




# 자바 입출력(IO)과 예외(Exception)

1조 김건영, 김동욱, 송대현, 이재원, 홍민혁, 정승주



프로그램이란 보통 0개 이상의 입력을 받고 처리를 통해 최소 1개 이상 출력을 하는것이 보통입니다.


자바 기본 API에서 이런 입출력(IO)을 해주는 많은 클래스들이 있습니다.

장치와 입출력을 위해서는 하드웨어 장치에 직접 접근이 필요한데

다양한 매체에 존재하는 데이터들을 사용하기 위한 API 가 이미 있으며 입출력 장치에서 데이터를 읽고 쓰는 클래스를

스트림(Stream) 이라고 하며, 모든 스트림은 단방향이며 각각의 장치마다 연결할 수 있는 스트림이 존재합니다.

또한 하나의 스트림은 입,출력을 동시에 수행할 수 없으므로 각각 입력과 출력을 수행하는 스트림이 필요하게 됩니다.



스트림은 바이트 단위로 처리하는 **바이트 기반 스트림**과  
데이터를 문자 단위로 처리하는 **문자 기반 스트림**이 있습니다.

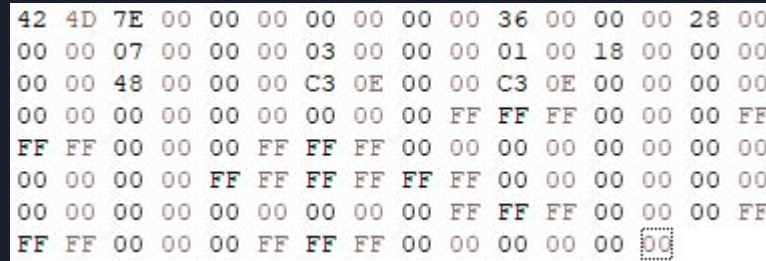
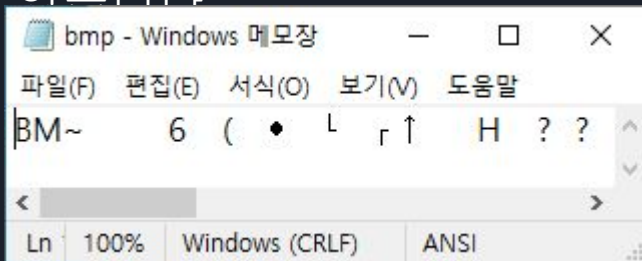
스트림을 바이트 기반 스트림과 문자 기반 스트림으로 제공하는  
이유는 우리가 프로그램을 만들때 문자 단위로 처리하게 되면

문자의 입출력에는 편리하지만 **그림, 오디오, 비디오 등과 같이  
문자로 표현하기 어려운 종류의 데이터를 입, 출력할 수 없다는**

단점이 있으며 반대로 모두 바이트 기반 스트림을 사용하게 되면  
모든 종류의 데이터를 입,출력 할 수 있지만 문자 단위로 처리하는게  
아닌 바이트 단위로 처리하기 때문에 사용 목적에 따라 추가적으로  
다른 형태(문자 등)로 변환해야 하는 단점이 있습니다.

그리고 또한 **문자 스트림은 문자단위(2byte) 로** 데이터를 입출력 하기  
때문에 한번에 처리하는 데이터량이 다를 수도 있습니다.

간단한 예시로 비교적 구조가 간단한 “BMP 이미지”의 데이터를  
바이너리 뷰어와 일반 텍스트 에디터로 확인해보면  
이미지 파일의 데이터를 문자 단위로 처리할 수 없다는 것을 알 수  
있습니다.



초반에 BM은 비트맵파일이라는 의미로 42(66) 4D(77) 부분에 해당  
하며 7E 00 00 00은 파일크기(126byte)를 의미하는데 텍스트  
에디터에서는 ~ 와 NUL로 인식하는 것을 볼 수 있습니다.

반대로 (KH) 를 입력한 “텍스트 파일”을 열어보면  
4B 48 = (75,72) = K,H 와 같이 해당하는 문자를 나타내는  
값이 있는것을 알 수 있습니다.



스트림(Stream)은 처리 단위 외에도

실제적으로 연결되는 스트림 (기반스트림) 과 기반 스트림을 보조할 목적으로 사용하는 보조스트림으로 나눌 수 있습니다.

기반과 보조의 가장 큰 차이점은

기반 스트림은 실제로 연결되는 역할을 하는 스트림으로 외부자원을 받아 동작하기 때문에 단독으로도 사용할 수 있지만

보조 스트림은 이름처럼 기반 스트림을 보조하거나 꾸며주는 역할로 기반 스트림과 연결되어 같이 사용하기 때문에 단독으로 사용하지 못하며, 스트림의 기능을 향상시키거나 새로운 기능을 추가하기 위해 사용하며 실제 데이터를 주고 받는 스트림은 아닙니다.

그 때문에 기반 스트림을 생성한 후 이를 이용하여 보조 스트림을 생성 해야 합니다.




기반 스트림의 생성자에 외부 자원으로 2가지를 실습해보았습니다.

`System.in` `System.out` 을 외부 자원으로 해서  
콘솔창을 이용하는 입출력과

`new File("file..path")` 또는 `String` 으로 `"file..path"` 를 넘겨서  
파일을 이용하는 입출력이 있었습니다.

그 외에도 간단하게 `Socket`의 `getInputStream()` 등을 이용해  
입력 스트림 등을 가져와 네트워크 통신에 `Stream` 을 사용했습니다.  
(네트워크 입출력은 바이트스트림만 지원됩니다)

스트림 클래스에 대한 객체 생성은 외부 자원과의 통로를  
만드는것으로 먼저 외부자원과 스트림부터 연결시키고,  
입/출력은 해당 객체의 읽기,쓰기 메소드를 사용합니다.




바이트 기반의 입력, 출력 스트림은 각각 **InputStream, OutputStream** 클래스를 상속받고 있으며 하위 클래스는 모두 이 최상위 클래스의 이름이 뒤에 붙어있습니다.

**InputStream**: 바이트 기반 입력 스트림의 최상위 클래스로 추상 클래스입니다.

바이트 기반 입력 스트림으로는 아래와 같이 있습니다

- **FileInputStream**
  - 파일로부터 바이트 단위의 입력을 받는 클래스
- **PipedInputStream**
  - 스레드에서 사용
- **ByteArrayInputStream**
  - 인자로 넘어온 바이트 배열에 데이터를 입력하는데 사용




**OutputStream** : 바이트 기반 출력 스트림의 최상위 클래스로 추상 클래스입니다.

바이트 기반 출력 스트림으로는 아래와 같이 있습니다

- **FileOutputStream**
  - 파일로부터 바이트 단위로 저장할 때 사용
- **PipedOutputStream**
  - 스레드에서 사용
- **ByteArrayOutputStream**
  - 인자로 넘어온 바이트 배열로 데이터를 출력하는데 사용

출력 스트림은 **flush()** 메소드로 버퍼에 잔류하는 모든 바이트를 출력하거나 스트림을 닫아주지 않으면 데이터의 일부가 버퍼에 남아있어 출력되지 않는 문제가 있을 수 있기 때문에 **작업이 끝나면** **입,출력 스트림을 close()** 로 닫아주도록 합니다.





문자 기반의 입력, 출력 스트림은 각각  
**Reader, Writer** 클래스를 상속받고 있으며  
하위 클래스는 모두 이 최상위 클래스의 이름이 뒤에 붙어있습니다.

**Reader** : 문자 기반 입력 스트림의 최상위 클래스로 추상  
클래스입니다.

문자 기반 입력 스트림으로는 아래와 같이 있습니다

- **CharArrayReader**
  - 문자 배열에서 읽어올때 사용
- **FileReader**
  - 텍스트 파일로부터 문자 단위로 읽을 때 사용
- **PipedReader**
  - 스레드에서 사용
- **StringReader**
  - 문자열을 읽어올때 사용



**Writer** : 문자 기반 출력 스트림의 최상위 클래스로 추상 클래스입니다.

문자 기반 출력 스트림으로는 아래와 같이 있습니다

- **CharArrayWriter**
  - 문자배열로 출력할 때 사용
- **FileWriter**
  - 텍스트 파일로부터 문자 단위로 출력할 때 사용
- **PipedWriter**
  - 스레드에서 사용
- **StringWriter**
  - 문자열을 출력할 때 사용한다.

출력 스트림은 **flush()** 메소드로 버퍼에 잔류하는 모든 문자열을 출력하거나 스트림을 닫아주지 않으면 데이터의 일부가 버퍼에 남아있어 출력되지 않는 문제가 있을 수 있기 때문에 **작업이 끝나면** 입,출력 스트림을 **close()** 로 닫아주도록 합니다.



다양한 보조 스트림 종류

### # 문자 변환 (InputStreamReader/OutputStreamWriter)

- 소스 스트림이 바이트 기반 스트림이지만 데이터가 문자일 경우 바이트 기반 스트림보다 편리하게 사용 가능합니다.

### # 입출력 성능 (BufferedInputStream/BufferedOutputStream)

- 느린 속도로 인해 입출력 성능에 영향을 미치는 입출력 소스를 이용하는 경우 사용하며 버퍼에 데이터를 모아 한꺼번에 작업을 하여 실행 성능을 향상합니다.

### # 기본 데이터 타입 출력 (DataInputStream, DataOutputStream)

- 기본 자료형별 데이터 읽고 쓰기가 가능하도록 기능을 제공합니다. (단, 입출력시 자료형의 순서가 일치해야 합니다)

### # 객체 입출력 (ObjectInputStream/ObjectOutputStream)

- 객체를 파일 또는 네트워크로 입출력 할 수 있는 기능을 제공합니다. (단, 객체는 문자가 아니므로 바이트 기반 스트림으로 데이터를 변경해주는 직렬화가 필수입니다)



## 직렬화

객체 상태 그대로 스트림을 전송할 수 없기 때문에  
객체 안에 기록된 **각 필드의 값들을 1바이트 단위로  
쪼개어 일렬로 나열** 시키는 것으로


직렬화를 적용할 클래스 헤더에 **java.io.Serializable**  
**인터페이스**를 상속 처리해야 합니다.

## 역직렬화

직렬화된 객체를 역직렬화할 때는 직렬화 했을 때와 같은 클래스를  
사용하지만 클래스 이름이 같더라도 **클래스 내용이 변경된 경우**  
**역직렬화가 실패**할 수 있습니다.

**serialVersionUID 필드 (private static final serialVersionUID)**

직렬화한 클래스와 같은 클래스임을 알려주는 식별자 역할로 **JVM이**  
**자동으로 생성** 시 **역직렬화에서 예상 못한 예외를 유발** 할 수 있어  
명시를 권장합니다.



객체 입출력 보조 스트림


객체를 파일 또는 네트워크로 입출력 할 수 있는 기능을  
제공하지만 객체는 문자가 아니므로 **바이트 기반  
스트림으로 데이터를 변경** 해주는 **직렬화가 필수**입니다.

## File 클래스

파일 시스템의 파일을 표현하는 클래스로 **파일 크기, 파일 속성, 파일  
이름 등의 정보와 파일 생성 및 삭제 기능** 제공합니다.

출력 스트림 객체를 생성할때 생성자에 **“File..path”**만 넘겨줄 경우  
기존 파일에 이어 작성하는게 아니라 해당 파일이 새로 작성해  
데이터가 유지되지 않을 수 있기 때문에 기존 데이터에 이어서  
작성하기를 원하면 생성자 매개변수에 **true** 를 붙여야 합니다.

**JAVA DOC API** 페이지에서 패키지에 있는 클래스, 메소드와  
발생할 수 있는 예외(**Exception**) 을 확인 할 수 있습니다.



JAVA API 문서에서 `java.io` 패키지를 보면 다양한 클래스가 있습니다.  
패키지에서 `OutputStream` 만 한번 알아보도록 하겠습니다.

`OutputStream`은 바이트 기반 출력 스트림의 상위  
추상 클래스로 몇가지 `Method`을 가지고 있습니다.

`close()`

이 출력 스트림을 닫고 이 스트림과 관련된 모든 시스템 리소스를  
해제합니다.

`flush()`

이 출력 스트림의 버퍼에 남아 있는 출력 바이트를 강제로 기록합니다.

`write(byte[] b)`


지정된 바이트 배열의 바이트를 출력 스트림에 출력합니다.

`write(byte[] b, int off, int len)`

지정된 배열에서 `off`(오프셋) 부터 시작하여 `len` 바이트를 출력  
스트림에 사용합니다.

`write(int b)`

지정된 바이트를 출력 스트림으로 출력합니다.




모든 입력, 출력 스트림은 꼭 사용 후 `close()` 메소드를 사용해 스트림을 닫고 이 스트림과 관련된 모든 시스템 리소스를 돌려줘야 합니다.

이번에는 예외(Exception)에 대해 알아보겠습니다.

프로그램을 작성하다보면 가끔 오류를 만나는 일이 있을 수 있습니다.

이 오류는 다음과 같이 3가지로 나눌 수 있습니다.

1. **컴파일 에러** : 프로그램의 실행을 막는 소스 상의 문법 에러, 소스 구문을 수정하여 해결
2. **런타임 에러** : 배열의 인덱스 범위를 벗어나거나 계산식의 오류 등으로 주로 IF 문 사용으로 에러를 처리
3. **시스템 에러** : 컴퓨터 오작동으로 인한 에러



소스 수정으로 해결 가능한 에러를 예외(Exception)이라고 하는데 이러한 예외 상황(예측 가능한 에러) 구문을 처리하는 방법인 예외 처리를 통해 해결합니다.

여기서 모든 예외의 최고 조상은 **Exception** 클래스이며 아래와 같이 나눌 수 있습니다.


### Checked Exception

반드시 코드 작성과 함께 예외 처리를 같이 작성해야 합니다.

### Unchecked Exception

주로 프로그래머의 부주의로 인한 오류인 경우가 많으며 예외 처리보다는 코드를 수정해야 하는 경우가 많습니다.  
대부분 **RuntimeException** 의 하위 클래스 입니다.






```
try {
    input = sc.nextInt();
} catch (java.util.InputMismatchException e) {
    System.out.println("숫자가 아닌 타입이 입력되어");
    System.out.println("Exception(예외) 발생 !");
}
System.out.println(input);
Scanner 클래스 사용시 발생할 수 있는 예외 및 출력 예시
```

코드 작성시 예외처리가 필수가 아닌 **Unchecked Exception** 중 하나의 예시로 **Scanner** 을 예외 처리해보았습니다.

**Scanner** 은 `nextInt()` 로 정수 타입을 입력받으려고 할 때 다른 타입 (문자) 가 입력되면 **InputMismatchException** 을 발생시키지만 평소 작성할때 예외처리를 필수로 하지는 않았습니다.



```
try {
    java.lang.String a = (String)new java.lang.Object();
    // java.lang Package 클래스들은 import 없이 사용 가능
} catch (java.lang.ClassCastException e) {
    e.printStackTrace();
    Cast 연산자 사용 시 타입 오류
    변환할 수 없는 타입으로 객체를 변환할 때 발생
    instanceof 연산자로 객체타입 확인 후 cast 연산
}
```

이번에는 ClassCastException 의 예외처리 예시입니다.

위 예외도 처리가 필수가 아닌 UnChecked Exception 이며  
캐스팅 이전에 instanceof 로 객체 타입을 확인해  
해당 예외가 발생하지 않도록 할 수 있습니다.


대부분 UnChecked Exception은 조건문을 활용해  
예외(Exception) 가 발생하지 않도록 할 수 있습니다.



코드 작성시 예외 처리도 필수로 해야 하는  
**Checked Exception** 도 있습니다.

Checked Exception 은 주로 입출력 부분에서 많이 볼 수 있습니다.


```
FileOutputStream f = null;
지역변수는 값을 직접 초기화 해야 한다.
레퍼런스 타입은 null 로 선언하고 사용하도록 한다.
try {
    f = new java.io.FileOutputStream(
        new java.io.File("test.dat"));
}
catch(FileNotFoundException e) {}
파일을 찾을 수 없을때 발생하는 예외
catch(IOException e) {}
입출력 동작 실패 또는 인터럽트 발생 시
finally {
    try { f.close();
    } catch (IOException e) {}
    //입출력 동작 실패 또는 인터럽트 발생 시}
}
```



```
FileOutputStream f = null;
지역변수는 값을 직접 초기화 해야 한다.
레퍼런스 타입은 null 로 선언하고 사용하도록 한다.
try {
    f = new java.io.FileOutputStream(
        new java.io.File("test.dat"));
}
catch(FileNotFoundException e) {}
파일을 찾을 수 없을때 발생하는 예외
catch(IOException e) {}
입출력 동작 실패 또는 인터럽트 발생 시
finally {
    try { f.close();
    } catch (IOException e) {}
    //입출력 동작 실패 또는 인터럽트 발생 시}
}
```

만약 예외 처리를 작성하지 않을 경우 컴파일러가  
예외처리 작성을 요구하며

처리 방법에는 try ~ catch , try ~ with ~ resource, throws 가 있습니다.  
위 예시는 try ~ catch 로 예외 처리한 예시입니다.



```
try (FileOutputStream f = new java.io.FileOutputStream  
    (new java.io.File("test.dat"))){  
}  
catch(FileNotFoundException e) {}  
파일을 찾을 수 없을때 발생하는 예외  
catch(IOException e) {}
```

조금 전 예외 처리 try ~ catch 문을  
try ~ with ~ resource 로 바꾸면 다음과 같습니다.

스트림을 닫는 부분을 생략 할 수 있으며 1.7부터 가능합니다.

그리고 마지막으로 throws 는 메소드에 붙여 자신을 호출한 곳으로  
발생한 예외 및 처리를 넘길 수 있으며 main 에서 넘기게 될 경우  
JVM이 처리하게 됩니다.