14. 자바 입출력 (I/O)

`InputStream`, `OutputStream`

🟦 개요: InputStream & OutputStream

`java.io` 패키지의 두 핵심 추상 클래스:

클래스	역할
`InputStream`	바이트 단위 **입력 스트림** (읽기)
`OutputStream`	바이트 단위 **출력 스트림** (쓰기)

모두 **바이트 기반 (byte streams)** 을 다룸.
→ 즉, **텍스트가 아니라 바이너리**를 처리하는 게 기본!

🔷 1. InputStream - 데이터를 읽는 스트림

```
InputStream in = new FileInputStream("file.txt");

int data = in.read(); // 1바이트 읽기
```

📌 주요 메서드

메서드	설명
`int read()`	1바이트 읽어 정수 반환 (0~255), 끝이면 -1
`int read(byte[] b)`	여러 바이트를 배열로 읽음
`int read(byte[], int off, int len)`	일부만 읽기
`void close()`	스트림 닫기 (리소스 반납)

🔁 예제: 파일에서 바이트 읽기

```
try (InputStream in = new FileInputStream("data.bin")) {
    int byteRead;
    while ((byteRead = in.read()) != -1) {
        System.out.print((char) byteRead);  // 텍스트일 경우
    }
}
```

🔶 2. OutputStream - 데이터를 쓰는 스트림

```
OutputStream out = new FileOutputStream("output.bin");

out.write(65); // 'A' (ASCII 65) 쓰기
```

📌 주요 메서드

메서드	설명
`void write(int b)`	1바이트 쓰기
`void write(byte[] b)`	배열로 한 번에 쓰기
`void write(byte[], int off, int len)`	일부 쓰기
`void flush()`	버퍼된 데이터를 즉시 출력
`void close()`	스트림 닫기

🔄 예제: 파일에 바이트 쓰기

```
try (OutputStream out = new FileOutputStream("output.bin")) {
    out.write("Hello".getBytes());  // 문자열을 바이트로 변환해 저장
}
```

🧱 3. 상속 구조 (요약)

```
InputStream
 ├─ FileInputStream
 ├─ BufferedInputStream
 ├─ ByteArrayInputStream
 └─ FilterInputStream (추상, 데코레이터)

OutputStream
 ├─ FileOutputStream
 ├─ BufferedOutputStream
 ├─ ByteArrayOutputStream
 └─ FilterOutputStream (추상, 데코레이터)
```

🛠️ 4. InputStream → 문자 변환

`InputStream`은 **바이트 단위**이므로 텍스트를 읽으려면 문자 변환이 필요하다:

```java
InputStream in = new FileInputStream("text.txt");
InputStreamReader reader = new InputStreamReader(in, "UTF-8");
BufferedReader br = new BufferedReader(reader);

String line = br.readLine();
```

🔐 5. 예외 처리 및 자원 닫기

Java 7 이후부터는 try-with-resources로 자동 닫기 가능:

```java
try (InputStream in = new FileInputStream("file.txt")) {
    // 읽기 작업
} catch (IOException e) {
    // 예외 처리
}
```

📌 요약 비교

항목	InputStream	OutputStream
목적	데이터 읽기 (입력)	데이터 쓰기 (출력)
데이터 단위	바이트 (byte)	바이트 (byte)
주요 구현체	FileInputStream, ByteArrayInputStream	FileOutputStream, ByteArrayOutputStream
주의사항	텍스트가 아닌 **바이너리 처리 기본**	flush(), close() 필수

📘 실전 예제

```java
// 바이너리 파일 복사기
try (InputStream in = new FileInputStream("input.jpg");
     OutputStream out = new FileOutputStream("copy.jpg")) {

    byte[] buffer = new byte[1024];
    int len;
    while ((len = in.read(buffer)) != -1) {
        out.write(buffer, 0, len);
    }
}
```

`Reader`, `Writer`

🟦 1. 개요: `Reader` / `Writer`

클래스	역할	처리 단위
`Reader`	문자 입력 (읽기) 스트림	**문자 (char)**
`Writer`	문자 출력 (쓰기) 스트림	**문자 (char)**

> UTF-8, UTF-16 같은 문자 인코딩 처리를 자동으로 해주기 때문에
> **텍스트 데이터를 다룰 때는 반드시 `Reader`/`Writer` 계열을 써야 함!**

🟩 2. Reader – 문자 기반 입력

```
Reader reader = new FileReader("input.txt");
int ch = reader.read();  // 한 문자 읽기
```

📌 주요 메서드

메서드	설명
`int read()`	한 문자(char)를 읽고 정수 반환, EOF 시 -1
`int read(char[] cbuf)`	문자 배열로 여러 글자 읽기
`int read(char[], int off, int len)`	배열의 일부만 읽기
`void close()`	스트림 닫기

✅ 예제

```
try (Reader reader = new FileReader("input.txt")) {
    int ch;
    while ((ch = reader.read()) != -1) {
        System.out.print((char) ch);
    }
}
```

🟥 3. Writer – 문자 기반 출력

```
Writer writer = new FileWriter("output.txt");
writer.write("Hello, world!");
```

📌 주요 메서드

메서드	설명
`void write(int c)`	한 문자 쓰기
`void write(char[] cbuf)`	문자 배열 쓰기
`void write(String str)`	문자열 쓰기
`void flush()`	버퍼된 데이터 출력
`void close()`	스트림 닫기

✅ 예제

```
try (Writer writer = new FileWriter("output.txt")) {
    writer.write("안녕하세요!\n");
    writer.write("문자 기반 출력입니다.");
}
```

🧱 4. 상속 구조

```
Reader
 ├─ FileReader
 ├─ BufferedReader
 ├─ InputStreamReader
 └─ CharArrayReader, StringReader 등

Writer
 ├─ FileWriter
 ├─ BufferedWriter
 ├─ OutputStreamWriter
 └─ CharArrayWriter, StringWriter 등
```

🧩 5. InputStreamReader / OutputStreamWriter

이건 **바이트 → 문자** 변환을 처리하는 **브리지 클래스**이다.

```
InputStream in = new FileInputStream("input.txt");
Reader reader = new InputStreamReader(in, StandardCharsets.UTF_8);
```

```
OutputStream out = new FileOutputStream("output.txt");
Writer writer = new OutputStreamWriter(out, "UTF-8");
```

💡 6. BufferedReader / BufferedWriter

속도 향상을 위한 **버퍼링된 문자 스트림**

```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
String line = br.readLine();  // 한 줄 읽기

BufferedWriter bw = new BufferedWriter(new FileWriter("file.txt"));
bw.write("한 줄 쓰기");
bw.newLine();  // 개행 문자 삽입
```

📘 Reader vs Writer vs Stream 비교표

항목	InputStream / OutputStream	Reader / Writer
처리 대상	바이트 (binary)	문자 (text)
텍스트 인코딩 처리	수동 필요 (직접 변환)	자동 처리
문자열 출력	byte[] → getBytes() 필요	String 바로 사용
클래스 예시	FileInputStream, FileOutputStream	FileReader, FileWriter

---

🔄 실전 예제: 텍스트 파일 복사

```
try (Reader reader = new FileReader("source.txt");
     Writer writer = new FileWriter("target.txt")) {
    char[] buffer = new char[1024];
    int len;
    while ((len = reader.read(buffer)) != -1) {
        writer.write(buffer, 0, len);
    }
}
```

✅ 요약

개념	핵심 요점
Reader	문자 단위 입력 (텍스트 읽기)
Writer	문자 단위 출력 (텍스트 쓰기)
FileReader / FileWriter	파일을 직접 열어 텍스트 입출력
BufferedReader / BufferedWriter	성능 향상용 버퍼 스트림
InputStreamReader / OutputStreamWriter	바이트 <-> 문자 변환 브리지

`BufferedReader`, `BufferedWriter`

🟦 1. BufferedReader & BufferedWriter 개요

클래스	설명
`BufferedReader`	버퍼를 사용하여 **문자 입력 성능 향상**
`BufferedWriter`	버퍼를 사용하여 **문자 출력 성능 향상**

- 내부적으로 **버퍼 배열(char[])** 을 사용해서 시스템 호출 횟수를 줄임
- 줄 단위 텍스트 읽기, 쓰기에 매우 유용함
- 디스크 I/O를 줄여주기 때문에 **성능 개선에 필수**

🟩 2. BufferedReader 사용법

생성자

```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
```

주요 메서드

메서드	설명
`String readLine()`	한 줄 읽기 (줄바꿈 문자는 포함하지 않음)
`int read()`	문자 하나 읽기
`int read(char[] cbuf)`	문자 배열로 읽기
`void close()`	스트림 닫기

예제

```
try (BufferedReader br = new BufferedReader(new FileReader("input.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        System.out.println("읽은 줄: " + line);
    }
}
```

🟥 3. BufferedWriter 사용법

생성자

```
BufferedWriter bw = new BufferedWriter(new FileWriter("file.txt"));
```

주요 메서드

메서드	설명
`void write(String s)`	문자열 쓰기
`void newLine()`	줄바꿈 문자 삽입 (`\r\n` 또는 `\n`)
`void flush()`	버퍼에 있는 내용을 즉시 출력
`void close()`	스트림 닫기 (자동으로 flush 포함)

예제

```
try (BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {
    bw.write("첫 번째 줄");
    bw.newLine();
    bw.write("두 번째 줄");
    bw.flush(); // 명시적 flush
}
```

📘 내부 버퍼 작동 방식

```
BufferedReader br = new BufferedReader(new FileReader("large.txt"));
```

- 내부 버퍼 크기는 기본적으로 **8192자** (`char[8192]`)
- 예를 들어 `readLine()`이 호출되면, 내부 버퍼에서 줄바꿈까지 미리 읽고 반환함
- `read()` 메서드를 많이 호출하면 FileReader의 성능보다 훨씬 좋아짐

🧩 4. 함께 쓰이는 구조

파일 복사 예시 (문자 기반)

```
try (BufferedReader br = new BufferedReader(new FileReader("input.txt"));
     BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"))) {

    String line;
    while ((line = br.readLine()) != null) {
        bw.write(line);
        bw.newLine();
    }
```

```
}
```

🔐 5. try-with-resources 활용

두 객체를 동시에 try 블록에 선언해서 자동으로 닫을 수 있음.

```
try (
    BufferedReader br = new BufferedReader(new FileReader("input.txt"));
    BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"));
) {
    // ...
}
```

💡 언제 사용하나?

상황	추천 스트림
줄 단위로 텍스트 파일 읽기	`BufferedReader.readLine()`
여러 줄 텍스트 출력	`BufferedWriter.write()` + `newLine()`
성능 중요 / 대용량 파일	반드시 `Buffered*` 계열 사용

✅ 요약

항목	BufferedReader	BufferedWriter
목적	텍스트 입력 성능 향상 (줄 읽기 가능)	텍스트 출력 성능 향상 (줄 쓰기 가능)
주요 메서드	`readLine()`, `read()`	`write()`, `newLine()`, `flush()`
내부 버퍼 크기	기본 8192자	기본 8192자
추천 사용 시기	파일 줄 단위 읽기, 사용자 입력 읽기	텍스트 파일 줄 단위 출력

파일 읽기/쓰기 (`File`, `FileReader`, `Files`)

🟦 1. `File` 클래스 – 파일/디렉토리 경로 관리용

> `java.io.File` 클래스는 **파일 자체를 다루는 게 아니라**, 경로(path) 정보를 다루는 **"파일 시스템 추상화 객체"**이다.

📌 주요 기능

메서드	설명
`exists()`	파일/디렉토리 존재 여부

메서드	설명
`isFile()`, `isDirectory()`	파일인지, 디렉토리인지 확인
`getName()`, `getPath()`	이름, 경로 정보
`length()`	파일 크기
`delete()`, `renameTo()`	삭제, 이름 변경
`mkdir()`, `mkdirs()`	디렉토리 생성
`listFiles()`	하위 파일 목록 반환

✅ 예제

```
File file = new File("test.txt");

if (file.exists()) {
    System.out.println("파일 존재: " + file.getAbsolutePath());
    System.out.println("크기: " + file.length() + " bytes");
} else {
    System.out.println("파일이 존재하지 않음.");
}
```

---

2. `FileReader` – 문자 기반 텍스트 파일 입력

> 파일의 내용을 **문자 단위로 읽을 때** 사용하며, 내부적으로 FileInputStream + 인코딩을 처리함.

✅ 기본 사용

```
try (FileReader fr = new FileReader("input.txt")) {
    int ch;
    while ((ch = fr.read()) != -1) {
        System.out.print((char) ch);
    }
}
```

⚠️ 특징

- 인코딩을 지정하지 않으면 **기본 플랫폼 인코딩** 사용
- 텍스트 파일만 처리 (바이너리 파일은 적합하지 않음)
- **BufferedReader** 와 함께 쓰면 성능 향상됨

🟥 3. `Files` 클래스 – NIO.2 기반 고성능 파일 처리

> `java.nio.file.Files`는 자바 7부터 추가된 **정적 메서드 중심 유틸리티 클래스**이다.
> `Path` 객체와 함께 사용하며, **파일을 한 번에 읽거나 쓰기**에 매우 편리하다.

🔹 주요 메서드

메서드	설명
`Files.readAllLines(Path)`	텍스트 파일을 줄 단위로 모두 읽음
`Files.readAllBytes(Path)`	전체 내용을 바이트 배열로 읽음
`Files.write(Path, List<String>)`	줄 리스트를 파일로 출력
`Files.write(Path, byte[])`	바이트 배열을 파일로 출력
`Files.copy(...)`, `Files.move(...)`	파일 복사, 이동
`Files.delete(Path)`	파일 삭제

✅ 예제 1: 한 줄씩 읽기

```
Path path = Paths.get("input.txt");
List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);

for (String line : lines) {
    System.out.println(line);
}
```

✅ 예제 2: 파일 쓰기

```
List<String> lines = List.of("Hello", "World", "Java");
Files.write(Paths.get("output.txt"), lines, StandardCharsets.UTF_8);
```

✅ 예제 3: 바이트 복사

```
byte[] data = Files.readAllBytes(Paths.get("logo.png"));
Files.write(Paths.get("copy_logo.png"), data);
```

🧩 `File` vs `FileReader` vs `Files` 비교

항목	File	FileReader	Files (NIO.2)
주 용도	경로 관리, 파일 존재 확인 등	텍스트 파일 문자 단위 읽기	전체 읽기/쓰기, 고성능 파일 유틸
입출력 기능	직접 입출력 기능 없음	입력 전용 (텍스트)	입력/출력 모두 지원

항목	File	FileReader	Files (NIO.2)
인코딩 처리	없음	기본 인코딩 사용 (UTF-8 권장 아님)	명시적으로 인코딩 지정 가능
쓰기 기능	없음	없음	`Files.write()`으로 간단히 처리
고성능 여부	낮음	중간	높음 (시스템 콜 줄임)

✅ 요약

방식	장점	단점	추천 상황
`File`	경로 및 속성 확인, 디렉토리 탐색 등 용이	입출력 기능은 없음	존재 여부, 파일 메타 정보 확인
`FileReader`	문자 기반 읽기용, 간단한 텍스트 입력 가능	성능 한계, 인코딩 제약	단순 텍스트 파일 읽기
`Files`	짧은 코드, 성능 우수, 인코딩 제어 용이	전체 파일만 처리 가능	실무 입출력, 설정 파일, 로그 등

직렬화 (`Serializable`, `transient`)

Java의 직렬화(serialization)는 객체를 **바이트 스트림으로 변환**하여 파일이나 네트워크로 저장/전송할 수 있게 하는 기능이다. 반대로 역직렬화(deserialization)는 이 바이트 스트림을 다시 객체로 복원하는 과정이다. 이 과정에서 핵심이 되는 인터페이스가 `Serializable`이고, 직렬화에서 제외할 필드는 `transient` 키워드를 사용해 처리한다.

🟦 1. `Serializable` 인터페이스

✅ 정의

```
public interface Serializable {
}
```

- **마커 인터페이스**: 아무 메서드도 정의되어 있지 않음.
- JVM은 이 인터페이스가 구현된 객체만 직렬화 허용함.

🔷 사용 예제

```
import java.io.*;

class Person implements Serializable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
// 직렬화
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("person.dat"));
Person p = new Person("Alice", 30);
out.writeObject(p);
out.close();
```

```
// 역직렬화
ObjectInputStream in = new ObjectInputStream(new FileInputStream("person.dat"));
Person p2 = (Person) in.readObject();
in.close();
```

🟧 2. `transient` 키워드

✅ 정의

- `transient` 키워드가 붙은 필드는 **직렬화 대상에서 제외**됨
- 주로 다음 상황에서 사용됨:
  - 민감 정보 (비밀번호 등)
  - 네트워크/파일 핸들 등 직렬화 불가능한 필드
  - 재계산 가능한 캐시 값

```
class User implements Serializable {
    private String username;
    private transient String password;  // 직렬화 대상 아님
}
```

🟥 주의 사항

❗ static 필드도 직렬화되지 않음

- `static`은 클래스 레벨이므로 인스턴스가 아니라 직렬화 대상 아님

🧩 serialVersionUID

```
1 private static final long serialVersionUID = 1L;
```

- 직렬화/역직렬화 시 클래스 호환성을 검사하는 **버전 ID**
- 생략하면 자동 생성되는데, 클래스 변경 시 호환 에러 발생 가능
- 명시적으로 선언하면 유지 보수에 안정성 증가

📋 전체 흐름 요약

요소	역할
`Serializable`	직렬화 가능한 객체 표시
`transient`	직렬화 제외 필드 지정
`ObjectOutputStream`	객체 → 바이트 스트림
`ObjectInputStream`	바이트 스트림 → 객체
`serialVersionUID`	클래스 버전 호환 관리용 ID

✅ 실전에서 주의할 점

- 클래스 변경 시 `serialVersionUID`를 명시해 호환성 유지
- DB 세션/네트워크 통신에서 객체 전달 시 유용
- 보안상 민감한 정보는 직렬화하지 말거나 암호화 필요