

Activity Week 2

Sumário

Atividade - 1	2
1. Encapsulamento	3
2. Abstração	6
3. Herança	9
4. Polimorfismo	13
Classe Principal	17
Atividade - 2	19
Referências	23

Atividade - 1

Objetivo

Meu objetivo neste laboratório é compreender e demonstrar os princípios da programação orientada a objetos (OOP) em C#, além de comparar as estruturas de dados comuns da linguagem.

Princípios OOP em C#

A programação orientada a objetos (OOP) é um paradigma fundamental no desenvolvimento de software. Ela se baseia no conceito de "objetos", que são instâncias de "classes". Cada objeto pode conter dados (atributos) e comportamentos (métodos). Os quatro pilares que sustentam a OOP são Encapsulamento, Abstração, Herança e Polimorfismo. Vou explicar cada um deles com exemplos práticos em C#.

1. Encapsulamento

Para mim, o encapsulamento é como proteger os dados internos de um objeto, controlando como eles podem ser acessados e modificados. É o ato de empacotar os dados e os métodos que operam sobre esses dados em uma única unidade (a classe), e restringir o acesso direto a alguns desses componentes. Isso é crucial para manter a integridade dos dados e simplificar o uso da classe.

Exemplo em C# (Classe `Logger`):

Neste exemplo, criei uma classe `Logger`. O caminho do arquivo de log (`_logFilePath`) é um detalhe interno e privado. Eu forneço métodos públicos (`LogMessage` e `ReadLogs`) para interagir com o log, mas a forma como o log é armazenado e lido é completamente encapsulada dentro da classe. Isso significa que quem usa a classe `Logger` não precisa se preocupar com os detalhes de implementação do arquivo.

```
// Encapsulation: Logger
public class Logger
{
    private readonly string _logFilePath; // Encapsula o caminho do
    arquivo de log, tornando-o privado.

    public Logger(string logFilePath)
    {
        _logFilePath = logFilePath;
    }

    public void LogMessage(string message)
    {
        try
        {
            // A lógica interna para escrever no arquivo é encapsulada
            aqui.
            File.AppendAllText(_logFilePath, $"[{DateTime.Now}]
{message}\n");
            Console.WriteLine($"Logged: {message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error logging message: {ex.Message}");
        }
    }
}
```

```
    }  
}  
  
public string ReadLogs() // Fornece acesso controlado aos logs,  
sem expor o arquivo diretamente.  
{  
    try  
    {  
        if (File.Exists(_logFilePath))  
        {  
            return File.ReadAllText(_logFilePath);  
        }  
        return "Log file does not exist.";  
    }  
    catch (Exception ex)  
    {  
        return $"Error reading logs: {ex.Message}";  
    }  
}  
}
```

```

Program.cs
// Encapsulation: Logger
2 usages
public class Logger
{
    private readonly string _logFilePath;

    1 usage
    public Logger(string logFilePath)
    {
        _logFilePath = logFilePath;
    }

    2 usages
    public void LogMessage(string message)
    {
        try
        {
            File.AppendAllText(_logFilePath, contents: $"{DateTime.Now} {message}\n");
            Console.WriteLine($"Logged: {message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error logging message: {ex.Message}");
        }
    }

    1 usage
    public string ReadLogs()
    {
        try
        {
            if (File.Exists(_logFilePath))
            {
                return File.ReadAllText(_logFilePath);
            }
            return "Log file does not exist.";
        }
        catch (Exception ex)
        {
            return $"Error reading logs: {ex.Message}";
        }
    }
}

```

Encapsulation

2. Abstração

Para mim, a abstração é sobre focar no "o quê" e não no "como". É o processo de ocultar os detalhes complexos de implementação e mostrar apenas a funcionalidade essencial ao usuário. Em C#, eu consigo isso principalmente usando classes abstratas e interfaces, que definem um contrato sem especificar todos os detalhes de como esse contrato será cumprido.

Exemplo em C# (Classes `NotificationService`, `EmailNotificationService`, `SmsNotificationService`):

Criei uma classe abstrata `NotificationService` que define o método `SendNotification`. Qualquer serviço de notificação (como e-mail ou SMS) deve implementar esse método. A classe abstrata também possui um método concreto `LogAttempt`, que é comum a todas as notificações. Isso me permite usar diferentes tipos de serviços de notificação de forma genérica, sem me preocupar com os detalhes específicos de envio de cada um.

```
// Abstraction: Notification Service
public abstract class NotificationService // Classe abstrata que
define um contrato para serviços de notificação.
{
    public abstract void SendNotification(string recipient, string
message); // Método abstrato que deve ser implementado pelas
subclasses.

    public void LogAttempt(string recipient, string message) // Método
concreto, comum a todas as notificações.
    {
        Console.WriteLine($"Attempting to send notification to
{recipient}: '{message}'");
    }
}

public class EmailNotificationService : NotificationService //
Implementação concreta para envio de e-mail.
{
    public override void SendNotification(string recipient, string
message)
    {
```

```

        LogAttempt(recipient, message);
        Console.WriteLine($"Sending email to {recipient}: {message}");
        // Aqui estaria a lógica real para enviar um e-mail.
    }
}

public class SmsNotificationService : NotificationService //
Implementação concreta para envio de SMS.
{
    public override void SendNotification(string recipient, string
message)
    {
        LogAttempt(recipient, message);
        Console.WriteLine($"Sending SMS to {recipient}: {message}");
        // Aqui estaria a lógica real para enviar um SMS.
    }
}

```



```

Program.cs
// Abstraction: Notification Service
4 usages 2 inheritors
public abstract class NotificationService
{
    2 usages 2 overrides
    public abstract void SendNotification(string recipient, string message);

    2 usages
    public void LogAttempt(string recipient, string message)
    {
        Console.WriteLine($"Attempting to send notification to {recipient}: '{message}'");
    }
}

1 usage
public class EmailNotificationService : NotificationService
{
    0+2 usages
    public override void SendNotification(string recipient, string message)
    {
        LogAttempt(recipient, message);
        Console.WriteLine($"Sending email to {recipient}: {message}");
        // Logic for sending actual email
    }
}

1 usage
public class SmsNotificationService : NotificationService
{
    0+2 usages
    public override void SendNotification(string recipient, string message)
    {
        LogAttempt(recipient, message);
        Console.WriteLine($"Sending SMS to {recipient}: {message}");
    }
}

```

Abstraction

3. Herança

A herança é um dos pilares mais poderosos da OOP para mim, pois permite a reutilização de código e a criação de uma hierarquia de classes. Uma nova classe (chamada classe derivada ou subclasse) pode herdar propriedades e comportamentos de uma classe existente (a classe base ou superclasse). Isso significa que eu não preciso reescrever o código comum para classes relacionadas.

Exemplo em C# (Classes `UserAccount`, `AdministratorAccount`, `DeveloperAccount`):

Eu modeliei diferentes tipos de contas de usuário. A classe `UserAccount` é a classe base, contendo propriedades e métodos comuns a todos os usuários (como `Username`, `Email` e `Login`). `AdministratorAccount` e `DeveloperAccount` herdam de `UserAccount`, ganhando todas as suas funcionalidades e adicionando comportamentos específicos (`ManageUsers`, `WriteCode`) e propriedades (`Department`, `ProgrammingLanguage`). Eu também sobrescrevo o método `DisplayRole` para personalizar a exibição para cada tipo de conta.

```
// Inheritance: User Accounts
public class UserAccount // Classe base para todos os tipos de contas
de usuário.
{
    public string Username { get; set; }
    public string Email { get; set; }

    public UserAccount(string username, string email)
    {
        Username = username;
        Email = email;
    }

    public virtual void DisplayRole() // Método virtual que pode ser
sobrescrito pelas subclasses.
    {
        Console.WriteLine($"User: {Username}, Email: {Email}");
    }

    public void Login() // Método comum a todas as contas de usuário.
    {
        Console.WriteLine($"{Username} logged in.");
    }
}
```

```

    }
}

public class AdministratorAccount : UserAccount //
AdministratorAccount herda de UserAccount.
{
    public string Department { get; set; }

    public AdministratorAccount(string username, string email, string
department) : base(username, email) // Chama o construtor da classe
base.
    {
        Department = department;
    }

    public override void DisplayRole() // Sobrescreve o método
DisplayRole para administradores.
    {
        Console.WriteLine($"Administrator: {Username}, Email: {Email},
Dept: {Department}");
    }

    public void ManageUsers() // Método específico de administradores.
    {
        Console.WriteLine($"Administrator {Username} is managing
users.");
    }
}

public class DeveloperAccount : UserAccount // DeveloperAccount herda
de UserAccount.
{
    public string ProgrammingLanguage { get; set; }

    public DeveloperAccount(string username, string email, string
programmingLanguage) : base(username, email)
    {
        ProgrammingLanguage = programmingLanguage;
    }
}

```

```
public override void DisplayRole() // Sobrescreve o método
DisplayRole para desenvolvedores.
{
    Console.WriteLine($"Developer: {Username}, Email: {Email},
Lang: {ProgrammingLanguage}");
}

public void WriteCode() // Método específico de desenvolvedores.
{
    Console.WriteLine($"Developer {Username} is writing code in
{ProgrammingLanguage}.");
}
}
```

```

Program.cs
// Inheritance: User Accounts
[2] 4 usages [2] 2 inheritors
public class UserAccount
{
    [2] 7 usages
    public string Username { get; set; }

    [2] 4 usages
    public string Email { get; set; }

    [2] 3 usages
    public UserAccount(string username, string email)
    {
        Username = username;
        Email = email;
    }

    [2] 1 usage [2] 2 overrides
    public virtual void DisplayRole()
    {
        Console.WriteLine($"User: {Username}, Email: {Email}");
    }

    [2] 3 usages
    public void Login()
    {
        Console.WriteLine($"{Username} logged in.");
    }
}

[2] 2 usages
public class AdministratorAccount : UserAccount
{
    [2] 2 usages
    public string Department { get; set; }

    [2] 1 usage
    public AdministratorAccount(string username, string email, string department) : base(username, email)
    {
        Department = department;
    }

    [2] 1+1 usages
    public override void DisplayRole()
    {
        Console.WriteLine($"Administrator: {Username}, Email: {Email}, Dept: {Department}");
    }

    [2] 1 usage
    public void ManageUsers()
    {
        Console.WriteLine($"Administrator {Username} is managing users.");
    }
}

[2] 2 usages
public class DeveloperAccount : UserAccount
{
    [2] 3 usages
    public string ProgrammingLanguage { get; set; }

    [2] 1 usage
    public DeveloperAccount(string username, string email, string programmingLanguage) : base(username, email)
    {
        ProgrammingLanguage = programmingLanguage;
    }

    [2] 1+1 usages
    public override void DisplayRole()
    {
        Console.WriteLine($"Developer: {Username}, Email: {Email}, Lang: {ProgrammingLanguage}");
    }

    [2] 1 usage
    public void WriteCode()
    {
        Console.WriteLine($"Developer {Username} is writing code in {ProgrammingLanguage}.");
    }
}

```

Inheritance

4. Polimorfismo

Polimorfismo, para mim, significa "muitas formas". É a capacidade de um objeto assumir muitas formas. Na prática, isso me permite tratar objetos de diferentes classes de maneira uniforme, desde que eles compartilhem uma classe base comum ou implementem a mesma interface. Isso é fundamental para escrever código flexível e extensível.

Exemplo em C# (Interfaces `IDataSerializer`, `JsonSerializer`, `XmlSerializer`):

Defini uma interface `IDataSerializer` com métodos para serializar e desserializar dados. Em seguida, criei duas implementações concretas: `JsonSerializer` e `XmlSerializer`. Ambas as classes implementam a mesma interface, mas a forma como serializam/desserializam os dados é diferente. Isso me permite usar qualquer um dos serializadores de forma polimórfica, ou seja, posso ter uma lista de `IDataSerializer` e chamar o método `Serialize` em cada um, e o comportamento correto será executado para cada tipo de serializador.

```
// Polymorphism: Data Serializers
public interface IDataSerializer // Interface que define o contrato
para serializadores de dados.
{
    string Serialize<T>(T data);
    T Deserialize<T>(string serializedData);
}

public class JsonSerializer : IDataSerializer // Implementação para
serialização JSON.
{
    public string Serialize<T>(T data)
    {
        Console.WriteLine($"Serializing data to JSON...");
        // Em uma aplicação real, eu usaria uma biblioteca como
        System.Text.Json ou Newtonsoft.Json.
        return $"{{ \"type\": \"{typeof
// Polymorphism: Data Serializers
public interface IDataSerializer
{
    string Serialize<T>(T data);
    T Deserialize<T>(string serializedData);
}
```

```

public class JsonSerializer : IDataSerializer
{
    public string Serialize<T>(T data)
    {
        Console.WriteLine($"Serializing data to JSON...");
        return $"{{ \"type\": \"{typeof(T).Name}\", \"data\": \"{data.ToString()}\", \"type\": \"data\" }}";
    }

    public T Deserialize<T>(string serializedData)
    {
        Console.WriteLine($"Deserializing data from JSON...");
        return default(T);
    }
}

public class XmlSerializer : IDataSerializer
{
    public string Serialize<T>(T data)
    {
        Console.WriteLine($"Serializing data to XML...");
        return $"<root><type>{typeof(T).Name}</type><data>{data.ToString()}</data></root>";
    }

    public T Deserialize<T>(string serializedData)
    {
        Console.WriteLine($"Deserializing data from XML...");
        return default(T);
    }
}

(T).Name}\", \"data\": \"{data.ToString()}\", \"type\": \"data\" }}";
}

public T Deserialize<T>(string serializedData)
{
    Console.WriteLine($"Deserializing data from JSON...");
    // Em uma aplicação real, eu usaria uma biblioteca para
    analisar JSON.
    return default(T);
}

```

```

    }
}

public class XmlSerializer : IDataSerializer // Implementação para
serialização XML.
{
    public string Serialize<T>(T data)
    {
        Console.WriteLine($"Serializing data to XML...");
        // Em uma aplicação real, eu usaria uma biblioteca como
        System.Xml.Serialization.
        return $"<root><type>{typeof(T).Name}</type><data>
{data.ToString()}</data></root>";
    }

    public T Deserialize<T>(string serializedData)
    {
        Console.WriteLine($"Deserializing data from XML...");
        // Em uma aplicação real, eu usaria uma biblioteca para
        analisar XML.
        return default(T);
    }
}

```



```

Program.cs
// Polymorphism: Data Serializers
6 usages 2 inheritors
public interface IDataSerializer
{
    3 usages 2 implementations
    string Serialize<T>(T data);
    2 usages 2 implementations
    T Deserialize<T>(string serializedData);
}

2 usages
public class JsonSerializer : IDataSerializer
{
    0+3 usages
    public string Serialize<T>(T data)
    {
        Console.WriteLine($"Serializing data to JSON...");
        return $"{{ \"type\": \"{typeof(T).Name}\", \"data\": \"{data.ToString()}\", }}";
    }

    0+2 usages
    public T Deserialize<T>(string serializedData)
    {
        Console.WriteLine($"Deserializing data from JSON...");
        return default(T);
    }
}

2 usages
public class XmlSerializer : IDataSerializer
{
    0+3 usages
    public string Serialize<T>(T data)
    {
        Console.WriteLine($"Serializing data to XML...");
        return "<root><type>{typeof(T).Name}</type><data>{data.ToString()}</data></root>";
    }

    0+2 usages
    public T Deserialize<T>(string serializedData)
    {
        Console.WriteLine($"Deserializing data from XML...");
        return default(T);
    }
}

```

Polymorphism

Classe Principal

Executando o programa

Atividade - 2

Nesta seção, vou comparar as três estruturas de dados mais comuns em C#: Arrays (Matrizes), Lists (Listas) e Dictionaries (Dicionários/Mapas). É fundamental entender as diferenças entre elas para escolher a estrutura mais adequada para cada cenário no desenvolvimento de software.

Característica / Estrutura	Arrays (Matrizes)	Lists (Listas)	Dictionaries (Dicionários/Mapas)
Definição	Uma coleção de elementos de tamanho fixo e tipo homogêneo, armazenados em posições de memória contíguas. Eu os uso quando sei exatamente quantos elementos terei.	Uma coleção dinâmica de elementos de tipo homogêneo, que pode crescer ou diminuir de tamanho conforme a necessidade. É muito flexível para adicionar ou remover itens.	Uma coleção de pares chave-valor, onde cada chave é única e mapeia para um valor. Eu a uso quando preciso buscar informações rapidamente por uma chave específica.
Tamanho	Fixo (definido na inicialização). Uma vez criado, seu tamanho não pode ser alterado.	Dinâmico (pode ser redimensionado). A lista gerencia automaticamente seu tamanho interno.	Dinâmico (pode ser redimensionado). O dicionário ajusta sua capacidade conforme adiciono ou removo pares.
Acesso a Elementos	Por índice (baseado em zero). O acesso é muito rápido, pois os elementos estão em posições contíguas na memória.	Por índice (baseado em zero). O acesso também é rápido, similar aos arrays.	Por chave. O acesso é extremamente rápido, quase constante, independentemente do número de elementos.
Ordem	Mantém a ordem de inserção. Os elementos são armazenados na ordem em que são adicionados.	Mantém a ordem de inserção. Os elementos permanecem na ordem em que são adicionados.	Não garante a ordem de inserção (historicamente). A partir do .NET 5, <code>Dictionary<TKey, TValue></code> mantém a ordem de inserção, mas não devo depender disso para ordenação.
Uso Comum	Eu uso arrays quando o número de elementos é conhecido.	Eu uso listas quando o número de elementos é variável.	Eu uso dicionários quando preciso mapear chaves para valores.

um	nhecido e fixo, e preciso de alta performance para acesso direto.	ável e preciso adicionar ou remover elementos frequentemente.	para valores e buscar informações rapidamente por uma chave específica, como um registro de usuário por ID.
Vantagens	Eficiente em memória e acesso muito rápido por índice. Ideal para coleções de tamanho fixo.	Flexível, fácil de adicionar e remover elementos. Muito versátil para a maioria dos cenários.	Busca extremamente rápida por chave, chaves únicas garantem integridade. Ideal para mapeamentos.
Desvantagens	Tamanho fixo, o que torna o redimensionamento custoso (cria um novo array e copia os elementos).	Inserções ou remoções no meio da lista podem ser lentas, pois exigem o deslocamento de elementos.	Consome mais memória que arrays ou listas para armazenar as chaves e os valores.

Exemplo de Código C#

```
int[] numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
Console.WriteLine(numbers[1]); // Saída: 20
```

```
List<string> names = new List<string>();
names.Add("Alice");
names.Add("Bob");
names.Remove("Alice");
Console.WriteLine(names[0]); // Saída: Bob
```

```
Dictionary<string, int> ages = new Dictionary<string, int>();
ages.Add("Alice", 30);
ages.Add("Bob", 25);
```

```
Console.WriteLine(ages["Alice"]); // Saída: 30  
ages["Charlie"] = 35; // Adiciona ou atualiza
```

Referências

- Propriedades (Guia de Programação C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/properties>)
- Modificadores de Acesso (Guia de Programação C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers>)
- Classes e Membros Abstratos e Selados (Guia de Programação C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members>)
- Interfaces (Guia de Programação C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/fundamentals/types/interfaces>)
- Herança (Guia de Programação C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/fundamentals/object-oriented/inheritance>)
- base (Referência de C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/base>)
- Polimorfismo (Guia de Programação C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/fundamentals/object-oriented/polymorphism>)
- virtual (Referência de C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/virtual>)
- override (Referência de C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/override>)
- Arrays (Guia de Programação C#) - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/csharp/language-reference/builtin-types/arrays>)
- Classe List - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/api/system.collections.generic.list-1>)
- Classe Dictionary<TKey, TValue> - Microsoft Learn (<https://learn.microsoft.com/pt-br/dotnet/api/system.collections.generic.dictionary-2>)