Steps for gui form

1)Open new project name as “gui form 1.cs”

2) form toolbox drop label to gui form1 .cs

3)name label1 as “login here”

4) drop another label and name as user name

5)drop input box for user name and variable name

As text 1

6) drop another label and name as password

7)drop input box for password and variable name

As text box 2

8)drag 2 buttons “cancel and login “respectively

Q1how to connect database with c# program are givern bellow

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using MySql.Data.MySqlClient;

namespace WindowsFormsApplication1
{
```

```
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }

      private void Form1_Load(object sender, EventArgs e)
      {
         string mysqlconnectString = "datasource=127.0.0.1;port=3306;username=root;password;
database=info";
         MySqlConnection databasecon = new MySqlConnection(mysqlconnectString);
         try
         {
            databasecon.Open();
            MessageBox.Show("Database ok");
         }
         catch
         {
            MessageBox.Show("Database Error");
         }
      }
   }
}
```

Q2) how to made simple log in form (gui)

Ans: Steps for gui form

1)Open new project name as “gui form 1.cs”

2) form toolbox drop label to gui form1 .cs

3)name label1 as “login here”

4) drop another label and name as user name

5)drop input box for user name and variable name

As text 1

6) drop another label and name as password

7)drop input box for password and variable name

As text box 2

8)drag 2 buttons “cancel and login “respectively

Program are given bellow simple form

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace roomgui
{
   public partial class Form1 : Form
   {
```

```
public Form1()
{
   InitializeComponent();
}

private void label1_Click(object sender, EventArgs e)
{

}

private void textBox1_TextChanged(object sender, EventArgs e)
{

}

private void textBox2_TextChanged(object sender, EventArgs e)
{

}

private void button2_Click(object sender, EventArgs e)
{
   if (textBox1.Text == "Admin" && textBox2.Text == "pass")
   {
      this.Hide();
      Home room = new Home();
      room.Show();
```

```
        }
        else
        {
            MessageBox.Show("Username/Password Incorrect");
        }
    }

    private void button1_Click(object sender, EventArgs e)
    {
        this.Close();
    }
  }
}
```

Q3) how to create login form and connect to data base

Ans

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace roomgui
```

```
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void label1_Click(object sender, EventArgs e)
        {

        }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {

        }

        private void textBox2_TextChanged(object sender, EventArgs e)
        {

        }

        private void button2_Click(object sender, EventArgs e)
        {
            string mysqlconnectString =
"datasource=127.0.0.1;port=3306;username=root;password;  database=info";
```

```
                MySqlConnection databasecon = new MySqlConnection(mysqlconnectString);
                try
                {
                   databasecon.Open();
                   MessageBox.Show("Database ok");
                   String user = textBox1.Text;
                   String pass = textBox1.Text;

                   Query = "Select *(count) from employee where username = 'user' and
password ='pass'";
                   if (Query) {
                      this.Hide();
                      Home room = new Home();
                      room.Show();
                   }
                   else
                   {
                      MessageBox.Show("Username/Password Incorrect");
                   }

                }
                catch
                {
                   MessageBox.Show("Database Error");
                }

             }
```

```
        private void button1_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }
}
```

Class example

Example

Create an object called "myObj" and use it to print the value of color:

```
class Car {
  string color = "red";
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

C# - What is OOP?

OOP stands for Object-Oriented Programming.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter developmen

C# - What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

Class Members

Fields and methods inside classes are often referred to as "Class Members":

Example

Create a `Car` class with three class members: **two fields** and **one method**.

```
// The class
class MyClass
{
  // Class members
  string color = "red";        // field
  int maxSpeed = 200;          // field
  public void fullThrottle()   // method
  {
    Console.WriteLine("The car is going as fast as it can!");
```

```
  }
}
```

Use Multiple Classes

Remember from the last chapter, that we can use multiple classes for better organization (one for fields and methods, and another one for execution). This is recommended:

Car.cs

```
class Car
{
  public string model;
  public string color;
  public int year;
  public void fullThrottle()
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

Program.cs

```
class Program
{
  static void Main(string[] args)
  {
    Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
```

```
    Ford.year = 1969;


    Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;


    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
Important for papper
```

Run example »

The `public` keyword is called an **access modifier**, which specifies that the fields of `Car` are accessible for other classes as well, such as `Program`.

You will learn more about Access Modifiers in a later chapter.

Constructors

A constructor is a **special method** that is used to initialize objects. The advantage of a constructor, is that it is called when an object of a class is created. It can be used to set initial values for fields:

Example

Create a constructor:

```
// Create a Car class
```

```
class Car

{

  public string model;  // Create a field


  // Create a class constructor for the Car class

  public Car()

  {

    model = "Mustang"; // Set the initial value for model

  }


  static void Main(string[] args)

  {

    Car Ford = new Car();  // Create an object of the Car Class (this
will call the constructor)

    Console.WriteLine(Ford.model);  // Print the value of model

  }

}


// Outputs "Mustang"

Important for paper
```

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like `void` or `int`).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, C# creates one for you. However, then you are not able to set initial values for fields.

```
```

Access Modifiers

By now, you are quite familiar with the `public` keyword that appears in many of our examples:

```
public string color;
```

The `public` keyword is an **access modifier**, which is used to set the access level/visibility for classes, fields, methods and properties.

C# has the following access modifiers:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the same class
`protected`	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter

`internal`	The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

---

Private Modifier

If you declare a field with a `private` access modifier, it can only be accessed within the same class:

Example

```
class Car
{
  private string model;

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang");
    Console.WriteLine(Ford.model);
  }
}
```

The output will be:

```
Mastang
```

---

public Modifier

If you declare a field with a `public` access modifier, it is accessible for all classes:

Example

```
class Car
{
  public string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
Mustang
```

Run example »

Why Access Modifiers?

To control the visibility of class members (the security level of each individual class and class member).

To achieve "**Encapsulation**" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as `private`. You will learn more about this in the next chapter.

Note: By default, all members of a class are `private` if you don't specify an access modifier:

Example

```
class Car
{
  string model;  // private
  string year;   // private
}
```

Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- **Derived Class** (child) - the class that inherits from another class
- **Base Class** (parent) - the class being inherited from

To inherit from a class, use the `:` symbol.

In the example below, the `Car` class (child) inherits the fields and methods from the `Vehicle` class (parent):

Example

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()             // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
  }
```

```
}


class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}


class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();


    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();


    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
   Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

Run example »

Why And When To Use "Inheritance"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Polymorphism and Overriding Methods

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called `Animal` that has a method called `animalSound()`. Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}
class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
```

```
  }
}
class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

```
```

Abstract Classes and Methods

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The `abstract` keyword is used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal
{
  public abstract void animalSound();
  public void sleep()
  {
    Console.WriteLine("Zzz");
  }
}
```

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Note: Abstraction can also be achieved with Interfaces, which you will learn more about in the next chapter.

Interfaces

Another way to achieve abstraction in C#, is with interfaces.

An `interface` is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies):

Example

```
// interface
interface Animal
{
  void animalSound(); // interface method (does not have a body)
  void run(); // interface method (does not have a body)
}
```

It is considered good practice to start with the letter "I" at the beginning of an interface, as it makes it easier for yourself and others to remember that it is an interface and not a class.

By default, members of an interface are `abstract` and `public`.

Note: Interfaces can contain properties and methods, but not fields.

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class. To implement an interface, use the `:` symbol (just like with inheritance). The body of the interface method is provided by the "implement" class. Note that you do not have to use the `override` keyword when implementing an interface:

Example

```
// Interface
interface IAnimal
{
  void animalSound(); // interface method (does not have a body)
}

// Pig "implements" the IAnimal interface
class Pig : IAnimal
{
  public void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}
```

```
class Program
{
  static void Main(string[] args)
  {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
  }
}
```

Run example »

Notes on Interfaces:

- Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables
- Interface members are by default `abstract` and `public`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) C# does not support "multiple inheritance" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the

class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface IFirstInterface
{
  void myMethod(); // interface method
}

interface ISecondInterface
{
  void myOtherMethod(); // interface method
}

// Implement multiple interfaces
class DemoClass : IFirstInterface, ISecondInterface
{
  public void myMethod()
  {
    Console.WriteLine("Some text..");
  }
  public void myOtherMethod()
  {
```

```
    Console.WriteLine("Some other text...");
  }
}


class Program
{
  static void Main(string[] args)
  {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

C# Enums

An `enum` is a special "class" that represents a group of **constants** (unchangeable/read-only variables).

To create an `enum`, use the `enum` keyword (instead of class or interface), and separate the enum items with a comma:

Example

```
enum Level
{
  Low,
  Medium,
  High
```

```
}
```

You can access `enum` items with the **dot** syntax:

```
Level myVar = Level.Medium;

Console.WriteLine(myVar);

Note

Enum is short for "enumerations", which means "specifically listed"
```

Enum Values

By default, the first item of an enum has the value 0. The second has the value 1, and so on.

To get the integer value from an item, you must explicitly convert the item to an `int`:

Example

```
enum Months

{

  January,    // 0

  February,   // 1

  March,      // 2

  April,      // 3

  May,        // 4

  June,       // 5

  July        // 6

}


static void Main(string[] args)
```

```
{
  int myNum = (int) Months.April;
  Console.WriteLine(myNum);
}
```

The output will be:

Why And When To Use Enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

 # Exceptions

When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an **exception** (throw an error).

C# try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

Syntax

```
try
{
  //  Block of code to try
}
```

```
catch (Exception e)
{
  //  Block of code to handle errors
}
```

finally

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
finally
{
  Console.WriteLine("The 'try catch' is finished.");
}
```

The throw keyword

The `throw` statement allows you to create a custom error.

The `throw` statement is used together with an **exception class**. There are many exception classes available in C#: `ArithmeticException`, `FileNotFoundException`, `IndexOutOfRangeException`, `TimeOutException`, etc:

Example

```
static void checkAge(int age)
{
  if (age < 18)
  {
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  }
  else
  {
    Console.WriteLine("Access granted - You are old enough!");
  }
}

static void Main(string[] args)
{
  checkAge(15);
}
```

Process and Thread

A process represents an application whereas a thread represents a module of the application. Process is heavyweight component whereas thread is lightweight. A thread can be termed as lightweight subprocess because it is executed inside a process.

C# Multithreading

Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C#, we need to use **System.Threding** namespace.

System.Threading Namespace

The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used classes are given below:

- Thread
- Mutex
- Timer
- Monitor
- Semaphore
- ThreadLocal
- ThreadPool
- Volatile etc.

C# Multithreading

Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C#, we need to use **System.Threding** namespace.

System.Threading Namespace

The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used classes are given below:

- Thread
- Mutex
- Timer
- Monitor
- Semaphore
- ThreadLocal
- ThreadPool
- Volatile etc.

Unstarted State

When the instance of Thread class is created, it is in unstarted state by default.

Runnable State

When start() method on the thread is called, it is in runnable or ready to run state.

Running State

Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.

Not Runnable State

The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.

Dead State

After completing the task, thread enters into dead or terminated state.

C# Threading Example: ThreadPriority

Let's see an example where we are changing the priority of the thread. The high priority thread can be executed first. But it is not guaranteed because thread is highly system dependent. It increases the chance of the high priority thread to execute before low priority thread.

```
using System;
using System.Threading;
public class MyThread
{
   public void Thread1()
   {
      Thread t = Thread.CurrentThread;
      Console.WriteLine(t.Name+" is running");
   }
}
public class ThreadExample
{
   public static void Main()
   {
      MyThread mt = new MyThread();
      Thread t1 = new Thread(new ThreadStart(mt.Thread1));
      Thread t2 = new Thread(new ThreadStart(mt.Thread1));
      Thread t3 = new Thread(new ThreadStart(mt.Thread1));
      t1.Name = "Player1";
      t2.Name = "Player2";
      t3.Name = "Player3";
      t3.Priority = ThreadPriority.Highest;
      t2.Priority = ThreadPriority.Normal;
      t1.Priority = ThreadPriority.Lowest;

      t1.Start();
      t2.Start();
      t3.Start();
   }
```

```
}
```

Output:

The output is unpredictable because threads are highly system dependent. It may follow any algorithm preemptive or non-preemptive.

C# Inheritance

In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

Advantage of C# Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

C# Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
using System;
  public class Employee
   {
     public float salary = 40000;
  }
  public class Programmer: Employee
```

```
{
   public float bonus = 10000;
}
class TestInheritance{
   public static void Main(string[] args)
    {
        Programmer p1 = new Programmer();

        Console.WriteLine("Salary: " + p1.salary);
        Console.WriteLine("Bonus: " + p1.bonus);

    }
}
```

Output:

```
Salary: 40000
Bonus: 10000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C# Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation and polymorphism.

There are two types of polymorphism in C#: compile time polymorphism and runtime polymorphism. Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding. Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

C# Runtime Polymorphism Example

Let's see a simple example of runtime polymorphism in C#.

```
using System;

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

}
public class TestPolymorphism
{
   public static void Main()
   {
      Animal a= new Dog();
      a.eat();
   }
}
```

Output:

```
eating bread...
```

C# Member Overloading

If we create two or more members having same name but different in number or type of parameter, it is known as member overloading. In C#, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

C# Method Overloading

Having two or more methods with same name but different in parameters, is known as method overloading in C#.

The **advantage** of method overloading is that it increases the readability of the program because you don't need to use different names for same action.

You can perform method overloading in C# by two ways:

1. By changing number of arguments
2. By changing data type of the arguments

C# Method Overloading Example: By changing no. of arguments

Let's see the simple example of method overloading where we are changing number of arguments of add() method.

```
using System;
public class Cal{
    public static int add(int a,int b){
        return a + b;
    }
    public static int add(int a, int b, int c)
    {
        return a + b + c;
    }
}
public class TestMemberOverloading
{
    public static void Main()
    {
        Console.WriteLine(Cal.add(12, 23));
        Console.WriteLine(Cal.add(12, 23, 25));
    }
}
```

Output:

```
35
```
60

Member Overloading Example: By changing data type of arguments

Let's see the another example of method overloading where we are changing data type of arguments.

```
using System;
public class Cal{
    public static int add(int a, int b){
        return a + b;
    }
    public static float add(float a, float b)
    {
        return a + b;
    }
}
public class TestMemberOverloading
{
    public static void Main()
    {
        Console.WriteLine(Cal.add(12, 23));
        Console.WriteLine(Cal.add(12.4f,21.3f));
    }
}
```

Output:

```
35
33.7
```

C# 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 you to provide specific implementation of the method which is already provided by its base class.

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

C# Method Overriding Example

Let's see a simple example of method overriding in C#. In this example, we are overriding the eat() method by the help of override keyword.

```
using System;
public class Animal{
    public virtual void eat(){
        Console.WriteLine("Eating...");
    }
}
public class Dog: Animal
{
    public override void eat()
    {
        Console.WriteLine("Eating bread...");
    }
}
public class TestOverriding
{
    public static void Main()
    {
        Dog d = new Dog();
        d.eat();
    }
}
```

Output:

```
Eating bread...
```

C# Enum

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.

Points to remember

- enum has fixed set of constants
- enum improves type safety
- enum can be traversed

C# Enum Example

Let's see a simple example of C# enum.

```
using System;
public class EnumExample
{
   public enum Season { WINTER, SPRING, SUMMER, FALL }

   public static void Main()
   {
      int x = (int)Season.WINTER;
      int y = (int)Season.SUMMER;
      Console.WriteLine("WINTER = {0}", x);
      Console.WriteLine("SUMMER = {0}", y);
   }
}
```

Output:

```
WINTER = 0
SUMMER = 2
```

C# Encapsulation

Encapsulation is the concept of wrapping data into a single unit. It collects data members and member functions into a single unit called class. The purpose of encapsulation is to prevent alteration of data from outside. This data can only be accessed by getter functions of the class.

A fully encapsulated class has getter and setter functions that are used to read and write data. This class does not allow data access directly.

Here, we are creating an example in which we have a class that encapsulates properties and provides getter and setter functions.

Example

```
namespace AccessSpecifiers
{
   class Student
   {
      // Creating setter and getter for each property
      public string ID { get; set; }
      public string Name { get; set; }
      public string Email { get; set; }
   }
}
using System;
namespace AccessSpecifiers
{
   class Program
{
   static void Main(string[] args)
   {
      Student student = new Student();
      // Setting values to the properties
      student.ID = "101";
      student.Name = "Mohan Ram";
      student.Email = "mohan@example.com";
      // getting values
      Console.WriteLine("ID = "+student.ID);
      Console.WriteLine("Name = "+student.Name);
```

```
            Console.WriteLine("Email = "+student.Email);
        }
    }
}
```

Output:

```
ID = 101
Name = Mohan Ram
Email = mohan@example.com
```