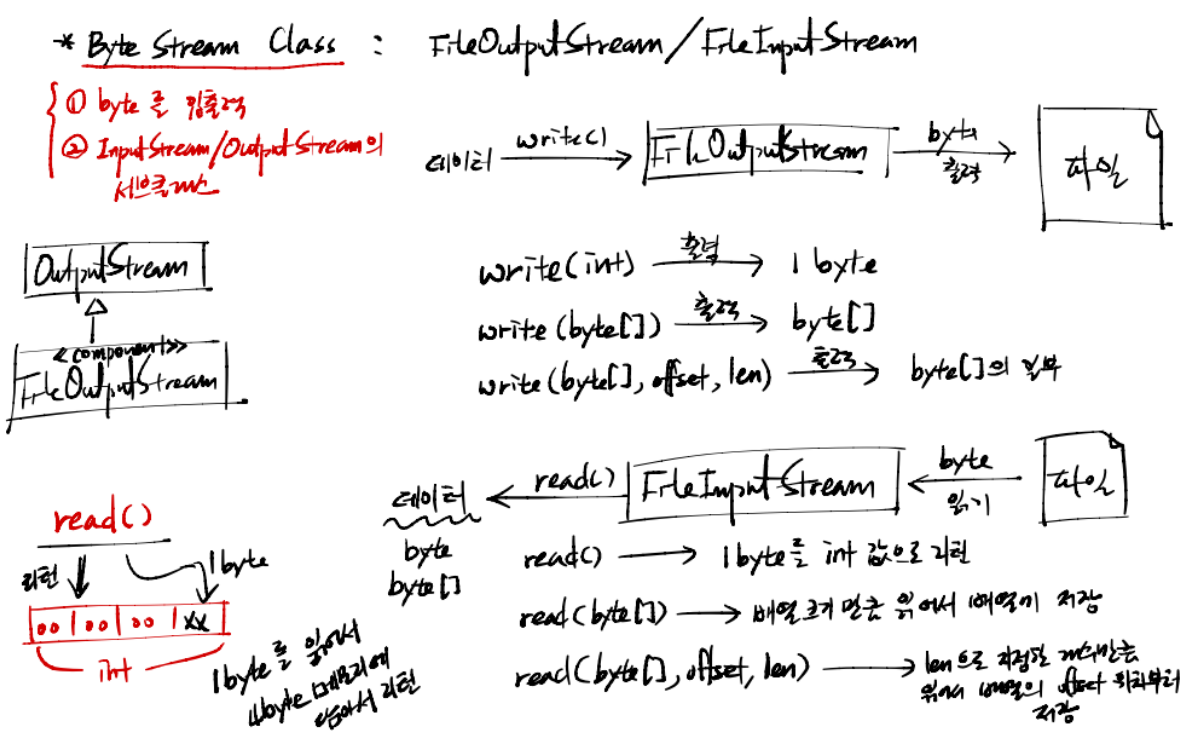
파일 시스템 (1)

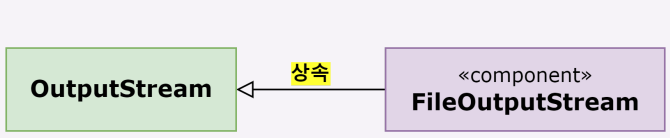
Confidence	0%
Last Edited	@August 1, 2024 11:06 AM
Class	java.io
생성 일시	@July 27, 2024 11:21 PM
다중 선택	
태그	

▷ Byte Stream (IO-EX02)

※ Byte Stream Class : FileOutputStream / FileInputStream



Byte Stream Class (Java 클래스 설명서 : java.io)



1. byte를 입출력
2. InputStream/OutputStream의 서브 클래스는 다 Byte Stream이다

FileOutputStream 클래스

```
                출력
write(int)  ————————>  1 byte

write(byte[])  ————————>  byte[]

write(byte[], offset, len)  ————————>  byte[]의 일부
```

byte 출력하기

1. **파일로 데이터를 출력하는 객체를 준비**

```
FileOutputStream out = new FileOutputStream("temp/test1.data");
                       new FileOutputStream(파일경로)
```

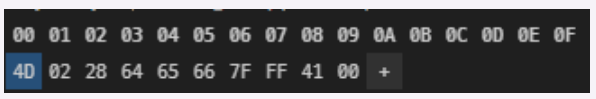
- 지정된 경로에 해당 파일을 자동 생성
- 기존에 같은 이름의 파일이 있으면 덮어쓴다.
- 파일 경로가 절대 경로가 아니면(윈도우 : c:\, d:\ 등으로 시작하지 않으면),

  현재 디렉토리가 기준이 된다.

2. **write(int)로 1바이트를 출력한다.**
   - **4 바이트 int를 넣으면 맨 끝 1바이트만 출력**

```
out.write(0x7a6b5c4d); // =>  0x4d
out.write(2);          // 0x00000002   =>  0x02
out.write(40);         // 0x00000028   =>  0x28
out.write(255);        // 0x000000ff
out.write('A');        // 0x0041   => 0x41
out.write('가');       // 0xac00   => 0x00
```

   - **파일 결과**



3. **출력 도구를 닫기**
   - **close( )**
     - 보통 **java.lang.AutoCloseable 인터페이스**를 **구현**하고 있다.

       (FileOutputStream 클래스에도 close() 존재)
     - **close( )를 호출하면,**
       - FileOutputStream이 작업하는 동안 사용했던 **버퍼(임시메모리)를 비운다**.
       - **OS에서 제공한 파일과의 연결을 끊는다**.

   - 24시간 365일 멈추지 않고 실행하는 **서버 프로그램인 경우**

     ⇒ **사용한 자원을 즉시 해제시키지 않으면, 자원 부족 문제가 발생**

   - **AutoCloseable 인터페이스를 구현한 클래스**를 **사용할 때**는

     사용하고 난 후 자원을 해제시는 **close( )를 반드시 호출**하라.

- **바이트 배열(byte[ ]) 출력**

```
byte[] bytes = {0x7a, 0x6b, 0x5c, 0x4d, 0x3e, 0x2f, 0x30};

2가지 방법
 1.  write(byte[]) : 배열의 값 전체를 출력한다.
```

```
2.  write(byte[], 시작인덱스, 출력개수) : 시작 위치부터 지정된 개수를 출력한다.
```

- **byte[ ] 전체 출력**

```
byte[] bytes = {0x7a, 0x6b, 0x5c, 0x4d, 0x3e, 0x2f, 0x30};

out.write(bytes);
```

- **byte[ ] 일부 출력**

```
byte[] bytes = {0x7a, 0x6b, 0x5c, 0x4d, 0x3e, 0x2f, 0x30};

out.write(bytes, 2, 3);    2번 데이터부터 3 바이트를 출력
```

FileInputStream 클래스


```
read()  ———>  1 byte를 int 값으로 return

read(byte[])  ———>  배열 크기만큼 읽어서 배열에 저장

read(byte[], offset, len)  ———>  len으로 지정한 개수만큼 읽어서 배열의 offset 위치 부터 저장
```

- **read( )**

```
         return                      1 byte를 읽어서 담는다.
read()  ======>  4 byte int값  [00][00][00][XX]
```

  - **read( )는 1 byte를 읽어서 4 byte int의 마지막 byte에 넣어 반환한다.**
    (앞의 3 byte는 0으로 채워져 있다)
  - **이유?**
    - 0 ~ 255까지의 값을 읽기 때문이다.
    - byte는 -128 ~ 127까지의 값만 저장한다.
  - **read( )를 호출할 때마다 이전에 읽은 바이트의 다음 바이트를 읽는다.**

- **파일을 전체 다 읽기**

```
FileInputStream in = new FileInputStream("temp/test1.data");
```

```
while ((b = in.read()) != -1) {
   System.out.printf("%02x ", b);
}
```

- **위의 코드는 아래 코드와 동일하다.**

```
int b;
while (true) {
   b = in.read();
   if (b == -1) //
      break;
   System.out.printf("%02x ", b);
}
```

  - **파일의 끝에 도달하면 -1을 리턴한다.**

    ⇒ **int를 return하면 앞의 3 byte는 00이므로 절대 (-)가 될 수 없으므로**

- **버퍼 (Buffer)**
  - **read( )는 배열을 안 만든다. ⇒ 미리 저장할 배열을 만들어서 넘겨줘야 한다.**
  - **이렇게 임시 데이터를 저장하기 위해 만든 바이트 배열을 보통 "버퍼(buffer)"라 한다**

  - **read(byte[ ])**

```
byte[] buf = new byte[100];

int count = in.read(buf);
```

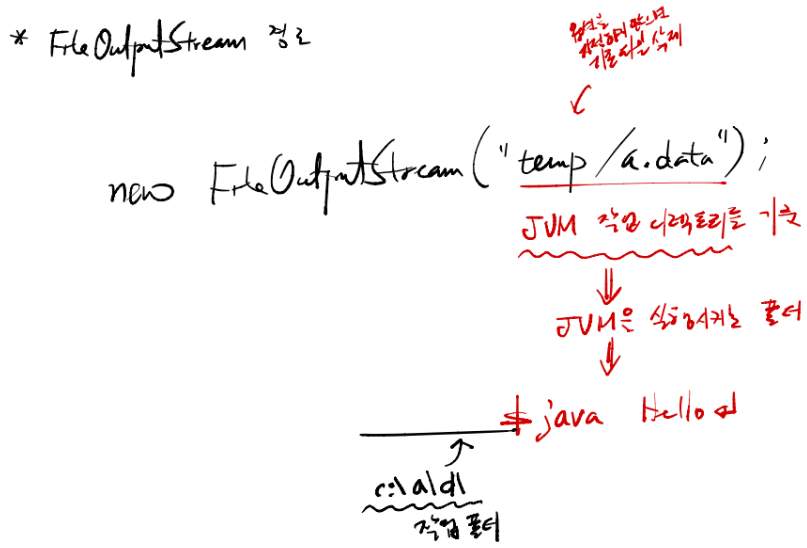
    - **버퍼가 꽉 찰 때까지 읽는다.**
    - 물론 버퍼 크기보다 파일의 데이터가 적으면 파일을 모두 읽어 버퍼에 저장한다.
    - **리턴 값은 읽은 바이트의 개수이다.**

  - **read(byte[ ], 저장할 위치, 저장하기를 희망하는 개수)**

```
byte[] buf = new byte[100];

int count = in.read(buf, 10, 40);    40바이트를 읽어 10번 방부터 저장
```

    - 읽은 데이터를 "저장할 위치"에 지정된 방부터 개수만큼 저장한다.
    - **리턴 값은 실제 읽은 바이트의 개수이다**
    - **"빈 배열을 줄 테니까 파일에서 40개를 읽어서 배열의 10번째부터 채워줘"**

※ FileOutputStream 경로



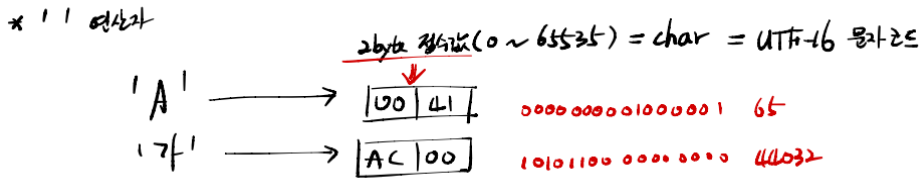
- **JVM 작업 디렉토리 = JVM을 실행시키는 폴더**

```
____________ $ java Hello
  c:\a\d\ : 현재 작업 폴더(Working directory)


이클립스의 경우 현재 작업 폴더  :  java-lang(프로젝트 폴더)\app\


$ pwd  =>  print working directory (현재 위치를 알려준다)
```

```
new FileOutputStream("temp/a.data");
                      └ JVM 작업 디렉토리를 기준
```

  - **만약에 같은 이름의 파일이 있으면 기존 파일을 지운다. (옵션을 지정하는 경우 이어서 작성 가능)**
  - **해당 경로에 파일이 존재하지 않으면 예외 발생 !**

※ '' 연산자




```
                      ┌ 2byte 정수값(0 ~ 65535) = char = UTF-16 문자코드
                      │
'A'  ---->  0x0041 : [00][41]  =>  0000 0000 0100 0001  =>  65

'가' ---->  0xAC00 : [AC][00]  =>  1010 1100 0000 0000  =>  44032


out.write('A');   // 0x0041   => 0x41 출력  (1 byte만 출력하므로)
out.write('가');  // 0xac00   => 0x00 출력
```

파일 포맷

- 파일 포맷에 대해 알아내어 데이터 읽기
  1. 파일 정보를 준비

     ```
     File file = new File("sample/photo1.jpg");
     ```

     - 작업 폴더 : java-lang\app\ (src\와 동일한 위치에 sample이 있다)

  2. 파일을 읽을 도구를 준비

     ```
     FileInputStream in = new FileInputStream(file);
     ```

     - 바이너리 데이터에서 파일 포맷에는 byte의 내용이 정해져 있다

  3. [Ex] SOI(Start of Image) Segment 읽기 : 2바이트

     ```
     int b1 = in.read(); // 00 00 00 ff
     int b2 = in.read(); // 00 00 00 d8
     int soi = b1 << 8 | b2;   합치기
     ```

     ▼ [Ex02-0410] FileInputStream 활용 - JPEG 파일 읽기 (포맷 정보를 알아내어 하는법)

     ```
     // FileInputStream 활용 - JPEG 파일 읽기
     package com.eomcs.io.ex02;

     import java.io.File;
     import java.io.FileInputStream;

     public class Exam0410 {

       public static void main(String[] args) throws Exception {

         // 1) 파일 정보를 준비한다.
         File file = new File("sample/photo1.jpg");   => 작업 폴더 : java-lang\app\  (src\와 동일한 위치에

         // 2) 파일을 읽을 도구를 준비한다.
         FileInputStream in = new FileInputStream(file);

     바이너리 데이터에서 파일 포맷은 byte의 내용이 정해져 있다.
         // => SOI(Start of Image) Segment 읽기: 2바이트
         int b1 = in.read(); // 00 00 00 ff
         int b2 = in.read(); // 00 00 00 d8
         int soi = b1 << 8 | b2;   합치기
         //   00 00 00 ff <== b1
         //   00 00 ff 00 <== b1 << 8
         // | 00 00 00 d8 <== b2
         // ------------------
         //   00 00 ff d8
         System.out.printf("SOI: %x\n", soi);

         // => JFIF-APP0 Segment Marker 읽기: 2바이트
         int jfifApp0Marker = in.read() << 8 | in.read();
         System.out.printf("JFIF APP0 Marker: %x\n", jfifApp0Marker);

         // => JFIF-APP0 Length: 2바이트
         int jfifApp0Length = in.read() << 8 | in.read();
         System.out.printf("JFIF APP0 정보 길이: %d\n", jfifApp0Length);
     ```

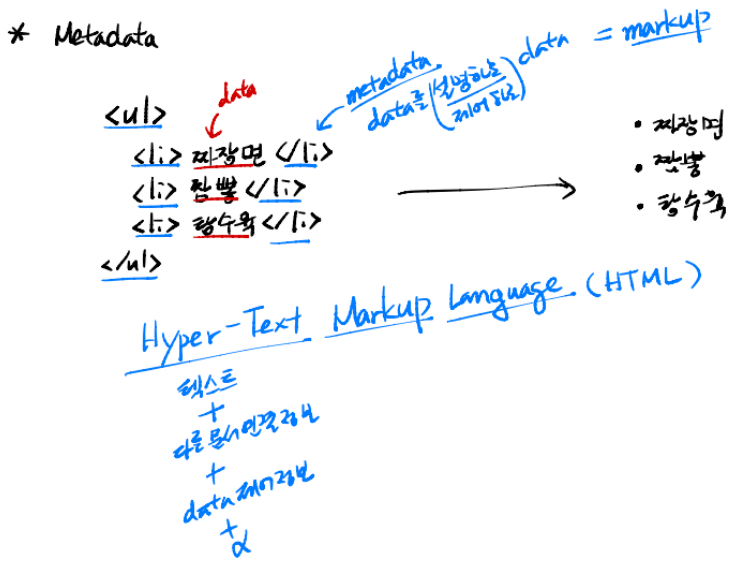
```
// => JFIF-APP0 정보: 16바이트(위에서 알아낸 길이)
byte[] jfifApp0Info = new byte[jfifApp0Length];
in.read(jfifApp0Info);

// => JFIF-APP0 Identifier: 5바이트
String jfifApp0Id = new String(jfifApp0Info, 0, 4);
System.out.printf("JFIF APP0 ID: %s\n", jfifApp0Id);

// SOF0(Start of Frame) 정보 읽기
// - 그림 이미지의 크기 및 샘플링에 관한 정보를 보관하고 있다
// - 0xFFC0 ~ 0xFFC2 로 표시한다.

// => SOF Marker 찾기
int b;
while (true) {
  b = in.read();
  if (b == -1) { // 파일 끝에 도달
    break;
  }
  if (b == 0xFF) {
    b = in.read();
    if (b == -1) { // 파일 끝에 도달
      break;
    }
    if (b >= 0xC0 && b <= 0xC2) {
      break;
    }
  }
}

if (b == -1) {
  System.out.println("유효한 JPEG 파일이 아닙니다.");
  return;
}

// => SOF Length 읽기: 2바이트
int sofLength = in.read() << 9 | in.read();
System.out.printf("SOF 데이터 크기: %d\n", sofLength);

// => SOF 데이터 읽기: 17바이트(위에서 알아낸 크기)
byte[] sofData = new byte[sofLength];
in.read(sofData);

    // => SOF 샘플링 정밀도: 1바이트
    System.out.printf("SOF 샘플링 정밀도: %d\n", sofData[0]);

    // => SOF 이미지 높이: 2바이트
    int height = ((sofData[1] << 8) & 0xff00) | (sofData[2] & 0xff);

    // => SOF 이미지 너비: 2바이트
    int width = ((sofData[3] << 8) & 0xff00) | (sofData[4] & 0xff);
    System.out.printf("SOF 이미지 크기(w x h): %d x %d\n", width, height);


    // 3) 읽기 도구를 닫는다.
    in.close();
  }

}
```

```
SOI: ffd8
JFIF APP0 Marker: ffe0
JFIF APP0 정보 길이: 16
JFIF APP0 ID: JFIF
SOF 데이터 크기: 17
SOF 샘플링 정밀도: 8
SOF 이미지 크기(w x h): 4032 x 3024
```

- 라이브러리를 사용하여 데이터 읽기

```
import com.drew.imaging.ImageMetadataReader;
import com.drew.metadata.Metadata;
import com.drew.metadata.exif.GpsDirectory;

File file = new File("sample/gps-test.jpeg");
Metadata metadata = ImageMetadataReader.readMetadata(file);
GpsDirectory gpsDirectory = metadata.getFirstDirectoryOfType(GpsDirectory.class);


if (gpsDirectory != null) {    사진 파일에 GPS 정보가 있을 경우,
  System.out.println(gpsDirectory.getGeoLocation().getLatitude());
  System.out.println(gpsDirectory.getGeoLocation().getLongitude());
}
```

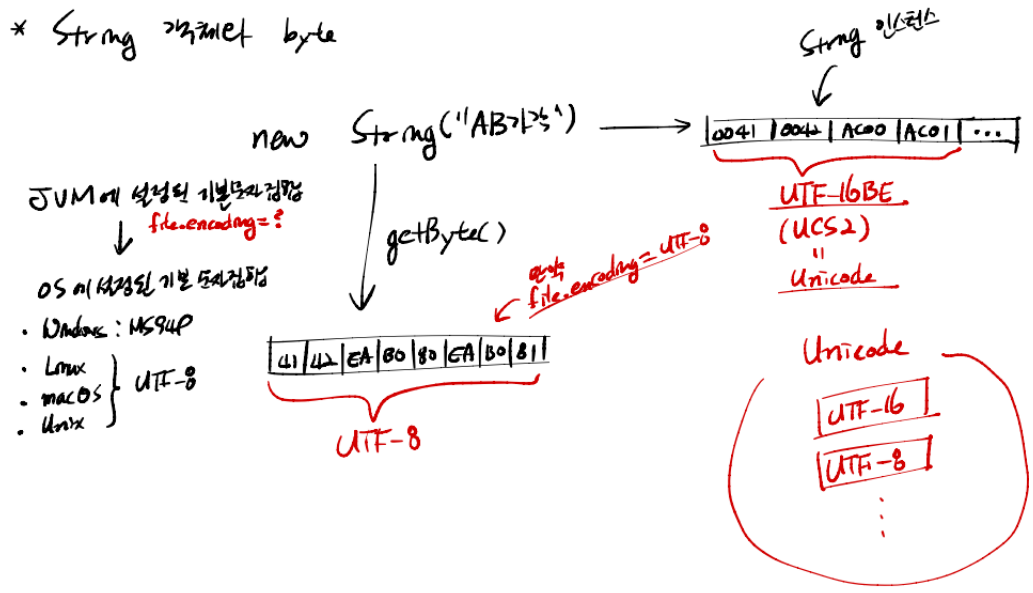
※ Metadata




```
<ul>     ┌─ data
  <l:>짜장면</l:>                 * 짜장면
  <l:>짬뽕</l:>     ──────────▶   * 짬뽕
  <l:>탕수육</l:>                 * 탕수육
</ul>
  └─ metadata : data를 설명하는(제어하는) 데이터   =   markup
```

- **Hyper-Text Markup Languauge (HTML)**
  - **텍스트 + 다른 문서 연결 정보 + data 제어 정보 + α**

※ ★ String 객체와 byte (데이터 쓰기)




```
                              2 byte                    클래스정보
new String("AB가각");     =>    [0041][0042][AC00][AC01][...]   :  String 인스턴스
                                  └──── UTF-16BE (UCS2)─┘
                                        =  Unicode
```

- JVM에서는 내부적으로 문자를 다룰 땐 UTF-16으로 다룬다.
  - String 객체 생성 시, 내부 문자열을 UTF-16(UCS2)로 인코딩하여 내부적으로 저장
  - String 객체의 데이터를 출력하려면 ⇒ 문자열을 담은 byte[ ] 배열을 리턴 받아야 한다.
  - byte[ ]을 만들기 위해서 String 클래스의 getBytes( )를 사용한다.

- String.getBytes( ) 할 때, 문자집합을 지정하지 않으면,

  ⇒ JVM에 설정된 기본 문자집합(file.encoding = ?)에 따라 인코딩하여 byte[ ]에 저장한다

  ⇒ 한글이 깨지는 이유

  - 이클립스를 사용하는 경우
    - 자바 앱을 실행할 때 file.encoding 변수의 값을 UTF-8로 설정한다.

      즉, 이클립스에서 애플리케이션을 실행할 때 다음과 같이 JVM 환경변수를 자동으로 붙인다.

      ```
      $ java -Dfile.encoding=UTF-8 ....
      ```

      - 그래서 getBytes( )가 리턴한 바이트 배열의 인코딩은 UTF-8이 되는 것이다.

    - getBytes( ) 시 만약 file.encoding = UTF-8 인 경우 ⇒ UTF-8 코드로 인코딩되서 출력
      - 문자열 인코딩 : UCS2 → UTF-8

      ```
                                  getBytes()
      new String("AB가각");  ──────────────▶  [41][42][EA][B0][80][EA][B0][81][....]
                                                └──────────  UTF-8  ──────────┘
      ```

- file.encoding JVM 환경 변수의 값이 설정되어 있지 않을 경우,

  (즉. 이클립스가 아니라 콘솔창에서 -Dfile.encoding=UTF-8 옵션 없이 실행하는 경우)

  ⇒ OS에 설정된 기본 문자집합 (Windows : MS949 / Linux, macOS, Unix : UTF-8)

  ⇒ getBytes( )가 리턴한 바이트 배열은 OS의 기본 인코딩으로 변환될 것이다.

  - OS에 상관없이 동일한 실행 결과를 얻고 싶다면, 다음과 같이 file.encoding 옵션을 붙여라

    JVM을 실행할 때 출력 데이터의 문자 코드표를 지정하는 방법

    ```
    $ java -Dfile.encoding=문자코드표 -cp 클래스경로 클래스명

    예) java -Dfile.encoding=UTF-8 -cp bin/main com.eomcs.io.ex03.Exam0110
    ```

  - 또는 getBytes( ) 호출할 때 인코딩할 문자집합을 지정하라.

    ```
    str.getBytes("UTF-8")
    ```

- 기타 참고
  - Unicode = [UTF-16] + [UTF-8] + ...
  - ASCII
    - 7 bit : 000 0000 ~ 111 1111
    - 1000 0000이면 ⇒ 한글이라고 보는 것

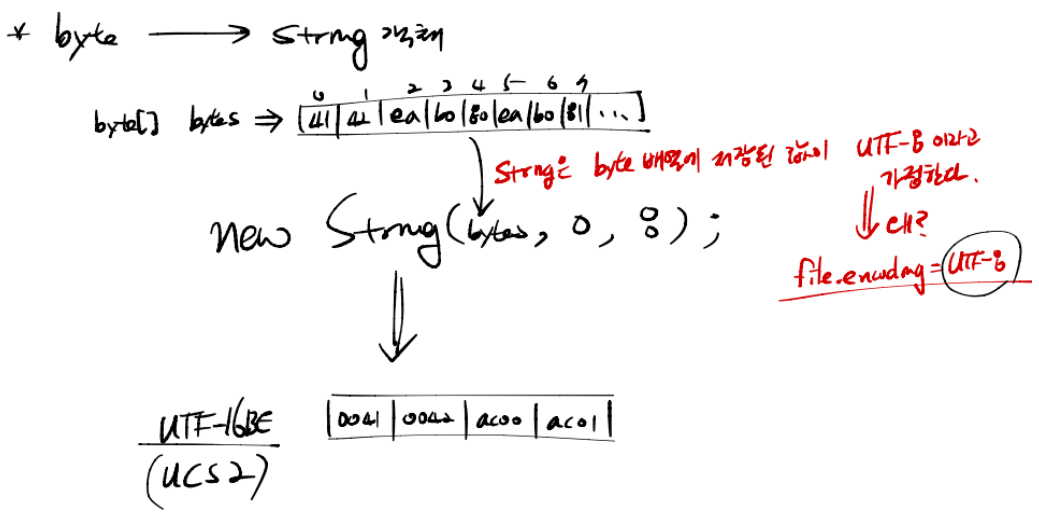
- JVM 환경 변수 'file.encoding' 값 ⇒ System.getProperty("file.encoding")

```
System.out.printf("file.encoding=%s\n", System.getProperty("file.encoding"));
```

- 특정 문자집합으로 인코딩 하여 byte[ ] 생성하기

```
byte[] bytes = str.getBytes("EUC-KR"); // UCS2 --> EUC-KR

byte[] bytes = str.getBytes("MS949"); // UCS2 --> MS949

byte[] bytes = str.getBytes("UTF-16BE"); // UCS2 --> UTF-16BE(= UCS2)

byte[] bytes = str.getBytes("UTF-16LE"); //  UCS2 --> UTF-16LE

byte[] bytes = str.getBytes("UTF-8"); //  UCS2 --> UTF-8
```

※ ★ byte ⇒ String 객체 (데이터 읽기)



- **String 객체**를 만들 때 **byte 배열**에 저장된 값이 **UTF-8**이라고 가정한다.

  ⇒ 왜? file.encoding = UTF-8 이라고 되어 있기 때문에

```
                0   1   2   3   4   5   6   7
byte[] bytes = [41][42][ea][b0][80][ea][b0][81][....]
         |
         └─┐
           |
new String(bytes, 0, 8);   =>    [0041][0042][ac00][ac01][....]
                                  └─── UTF-16BE(UCS2)─┘
```

  - **UTF-8 ⇒ UTF-16**
    - 영어는 원래의 1 byte에 앞에 1 byte(00)를 붙여서 2 byte로 만든다.

      'A' : 0x41 ⇒ 0x0041

    - 한글은 3 byte를 읽어서 2 byte로 만든다.

      '가' : 0xEAB080 ⇒ 0xAC00

- 특정 문자집합으로 인코딩된 텍스트 데이터 읽기
  - **MS949**로 인코딩된 텍스트 읽기

```
FileInputStream in = new FileInputStream("sample/ms949.txt");

while ((b = in.read()) != -1) {
  if (b >= 0x80) {     읽은 바이트가 한글에 해당한다면
     b = b << 8 | in.read();
  }
```

    - 단순히 1바이트를 읽어서는 안된다. ⇒ 한글은 2바이트를 읽어야 한다.

- 데이터를 한 번에 읽어서 String 객체로 만들기

```
FileInputStream in = new FileInputStream("UTF-8.txt");

UTF-8.txt  :  [41][42][ea][b0][80][ea][b0][81]  ("AB가각")

byte[] buf = new byte[1000];
```

```
int count = in.read(buf);
```

1. 읽을 데이터가 저장될 배열을 먼저 만들어야 한다. (버퍼 만들기)

   ⇒ 배열을 주면서 read( )를 호출한다. ⇒ 만든 배열에 값을 저장한다.

2. 만든 배열인 byte[ ]로 String 객체를 만든다.

   ```
   String str = new String(buf, 0, count);
   ```

   - 바이트 배열에 들어 있는 코드 값이 어떤 문자 집합의 값인지 알려주지 않는다면,
     - JVM 환경 변수 file.encoding에 설정된 문자 집합으로 인코딩된 것으로 간주하고 변환한다.
       - JVM이 사용하는 문자 집합(UTF16BE = UCS2)의 코드 값으로 변환한다.
       - 왜냐하면 JVM에서는 내부적으로 문자를 다룰 땐 UTF-16으로 다룬다.
         - String 객체 생성 시, 내부 문자열을 UTF-16(UCS2)로 인코딩하여 내부적으로 저장

   - 이클립스에서 실행 ⇒ (성공!) (Linux / Unix / MacOS : UTF-8)
     - JVM 실행 옵션에 '-Dfile.encoding=UTF-8' 환경 변수가 자동으로 붙는다.
     - 즉, 원본 문서가 무엇으로 인코딩되어서 byte[ ]에 들어갔는지 알려주지 않으면

       ⇒ 그냥 UTF-8로 했다고 간주하고 UTF-8 문자표를 이용해 UTF-16으로 변환한다.

       ⇒ 즉, String 클래스는 바이트 배열의 값을 UCS2로 바꿀 때 UTF-8 문자표를 사용한다.

     ```
     utf8.txt  =>  41 42 ea b0 80 ea b0 81

     UCS2      =>  0041 0042 ac00 ac01  <== 정상적으로 바뀐다.
     ```

     - 하지만, MS949로 인코딩된 바이트 배열이였다면

       ⇒ UTF-8 문자표를 사용해서 변환하니까, 잘못된 문자로 변환된다.

   - Windows 콘솔에서 실행 ⇒ (실패!)
     - 이클립스를 쓰지 않고 JVM을 실행할 때 file.encoding을 설정하지 않으면

       OS의 기본 문자집합으로 설정한다.
       - Windows의 기본 문자집합은 MS949 이다. (file.encoding = MS949)

     - 바이트 배열은 UTF-8로 인코딩 되었는데, MS949 문자표를 사용하여

       UCS2로 변환하려 하니까 잘못된 문자로 변환되는 것이다.

3. 해결책?
   - JVM을 실행할 때 file.encoding 옵션에 정확하게 해당 파일의 인코딩을 설정하라.
   - JVM을 실행할 때 출력 데이터의 문자 코드표를 지정하는 방법

     ```
     $ java -Dfile.encoding=UTF-8 -cp bin/main .....

         //       java -Dfile.encoding=문자코드표 -cp 클래스경로 클래스명
         //   예) java -Dfile.encoding=UTF-8 -cp bin/main com.eomcs.io.ex03.Exam0110
     ```

     즉 utf8.txt 파일은 UTF-8로 인코딩 되었기 때문에

'-Dfile.encoding=UTF-8' 옵션을 붙여서 실행해야 UCS2로 정상 변환된다.

- 또는 String 객체를 만들 때 인코딩할 문자집합을 지정하라.

```
String str = new String(buf, 0, count, "UTF-8");

String str = new String(buf, 0, count, "CP949");   (MS949)
```

한글이 깨지는 이유

```
FileInputStream in = new FileInputStream("sample/ms949.txt"); // 41 42 b0 a1 b0 a2(AB가각)

byte[] buf = new byte[1000];
int count = in.read(buf);

String str = new String(buf, 0, count);
```

- 결과
  1. 원래 파일은 MS949로 인코딩 되어 파일에 저장되었다.

     ```
     ms949.txt   =>   41 42 b0 a1 b0 a2   (AB가각)

     =  01000001 01000010 10110000 10100001 10110000 10100010
     ```

  2. read(byte[ ])로 읽어서 byte[ ]에 저장할 때는 그대로 읽어온다.
  3. 이 byte[ ]로 String을 만들 때 내부의 문자를 UTF-16으로 변환해야 한다. (JVM은 UTF-16 사용)
  4. UTF-16으로 변환할 때 인코딩 될 때 사용된 문자집합을 알아야 한다.
  5. 근데 직접 파일 인코딩을 알려주지 않고 이클립스에서 실행하면
  6. String 클래스는 UTF-8(이클립스 기본 문자집합)이라고 생각해서
     UTF-8 문자표를 이용하여 UTF-16으로 변환한다. (원래는 MS949인데)

     ```
     byte(UTF-8) => char(UCS2)

     01000001  -> 00000000 01000001 (00 41) = 'A' <-- 정상적으로 변환되었음.
     01000010  -> 00000000 01000010 (00 42) = 'B' <-- 정상적으로 변환되었음.
     10110000  -> 꽝(xx xx) <- 해당 바이트가 UTF-8 코드 값이 아니기 때문에 UCS2로 변환할 수 없다
     10100001  -> 꽝(xx xx) <- 그래서 꽝을 의미하는 특정 코드 값이 들어 갈 것이다.
     10110000  -> 꽝(xx xx) <- 그 코드 값을  문자로 출력하면 => ?
     10100010  -> 꽝(xx xx)
     ```

  7. 글자가 깨진다.

     - 만약, 이 경우에 Windows 환경의 콘솔창에서 파일 인코딩을 알려주지 않는다면
       글자가 깨지지 않는다. (OS의 기본 문자집합인 MS949 문자표를 이용하므로)

- JVM 환경 변수 'file.encoding'에 설정된 문자표에 상관없이
  String 객체를 만들 때 바이트 배열의 인코딩 문자 집합을 정확하게 알려주는 것이 좋다 !!

- **MS949(= CP949)면 MS949라고 알려줘야 한다. (알려주면 UTF-16으로 잘 바꾼다.)**

```
FileInputStream in = new FileInputStream("sample/ms949.txt");
byte[] buf = new byte[1000];
int count = in.read(buf);
String str = new String(buf, 0, count, "CP949");

FileInputStream in = new FileInputStream("sample/utf16be.txt"); // 0041 0042 ac00 ac01(AB가각)
String str = new String(buf, 0, count, "UTF-16"); // UTF-16 == UTF-16BE

FileInputStream in = new FileInputStream("sample/utf16le.txt");
String str = new String(buf, 0, count, "UTF-16LE");
```

- **UTF-16LE는 작은 수가 앞에 먼저 오게 바이트 순서가 바뀌는 것**

  UTF-16BE = UTF-16

전체 과정 요약 ★

String 객체를 **출력**하여 **.txt 파일**을 만들고 **.txt 파일**을 **읽어서** 다시 **String 객체**를 만드는 **전체 과정**

1. **String 객체**

```
                                  2 byte                    클래스정보
new String("AB가각");     =>    [0041][0042][AC00][AC01][...]   :  String 인스턴스
                                    └─── UTF-16BE (UCS2)──┘
```

2. **getBytes( )를 호출하여 byte[ ] 배열로 전환**

```
                          getBytes()
new String("AB가각");  ──────────────▶  [41][42][EA][B0][80][EA][B0][81][....]
                                          └──────────  UTF-8  ──────────┘
```

3. **FileOutputStream의 write( )**

```
FileOutputStream out = new FileOutputStream("temp/utf.txt");
out.write(bytes);

utf.txt  =>  [41][42][ea][b0][80][ea][b0][81]  ("AB가각")
             └──────────  UTF-8  ──────────┘
```

4. **저장할 byte[ ] 생성 후 FileInputStream의 read(byte[ ]) 호출**

```
FileInputStream in = new FileInputStream("temp/utf.txt");
byte[] buf = new byte[1000];
int count = in.read(buf);
```

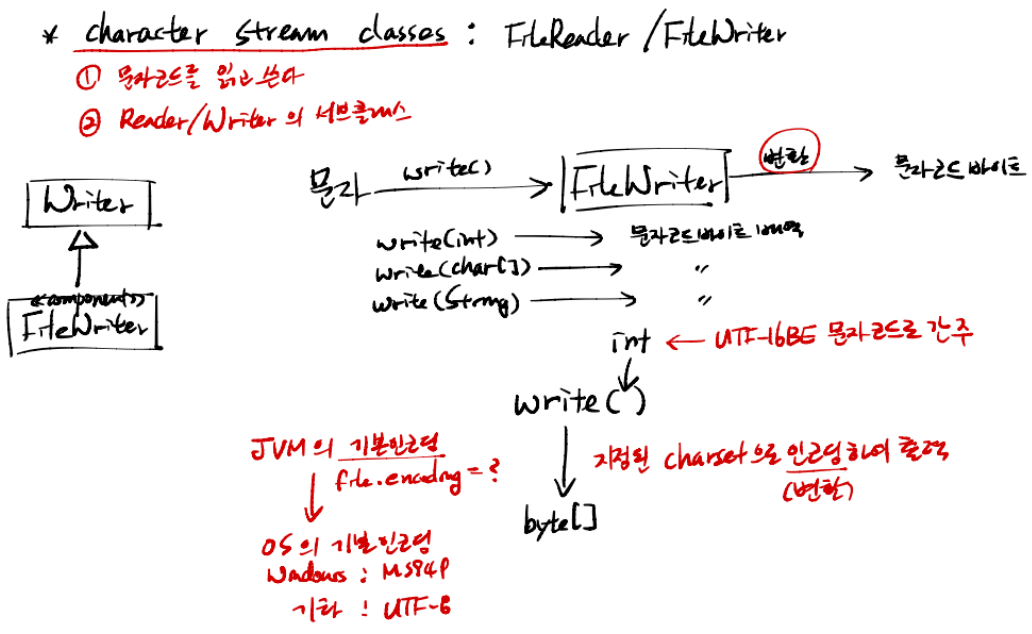
5. **byte[ ]로 String 객체를 만든다.**

   String 생성 시 UTF-8로 직접 설정

```
String str = new String(buf, 0, count, "UTF-8");

UCS2      =>  [0041][0042][ac00][ac01]   <==  정상적으로 바뀐다.  (AB가각)
                  String 인스턴스
```

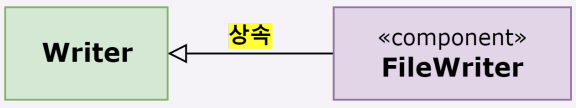
▷ Character Stream (IO-EX03)

※ Character Stream Class : FileReader / FileWriter



Character Stream Class (Java 클래스 설명서 : java.io)

1. 문자 코드를 읽고 쓴다.
2. Reader / Writer의 서브 클래스는 다 character Stream이다



- JVM의 문자열을 파일로 출력할 때, FileOutputStream과 같은 바이트 스트림 클래스를 사용하면 문자집합을 지정해야 하는 번거로움이 있었다.
  ⇒ 이런 번거로움을 해결하기 위해 만든 스트림 클래스가 문자 스트림 클래스이다.

FileWriter 클래스


```
write(int)  ————>  문자 코드 바이트 배열

write(char[])  ————>  문자 코드 바이트 배열

write(String)  ————>  문자 코드 바이트 배열
```

- write(int)
  - FileWriter의 write( )는 파라미터로 받은 int를 UTF-16BE 문자코드로 간주한다.
    (즉, 저장된 바이트가 UTF-16이라고 생각한다)
  - 리턴 : byte[ ] (지정된 character set(기본 인코딩)으로 인코딩(변환)하여 출력)
    (JVM의 기본 인코딩(file.encoding = ?) ⇒ 설정 안되었으면 OS의 기본 인코딩 사용)

문자 출력하기

1. 문자 단위로 출력할 도구 준비

```
FileWriter out = new FileWriter("temp/test2.txt");
```

2. 문자 출력하기

- JVM은 문자 데이터를 다룰 때 UCS2(UTF16BE, 2바이트) 유니코드를 사용

- character stream 클래스인 FileWriter는 문자 데이터를 출력할 때
  UCS2 코드를 JVM 환경변수 file.encoding 에 설정된 character set 코드로 변환하여 출력한다.
  (JVM 실행할 때 출력 데이터의 문자 집합 지정 방법)

3. write(int) 시 앞의 2바이트는 버리고, 뒤의 2바이트(UCS2)를 UTF-8 코드표에 따라
   1 ~ 4 바이트 값으로 변환하여 파일에 쓴다. (이클립스 환경에서 지정 안 한 경우)

- int는 문자(UTF-16)를 4 byte int 값으로 변환한 값이다. 문자를 읽어 UTF-16 형식 그대로
  4 byte 중 뒤의 2 byte에 넣는다.
  ⇒ 따라서 앞의 2 byte는 필요 없기 때문에 write( ) 시 버린다.
  - write(String), write(char[ ])도 마찬가지로 자동으로 int 값으로 변환하여 처리한다.

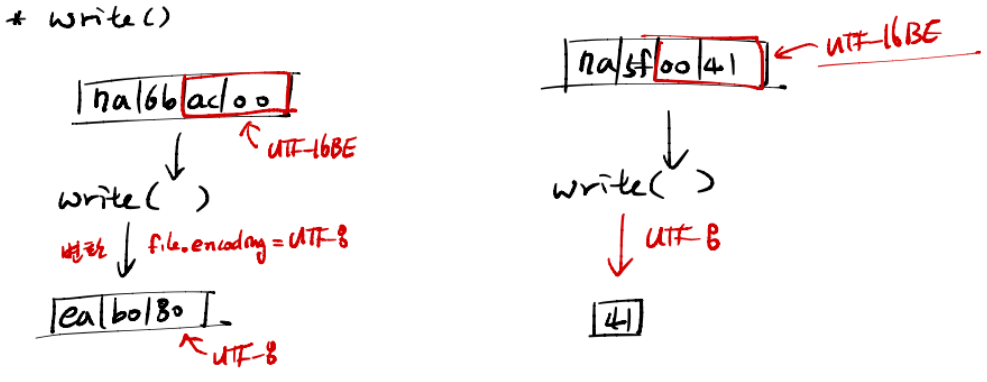
- UCS2에서 한글 '가'는 ac00이다.

```
out.write(0x7a6bac00);

1. 앞의 2바이트(7a6b)는 버린다.
2. 뒤의 2바이트(ac00)   =>   UTF-8 (ea b0 80)로 변환되어 파일에 출력
```

- UCS2에서 영어 'A'는 0041이다.

```
out.write(0x7a5f0041);

1. 앞의 2바이트(7a5f)는 버린다.
2. 뒤의 2바이트(0041)   =>   UTF-8 (41)로 변환되어 파일에 출력
```

※ write( )



- 출력 스트림 객체를 생성할 때 문자 집합을 지정하지 않으면

  UCS2 문자열을 기본 문자집합으로 인코딩 한다. (이클립스 환경에선 UTF-8)

  ```
  FileWriter out = new FileWriter("temp/test2.txt");
  ```

1. write(int)는 4 byte int 값에서 앞의 2 byte는 무시하고

   뒤에 2 byte를 UTF-16BE로 저장된 문자코드라고 생각을 한다.

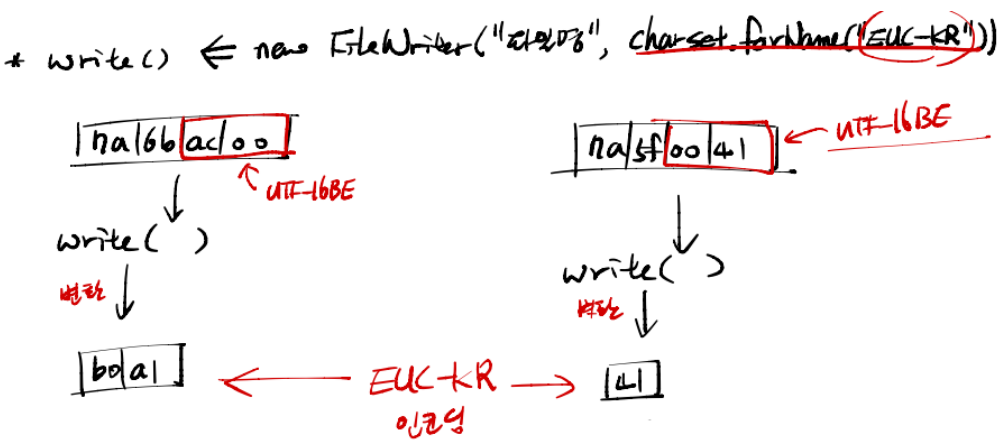
   ```
   [7a][6b][ac][00]       [7a][5f][00][41]

   out.write(0x7a6bac00);
   out.write(0x7a5f0041);
   ```

2. write(int)는 출력할 때 file.encoding = UTF-8(이클립스 실행)이므로

   아래와 같이 UTF-8로 변환해서 출력할 것이다.

   ```
   [ac][00]   ------>   [ea][b0][80]   ('가')

   [00][41]   ------>   [41]   ('A')
   ```

※ write( ) + new FileWriter("파일명", charset.forName("EUC-KR"))



- 출력 스트림 객체를 생성할 때 문자 집합을 지정하면 UCS2 문자열을 해당 문자집합으로 인코딩 한다.

  ```
  Charset charset = Charset.forName("EUC-KR");
  FileWriter out = new FileWriter("temp/test2.txt", charset);
  ```

  - factory method를 통해서 character set 객체 만들기

- **Java에서 EUC-KR로 지정하면 실제로는 MS949로 지정된다. (MS949는 EUC-KR의 확장)**

1. **write( )는 앞의 2 byte는 무시하고 뒤에 2 byte를 UTF-16BE로 저장된 문자코드라고 생각을 할 것이다.**

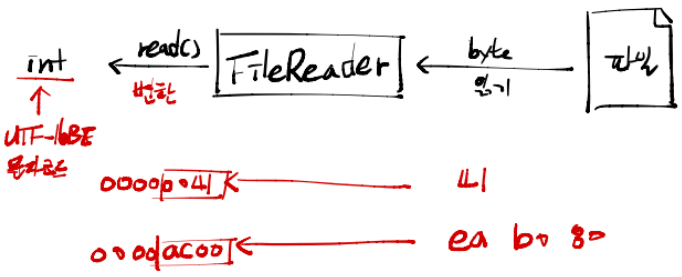
```
[7a][6b][ac][00]       [7a][5f][00][41]
```

2. **write( )는 출력할 때 charset.forName("ECU-KR")으로 지정한 문자집합에 따라**
   아래와 같이 MS949(EUC-KR)로 변환해서 출력할 것이다.

```
[ac][00]   ------>   [b0][a1]   ('가')

[00][41]   ------>   [41]   ('A')
```

※ FileReader 클래스

* FileReader




```
0000 0041   ◀──────────────   41

0000 ac00   ◀──────────────   ea b00 80  (3byte를 읽어서 변환)
```

문자 읽기

1. 파일의 데이터를 읽는 일을 하는 객체를 준비

```
FileReader in = new FileReader("sample/utf8.txt");
41 42 ea b0 81 ea b0 81  (UTF-8)
```

2. 문자 읽기

- JVM을 실행할 때 JVM 환경 변수 'file.encoding' 옵션을 지정하지 않으면
  JVM은 OS의 기본 문자표라고 가정하고 파일을 읽는다. (OS에 따라 다르게)

- 만약 이 때 이클립스에서 실행한다면, UTF-8이라고 가정하고 읽고
  UTF-8 문자표(file.encoding=UTF-8)를 바탕으로 UCS2로 변환하여 리턴한다.
  - 한글은 3 바이트를 읽어서 2바이트 UCS2로 변환한다.

```
int ch3 = in.read(); // ea b0 80 => ac00('가')
int ch4 = in.read(); // ea b0 81 => ac01('각')
```

  - 영어은 1 바이트를 읽어서 2바이트 UCS2로 변환한다.

```
int ch1 = in.read(); // 41 => 0041('A')
int ch2 = in.read(); // 42 => 0042('B')
```

- 출력 스트림 객체를 생성할 때 파일의 문자 집합을 지정하면
  JVM 환경 변수 'file.encoding'에 설정된 값은 무시한다.

- MS949.txt 파일 읽기 (이클립스 환경)
  - 출력 스트림 객체를 생성할 때 파일의 문자 집합을 지정하지 않으면 (UTF-8) ⇒ 글자 깨짐 !

```
FileReader in = new FileReader("sample/ms949.txt");

41 42 b0 a1 b0 a2
```

<영어>

```
int ch1 = in.read();      41   =>   0041('A')
int ch2 = in.read();      42   =>   0042('B')
```

- MS949도 영어인 경우 UTF-8과 같이 1바이트를 읽어서 2바이트 UCS2로 변환한다.

<한글>

```
int ch3 = in.read();      b0 a1   =>   fffd => ?
int ch4 = in.read();      b0 a2   =>   fffd => ?
```

- 잘못되었다는 의미로 특정 값(fffd)으로 변환

  - 출력 스트림 객체를 생성할 때 파일의 문자 집합을 지정 (MS949)

```
FileReader in = new FileReader("sample/ms949.txt", Charset.forName("MS949"));

41 42 b0 a1 b0 a2
```

<영어>

```
int ch1 = in.read();      41   =>   0041('A')
int ch2 = in.read();      42   =>   0042('B')
```

- **MS949 문자표에 따라 UCS2로 변환시킨다.**

<한글>

```
int ch3 = in.read();      b0 a1   =>   ac00 => '가'
int ch4 = in.read();      b0 a2   =>   ac01 => '각'
```

- **MS949 문자표에 따라 UCS2로 변환시킨다.**

※ char[ ] 문자 배열 출력하기


```
FileWriter out = new FileWriter("temp/test2.txt");

char[]   =   {'A', 'B', 'C', '0', '1', '2', '가', '각', '간', '똘', '똥'}
```

- **char[ ]의 코드가 위와 같을 때 ⇒ 메모리에는 아래와 같이 저장된다.**

```
[0041][0042][0043][0030][0031][0032][ac00][ac01][ac04][b618][b625]
 └──────────────────────── UTF-16BE (UCS2)──────────────────────────┘
```

  - **char[ ]은 UTF-16이다. JVM은 무조건 문자를 다룰 땐 UTF-16으로 다룬다.**

- **FileWriter.write(char[ ]) 하면 UTF-8로 변환하여 출력 (이클립스 환경에서 문자집합 지정 안 한 경우)**

```
FileWriter.write(chars)
```

```
[41][42][43][30][31][32][ea][b0][80][ea][b0][81][ea][b0][84][ea][98][98][ea][98][a5]
 └──────────────────────────────────────── UTF-8 ──────────────────────────────────────┘
```

- UTF-8 : 영어는 1바이트로 변환되어 출력 / 한글은 3바이트로 변환되어 출력

```
JVM(UCS2)         File(UTF-8)
00 41      ==>    41
00 30      ==>    30
ac 00      ==>    ea b0 80
b6 18      ==>    eb 98 98
b6 25      ==>    eb 98 a5
```

- 문자 배열의 특정 부분을 출력하기

```
FileWriter out = new FileWriter("temp/test2.txt");
char[] chars = new char[] {'A','B','C','가','각','간','똘','똥'};

out.write(chars, 2, 3);
```

- 2번 문자부터 3 개의 문자를 출력한다.

- String 출력하기

```
FileWriter out = new FileWriter("temp/test2.txt");
String str = new String("AB가각");

out.write(str);
```

- FileWriter 객체 생성 시 문자 집합 지정을 안 했으므로 UTF-8로 출력 (이클립스)

데이터를 읽어 char[ ] 배열에 저장하고 String 만들기

- 데이터 읽기
  - UCS2 문자 코드 값을 저장할 배열을 준비한다.

    이렇게 임시 데이터를 저장하기 위해 만든 배열을 보통 "버퍼(buffer)"라 한다.

```
FileReader in = new FileReader("temp/test2.txt");

char[] buf = new char[100];
```

  - read(버퍼의 주소)
    - 버퍼가 꽉 찰 때까지 읽는다. (버퍼 보다 작으면 모두 읽는다)
    - 리턴 값은 읽은 문자의 개수이다. (바이트의 개수가 아니다!!!!!)

      ※ FileInputStream.read( )의 리턴 값은 읽은 바이트의 개수였다.

    - 파일을 읽을 때 JVM 환경 변수 'file.encoding'에 설정된 문자코드표에 따라 바이트를 읽는다.

      그리고 2바이트 UCS2 코드 값으로 변환하여 리턴한다.

- UTF-8이라면 파일을 읽을 때,

  영어나 숫자, 특수기호 ⇒ 1바이트를 읽어 UCS2으로 변환

  한글 ⇒ 3바이트를 읽어 UCS2으로 변환

- 읽은 데이터를 문자 배열의 특정 위치에 저장하기
  - read(버퍼의주소, 저장할위치, 읽을바이트개수) ⇒ 리턴 값은 실제 읽은 문자의 개수이다.

```
FileReader in = new FileReader("temp/test2.txt");
char[] buf = new char[100];

int count = in.read(buf, 10, 40);
```

  - 40개의 문자를 읽어 10번 방부터 저장한다.

- char[ ] ⇒ String 변환하기

```
FileReader in = new FileReader("temp/test2.txt");
char[] buf = new char[100];

int count = in.read(buf);
String str = new String(buf, 0, count);
```

  - char 배열에 담을 때 UTF-16BE 코드 값으로 변환한다.
  - 그래서 String 객체를 만들 때 문자집합을 지정할 필요가 없다.

- java.nio의 CharBuffer를 사용

```
import java.nio.CharBuffer;
FileReader in = new FileReader("temp/test2.txt");
```

  - FileReader 객체가 읽을 데이터를 저장할 메모리를 준비하고

    읽은 데이터를 CharBuffer에 저장한다.

```
CharBuffer charBuf = CharBuffer.allocate(100);
int count = in.read(charBuf);
in.close();
```

  - flip( ) 메서드

```
charBuf.flip();
System.out.printf("[%s]\n", charBuf.toString());
```

    - 버퍼의 데이터를 꺼내기 전에 읽은 위치를 0으로 초기화시킨다.
    - read( ) 메서드가 파일에서 데이터를 읽어서 버퍼에 채울 때 마다 커서의 위치는 다음으로 이동한다.
    - 버퍼의 데이터를 읽으려면 커서의 위치를 처음으로 되돌려야 한다. (flip)
    - flip() 메서드를 호출하여 커서를 처음으로 옮긴다. 그런 후에 버퍼의 텍스트를 읽어야 한다.

- **Decorator 붙이기**

```
FileReader in = new FileReader("temp/test2.txt");

BufferedReader in2 = new BufferedReader(in);

System.out.println(in2.readLine());
```

  - **기존의 FileReader에 Decorator인 BufferedReader를 붙이면,**

    ⇒ 버퍼 기능 + 한 줄 읽기 기능을 사용할 수 있다. (Decorator로 기능 추가가 자유롭다)

▷ 데이터 입/출력 (IO-EX04)

데이터 출력 / 읽기 - int 값

- **int 값 출력**

```
FileOutputStream out = new FileOutputStream("temp/test3.data");

int money = 1_3456_7890;      0x080557d2
```

  - **int 메모리의 모든 바이트를 출력하려면, 각 바이트를 맨 끝으로 이동한 후 write( )로 출력한다.**

    ⇒ FileOutputStream의 write( )는 항상 변수의 마지막 1바이트만 출력하기 때문이다.

```
out.write(money >> 24);    00000008|0557d2
out.write(money >> 16);    00000805|57d2
out.write(money >> 8);     00080557|d2
out.write(money);          080557d2
```

- **int 값 읽기**

  - **read( )는 1바이트를 읽어 int 값으로 만든 후 리턴한다.**

    (read( ) ⇒ 실제 리턴한 값 = 0x000000D2)

```
FileInputStream in = new FileInputStream("temp/test3.data");   080557d2
```

  - **파일에서 4바이트를 읽어 4바이트 int 변수에 저장할 때,**

    읽은 바이트를 비트이동 연산자를 값을 이동 시킨 후 변수에 저장해야 한다.

```
int value = in.read() << 24;      00000008 =>   08000000
value += (in.read() << 16);       00000005 => + 00050000
```

```
    value += (in.read() << 8);        00000057 => + 00005700
    value += in.read();               000000d2 => + 000000d2
================================================>   080557d2
```

데이터 출력 / 읽기 - long 값

- **long 값 출력**

```
FileOutputStream out = new FileOutputStream("temp/test3.data");

long money = 400_0000_0000_0000L;     0x00016bcc|41e90000
```

  - **long 메모리의 모든 바이트를 출력하려면, 각 바이트를 맨 끝으로 이동한 후 write( )로 출력한다.**

    ⇒ FileOutputStream의 write( )는 항상 변수의 마지막 1바이트만 출력하기 때문이다.

```
out.write((int)(money >> 56));   // 00000000|00000000|016bcc41e90000
out.write((int)(money >> 48));   // 00000000|00000001|6bcc41e90000
out.write((int)(money >> 40));   // 00000000|0000016b|cc41e90000
out.write((int)(money >> 32));   // 00000000|00016bcc|41e90000
out.write((int)(money >> 24));   // 00000000|016bcc41|e90000
out.write((int)(money >> 16));   // 00000001|6bcc41e9|0000
out.write((int)(money >> 8));    // 0000016b|cc41e900|00
out.write((int)money);           // 00016bcc|41e90000|
```

- **long 값 읽기**

```
FileInputStream in = new FileInputStream("temp/test3.data");  00016bcc41e90000
```

  - **파일에서 8바이트를 읽어 8바이트 long 변수에 저장할 때,**

    읽은 바이트를 비트이동 연산자를 값을 이동 시킨 후 변수에 저장해야 한다.

```
long value = (long)in.read() << 56;
value += (long)in.read() << 48;
value += (long)in.read() << 40;
value += (long)in.read() << 32;
value += (long)in.read() << 24;
value += (long)in.read() << 16;
value += (long)in.read() << 8;
value += in.read();
```

데이터 출력 / 읽기 - String 값

- **String 값 출력**

```
FileOutputStream out = new FileOutputStream("temp/test3.data");

String str = "AB가각간";

out.write(str.getBytes("UTF-8"));
```

- **str.getBytes(문자코드표)**

  문자열을 지정한 문자코드표에 따라 인코딩하여 바이트 배열을 만든다.

- **String 값 읽기**

```
FileInputStream in = new FileInputStream("temp/test3.data");

byte[] buf = new byte[100];

int count = in.read(buf);

String str = new String(buf, 0, count, "UTF-8"); UTF-8이고 UTF-16으로 바꿔라!!!??
```

  - **바이트 배열에 읽어 들이고 바이트 배열에 들어있는 값을 사용하여 String 인스턴스를 만든다.**

```
new String(바이트배열, 시작번호, 개수, 문자코드표)

예) new String(buf, 0, 10, "UTF-8");
```

  ⇒ **"String 객체야 바이트 배열을 UTF-8 문자표를 이용해 UTF-16으로 바꿔서 저장해라"**

데이터 출력 / 읽기 - float, double 값

- **float, double 값 출력**

  - **비트 이동 연산자를 쓸 수 없다.**

    ⇒ **Float, Double 클래스의 메서드를 사용하면 정수인양 비트 이동이 가능하다.**

```
FileOutputStream out = new FileOutputStream("temp/test3.data");

float f = 12.375f; // hex: 41460000
double d = 12.375; // hex: 4028c00000000000

f 출력
out.write(Float.floatToIntBits(f) >> 24);
out.write(Float.floatToIntBits(f) >> 16);
out.write(Float.floatToIntBits(f) >> 8);
out.write(Float.floatToIntBits(f));


d 출력
out.write((int)(Double.doubleToLongBits(d) >> 56));
out.write((int)(Double.doubleToLongBits(d) >> 48));
out.write((int)(Double.doubleToLongBits(d) >> 40));
out.write((int)(Double.doubleToLongBits(d) >> 32));
out.write((int)(Double.doubleToLongBits(d) >> 24));
out.write((int)(Double.doubleToLongBits(d) >> 16));
```

```
out.write((int)(Double.doubleToLongBits(d) >> 8));
out.write((int)(Double.doubleToLongBits(d)));
```

- **float, double 값 읽기**
  - **float 값에 해당하는 바이트를 읽어서 int 메모리에 담는다. (int 인 것 처럼)**

    ⇒ 이 int 메모리를 Float 클래스의 메서드로 float 값으로 리턴한다.
  - **double 값에 해당하는 바이트를 읽어서 long 메모리에 담는다. (long 인 것 처럼)**

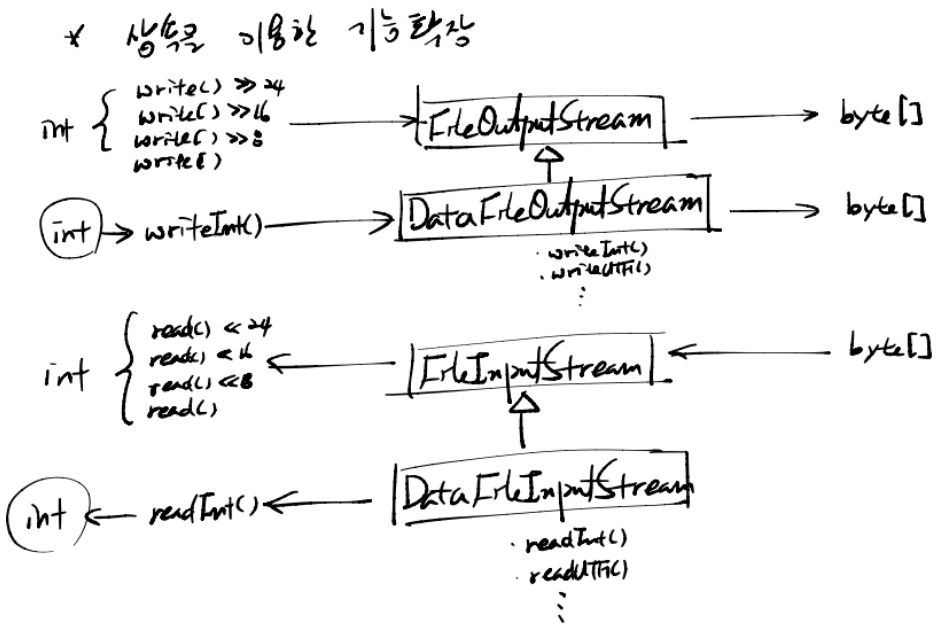
    ⇒ 이 long 메모리를 Double 클래스의 메서드로 double 값으로 리턴한다.

```
FileInputStream in = new FileInputStream("test6.data");

float 값에 해당하는 바이트 읽기
int temp = (in.read() << 24)
     + (in.read() << 16)
     + (in.read() << 8)
     + in.read();

int 변수에 저장된 것을 float 변수에 담기
float f = Float.intBitsToFloat(temp);
System.out.printf("%f\n", f);


double 값에 해당하는 바이트 읽기
long temp2 = (((long) in.read()) << 56)
     + (((long) in.read()) << 48)
     + (((long) in.read()) << 40)
     + (((long) in.read()) << 32)
     + (((long) in.read()) << 24)
     + (((long) in.read()) << 16)
     + (((long) in.read()) << 8)
     + in.read();

long 변수에 저장된 것을 double 변수에 담기
double d = Double.longBitsToDouble(temp2);
System.out.printf("%f\n", d);
```

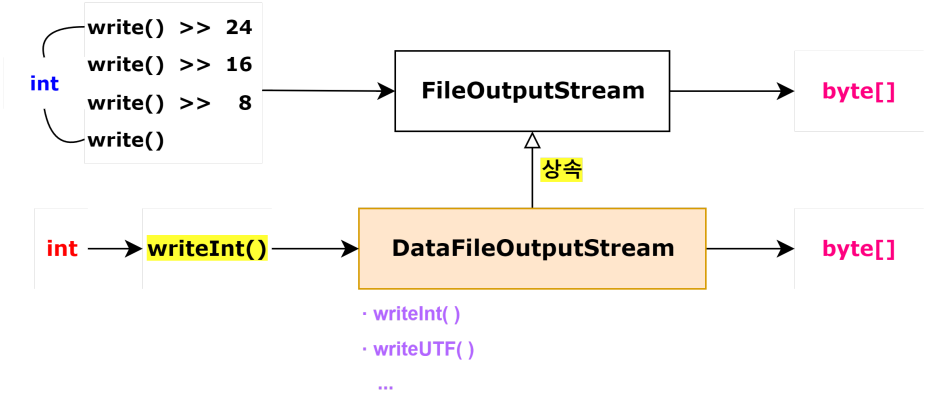
Decorator 패턴 설계 과정 - IO-EX05-10

▷ 객체 입/출력 (IO-EX05)

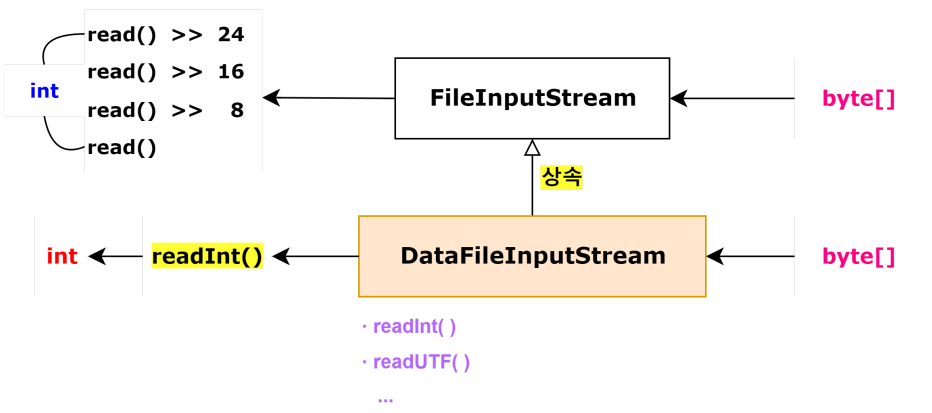
※ 상속을 이용한 기능 확장



- **출력**



- **입력**



FileOutputStream / FileInputStream

- 인스턴스의 값을 출력

```
FileOutputStream out = new FileOutputStream("temp/test4.data");

Member member = new Member();
member.name = "AB가각간";
member.age = 27;
member.gender = true;
```

1. **이름 출력 (String)**

```
byte[] bytes = member.name.getBytes("UTF-8");
out.write(bytes.length); // 1 바이트
out.write(bytes); // 문자열 바이트
```

2. **나이 출력 (int : 4바이트)**

```
out.write(member.age >> 24);
out.write(member.age >> 16);
out.write(member.age >> 8);
out.write(member.age);
```

3. **성별 출력 (boolean : 1바이트)**

```
if (member.gender)
  out.write(1);
else
  out.write(0);
```

- **인스턴스의 값을 읽기**

```
FileInputStream in = new FileInputStream("temp/test4.data");

Member member = null;
member = new Member();
```

1. **이름 읽기 (String)**

```
int size = in.read(); // 이름이 저장된 바이트 배열의 수
    byte[] buf = new byte[size];
    in.read(buf); // 이름 배열 개수 만큼 바이트를 읽어 배열에 저장한다.
    member.name = new String(buf, "UTF-8");
```

2. **나이 읽기 (int : 4바이트)**

```
member.age = in.read() << 24;
    member.age += in.read() << 16;
    member.age += in.read() << 8;
    member.age += in.read();
```

3. **성별 읽기 (boolean : 1바이트)**

```
if (in.read() == 1)
      member.gender = true;
    else
      member.gender = false;
```

DataFileOutputStream / DataFileInputStream

- 인스턴스의 값을 출력

```
DataFileOutputStream out = new DataFileOutputStream("temp/test4_2.data");

Member member = new Member();
member.name = "AB가각간";
member.age = 27;
member.gender = true;
```

1. 이름 출력 (String)

```
out.writeUTF(member.name);
```

2. 나이 출력 (int : 4바이트)

```
out.writeInt(member.age);
```

3. 성별 출력 (boolean : 1바이트)

```
out.writeBoolean(member.gender);
```

- 인스턴스의 값을 읽기

```
DataFileInputStream in = new DataFileInputStream("temp/test4_2.data");

Member member = null;

member = new Member();
```

1. 이름 읽기 (String)

```
member.name = in.readUTF();
```

2. 나이 읽기 (int : 4바이트)

```
member.age = in.readInt();
```

3. 성별 읽기 (boolean : 1바이트)

```
member.gender = in.readBoolean();
```