfaces, which is useful since C# doesn't support multiple inheritance with classes.
- Keyword: The interface keyword is used to declare an interface.
- Default Access Modifier: Members of an interface are public by default, and you cannot use any other access modifiers.

Example:

```
using system;

// Interface declaration

public interface IShape

{

    void Draw();               // Method declaration

    double CalculateArea();    // Another method declaration

}

// Class implementing the interface

public class Circle : IShape

{

    public double Radius { get; set; }

    public Circle(double radius)

    {

        Radius = radius;

    }

    public void Draw()

    {

        Console.WriteLine("Drawing a Circle");

    }

    public double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}
// Program to demonstrate the use of interfaces
class Program
{
    static void Main()
    {
        // Create an instance of Circle
        IShape circle = new Circle(5.0); // Radius = 5
        circle.Draw();
        double Area = circle.CalculateArea();
        Console.WriteLine("Area of Circle: {0}", Area);
    }
}
```

Polymorphism

Polymorphism in C# is the ability of a method to take on multiple forms, allowing the same interface or method to behave differently based on the object or context it operates on. It is a key principle of object-oriented programming, enabling flexibility and reusability by allowing a base class reference to invoke methods that are implemented differently in derived classes.

There are two types of polymorphism:

a. Static Polymorphism

Static polymorphism in C# is the ability of a class to resolve method calls at compile time rather than at runtime. It is achieved through method overloading or operator overloading, where multiple methods or operators have the same name but differ in the number or type of parameters. The decision about which method to call is made by the compiler based on the method signature. Static polymorphism enhances flexibility and code clarity by allowing a single method name to handle various input scenarios.

Example:

```
public class Calculator
{
    // Method to add two numbers
    public int Add(int a, int b)
    {
        return a + b;
    }
    // Method to add three numbers
    public int Add(int a, int b, int c)
    {
        return a + b + c;
    }
}
class Program
{
    static void Main()
    {
        Calculator calc = new Calculator();
        Console.WriteLine(calc.Add(5, 10));       // Calls Add with 2 parameters
        Console.WriteLine(calc.Add(5, 10, 15));  // Calls Add with 3 parameters
    }
}
```

b. Dynamic Polymorphism

Dynamic polymorphism in C# is the ability of a program to resolve method calls at runtime rather than at compile time. It is achieved through method overriding, where a derived class provides a specific implementation of a method that is already defined in its base class. The base class method is marked as virtual, and the derived class overrides it using the override keyword. When a base class reference is used to call the method, the actual method executed is determined by the runtime type of the object. This enables flexibility and allows the same code to work with objects of different types dynamically.

Example:

```
public class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("The animal makes a sound.");
    }
}
public class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("The dog barks.");
    }
}
public class Cat : Animal
{
    public override void Speak()
    {
        Console.WriteLine("The cat meows.");
    }
}
class Program
{
    static void Main()
    {
        Animal animal1 = new Dog(); // Animal reference, Dog object
        Animal animal2 = new Cat(); // Animal reference, Cat object
        animal1.Speak(); // Calls Dog's Speak method
        animal2.Speak(); // Calls Cat's Speak method
    }
}
```

Delegate

A delegate in C# is like a pointer to a function. It allows passing methods as arguments to other methods. Delegates are used to encapsulate a reference to a method inside a delegate object. This makes it possible to call a method without knowing which method will be invoked at compile time.

Delegate is like a template that describes what a method should look like, including its return type and the type of its input. We can assign any method that matches this description to the delegate, making it easy to call that method later. Delegates are especially useful for handling events, running specific actions when something happens, or creating custom features, as they can connect methods to tasks in a flexible way.

Example:

```
using System;

class Program

{

    // Step 1: Declare a delegate

    delegate int MathOperation(int a, int b);

    // Step 2: Create methods matching the delegate signature

    static int Add(int x, int y)

    {

        return x + y;

    }

    static int Multiply(int x, int y)

    {

        return x * y;

    }

    static void Main()

    {

        // Step 3: Assign methods to the delegate

        MathOperation operation;

        operation = Add; // Assign Add method to the delegate

        Console.WriteLine("Addition: " + operation(5, 3)); // Output: 8

        operation = Multiply; // Assign Multiply method to the delegate

        Console.WriteLine("Multiplication: " + operation(5, 3)); // Output: 15

    }

}
```

Multicast Delegate

A multicast delegate allows to assign multiple methods to the same delegate. When the delegate is invoked, all the methods it points to are called in the order they were assigned. This is useful for scenarios like event handling, where multiple methods need to respond to the same event.

A multicast delegate differs from a regular delegate because it can hold references to multiple methods instead of just one. When a multicast delegate is called, all the methods assigned to it are executed in sequence. We can easily add methods to a multicast delegate using the += operator and remove them using the -= operator.

Example:

```
using System;

class Program

{

    // Declare a delegate type

    delegate void MessageHandler(string message);

    // Methods matching the delegate signature

    static void ShowMessage(string message)

    {

        Console.WriteLine("Message: " + message);

    }

    static void LogMessage(string message)

    {

        Console.WriteLine("Logging: " + message);

    }

    static void Main()

    {

        // Create a multicast delegate and assign multiple methods

        MessageHandler handler = ShowMessage;  // Assign the ShowMessage method

        handler += LogMessage;                 // Add the LogMessage method to the delegate

        // Invoke the multicast delegate

        handler("Hello, World!");

    }

}
```

Event

An event in C# is a mechanism that enables a class to notify other classes or objects when something of interest happens. It is based on delegates, but it provides an additional layer of security, ensuring that the event can only be triggered from within the class that declares it.

Events help keep things secure by not letting code outside the class directly call the delegate, giving the class full control over how it works. They follow a publisher-subscriber model, where the class that raises the event is called the publisher, and the ones that react to it are the subscribers. This setup is common in programs like graphical user interfaces (GUIs) and real-time systems, where quick and dynamic responses are important.

Example: A button that triggers an event when clicked

```
using System;

class Button

{

    // Step 1: Declare a delegate for the event

    public delegate void ClickEventHandler();

    // Step 2: Declare the event using the delegate

    public event ClickEventHandler Click;

    // Method to simulate the button click

    public void OnClick()

    {

        Console.WriteLine("Button clicked!");

        // Step 3: Raise the event

        if (Click != null)

        {

            Click(); // Notify all subscribers

        }

    }

}

class Program

{

    static void Main()

    {

        // Create an instance of the Button

        Button myButton = new Button();
        // Step 4: Subscribe to the event
        myButton.Click += OnButtonClicked;
        // Simulate clicking the button
        myButton.OnClick();
    }
    // Event handler (method to respond to the event)
    static void OnButtonClicked()
    {
        Console.WriteLine("The button was clicked. Event handled!");
    }
}
```

Partial Class

In C#, a partial class allows to divide the definition of a class across multiple files, which is helpful when working with large classes or code that is automatically generated. This feature is particularly useful in scenarios like designing Windows Forms or WPF applications, where parts of the code are generated by tools.

All parts of a partial class must be marked with the partial keyword, and even though the class is split across different files, the C# compiler treats it as a single class. This approach is commonly used in situations where we need to separate custom code from automatically generated code, like in code generation tasks.

Collections

In C#, collections are special data structures used to store and manage groups of related objects. Unlike arrays, which have a fixed size, collections are dynamic and can grow or shrink as needed, making them more flexible. They also come with built-in methods to add, remove, search, and sort elements, which simplifies data handling.

Example:

```
using System;
using System.Collections;
class Program
{
    static void Main()
    {
        ArrayList mixedData = new ArrayList();
        // Adding different data types
        mixedData.Add(42);             // Integer
        mixedData.Add("Hello, World"); // String
        mixedData.Add(3.14);           // Double
        mixedData.Add(true);           // Boolean
        // Displaying all elements
        foreach (var item in mixedData)
        {
            Console.WriteLine(item);
        }
    }
}
```

Generics

Commonly Used Generic Collections:

a. List<Type>

- A dynamic array that can grow or shrink.
- Stores elements of the same type.

Example:

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        List<int> numbers = new List<int>(); // Create a list of integers
        numbers.Add(10);  // Add elements
        numbers.Add(20);
        numbers.Add(30);
        Console.WriteLine("List items:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);  // Access elements
        }
        numbers.Remove(20);  // Remove an element
        Console.WriteLine("After removing 20:");
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}
```

b. Dictionary<TKey, TValue>

- A collection of key-value pairs.
- Keys must be unique.

Example:

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        Dictionary<int, string> students = new Dictionary<int, string>();
        students.Add(1, "Alice");
        students.Add(2, "Bob");
        students.Add(3, "Charlie");
        Console.WriteLine("Student Names:");
        foreach (KeyValuePair<int, string> student in students)
        {
            Console.WriteLine($"ID: {student.Key}, Name: {student.Value}");
        }
        // Access value using a key
        Console.WriteLine($"Student with ID 2: {students[2]}");
    }
}
```

c. Queue<Type>

- A collection that works in a First-In-First-Out (FIFO) manner.

Example:

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        Queue<string> tasks = new Queue<string>();
        tasks.Enqueue("Task 1"); // Add to the queue
        tasks.Enqueue("Task 2");
        tasks.Enqueue("Task 3");
        Console.WriteLine("Processing tasks:");
        while (tasks.Count > 0)
        {
            Console.WriteLine(tasks.Dequeue()); // Remove and process the first item
        }
    }
}
```

d. Stack<Type>

- A collection that works in a Last-In-First-Out (LIFO) manner.

Example:

```
using System;

using System.Collections.Generic;

class Program

{

    static void Main()

    {

        Stack<string> books = new Stack<string>();

        books.Push("Book A"); // Add to the stack

        books.Push("Book B");

        books.Push("Book C");

        Console.WriteLine("Books in stack:");

        while (books.Count > 0)

        {

            Console.WriteLine(books.Pop()); // Remove and process the last item

        }

    }

}
```

LINQ

LINQ (Language Integrated Query) is a feature in C# that allows to easily query and work with data from different sources, such as lists, arrays, databases, and XML, directly within C# code. It makes querying data simpler by writing queries in C# itself, instead of using SQL or other query languages. LINQ offers two main ways to write queries: Query Syntax, which is similar to SQL and is easy for beginners to understand, and Method Syntax, which is more flexible and uses extension methods like Where(), Select(), and others. Another key feature of LINQ is Deferred Execution, meaning that the query doesn't run until it actually needs the data, such as when you loop through the results or convert them to a list. Additionally, LINQ allows to create Anonymous Types, which are temporary objects that don't require to define a class first. This makes LINQ a powerful and easy-to-use tool for working with data in C#.

Examples: Filtering Even Numbers

```
using System;
public class Program
{
    public static void Main()
    {
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Query Syntax
        var evenNumbersQuery = from num in numbers
                               where num % 2 == 0
                               select num;
        Console.WriteLine("Even Numbers (Query Syntax):");
        foreach (var num in evenNumbersQuery)
        {
            Console.WriteLine(num);
        }
        // Method Syntax
        var evenNumbersMethod = numbers.Where(num => num % 2 == 0);
        Console.WriteLine("\nEven Numbers (Method Syntax):");
        foreach (var num in evenNumbersMethod)
        {
            Console.WriteLine(num);
        }
    }
}
```

Example 2: Grouping data by grade

```
public class Student
{
    public string Name { get; set; }
    public int Score { get; set; }
}
public class Program
{
    public static void Main()
    {
        List<Student> students = new List<Student>
        {
            new Student { Name = "Alice", Score = 85 },
            new Student { Name = "Bob", Score = 92 },
            new Student { Name = "Charlie", Score = 60 },
            new Student { Name = "David", Score = 75 }
        };
        var groupedByGrade = from student in students
                             group student by student.Score >= 80 ? "A" : "B" into gradeGroup
                             select new
                             {
                                 Grade = gradeGroup.Key,
                                 Students = gradeGroup
                             };
        foreach (var group in groupedByGrade)
        {
            Console.WriteLine(group.Grade + ":");
            foreach (var student in group.Students)
            {
                Console.WriteLine(student.Name);
            }
        }
    }
}
```

Lambda Expression

A lambda expression in C# is a concise way to write anonymous methods (methods without a name) using the => operator. It allows you to define inline functionality, making code more readable and compact. Lambda expressions are commonly used with delegates, LINQ queries, and event handling. They are particularly useful for performing operations like filtering, sorting, and transforming collections. For instance, instead of defining a separate method to double numbers in a list, you can use a lambda expression directly.

Example: Filtering the price

```
List<int> prices = new List<int> { 50, 120, 80, 200 };

var expensiveProducts = prices.Where(price => price > 100);

Console.WriteLine("Expensive Products:");

foreach (var price in expensiveProducts)

{

   Console.WriteLine(price);

}
```

Exception Handling

Exception handling in C# is a way to handle errors that happen while the program is running. It ensures the program can keep working or work properly without crashing even if any error have occurred. This is done using four main keywords: try, catch, finally, and throw.

Keywords in Exception Handling:

- try:

Contains the code that might cause an error.

- catch:

Deals with errors that happen in the try block. You can specify the type of error to handle specific issues.

- finally:

Runs code that should always execute, whether an error happened or not (e.g., cleaning up resources).

- throw:

Used to create and trigger an error on purpose.

Basic Syntax:

```
try

{

   // Code that may cause an exception

}
```

```
catch (ExceptionType e)
{
    // Code to handle the exception
}
finally
{
    // Code that will always execute
}
```

Example 1: Demonstration of handling divide by zero expression

```
using System;
class Program
{
    static void Main()
    {
        try
        {
            Console.WriteLine("Enter a number: ");
            int number = int.Parse(Console.ReadLine());
            Console.WriteLine("Enter a divisor: ");
            int divisor = int.Parse(Console.ReadLine());
            int result = number / divisor;
            Console.WriteLine($"Result: {result}");
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("Error: You cannot divide by zero.");
        }
        catch (FormatException ex)
        {
            Console.WriteLine("Error: Please enter a valid number.");
        }
        finally
        {
            Console.WriteLine("Execution completed.");
        }
    }
}
```

Example 2: Demonstration of custom/user defined exception exception.

While defining a custom exception, the custom exception should inherit from the base class Exception.

```
using System;
// Custom exception class
public class InvalidAgeException : Exception
{
    public InvalidAgeException() { }
    public InvalidAgeException(string message) : base(message) { }
}
class Program
{
    static void Main()
    {
        try
        {
            int age = -5;
            if (age < 0)
            {
                throw new InvalidAgeException("Age cannot be negative.");
            }
        }
        catch (InvalidAgeException ex)
        {
            Console.WriteLine($"Custom Exception Caught: {ex.Message}");
        }
    }
}
```

Example 3: Implementation of predefined exception using throw keyword

```
using System;
class Program
{
    static void Main()
    {
        try
        {
            int age = -5;
            if (age < 0)
            {
                throw new ArgumentException("Age cannot be negative.");
            }
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine($"Custom Exception Caught: {ex.Message}");
        }
    }
}
```