Day 06

01 Object Members overview

02 Generic

very similar to c++ Templates

```
int Add(int u , int v){
  return u+v;
}

//Generic Function
T Add<T>(T u , T v){
  return u+v;
}

//Generic Class

class Dummy<T>{

T Get() return ch;

void Set(T ch){
  this.ch=ch;
}

}

Main(){
  Dummy<int> d1=new Dummy<int>();
}
```

03 Exception Handling

01 Exception Handling

Exception handling is an important part of writing robust and reliable software in C#. It allows you to handle runtime errors and unexpected conditions that may occur during program execution, instead of crashing the application or displaying cryptic error messages to the user.

In C#, exception handling is typically done using the `try-catch-finally` block. Here's an example:

```
try {
    // some code that may throw an exception
}
catch (Exception ex) {
    // handle the exception
    Console.WriteLine($"An error occurred: {ex.Message}");
}
finally {
    // cleanup code that always runs, regardless of whether an exception occurred or not
}
```

In this example, the `try` block contains the code that may throw an exception. If an exception is thrown, the `catch` block is executed, which contains code to handle the exception. In this case, we're simply printing the error message to the console, but you could also log the error, display a user-friendly error message, or take other appropriate action.

The `finally` block contains code that is always executed, regardless of whether an exception was thrown or not. This block is used to perform any necessary cleanup tasks, such as closing open files or releasing resources.

You can also catch specific types of exceptions by specifying the exception type in the catch block, like this:

```
try {
    // some code that may throw an exception
}
catch (ArgumentException ex) {
    // handle ArgumentException
}
catch (InvalidOperationException ex) {
    // handle InvalidOperationException
}
catch (Exception ex) {
    // handle all other types of exceptions
}
finally {
    // cleanup code that always runs, regardless of whether an exception occurred or not
}
```

In this example, we're catching and handling specific types of exceptions, and then using a catch block for all other types of exceptions (using the base `Exception` class).

▼ sir's Code

```
using System;

namespace csharp_programming {
    class FundException : Exception {
        public FundException(decimal balance)
            : base("Insufficient funds. Total funds are: " + balance) {
        }
    }

    class Account {
        //Field
        decimal balance;

        //property
        public decimal Balance {
            get { return balance; }
        }

        //Methods
        public void Deposit(decimal amount) {
            if (amount < 0)
                throw new ArgumentException("Amount cannot be negative");
            balance += amount;
        }
        public void Withdraw(decimal amount) {
            if (amount < 0)
                throw new ArgumentException("Amount cannot be negative");
            if (amount > balance)
                throw new FundException(balance);
            balance -= amount;
        }
    }

    class Program {
        static Account account = null;

        static void Main() {
            try {
                string input;
                do {
                    Console.Clear();
                    if (account == null)
                        Console.WriteLine("Account: Not Opened.");
                    else {
                        Console.WriteLine("Account: Opened. Balance: {0}", account.Balance);
                    }
                    Console.WriteLine();
                    Console.WriteLine("1. Open New Account.");
                    Console.WriteLine("2. Deposit Money.");
                    Console.WriteLine("3. Withdraw Money.");
                    Console.WriteLine("4. Close Account.");
                    Console.WriteLine("5. Exit.");
                    Console.Write("Enter your choice: ");
                    input = Console.ReadLine();
                    switch (input) {
                        case "1":
                            OpenNewAccount();
                            break;
                        case "2":
                            Console.WriteLine();
                            try {
                                Console.Write("Enter Amount to Deposit: ");
                                input = Console.ReadLine();
                                account.Deposit(Convert.ToDecimal(input));
                            }
                            catch (ArgumentException e) {
                                Console.WriteLine(e.Message);
                            }
                            finally {
                                Console.WriteLine("Press any key to continue...");
                                Console.ReadKey();
                            }
                            break;
                        case "3":
                            Console.WriteLine();
                            try {
                                Console.Write("Enter Amount to Withdraw: ");
                                input = Console.ReadLine();
                                account.Withdraw(Convert.ToDecimal(input));
                            }
                            catch (FundException e) {
                                Console.WriteLine(e.Message);
                            }
                            catch (ArgumentException e) {
                                Console.WriteLine(e.Message);
                            }
                            finally {
                                Console.WriteLine("Press any key to continue...");
                                Console.ReadKey();
                            }
                            break;
                        case "4":
                            CloseAccount();
                            break;
                        case "5":
                            break;
                        default:
                            Console.WriteLine("Invalid Operation. Press any key to continue...");
                            Console.ReadKey();
                            break;
                    }
                } while (input != "5");
            }
            finally {
                if (account != null)
                    Console.WriteLine("Final Balance: {0}", account.Balance);
                CloseAccount();
            }
        }

        public static void OpenNewAccount() {
            account = new Account();
        }

        public static void CloseAccount() {
            account = null;
        }
    }
}
```

02 Exception Filter

In C#, an exception filter is a clause that can be added to a catch block to further specify which exceptions the catch block should handle. An exception filter is an expression that returns a Boolean value. If the expression evaluates to true, the catch block is executed. If the expression evaluates to false, the catch block is skipped and the exception is propagated to the next catch block (if any) or the calling method.

Here's an example of using an exception filter:

```
try {
    // some code that may throw an exception
}
catch (Exception ex) when (ex is InvalidOperationException) {
    // handle InvalidOperationException
}
```

In this example, the catch block will only handle exceptions of type `InvalidOperationException`. If any other type of exception is thrown, it will not be caught by this block and will propagate to the next catch block (if any) or the calling method.

Exception filters can also use any valid expression that returns a Boolean value, not just `is` or `as` type checks. For example:

```
try {
    // some code that may throw an exception
}
catch (Exception ex) when (ex.Message.Contains("File not found")) {
    // handle file not found exception
}
```

In this example, the catch block will only handle exceptions whose error message contains the string "File not found".

Using exception filters can make your code more precise and targeted in handling specific exceptions. However, they should be used sparingly and only when necessary, as they can make the code harder to read and understand.

04 Attributes

Attributes are annotation or metadata that provide additional information about types , methods , properties , parameter and other program element.

In C#, an attribute is a declarative tag or annotation that provides additional information about a type, method, field, property, event, or parameter in your code. Attributes are used to specify various aspects of the code, such as how the code should be compiled or executed, how it should be accessed by other code, or how it should be treated by various tools or frameworks.

Attributes are defined using square brackets ([]), followed by the name of the attribute class, and any parameters that the attribute class requires. For example, the following code shows how to define an attribute called "MyAttribute":

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class MyAttribute : Attribute
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

This attribute can then be applied to a class or method in the code, as shown below:

```
[My(Name = "John", Age = 30)]
public class MyClass
{
    // ...
}
```

In this example, the "MyAttribute" attribute is applied to the "MyClass" class, with the "Name" and "Age" properties set to specific values. Other code can then use reflection to read the attribute values and take appropriate actions based on them.

C# includes several built-in attributes, such as "Obsolete" (which marks code as deprecated), "Serializable" (which indicates that an object can be serialized), and "DllImport" (which specifies that a method is implemented in unmanaged code). You can also define your own custom attributes to suit the needs of your code.

▼ Sir’s Code

```
//Person.cs
namespace cshar_programming
{
    [Comment(Text ="Person Class")]
    internal class Person
    {
        [Comment(Text ="person Name")]
        public string Name { get; set; }

        [Comment(Text ="person age")]
        public int Age { get; set; }

        [Comment(Text ="Person Marital Status")]
        public bool Married { get; set; }

        [Comment(Text ="Is person graduate or undergraduate")]
        public bool IsGraduate { get; set; }

        public override string ToString()
        {
            return string.Format("name={0} , Age={1}, Married={2} , Graduated={3} ", Name, Age, Married, IsGraduate);
        }
    }
}


//new file
//CommentAttribute.cs
namespace cshar_programming
{
    [AttributeUsage(AttributeTargets.All,AllowMultiple=true , Inherited =true)]
    internal class CommentAttribute:Attribute
    {
        public string Text { get; set; }
    }
}
```

05 Nullble Types

A nullable variable in C# is a variable that can hold a value of its underlying value type or a null value. It is defined by appending a question mark "?" to the end of the variable's type.

For example, if you want to declare a nullable integer variable, you can write:

```
int? nullableInt = null;
```

Here, the "nullableInt" variable can hold an integer value or a null value. To assign a value to it, you can use the regular assignment operator:

```
nullableInt = 42;
```

If you want to check if a nullable variable has a value, you can use the "HasValue" property:

```
if (nullableInt.HasValue)
{
    int value = nullableInt.Value;
    // Do something with the value...
}
else
{
    // The variable is null...
}
```

If the variable has a value, you can get its value using the "Value" property. However, if the variable is null and you try to access its value, a "System.InvalidOperationException" will be thrown.

You can also use the null-coalescing operator ("??") to provide a default value in case the nullable variable is null. For example:

```
int value = nullableInt ?? 0;
```

Here, the "value" variable will be set to the value of "nullableInt", or 0 if "nullableInt" is null.

Nullable variables are useful when you need to represent the absence of a value, such as in database programming, where null values are often used to represent missing or unknown data. By using nullable variables, you can avoid having to use special sentinel values (such as -1 or DateTime.MinValue) to represent null values.

06 Difference Betn struct & Class

In C#, both structs and classes are used to define custom data types, but they have some important differences in terms of how they are stored, copied, and used in your code.

1. Memory allocation:
   A struct is a value type, which means that it is stored directly on the stack and allocated inline with the containing object. Whereas, a class is a reference type, which means that it is stored on the heap and only a reference to the object is stored on the stack.
2. Assignment and copy behavior:
   When you assign a struct variable to another struct variable, a copy of the entire struct is created. This means that changes to one variable do not affect the other variable. In contrast, when you assign a class variable to another class variable, only a reference to the object is copied. This means that both variables point to the same object, and changes to one variable will affect the other variable.
3. Inheritance:
   A struct cannot inherit from another struct or class, and it cannot be used as a base type. However, a class can inherit from another class, and it can also implement interfaces.
4. Default initialization:
   When you create a new struct, all of its fields are automatically initialized to their default values (e.g. 0 for numeric types). In contrast, when you create a new class, all of its fields are initialized to null (or zero for value types that are declared inside the class).
5. Performance:
   Structs can be faster than classes in some situations because they are stored directly on the stack and don't require heap allocation. However, this can also lead to larger memory usage in some situations and may result in more copying of values.

In general, you should use a struct when you have a small, simple data type that can be copied efficiently, and you don't need inheritance or reference semantics. Use a class when you need to represent more complex data, you need inheritance or reference semantics, or you need to share the object across multiple parts of your code.