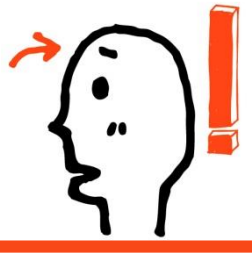
Android Java

23장 파일과 I/O 스트림

파일과 I/O 스트림



01 File I/O에 대한 소개

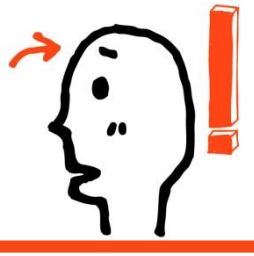
02 필터 스트림의 이해와 활용

03 문자 스트림의 이해와 활용

04 스트림을 통한 인스턴스의 저장

05 Random access 파일과 file 클래스

I/O의 범위와 간단한 I/O 모델의 소개



일반적인 입출력의 대상

- 키보드와 모니터
- 하드디스크에 저장되어 있는 파일
- USB와 같인 외부 메모리 장치
- 네트워크로 연결되어 있는 컴퓨터
- 사운드카드, 오디오카드와 같은 멀티미디어 장치
- 프린터, 팩시밀리와 같은 출력장치

입출력 대상이 달라지면 프로그램상에서의 입출력 방식도 달라지는 것이 보통이다. 그런데 자바에서는 입출력 대상에 상관없이 입출력의 진행 방식이 동일하도록 별도의 'I/O 모델'을 정의하고 있다.

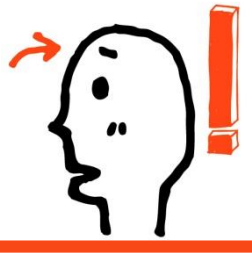
I/O 모델의 정의로 인해서 입출력 대상의 차이에 따른 입출력 방식의 차이는 크지 않다. 기본적인 입출력의 형태는 동일하다. 그리고 이것이 JAVA의 I/O 스트림이 갖는 장점이다.

자바 스트림의 큰 분류

• 입력 스트림(Input Stream)	프로그램으로 데이터를 읽어 들이는 스트림
• 출력 스트림(Output Stream)	프로그램으로부터 데이터를 내보내는 스트림

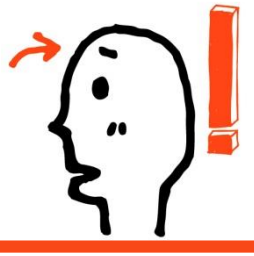
데이터의 입력을 위해서는 입력 스트림을, 출력을 위해서는 출력 스트림을 형성해야 한다. 그리고 여기서 말하는 스트림이라는 것도 인스턴스의 생성을 통해서 형성된다.

스트림



- 자바의 스트림
  - 자바 스트림은 입출력 장치와 자바 응용 프로그램 연결
  - 입출력 장치와 프로그램 사이의 데이타 흐름을 처리하는 소프트웨어 모듈
  - 입력 스트림
    - 입력 장치로부터 자바 프로그램으로 데이터를 전달하는 소프트웨어 모듈
  - 출력 스트림
    - 자바 프로그램에서 출력 장치로 데이터를 보내는 소프트웨어 모듈
- 입출력 스트림 기본 단위 : 바이트
- 자바 입출력 스트림 특징
  - 단방향 스트림, 선입선출 구조

파일 기반의 입력 스트림 형성

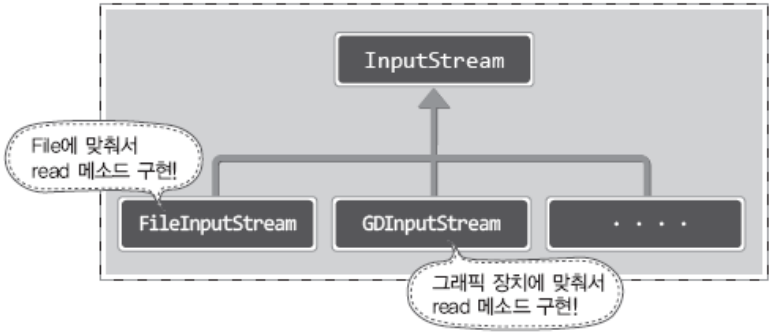


파일 run.exe 대상의 입력 스트림 생성

```
InputStream in=new FileInputStream("run.exe");
```

- 스트림의 생성은 결국 인스턴스의 생성.
- FileInputStream 클래스는 InputStream 클래스를 상속한다.

InputStream 클래스는 모든 입력 스트림 클래스의 최상위 클래스



InputStream 클래스의 대표적인 두 메소드

- public abstract int read() throws IOException
- public void close() throws IOException

이렇듯 입력의 대상에 적절하게 read 메소드가 정의되어 있다. 그리고 입력의 대상에 따라서 입력 스트림을 의미하는 별도의 클래스가 정의되어 있다.

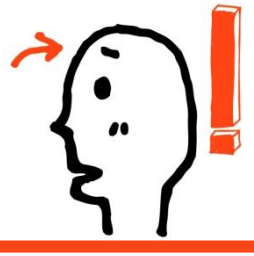
파일 대상의 입력 스트림 생성

```
InputStream in=new FileInputStream("run.exe");
int bData=in.read();    // 오버라이딩에 의해 FileInputStream의 read 메소드 호출!
```

그래픽 디바이스 대상의 입력 스트림 생성(가상의 코드)

```
InputStream in=new GDInputStream(0x2046);   // 0x2046가 그래픽 장치의 할당 주소라 가정!
int bData=in.read();    // 오버라이딩에 의해 GDInputStream의 read 메소드 호출!
```

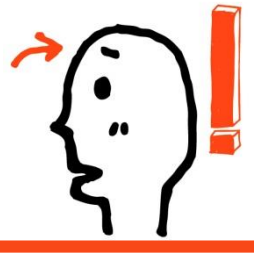
파일 기반의 입력 스트림 형성



- InputStream 클래스의 주요 메소드

리턴타입	메소드	설명
int	read()	입력 스트림으로부터 1 바이트를 읽고 읽은 바이트를 리턴한다.
int	read(byte[] b)	입력 스트림으로부터 읽은 바이트들을 매개값으로 주어진 바이트 배열 b 에 저장하고 실제로 읽은 바이트 수를 리턴한다.
int	read(byte[] b, int off, int len)	입력 스트림으로부터 len 개의 바이트 만큼 읽고 매개값으로 주어진 바이트 배열 b[off] 부터 len 개까지 저장한다. 그리고 실제로 읽은 바이트 수인 len 개를 리턴한다. 만약 len 개를 모두 읽지 못하면 실제로 읽은 바이트 수를 리턴한다.
void	close()	사용한 시스템 자원을 반납하고 입력 스트림을 닫는다.

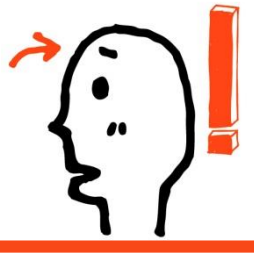
파일 기반의 출력 스트림 형성



- OutputStream 클래스의 주요 메소드

리턴타입	메소드	설명
void	wrtie(int b)	출력 스트림으로 1 바이트를 보낸다.
void	write(byte[] b)	출력 스트림에 매개값으로 주어진 바이트 배열 b 의 모든 바이트를 보낸다.
void	write(byte[] b, int off, int len)	출력 스트림에 매개값으로 주어진 바이트 배열 b[off] 부터 len 개까지의 바이트를 보낸다.
void	flush()	버퍼에 잔류하는 모든 바이트를 출력한다.
void	close()	사용한 시스템 자원을 반납하고 출력 스트림을 닫는다.

파일 대상의 출력 스트림 형성



InputStream	↔	OutputStream
FileInputStream	↔	FileOutputStream

입출력 스트림은 대부분 쌍(Pair)을 이룬다.

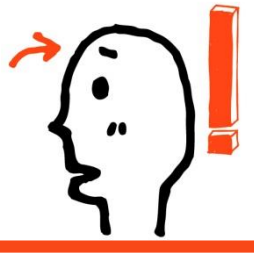
OutputStream 클래스의 대표적인 메소드

- public abstract void write(int b) throws IOException
- public void close() throws IOException

```
OutputStream out=new FileOutputStream("home.bin");
out.write(1);    // 4바이트 int형 정수 1의 하위 1바이트만 전달된다.
out.write(2);    // 4바이트 int형 정수 2의 하위 1바이트만 전달된다.
out.close;       // 입력 스트림 소멸
```

파일 대상의 출력 스트림 생성 및 데이터 전송

스트림 기반의 파일 입출력 예제



ByteFileCopy.java

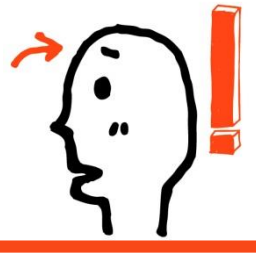
```
public static void main(String[] args) throws IOException
{
    InputStream in=new FileInputStream("org.bin");
    OutputStream out=new FileOutputStream("cpy.bin");

    int copyByte=0;
    int bData;

    while(true)
    {
        bData=in.read();
        if(bData==-1)
            break;

        out.write(bData);
        copyByte++;
    }

    in.close();
    out.close();
    System.out.println("복사된 바이트 크기 "+ copyByte);
}
```

실행 결과

```
복사된 바이트 크기 85528870
```

위 예제에서는 바이트 단위 복사(1바이트 씩 복사)가 진행된다. 따라서 50MB가 넘는 크기의 파일 복사에는 오랜시간이 걸린다.

보다 빠른 속도의 파일복사 프로그램



```
public int read(byte[] b) throws IOException
public void write(byte[] b, int off, int len) throws IOException
```

바이트 단위 read & write 메소드를 대신해서 바이트 배열 단위의 다음 두 메소드를 호출하는 것이 핵심

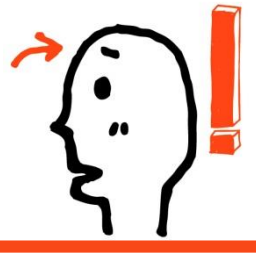
BufferFileCopy.java

```
public static void main(String[] args) throws IOException
{
    InputStream in=new FileInputStream("org.bin");
    OutputStream out=new FileOutputStream("cpy.bin");

    int copyByte=0;
    int readLen;
    byte buf[]=new byte[1024];

    while(true)
    {
        readLen=in.read(buf);
        if(readLen==-1)
            break;
        out.write(buf, 0, readLen);
        copyByte+=readLen;
    }
    in.close();
    out.close();
    System.out.println("복사된 바이트 크기 "+ copyByte);
}
```

이전 예제와의 가장 큰 차이점은 1KB 크기의 버퍼를 이용해서 데이터를 입출력한다는 점이다.

실제로 속도의 향상을 느낄 수 있다.

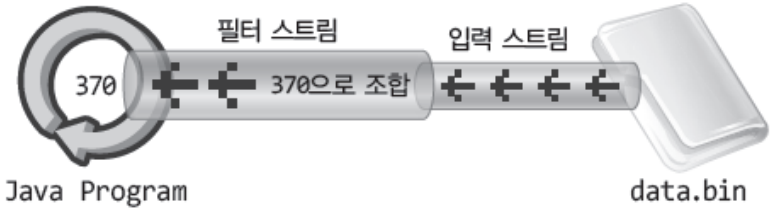
바이트 단위로 데이터를 읽고 쓸 줄은 알지만



```
InputStream is=new FileInputStream("yourAsking.bin");
byte[] buf=new byte[4];
is.read(buf);
. . . .
```

위의 코드는 파일로부터 4바이트를 읽어 들인다. 그러나 byte 단위의 배열 형태로만 읽어 들일 수 있다. 즉, 파일에 4바이트 크기의 int형 정수가 저장되어 있을 때, 이를 정수의 형태로 꺼내는 것은 불가능하다. 이를 위해서는 byte 단위로 4개의 바이트를 읽어 들인 다음에 이를 int형 데이터로 조합해야 한다.

이러한 조합을 중간에서 대한 해 주는 스트림을 가리켜 필터 스트림이라 한다.

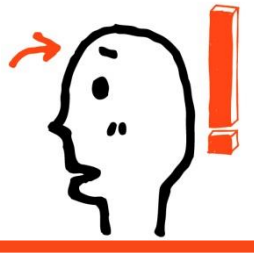


필터 스트림은 물이 뿜어져 나오는 호스에 연결된 샤워기 꼭지에 비유할 수 있다.

- 필터 입력 스트림 — 입력 스트림에 연결하는 필터 스트림
- 필터 출력 스트림 — 출력 스트림에 연결하는 필터 스트림

필터 스트림도 입력용과 출력용이 구분된다.

기본자료형 단위의 데이터 입출력



```
public static void main(String[] args) throws IOException
{
    OutputStream out=new FileOutputStream("data.bin");
    DataOutputStream filterOut=new DataOutputStream(out);
    filterOut.writeInt(275);
    filterOut.writeDouble(45.79);
    filterOut.close();

    InputStream in=new FileInputStream("data.bin");
    DataInputStream filterIn=new DataInputStream(in);
    int num1=filterIn.readInt();
    double num2=filterIn.readDouble();
    filterIn.close();

    System.out.println(num1);
    System.out.println(num2);
}
```

DataOutputStream 클래스와 DataInputStream 클래스는 기본 자료형 단위의 데이터 입출력을 가능하게 하는 필터 스트림이다.

실행결과

```
275
45.79
```

OutputStream out=new FileOutputStream("data.bin");
DataOutputStream filterOut=new DataOutputStream(out);

출력 스트림과 필터 스트림과의 연결!

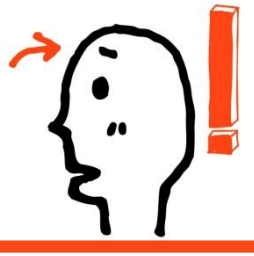
InputStream in=new FileInputStream("data.bin");
DataInputStream filterIn=new DataInputStream(in);

입력 스트림과 필터 스트림과의 연결!

- 필터 입력 스트림 클래스 — FilterInputStream 클래스를 상속한다.
- 필터 출력 스트림 클래스 — FilterOutputStream 클래스를 상속한다.

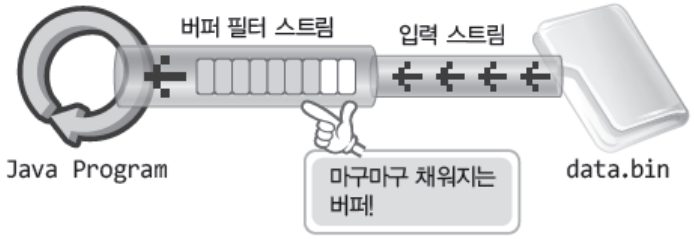
필터 입출력 스트림의 특징, 이를 통해서 필터 스트림 클래스의 구분이 가능

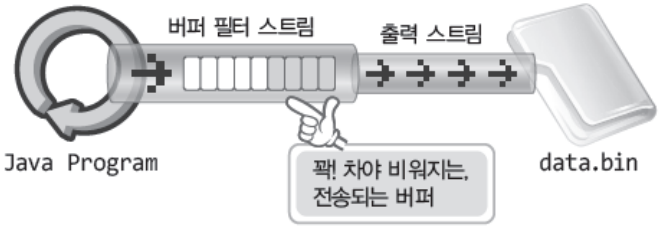
버퍼링 기능을 제공하는 필터 스트림



```
public static void main(String[] args) throws IOException
{
    InputStream in=new FileInputStream("org.bin");
    OutputStream out=new FileOutputStream("cpy.bin");
    BufferedInputStream bin=new BufferedInputStream(in);
    BufferedOutputStream bout=new BufferedOutputStream(out);
    int copyByte=0;
    int bData;
    while(true)
    {
        bData=bin.read();
        if(bData==-1)
            break;
        bout.write(bData);
        copyByte++;
    }
    bin.close();
    bout.close();
    System.out.println("복사된 바이트 크기 "+ copyByte);
}
```

버퍼링 되므로,
read, write 함수의 호출이 빠르게 진행된다.



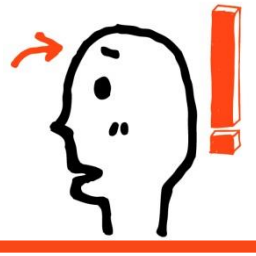


- BufferedInputStream 버퍼 필터 입력 스트림
- BufferedOutputStream 버퍼 필터 출력 스트림

BufferedInputStream은 입력버퍼,
BufferedOutputStream은 출력버퍼 제공!

BufferedOutputStream 클래스의 flush 메소드 호출을 통해서 버퍼링 된 데이터의 목적지 전송이 가능하다! 또한 close 메소드를 통해서 스트림을 종료하면, 스트림의 버퍼는 flush! 된다.

파일에 double형 데이터 저장 + 버퍼링



기본 자료형 데이터 입출력 스트림

```
OutputStream out=new FileOutputStream("data.bin");
DataOutputStream filterOut=new DataOutputStream(out);
```

버퍼 스트림

```
OutputStream out=new FileOutputStream("data.bin");
BufferedOutputStream filterOut=new BufferedOutputStream(out);
```

기본 자료형 데이터 입출력 + 버퍼 스트림

```
OutputStream out=new FileOutputStream("data.bin");
BufferedOutputStream bufFilterOut=new BufferedOutputStream(out);
DataOutputStream dataFilterOut=new DataOutputStream(bufFilterOut);
```

double 데이터의 입출력도 가능하고 버퍼링으로 인해 성능도 개선된다.

Constructor Detail

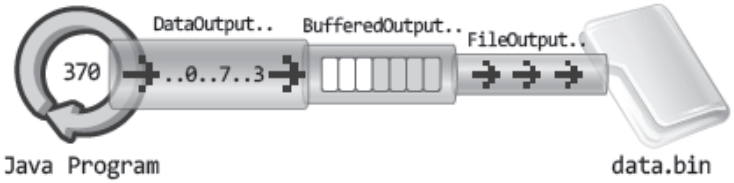
DataOutputStream

public **DataOutputStream**(OutputStream out)

java.io

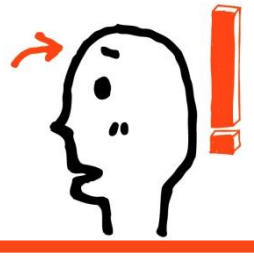
Class BufferedOutputStream

java.lang.Object
  └ java.io.OutputStream
    └ java.io.FilterOutputStream
      └ **java.io.BufferedOutputStream**



생성자와 상속의 관계를 통해서 스트림의 연결 가능성을 확인해야 한다!

파일에 double형 데이터 저장 + 버퍼링



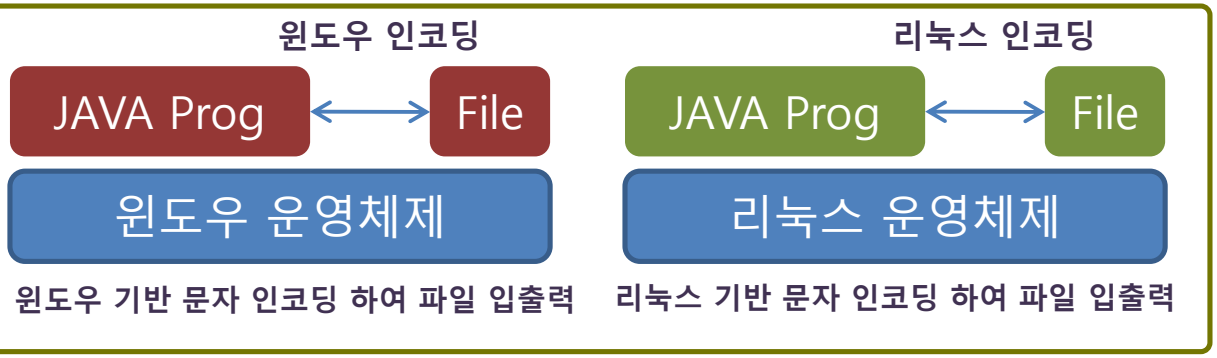
바이트 스트림의 데이터 송수신 특성

바이트 스트림은 데이터를 있는 그대로 송수신 하는 스트림이다. 그리고 이 바이트 스트림을 이용하여 문자를 파일에 저장하는 것도 가능하다. 물론 이렇게 저장된 데이터를 자바 프로그램을 이용해서 읽으면 문제되지 않는다. 하지만 다른 프로그램을 이용해서 읽으면 문제가 될 수 있다.

바이트 스트림을 이용한 파일 대상의 문자 저장의 문제점

운영체제 별로 고유의 문자표현방식이 존재한다. 그리고 운영체제에서 동작하는 프로그램은 해당 운영체제의 문자표현 방식을 따른다. 따라서 파일에 저장된 데이터는 해당 운영체제의 문자표현 방식을 따라서 저장되어 있어야 한다. 해당 운영체제에서 동작하는 다른 프로그램에 의해서 참조가 되는 파일이라면...

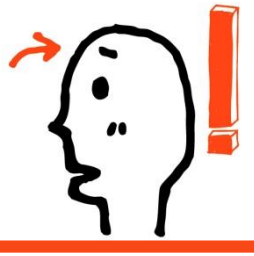
문자 스트림은 해당 운영체제에 따른 인코딩 방식을 지원



문자 스트림은 해당 운영체제의 문자 인코딩 기준을 따라서 데이터 입출력 진행

대부분의 문자 스트림은 바이트 스트림과 1대 1의 대응을 이룬다!

파일에 double형 데이터 저장 + 버퍼링



바이트 스트림		문자 스트림
InputStream, OutputStream	대응	Reader, Writer
FileInputStream, FileOutputStream	대응	FileReader, FileWriter

Reader의 대표적인 메소드

```
public int read() throws IOException
public abstract int read(char[] cbuf, int off, int len) throws IOException
```

Writer의 대표적인 메소드

```
public void write(int c) throws IOException
public abstract void write(char[] cbuf, int off, int len) throws IOException
```

```
public static void main(String[] args) throws IOException
{
    char ch1='A';
    char ch2='B';

    Writer out=new FileWriter("hyper.txt");
    out.write(ch1);
    out.write(ch2);
    out.close();
}
```

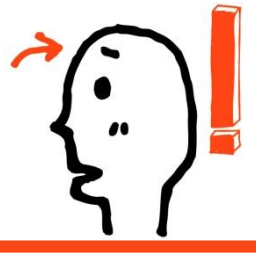
자바에서는 각각 2바이트로 표현됨

실행 운영체제에 따라서 1바이트씩 인코딩 되어서 저장!

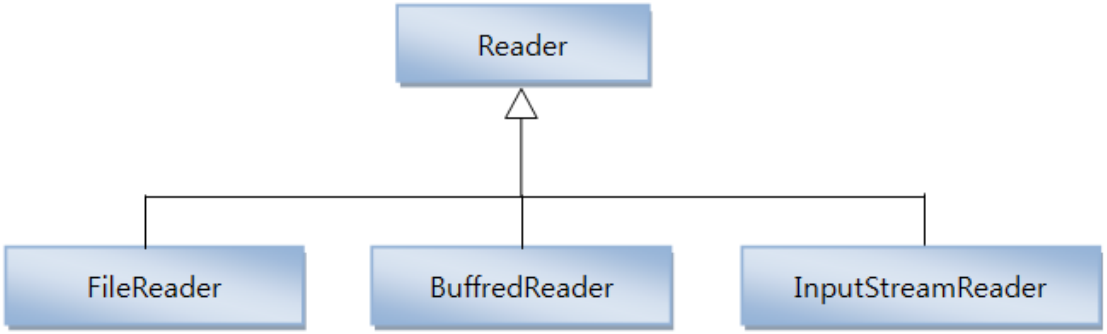
```
public static void main(String[] args) throws IOException
{
    char[] cbuf=new char[10];
    int readCnt;

    Reader in=new FileReader("hyper.txt");
    readCnt=in.read(cbuf, 0, cbuf.length);
    for(int i=0; i<readCnt; i++)
        System.out.println(cbuf[i]);
    in.close();
}
```

읽어 들이는 과정에서 1바이트 데이터가 2바이트로 표현될 수 있다.

문자 기반의 입력 스트림 형성



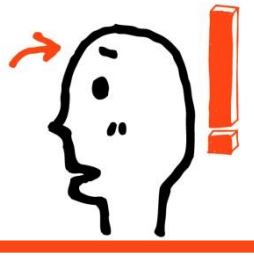
- Reader : 문자 기반 입력 스트림의 최상위 클래스로 추상 클래스

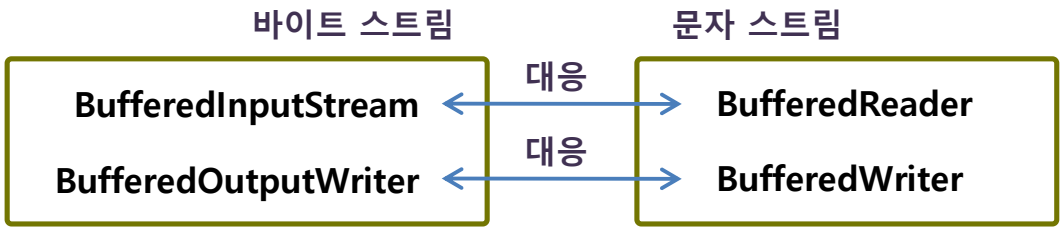


- Reader 클래스의 주요 메소드

메소드		설명
int	read()	입력 스트림으로부터 한개의 문자를 읽고 리턴한다.
int	read(char[] cbuf)	입력 스트림으로부터 읽은 문자들을 매개값으로 주어진 문자 배열 cbuf 에 저장하고 실제로 읽은 문자 수를 리턴한다.
int	read(char[] cbuf, int off, int len)	입력 스트림으로부터 len 개의 문자를 읽고 매개값으로 주어진 문자 배열 cbuf[off] 부터 len 개까지 저장한다. 그리고 실제로 읽은 문자 수인 len 개를 리턴한다.
void	close()	사용한 시스템 자원을 반납하고 입력 스트림을 닫는다.

BufferedReader & BufferedWriter





- 문자열의 입력

  BufferedReader 클래스의 다음 메소드

  ```
  public String readLine() throws IOException
  ```

- 문자열의 출력

  Writer 클래스의 다음 메소드

  ```
  public void write(String str) throws IOException
  ```

일관성 없는 문자열의 입력방식과 출력방식!
그러나 문자열의 입력뿐만 아니라 출력도 버퍼링의 존재는 도움이 되므로 입력과 출력 모두에 버퍼링 필터를 적용하자!

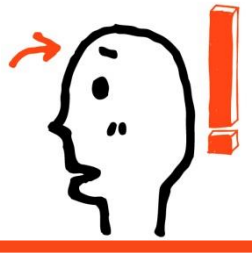
문자열 출력을 위한 스트림의 구성

```
BufferedWriter out= new BufferedWriter(new FileWriter("Strint.txt"));
```

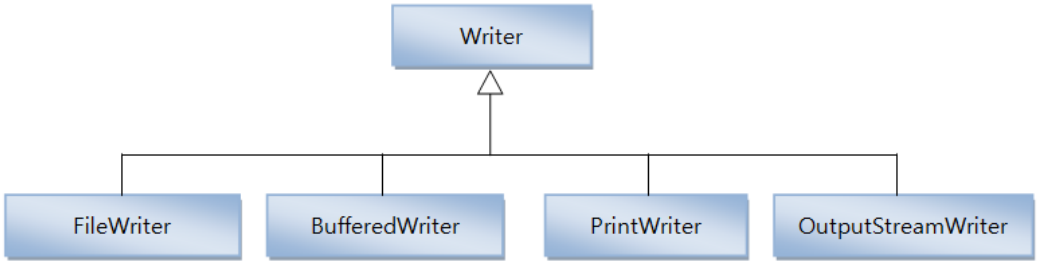
문자열 입력을 위한 스트림의 구성

```
BufferedReader in= new BufferedReader(new FileReader("Strint.txt"));
```

문자 기반의 출력 스트림 형성



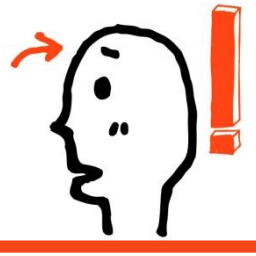
- Writer : 문자 기반 출력 스트림의 최상위 클래스로 추상 클래스



- Writer 클래스의 주요 메소드

리턴타입	메소드	설명
void	wrtie(int c)	출력 스트림으로 매개값으로 주어진 한 문자를 보낸다.
void	write(char[] cbuf)	출력 스트림에 매개값으로 주어진 문자 배열 cbuf 의 모든 문자를 보낸다.
void	write(char[] cbuf, int off, int len)	출력 스트림에 매개값으로 주어진 문자 배열 cbuf[off] 부터 len 개까지의 문자를 보낸다.
void	write(String str)	출력 스트림에 매개값으로 주어진 문자열을 전부 보낸다.
void	write(String str, int off, int len)	출력 스트림에 매개값으로 주어진 문자열 off 순번부터 len 개까지의 문자를 보낸다.
void	flush()	버퍼에 잔류하는 모든 문자열을 출력한다.
void	close()	사용한 시스템 자원을 반납하고 출력 스트림을 닫는다.

문자열 입출력의 예



```
public static void main(String[] args) throws IOException
{
    BufferedWriter out= new BufferedWriter(new FileWriter("Strint.txt"));

    out.write("박지성 - 메시 멈추게 하는데 집중하겠다.");
    out.newLine();
    out.write("올 시즌은 나에게 있어 최고의 시즌이다.");
    out.newLine();
    out.write("팀이 승리하는 것을 돕기 위해 최선을 다하겠다.");
    out.newLine();
    out.write("환상적인 결승전이 될 것이다.");
    out.newLine();
    out.newLine();
    out.write("기사 제보 및 보도자료");
    out.newLine();
    out.write("press@goodnews.co.kr");
    out.close();
    System.out.println("기사 입력 완료.");
}
```

파일 대상의 문자열 출력.
개행은 newLine 메소드의 호출을 통해서 이뤄진다. 이렇듯 ₩n이 아닌 newLine 메소드 호출을 통해서 개행을 구분하는 이유는 시스템에 따라서 개행을 표현하는 방법이 다르기 때문이다.

```
public static void main(String[] args) throws IOException
{
    BufferedReader in= new BufferedReader(new FileReader("Strint.txt"));

    String str;
    while(true)
    {
        str=in.readLine();
        if(str==null)
            break;

        System.out.println(str);
    }
    in.close();
}
```

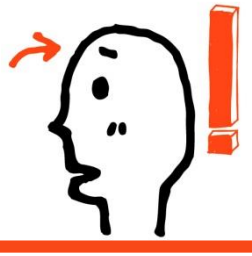
파일 대상의 문자열 입력

실행 결과

```
박지성 - 메시 멈추게 하는데 집중하겠다.
올 시즌은 나에게 있어 최고의 시즌이다.
팀이 승리하는 것을 돕기 위해 최선을 다하겠다.
환상적인 결승전이 될 것이다.

기사 제보 및 보도자료
press@goodnews.co.kr
```

readLine 메소드 호출 시 개행 정보는 문자열의 구분자로 사용되고 버려진다. 따라서 문자열 출력 후 개행을 위해서는 println 메소드를 호출해야 한다.

ObjectInputStream & ObjectOutputStream



ObjectOutputStream 클래스의 메소드:인스턴스 저장

```
public final void writeObject(Object obj) throws IOException
```

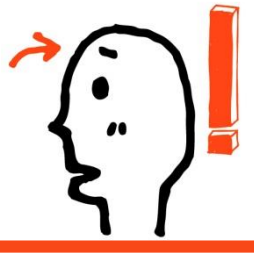
ObjectInputStream 클래스의 메소드: 인스턴스 복원

```
public final Object readObject() throws IOException, ClassNotFoundException
```

직렬화의 대상이 되는 인스턴스의 클래스는 java.io.Seiralizable 인터페이스를 구현해야 한다. 이 인터페이스는 '직렬화의 대상임을 표시'하는 인터페이스일 뿐, 실제 구현해야 할 메소드가 존재하는 인터페이스는 아니다.

인스턴스가 파일에 저장되는 과정(저장을 위해 거치는 과정)을 가리켜 직렬화(serialization)이라 하고, 그 반대의 과정을 가리켜 '역직렬화(deserialization)'이라 한다.

파일 대상의 인스턴스 저장과 복원의 예



직렬화 가능!

```
class Circle implements Serializable
{
    int xPos;
    int yPos;
    double rad;

    public Circle(int x, int y, double r)
    {
        xPos=x;
        yPos=y;
        rad=r;
    }
    public void showCirlceInfo()
    {
        System.out.printf("[%d, %d] \n", xPos, yPos);
        System.out.println("rad : "+rad);
    }
}
```

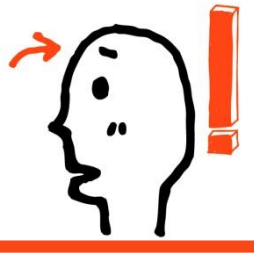
```
public static void main(String[] args)
    throws IOException, ClassNotFoundException
{
    /* 인스턴스 저장 */
    ObjectOutputStream out=
        new ObjectOutputStream(new FileOutputStream("Object.ser"));
    out.writeObject(new Circle(1, 1, 2.4));
    out.writeObject(new Circle(2, 2, 4.8));
    out.writeObject(new String("String implements Serializable"));
    out.close();

    /* 인스턴스 복원 */
    ObjectInputStream in=
        new ObjectInputStream(new FileInputStream("Object.ser"));
    Circle cl1=(Circle)in.readObject();
    Circle cl2=(Circle)in.readObject();
    String message=(String)in.readObject();
    in.close();

    /* 복원된 정보 출력 */
    cl1.showCirlceInfo();
    cl2.showCirlceInfo();
    System.out.println(message);
}
```

실행결과

```
[1, 1]
rad : 2.4
[2, 2]
rad : 4.8
String implements Serializable
```

스트림을 통한 인스턴스의 저장

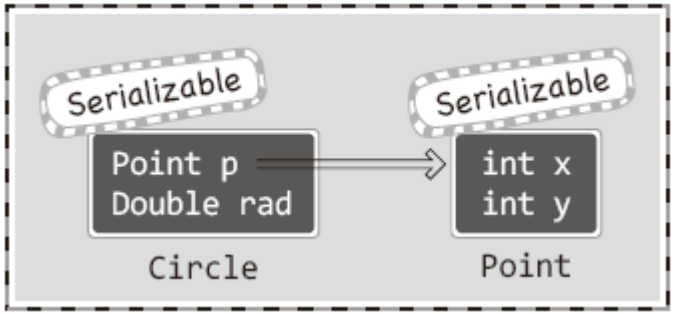
줄줄이 사탕으로 엮여 들어갑니다.



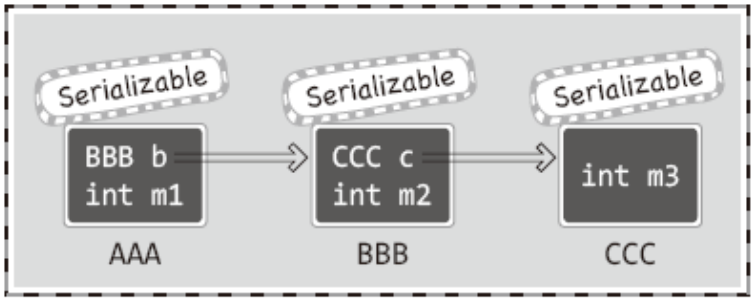
```
class Point implements Serializable
{
    int x, y;

    public Point(int x, int y)
    {
        this.x=x;
        this.y=y;
    }
}

class Circle implements Serializable
{
    Point p;
    double rad;

    public Circle(int x, int y, double r)
    {
        p=new Point(x, y);
        rad=r;
    }
    public void showCirlceInfo()
    {
        System.out.printf("[%d, %d] \n", p.x, p.y);
        System.out.println("rad : "+rad);
    }
}
```

직렬화 가능!

직렬화 가능!

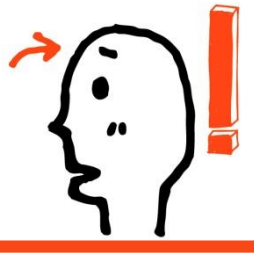


멤버인 p가 참조하는 인스턴스도 직렬화 가능하다면(Serializable 인터페이스를 구현하는 클래스의 인스턴스라면), p가 참조하는 인스턴스도 Circle 인스턴스가 직렬화 될 때 함께 직렬화가 된다.



위의 경우도 AAA 인스턴스가 직렬화 되면, AAA가 참조하는 BBB 인스턴스도, BBB가 참조하는 CCC 인스턴스도 함께 직렬화 된다.

직렬화의 대상에서 제외! transient!



```
class PersonalInfo implements Serializable
{
    String name;
    transient String secretInfo;
    int age;
    transient int secretNum;
    public PersonalInfo(String name, String sInfo, int age, int sNum)
    {
        this.name=name;
        secretInfo=sInfo;
        this.age=age;
        secretNum=sNum;
    }
    public void showCirlceInfo()
    {
        System.out.println("name : "+name);
        System.out.println("secret info : "+secretInfo);
        System.out.println("age : "+age);
        System.out.println("secret num : "+secretNum);
        System.out.println("");
    }
}
```

오리지널 데이터의 출력

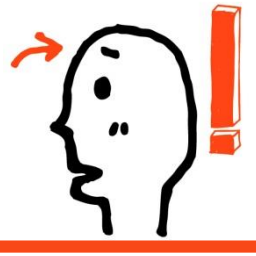
```
name : John
secret info : baby
age : 3
secret num : 42
```

직렬화, 역직렬화 후의 데이터 출력

```
name : John
secret info : null
age : 3
secret num : 0
```

transient로 선언된 멤버는 직렬화의 대상에서 제외된다! 따라서 복원 시 자료형 별 디폴트 값(null, 0, 0.0 등)이 대신 저장된다.

RandomAccessFile 클래스



RandomAccessFile 클래스의 특징!

- 입력과 출력이 동시에 이뤄질 수 있다.
- 입출력 위치를 임의로 변경할 수 있다.
- 파일을 대상으로만 존재하는 스트림이다.

RandomAccessFile 클래스는 사실상 자바 IO의 일부가 아니다. 다만, 컨트롤의 대상이 파일이기 때문에 IO와 함께 다뤄질 뿐이고, 편의상 스트림으로 분류하기도 하지만, 엄밀히 말해서 스트림이 아니다. 스트림은 임의의 위치에 데이터를 읽고 쓸 수 없다.

RandomAccessFile 클래스의 대표적인 메소드

```
public int read() throws IOException
public int read(byte[] b, int off, int len) throews IOException
public final int readInt() throws IOException
public final double readDouble() throws IOException

public void write(int b) throws IOException
public void write(byte[] b, int off, int len) throws IOException
public final void writeInt(int v) throws IOException
public final void writeDouble(double v) throws IOException

public long getFilePointer() throws IOException
public void seek(long pos) throws IOException
```

파일의 위치정보를 얻거나 변경하는 메소드

RandomAccessFile 인스턴스의 생성 및 활용의 예

생성자

```
public RandomAccessFile(String name, String mode) throws FileNotFoundException
```

```
public static void main(String[] args) throws IOException
{
    RandomAccessFile raf=new RandomAccessFile("data.bin", "rw");
    System.out.println("Write.............");
    System.out.printf("현재 입출력 위치 : %d 바이트 \n", raf.getFilePointer());

    raf.writeInt(200);
    raf.writeInt(500);
    System.out.printf("현재 입출력 위치 : %d 바이트 \n", raf.getFilePointer());

    raf.writeDouble(48.65);
    raf.writeDouble(52.24);
    System.out.printf("현재 입출력 위치 : %d 바이트 \n", raf.getFilePointer());
    System.out.println("Read.............");
    raf.seek(0);    // 맨 앞으로 이동
    System.out.printf("현재 입출력 위치 : %d 바이트 \n", raf.getFilePointer());

    System.out.println(raf.readInt());
    System.out.println(raf.readInt());
    System.out.printf("현재 입출력 위치 : %d 바이트 \n", raf.getFilePointer());

    System.out.println(raf.readDouble());
    System.out.println(raf.readDouble());
    System.out.printf("현재 입출력 위치 : %d 바이트 \n", raf.getFilePointer());
    raf.close();
}
```

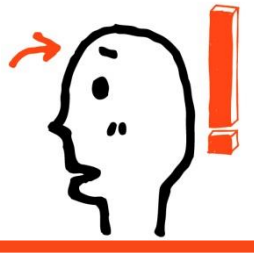
생성 모드

"r"	읽기 위한 용도
"rw"	읽고 쓰기 위한 용도

실행결과

```
Write.............
현재 입출력 위치 : 0 바이트
현재 입출력 위치 : 8 바이트
현재 입출력 위치 : 24 바이트

Read.............
현재 입출력 위치 : 0 바이트
200
500
현재 입출력 위치 : 8 바이트
48.65
52.24
현재 입출력 위치 : 24 바이트
```

File 클래스



- 디렉터리의 생성, 소멸
- 파일의 소멸
- 디렉터리 내에 존재하는 파일이름 출력

File 클래스가 지원하는 기능

```
public static void main(String[] args) throws IOException
{
    File myDir=new File("C:\\YourJava\\JavaDir");    // 디렉터리 위치 정보
    myDir.mkdir();   // 디렉터리 생성
    . . . .
}
```

디렉터리 생성의 예

```
public static void main(String[] args) throws IOException
{
    File myFile=new File("C:\\MyJava\\my.bin");
    File reFile=new File("C:\\YourJava\\my.bin");
    myFile.renameTo(reFile);      // 파일의 이동
    . . . .
}
```

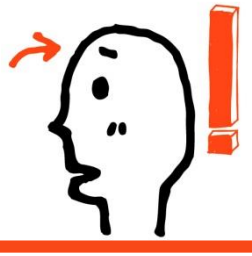
파일 이동의 예

renameTo 는 파일의 이름을 변경하는 메소드인데, 경로의 변경에 사용이 가능하다.

```
File myFile=
    new File("C:"+File.separator+"MyJava"+File.separator+"my.bin");
if(myFile.exists()==false)
{
    System.out.println("원본 파일이 준비되어 있지 않습니다.");
    return;
}
```

File.separator는 운영체제에 따른 구분자로 각각 치환된다.

File 클래스



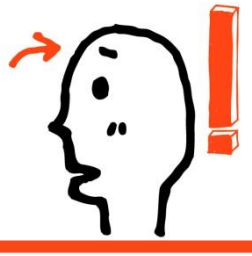
- 파일 또는 디렉토리 존재 유무 확인 메소드

```
boolean isExist = file.exists();
```

- 파일 및 디렉토리 생성 및 삭제 메소드

리턴타입	메소드	설명
boolean	createNewFile()	새로운 파일을 생성
booelan	mkdir()	새로운 디렉토리를 생성
boolean	mkdirs()	경로상에 없는 모든 디렉토리를 생성
boolean	delete()	파일 또는 디렉토리 삭제

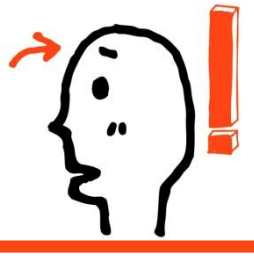
File 클래스



- 파일 및 디렉토리의 정보를 리턴하는 메소드

리턴타입	메소드	설명
boolean	canExecute()	실행할 수 있는 파일인지 여부
boolean	canRead()	읽을 수 있는 파일인지 여부
boolean	canWrite()	수정 및 저장할 수 있는 파일인지 여부
String	getName()	파일의 이름을 리턴
String	getParent()	부모 디렉토리를 리턴
File	getParentFile()	부모 디렉토리를 File 객체로 생성후 리턴
String	getPath()	전체 경로를 리턴
boolean	isDirectory()	디렉토리인지 여부
boolean	isFile()	파일인지 여부
boolean	isHidden()	숨김 파일인지 여부
long	lastModified()	마지막 수정 날짜 및 시간을 리턴
long	length()	파일의 크기 리턴
String[]	list()	디렉토리에 포함된 파일 및 서브디렉토리 목록 전부를 String 배열로 리턴
String[]	list(FilenameFilter filter)	디렉토리에 포함된 파일 및 서브디렉토리 목록 중에 FilenameFilter 에 맞는 것만 String 배열로 리턴
File[]	listFiles()	디렉토리에 포함된 파일 및 서브 디렉토리 목록 전부를 File 배열로 리턴
File[]	listFiles(FilenameFilter filter)	디렉토리에 포함된 파일 및 서브디렉토리 목록 중에 FilenameFilter 에 맞는 것만 File 배열로 리턴

상대경로 기반의 예제 작성



실제 프로그램 개발에서는 절대경로가 아닌 상대경로를 이용하는 것이 일반적이다. 그래야 실행환경 및 실행위치의 변경에 따른 문제점을 최소화할 수 있기 때문이다.

```
File subDir1=new File("AAA");  현재 디렉터리 기준으로...
File subDir2=new File("AAA\\BBB"); 현대 디렉터리에 존재하는 AAA의 하위 디렉터리인 BBB...
File subDir3=new File("AAA"+File.separator+"BBB"); AAA₩₩BBB의 운영체제 독립버전...
```

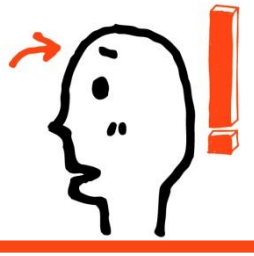
```
class RelativePath
{
    public static void main(String[] args)
    {
        File curDir=new File("AAA");
        System.out.println(curDir.getAbsolutePath());

        File upperDir=new File("AAA"+File.separator+"BBB");
        System.out.println(upperDir.getAbsolutePath());
    }
}
```

실행결과

```
C:\MyJava\YourJava>java RelativePath
C:\MyJava\YourJava\AAA
C:\MyJava\YourJava\AAA\BBB
```

위의 예제는 운영체제에 상관없이 실행이 가능하다!

File 클래스 기반의 IO 스트림 형성

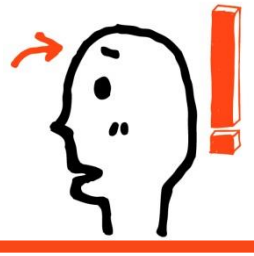


- public FileInputStream(File file) // FileInputStream의 생성자
  throws FileNotFoundException
- public FileOutputStream(File file) // FileOutputStream의 생성자
  throws FileNotFoundException
- public FileReader(File file) // FileReader의 생성자
  throws FileNotFoundException
- public FileWriter(File file) // FileWriter의 생성자
  throws IOException

```
File inFile=new File("data.bin");
if(inFile.exists()==false)
{
    // 데이터를 읽어 들일 대상 파일이 존재하지 않음에 대한 적절한 처리
}
InputStream in=new FileInputStream(inFile);
```

File 인스턴스를 생성한 다음에, 이를 이용해서 스트림을 형성하면, 보다 다양한 메소드의 호출이 가능하다.

Paths와 Path 클래스



java.nio.file.Path

파일 및 디렉토리의 경로 표현을 위해 자바 7에서 추가된 인터페이스

ex) Path path = Paths.get("C:\\JavaStudy\\PathDemo.java");

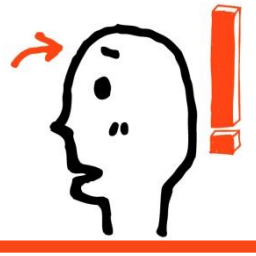
```
Path getRoot()      // 루트 디렉토리 반환
Path getParent()        // 부모 디렉토리 반환
Path getFileName()    // 파일 이름 반환
```

```
class PathDemo {
   public static void main(String[] args) {
      Path pt1 = Paths.get("C:\\JavaStudy\\PathDemo.java");
      Path pt2 = pt1.getRoot();
      Path pt3 = pt1.getParent();
      Path pt4 = pt1.getFileName();

      System.out.println("Absolute: " + pt1);
      System.out.println("Root: " + pt2);
      System.out.println("Parent: " + pt3);
      System.out.println("File: " + pt4);
}
```

Get 메소드 호출의 성공 여부는 해당 파일 또는 디렉토리의 존재 여부와 상관 없다.

```
명령 프롬프트
C:₩JavaStudy>java PathDemo
Absolute: C:₩JavaStudy₩PathDemo.java
Root: C:₩
Parent: C:₩JavaStudy
File: PathDemo.java

C:₩JavaStudy>_
```

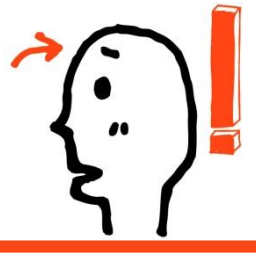
현재 디렉토리 정보의 출력 예



```
class CurrentDir {
   public static void main(String[] args) {
      Path cur = Paths.get("");   // 현재 디렉토리 정보 '상대 경로' 형태로 담긴 인스턴스 생성
      String cdir;

      if(cur.isAbsolute())
         cdir = cur.toString();
      else
         cdir = cur.toAbsolutePath().toString();

      System.out.println("Current dir: " + cdir);
   }
}
```

```
명령 프롬프트
C:₩JavaStudy>java CurrentDir
Current dir: C:₩JavaStudy

C:₩JavaStudy>
```

파일 및 디렉토리의 생성과 소멸



```
public static Path createFile(Path path, FileAttribute<?>...attrs) throws IOException
```

→ 지정한 경로에 빈 파일 생성, 경로가 유효하지 않거나 파일이 존재하면 예외 발생

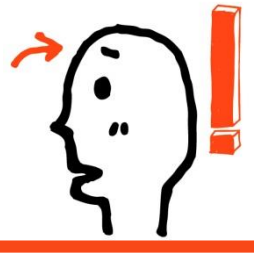
```
public static Path createDirectory(Path dir, FileAttribute<?>...attrs) throws IOException
```

→ 지정한 경로에 디렉토리 생성, 경로가 유효하지 않으면 예외 발생

```
public static Path createDirectories(Path dir, FileAttribute<?>...attrs) throws IOException
```

→ 지정한 경로의 모든 디렉토리 생성

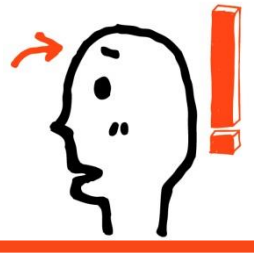
파일 및 디렉토리 생성의 예



```
public static void main(String[] args) throws IOException {
    Path fp = Paths.get("C:\\JavaStudy\\empty.txt");
    fp = Files.createFile(fp);     // 파일 생성

    Path dp1 = Paths.get("C:\\JavaStudy\\Empty");
    dp1 = Files.createDirectory(dp1);     // 디렉토리 생성

    Path dp2 = Paths.get("C:\\JavaStudy2\\Empty");
    dp2 = Files.createDirectories(dp2);     // 경로의 모든 디렉토리 생성

    System.out.println("File: " + fp);
    System.out.println("Dir1: " + dp1);
    System.out.println("Dir2: " + dp2);
}
```

```
명령 프롬프트

C:₩JavaStudy>java MakeFileAndDir
File: C:₩JavaStudy₩empty.txt
Dir1: C:₩JavaStudy₩Empty
Dir2: C:₩JavaStudy2₩Empty

C:₩JavaStudy>_
```

파일을 대상으로 하는 간단한 입력 및 출력



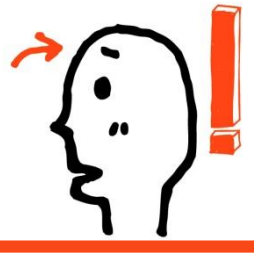
java.nio.file.Files의 메소드들(바이트 단위 입출력)

```
public static byte[] readAllBytes(Path path) throws IOException

public static Path write(Path path, byte[] bytes, OpenOption...options) throws IOException
```

- APPEND 파일의 끝에 데이터를 추가한다.
- CREATE 파일이 존재하지 않으면 생성한다.
- CREATE_NEW 새 파일을 생성한다. 이미 파일이 존재하면 예외 발생
- TRUNCATE_EXISTING 쓰기 위해 파일을 여는데 파일이 존재하면 파일의 내용을 덮어쓴다.

I/O 스트림을 기반으로 하는 방법에 비해 매우 단순하고 간단한 방법, 따라서 입출력할 데이터의 양이 적고 성능이 문제되지 않는 경우에 한해 이 방법 사용해야 한다.

간단한 입력 및 출력의 예(바이트 단위)



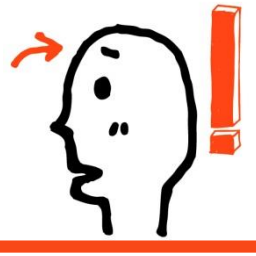
```
public static void main(String[] args) throws IOException {
   Path fp = Paths.get("C:\\JavaStudy\\simple.bin");

   // 파일 생성, 파일이 존재하면 예외 발생
   fp = Files.createFile(fp);

   byte buf1[] = {0x13, 0x14, 0x15};    // 파일에 쓸 데이터
   for(byte b : buf1)    // 저장할 데이터의 출력을 위한 반복문
      System.out.print(b + "\t");
   System.out.println();

   // 파일에 데이터 쓰기
   Files.write(fp, buf1, StandardOpenOption.APPEND);

   // 파일로부터 데이터 읽기
   byte buf2[] = Files.readAllBytes(fp);

   for(byte b : buf2)    // 읽어 들인 데이터의 출력을 위한 반복문
      System.out.print(b + "\t");
   System.out.println();
}
```

```
명령 프롬프트
C:₩JavaStudy>java SimpleBinWriteRead
19      20      21
19      20      21

C:₩JavaStudy>
```

파일 open, close 과정 없음

데이터 읽을 때 배열도 준비해 둘 필요가 없음

문자 데이터의 간단한 입력 및 출력



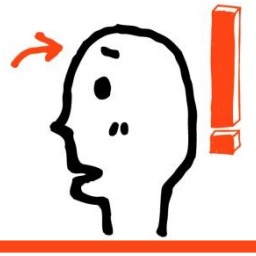
java.nio.file.Files의 메소드들(문자열 단위 입출력)

```
public static List<String> readAllLines(Path path) throws IOException

public static Path write(
     Path path,
     Iterable<? extends CharSequence> lines,
     OpenOption...options
) throws IOException
```

Iterable<E> 인터페이스를 Collection<E> 인터페이스가 상속한다.
CharSequence 인터페이스를 String 클래스가 구현한다.

```
String st1 = "One Simple String";
String st2 = "Two Simple String";
List<String> lst = Arrays.asList(st1, st2); // write 메소드의 두 번째 인자로 전달 가능
```

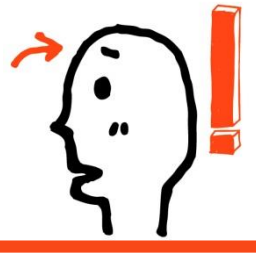
문자 데이터의 간단한 입력 및 출력의 예



```
class SimpleTxtWriteRead {
   public static void main(String[] args) throws IOException {
      Path fp = Paths.get("C:\\JavaStudy\\simple.txt");
      String st1 = "One Simple String";
      String st2 = "Two Simple String";
      List<String> lst1 = Arrays.asList(st1, st2);

      Files.write(fp, lst1); // 파일에 문자열 저장하기
      List<String> lst2 = Files.readAllLines(fp); // 파일로부터 문자열 읽기
      System.out.println(lst2);
   }
}
```

```
명령 프롬프트
C:₩JavaStudy>java SimpleTxtWriteRead
[One Simple String, Two Simple String]

C:₩JavaStudy>
```

파일 및 디렉토리의 복사와 이동



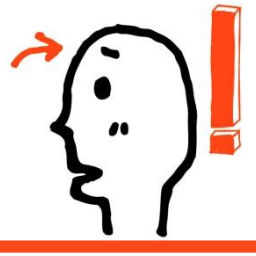
```
public static Path copy(
        Path source, Path target, CopyOption...options) throws IOException
```

- REPLACE_EXISTING 이미 파일이 존재한다면 해당 파일을 대체한다.
- COPY_ATTRIBUTES 파일의 속성까지 복사를 한다.

```
public static Path move(
        Path source, Path target, CopyOption...options) throws IOException
```

- REPLACE_EXISTING 이미 파일이 존재한다면 해당 파일을 대체한다.

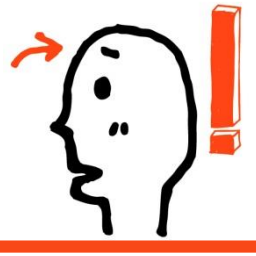
파일 및 디렉토리의 복사와 이동의 예



```
class CopyFileFromFiles {
   public static void main(String[] args) throws IOException {
      Path src = Paths.get("C:\\JavaStudy\\CopyFileFromFiles.java");
      Path dst = Paths.get("C:\\JavaStudy\\CopyFileFromFiles2.java");

      // src가 지시하는 파일을 dst가 지시하는 위치와 이름으로 복사
      Files.copy(src, dst, StandardCopyOption.REPLACE_EXISTING);
   }
}
```

```
class MoveFileFromFiles {
   public static void main(String[] args) throws IOException {
      Path src = Paths.get("C:\\JavaStudy\\Dir1");
      Path dst = Paths.get("C:\\JavaStudy\\Dir2");

      // src가 지시하는 디렉토리를 dst가 지시하는 디렉토리로 이동
      Files.move(src, dst, StandardCopyOption.REPLACE_EXISTING);
   }
}
```

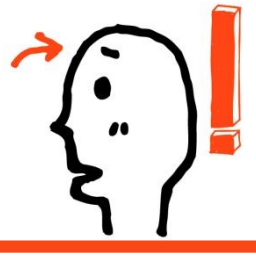
바이트 스트림의 생성 (NIO.2 기반)



```
public static void main(String[] args) {
   Path fp = Paths.get("data.dat");

   try(DataOutputStream out = new DataOutputStream(Files.newOutputStream(fp))) {
      out.writeInt(370);
      out.writeDouble(3.14);
   }
   catch(IOException e) {
      e.printStackTrace();
   }
}
```

```
public static void main(String[] args) {
   Path fp = Paths.get("data.dat");

   try(DataInputStream in = new DataInputStream(Files.newInputStream(fp))) {
      int num1 = in.readInt();
      double num2 = in.readDouble();
      System.out.println(num1);
      System.out.println(num2);
   }
   catch(IOException e) {
      e.printStackTrace();
   }
}
```

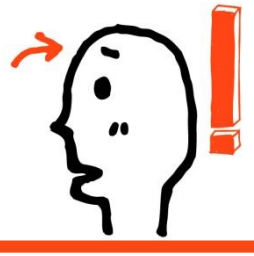
문자 스트림의 생성 (NIO.2 기반)



```
public static void main(String[] args) {
   String ks = "공부에 있어서 . . . ";
   String es = "Life is long if . . . ";
   Path fp = Paths.get("String.txt");

   try(BufferedWriter bw = Files.newBufferedWriter(fp)) {
      bw.write(ks, 0, ks.length());
      bw.newLine();
      bw.write(es, 0, es.length());
   }
   catch(IOException e) {
      e.printStackTrace();
   }
}
```

```
public static void main(String[] args) {
   Path fp = Paths.get("String.txt");

   try(BufferedReader br = Files.newBufferedReader(fp)) {
      String str;
      while(true) {
         str = br.readLine();
         if(str == null)
            break;
         System.out.println(str);
      }
   }
   catch(IOException e) {
      e.printStackTrace();
   }
}
```

NIO의 채널(Channel)과 버퍼(Buffer)



스트림과 채널의 공통점

"스트림도 채널도 데이터의 입력 및 출력을 위한 통로가 된다."

스트림과 채널의 차이점

"스트림은 한 방향으로만 데이터가 이동하지만 채널은 양방향으로 데이터 이동이 가능하다."

채널에만 존재하는 제약사항

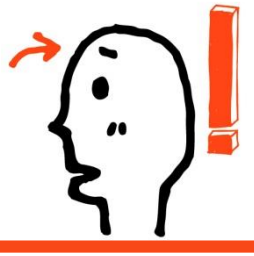
"채널은 반드시 버퍼에 연결해서 사용해야 한다."

채널 기반 데이터 입출력 경로

출력 경로: 데이터 ⇨ 버퍼 ⇨ 채널 ⇨ 파일

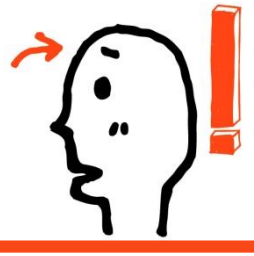
입력 경로: 데이터 ⇦ 버퍼 ⇦ 채널 ⇦ 파일

NIO 기반 파일 복사 예제



```
public static void main(String[] args) {
   Path src = 복사할 대상 파일
   Path dst = 사본 이름

   // 하나의 버퍼 생성
   ByteBuffer buf = ByteBuffer.allocate(1024);

   // try에서 두 개의 채널 생성
   try(FileChannel ifc = FileChannel.open(src, StandardOpenOption.READ);
         FileChannel ofc = FileChannel.open(dst, StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
      int num;
      while(true) {
         num = ifc.read(buf); // 채널 ifc에서 버퍼로 읽어 들임
         if(num == -1) // 읽어 들인 데이터가 없다면
            break;

         buf.flip(); // 모드 변환!
         ofc.write(buf); // 버퍼에서 채널 ofc로 데이터 전송
         buf.clear(); // 버퍼 비우기
      }
   }
   catch(IOException e) {
      e.printStackTrace();
   }
}
```

성능 향상 포인트는 어디에?



효율적인 버퍼링

기본 IO 스트림 기반의 복사 프로그램과 달리 하나의 버퍼만을 사용하였다.

버퍼의 수가 줄은 것이 핵심이 아니라, 이로 인해 버퍼에서 버퍼로의 데이터 이동이 불필요해진 부분이 핵심

Non-direct 버퍼를 대신하는 Direct 버퍼의 생성

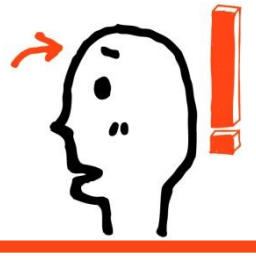
```
ByteBuffer buf = ByteBuffer.allocate(1024);   // Non-direct 버퍼
```

: 파일 ⇨ 운영체제 버퍼 ⇨ 가상머신 버퍼 ⇨ 실행 중인 자바 프로그램

```
ByteBuffer buf = ByteBuffer.allocateDirect(1024);   // Direct 버퍼 생성
```

: 파일 ⇨ 운영체제 버퍼 ⇨ 실행 중인 자바 프로그램

파일 랜덤 접근(File Random Access)



```
public static void main(String[] args) {
   Path fp = Paths.get("data.dat");
   ByteBuffer wb = ByteBuffer.allocate(1024); // 버퍼 생성
   wb.putInt(120); // int형 데이터 버퍼에 저장
   wb.putInt(240);
   wb.putDouble(0.94); // double형 데이터 버퍼에 저장
   wb.putDouble(0.75);

   // 하나의 채널 생성
   try(FileChannel fc = FileChannel.open(fp, StandardOpenOption.CREATE,
         StandardOpenOption.READ, StandardOpenOption.WRITE)) {
      // 파일에 쓰기
      wb.flip();
      fc.write(wb);

      // 파일로부터 읽기
      ByteBuffer rb = ByteBuffer.allocate(1024); // 버퍼 생성
      fc.position(0); // 채널의 포지션을 맨 앞으로 이동
      fc.read(rb);

      // 이하 버퍼로부터 데이터 읽기
      rb.flip();
      System.out.println(rb.getInt());
      rb.position(Integer.BYTES * 2); // 버퍼의 포지션 이동
      System.out.println(rb.getDouble());
      System.out.println(rb.getDouble());

      rb.position(Integer.BYTES); // 버퍼의 포지션 이동
      System.out.println(rb.getInt());
   } catch(IOException e) {
      e.printStackTrace();
   }
}
```

```
명령 프롬프트
C:₩JavaStudy>java FileRandomAccess
120
0.94
0.75
240

C:₩JavaStudy>_
```

파일 랜덤 접근

파일에 데이터를 쓰거나 읽을 때

원하는 위치에 쓰거나 읽는 것을 의미한다.