

OBJECT ORIENTED PROGRAMMING CONCEPTS IN C#

10/22/2024

Chapter Three

Contents

2

1. Introduction
2. Classes & Objects
3. Constructors & Destructors
4. Functions / Methods in C#
5. Properties and Indexer in C#
6. Inheritance and Polymorphism
7. Interfaces

Object Oriented Programming

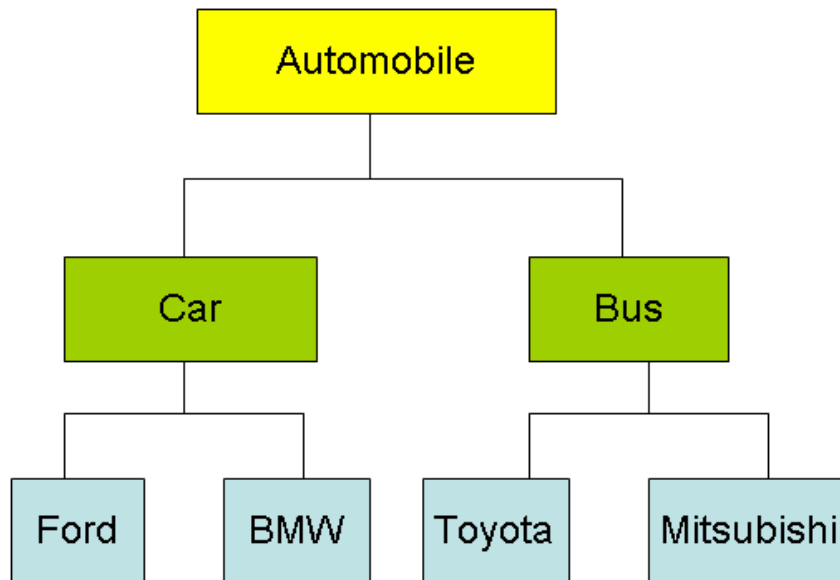
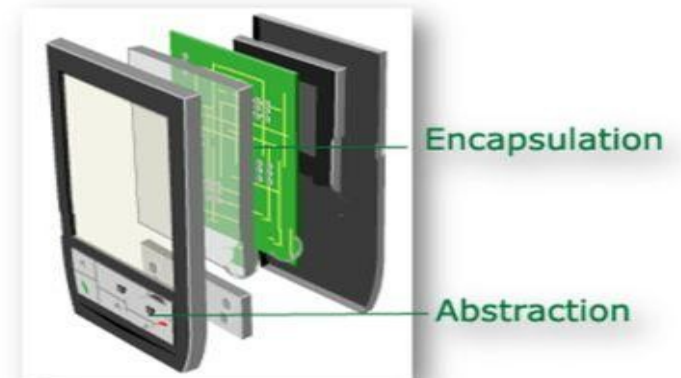
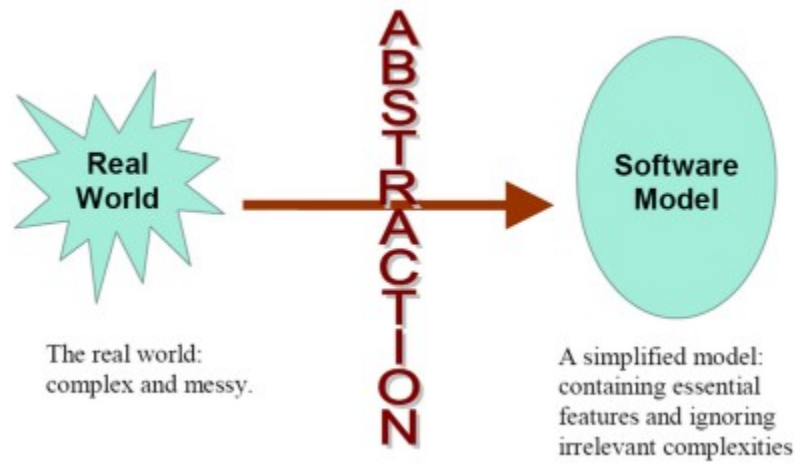
3

- Object-oriented programming (OOP) is a way to organize and conceptualize a program as a set of interacting **objects**.
 - The programmer **defines the types of objects** that will exist.
 - The programmer **creates object instances** as they are needed.
 - The programmer **specifies how these various object will communicate and interact** with each other.

Objected Oriented Principles

4

- OOP contains the following fundamentals principles
 - **Abstraction**
 - Allows us to consider complex ideas while **ignoring irrelevant detail** that would confuse us.
 - **Encapsulation**
 - Allows us to focus on **what something does** without considering the **complexities of how it works**.
 - **Inheritance**
 - Allows us to define **general** characteristics and operation of an object and allow us to create more **specialized** versions of this object
 - **Polymorphism**
 - Allows us to interact with an object as its generalized category regardless of its more specialized category.



What Exactly is OOP?

6

- Identifying an Object?
 - You can also think of other non physical things as objects:- such as a bank account
 - A bank account is not something that can be physically touched but intellectually we can consider a bank account to be an object.
- OOP is a method of programming that involves the creation of **intellectuals objects** that model a business problem we are trying to solve.
 - In creating an OO program we define the properties of a class of objects and then create individual objects from the class

Benefits of OOP Approach

7

- Better **abstraction**
 - Modeling information and behavior together
- Better **maintainability**
 - More comprehensible, less fragile software
- Better **usability**
 - Classes as encapsulated components that can be used in other systems

What is Object?

8

- **Object:** A single software unit that combines attributes and methods. It represents an **entity in the real world**.
- **Attribute:** A "**characteristic**" of an object; like a **variable** associated with a kind of object.

10/22/2024

□ Example

- **Dog**

- Attributes: breed, color, hungry, tired, etc.
- Behaviors: eating, sleeping, etc.

- **Bank Account**

- Attributes: account number, owner, balance
- Behaviors: withdraw, deposit

What is Classes?

10

- The definitions of the **attributes and methods of an object** are organized into a **class**.
 - Thus, a class is the **generic definition for a set of similar objects** (i.e. Person as a generic definition for different persons)
 - A **class** is an **abstract description** of a set of objects.
 - A class can be thought of as a **template used to create a set of objects**. (A blue print to create (instantiate) an

10/12/2021

- We actually **write code for a class**, not object
- The **objects** are called **instances of the class**.
- Every instance of the same class will have the same set of attributes;
 - Every object has **the same attributes** but,
 - Each instance will have its **own distinct values** for those attributes.

Bank Example

- The "**account**" class describes the **attributes** and **behaviors** of bank accounts.
- The "account" class defines **two state variables** (account number and balance) and **two methods** (deposit and withdraw).

class: Account

number:

balance:

deposit()

withdraw()

Bank Example – three objects

- When the program runs there will be many instances of the account class.
- Each instance will have its own account number and balance (*object state*)
 - **Methods can only be invoked .**

Instance #1

number: 054

balance: \$19

Instance #2

number: 712

balance: \$240

Instance #3

number: 036

balance: \$941

Instance Variable and Instance Methods

14

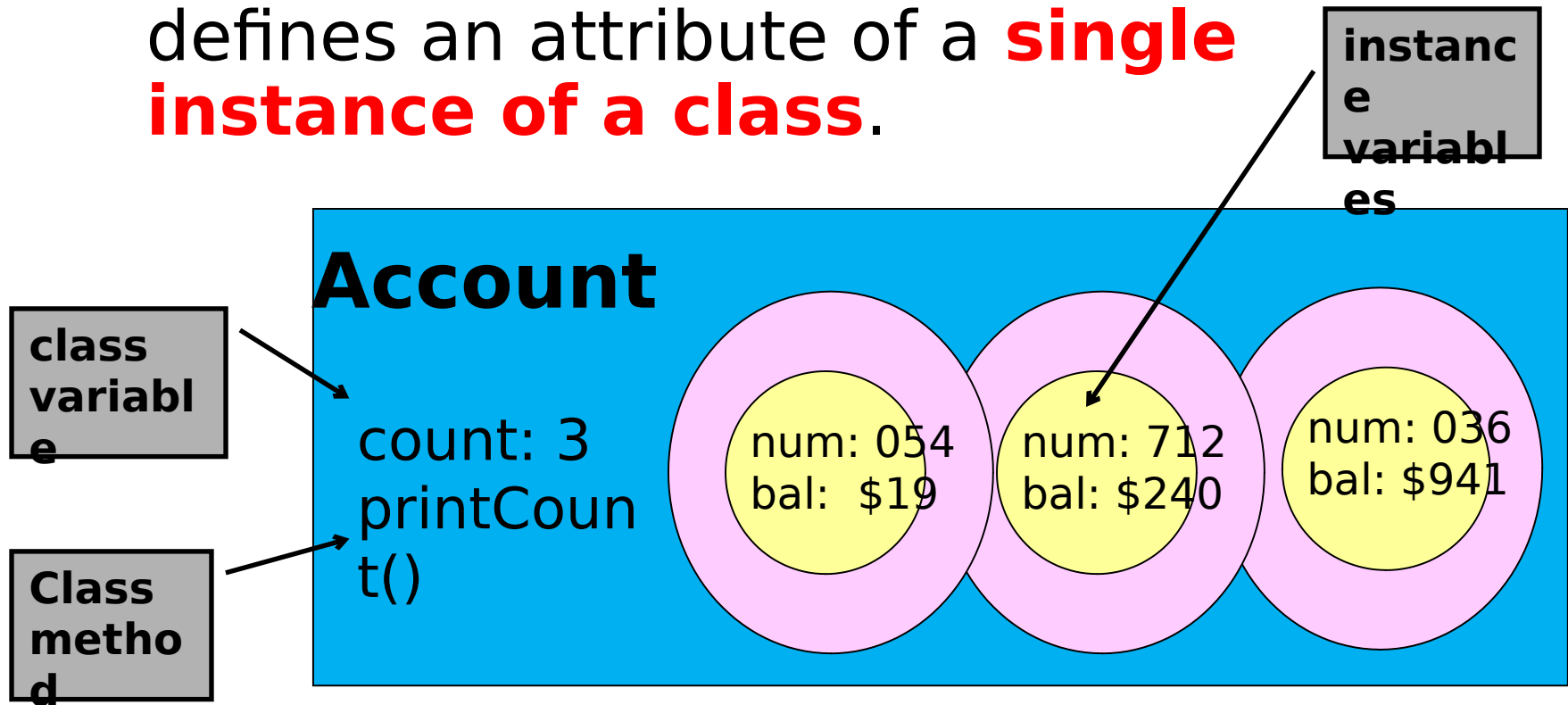
- A state-variables/ methods that are **associated with the one instance of a class** – instance variable/method.
 - Instance variables and instance methods can be **public or private**.
 - It is **necessary to instantiate** (create an instance of) a class **to use its instance variables and instance methods**.

Class Variables and Class Methods

15

- In addition to instance variables and instance methods, classes can also define class methods and class variables.
 - These are attributes and behaviors **associated with the class as a whole, not any one instance.**
 - Class variables and class methods can be **public or private.**
 - It is **not necessary to instantiate** a class **to use its class variables and class methods.**
- Note: Class variables and methods are declared with the **static** keyword in C#.

- A **class variable** defines an **attribute of an entire class**.
- In contrast, an **instance variable** defines an attribute of a **single instance of a class**.



Access Modifiers

17

- All types and type members have an **accessibility level**, which controls whether they **can be used from other code in your assembly or other assemblies**.
 - **Access modifiers**(or access specifies) are **keywords** that **set the accessibility** of classes, methods, and other members
- **public access modifiers**
 - The type or member can be **accessed by any other code** in the same assembly or another assembly that references it.

□ **private** access modifiers

- The type or member can be accessed **only by code in the same class**

□ **protected** access modifiers

- The type or member can be **accessed only by code in the same class**, or in a class that is **derived from that class**

□ **internal** access modifiers

- The type or member can be **accessed by any code in the same assembly**, but not from another assembly.
- An assembly is a file that is automatically generated by the compiler upon successful compilation of every .NET application. It can be either a Dynamic Link Library or an executable file

□ ~~Defaults~~ access modifiers

- The default access modifier for a **class** is **internal**
- The default access modifier for a **class member** is **private**

□ **protected internal** access modifiers

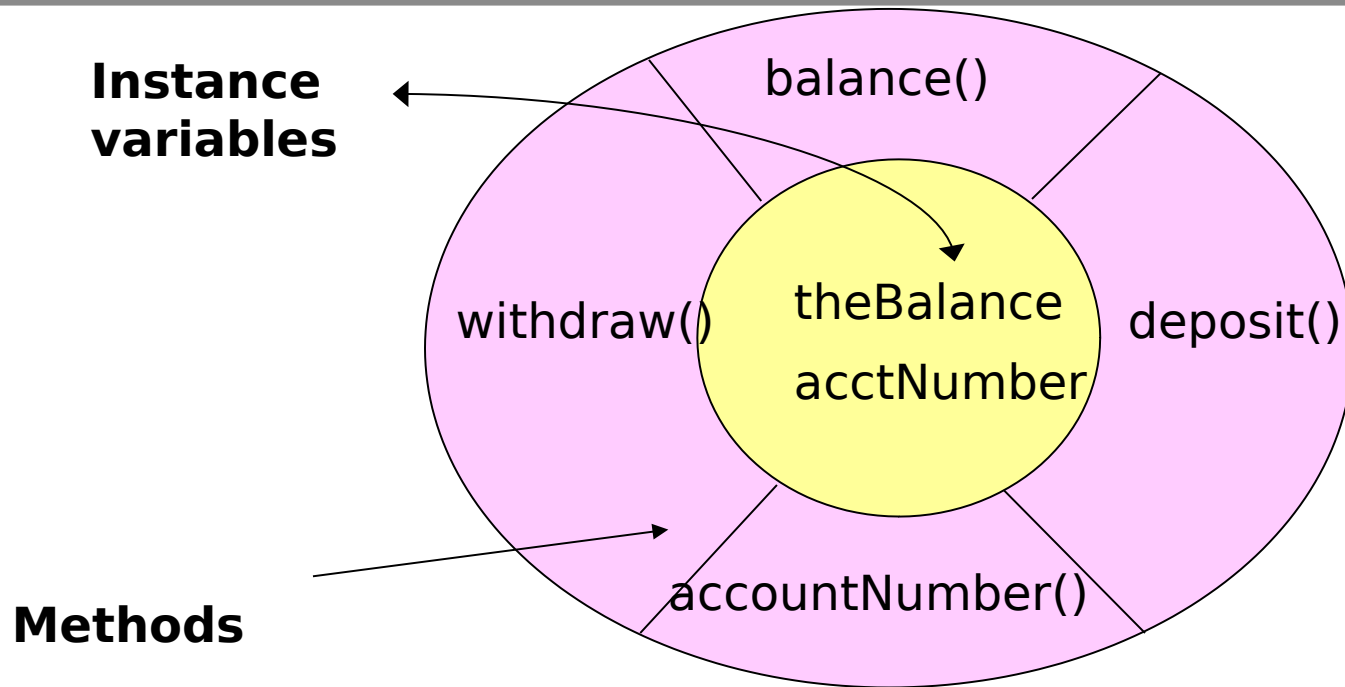
- The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly

Encapsulation

20

- When classes are defined, programmers can specify that **certain methods or state variables remain hidden inside the class.**
 - These variables and methods are accessible from within the class, but not accessible outside it.
 - The combination of collecting all the attributes of an object into a single class definition, combined with the ability to hide some definitions and type information within the class, is known as **encapsulation.**

Graphical Model of an Object



State variables make up the nucleus of the object. Methods surround and hide (**encapsulate**) the state variables from the rest of the program.

- OOP encapsulation is roughly analogous to only **allowing a user to access data in a database via predefined stored procedures.**
 - Users can access the data, but not directly and without having to have direct knowledge of its structure.

Defining a Class

23

- In object oriented programming, a class defines certain **properties, fields, events, method** etc.
 - A class defines the kinds of data and the functionality their objects will have.
- A class definition starts with the keyword **class** followed by the **class name**; and the **class body** enclosed by a pair of curly braces.

- Following is the general format of class definition

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

- Example

```
public class Customer {  
    // Fields, properties, methods  
    and events go here...  
}
```

□ Note

- Access specifiers **specify the access rules** for the members as well as the class itself.
- If not mentioned, then the default access specifier for a class type is **internal**.
- Default access for the members is **private**.

Creating Objects

26

- Objects can be created by using the **new** keyword followed by the name of the class that the object will be based on, like this
 - **Customer cust1= new Customer();**
- When an instance of a class is created, a reference to the object is passed back to the programmer.
 - In the previous example, **cust1** is a reference to an object that is based on Customer
 - This reference refers to the new object but does not contain the object data itself.

Class Declaration

```
class Circle
{
    double radius = 1.0;
    double findArea(){
        return radius *
        radius * 3.14159;
    }
}
```

Declaring Object reference variables

```
Circle myCircle;
```

Creating Objects

```
myCircle=new Circle();
```

...in a single step

```
Circle myCircle=new Circle();
```

□ Accessing Objects

- Referencing the object's data

`objectReference.data`

`mycircle.radius`

- Invoking the object's method:

`objectReference.method`

`myCircle.findArea()`

Defining Method

29

- A method can be defined using the following templates

```
{modifier} {return type}  MethodName  
({parameters})  
{  
  // body  
}
```

- Example

```
public void MyMethod(int parameter1, string parameter2)  
{  
  // write your method code here..  
}
```

Defining Field

30

- Field is a class level variable that can holds a value.
 - Generally field members should have a **private access** modifier and used with a **property**.
- **Member fields** are the attributes of an object (from design perspective) and they **are kept private to implement encapsulation**.
 - These variables can only be accessed using the public member functions.

Constructors

31

- A class **constructor** is a special member method/function of a class that is **executed whenever we create new objects of that class.**
 - A constructor has exactly the **same name as that of class** and it **does not have any return type.**
- A class can have **parameterized** or **parameter less** constructors.
 - The constructor will be **called when you create an instance of a class.**
 - Constructors can be defined by using an access modifier and class name: **<access modifiers> <class name>(){ }**

- Important points to remember about Constructors
 - A constructor **can not be abstract, final, and Synchronized**.
 - Within a class, you can create **only one static constructor**.
 - A **static** constructor **cannot be a parameterized constructor**.
 - Constructor of a class **must have the same name** as the class name in which it resides.
 - A constructor **doesn't have any return type, not even void**.
 - A class can have **any number of constructors**.
 - Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

□ Types of Constructors

- **Default Constructor**

- A constructor with **no parameters** is called a default constructor (initializes all numeric **fields to zero and all string and object fields to null inside a class**)

- **Parameterized Constructor**

- A constructor have **at least one parameter** is called a parameterized constructor. (helps you to assign initial value to an object at the time of its creation)

- **Copy Constructor**

- This constructor will creates an object by copying variables from another object. (initialize a new instance to the values of an existing instance)

Example

34

```
class Employee
{
    private name, location;
    //Default Constructor
    public Employee()
    {
        name = "Alex Len";
        location = "San Francisco";
    }
    //Parameterized Constructor
    public Employee(string a, string b)
    {
        name = a;
        location = b;
    }
    public Employee(Employee s)
    {
        name= s.name;
        location= s.location
    }
}
```

```
Employee u1= new
    Employee();
Employee u2 = new
    Employee("Moges","AA");
Employee u3 = new
    Employee(u1) ;
```

10/22/2024

Destructors

35

- A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope.
 - A **destructor** has exactly the **same name as that of the class with a prefixed tilde (~)** and it can **neither return a value nor can it take any parameters**.
- Destructor can be very useful for **releasing memory resources before exiting the program**.
 - Destructors cannot be **inherited** or **overloaded**.

□ Important Points:

- A Destructor is **unique to its class** i.e. there cannot be more than one destructor in a class.
- A Destructor has **no return type** and has **exactly the same name as the class name** (Including the same case).
 - It is distinguished apart from a constructor because of the ***Tilde symbol (~)* prefixed** to its name.
- A Destructor does **not accept any parameters** and **modifiers**.
 - It cannot be defined in Structures. It is only used with classes.
 - It cannot be **overloaded** or **inherited**.
- It is **called when the program exits**.
- The destructor **will invoke automatically**, whenever an instance of **class** is no longer needed.

Static Members of a C# class

37

- We can define class members as static using the **static** keyword.
 - When we declare a member of a class as static, it means no matter how many objects of the class are created, **there is only one copy of the static member.**
- The keyword **static** implies that only **one instance of the member exists for a class.**
 - Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it.
- Static variables can be initialized outside the member function or class definition.

Static Method/Function

38

- Sometimes a method performs a task that **does not depend on the contents of any object.**
 - Such a method applies to the class in which it's declared as a whole and is known as a **static method.**
- To declare a method as static, place the keyword **static** before the return type in the method's declaration.
 - You call any static method by specifying the name of the class in which the method is declared, followed by the member access (.) operator and the method name, as in ***ClassName.MethodName(arguments)***
eg: Math.Sqrt(900.0)
- Such functions can **access only static variables.**

10/22/2024

Exercise -1

39

- Create a class "House", with an attribute "length", "width", "area", a constructor that sets their values and a method "ShowData" to display "I am a house, my area is 200 m2" (instead of 200, it will show the real surface).
- Write a second class to test your House class.
 - The second class should allow the user to input length and width of the house.
 - Display the area of the house using ShowData method.

Exercise 1.b:

40

- Write a C# Sharp program that accept two integers and return true if either one is 5 or their sum or difference is 5.
 - Class name: Program2, TestProgram2(Main method)
 - Attributes: number1 and number2
 - Method: checkIf5
 - Output:
Enter the first number: 5
Enter the second number: 3
True

Enter the first number: 3
Enter the second number: 4
False

Exercise 1.c:

41

- Write a C# Sharp program to check if it is possible to add two integers to get the third integer from three given integers
 - Class name: Program3, TestProgram3(Main method)
 - Attributes: number1 , number2 and number3
 - Method: checkIfPossible
 - Output:
Enter the first number: 5
Enter the second number: 3
Enter the third number: 2
True

Enter the first number: 3
Enter the second number: 4
Enter the third number: 5
False

Exercise 1.d:

42

- Write a C# Sharp program to check if two or more non-negative given integers have the same rightmost digit.
 - Class name: Program4 , TestProgram4(Main method)
 - Attributes: number1 , number2 and number3
 - Method: checkRightmostDigit
 - Output:
Enter the first number: 50
Enter the second number: 30
Enter the third number: 20
True

Enter the first number: 31
Enter the second number: 43
Enter the third number: 52
False

Exercise 1.e:

43

- Write a C# Sharp program to calculate the area of the rectangle.
 - Class name: Program5 , TestProgram5(Main method)
 - Attributes: length, width
 - Method: caculateArea, getLength, getWidth, setLength, setWidth
 - Output:
Enter the length of the rectangle: 6
Enter the width of the rectangle: 7

The length of the rectangle is: 6
The width of the rectangle is: 7
The area of the rectangle is: 42

Exercise 1.f:

44

- Write a C# Sharp program to find the **larger** from two given integers. However if the two integers have the same remainder when divided by 7, then return the **smaller integer**. If the two integers are the same, **return 0**.
 - Class name: Program6 , TestProgram6(Main method)
 - Attributes: number1, number2
 - Method: checkLargerNumber
 - Output:
Enter the first number: 17
Enter the second number: 10
10

Exercise 1.g:

45

- Write a C# Sharp program to check if a given positive number is a multiple of 3 or a multiple of 7.
 - Class name: Program7 , TestProgram7(Main method)
 - Attributes: number1
 - Method: checkMultipleOf3And7
 - Output:
Enter the number: 17
False

Exercise 1.h:

46

- Write a C# Sharp program to check which number nearest to the value 100 among two given integers. Return 0 if the two numbers are equal.
 - Class name: Program9 , TestProgram9(Main method)
 - Attributes: number1, number2
 - Method: checkNearest
 - Output:
Enter the first number: 89
Enter the second number: 92
92

Exercise-2

47

- Create a class to store details of student like rollno, name, course joined and fee paid so far. Assume courses are C# and ASP.NET with course fees being 2000 and 3000.
 - Provide the a **constructor** to take rollno, name and course.
- Provide the following methods:
 - Payment(amount)
 - Print()

Properties in C#

48

- Properties are also known as the smart fields in C#
 - They are the extensions of the **C# data fields**
- In a class we declare the data fields as private and will provide the public **SET/GET methods** to access the data fields
 - The access modifier can be private, public, protected or internal
 - The return type can be any valid C# data type

20/12/2024

Properties : get and set Accessors

49

- Properties contain
 - **getters (get{})** to retrieve the value of the underlying field and
 - **setters (set{})** to set the value of the underlying field

- **Syntax of Properties**

```
<acces_modifier> <return_type>  
<property_name>
```

```
{  
    get  
    {  
    }  
    set  
    {  
    }  
}
```

□ Example

```
private int length;  
    public int Length  
    {  
        get  
        {  
            return length;  
        }  
        set  
        {  
            length = value;  
        }  
    }
```

- By convention, we name each **property with the capitalized name of the instance variable** that it manipulates.
 - e.g., **Length** is the property that represents instance variable **length** - C# is case sensitive, so these are distinct identifiers.
- A private field which cannot be accessed directly. It will only be accessed via a Property.
 - Calling a getter or a setter in C#
 - `objectOfTheClass.propertyName`
 - Getters and setters let you **treat the values like public properties.**

- We can also apply some additional logic in get and set, as in the below example.

```
public int Length
{
    get { return length/2; }
    set {
        if (value > 0)
            length = value;
        else
            length = 0;
    }
}
```

□ **Auto-implemented Property**

- property declaration has been made easy if you don't want to apply some logic in get or set.
- The following is an example of an auto-implemented property:

□ **public int Length{ get; set; }**

Indexers in C#

54

- An indexer allows an instance of a class to be indexed as an array.
 - If the user will define an indexer for a class, then **the class will behave like a virtual array**.
 - Array access operator i.e ([]) is used to access the instance of the class which uses an indexer.
 - A user can retrieve or set the indexed value without pointing an instance or a type member.
- Indexers are almost similar to the **Properties**.
 - The main difference between Indexers and **Properties** is that the **accessors** of the **Indexers will take parameters**.

□ Syntax

```
[access_modifier] [return_type] this [argument_list]  
{  
  get  
  {  
    // get block code  
  }  
  set  
  {  
    // set block code  
  }  
}
```

- **Important Points About Indexers:**
 - There are two types of Indexers i.e. **One Dimensional Indexer & MultiDimensional Indexer**
- Indexers can be **overloaded**.
- Indexers are also known as the ***Smart Arrays or Parameterized Property in C#***.
- This enables the object to be indexed in a similar way to arrays.
 - A **set accessor** will always assign the value while the **get accessor** will return the value.
 - “***this***” keyword is always used to **declare an indexer**.
 - To define the value being **assigned by the set indexer**, “***value***” keyword is used.
- Indexer **can't be a static member** as it is an instance member of the class.

Exercise-3

57

- Create a Month class that has a single data member of month number.
 - Include a member method that returns the name of the month and another method that returns the number of days in the month.
 - The ToString() method should return the name and number of days.
- Write a second class to test your Month class.
 - The second class should allow the user to input a month number.
 - Display the name of the month associated with the number entered and the number of days in that month. For this exercise, use 28 for February. If the user inputs an invalid entry, display an appropriate message

Exercises-4

58

1. **Define a class Student**, which contains the following information about students: full name, course, subject, university, e-mail and phone number.
2. **Declare several constructors** for the class Student, which have different lists of parameters (for complete information about a student or part of it).
 - Data, which has no initial value to be initialized with null.
 - Use nullable types for all non-mandatory data.
3. **Add a static field** for the class Student, which holds the number of created objects of this class.

4. Add a method in the class Student, which **displays complete information about the student.**
5. Modify the current source code of Student class so as to encapsulate the data in the class using properties.
6. **Write a class StudentTest**, which has to test the functionality of the class Student.
7. **Add a static method in class StudentTest**, which creates several objects of type Student and store them in static fields. Create a static property of the class to access them. Write a test program, which displays the information about them in the console.

Inheritance

60

- Inheritance allows a software developer to **derive a new class from an existing one.**
 - The **existing class** is called the **parent, super,** or **base** class.
 - The **derived class** is called a **child** or **subclass.**
- The child **inherits characteristics** of the parent.
 - The child has special rights to the parents methods and data.
 - Public access like any one else
 - **Protected** access available only to child classes (and their descendants).
- The child has its own **unique behaviors and data.**

10/22/2024

- The advantage of making a new class a subclass is that it will **inherit attributes** and **methods** of its parent class (also called the **super-class**).
- **Subclasses extend existing classes in three ways:**
 - By **defining new** (additional) attributes and methods.
 - By **overriding** (changing the behavior) existing attributes and methods.
 - By **hiding existing** attributes and methods.

Example: extending existing classes

62

```
1  class A
2  {
3      protected int v1;
4      protected virtual float calc()
5      {
6          //
7      }
8      protected float getMyValue()
9      {
10         //
11     }
12 }
```

10/22/2024

```
class B:A
{
    int v2;
    float area()
    {
        //
    }
}
```

```
class C:A
{
    public override float calc()
    {
        //
    }
}
```

```
class D:A
{
    int v1;
    public float getMyValue()
    {
        //
    }
}
```

□ Subclasses

- When a new class is developed a programmer can define it to be a ***subclass*** of an existing class.
- Subclasses are used to define **special cases, extensions, or other variations** from the originally defined class.

□ Examples:

- SavingsAccount and CheckingAccount can be derived from the Account class (see the next slide).

Examples: Base Classes and Derived Classes

Base class	Derived classes
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan MortgageLoan
Employee	FacultyMember StaffMember
Account	CheckingAccount SavingsAccount

- Inheritance should create an ***is-a* relationship**, meaning the **child *is a* more specific version of the parent**.
- The ability to use inheritance makes programs **easier to write, less error-prone, and easier to understand**
- Terminology
 - **Super (Base) class**: a class that is used as a basis for inheritance
 - **Subclass (derived/extended)**: a class that inherits from a base class.
- Inheritance is **transitive**; that means **a child inherits all the members of all its ancestors**

□ Extending Classes

- When you create a class that is an extension or child of another class, you use a **single colon** between the **derived class name** and its **base class name**
 - Inheritance **works in one direction**
- The keyword **protected** provides you with an intermediate level of security
- A protected data field or method can be used within its own class or in any classes extended from that class, but it cannot be used by “outside” classes
- C# and Java **support single inheritance**, meaning that a derived class **can have only one parent class**.

- Declaring a derived class
 - Define a new class `DerivedClass` which extends `BaseClass`

```
class BaseClass
{
    // class contents
}

class DerivedClass : BaseClass
{
    // class contents
}
```

Example: Inheritance

69

```
class Shape
{
    public void setWidth(int w)
    {
        width = w;
    }
    public void setHeight(int h)
    {
        height = h;
    }
    protected int width;
    protected int height;
}
// Derived class
class Rectangle: Shape
{
    public int getArea()
    {
        return (width * height);
    }
}
```

10/22/2024

```
class RectangleTester
{
    static void Main(string[] args)
    {
        Rectangle Rect = new Rectangle();
        Rect.setWidth(5);
        Rect.setHeight(7);
        // Print the area of the object.
        Console.WriteLine("Total area: {0}", Rect.getArea());
        Console.ReadKey();
    }
}
```

Output

```
Total area: 35
```

Overriding Methods

71

- A child class can ***override*** the definition of an inherited method in favor of its own
 - That is, a child can redefine a method that it inherits from its parent
- The new method must **have the same signature** as the parent's method, but can have a **different implementation**.
 - The type of the object executing the method determines which version of the method is invoked.

- Derived classes can also override inherited members by providing an **alternate implementation**.
 - In order to be able to override a member, **the member in the base class** must be marked with the virtual keyword.
- By default, base class members are **not marked as virtual** and **cannot be overridden**.
- Using the same method name to indicate different implementations is called **polymorphism**.

- Method Overriding (**virtual** and **override** keyword)
 - In C#, for overriding the base class method in derived class, you have to declare **base class method as virtual** and **derived class method as override** as shown below:

```
class A
{
    public virtual void Test()
    {
        Console.WriteLine("A::Test()");
    }
}
class B : A
{
    public override void Test()
    {
        Console.WriteLine("B::Test()");
    }
}
```

□ Note

1. The **virtual** keyword is used to **modify a method, property, indexer, or event declared in the base class and allow it to be overridden in the derived class.**
2. The **override** keyword is used **to extend or modify a virtual/abstract method, property, indexer, or event of base class into derived class.**
3. The **new** keyword is used **to hide a method, property, indexer, or event of base class into derived class**

□ References and inheritance

- An object reference can refer to an object of its class, or to an object of any class derived from it by inheritance.

□ Dynamic Binding

- A **polymorphic reference** is one which can refer to different types of objects at different times. It **morphs!**
- The type of the **actual instance**, not the declared type, determines which method is invoked.
- **Polymorphic references** are therefore resolved at run-time, not during compilation.
 - This is called **dynamic binding**.

```
Holiday day;  
day = new Holiday();  
...  
day = new Christmas();
```

- How do access **Super class constructor** and **Methods** from a Subclass?
 - The **base** keyword is used to access **members of the base class** from **within a derived class**:
 - Call a method on the base class that has been overridden by another method.
 - Specify which base-class constructor should be called when creating instances of the derived class.
 - A base class access is permitted only in a constructor, an instance method, or an instance property accessor.
- **It is an error to use the base keyword from within a static method.**

```

public class BaseClass
{
    int num;
    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }
    public BaseClass(int i)
    {
        num = i; Console.WriteLine("in BaseClass(int i)");
    }
    public int GetNum()
    {
        return num;
    }
}

```

```

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    { }
    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    { }
    static void Main()
    {
        DerivedClass md = new DerivedClass(); DerivedClass mdl = new DerivedClass(1);
    }
}
/* Output: in BaseClass() in BaseClass(int i) */

```

Exercise-5

78

- Define a class **Human** with properties "first name" and "last name". Define the class **Student** inheriting Human, which has the property "mark". Define the class **Worker** inheriting Human with the property "wage" and "hours worked". Implement a "calculate hourly wage" method, which calculates a worker's hourly pay rate based on wage and hours worked. Write the corresponding constructors and encapsulate all data in properties.

10/22/2024

Polymorphism

79

- This is the ability of an object to perform in a wide variety of ways. There are two types:
 - **Static polymorphism** (compile time). You can achieve static polymorphism through **function overloading** and **operator overloading**.
 - **Dynamic polymorphism** (runtime time). You can obtain this type through executing **function overriding**.

Static Polymorphism

80

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

Method Overloading

81

- It is the ability to **redefine a function/method** in more than one form.
 - A user can implement function overloading by defining two or more functions in a class sharing the same name.
- C# can distinguish the methods with **different method signatures**.
 - i.e. the methods can have **the same name** but with **different parameters list** (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.

- Overloaded methods are **differentiated based** on the **number** and **type of the parameters** passed as arguments to the methods.
 - You can not define more than one method with the same name, Order and the type of the arguments. (It would be compiler error.)
- The compiler **does not consider the return type** while differentiating the overloaded method.
 - But you **cannot declare two methods with the same signature** and **different return type**.

□ Different ways of doing overloading methods-

Method overloading can be done by changing:

1. The **number of parameters** in two methods.
2. The **data types** of the parameters of methods.
3. The **order of the parameters** of methods

□ Why do we need Method Overloading ?

- If we need to do the same kind of the operation in different ways i.e. for different inputs.

```
using System;
namespace PolymorphismApplication
{
    class Printdata {
        void print(int i) {
            Console.WriteLine("Printing int: " + i );
        }
        void print(double f) {
            Console.WriteLine("Printing float: " + f);
        }
        void print(string s) {
            Console.WriteLine("Printing string: " + s);
        }
        static void Main(string[] args) {
            Printdata p = new Printdata();
            p.print(5);
            p.print(500.263);
            p.print("Hello C++");
            Console.ReadKey();
        }
    }
}
```

Output

```
5
500.263
Hello C++
```

Operator Overloading

85

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

□ Syntax

access_specifier return_type **operator** Operator_symbol (parameters)
{ // Code }

□ Example

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

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

□ **Overloadable and Non-Overloadable Operators**

- The following table describes the overload

N o.	Operators & Description
1	+, -, !, ~, ++, -- : These unary operators take one operand and can be overloaded.
2	+, -, *, /, % : These binary operators take one operand and can be overloaded.
3	==, !=, <, >, <=, >= : The comparison operators can be overloaded.
4	&&, : The conditional logical operators cannot be overloaded directly.
5	+=, -=, *=, /=, %= : The assignment operators cannot be overloaded.

Dynamic Polymorphism

88

- Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.
- C# allows you to create **abstract classes** that are used to provide partial class implementation of an interface.
 - **Implementation is completed when a derived class inherits from it.**

Abstract Classes

89

- An **abstract class** is one from which you cannot create any concrete objects, but from which you can inherit
 - An **abstract method** has no method statements;
- Any class derived from a class containing an abstract method must override the abstract method by providing a body for it
 - When you create an abstract method, you provide the keyword **abstract** and the intended method type, name, and argument
 - When you create a subclass that inherits an abstract method from a parent, you must use the **override** keyword

- The **abstract** modifier indicates that the thing being modified has **a missing or incomplete implementation**.
 - The abstract modifier can be used with classes, methods, properties, indexers, and events.
- Use the abstract modifier in a class declaration to **indicate that a class is intended only to be a base class of other classes**.
 - **Members marked as abstract, or included in an abstract class**, must be implemented by classes that derive from the abstract class.

- Abstract classes have the following features:
 - An abstract class **cannot be instantiated**.
 - An abstract class may **contain abstract methods and accessors**.
 - It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings.
 - The **sealed modifier** prevents a class from being inherited and the abstract modifier requires a class to be inherited.
 - A non-abstract class derived from an abstract class must **include actual implementations** of all inherited abstract methods and accessors.

- An abstract method is **implicitly** a virtual method.
- **Abstract method** declarations are only permitted **in abstract classes**.
- Because an abstract method declaration provides no actual implementation, there is no method body; the **method declaration simply ends with a semicolon** and **there are no curly braces** ({ }) following the signature.

```
using System;
namespace PolymorphismApplication {
    abstract class Shape {
        public abstract int area();
    }
    class Rectangle: Shape {
        private int length;
        private int width;
        public Rectangle( int a = 0, int b = 0) {
            length = a;
            width = b;
        }
        public override int area () {
            Console.WriteLine("Rectangle class area :");
            return (width * length);
        }
    }
    class RectangleTester {
        static void Main(string[] args) {
            Rectangle r = new Rectangle(10, 7);
            double a = r.area();
            Console.WriteLine("Area: "+a);
            Console.ReadKey();
        }
    }
}
```

Output

Rectangle class area :
Area: 70

Interface

94

- An **interface** is a purely logical construct that describes functionality **without specifying implementation**
 - An interface in C# contains only the **declaration** of the methods, properties, and events, **but not the implementation**
- An interface generally defines set of methods that will be implemented by the class
 - But interface doesn't implement any method itself
 - It is left to the class that implements the interface by providing implementation for all the members of the interface.

□ Declaring Interface

- You define an interface by using the **interface** keyword
- It is similar to class declaration.
- Interface statements are public by default.

□ Following is an example of an interface declaration

```
interface Iaddition
{
    int add(int x, int y);
}
```

- Any class implementing Compute will be able to implement both these methods.

INTERFACES CAN EXTEND ONLY INTERFACES, THEY CAN'T EXTEND CLASSES.

- **IMPLEMENTING INTERFACES**

- An Interface can be implemented as follows:

```
class classname: Interfacename {  
    body of classname  
}
```

- A class can **derive from a single class** and implement as many interfaces required:
**class classname: superclass,
 interface1, interface 2, ...
{
 body of class name
}**
- Just as a class can implement multiple interfaces, one single interface can be implemented by multiple classes.

□ **Limitations of interface**

- Interfaces cannot have data members.
- They cannot define constructors, destructors .
- No member can be declared static.

□ **Difference and similarities between interface and abstract class**

- Similarities **Neither can be instantiated**
 - Neither can be sealed
- Differences
 - Interfaces **cannot contain any implementation**
 - Interfaces **cannot declare non-public members**
 - Interfaces **cannot extend non-interfaces**

Questions?

Thank You