

업무에서 다양한 C# 활용법 (with KICE 준비)

1. 강의 개요

비즈니스에서 자주 활용되는 C#의 주요 기능을 알아보며 실습을 통해 나만의 기능을 만들어 KICE 시험에 대비 할 수 있습니다.

2. C#의 특성 및 비즈니스에서 활용하는 주요 라이브러리

- 2.01 [C# 특성](#)
- 2.02 [주요 기능 및 라이브러리](#)

3. 비즈니스에서 주로 사용하는 기능 및 사용 예

- 3.01 [Console, WPF, WinForm을 다양하게 활용 하는 법](#)
- 3.02 [파일 입출력](#)
- 3.03 [네트워크 프로그래밍](#)
- 3.04 [쓰레드 / Task](#)
- 3.05 [자주 사용하는 자료형 \(List, Array, Dictionary, HashSet\)](#)
- 3.06 [LINQ 다루기](#)
- 3.07 [UTF / ANSI 변환 / Unicode 변환](#)
- 3.08 [Json / Xml 다루기](#)
- 3.09 [DB처리](#)
- 3.10 [암호화 및 보안 처리](#)
- 3.11 [다양한 실무 알고리즘](#)
- 3.12 [기타 / 로깅](#)

4. KICE 대비

- 4.01 [KICE 시험 유형 및 대비 방법](#)
- 4.02 [시험문제 풀이](#)
- 시험문제 다운로드 (Pilot)

시험문제 프로젝트 다운로드

https://drive.google.com/file/d/1IZYrylwAA7jMSmp05KjvsBc50_gDqbd/view?usp=sharing

5. 마무리

2.01 - C#의 특성

C#(C-Sharp)은 마이크로소프트가 개발한 **객체 지향 프로그래밍 언어**로, .NET 플랫폼 위에서 실행됨.

C#은 Java와 C++의 장점을 가져오되, 더 간결하고 안정적으로 설계되어 **개발 생산성, 안전성, 성능** 등을 고려한 현대적인 언어.

C#의 역사 [바로가기](#)

C#의 주요 특성과 설명:

1. 객체 지향 프로그래밍(OOP) 지원

- 클래스(class), 객체(object), 상속(inheritance), 다형성(polymorphism), 캡슐화(encapsulation) 등의 개념을 완전히 지원.
- 인터페이스와 추상 클래스를 통한 설계 유연성 제공.

```
public interface IAnimal { void Speak(); }

public class Dog : IAnimal { public void Speak() => Console.WriteLine("멍멍"); }
```

2. 강력한 형식(type safety)과 정적 타입(static typing)

- 컴파일 타임에 타입 오류를 잡을 수 있어 **안정적인 코드 작성** 가능.
- 명시적 타입뿐 아니라 `var` 키워드로 **암시적 타입 추론**도 지원.

```
int number = 10;
var message = "Hello"; // string으로 추론
```

3. 메모리 관리 (가비지 컬렉션)

- `new`로 생성한 객체는 자동으로 가비지 컬렉터(GC)가 관리.
- 개발자가 직접 메모리를 해제할 필요가 없어 **메모리 누수 위험이 적음**.

4. LINQ (Language Integrated Query)

- SQL과 유사한 문법으로 컬렉션, 배열, XML, 데이터베이스 등에 대해 **질의(Query)** 가능.
- 코드의 **가독성과 생산성 향상**에 기여.

```
var adults = people.Where(p => p.Age >= 18).ToList();
```

5. 이벤트 및 델리게이트(Delegate)

- 콜백 함수를 구현할 수 있는 델리게이트.
- UI 및 비동기 작업에서 중요한 이벤트 기반 프로그래밍 지원.

```
public delegate void Notify();  
public event Notify OnProcessCompleted;
```

6. 비동기 프로그래밍 (async / await)

- 비동기 메서드로 **비차단(non-blocking)** 코드 작성 가능.
- `Task` 기반의 병렬 프로그래밍을 쉽게 구현.

```
public async Task<string> DownloadAsync(string url)  
{  
    using var client = new HttpClient();  
    return await client.GetStringAsync(url);  
}
```

7. 풍부한 표준 라이브러리와 프레임워크

- .NET Base Class Library(BCL)를 통해 다양한 기능 제공:
 - 파일 입출력, 네트워크, 컬렉션, 스레드, 암호화 등
- ASP.NET, WPF, WinForms, Xamarin, MAUI 등 다양한 응용 분야 지원

8. 속성(Attribute) 기반 메타프로그래밍

- 클래스, 메서드, 필드 등에 부가 정보를 선언하여 런타임이나 컴파일 타임에 반영.
- 예: `[Obsolete]`, `[Serializable]`, `[DataContract]`

```
[Obsolete("이 메서드는 곧 제거됩니다.")]  
public void OldMethod() { }
```

9. 안전성 & 보안

- 널 참조 검사, 예외 처리, 접근 제한자 등으로 안정성 확보.
- `try-catch-finally` 구조를 통한 예외 안전한 코드 작성.

10. 플랫폼 독립성 (.NET Core/.NET 5 이상)

- Windows에만 국한되지 않고 **Linux, macOS** 등 다양한 OS에서 실행 가능.
- .NET 6 이상에서는 단일 파일 배포, AOT 컴파일 등 고성능 지원.

요약

특성	설명
객체지향 지원	클래스, 상속, 다형성
정적 타입, 타입 안정성	컴파일 타임 검증
메모리 관리	GC 자동 관리
LINQ	통합 쿼리 지원
델리게이트 & 이벤트	이벤트 중심 프로그래밍
async/await	비동기 처리 간편
풍부한 라이브러리	.NET BCL 지원
속성(Attribute) 시스템	메타데이터 기반 프로그래밍
플랫폼 독립 실행 가능	크로스 플랫폼 지원

2.02 - C# 비즈니스 개발자를 위한 핵심 기능 및 라이브러리

1. C# 핵심 언어 기능

기능 / 기술	설명	주요 용도 / 대표 API
LINQ	컬렉션 및 데이터 쿼리	<code>var result = list.Where(x => x.Age > 30).ToList();</code>
async / await	비동기 프로그래밍	<code>await HttpClient.GetAsync(...)</code>
Thread / Task	병렬 처리	<code>Task.Run(...)</code> , <code>Parallel.For(...)</code>
Timer	주기적 작업 실행	<code>System.Timers.Timer</code>
Exception Handling	예외 안정성 확보	<code>try-catch-finally</code>
Dependency Injection	의존성 주입	<code>IServiceCollection.AddScoped<></code>

2. 웹 개발 (서버 & 클라이언트)

서버: ASP.NET Core 기반

구성요소	설명
ASP.NET Core Web API	고성능 RESTful 서버 구축
Controller & Routing	<code>[ApiController]</code> , <code>[HttpGet]</code>
Model Validation	<code>FluentValidation</code> , <code>DataAnnotations</code>
Swagger	API 문서 자동 생성
Middleware	인증, 로깅, 에러 처리 구성 가능

```
[ApiController]
[Route("api/[controller]")]
public class ProductController : ControllerBase
{
    [HttpGet("{id}")]
    public IActionResult Get(int id) => Ok(new { Id = id, Name = "Item" });
}
```

클라이언트

기술	설명
HttpClient	REST API 호출 기본 객체
Refit / RestSharp	선언형 API 호출 or 고급 인증/요청 지원
Polly	재시도, 회로차단 등 실패 복구 처리

```
var client = new HttpClient();
var result = await client.GetStringAsync("https://api.example.com/items");
```

3. 데이터 접근

기술	설명
Entity Framework Core	마이크로소프트 공식 ORM, Code-First
Dapper	경량 고성능 ORM
ADO.NET	저수준 SQL 제어 (SqlCommand, SqlConnection)

4. 로깅 및 모니터링

라이브러리	설명
Serilog	구조화 로그, 다양한 출력 지원 (파일, 콘솔 등)
NLog	다양한 타겟, 유연한 설정
log4net	오래된 시스템과의 호환성 우수

```
Log.Logger = new LoggerConfiguration().WriteTo.Console().CreateLogger();
Log.Information("서버 시작됨");
```

5. 테스트 및 품질

라이브러리	설명
xUnit	기본 단위 테스트 프레임워크
Moq	인터페이스/서비스 모킹 가능

```
var mockService = new Mock<IOrderService>();
mockService.Setup(s => s.GetOrder(1)).Returns(new Order { Id = 1 });
```

6. 문서화 & 보고서 자동화

도구	설명
Swagger/OpenAPI	API 문서 자동화 및 테스트
ClosedXML / EPPlus	Excel 보고서 자동 생성
PdfSharp / iText7	PDF 계약서, 송장 출력
OpenXML SDK	Word 문서 템플릿 처리
RazorLight	HTML 템플릿 기반 PDF 생성

7. UI & 플랫폼 연동

플랫폼	설명
WPF / WinForms	데스크탑 앱 제작
MAUI / Uno Platform	크로스 플랫폼 UI 앱
Win32 API (P/Invoke)	메시지 후킹, 창 제어 등

```
[DllImport("user32.dll")]
public static extern int MessageBox(IntPtr hwnd, string text, string caption,
uint type);

MessageBox(IntPtr.Zero, "Hello world", "Win32 호출", 0);
```

8. 시스템 통합 & 보안

범주	설명
SignalR	실시간 이벤트 처리 (대시보드, 채팅 등)
gRPC / Named Pipes	고성능 시스템 간 통신
RabbitMQ / Kafka	메시지 기반 비동기 처리
IdentityServer4 / Duende	OAuth2, OpenID 기반 인증
BCrypt.Net / DPAPI	암호화 및 패스워드 해싱 처리

9. 백그라운드 작업 & 스케줄링

기술	설명
Hangfire / Quartz.NET	예약 작업 및 반복 실행 지원
Windows Task Scheduler 연동	시스템 예약 작업 등록
MailKit / Smtplib	이메일 전송 기능

실무 예제 조합

시나리오	기술 조합
웹 API + DB + 인증	ASP.NET Core + EF Core + Swagger + FluentValidation + IdentityServer
고성능 API 호출	HttpClient + Polly + Dapper
데스크탑 보고서 도구	WinForms + ClosedXML + PDFSharp + Win32 API
예약 이메일 발송	Hangfire + MailKit + Serilog
실시간 대시보드	ASP.NET Core + SignalR + Blazor/WPF
단위 테스트 & 품질 관리	xUnit + Moq + Serilog

3.01 - C# 플랫폼별 애플리케이션 활용 (Console / WinForm / WPF)

1. 개요

목적

- C#의 대표적 UI 및 비 UI 애플리케이션 플랫폼인 Console, WinForms, WPF의 개념과 차이를 이해한다.
- 각 플랫폼의 장단점과 활용 사례를 통해 실제 업무에 맞는 선택 기준을 제시한다.
- 실습을 통해 기본적인 애플리케이션을 직접 개발해본다.

2. 플랫폼별 개요 및 특징

플랫폼	UI 여부	주요 용도	특징
Console	비 UI	시스템 유틸리티, 빠른 테스트	빠르고 가벼움, 텍스트 입출력
WinForms	UI	전통적인 데스크탑 앱	이벤트 기반, 쉬운 UI 작성
WPF	UI	현대적인 데스크탑 앱	XAML 기반, 강력한 데이터 바인딩, 스타일링 가능

3. Console Application

3.1 특징

- 가장 간단한 C# 응용프로그램 유형
- 입출력은 `Console.WriteLine`, `Console.ReadLine` 중심

3.2 실습 예제

```
Console.WriteLine("이름을 입력하세요:");
string name = Console.ReadLine();
Console.WriteLine($"안녕하세요, {name}님!");
```

3.3 활용 팁

- 디버깅용 유틸리티 제작
- 자동화 스크립트 실행용 CLI 툴

4. WinForms Application

4.1 특징

- Windows Forms 기반 GUI 개발
- Drag & Drop 디자이너 지원
- `Button`, `TextBox`, `Label` 등 컨트롤 사용

4.2 실습 예제

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        Button btn = new Button() { Text = "눌러보세요" };
        btn.Click += (s, e) => MessageBox.Show("클릭됨!");
        Controls.Add(btn);
    }
}
```

4.3 사용 시 유의점

- 복잡한 UI 구성에는 한계
- MVVM 구조를 적용하기 어려움

5. WPF Application

5.1 특징

- XAML(선언적 UI 언어) 기반
- MVVM 패턴과 잘 어울림
- 데이터 바인딩, 스타일링, 애니메이션 기능 우수

5.2 기본 XAML + C# 예제

MainWindow.xaml

```
<window x:class="HelloWPF.Mainwindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Title="WPF 예제" Height="200" Width="300">
    <StackPanel>
        <TextBox Name="txtName" Margin="10"/>
        <Button Content="인사하기" Margin="10" Click="Button_Click"/>
    </StackPanel>
</window>
```

MainWindow.xaml.cs

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show($"안녕하세요, {txtName.Text}님!");
}
```

5.3 장점 및 유의사항

- UI/UX가 중요한 애플리케이션에 적합
- 학습 난이도는 다소 높음 (XAML, MVVM 이해 필요)

6. 플랫폼 비교

항목	Console	WinForms	WPF
개발 속도	빠름	중간	느림
유지보수	쉬움	중간	구조화하면 용이
UI 표현력	없음	제한적	풍부함
학습 난이도	낮음	낮음	높음
대표 패턴	절차지향	이벤트 중심	MVVM

7. 활용 전략

플랫폼별 특징

- Console은 로직 중심 학습용 또는 CLI 도구 제작에 적합
- WinForms는 빠른 UI 프로토타입 제작에 효과적
- WPF는 대규모 데스크탑 앱에 적합한 현대적 구조 제공

개발자 입장에서의 선택 기준

- UI 복잡성이 낮고 빠르게 결과물을 만들어야 할 때는 **WinForms**
- 확장성과 유지보수가 중요하거나 UX가 중요한 경우는 **WPF**
- 백엔드 알고리즘 테스트, 자동화, CLI 도구는 **Console**

8. 마무리 및 실습 과제

실습 과제

1. Console App으로 사칙연산 계산기 제작
2. WinForms로 간단한 주소록 앱 구현 (ListView 포함)
3. WPF로 로그인 창 및 사용자 대시보드 샘플 제작

1. Console 사칙연산 계산기

```
=== C# Console 계산기 ===
첫 번째 숫자를 입력하세요: 100
두 번째 숫자를 입력하세요: 50
연산자를 입력하세요 (+, -, *, /): *
결과: 100 * 50 = 5000
```

설명

항목	내용
입력	<code>Console.ReadLine()</code> 으로 숫자와 연산자 입력
예외 처리	잘못된 숫자 또는 0으로 나누기 방지
계산 처리	<code>switch</code> 구문으로 연산자에 따라 계산 수행

2. WinForms 주소록 앱 구성

기능

- 이름(Name), 전화번호(Phone) 입력
- [추가] 버튼으로 ListView에 주소록 항목 추가
- [삭제] 버튼으로 선택된 항목 삭제

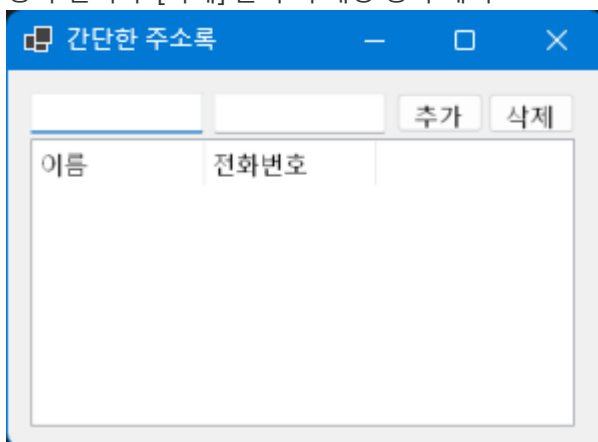
폼 디자인 구성 (MainForm.cs Designer 기준)

- `TextBox` : `txtName`, `txtPhone`
- `Button` : `btnAdd`, `btnDelete`
- `ListView` : `lvAddressBook`

폼 디자이너에서 미리 UI 요소를 배치하거나, 코드로 작성해도 무관.

실행 예시 시나리오

1. 이름: 홍길동, 전화번호: 010-1234-5678 입력 후 [추가] 클릭
2. ListView에 항목 추가됨
3. 항목 선택 후 [삭제] 클릭 시 해당 항목 제거



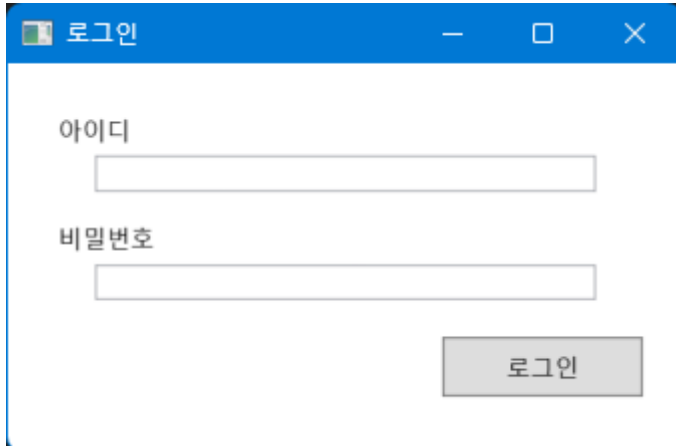
3. WPF 로그인 및 환영 대시보드 구성

윈도우 구성

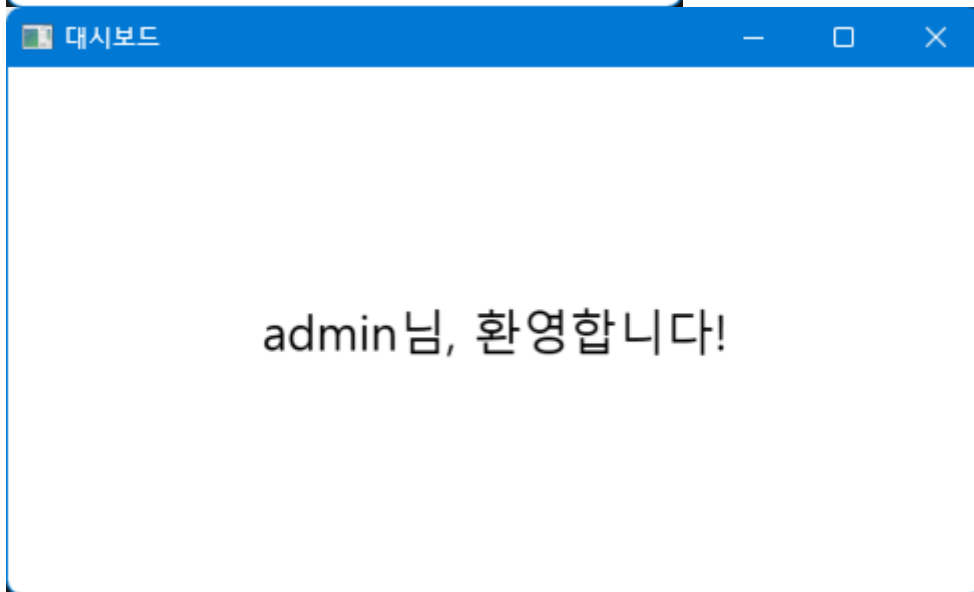
1. **LoginWindow.xaml** - 사용자 ID/비밀번호 입력
2. **DashboardWindow.xaml** - 로그인 성공 시 표시되는 메인 대시보드

실행 시나리오

1. 앱 실행 시 로그인 창 표시
2. ID: `admin`, PW: `1234` 입력 시 로그인 성공
3. 대시보드 창으로 이동, 환영 메시지 출력
4. 실패 시 경고 메시지 출력



A screenshot of a Windows application window titled '로그인' (Login). The window has a blue title bar with standard minimize, maximize, and close buttons. The main content area is white and contains two text input fields. The first field is labeled '아이디' (ID) and the second is labeled '비밀번호' (Password). Below the password field is a gray button labeled '로그인' (Login).



3.02 - C# 파일 입출력 주요 방법

방법	설명	특징
<code>File.ReadAllText</code> , <code>File.WriteAllText</code>	전체 텍스트 파일을 한 번에 읽거나 씀	간단한 텍스트 파일에 적합
<code>File.ReadAllLines</code> , <code>File.WriteAllLines</code>	한 줄씩 배열로 읽거나 씀	줄 단위 처리에 유리
<code>FileStream</code>	바이트 단위의 파일 읽기/쓰기	바이너리 파일, 대용량 처리에 적합
<code>StreamReader</code> , <code>StreamWriter</code>	텍스트 기반 스트림 처리	성능 우수, 줄 단위/문자 단위 처리
<code>BinaryReader</code> , <code>BinaryWriter</code>	바이너리 데이터 입출력	구조체, 수치, 문자열 등 바이너리 저장
<code>FileInfo</code> , <code>DirectoryInfo</code>	파일/디렉토리 조작 클래스	파일 속성 조작에 유리

1. `File.ReadAllText` & `File.WriteAllText`

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string path = "sample.txt";
        File.WriteAllText(path, "Hello, File I/O in C#!");

        string content = File.ReadAllText(path);
        Console.WriteLine("읽은 내용: " + content);
    }
}
```

장점: 간단하고 직관적

주의: 대용량 파일에 부적합

2. `File.ReadAllLines` & `File.WriteAllLines`

```
using System;
using System.IO;

class Program
{

```

```

static void Main()
{
    string path = "lines.txt";
    string[] lines = { "첫 번째 줄", "두 번째 줄", "세 번째 줄" };
    File.WriteAllLines(path, lines);

    string[] readLines = File.ReadAllLines(path);
    foreach (string line in readLines)
    {
        Console.WriteLine(line);
    }
}

```

3. StreamReader & StreamWriter

```

using System;
using System.IO;

class Program
{
    static void Main()
    {
        string path = "stream.txt";
        using (StreamWriter writer = new StreamWriter(path))
        {
            writer.WriteLine("StreamWriter를 사용한 첫 줄");
            writer.WriteLine("두 번째 줄");
        }

        using (StreamReader reader = new StreamReader(path))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                Console.WriteLine("읽은 줄: " + line);
            }
        }
    }
}

```

장점: 파일 크기에 상관없이 효율적

주의: 파일 닫힘 (Dispose) 주의 - using으로 처리

4. FileStream (바이너리 파일)

```

using System;
using System.IO;
using System.Text;

class Program
{

```

```

static void Main()
{
    string path = "binary.dat";

    using (FileStream fs = new FileStream(path, FileMode.Create))
    {
        byte[] data = Encoding.UTF8.GetBytes("Binary File Example");
        fs.Write(data, 0, data.Length);
    }

    using (FileStream fs = new FileStream(path, FileMode.Open))
    {
        byte[] buffer = new byte[fs.Length];
        fs.Read(buffer, 0, buffer.Length);
        string result = Encoding.UTF8.GetString(buffer);
        Console.WriteLine("읽은 내용: " + result);
    }
}

```

장점: 이미지, 바이너리 파일 처리에 적합

주의: 바이트 처리 로직 필요

5. BinaryWriter & BinaryReader

```

using System;
using System.IO;

class Program
{
    static void Main()
    {
        string path = "data.bin";

        using (BinaryWriter writer = new BinaryWriter(File.Open(path,
        FileMode.Create)))
        {
            writer.Write(1234);           // 정수
            writer.Write(3.14);           // 실수
            writer.Write("Hello Binary!"); // 문자열
        }

        using (BinaryReader reader = new BinaryReader(File.Open(path,
        FileMode.Open)))
        {
            int number = reader.ReadInt32();
            double pi = reader.ReadDouble();
            string text = reader.ReadString();

            Console.WriteLine($"정수: {number}, 실수: {pi}, 문자열: {text}");
        }
    }
}

```

6. FileInfo / DirectoryInfo

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        FileInfo file = new FileInfo("info.txt");
        using (StreamWriter writer = file.CreateText())
        {
            writer.WriteLine("FileInfo로 파일 생성");
        }

        Console.WriteLine($"파일 크기: {file.Length} bytes");
        Console.WriteLine($"생성 시간: {file.CreationTime}");
    }
}
```

정리: 상황별 추천

목적	추천 방법
간단한 텍스트 파일 입출력	<code>File.ReadAllText</code> , <code>File.WriteAllText</code>
줄 단위 텍스트	<code>File.ReadAllLines</code> , <code>File.WriteAllLines</code> , <code>StreamReader</code>
바이너리 파일	<code>FileStream</code> , <code>BinaryWriter</code> , <code>BinaryReader</code>
대용량 텍스트 파일	<code>StreamReader</code> , <code>StreamWriter</code>
파일 속성 확인 및 조작	<code>FileInfo</code> , <code>DirectoryInfo</code>

3.03 - C# 네트워크 프로그래밍 종류

종류	설명	주요 클래스
1. TCP 통신	연결 지향적인 안정적인 스트림 기반 통신	TcpClient, TcpListener, NetworkStream
2. UDP 통신	연결 없는 빠른 통신, 신뢰성 낮음	UdpClient
3. HTTP 통신	웹 서버와 REST API 통신	HttpClient, HttpWebRequest
4. WebSocket	실시간 양방향 통신 (웹 기반)	ClientWebSocket
5. REST API 서버	HTTP 기반의 웹 API 제공	ASP.NET Core Web API

1. TCP 통신

서버 (TcpListener)

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpServer
{
    static void Main()
    {
        TcpListener server = new TcpListener(IPAddress.Any, 9000);
        server.Start();
        Console.WriteLine("Server started...");

        TcpClient client = server.AcceptTcpClient();
        NetworkStream stream = client.GetStream();

        byte[] buffer = new byte[1024];
        int bytes = stream.Read(buffer, 0, buffer.Length);
        Console.WriteLine("Received: " + Encoding.UTF8.GetString(buffer, 0, bytes));

        stream.Write(Encoding.UTF8.GetBytes("Hello client!"));
        client.Close();
        server.Stop();
    }
}
```

클라이언트 (TcpClient)

```

using System;
using System.Net.Sockets;
using System.Text;

class TcpClientApp
{
    static void Main()
    {
        TcpClient client = new TcpClient("127.0.0.1", 9000);
        NetworkStream stream = client.GetStream();

        byte[] data = Encoding.UTF8.GetBytes("Hello Server!");
        stream.Write(data, 0, data.Length);

        byte[] buffer = new byte[1024];
        int bytes = stream.Read(buffer, 0, buffer.Length);
        Console.WriteLine("Server response: " + Encoding.UTF8.GetString(buffer,
0, bytes));

        client.Close();
    }
}

```

2. UDP 통신

서버 (UdpServer)

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

class UdpServer
{
    static void Main()
    {
        UdpClient server = new UdpClient(9001);
        IPEndPoint remoteEP = new IPEndPoint(IPAddress.Any, 0);

        byte[] data = server.Receive(ref remoteEP);
        Console.WriteLine("Received: " + Encoding.UTF8.GetString(data));

        byte[] response = Encoding.UTF8.GetBytes("UDP Response!");
        server.Send(response, response.Length, remoteEP);
    }
}

```

클라이언트 (UdpClient)

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

```

```

class UdpClientApp
{
    static void Main()
    {
        UdpClient client = new UdpClient();
        byte[] data = Encoding.UTF8.GetBytes("Hello via UDP");

        client.Send(data, data.Length, "127.0.0.1", 9001);

        IPEndPoint serverEP = new IPEndPoint(IPAddress.Any, 0);
        byte[] received = client.Receive(ref serverEP);
        Console.WriteLine("Server: " + Encoding.UTF8.GetString(received));
    }
}

```

3. HTTP HttpClient / RestSharp / HttpListener (REST API)

HttpClient

```

using System;
using System.Net.Http;
using System.Threading.Tasks;

class HttpExample
{
    static async Task Main()
    {
        HttpClient client = new HttpClient();
        HttpResponseMessage response = await
client.GetAsync("https://jsonplaceholder.typicode.com/posts/1");
        string result = await response.Content.ReadAsStringAsync();

        Console.WriteLine(result);
    }
}

```

RestSharp (Nuget 설치)

```

using RestSharp;
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        var client = new RestClient("https://jsonplaceholder.typicode.com");
        var request = new RestRequest("posts", Method.Post);

        request.AddJsonBody(new
        {
            title = "foo",

```

```

        body = "bar",
        userId = 1
    });

    var response = await client.ExecuteAsync(request);

    Console.WriteLine($"Status: {response.StatusCode}");
    Console.WriteLine($"Body:\n{response.Content}");
}
}

```

HttpListener

```

using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading.Tasks;

class SimpleRestServer
{
    private static readonly HttpListener listener = new HttpListener();

    static async Task Main(string[] args)
    {
        string urlPrefix = "http://localhost:5000/";
        listener.Prefixes.Add(urlPrefix);
        listener.Start();
        Console.WriteLine($"[서버 시작] {urlPrefix}");

        while (true)
        {
            var context = await listener.GetContextAsync();
            _ = Task.Run(() => HandleRequest(context));
        }
    }

    private static void HandleRequest(HttpListenerContext context)
    {
        string method = context.Request.HttpMethod;
        string path = context.Request.Url.AbsolutePath;

        Console.WriteLine($"[{DateTime.Now}] {method} {path}");

        if (path == "/hello" && method == "GET")
        {
            string responseText = "{\"message\": \"Hello, world!\"}";
            WriteResponse(context, 200, responseText, "application/json");
        }
        else if (path == "/echo" && method == "POST")
        {
            using var reader = new StreamReader(context.Request.InputStream,
            context.Request.ContentEncoding);
            string body = reader.ReadToEnd();

            string responseText = $"{{\"you_sent\": {body}}}\";
            WriteResponse(context, 200, responseText, "application/json");
        }
    }
}

```

```

    }
    else
    {
        WriteResponse(context, 404, "{\"error\": \"Not Found\"}",
"application/json");
    }
}

private static void WriteResponse(HttpListenerContext context, int
statusCode, string content, string contentType)
{
    context.Response.StatusCode = statusCode;
    context.Response.ContentType = contentType;
    byte[] buffer = Encoding.UTF8.GetBytes(content);
    context.Response.ContentLength64 = buffer.Length;
    context.Response.OutputStream.Write(buffer, 0, buffer.Length);
    context.Response.Close();
}
}

```

사전 조건

- 운영체제에서 직접 HTTP 요청을 수신함 (Windows 전용)
- 관리자 권한으로 실행해야 합니다 (포트 등록 필요)

4. WebSocket 클라이언트

```

using System;
using System.Net.WebSockets;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

class WebSocketClient
{
    static async Task Main()
    {
        using var ws = new ClientWebSocket();
        await ws.ConnectAsync(new Uri("wss://echo.websocket.org"),
CancellationToken.None);

        string message = "Hello WebSocket!";
        byte[] buffer = Encoding.UTF8.GetBytes(message);
        await ws.SendAsync(new ArraySegment<byte>(buffer),
WebSocketMessageType.Text, true, CancellationToken.None);

        var recvBuffer = new byte[1024];
        var result = await ws.ReceiveAsync(new ArraySegment<byte>(recvBuffer),
CancellationToken.None);
        Console.WriteLine("Echoed: " + Encoding.UTF8.GetString(recvBuffer, 0,
result.Count));
    }
}

```

WebSocket 서버는 직접 구축하거나 외부 공개 서버를 사용해야 하며, 위 예시는 echo 서버 사용 기준.

5. REST API 서버 (ASP.NET Core)

주요 파일 구조

```
SampleApi/  
├── Controllers/  
│   └── HelloController.cs  
└── Program.cs
```

HelloController.cs

```
using Microsoft.AspNetCore.Mvc;  
  
namespace SampleApi.Controllers  
{  
    [ApiController]  
    [Route("api/[controller]")]  
    public class HelloController : ControllerBase  
    {  
        [HttpGet]  
        public IActionResult Get() => Ok("Hello from REST API!");  
  
        [HttpGet("{name}")]  
        public IActionResult Greet(string name) => Ok($"Hello, {name}!");  
    }  
}
```

Program.cs (ASP.NET Core 6 이상)

```
var builder = webApplication.CreateBuilder(args);  
builder.Services.AddControllers();  
var app = builder.Build();  
  
app.UseHttpsRedirection();  
app.UseAuthorization();  
app.MapControllers();  
  
app.Run();
```

참고사항

- TCP는 신뢰성 보장, UDP는 속도 우선
- HTTP는 REST API와 외부 서버 연동에 적합
- WebSocket은 실시간 통신이 필요한 서비스 (챗, 알림 등)에 적합

3.04 - 스레드 / Task

1. Thread 클래스 사용 (기본적인 스레드 생성)

특징

- 가장 저수준 스레드 컨트롤.
- 명시적으로 스레드를 시작하고 종료 가능.

샘플

```
using System;
using System.Threading;

class Program
{
    static void PrintNumbers()
    {
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine($"[Thread] {i}");
            Thread.Sleep(500); // 0.5초 대기
        }
    }

    static void Main()
    {
        Thread thread = new Thread(PrintNumbers);
        thread.Start();

        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine($"[Main] {i}");
            Thread.Sleep(500);
        }

        thread.Join(); // 스레드가 끝날 때까지 대기
    }
}
```

2. ThreadPool 사용 (재사용 가능한 스레드)

특징

- 많은 수의 작업을 효율적으로 처리.
- 스레드 재사용으로 성능 최적화.

샘플

```

using System;
using System.Threading;

class Program
{
    static void work(object state)
    {
        Console.WriteLine($"[ThreadPool] 작업 시작 - Thread ID: {Thread.CurrentThread.ManagedThreadId}");
        Thread.Sleep(1000);
        Console.WriteLine($"[ThreadPool] 작업 완료");
    }

    static void Main()
    {
        ThreadPool.QueueUserWorkItem(work);
        Console.WriteLine("[Main] 메인 스레드 종료 대기");
        Thread.Sleep(1500); // 대기
    }
}

```

3. Task 클래스 사용 (비동기 작업 처리)

특징

- .NET 4.0 이상에서 권장되는 비동기 처리 방식.
- 병렬 작업, 연속 작업, 예외 처리 등에 유리.

샘플

```

using System;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Task task = Task.Run(() =>
        {
            Console.WriteLine("[Task] 작업 시작");
            Task.Delay(1000).wait(); // 비동기 대기
            Console.WriteLine("[Task] 작업 완료");
        });

        task.wait(); // 작업 완료 대기
        Console.WriteLine("[Main] 메인 종료");
    }
}

```

4. async / await 사용 (비동기 메서드)

특징

- 가독성 좋은 비동기 코드 작성 가능.
- UI 어플리케이션(WPF, WinForms)에서 UI 블로킹 없이 동작.

샘플

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task RunAsync()
    {
        Console.WriteLine("[Async] 작업 시작");
        await Task.Delay(1000); // 비동기 대기
        Console.WriteLine("[Async] 작업 완료");
    }

    static async Task Main()
    {
        await RunAsync();
        Console.WriteLine("[Main] 메인 종료");
    }
}
```

5. Parallel 클래스 사용 (병렬 처리)

특징

- 간단한 루프 병렬화에 적합.
- 내부적으로 Task 및 ThreadPool 활용.

샘플

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Parallel.For(0, 5, i =>
        {
            Console.WriteLine($"[Parallel] {i} - Thread ID:
{Thread.CurrentThread.ManagedThreadId}");
            Thread.Sleep(500);
        });

        Console.WriteLine("[Main] 메인 종료");
    }
}
```

비교 요약

방식	사용 용도	장점	단점
Thread	직접 제어가 필요한 경우	명확한 시작/종료 제어 가능	리소스 소모 큼
ThreadPool	짧고 빈번한 작업 처리	스레드 재사용 가능, 효율적	장시간 작업에 비적합
Task	병렬 작업, 연속 작업 등	예외처리, 연속 실행에 유리	설정이 복잡할 수 있음
async/await	I/O 비동기, UI 앱	직관적인 문법, UI 스레드 활용 가능	디버깅 어려움
Parallel	반복 작업 병렬 처리	루프 병렬화 간단	과도한 스레드 생성 주의

3.05 - 자주 사용하는 자료형

1. Array

사용 시점:

- 정해진 크기의 데이터를 다룰 때
- 빠른 접근 속도가 필요할 때
- 메모리 사용량이 중요하거나, 고성능 연산이 필요할 때

샘플

```
int[] numbers = new int[5] { 1, 2, 3, 4, 5 };
for (int i = 0; i < numbers.Length; i++)
{
    Console.WriteLine($"Index {i}: {numbers[i]}");
}
```

2. List<T>

사용 시점:

- 동적으로 크기가 변하는 컬렉션이 필요할 때
- 순서가 중요한 경우 (인덱스로 접근)
- 자주 추가/삭제가 발생하는 경우

샘플

```
List<string> fruits = new List<string>();
fruits.Add("Apple");
fruits.Add("Banana");
fruits.Add("Orange");

foreach (string fruit in fruits)
{
    Console.WriteLine(fruit);
}
```

3. Dictionary<TKey, TValue>

사용 시점:

- 키(key) 기반으로 값을 빠르게 찾고자 할 때
- 특정 키에 매핑된 값을 저장하고 검색할 때
- 중복 키를 허용하지 않아야 할 때

샘플

```
Dictionary<string, int> phoneBook = new Dictionary<string, int>();
phoneBook["Alice"] = 1234;
phoneBook["Bob"] = 5678;

if (phoneBook.ContainsKey("Alice"))
{
    Console.WriteLine($"Alice's number: {phoneBook["Alice"]}");
}
```

4. HashSet<T>

사용 시점:

- 중복을 허용하지 않는 유일한 데이터를 저장할 때
- 포함 여부(Contains)를 빠르게 확인할 때
- 교집합, 합집합, 차집합 같은 집합 연산이 필요할 때

샘플

```
HashSet<int> uniqueNumbers = new HashSet<int>();
uniqueNumbers.Add(1);
uniqueNumbers.Add(2);
uniqueNumbers.Add(2); // 중복은 무시됨
uniqueNumbers.Add(3);

foreach (int num in uniqueNumbers)
{
    Console.WriteLine(num);
}
```

요약

자료형	특징	중복 허용	순서 유지	인덱스 접근	키-값 구조
Array	고정 크기, 빠른 접근	O	O	O	X
List<T>	동적 크기, 순서 중요	O	O	O	X
Dictionary<K,V>	키 기반 저장 및 빠른 검색	키는 X	X	X	O
HashSet<T>	중복 제거, 빠른 검색	X	X	X	X

3.06 - LINQ 다루기

C#의 ****LINQ (Language Integrated Query)****는 데이터에 대해 **일관된 방식으로 쿼리할 수 있게 해주**는 문법

LINQ는 배열, 리스트, XML, DB, 컬렉션 등 다양한 데이터 소스에 대해 SQL과 유사한 방식으로 데이터를 질의할 수 있음.

LINQ 주요 특징 요약

항목	설명
통합 쿼리 문법	SQL처럼 데이터를 질의 (Where, Select 등)
강한 형식 지원	컴파일 타임에 타입 검사 가능
다양한 소스 지원	배열, List, Dictionary, XML, DB 등
지연 실행	실제로 데이터를 사용할 때 실행됨 (IEnumerable 기반)

LINQ 사용 방식

1. Query 문법 (쿼리식)
2. Method 문법 (메서드 체이닝)

사용 예제: 다양한 케이스

1. 리스트에서 조건에 맞는 항목 필터링 (Where)

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

// 짝수만 선택 (Query 문법)
var evenNumbersQuery = from n in numbers
                        where n % 2 == 0
                        select n;

// 메서드 문법
var evenNumbersMethod = numbers.Where(n => n % 2 == 0);

foreach (var num in evenNumbersMethod)
    Console.WriteLine(num); // 출력: 2, 4, 6
```

2. 특정 필드만 추출하기 (Select)

```
var users = new[]
{
    new { Id = 1, Name = "Alice" },
    new { Id = 2, Name = "Bob" }
};

var names = users.Select(u => u.Name);
foreach (var name in names)
    Console.WriteLine(name); // 출력: Alice, Bob
```

3. 정렬 (OrderBy, OrderByDescending)

```
var names = new List<string> { "Charlie", "Alice", "Bob" };

var sorted = names.OrderBy(n => n);
foreach (var n in sorted)
    Console.WriteLine(n); // 출력: Alice, Bob, Charlie
```

4. 그룹화 (GroupBy)

```
var scores = new[] {
    new { Name = "Alice", Subject = "Math", Score = 90 },
    new { Name = "Bob", Subject = "Math", Score = 80 },
    new { Name = "Alice", Subject = "English", Score = 85 }
};

var grouped = scores.GroupBy(s => s.Name);

foreach (var group in grouped)
{
    Console.WriteLine($"학생: {group.Key}");
    foreach (var item in group)
        Console.WriteLine($" - {item.Subject}: {item.Score}");
}
```

5. 집계 (Count, Sum, Average, Max, Min)

```
var nums = new[] { 1, 2, 3, 4, 5 };

Console.WriteLine(nums.Count()); // 5
Console.WriteLine(nums.Sum()); // 15
Console.WriteLine(nums.Average()); // 3
Console.WriteLine(nums.Max()); // 5
Console.WriteLine(nums.Min()); // 1
```

6. Any, All, Contains

```
var nums = new[] { 1, 2, 3, 4 };

// 조건을 만족하는 항목이 하나라도 있는가?
Console.WriteLine(nums.Any(n => n > 3)); // true

// 모든 항목이 조건을 만족하는가?
Console.WriteLine(nums.All(n => n > 0)); // true

// 특정 값이 포함되어 있는가?
Console.WriteLine(nums.Contains(3)); // true
```

7. 중복 제거 (Distinct)

```
var items = new[] { 1, 2, 2, 3, 3, 3 };

var unique = items.Distinct(); // 1, 2, 3
```

8. 리스트 합치기 (Concat, Union)

```
var a = new[] { 1, 2, 3 };
var b = new[] { 3, 4, 5 };

var result1 = a.Concat(b); // 중복 포함: 1, 2, 3, 3, 4, 5
var result2 = a.Union(b); // 중복 제거: 1, 2, 3, 4, 5
```

메서드 체이닝 방식과 쿼리식(Query Expression)

1. 메서드 체이닝 (Method Syntax)

- 람다 식과 메서드 호출을 연결(chain) 해서 작성합니다.
- `where()`, `Select()`, `OrderBy()` 등 LINQ 확장 메서드를 사용합니다.
- 가독성이 높은 단순 필터링/매핑에 적합합니다.

예제:

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

// 짝수만 선택하고 2배로 만든 후 정렬
var result = numbers
    .Where(n => n % 2 == 0)
    .Select(n => n * 2)
    .OrderBy(n => n);

foreach (var n in result)
    Console.WriteLine(n); // 출력: 4, 8, 12
```

2. 쿼리식 (Query Expression Syntax)

- SQL처럼 `from`, `where`, `select` 키워드를 사용합니다.
- 내부적으로는 메서드 체이닝으로 변환됩니다.
- 복잡한 조건, 그룹핑, 조인 등에 가독성이 뛰어납니다.

동일한 예제 (쿼리식으로 표현):

```
var numbers = new List<int> { 1, 2, 3, 4, 5, 6 };

var result = from n in numbers
              where n % 2 == 0
              orderby n
              select n * 2;

foreach (var n in result)
    Console.WriteLine(n); // 출력: 4, 8, 12
```

LINQ 사용 시 유의점

항목	설명
<code>IEnumerable</code> 는 지연 실행됨	쿼리를 선언만 하면 실행되지 않음
성능 고려 필요	복잡한 쿼리는 성능 저하 가능
가독성	쿼리가 길어지면 메서드 체이닝보다 쿼리식이 더 명확함
Null 체크 필수	<code>NullSource</code> 또는 <code>NullSelector</code> 조심

요약

- LINQ는 데이터 탐색, 필터링, 정렬, 집계 등 다양한 작업을 간결하게 처리
- 다양한 컬렉션과 구조에 쉽게 적용 가능
- Query식과 Method식 모두 활용 가능

3.07 - UTF / ANSI 변환 / Unicode 변환

C#에서 문자열을 **UTF-8**, **ANSI**, **Unicode(UTF-16)** 간에 변환하는 작업은 `Encoding` 클래스를 활용하면 간단하게 처리할 수 있음.

주요 인코딩 종류

인코딩	설명
UTF-8	가변 길이 문자 인코딩, ASCII와 호환됨
ANSI (Default Encoding)	Windows에서 사용하는 시스템 기본 인코딩 (보통 <code>Encoding.Default</code>)
Unicode (UTF-16)	C#의 기본 문자열 인코딩, <code>Encoding.Unicode</code> 사용

예제

시나리오: 문자열 "안녕하세요 Hello" 를 각각 UTF-8, ANSI, Unicode로 변환 후 다시 문자열로 복원.

1. UTF-8 <-> 문자열

```
using System;
using System.Text;

class Program
{
    static void Main()
    {
        string original = "안녕하세요 Hello";

        // 문자열 -> UTF-8 바이트 배열
        byte[] utf8Bytes = Encoding.UTF8.GetBytes(original);
        Console.WriteLine("UTF-8 인코딩 바이트 길이: " + utf8Bytes.Length);

        // UTF-8 바이트 배열 -> 문자열
        string fromUtf8 = Encoding.UTF8.GetString(utf8Bytes);
        Console.WriteLine("UTF-8 -> 문자열: " + fromUtf8);
    }
}
```

2. ANSI <-> 문자열

```
using System;
using System.Text;
```

```

class Program
{
    static void Main()
    {
        string original = "안녕하세요 Hello";

        // 문자열 -> ANSI 바이트 배열 (Encoding.Default)
        byte[] ansiBytes = Encoding.Default.GetBytes(original);
        Console.WriteLine("ANSI 인코딩 바이트 길이: " + ansiBytes.Length);

        // ANSI 바이트 배열 -> 문자열
        string fromAnsi = Encoding.Default.GetString(ansiBytes);
        Console.WriteLine("ANSI -> 문자열: " + fromAnsi);
    }
}

```

`Encoding.Default`는 시스템 설정에 따라 결과가 달라짐. 예를 들어 한글 Windows에서는 CP949로 처리됨. 비ASCII 문자는 손상될 수 있음.

3. Unicode (UTF-16) <-> 문자열

```

using System;
using System.Text;

class Program
{
    static void Main()
    {
        string original = "안녕하세요 Hello";

        // 문자열 -> Unicode (UTF-16) 바이트 배열
        byte[] unicodeBytes = Encoding.Unicode.GetBytes(original);
        Console.WriteLine("Unicode 인코딩 바이트 길이: " + unicodeBytes.Length);

        // 바이트 배열 -> 문자열
        string fromUnicode = Encoding.Unicode.GetString(unicodeBytes);
        Console.WriteLine("Unicode -> 문자열: " + fromUnicode);
    }
}

```

서로 다른 인코딩 간 변환 예제 (UTF-8 -> ANSI)

```

using System;
using System.Text;

class Program
{
    static void Main()
    {
        string original = "안녕하세요 Hello";

```

```
// 문자열을 UTF-8 바이트 배열로 변환
byte[] utf8Bytes = Encoding.UTF8.GetBytes(original);

// UTF-8 바이트를 문자열로 디코딩
string decoded = Encoding.UTF8.GetString(utf8Bytes);

// 다시 ANSI 바이트로 인코딩
byte[] ansiBytes = Encoding.Default.GetBytes(decoded);

// ANSI 바이트 -> 문자열
string ansiResult = Encoding.Default.GetString(ansiBytes);

Console.WriteLine("최종 변환된 문자열 (UTF8 -> ANSI): " + ansiResult);
}
}
```

UTF-8에 존재하는 모든 문자가 ANSI에 존재하는 것은 아니므로 변환 시 손실 가능성 있음.

유의사항

- UTF-8 ↔ Unicode 간 변환은 안전하며 손실 없음.
- ANSI ↔ Unicode/UTF-8 변환 시, 특정 문자는 표현 불가하여 ?로 변환될 수 있음.
- 파일 입출력 시 인코딩을 명시하지 않으면 `Encoding.UTF8` 이 기본값임.

파일로 인코딩 저장 예시

```
File.WriteAllText("utf8.txt", original, Encoding.UTF8);
File.WriteAllText("ansi.txt", original, Encoding.Default);
File.WriteAllText("unicode.txt", original, Encoding.Unicode);
```

3.08 - Json / Xml 다루기

JSON 처리 라이브러리

1. System.Text.Json (내장, .NET Core 3.0 이상)

주요 사용 사례

- .NET Core 기반 프로젝트에서 빠르고 가벼운 JSON 처리
- 웹 API (ASP.NET Core)에서 기본 직렬화 도구

사용 방법 및 샘플

```
// JsonSerializer 사용
using System.Text.Json;

public class User
{
    public string Name { get; set; }
    public int Age { get; set; }
}

var user = new User { Name = "Alice", Age = 25 };

// 직렬화
string json = JsonSerializer.Serialize(user);

// 역직렬화
User deserialized = JsonSerializer.Deserialize<User>(json);

Console.WriteLine(json); // {"Name":"Alice","Age":25}
```

주의 사항

- 기본적으로 **camelCase**를 따름
- 속성 이름 매핑이나 포매팅이 제한적 → 커스터마이징 시 `JsonSerializerOptions` 사용

```
// JsonDocument 사용
// JSON 문자열을 읽고 탐색(파싱)할 수 있도록 해주는 읽기 전용 DOM API
// JSON을 클래스에 매핑하지 않고 key-value 형태로 빠르게 접근하고 싶을 때 유용
using System;
using System.Text.Json;

class Program
{
    static void Main()
    {
        string json = @"{
            ""name"": ""홍길동"",
            ""age"": 30,
            ""isAdmin"": true,
            ""skills"": [""C#"", ""ASP.NET"", ""Azure""]
        }";
```

```

    }";

    using (JsonDocument doc = JsonDocument.Parse(json))
    {
        JsonElement root = doc.RootElement;

        // 기본 데이터 추출
        string name = root.GetProperty("name").GetString();
        int age = root.GetProperty("age").GetInt32();
        bool isAdmin = root.GetProperty("isAdmin").GetBoolean();

        Console.WriteLine($"이름: {name}");
        Console.WriteLine($"나이: {age}");
        Console.WriteLine($"관리자 여부: {isAdmin}");

        // 배열 데이터 추출
        JsonElement skills = root.GetProperty("skills");
        Console.WriteLine("기술 목록:");
        foreach (JsonElement skill in skills.EnumerateArray())
        {
            Console.WriteLine($"- {skill.GetString()}");
        }
    }
}

```

JsonDocument 사용 시 주의사항

주의사항	설명
using 블록	JsonDocument는 IDisposable을 구현하므로 반드시 using 블록이나 Dispose()로 해제 필요
속성 이름 확인	GetProperty("key") 사용 시 존재하지 않는 키에 접근하면 예외 발생하므로 TryGetProperty() 사용 고려 가능
읽기 전용	JsonDocument는 JSON 값을 수정할 수 없음. 수정하려면 JsonNode 또는 역직렬화 후 조작 필요

2. Newtonsoft.Json (Json.NET)

주요 사용 사례

- 다양한 JSON 포맷과 유연한 구조 처리
- `PropertyName` 커스터마이징, 깊은 중첩 구조, JWT 토큰 기반 동적 처리

사용 방법 및 샘플

```

using Newtonsoft.Json;

public class Product
{
    [JsonProperty("product_name")]
    public string Name { get; set; }
}

```

```

    public decimal Price { get; set; }
}

var product = new Product { Name = "Keyboard", Price = 49.99m };

// 직렬화
string json = JsonConvert.SerializeObject(product, Formatting.Indented);

// 역직렬화
Product deserialized = JsonConvert.DeserializeObject<Product>(json);

Console.WriteLine(json);

```

주의 사항

- .NET Core에서는 기본 포함되지 않음 → NuGet 설치 필요 (`Newtonsoft.Json`)
- 대규모 JSON 또는 성능 민감한 환경에서는 `System.Text.Json` 보다 느릴 수 있음

3. JObject, JToken을 통한 동적 JSON 처리 (Newtonsoft.Json)

```

using Newtonsoft.Json.Linq;

string json = @"{ 'name': 'Tom', 'skills': ['C#', 'SQL', 'Azure'] }";

JObject obj = JObject.Parse(json);
string name = obj["name"]?.ToString(); // "Tom"
var skills = obj["skills"]?.ToObject<List<string>>();

Console.WriteLine(string.Join(", ", skills)); // C#, SQL, Azure

```

XML 처리 라이브러리

1. System.Xml.Serialization.XmlSerializer

주요 사용 사례

- C# 클래스 ↔ XML 간 매핑
- 간단한 데이터 저장/불러오기

사용 방법 및 샘플

```

using System.Xml.Serialization;
using System.IO;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

var person = new Person { Name = "John", Age = 30 };

// 직렬화

```

```
var serializer = new XmlSerializer(typeof(Person));
using (var writer = new StreamWriter("person.xml"))
{
    serializer.Serialize(writer, person);
}

// 역직렬화
using (var reader = new StreamReader("person.xml"))
{
    var deserialized = (Person)serializer.Deserialize(reader);
    Console.WriteLine(deserialized.Name);
}
```

주의 사항

- 매핑되지 않은 속성은 무시됨 ([XmlIgnore] 가능)
- 생성자 없는 타입, Dictionary 등은 기본적으로 지원되지 않음

2. System.Xml.Linq (LINQ to XML)

주요 사용 사례

- XML 구조를 코드에서 직접 구성하거나 조작할 때
- 동적 XML 생성, 수정, 검색

사용 방법 및 샘플

```
using System.Xml.Linq;

var xml = new XElement("People",
    new XElement("Person",
        new XAttribute("Name", "Alice"),
        new XElement("Age", 28)
    ),
    new XElement("Person",
        new XAttribute("Name", "Bob"),
        new XElement("Age", 35)
    )
);

Console.WriteLine(xml);

foreach (var person in xml.Elements("Person"))
{
    Console.WriteLine($"{person.Attribute("Name")?.Value} - {person.Element("Age")?.Value}");
}
```

주의 사항

- 구조는 명확히 파악되지만, 클래스 ↔ XML 자동 변환이 아님

요약

구분	라이브러리	특징 / 장점	단점 및 주의 사항
JSON (기본)	<code>System.Text.Json</code>	빠르고 가볍고 기본 포함	복잡한 JSON 처리에 제약
JSON (고급)	<code>Newtonsoft.Json</code>	유연하고 커스터마이징 쉬움	성능 조금 떨어질 수 있음
JSON (동적)	<code>JsonToken</code> , <code>JsonObject</code>	비정형 JSON 처리에 유리	타입 안정성이 낮음
XML (정형화)	<code>XmlSerializer</code>	클래스 기반 직렬화/역직렬화	제한된 타입만 지원
XML (동적)	<code>System.Xml.Linq</code>	XML 조작에 매우 유용	클래스 ↔ XML 자동 매핑 아님

--

3.09 - DB 다루기

1. ADO.NET (가장 기본적인 방법)

설명:

- .NET Framework의 기본 DB 접근 방식.
- SQL 명령문을 직접 작성.
- 연결, 명령 실행, 리더 등 모든 작업을 수동으로 처리.

샘플 (SQL Server 기준):

```
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connStr = "Server=localhost;Database=TestDB;User
Id=sa;Password=your_password;";
        using (SqlConnection conn = new SqlConnection(connStr))
        {
            conn.Open();
            string sql = "SELECT Id, Name FROM Users";
            using (SqlCommand cmd = new SqlCommand(sql, conn))
            using (SqlDataReader reader = cmd.ExecuteReader())
            {
                while (reader.Read())
                {
                    Console.WriteLine($"{reader["Id"]}, {reader["Name"]}");
                }
            }
        }
    }
}
```

2. Dapper (Micro ORM)

설명:

- ADO.NET 기반이지만 훨씬 간결하고 성능이 좋음.
- SQL 문은 직접 작성하지만 매핑이 자동으로 처리됨.

NuGet 설치:

```
Dapper
```

샘플:

```
using System;
using System.Data.SqlClient;
using Dapper;
using System.Collections.Generic;

class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Program
{
    static void Main()
    {
        string connStr = "Server=localhost;Database=TestDB;User
Id=sa;Password=your_password;";
        using (var conn = new SqlConnection(connStr))
        {
            var users = conn.Query<User>("SELECT Id, Name FROM Users");
            foreach (var user in users)
            {
                Console.WriteLine($"{user.Id}, {user.Name}");
            }
        }
    }
}
```

3. Entity Framework Core (Full ORM)

설명:

- 객체지향 방식의 ORM.
- LINQ를 통한 쿼리 작성 가능.
- 관계 매핑, 변경 추적, 마이그레이션 등 지원.

NuGet 설치:

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
```

샘플:

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Linq;

class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```

}

class AppDbContext : DbContext
{
    public DbSet<User> Users { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlServer("Server=localhost;Database=TestDB;User
Id=sa;Password=your_password;");
}

class Program
{
    static void Main()
    {
        using (var context = new AppDbContext())
        {
            var users = context.Users.ToList();
            foreach (var user in users)
            {
                Console.WriteLine($"{user.Id}, {user.Name}");
            }
        }
    }
}

```

4. LINQ to SQL (레거시, 유지보수 어려움)

설명:

- LINQ 기반 ORM.
- Visual Studio에서 `.dbml` 파일을 이용해 매핑.
- 현재는 Entity Framework에 비해 사용이 줄었음.

5. 기타 방법들

방법	특징
NHibernate	복잡한 매핑 가능, 대형 프로젝트에 적합
ServiceStack ORM	경량 ORM, POCO 지원
EF Database First	DB 먼저 설계 후 모델 자동 생성
EF Code First	코드로 모델 정의 → DB 생성 가능

요약

방법	성능	코드량	유지보수	추천 용도
ADO.NET	★★★★☆	많음	낮음	단순한 DB 작업, 성능 우선
Dapper	★★★★★	적음	높음	빠르고 간단한 CRUD
EF Core	★★★☆☆	보통	높음	복잡한 도메인, 유지보수 중요
LINQ to SQL	★★☆☆☆	적음	낮음	기존 코드 유지 시

3.10 - 암호화 및 보안 처리

C#에서 암호화 및 보안 처리를 위해 자주 사용하는 라이브러리

- 해시 함수 (SHA256, SHA512 등)
- 대칭키 암호화 (AES)
- 비대칭키 암호화 (RSA)
- 기타 보안 관련 도구 (HMAC, PBKDF2 등)

1. SHA256 해시 함수

입력값을 고정된 길이의 해시값으로 변환 (복호화 불가능, 무결성 검증에 사용)

사용 예

```
using System;
using System.Security.Cryptography;
using System.Text;

public class HashExample
{
    public static string ComputeSHA256(string rawData)
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(rawData));
            StringBuilder builder = new StringBuilder();
            foreach (var b in bytes)
                builder.Append(b.ToString("x2")); // 16진수 문자열로 변환
            return builder.ToString();
        }
    }

    public static void Main()
    {
        string text = "password123";
        string hash = ComputeSHA256(text);
        Console.WriteLine($"SHA256 Hash: {hash}");
    }
}
```

2. AES 대칭키 암호화

동일한 키로 암호화 및 복호화. 속도가 빠르고 파일/텍스트 보호에 사용됨.

사용 예

```
using System;
using System.IO;
```

```

using System.Security.Cryptography;
using System.Text;

public class AesExample
{
    public static byte[] Encrypt(string plainText, byte[] Key, byte[] IV)
    {
        using (Aes aes = Aes.Create())
        {
            aes.Key = Key;
            aes.IV = IV;
            using var encryptor = aes.CreateEncryptor();
            using var ms = new MemoryStream();
            using (var cs = new CryptoStream(ms, encryptor,
CryptoStreamMode.Write))
                using (var sw = new StreamWriter(cs))
                {
                    sw.Write(plainText);
                }
            return ms.ToArray();
        }
    }

    public static string Decrypt(byte[] cipherText, byte[] Key, byte[] IV)
    {
        using (Aes aes = Aes.Create())
        {
            aes.Key = Key;
            aes.IV = IV;
            using var decryptor = aes.CreateDecryptor();
            using var ms = new MemoryStream(cipherText);
            using var cs = new CryptoStream(ms, decryptor,
CryptoStreamMode.Read);
            using var sr = new StreamReader(cs);
            return sr.ReadToEnd();
        }
    }

    public static void Main()
    {
        string original = "Secret Message";
        using (Aes aes = Aes.Create())
        {
            byte[] encrypted = Encrypt(original, aes.Key, aes.IV);
            string decrypted = Decrypt(encrypted, aes.Key, aes.IV);

            Console.WriteLine($"Encrypted (Base64):
{Convert.ToBase64String(encrypted)}");
            Console.WriteLine($"Decrypted: {decrypted}");
        }
    }
}

```

3. RSA 비대칭키 암호화

공개키로 암호화, 개인키로 복호화 (또는 그 반대). 주로 키 전달 및 디지털 서명에 사용.

사용 예

```
using System;
using System.Security.Cryptography;
using System.Text;

public class RsaExample
{
    public static void Main()
    {
        string plainText = "Top Secret Info";

        using (RSA rsa = RSA.Create(2048))
        {
            // 공개키 암호화
            byte[] encrypted = rsa.Encrypt(Encoding.UTF8.GetBytes(plainText),
            RSAEncryptionPadding.OaepSHA256);

            // 개인키 복호화
            byte[] decrypted = rsa.Decrypt(encrypted,
            RSAEncryptionPadding.OaepSHA256);

            Console.WriteLine($"Encrypted (Base64):
            {Convert.ToBase64String(encrypted)}");
            Console.WriteLine($"Decrypted Text:
            {Encoding.UTF8.GetString(decrypted)}");
        }
    }
}
```

4. HMAC (Hash-based Message Authentication Code)

메시지 무결성과 인증을 위한 해시. SHA256과 함께 사용됨.

사용 예

```
using System;
using System.Security.Cryptography;
using System.Text;

public class HmacExample
{
    public static string ComputeHMACSHA256(string message, string key)
    {
        using var hmac = new HMACSHA256(Encoding.UTF8.GetBytes(key));
        byte[] hash = hmac.ComputeHash(Encoding.UTF8.GetBytes(message));
        return Convert.ToBase64String(hash);
    }

    public static void Main()
    {
        string message = "message to protect";
        string key = "super_secret_key";
    }
}
```

```
string hash = ComputeHMACSHA256(message, key);
Console.WriteLine($"HMAC: {hash}");
}
}
```

5. PBKDF2 (비밀번호 기반 키 생성)

비밀번호로부터 안전한 암호화 키 생성 (해커가 무차별 대입 못하도록)

사용 예

```
using System;
using System.Security.Cryptography;

public class Pbkdf2Example
{
    public static void Main()
    {
        string password = "my_password";
        byte[] salt = RandomNumberGenerator.GetBytes(16);

        using var pbkdf2 = new Rfc2898DeriveBytes(password, salt, 100_000,
HashAlgorithmName.SHA256);
        byte[] key = pbkdf2.GetBytes(32); // AES 256비트 키

        Console.WriteLine($"Derived Key (Base64):
{Convert.ToBase64String(key)}");
    }
}
```

참고사항 및 유의점

항목	주의사항
SHA256	복호화 불가 (단방향), 비밀번호 저장 등에 적합
AES	키와 IV 보안 중요. 고정된 값 사용 지양
RSA	공개/개인 키 저장 방식 주의 (XML 또는 PEM)
HMAC	인증키 노출되면 무의미. 키 보안 필수
PBKDF2	충분한 반복 횟수(10만 이상) 설정 권장

3.11 - 다양한 실무 알고리즘

1. 날짜 계산 (월말, 분기, 휴일 여부)

```
// 이번 달 말일 구하기
DateTime today = DateTime.Today;
DateTime endOfMonth = new DateTime(today.Year, today.Month,
    DateTime.DaysInMonth(today.Year, today.Month));
Console.WriteLine($"이번 달 말일: {endOfMonth:yyyy-MM-dd}");

// 현재 날짜가 분기 몇 분기인지 확인
int quarter = (today.Month - 1) / 3 + 1;
Console.WriteLine($"{today:yyyy-MM-dd}는 {quarter}분기입니다.");

// 특정 날짜가 토/일/공휴일 여부 판별
bool IsHoliday(DateTime date, List<DateTime> holidays)
{
    return date.DayOfWeek == DayOfWeek.Saturday ||
        date.DayOfWeek == DayOfWeek.Sunday ||
        holidays.Contains(date.Date);
}

var holidays = new List<DateTime> { new DateTime(2025, 1, 1), new DateTime(2025,
    12, 25) };
Console.WriteLine(IsHoliday(today, holidays) ? "휴일입니다." : "근무일입니다.");
```

2. 문자열 파싱 (예: CSV, 고정폭)

```
// CSV 문자열 파싱
string csv = "홍길동,30,서울";
var parts = csv.Split(',');
Console.WriteLine($"이름: {parts[0]}, 나이: {parts[1]}, 주소: {parts[2]}");

// 고정폭 문자열 파싱 (예: 10자리 이름, 3자리 나이)
string fixedLine = "김철수 025서울특별시";
string name = fixedLine.Substring(0, 10).Trim();
string age = fixedLine.Substring(10, 3).Trim();
string city = fixedLine.Substring(13).Trim();
Console.WriteLine($"이름: {name}, 나이: {age}, 주소: {city}");
```

3. 중복 제거

```
var names = new List<string> { "홍길동", "김철수", "홍길동", "이영희" };
var uniqueNames = names.Distinct().ToList();
Console.WriteLine(string.Join(", ", uniqueNames));
```

4. 정렬

```
var employees = new List<(string Name, int Age)>
{
    ("홍길동", 30), ("김철수", 25), ("이영희", 35)
};

// 나이순 정렬
var sorted = employees.OrderBy(e => e.Age).ToList();
foreach (var emp in sorted)
    Console.WriteLine($"{emp.Name}: {emp.Age}세");
```

5. 그룹핑

```
var orders = new List<(string Customer, int Amount)>
{
    ("A사", 100), ("B사", 200), ("A사", 300), ("C사", 150)
};

var grouped = orders.GroupBy(o => o.Customer)
    .Select(g => new { Customer = g.Key, Total = g.Sum(o =>
o.Amount) });

foreach (var group in grouped)
    Console.WriteLine($"{group.Customer}: 총 금액 = {group.Total}원");
```

6. Dictionary

```
var productList = new List<(string Code, string Name)>
{
    ("P001", "노트북"), ("P002", "모니터"), ("P003", "마우스")
};

var productCache = productList.ToDictionary(p => p.Code, p => p.Name);

// 코드로 제품명 조회
string inputCode = "P002";
if (productCache.TryGetValue(inputCode, out string productName))
    Console.WriteLine($"코드 {inputCode}는 {productName}입니다.");
else
    Console.WriteLine("해당 제품이 없습니다.");
```

7. 조건부 필터링 + 계산 (예: 특정 조건만 합산)

```
var sales = new List<(string Region, int Amount)>
{
    ("서울", 1000), ("부산", 500), ("서울", 1500), ("대구", 700)
};

int seoulSales = sales.Where(s => s.Region == "서울").Sum(s => s.Amount);
Console.WriteLine($"서울 지역 매출 합계: {seoulSales}원");
```

8. 날짜 차이 계산 / 영업일 계산

```
DateTime start = new DateTime(2025, 7, 1);
DateTime end = new DateTime(2025, 7, 10);

int GetBusinessDays(DateTime from, DateTime to)
{
    int count = 0;
    for (var date = from; date <= to; date = date.AddDays(1))
    {
        if (date.DayOfWeek != DayOfWeek.Saturday && date.DayOfWeek !=
            DayOfWeek.Sunday)
            count++;
    }
    return count;
}

Console.WriteLine($"영업일 수: {GetBusinessDays(start, end)}일");
```

9. 다중 조건 검색

```
var users = new List<(string Name, int Age, string City)>
{
    ("홍길동", 30, "서울"), ("김철수", 40, "부산"), ("이영희", 35, "서울")
};

var result = users.Where(u => u.Age > 30 && u.City == "서울").ToList();

foreach (var u in result)
    Console.WriteLine($"{u.Name}, {u.Age}세, {u.City}");
```

10. 데이터 포맷 변환 (숫자 → 통화, 날짜 포맷)

```
decimal price = 1234567.89m;
Console.WriteLine(price.ToString("C")); // ₩1,234,567.89

DateTime dt = DateTime.Now;
Console.WriteLine(dt.ToString("yyyy-MM-dd HH:mm:ss")); // 2025-07-22 10:15:00
```

11. FluentValidation

설명: .NET 객체의 유효성 검사 라이브러리

실제 예제: 회원 가입 모델 검증

```
public class RegisterModel
{
    public string Email { get; set; }
    public string Password { get; set; }
}

public class RegisterValidator : AbstractValidator<RegisterModel>
{
    public RegisterValidator()
    {
        RuleFor(x => x.Email).NotEmpty().EmailAddress();
        RuleFor(x => x.Password).NotEmpty().MinimumLength(6);
    }
}

var model = new RegisterModel { Email = "", Password = "123" };
var result = new RegisterValidator().Validate(model);

if (!result.IsValid)
{
    foreach (var error in result.Errors)
        Console.WriteLine(error.ErrorMessage);
}
```

IN: 사용자 입력 모델

OUT: 유효성 결과 (IsValid, Errors)

유의점: ASP.NET Core에서는 자동 통합 설정 필요

3.12 - 기타 / 로깅

1. 유효성 검사 (Validation) 라이브러리

라이브러리	특징
FluentValidation	선언적 방식, 다양한 조건 설정, ASP.NET Core와 통합 용이
DataAnnotations (System.ComponentModel.DataAnnotations)	.NET 내장, 속성(Attribute) 기반 검증
ExpressiveAnnotations	수식 기반 검증 가능 ("Age > 18" 등), UI 모델에 적합

FluentValidation (대표)

- 특징: LINQ 스타일로 유효성 규칙을 정의할 수 있고, ASP.NET Core와도 쉽게 통합 가능
- 설치: NuGet 패키지 관리자를 통해 설치 (FluentValidation)

샘플 코드:

```
using FluentValidation;
using System;

public class User
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class UserValidator : AbstractValidator<User>
{
    public UserValidator()
    {
        RuleFor(user => user.Name)
            .NotEmpty().WithMessage("이름은 필수입니다.")
            .Length(2, 50).WithMessage("이름은 2~50자여야 합니다.");

        RuleFor(user => user.Age)
            .InclusiveBetween(18, 99).WithMessage("나이는 18~99세 사이여야 합니
다.");
    }
}

class Program
{
    static void Main()
    {
        var user = new User { Name = "", Age = 17 };
        var validator = new UserValidator();
```

```

var result = validator.Validate(user);

if (!result.IsValid)
{
    foreach (var failure in result.Errors)
    {
        Console.WriteLine($"속성: {failure.PropertyName}, 오류: {failure.ErrorMessage}");
    }
}
else
{
    Console.WriteLine("유효성 검사 통과");
}
}
}

```

2. 로깅 라이브러리

라이브러리	특징
Serilog	구조적 로깅, 콘솔/파일/DB/ElasticSearch/Seq 지원
NLog	XML 기반 설정, 성능 우수, 다양한 Sink 제공
log4net	오래된 레거시 프로젝트에 적합, 안정성 높음
Microsoft.Extensions.Logging	.NET Core 표준 로깅 추상화, 다양한 공급자와 연동 가능 (Serilog, NLog 등)

Serilog (대표)

- 특징: 콘솔, 파일, DB, Seq 등 다양한 대상에 로깅 가능
- 설치:

```

dotnet add package Serilog
dotnet add package Serilog.Sinks.Console
dotnet add package Serilog.Sinks.File

```

샘플 코드:

```

using Serilog;
using System;

class Program
{
    static void Main()
    {
        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Debug()
            .WriteTo.Console()
            .WriteTo.File("logs/log.txt", rollingInterval: RollingInterval.Day)
            .CreateLogger();
    }
}

```

```

        Log.Information("프로그램 시작");

        try
        {
            int result = 10 / int.Parse("0"); // 예외 발생
        }
        catch (Exception ex)
        {
            Log.Error(ex, "예외 발생!");
        }

        Log.Information("프로그램 종료");
        Log.CloseAndFlush();
    }
}

```

DI / IoC (의존성 주입)

라이브러리	특징
Microsoft.Extensions.DependencyInjection	.NET Core 내장 DI 컨테이너
Autofac	복잡한 DI 시나리오, 모듈 기반 설정 지원
SimpleInjector	경량 DI 컨테이너, 성능 우수

매핑 (DTO ↔ Entity 등)

라이브러리	특징
AutoMapper	객체 간 자동 매핑, 설정으로 변환 로직 제거
Mapster	빠른 성능, 선언적 방식도 가능, AutoMapper 대체로 인기 상승 중

AutoMapper 예제

```

using AutoMapper;

Order order = new()
{
    Id = 120,
    Name = "범범조조",
    DeliverAddress = "대한민국 경기도 어퍼구 저퍼구"
};

// AutoMapper 정보 설정
var config = new MapperConfiguration(cfg => cfg.CreateMap<Order, OrderDto>());

// Mapping
var mapper = new Mapper(config);
OrderDto dto = mapper.Map<OrderDto>(order);

// OrderDto 객체 출력

```

```
Console.WriteLine($"Id : {dto.Id}, Name : {dto.Name}, DeliverAddress :  
{dto.DeliverAddress}");
```

```
public class Order  
{  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public string DeliverAddress { get; set; }  
}
```

```
public class OrderDto  
{  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public string DeliverAddress { get; set; }  
}
```

4.01 - KICE 시험 유형 및 대비 방법

개요

- 해당 시스템 구현을 통해 요구사항 분석, 데이터 구조화, HTTP Server/Client 구현 등의 기술역량 및 프로그램 구현 역량을 측정하기 위한 문제.
- Pilot 기준 3문제 출제 되었으며 선행 문제를 해결하여야 후속문제 처리가 가능한 구조
- 인터넷 검색 사용 가능 함으로 인터넷을 통한 샘플코드 검색 가능하나 AI검색은 활용 불가.(시험시간 총 3시간)
- 본시험은 유형이 변경될 수 있음.

알아야 할 기본 사항 (Pilot시험 기준)

1. 입출력

- Console 기반 입출력 : Console.ReadLine / Console.WriteLine
- File 입력 : File.Exists / File.ReadAllLines / StreamReader (본고사에 출력이 있을수 있음)

2. 자료구조

- Dictionary / HashSet / Class기반 structure
- String 처리 (Split, 대소문자 변환)
- Linq를 통한 Array / List handling 숙달 필요

3. 꼭 알아야 하는 것들

- HttpClient (Rest client)
- HttpListener (Rest server)
- Http Method 및 Content-type 관련
- Json handling (Serialize / Deserialize / JsonDocument / Newtonsoft.Json)
- XML handling (XmlSerializer) - 본고사에 나올 가능성

Pilot 시험에서는 3문제에 대한 기본 프로젝트가 제공 되었는데 모두 Console 어플리케이션이었기 때문에 ASP.NET을 사용하기 어려울 수 있음.

3. 시험 대비 방법

기본적인 자료구조 핸들링 연습 필요

- 인터넷 검색이 가능하지만 시간 사용의 효율을 위해 기본적인 자료구조 핸들링 방법은 숙지 필요 (String, File, Linq, Json 등)
- 익숙하지 않을 경우 별도로 샘플 코드 작성 후 개인 블로그나 GitHub 작성 후 시험시 참조

인터넷 검색에 대한 장점을 활용

- 기본적인 HttpClient / HttpListener 등을 사용한 Method를 작성 한 후 개인 블로그나 GitHub등에 기재하고 시험시 찾아 볼 수 있다.

- C# 관련하여 사용법이나 알고리즘등이 자세히 수록된 기술 블로그 등을 미리 숙지 하고 바로 찾아 볼 수 있도록 한다.

시험 문제 끝까지 자세히 읽어보고 이해한 후에 답안 작성 필요

- 각 문항만 읽고 각 문항의 답안을 작성 하게 되면 출제자의 요구사항을 놓칠 수 있음.
- 시험 문제 끝에 각 문항의 실행 방법 및 결과를 샘플로 제시하고 있음.
- 각 문제는 연결되는 구조이므로 마지막 문제를 해결하기 위해 어떻게 모듈화 하는것이 좋을지 판단해서 작성하는 것이 좋음.

4. Tips

- C# .NET환경에서 nullable 에 대한 warning은 시간관계상 무시해도 무방.
- VDI환경에서 해상도로 인해 글자가 너무 작다고 느껴지는 경우 VDI 환경이 아닌 로컬 환경의 해상도를 조정하면 VDI도 같이 적용됨.
- 인터넷은 로컬이 아닌 VDI에서 검색 할 수 있음.
- 난이도는 어렵지 않으나 출제자의 요구사항을 명확히 반영하지 않으면 0점 처리 됨. 문제 끝까지 확인 필수.

KICE
시스템&솔루션개발
실기형 문제지

[2025년 파일럿 차수]

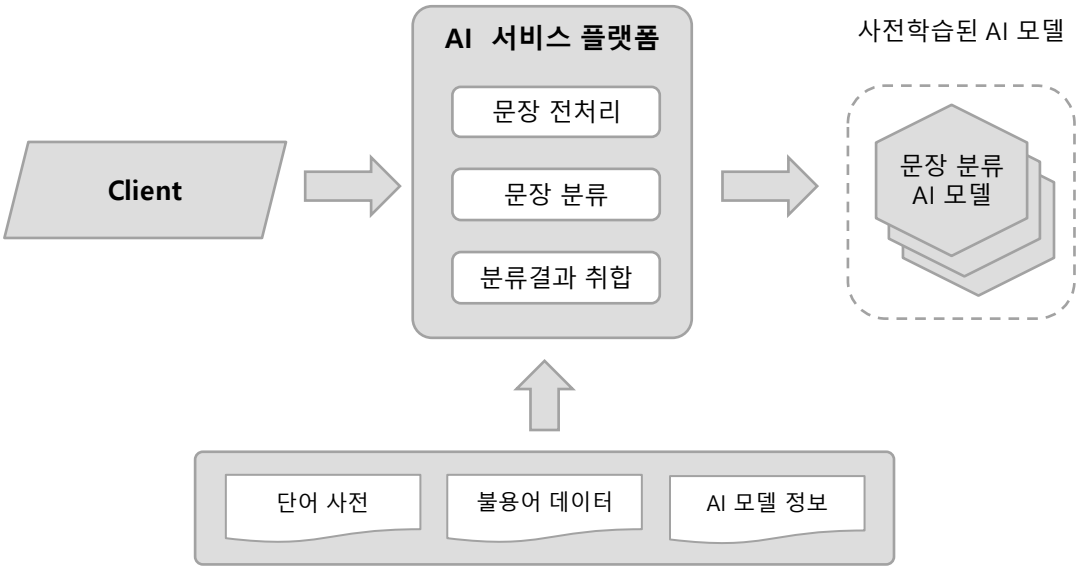
AI 서비스 플랫폼

개요

해당 시스템 구현을 통해 요구사항 분석, 데이터 구조화, HTTP Server/Client 구현 등의 기술역량 및 프로그램 구현 역량을 측정하기 위한 문제입니다.

설명

본 프로그램은 사전학습된 AI 모델을 활용하여 요청 받은 Query 문장들에 대한 분류를 수행하고 결과를 제공하는 AI 서비스 플랫폼입니다.



[기능 요약]

- 클라이언트가 AI 모델 추론 요청하면 먼저 각각의 Query 문장들에 대한 '문장 전처리'를 수행한다.
- 사전학습된 AI 모델로 '문장 분류'를 요청한다. ('문장 전처리' 결과를 송신하고, '분류코드'를 수신한다)
- AI 모델로부터 수신한 '분류코드'를 '분류결과'로 변환하여 취합한 후 클라이언트에 응답한다.

주의사항

실행 결과로 평가하고 부분점수는 없으므로 아래사항을 필히 주의해야 함

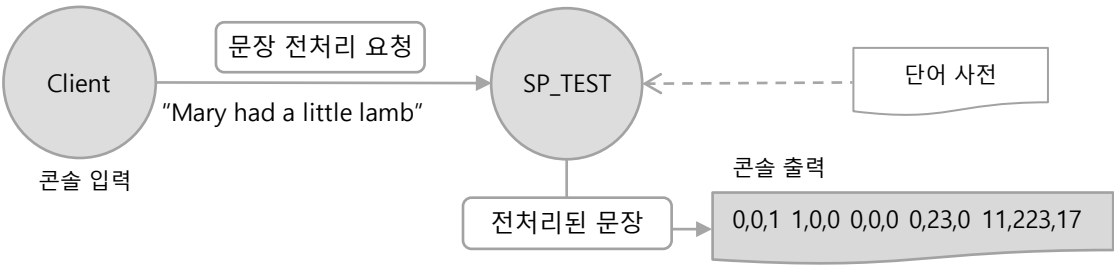
- 구현된 프로그램은 실행 완결성 필수 (명확한 실행&정확한 결과 출력, 통상의 실행 시간)
- 소 문항별 결과 검수 필수 (선행문항 오류 시, 후속문항 전체에 오류가 발생할 수 있음)
- 제시된 조건이 없는 한 선행요구사항 유지 필수
- 프로그램 실행 위치 및 실행결과출력 (위치, 파일명, 데이터포맷)은 요구사항과 정확히 일치 필수 (콘솔 출력이 평가 대상일 경우 불필요한 로그 출력 금지)
- 제시된 모든 위치는 상대경로 사용 필수 (프로그램 실행 위치 기준)
- 프로그램 종료조건에 맞는 처리 필수 (불필요한 입력대기를 하거나, 요구사항과 다르게 종료하면 안됨)
- 제공되는 샘플 파일과 다른 데이터로 채점하므로 제공되는 파일의 내용을 하드코딩하지 말 것
- 모든 문자는 요구사항에 맞는 대소문자 구분 필수

문제

아래 제시된 문항은 문항번호가 증가할수록 점진적 개선을 요구하는 방식으로 구성되어 있으며, 제시된 문항번호 별로 각각 **구현된 소스와 컴파일 된 실행파일을 제출**하시오.
cf) 1번 구현 → 1번 소스복사 → 2번 구현 → 2번 소스복사 → ...

- 1. 콘솔 입력을 통해서 "문장"을 입력하면 "단어"로 토큰화하고 "단어 사전"을 참조하여 워드임베딩된 결과(문장 전체리 결과)를 출력하시오 (20점)

상세설명

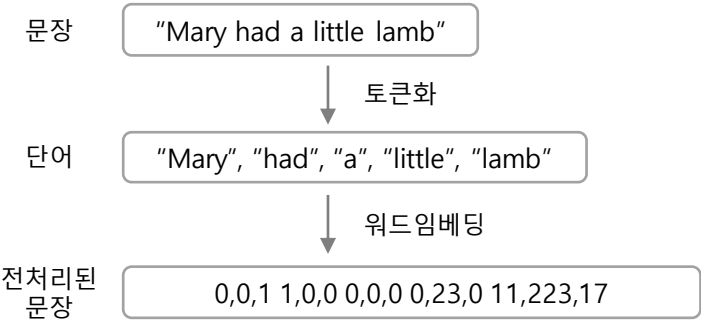


- ※ 토큰화
 - "문장"을 공백문자로 구분된 "단어"로 쪼개는 작업
 - ex) "Mary had a little lamb"은 "Mary", "had", "a", "little", "lamb"의 5 단어로 쪼개짐

- ※ 워드임베딩
 - 각각의 <단어>를 "단어 사전"을 참조하여 <Embedding Vector>로 변환
 - ex) 토큰화된 "Mary", "had", "a", "little", "lamb"의 5 단어를 각각 "단어 사전"을 참조하여 <Embedding Vector>로 변환 (단어 사전은 알파벳 소문자로만 구성되므로 소문자 변환 후 사전에서 찾을 것)

단어	소문자 변환	Embedding Vector
Mary	mary	0,0,1
had	had	1,0,0
a	a	0,0,0
little	little	0,23,0
lamb	lamb	11,223,17

- ※ 전체리된 문장
 - "문장"을 토큰화 한 후에 워드임베딩하여 <Embedding Vector>로 변환된 결과
 - 예시) "Mary had a little lamb" 전체리 결과



형식정보

- ※ 입력 문장 형식정보
- 형식 : <단어> + ' '(공백 문자) + ... + ' '(공백 문자) + <단어>

- 입력 문장 내 단어의 알파벳은 대소문자로 구성됨.
- ※ 단어 사전 형식정보

- 파일명 : DICTIONARY.TXT (각 소문항 홈 아래)

- 파일 형식 : <단어> + '#' + <Embedding Vector>

a#0,0,0

an#0,0,0

the#0,0,0

...

good#7,112,9816

...

- <단어> : 단어 사전 내 단어의 알파벳은 소문자로 구성됨

- <Embedding Vector> : 정수 3개로 구성되며 ',' (comma)로 구분됨

※ 콘솔 입/출력

- 입력 포맷 : <query 문장>

- 출력 포맷 : <Embedding Vector> + ' '(공백 문자) ... + ' '(공백 문자) + <Embedding Vector>

Mary had a little lamb

0,0,1 1,0,0 0,0,0 0,23,0 11,223,17

<프로그램 종료하지 않고 계속 입력 및 결과 출력>

← 콘솔 출력

← 콘솔 입력

평가대상

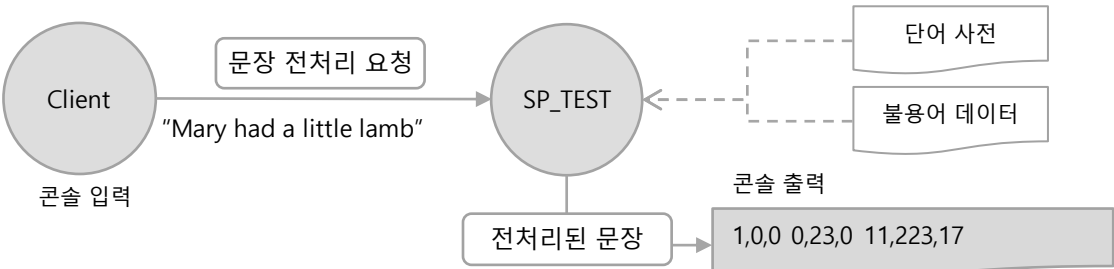
프로그램 정상 실행, 콘솔 입/출력 결과, 프로그램 종료 없음
자가 검수를 위해 제공되는 샘플은 채점용 데이터와 다름

3

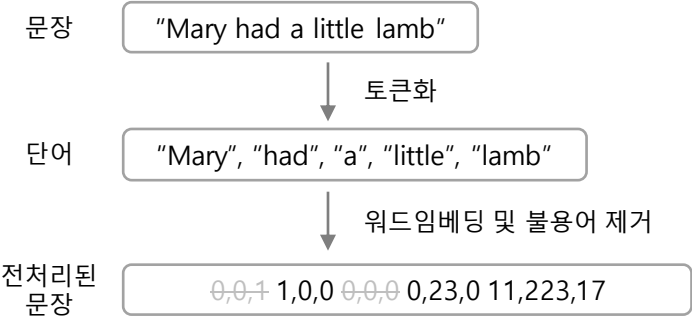
문제

2. 위 1번 문항까지 구현된 내용을 기준으로, 아래 사항을 추가로 반영하여 구현하시오. (30점)
- 불용어(분석에 사용하지 않는 단어) 데이터 추가
 - 불용어를 제거한 후 워드임베딩된 결과(문장 전처리 결과)를 출력

상세설명

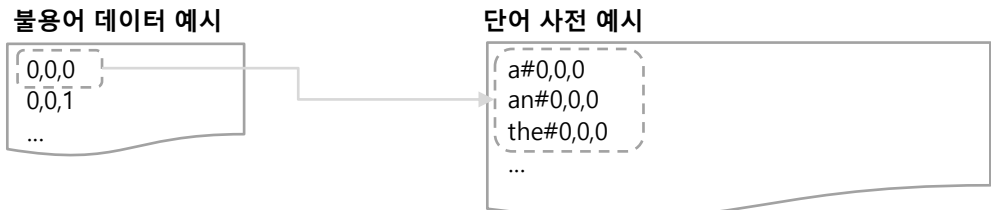


- ※ 불용어 제거
- 토큰화된 "단어" 가운데 분석에 사용하지 않는 단어를 제거하는 전처리 작업
 - ex) 예를 들어, 불용어 데이터에 "0,0,0"이 포함되어 있다면, <Embedding Vector>가 "0,0,0"인 모든 "단어" (예시에서는 'a', 'an', 'the'가 제외됨)
- ※ 전처리된 문장
- "문장"을 토큰화 한 후에 불용어를 제외하고 워드임베딩하여 <Embedding Vector>로 변환된 결과
 - 예시) "0,0,0", "0,0,1"이 불용어인 경우, "Mary had a little lamb" 전처리 결과



형식정보

- ※ '불용어 데이터' 파일 형식정보
- 파일명 : STOPWORD.TXT (각 소문항 홈 아래)
 - 파일 형식 : <Embedding Vector>



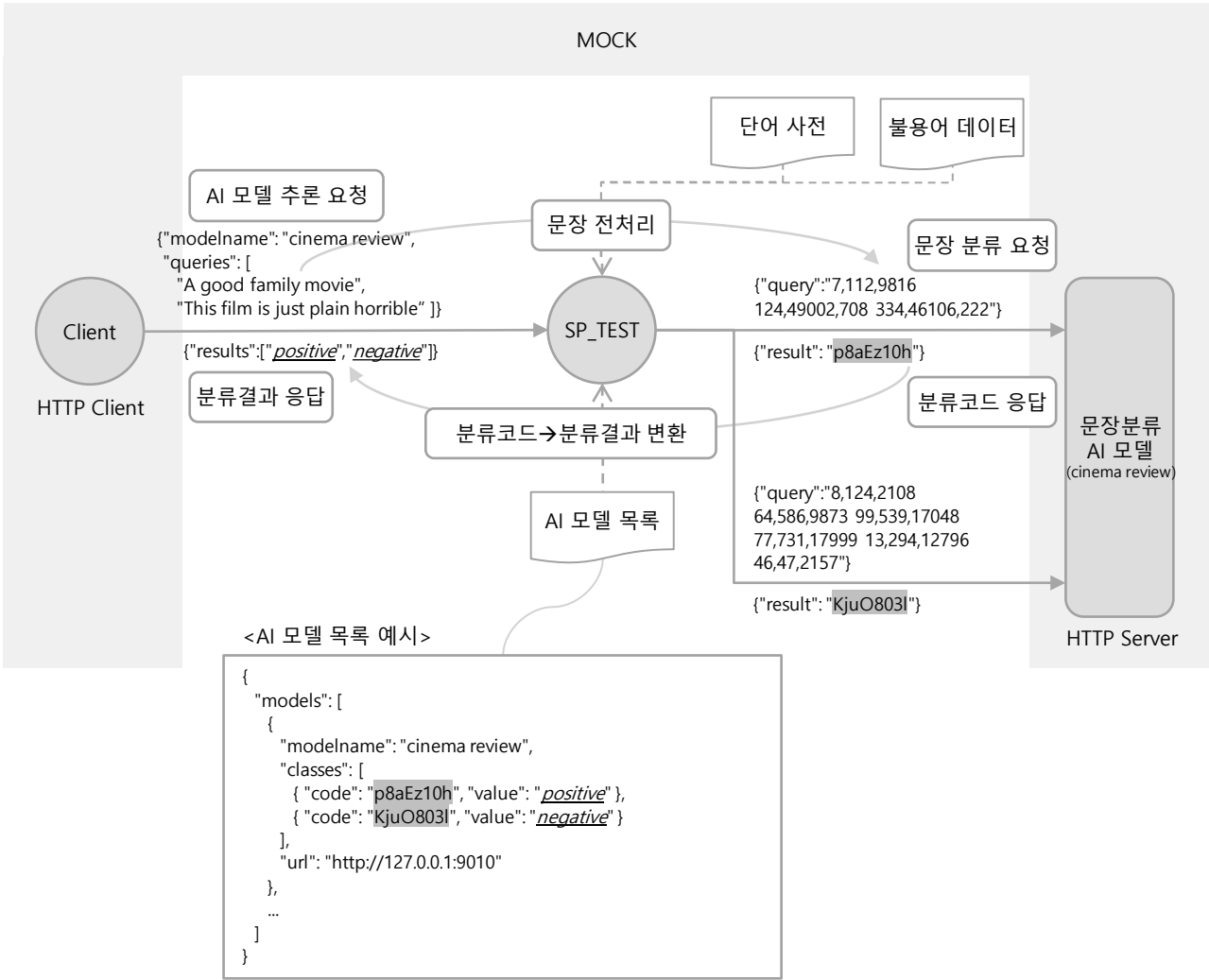
평가대상

프로그램 정상 실행, 콘솔 입/출력 결과, 프로그램 종료 없음
자가 검수를 위해 제공되는 샘플은 채점용 데이터와 다름

문제

3. 위 2번 문항까지 구현된 내용을 기준으로, 아래 사항을 추가로 반영하여 AI 서비스 플랫폼을 구현하시오. (50점)
- 사전학습된 모델의 상세정보가 '문장 분류 AI 모델 목록'으로 제공됨
 - HTTP 통신으로 AI 모델 추론 요청을 수신하면 사전학습된 문장 분류 AI 모델을 활용하여 분류 결과를 응답

상세설명



- ※ 문장 분류 AI 모델
- 사전학습되어 전처리(토큰화, 워드임베딩, 불용어제거)된 문장을 입력하면 **분류코드**를 응답
- ※ AI 모델 추론 요청
- 사전학습된 문장 분류 AI 모델 중 하나를 선택하여, **여러 개의 Query 문장**에 대하여 추론 요청
 - 여러 개의 Query 문장에 대한 분류 결과를 배열로 **순서대로** 응답
 - AI 모델은 <분류코드>를 응답하므로 <분류결과>로 변환 필요

형식정보

- ※ "AI 모델 목록" 형식정보
- 파일 경로 : MODELS.JSON
 - 아래의 Json 형식

```
{
  "models": [ {
    "modelName": <AI 모델명>,
    "classes": [
      { "code":<분류코드>, "value":<분류결과> },
      ...
    ],
    "url":<AI 모델 Url>,
  }, ...
]
```

- <AI 모델명> : 문장 분류에 사용할 사전학습된 AI 모델명
- <분류코드> : 문장 분류 AI 모델이 응답하는 분류 코드
- <분류결과> : 분류 코드에 해당하는 실제 분류 결과
- <AI 모델 Url> : 문장 분류를 수행하기 위해 호출할 Url

- ※ "AI 모델 추론 요청" 형식정보
- 아래의 Json 형식

```
{
  "modelName": <AI 모델명>
  "queries": [<Query 문장>,...,<Query 문장>],
}
```

- <AI 모델명> : 문장 분류에 사용할 사전학습된 AI 모델명
- <Query 문장> : 분류할 문장

AI 모델 추론 요청 예시

```
{
  "modelName": "cinema review",
  "queries": ["A good family movie", "This film is just plain horrible"]
}
```

- ※ "문장 분류 요청" 형식정보
- 아래의 Json 형식

```
{
  "query": <전처리된 문장>
}
```

- <전처리된 문장> : 문장 전처리 과정을 통해 '토큰화' 및 '워드임베딩'된 문장

문장 분류 요청 예시

```
{
  "query": "7,112,9816 124,49002,708 334,46106,222"
}
```

형식정보
(계속)

- ※ AI 모델 추론 요청
- URI : POST http://127.0.0.1:8080/
 - 요청 Body : Json 문자열 형식 (※ "AI 모델 추론 요청 " 형식정보 참조)
 - 동작 : 'AI 모델 목록' 중에 지정된 문장 분류 AI 모델을 활용하여 요청된 문장들을 **하나씩 순서대로** 분류한 후 결과 목록을 응답
 - 응답 Status Code : 200 OK
 - 응답 Body : Json 문자열 형식
{ "results": [<분류결과>, ... , <분류결과>] }
ex) { "results": ["positive", "negative"] }
- ※ 문장 분류 요청
- URI : POST <AI 모델 Url>
 - 요청 Body : Json 문자열 형식 (※ "문장 분류 요청 " 형식정보 참조)
 - 동작 : 문장 분류를 수행하여 분류코드를 응답
 - 응답 Status Code : 200 OK
 - 응답 Body : Json 문자열 형식
{ "result": <분류코드> }
ex) { "result": "p8aEz10h" }
- ※ 제공프로그램(MOCK.EXE)
- MOCK.EXE를 콘솔에서 실행하면 테스트 시나리오에 따라 요구사항을 순차적으로 테스트함
 - MOCK.EXE는 자가 검수의 기능도 수행하며, 모든 테스트 시나리오 성공 시 다음의 문구가 콘솔에 출력

C:\W>MOCK.EXE<엔터키>
...
테스트에 성공했습니다!

← 제공프로그램 실행
← 테스트 시나리오에 따른 테스트 실행
← 소문항의 모든 테스트시나리오 성공

평가대상

프로그램 정상 실행, HTTP 요청, HTTP 응답 결과, **프로그램 종료 없음**
자가 검수를 위해 제공되는 샘플과 MOCK은 채점용 데이터와 다름

폴더 정보

- ※ 프로그램 및 파일 위치 정보 (실행위치 기반 상대경로 사용 필수)
 - 구현할 프로그램 위치 및 실행 위치 : 각 소문항 홈 (SUB1/SUB2/SUB3)
 - 자가 검수용 참고 파일명 : COMPARE 폴더 내 CMP_CONSOLE.TXT (SUB1/SUB2)
 - 제공되는 MOCK 프로그램 파일명 : 각 소문항 홈 아래 MOCK.EXE (SUB3)
- * 제공되는 파일들은 문항에 따라 다를 수 있음
- * 자가 검수를 위해 제공되는 샘플은 채점용 데이터와 다름

실행 방식

- ※ 구현할 프로그램 형식
 - 프로그램 형태 : 콘솔(Console) 프로그램
 - 프로그램 파일명 : SP_TEST
 - 실행 방식(문항1~2) : 콘솔 실행 → 콘솔 입/출력처리 (종료 없음)
- C:\W>SP_TEST<엔터키> ← 구현한 프로그램 실행 (Argument 없음)

...

- 실행방식(문항3) : 콘솔 실행 → 실시간 HTTP 요청 수신 (종료 없음)
- C:\W>SP_TEST<엔터키> ← 구현한 프로그램 실행 (Argument 없음)

...

제공 및 제출

- ✓ 각 언어별 제공파일 압축 해제 후 자동 생성된 폴더 사용 필수
- ✓ 제공되는 주요 내용
 - 샘플 파일
 - 제공 프로그램 실행파일 (MOCK.EXE)
 - 제출시 사용할 문항별 폴더 구조
- ✓ 제출 파일 및 폴더 상세 내용 (각 언어별 실기 가이드 참고)

테스트 방법 ※ 자가 검수를 위해 제공되는 샘플은 채점용 데이터와 다름

검수를 위한 샘플 결과 파일은 각 문항 출력 폴더(COMPARE)에 사전 제공됨

[문항1]

- SP_TEST를 실행한 후 콘솔 입/출력 결과를 샘플 결과 파일(CMP_CONSOLE.TXT)와 동일한지 비교

```
C:\W>SP_TEST<엔터키>          ← 구현한 프로그램 실행 (Argument 없음)
A good family movie              ← 콘솔 입력
0,0,0 7,112,9816 124,49002,708 334,46106,222      ← 콘솔 출력
It was great to see some of my favorite ...        ← 콘솔 입력
3,750,1951 81,220,6524 91,720,6265 4,708,4711 ...  ← 콘솔 출력
```

[문항2]

- SP_TEST를 실행한 후 콘솔 입/출력 결과를 샘플 결과 파일(CMP_CONSOLE.TXT)와 동일한지 비교

```
C:\W>SP_TEST<엔터키>          ← 구현한 프로그램 실행 (Argument 없음)
A good family movie              ← 콘솔 입력
7,112,9816 124,49002,708 334,46106,222            ← 콘솔 출력
It was great to see some of my favorite ...        ← 콘솔 입력
3,750,1951 81,220,6524 91,720,6265 92,164,7646 ... ← 콘솔 출력
```

[문항3]

- SP_TEST를 실행한 후, MOCK.EXE를 실행
 - MOCK.EXE의 콘솔에 출력되는 테스트 결과 확인
- (SP_TEST가 문항의 요건을 만족하지 못하는 경우, MOCK.EXE의 콘솔에 테스트 Fail의 사유 또는 에러 메시지가 출력되므로 자가 검수에 참고 요망)

```
C:\W>MOCK.EXE<엔터키>
테스트를 시작합니다.
...
테스트에 성공했습니다!
```

수고하셨습니다.