UPskill – Java+.NET

Programação Orientada a Objetos – Ficheiros

UPskill
Digital Skills & Jobs

P.PORTO

EXERCÍCIO 1 – ANÁLISE DE EXEMPLO SOBRE UTILIZAÇÃO DE FICHEIRO DE TEXTO E SERIALIZAÇÃO EM JSON

Examine e, sempre que achar necessário, execute os exemplos a seguir que ilustram a utilização de ficheiros de texto e serialização em JSON[1].

1. Criar uma classe Pet.

   A classe Pet representa um animal de estimação com as propriedades Name (nome) e Age (idade). A propriedade Name é inicializada apenas no construtor e, portanto, é imutável após a criação do objeto. A classe inclui um construtor marcado com [JsonConstructor] para permitir a deserialização de instâncias de Pet diretamente de um JSON, facilitando a integração com métodos de serialização JSON.

```
using System.Text.Json.Serialization;
namespace FP_12_1;

public class Pet {
    public string? Name { get; init; }
    public int Age { get; set; }

    public Pet(string name) : this(name, 0) { }

    [JsonConstructor]  // Construtor que será utilizado na deserialização
    public Pet(string name, int age) {
        Name = name;
        Age = age;
    }
}
```

2. Criar uma classe Person.

   A classe Person representa uma pessoa com um nome (Name) e uma lista de animais de estimação (Pets). A propriedade SensitiveInfo não é serializada, protegendo dados confidenciais. A classe possui sobrecarga de construtores para criação com e sem informações

[1] https://www.json.org/json-en.html

sensíveis. Além disso, métodos AddPet e RemovePet permitem adicionar e remover animais de estimação da lista de maneira dinâmica. O método ToString retorna uma representação legível da pessoa e de seus animais, exibindo SensitiveInfo quando presente.

```
using System.Text;
using System.Text.Json.Serialization;

namespace FP_12_1;

internal class Person {
    public string? Name { get; set; }
    public List<Pet> Pets { get; set; }

    [JsonIgnore] // Propriedade que não será serializada
    public string? SensitiveInfo;

    [JsonConstructor] // Construtor que será utilizado na deserialização
    public Person(string Name, List<Pet>? Pets = null) {
        this.Name = Name;
        this.Pets = Pets ?? new List<Pet>();
    }

    public Person(string Name, string SensitiveInfo) {
        this.Name = Name;
        this.Pets = new List<Pet>();
        this.SensitiveInfo = SensitiveInfo;
    }

    public void AddPet(Pet pet) {
        Pets?.Add(pet);
    }

    public void RemovePet(Pet pet) {
        Pets?.Remove(pet);
    }

    public override string ToString() {
```

```
        var result = new StringBuilder();
        result.Append($"{Name} has {Pets.Count} pets:");
        for (int i = 0; i < Pets.Count; i++) {
            result.Append($"\n{Pets[i].Name}, {Pets[i].Age} years old");
        }
        if (SensitiveInfo != null) {
            result.Append($"\nSensitive Info: {SensitiveInfo}");
        }

        return result.ToString();
    }
}
```

3. Criar uma classe para guardar e carregar uma instância de Person em ficheiro de texto.

   A classe TextFile permite guardar e carregar informações de uma instância Person num arquivo de texto. O método SaveToTextFile escreve o nome da pessoa e os detalhes de cada animal de estimação no ficheiro. O método LoadFromTextFile lê essas informações e recria a instância Person, incluindo os animais de estimação listados. Esse formato simplificado de escrita em texto é uma alternativa leve ao JSON, mas requer uma estrutura de arquivo específica.

```
namespace FP_12_1;

internal static class TextFile {
    public static void SaveToTextFile(Person person, string filePath) {
        using (StreamWriter writer = new StreamWriter(filePath)) {
            writer.WriteLine($"Person: {person.Name}");
            if (person.Pets != null) {
                foreach (var pet in person.Pets) {
                    writer.WriteLine($"Pet: {pet.Name}, Age: {pet.Age}");
                }
            }
        }
    }

    public static Person LoadFromTextFile(string filePath) {
```

```
        using (StreamReader reader = new StreamReader(filePath)) {
            string? firstLine = reader.ReadLine();

            if (firstLine == null || !firstLine.StartsWith("Person:")) {
                throw new InvalidOperationException("Invalid file format.");
            }

            string name = firstLine.Substring(8).Trim();
            Person person = new Person(name);

            string? line;
            while ((line = reader.ReadLine()) != null) {
                if (line.StartsWith("Pet:")) {
                    string[] petInfo = line.Split(',');
                    string petName = petInfo[0].Split(':')[1].Trim();
                    int petAge = int.Parse(petInfo[1].Split(':')[1].Trim());
                    person.AddPet(new Pet(petName, petAge));
                }
            }
            return person;
        }
    }

}
```

4. Criar uma classe para serialização e deserialização de uma instância de Person.

   A classe JsonSerialization facilita a serialização de uma instância Person para JSON e a deserialização de um JSON para uma instância Person. O método SerializeJson converte o objeto numa string JSON formatada e a guarda num ficheiro. O método DeserializeJson lê o JSON do arquivo e recria a instância Person, permitindo a persistência de dados num formato estruturado.

```
using System.Text.Json;

namespace FP_12_1;
internal static class JsonSerialization {
```

```
    public static void SerializeJson(Person person, string filePath) {
        string json = System.Text.Json.JsonSerializer.Serialize(person, new
JsonSerializerOptions { WriteIndented = true });
        File.WriteAllText(filePath, json);
    }

    public static Person? DeserializeJson(string filePath) {
        string json = File.ReadAllText(filePath);
        return System.Text.Json.JsonSerializer.Deserialize<Person>(json);
    }
}
```

5. Criar uma classe para o sistema de logs.

   A classe Logger registra eventos importantes numa lista de mensagens de log com marcação de data e hora. Os logs podem ser exportados para um ficheiro usando o método ExportToFile ou exibidos na console com o método DisplayLogs. Esses logs são úteis para monitorização e diagnóstico. Esta abordagem não é ideal para aplicações reais. Em cenários profissionais, é recomendável utilizar bibliotecas especializadas e mais robustas para *logging*, como NLog, Serilog ou log4net. Essas bibliotecas oferecem recursos avançados, como diferentes níveis de log (informação, aviso, erro, etc.), formatação personalizável, e integração com sistemas de monitorização e visualização de logs. Essas ferramentas garantem maior flexibilidade, desempenho e facilidades de configuração e manutenção dos logs.

```
namespace FP_12_1;

internal class Logger {
    private readonly List<string> _logMessages;

    public Logger() {
        _logMessages = new List<string>();
    }

    public void Log(string message) {
```

```
        _logMessages.Add($"{DateTime.Now:yyyy-MM-
dd HH:mm:ss} - {message}");
    }

    public void ExportToFile(string filePath) {
        try {
            File.WriteAllLines(filePath, _logMessages);
            Console.WriteLine($"Logs exportados para o arquivo: {filePath}"
);
        } catch (Exception ex) {
            Console.WriteLine($"Erro ao exportar os logs: {ex.Message}");
        }
    }

    public void DisplayLogs() {
        foreach (var log in _logMessages) {
            Console.WriteLine(log);
        }
    }
}
```

6. Exemplificação do uso das classes.

```
using FP_12_1;

Logger logger = new Logger();

string fileName;
string BaseDirectory = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
"Data");

Directory.CreateDirectory(BaseDirectory);

var person = new Person("John", "12345");
person.AddPet(new Pet("Fluffy", 3));
person.AddPet(new Pet("Spot", 2));
```

```
person.AddPet(new Pet("Rex", 5));

logger.Log("Person created");

Console.WriteLine("Person:");
Console.WriteLine(person);
Console.WriteLine();

logger.Log("Person displayed");

// Gravação em Texto
fileName = Path.Combine(BaseDirectory, "Person.txt");
TextFile.SaveToTextFile(person, fileName);
Person personFromText = TextFile.LoadFromTextFile(fileName);

logger.Log("Person saved and recovered form text file");

Console.WriteLine("Person from text file:");
Console.WriteLine(personFromText);
Console.WriteLine();

logger.Log("Person displayed");

// Serialização JSON
fileName = Path.Combine(BaseDirectory, "Person.json");
JsonSerialization.SerializeJson(person, fileName);
var personFromJson = JsonSerialization.DeserializeJson(fileName);

logger.Log("Person serialized and deserialized");

Console.WriteLine("Person from JSON:");
Console.WriteLine(personFromJson);
Console.WriteLine();

logger.Log("Person displayed");
```

EXERCÍCIO 2 – SISTEMA DE GESTÃO DE BIBLIOTECA COM SERIALIZAÇÃO, INTERFACES E EXCEÇÕES

Desenvolver uma aplicação em C# que utilize serialização em JSON para armazenar e recuperar dados de uma classe que contém uma lista de outra classe. A aplicação deve aplicar conceitos de interfaces e tratamento de exceções para garantir a robustez do sistema. Além disso, deverá incluir um sistema de logs para registar as operações realizadas e exportá-los para um ficheiro de texto.

Descrição do Problema

Neste exercício, deve criar um sistema para gerenciar uma biblioteca. O sistema permitirá que informações sobre livros e leitores sejam guardadas num ficheiro JSON e recuperadas posteriormente. A estrutura de dados incluirá três classes principais: Livro, Leitor e Biblioteca. A biblioteca armazena uma lista de leitores, e cada leitor pode ter uma lista de livros que está a ler.

Requisitos do Sistema

1. **Classes e Interfaces**:
    - Crie a interface IEntidade com as seguintes propriedades:
        - Id (string) - Identificador único da entidade.
        - ObterInfo() - Método que retorna uma breve descrição da entidade.
    - Crie a classe Livro, que implementa IEntidade:
        - Propriedades:
            - Id (string) - ID único do livro.
            - Titulo (string) - Título do livro.
            - Autor (string) - Autor do livro.
            - Ano (int) - Ano de publicação.
        - Construtor para inicializar o ID, título, autor e ano.
        - Implementação do método ObterInfo(), que retorna uma string com o título e o autor do livro.
    - Crie a classe Leitor, que também implementa IEntidade:
        - Propriedades:
            - Id (string) - ID único do leitor.
            - Nome (string) - Nome do leitor.

        - Livros (List<Livro>) - Lista de livros que o leitor está a ler.
    - Construtor para inicializar o ID e o nome.
    - Método AdicionarLivro(Livro livro), que adiciona um livro à lista do leitor.
    - Método RemoverLivro(Livro livro), que remove um livro da lista.
    - Implementação do método ObterInfo(), que retorna uma string com o nome do leitor e o número de livros que está a ler.
  - Crie a classe Biblioteca:
    - Propriedades:
      - Leitores (List<Leitor>) - Lista de leitores registados na biblioteca.
    - Métodos:
      - RegistrarLeitor(Leitor leitor), que adiciona um leitor à biblioteca.
      - RemoverLeitor(Leitor leitor), que remove um leitor da biblioteca.
      - ProcurarLeitorPorId(string id), que busca um leitor pelo ID e lança uma exceção personalizada LeitorNaoEncontradoException se o leitor não for encontrado.

2. **Sistema de Logs**:
   - Crie uma classe Logger para registar todas as operações realizadas no sistema (ex: registo e remoção de leitores, adição e remoção de livros).
   - Cada entrada de log deve incluir a data e hora da operação e um nível de log, e.g., Info, Warning, Error.
   - Implemente um método ExportarLogs(string filePath) para guardar o log num ficheiro de texto.
3. **Serialização e Desserialização em JSON**:
   - Implemente a serialização JSON para salvar o estado atual da biblioteca (leitores e livros) num ficheiro.
   - Implemente a desserialização JSON para carregar os dados do ficheiro e restaurar o estado da biblioteca.
4. **Tratamento de Exceções**:
   - Crie uma exceção personalizada LeitorNaoEncontradoException, que deve ser lançada quando um leitor procurado não for encontrado na biblioteca.
   - Use blocos try-catch para capturar e registar exceções no sistema de logs.
5. **Exemplos de Implementação**

1. **Registo de Leitor**:
    - Crie um novo leitor e adicione-o à biblioteca usando o método RegistrarLeitor.
    - Adicione livros ao leitor usando AdicionarLivro.
2. **Remoção de Leitor**:
    - Tente remover um leitor da biblioteca pelo ID usando ProcurarLeitorPorId.
    - Capture a exceção LeitorNaoEncontradoException se o leitor não existir e registe no log.
3. **Serialização e Exportação de Dados**:
    - Salve a lista atual de leitores e livros num ficheiro JSON usando a serialização.
    - Exporte os logs para um ficheiro de texto no final da execução.