# C#
# -CAHPTER9-

SOUL SEEK

# 목차

# 파일다루기

# 1. 파일다루기

## 파일 정보와 디렉토리 정보 다루기

- **.NET Framework**에서는 파일과 디렉토리 정보를 손쉽게 다룰 수 있도록 **System.IO** 네임스페이스 클래스들을 제공한다.

| 클래스 | 설명 |
|---|---|
| **File** | 파일의 생성, 복사, 삭제, 이동, 조회를 처리하는 정적 메소드를 제공한다. |
| **FileInfo** | **File** 클래스와 하는 일은 동일하지만 정적 메소드 대신 인스턴스 메소드를 제공한다. |
| **Directory** | 디렉토리의 생성, 삭제, 이동, 조회를 처리하는 정적 메소드를 제공한다. |
| **DirectoryInfo** | **Directory**클래스와 하는 일은 동일하지만 정적 메소드 대신 인스턴스 메소드를 제공한다. |

- 각 클래스가 제공하는 주요 메소드와 프로퍼티 – **()**있으면 메소드, 없으면 프로퍼티

| 기능 | File | FileInfo | Directory | DirectoryInfo |
|---|---|---|---|---|
| 생성 | **Create()** | **Create()** | **CreateDirectory()** | **Create()** |
| 복사 | **Copy()** | **CopyTo()** | - | - |
| 삭제 | **Delete()** | **Delete()** | **Delete()** | **Delete()** |
| 이동 | **Move()** | **Move()** | **Move()** | **Move()** |
| 존재 여부 확인 | **Exists()** | **Exists()** | **Exists()** | **Exists()** |

# 1. 파일다루기

| 기능 | File | FileInfo | Directory | DirectoryInfo |
|---|---|---|---|---|
| 속성 조회 | **GetAttribute()** | **Attributes** | **GetAttribute()** | **Attributes** |
| 하위 디렉토리 조회 | - | - | **GetDirectories()** | **GetDirectories()** |
| 하위 파일 조회 | - | - | **GetFiles()** | **GetFiles()** |

- **File**클래스와 **FileInfo** 클래스는 하는 일은 같으니 사용하는 스타일만 비교해보면 된다.

| 기능 | File | FileInfo |
|---|---|---|
| 생성 | **FileStream fs = File.Create(“a.dat”);** | **FileInfo file = new FileInfo(“a.dat”);**<br>**FileStream fs = file.Create();** |
| 복사 | **File.Copy(“a.dat”, “b.dat”);** | **FileInfo src = new FileInfo(“a.dat”);**<br>**FileInfo dst = src.CopyTo(“b.dat”);** |
| 삭제 | **File.Delete(“a.dat”);** | **FileInfo file = new FileInfo(“a.dat”);**<br>**File.Delete(“b.dat”);** |
| 이동 | **File.Move(“a.dat”, “b.dat”);** | **FileInfo file = new FileInfo(“a.dat”);**<br>**File.MoveTo(“b.dat”);** |
| 존재 여부 확인 | **if(File.Exists(“a.dat”)) //…** | **FileInfo file = new FileInfo(“a.dat”);**<br>**If(file.Exists) //…** |
| 속성 조회 | **Console.WriteLine (File.GetAttributes(“a.dat”));** | **FileInfo file = new FileInfo(“a.dat”);**<br>**Console.WriteLine(file.Attributes);** |

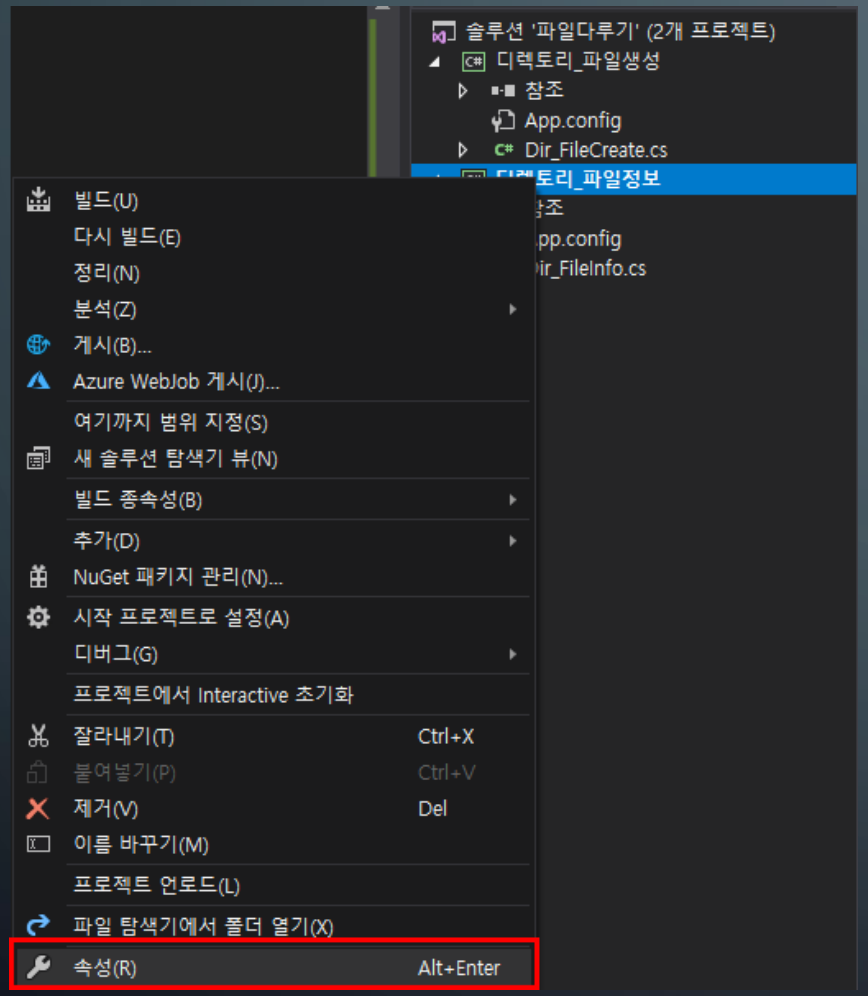
# 1. 파일다루기

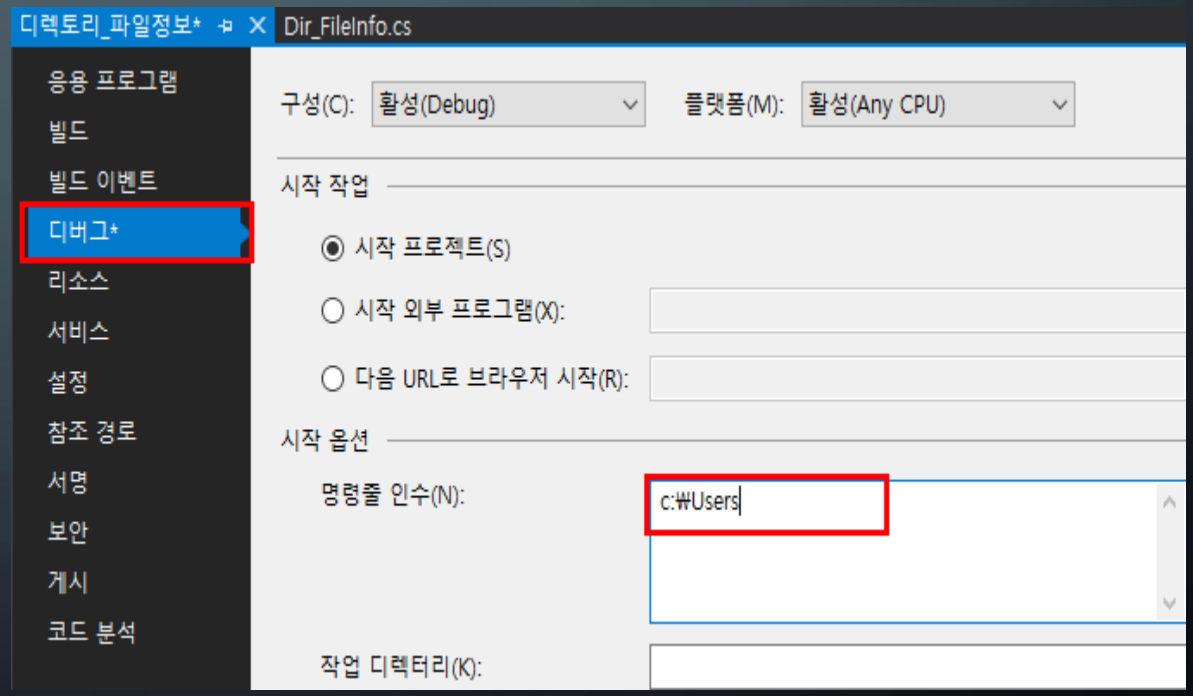
- **Directory**클래스와 **DirectoryInfo**클래스 사용 방법을 보겠다.

| 기능 | Directory | DirectoryInfo |
|---|---|---|
| 생성 | DirectoryInfo dir = Directory.CreateDirectory(“a”); | DirectoryInfo dir = new DirectoryInfo(“a”); dir.Create(); |
| 삭제 | Directory.Delete(“a”); | DirectoryInfo dir = new DirectoryInfo(“a”); dir.Delete(); |
| 이동 | Directory.Move(“a”, “b”); | DirectoryInfo dir = new DirectoryInfo(“a”); Dir.MoveTo(“b”) |
| 존재 여부 확인 | If(Directory.Exists(“a.dat”)) //.. | DirectoryInfo dir = new DirectoryInfo(“a”); If(dir.Exists) //.. |
| 속성 조회 | Console.WriteLine(Directory.GetAttribute(“a”)); | DirectoryInfo dir = new DirectoryInfo(“a”); Console.WirteLine(dir.Attributes); |
| 하위 디렉토리 조회 | string[] dirs = Directory.GetDirectories(“a”); | DirectoryInfo dir = new DirectoryInfo(“a”); DirectoryInfo[] dirs. = dir.GetDirectories(); |
| 하위 파일 조회 | String[] files = Directory.GetFiles(“a”); | DirectoryInfo dir = new DirectoryInfo(“a”); FileInfo[] files = dir.GetFiles(); |

# 1. 파일다루기

**디렉토리_파일정보, 디렉토리_파일생성** 두 개의 프로젝트를 이용해 사용법을 익혀 보자. 두 프로젝트는 사용자 매개 변수를 입력하여 실행하는 부분이 있으므로 이에 대해 한번 알아보자.



# 1. 파일다루기

## 파일을 읽고 쓰기를 위한 정보

### Stream

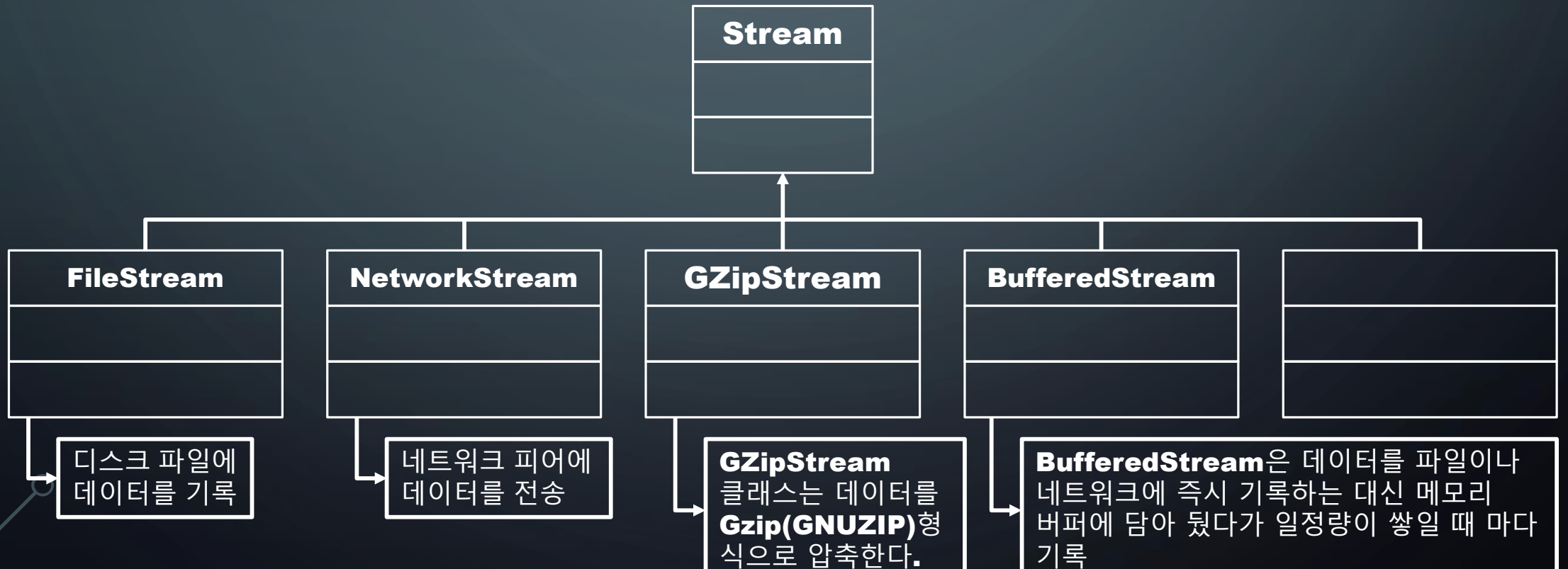
- 데이터가 흐르는 통로
- 메모리에 있는 내용을 하드디스크의 파일에 쓰거나 반대로 읽을 경우에 하드디스크와 메모리 사이에 **Stream**을 만들어서 둘 사이를 연결한 뒤에 바이트 단위로 옮기게 된다.
- **Stream**은 데이터 흐름이기 때문에 처음 부터 끝까지 순서대로 읽고 쓰는 순차접근**(Sequential Access)** 방식을 취한다.
- 네트워크나 데이터 백업 장치의 입/출력 구조와 동일하다.

# 1. 파일다루기

## System.IO.Stream 클래스

- **C#**에서 사용하는 입출력 스트림**,** 순차 접근방식과 임의 접근 방식을 모두 지원한다**.**
- **Stream** 클래스는 추상 클래스이기 때문에 이 클래스의 인스턴스를 직접 만들어 사용할 수는 없고 이 클래스로 부터 파생된 클래스를 이용해야 한다**.**
  ➔ 스트림을 다루는 다양한 매체나 장치들에 대한 파일 입출력을 스트림 모델 하나로 다룰 수 있도록 하기 위함이다**.**

**Stream** 클래스와 이를 상속하는 다양한 파생 클래스들의 계보를 나타낸다**.**



# 1. 파일다루기

## FileStream

- 인스턴스 생성

```
Stream stream1 = new FileStream(“a.dat”, FileMode.Create);              // 새 파일 생성
Stream stream2 = new FileStream(“b.dat”, FileMode.Open);                // 파일 열기
Stream stream3 = new FileStream(“c.dat”, FileMode.OpenOrCreate);  // 파일을 열거나
                                                                                        // 파일이 없으면 생성
Stream stream4 = new FileStream(“d.dat”, FileMode.Truncate);     // 파일을 비워서 열기
Stream stream5 = new FileStream(“e.dat”, FileMode.Append);       // 덧붙이기 모드로 열기
```

- **Data** 쓰기

```
public override void Write(
                                byte[] array,       // 쓸 데이터가 담겨 있는 byte 배열
                                int offset,         // byte 배열 내의 시작 오프셋
                                int count);         // 기록할 데이터의 총 길이(단위는 바이트)

public override void WriteByte(byte value);
```

안타깝게도 **Byte**형식으로 **Data**를 전환하지 않으면 기록하거나 읽어서 쓸 수가 없다.

# 1. 파일다루기

- **BitConverter**의 도움으로 데이터를 **Byte**로 전환, 그 후 파일에 기록.

```
long someValue = 0x123456789ABCDEF0;

// 파일 스트림 생성
Stream outstream = new FileStream("a.dat", FileMode.Create);

// someValue(long 형식)을 byte 배열로 변환
byte[] wBytes = BitConverter.GetBytes(someValue);

// 변환한 byte 배열을 파일 스트림을 통해 파일에 기록
outstream.Write(wBytes, 0, wBytes.Length);

// 파일 스트림에 닫기
outstream.Close();
```

- **Data** 읽기

```
public override int Read(
                                byte[] array,      // 읽은 데이터를 담을 byte 배열
                                int offset,        // byte 배열 내의 시작 오프셋
                                int count);        // 읽을 데이터의 최대 바이트 수

public override int ReadyByte();
```

# 1. 파일다루기

- **Byte**형식의 **FileData**를 읽어서 변환 하기**.**

**Byte[] rBytes = new byte[8];**

// 파일 스트림 생성
**Stream instream = new FileStream(“a.dat”, FileMode.Open);**

// **rBytes**의 길이만큼**(8**바이트**)** 데이터를 읽어 **rBytes**에 저장
**instream.Read(rBytes, 0, rBytes.Length);**

// **BitConverter**를 이용하여 **rBytes**에 담겨 있는 값을 **long** 형식으로 변환
**Long readValue = BitConverter.ToInt64(rbytes, 0);**

// 파일 스트림 닫기
**inStream.Close();**

**Long**을 입출력 예문을 자세히 살펴보면 결과가 조금 이상하게 나타날 것이다**.**
➔순서가 뒤집혀져서 입력된다**.**

- **CLR**이 설치되어 있는 컴퓨터 아키텍처가 지원하는 바이트 오더**(Byte Order)**가 데이터의 낮은 주소부터 기록하는 리틀 엔디안 방식이기 때문에 나타나는 현상이다**.**

- **x86(32**비트**)** 계열의 **CPU**들은 리틀 엔디안 방식으로 동작하지만 **Power CPU**나 **Sparc** 계열의 **CPU**는 빅 엔디안 방식으로 동작한다**.** 그렇기 때문에 각 스트림 방식의 바이트 오더**(Byte Order)**에 대한 개념이 필요하다**,** 네트워크 프로그래밍에서도 **Stream**이 필요하기 때문에 **Byte Order**가 필요**.**

# 1. 파일다루기

**Stream** 클래스의 프로퍼티인 **Position**이나 **Seek()** 메소드를 호출하면 지정한 위치 부터 읽거나 쓰기가 가능하다.

```
Stream outstream = new FileStream("a.dat", FileMode.Create);
// ...
outStream.Seek(5, SeekOrigin.Current);
outstream.WriteByte(0x04);
```

현재 위치에서 5바이트 뒤로 이동

코드

파일

new FileStream()

WriteByte(0x01)

WriteByte(0x02)

WriteByte(0x03)

Seek(5, SeekOrigin.Current)

WriteByte(0x04)

# 1. 파일다루기

## 2진 데이터 처리를 위한 BinaryWrite / BinaryReader

**FileStream** 클래스의 사용상의 불편함을 지원하기 위한 클래스
이진 데이터**(Binary Data)**를 스트림에 기록 또는 읽어 오기 위한 클래스
도우미의 역할을 하기 때문에 **Stream**으로 부터 파생된 클래스의 인스턴스가 필요하다**.**

- **BinaryWrite**와 **FileStream**을 같이 사용하는 예**.**

```
BinaryWriter bw = new BinaryWriter(new FileStream("a.dat", FileMode.Create));

bw.Write(32);
bw.Write("Good Morning!");
bw.Write(3.14);

bw.Close();
```

**Write()** 메소드는 **C#**이 제공하는 모든 기본 데이터 형식에 대해 오버로딩 되어있다**.**

- **BinaryReader**와 **FileStream**을 사용하는 예**.**

```
BinaryReader br = new BinaryReader(new FileStream("a.dat", FileMode.Open));

Int      a = br.ReadInt32();
String   b = br.ReadString();
double   c = br.ReadDouble();

br.Close();
```

**BinaryReader**는 읽을 데이터의 형식별로 **ReadXXX()** 메소드를 제공한다**. – XXX**는 읽을 데이터의 원본 이름을 말한다**.**

# 1. 파일다루기

## 텍스트 파일 처리를 위한 **StreamWrite / StreamReader**

**StreamWriter** 사용하기.

```
StreamWriter sw = new StreamWriter(new FileStream("a.dat", FileMode.Create));

sw.Write(32);
sw.WriteLine("Good Morning!");
sw.WriteLine(3.14);

sw.Close();

StreamReader sr = new StreamReader(new FileStream("a.dat", FileMode.Open));

While(sr.EndOfStream == false)
{
      Console.WriteLine(sr.ReadLine());
}

sr.Close();
```

**Write()**와 **WriteLine()** 메소드는 **C#**이 제공하는 모든 기본 데이터 형식에 대해 오버로딩 되어 있다.

**EndOfStream** 프로퍼티는 스트림의 끝에 도달했는지를 알려준다.

# 1. 파일다루기

## 객체 직렬화하기(Serialization)

- **BinaryWriter/Reader** 와 **StreamWriter / Reader**은 기본 데이터 형식을 스트림에 쓰고 읽을 수 있도록 메소드를 제공하지만, 프로그래머가 직접 정의한 클래스나 구조체 같은 복합 데이터 형식은 지원하지 않는다.
- 복합데이터 형식은 형식이 가지고 있는 필드의 값을 저장할 순서를 정한 후, 이 순서대로 저장하고 읽는 코드를 작성해야 한다.
- **C#**은 직렬화**(Serialization)**라는 메커니즘을 지원해줘서 복합데이터 형식을 쉽게 스트림에 쓰기/읽기가 가능하게 해준다.
- 직렬화 **:** 객채의 상태(여기서는 객체 필드에 저장된 값들을 의미)를 메모리나 영구 저장 장치에 저장이 가능한 **0**과 **1**의 순서로 바꾸는 것을 말한다.
- **C#**은 이진 형식, **JSON(JavaScript Object Notation), XML** 같은 텍스트 형식으로의 직렬화를 지원한다.

**직렬화 형식**

```
[Serializable]
class MyClass
{
        // ..
}
```

# 1. 파일다루기

**Stream** 클래스와 **BinaryFormatter**를 이용해서 간단히 저장 할 수 있다.

```
Stream ws = new FileStream(“a.dat”, FileMode.Create);
BinaryFormatter serializer = new BinaryFormatter();

MyClass obj = new MyClass();
// obj의 필드에 값 저장

serializer.Serialize(ws, obj);  ➔ 직렬화
ws.Close();
```

**BinaryFormatter**

- **System.Runtime.Serialization.Formatter.Binary** 네임스페이스에 소속되어 있다.
- 직렬화하거나 역직렬화 하는 역할을 수행한다.

역직렬화의 예

```
Stream rs = new FileStream(“a.dat”, FileMode.Open);
BinaryFormatter deserializer = new BinaryFormatter();

MyClass obj = (MyClass)deserialzer.Deserialize(rs);
rs.Close();
```

# 1. 파일다루기

**NonSerialized** 애트리뷰트

직렬화 수행 중에 직렬화하고 싶지 않은 필드가 있을 때 사용한다.

```
[Serializable]
class MyClass
{
        public int myField1;
        public int myField2;

        [NonSerialized]
        pubic int myField3; ➔ myField3 필드를 제외한 나머지 필드들만 직렬화 된다.
}
```

복합 데이터 형식을 직렬화할 때 주의할 점.
**Serializable** 애트리뷰트를 이용해서 복합 데이터를 직렬화하고자 할 때, 직렬화 하지 '않는' 필드뿐 아니라 직렬화하지 '못하는' 필드도 **Nonserializable** 애트리뷰트로 수식해야 한다.
➔ 직렬화할 복합 데이터 형식에 선언된 필드중에 복합 데이터 형식이 존재 한다면 직렬화 할 수 없다.
➔ 복합 데이터 형식을 직렬화 하던지, 해당 필드를 직렬화에서 제외하던지 해야 한다.

```
class NonserializableClass
{
        public int myField;
}
```

```
[Serializable]
Class MyClass
{
        public int myField1;

        public NonserializableClass myField2;
}
```

# 1. 파일다루기