JAVA

- 파일과 I/O 스트림

파일과 I/O 스트림

스트림(Stream)

I/O의 범위와 간단한 I/O 모델의 소개

- 일반적인 입출력

  → 키보드와 모니터

  → 하드디스크에 저장되어 있는 파일

  → USB와 같은 외부 메모리 장치

  → 네트워크로 연결되어 있는 컴퓨터

  → 사운드카드, 오디오카드와 같은 멀티미디어 장치

  → 프린터, 팩스와 같은 출력 장치

I/O의 범위와 간단한 I/O 모델의 소개

입출력 대상이 달라지면 프로그램 상에서의 입출력 방식도 달라지는 것이 보통이다.

그런데 자바에서는 입출력대상에 상관없이 입출력의 진행방식이 동일 하도록 별도의 'I/O 모델'을 정의하고 있다.

I/O 모델의 정의로 인해서 입출력 대상의 차이에 따른 입출력방식의 차이는 크지 않다. 기본적인 입출력의 형태는 동일하다.

그리고 이것이 JAVA의 I/O 스트림이 갖는 장점이다.

I/O의 범위와 간단한 I/O 모델의 소개

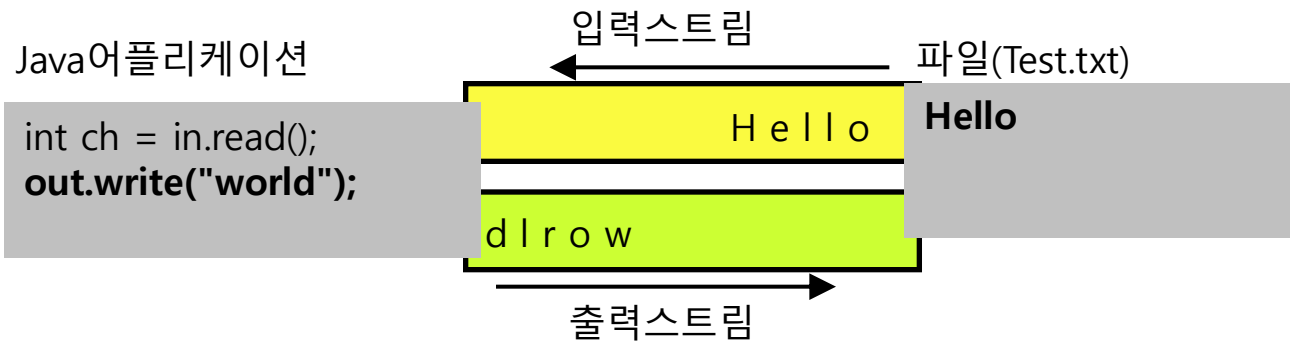
입출력(I/O)과 스트림(stream)

▶ **입출력(I/O)이란?**

- 입력(Input)과 출력(Output)을 줄여 부르는 말
- 두 대상 간의 데이터를 주고 받는 것

▶ **스트림(stream)이란?**

- 데이터를 운반(입출력)하는데 사용되는 연결통로
- 연속적인 데이터의 흐름을 물(stream)에 비유해서 붙여진 이름
- 하나의 스트림으로 입출력을 동시에 수행할 수 없다.(단방향 통신)
- 입출력을 동시에 수행하려면, 2개의 스트림이 필요하다.



스트림의 개념과 종류

- **자바의 스트림**
  - 입·출력 장치와 프로그램을 연결하며, 이들 사이의 데이터 흐름을 처리하는 소프트웨어 모듈
  - 데이터를 처리하기 위한 공통된 방법을 제공함
  - 입력 스트림 : 입력 장치로부터 자바 프로그램으로 전달되는 데이터의 흐름 혹은 데이터 전송 소프트웨어 모듈
  - 출력 스트림 : 자바 프로그램에서 출력 장치로 보내는 데이터의 흐름 또는 데이터 전송 소프트웨어 모듈

스트림의 개념과 종류

- **자바 입·출력 스트림의 특징**
  - 스트림은 FIFO 구조
    - FIFO : 먼저 들어간 것이 먼저 나오는 형태 → 데이터의 순서가 바뀌지 않음
  - 스트림은 단방향
    - 읽기, 쓰기가 동시에 되지 않음
    - 읽는 스트림과 쓰는 스트림을 하나씩 열어 사용해야 함
  - 스트림은 지연될 수 있음
    - 스트림은 넣어진 데이터가 처리되기 전까지는 스트림에 사용되는 스레드가 지연상태에 빠짐
    - 네트워크 내에서는 데이터가 모두 전송되기 전까지 네트워크 스레드는 지연상태를 유지함

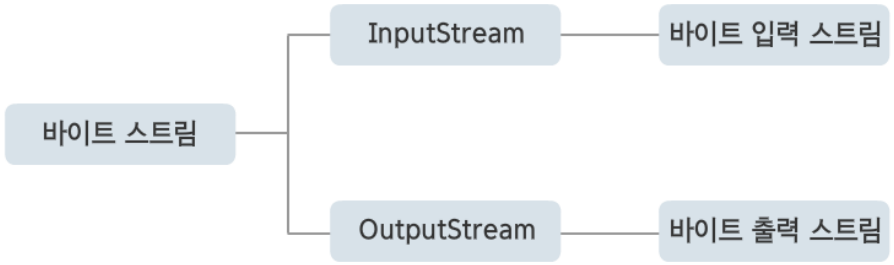
스트림의 개념과 종류

- **자바 입·출력 스트림의 종류**
  - **바이트 스트림**
    - 문자 단위/바이트 단위
    - 바이트 스트림은 1바이트를 입·출력 할 수 있는 스트림
    - 자바에서 입·출력 스트림을 통해 흘러가는 데이터의 기본 단위
    - 일반적으로 바이트로 구성된 파일을 처리하기에 적합
  - **문자 스트림**
    - 유니코드로 된 문자를 입·출력 하는 스트림
    - 2바이트를 입·출력 - 세계 모든 언어로 구성된 파일을 입·출력 하기에 적합
    - 이미지, 동영상과 같은 바이너리 데이터는 입·출력 할 수 없음
    - 문자 데이터만 입·출력 가능
  - **버퍼 스트림**
    - 문자 입력 스트림으로부터 문자를 읽어 들이거나 문자 출력 스트림으로 문자를 내보낼 때 버퍼링을 함으로써 문자, 문자 배열, 문자열 라인 등을 보다 효율적으로 처리

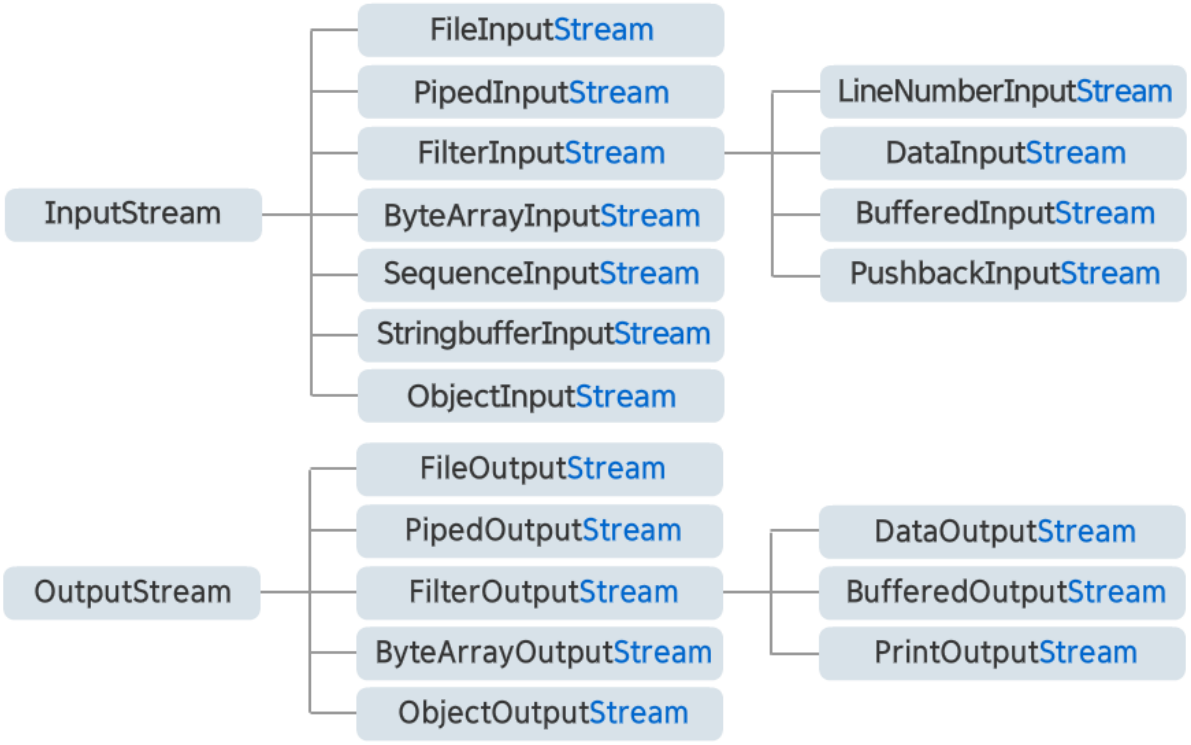
FILE I/O에 대한 소개

바이트 스트림

- 바이트 스트림의 종류



- 바이트 스트림의 구조



바이트 스트림

- **바이트 스트림 클래스**
  - JAVA . IO 패키지 포함
  - **InputStream/OutputStream**
    → 추상 클래스로써 바이트 입·출력 스트림을 다루는 모든 클래스의 슈퍼 클래스
  - **FileInputStream/FileOutputStream**
    → 파일로 부터 바이트 단위로 읽거나 저장하는 클래스
    → 바이너리 파일을 입·출력을 할 때 사용

I/O의 범위와 간단한 I/O 모델의 소개

- 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 형성 해야 한다.
그리고 여기서 말하는 스트림 이라는 것도 인스턴스의 생성을 통해서
형성된다.

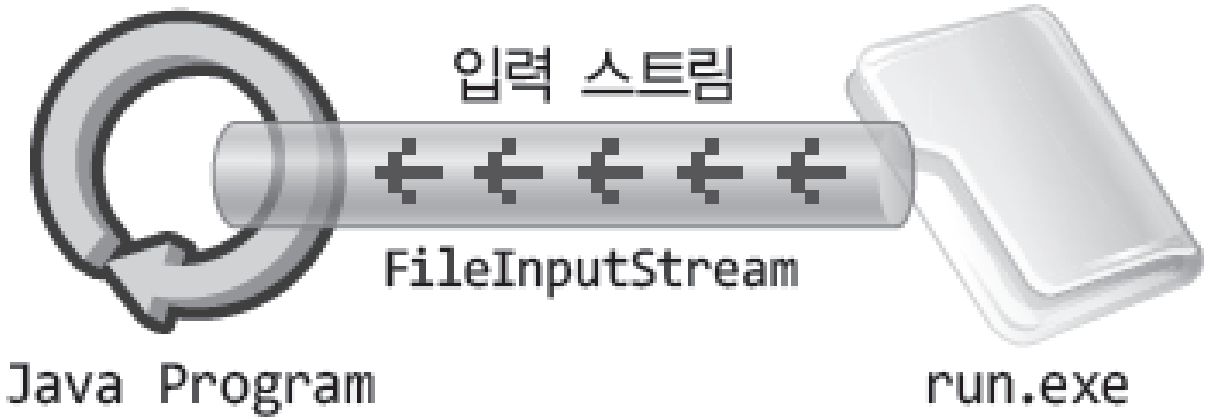
파일 기반의 입력 스트림 형성

- 파일run.exe 대상의 입력 스트림 생성

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

  → 스트림의 생성은 결국 인스턴스의 생성.

  → FileInputStream 클래스는 InputStream 클래스를 상속한다.



파일 기반의 입력 스트림 형성

- 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 기반의 입력 스트림 형성

Inputstream ⟷ Outputstream

FileInputstream ⟷ FileOutputstream

입출력 스트림은 대부분 쌍(Pair)을 이룬다.

파일 기반의 입력 스트림 형성

```
InputStream in=new FileInputStream("run.exe");
Int bData=in.read() // 1byte 데이터
// 오버라이딩에 의해 FileInputStream의 read 메소드 호출
// 더 이상 읽어들일 데이터가 없다면 -1을 반환
In.close(); // 입력 스트림 소멸
```

파일 대상의 출력 스트림 형성

- 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 전달한다.
Out.write(1);  // 4바이트 int형 정수 2의 하위 1바이트만 전달한다.
Out.close;     // 입력 스트림 소멸
```

스트림 기반의 파일 입출력 예제

```
import java.io.*;
class ByteFileCopy
{
        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);
        }
}
```

보다 빠른 속도의 파일 복사 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

보다 빠른 속도의 파일 복사 프로그램

```
import java.io.*;
class BufferFileCopy
{
        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]; //1kb짜리 버프 생성
                while(true)
                {
                        readLen=in.read(buf); //배열 buf 에 데이터 입력
                        if(readLen==-1)                 break;
                        out.write(buf, 0, readLen);
                        copyByte+=readLen;
                }
                in.close();
                out.close();
                System.out.println("복사된 바이트 크기 "+ copyByte);
        }
}
```

필터 스트림의 이해와 활용

바이트 단위로 데이터를 읽고 쓸 줄은 알지만

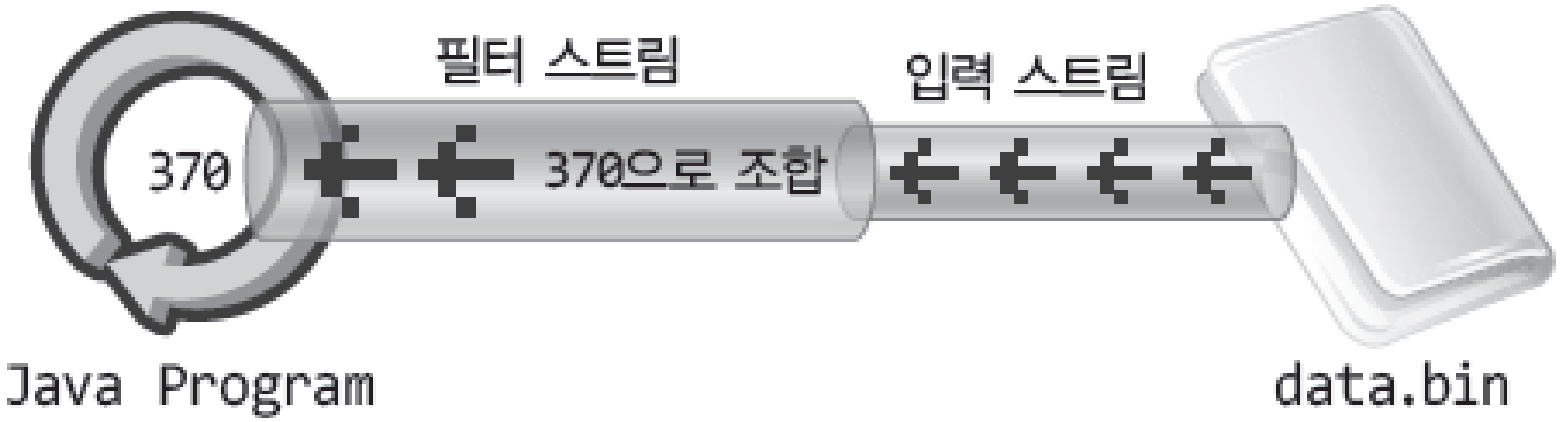
필터 스트림 (보조 스트림)

- 스트림의 기능을 향상시키거나 새로운 기능을 추가하기 위해 사용
- 독립적으로 입출력을 수행할 수 없다.

```
// 먼저 기반스트림을 생성한다.
FileInputStream fis = new FileInputStream("test.txt");
// 기반스트림을 이용해서 보조스트림을 생성한다.
BufferedInputStream bis = new BufferedInputStream(fis);

bis.read();  // 보조스트림인 BufferedInputStream으로부터 데이터를 읽는다.
```

입력	출력	설명
FilterInputStream	FilterOutputStream	필터를 이용한 입출력 처리
BufferedInputStream	BufferedOutputStream	버퍼를 이용한 입출력 성능향상
DataInputStream	DataOutputStream	int, float와 같은 기본형 단위(primitive type)로 데이터를 처리하는 기능
SequenceInputStream	SequenceOutputStream	두 개의 스트림을 하나로 연결
LineNumberInputStream	없음	읽어 온 데이터의 라인 번호를 카운트 (JDK1.1부터 LineNumberReader로 대체)
ObjectInputStream	ObjectOutputStream	데이터를 객체단위로 읽고 쓰는데 사용. 주로 파일을 이용하며 객체 직렬화와 관련있음
없음	PrintStream	버퍼를 이용하며, 추가적인 print관련 기능(print, printf, println메서드)
PushbackInputStream	없음	버퍼를 이용해서 읽어 온 데이터를 다시 되돌리는 기능 (unread, push back to buffer)

바이트 단위로 데이터를 읽고 쓸 줄은 알지만



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

버퍼링 기능을 제공하는 필터 스트림

BufferedInputStream과 BufferedOutputStream

- 입출력 효율을 높이기 위해 버퍼(byte[])를 사용하는 보조스트림

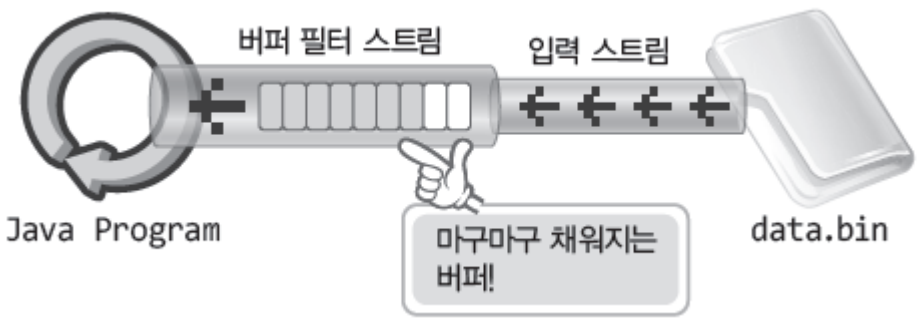
메서드 / 생성자	설 명
BufferedInputStream(InputStream in, int size)	주어진 InputStream인스턴스를 입력소스(input source)로하며 지정된 크기(byte단위)의 버퍼를 갖는 BufferedInput Stream인스턴스를 생성한다.
BufferedInputStream(InputStream in)	주어진 InputStream인스턴스를 입력소스(input source)로하며 버퍼의 크기를 지정해주지 않으므로 기본적으로 8192 byte 크기의 버퍼를 갖게 된다.

메서드 / 생성자	설 명
BufferedOutputStream(OutputStream out, int size)	주어진 OutputStream인스턴스를 출력소스(output source)로하며 지정된 크기(단위byte)의 버퍼를 갖는 BufferedOutputStream인스턴스를 생성한다.
BufferedOutputStream(OutputStream out)	주어진 OutputStream인스턴스를 출력소스(output source)로하며 버퍼의 크기를 지정해주지 않으므로 기본적으로 8192 byte 크기의 버퍼를 갖게 된다.
flush()	버퍼의 모든 내용을 출력소스에 출력한 다음, 버퍼를 비운다.
close()	flush()를 호출해서 버퍼의 모든 내용을 출력소스에 출력하고, BufferedOutputStream인스턴스가 사용하던 모든 자원을 반환한다.

- 보조스트림을 닫으면 기반스트림도 닫힌다.

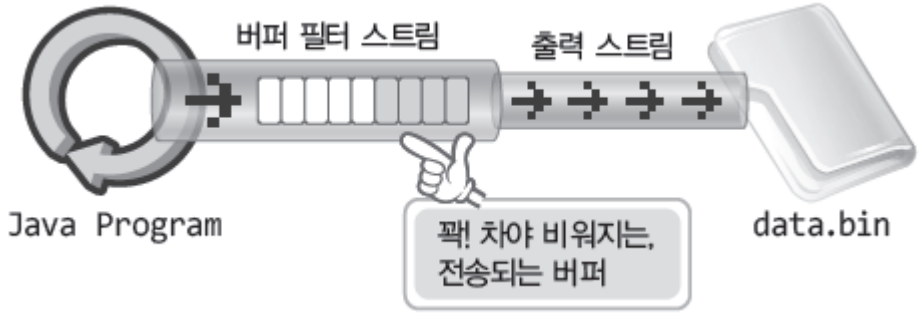
```
public class FilterOutputStream extends OutputStream {
    protected OutputStream out;
    public FilterOutputStream(OutputStream out) {
        this.out = out;
    }
    ...
    public void close() throws IOException {
        try {  flush(); } catch (IOException ignored) {}
        out.close(); // 기반 스트림의 close()를 호출한다.
    }
}
```

버퍼링 기능을 제공하는 필터 스트림



BufferedInputStream

버파필터 입력 스트림



BufferedOutputStream

버파필터 출력 스트림

BufferedOutputStream 클래스의 flush 메소드 호출을 통해서 버퍼링된 데이터의 목적지 전송이 가능하다!

또한 close 메소드를 통해서 스트림을 종료하면, 스트림의 버퍼는 flush! 된다.

버퍼링 기능을 제공하는 필터 스트림 예제

```
import java.io.*;
class ByteBufferedFileCopy
{
        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);
        }
}
```

문자 스트림의 이해와 활용

바이트 스트림과 문자 스트림의 차이

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 스트림과 문자 스트림의 차이

바이트 스트림을 이용한 파일 대상의 문자 저장의 문제점

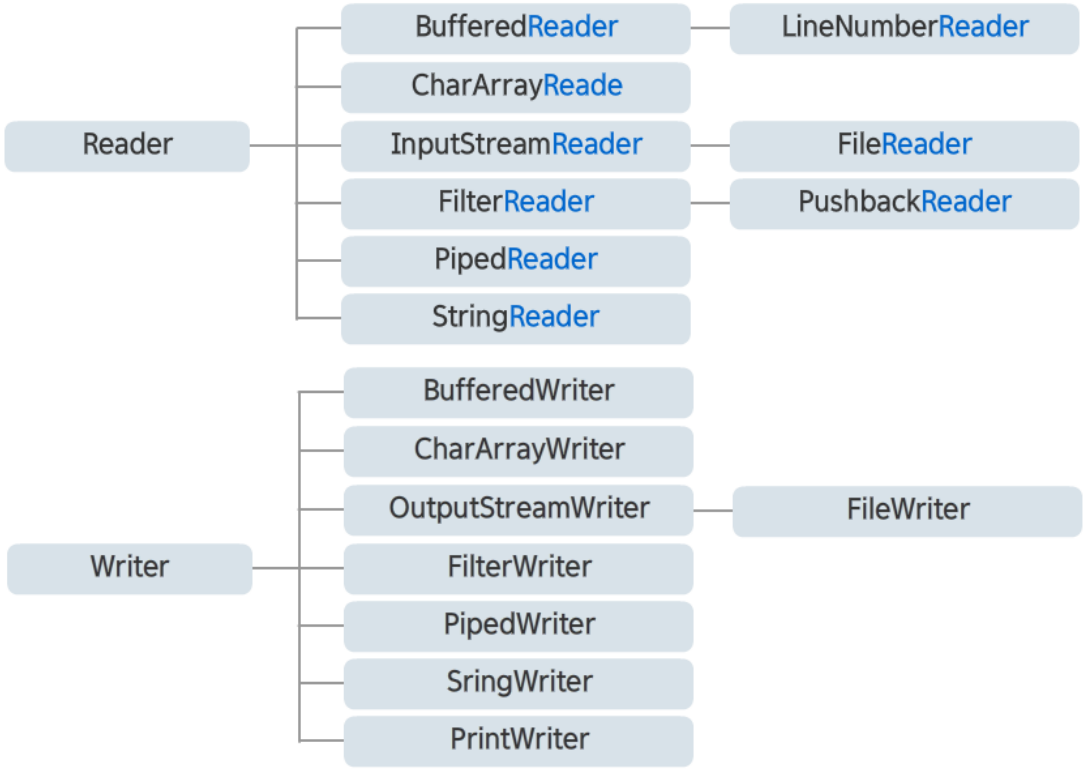
운영체제 별로 고유의 문자표현방식이 존재한다.

그리고 운영 체제에서 동작 하는 프로그램은 해당 운영체제의 문자 표현 방식을 따른다.

따라서 파일에 저장된 데이터는 해당 운영체제의 문자 표현방식을 따라서 저장 되어있어야 한다.

해당 운영체제에서 동작하는 JAVA 프로그램이 아닌 다른 프로그램에 의해서 참조가 되는 파일 이라면 정상적인 처리가 불가능합니다.

- **문자 스트림**
  - 유니코드로 된 문자를 입·출력 하는 스트림으로 문자 데이터만 입·출력 가능
  - 2바이트를 입·출력
    - 세계 모든 언어로 구성된 파일을 입·출력 하기에 적합
  - 이미지, 동영상과 같은 바이너리 데이터는 입·출력 할 수 없음
  - 문자 스트림의 구조



FileReader & FileWriter

바이트 스트림	문자 스트림
InputStream, OutputStream	Reader, Writer
FileInputStream, FileOutputStream	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 int **write**(int c) throws IOException

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

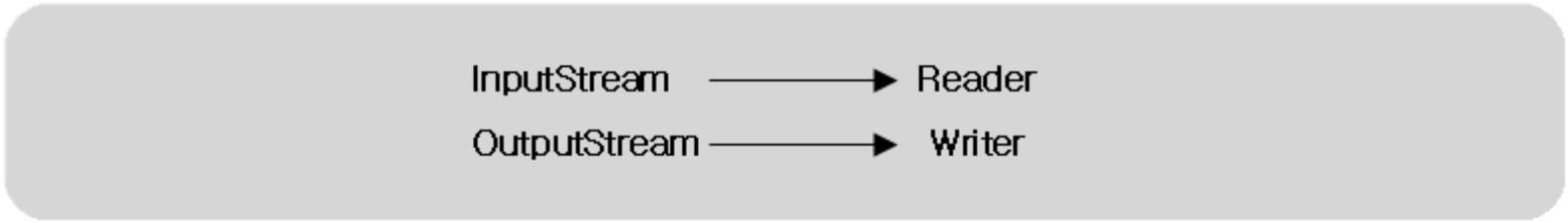
바이트 스트림과 문자 스트림의 차이

문자기반 스트림 – Reader, Writer

- 입출력 단위가 문자(char, 2 byte)인 스트림. 문자기반 스트림의 최고조상

바이트기반 스트림	문자기반 스트림	대상
FileInputStream **File**OutputStream	**File**Reader **File**Writer	파일

바이트기반 보조스트림	문자기반 보조스트림
BufferedInputStream **Buffered**OutputStream	**Buffered**Reader **Buffered**Writer



InputStream	Reader
abstract int read() int read(**byte[] b**) int read(**byte[] b**, int off, int len)	int read() int read(**char[]** cbuf) **abstract** int read(**char[]** cbuf, int off, int len)
OutputStream	**Writer**
abstract void write(int b) void write(**byte[]** b) void write(**byte[]** b, int off, int len)	void write(int c) void write(**char[]** cbuf) **abstract** void write(**char[]** cbuf, int off, int len) void write(String str) void write(String str, int off, int len)

FileReader & FileWriter

```
import java.io.*;

class FileWriterStream
{
	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();
	}
}
```

FileReader & FileWriter

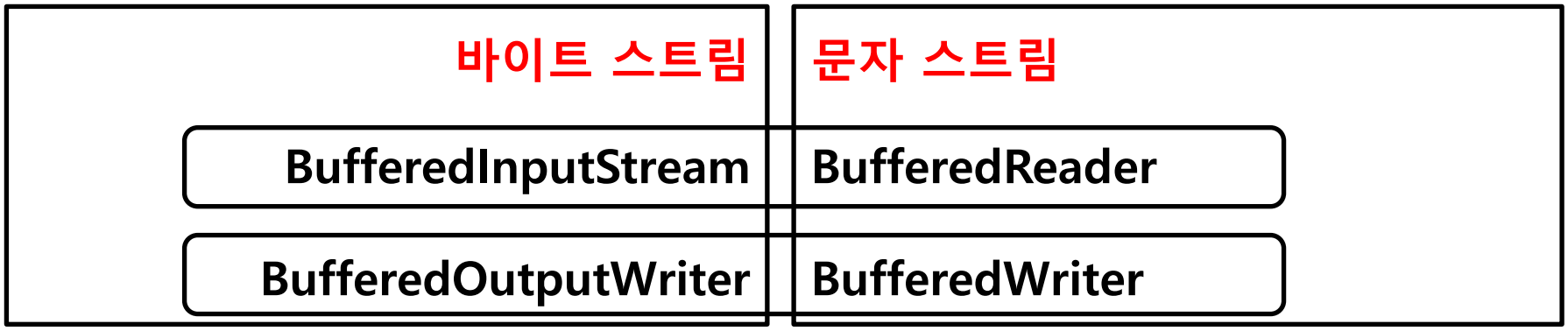
```
import java.io.*;

class FileReaderStream
{
	public static void main(String[] args) throws IOException
	{
		char[] cbuf=new char[10]; //최대 10개의 문자 읽어 저장
		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();
	}
}
```

BufferedReader와 BufferedWriter

- 입출력 효율을 높이기 위해 버퍼(char[])를 사용하는 보조스트림
- 라인(line)단위의 입출력이 편리하다.

```
String readLine() - 한 라인을 읽어온다. (BufferedReader의 메서드)
void newLine() - '라인 구분자(개행문자)'를 출력한다. (BufferedWriter의 메서드)
```

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 입력 방식과 출력방식!
그러나 문자열의 입력뿐만 아니라 출력도 버퍼링의 존재는 도움이 되므로 입력과 출력 모두에 버퍼링 필터를 적용하자!

BufferedReader & BufferedWriter

- **문자열의 입력을 위한 스트림의 구성**

  BufferedReader in=new BufferedReader(new FileReder("String.txt"));

- **문자열의 출력을 위한 스트림 구성**

  BufferedWriter out=new BufferedWriter(new FIleWrite("String.txt"));

문자열 입출력의 예

```
import java.io.*;
class StringWriter
{
	public static void main(String[] args) throws IOException
		{
			BufferedWriter out=new BufferedWriter(new FileWriter("String.txt"));

			out.write("손흥민 – 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.");
		}
}
```

문자열 입출력의 예

```
import java.io.*;

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

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

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

문자 필터 스트림 PrintWriter

System.out이 PrintStream임을 기억하고,

public static final PrintStream out;

이 이상으로 PrintStream을 활용하지 않는다.

printf, print등 문자열 단위의 출력이 필요하다면

반드시 PrintWriter를 사용한다.

PrintStream과 PrintWriter는 유사하다.

PrintWriter는 PrintStream을 대신할 수 있도록 정의된 클래스이다.

따라서 PrintWriter의 활용을 권고한다.

이는 입력 필터 스트림이 존재하지 않는 대표적인 스트림 클래스이다.

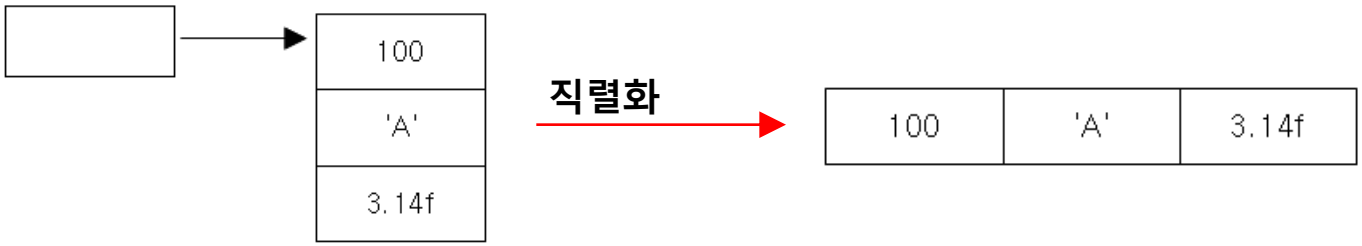
PrintWriter 예제

```
import java.io.*;

class PrintWriterStream
{
	public static void main(String[] args) throws IOException
	{
		PrintWriter out=
			new PrintWriter(new FileWriter("printf.txt"));

		out.printf("제 나이는 %d살 입니다.", 24);
		out.println("");

		out.println("저는 자바가 좋습니다.");
		out.print("특히 I/O 부분에서 많은 매력을 느낍니다.");
		out.close();
	}
}
```

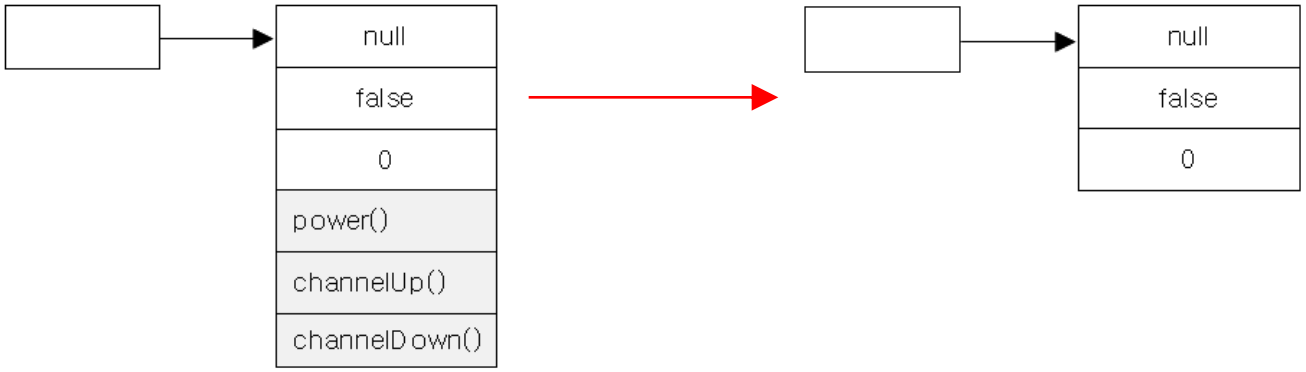
스트림을 통한 인스턴스의 저장

직렬화(serialization)란?

- 객체를 '연속적인 데이터'로 변환하는 것. 반대과정은 '역직렬화'라고 한다.
- 객체의 인스턴스변수들의 값을 일렬로 나열하는 것



- 객체를 저장하기 위해서는 객체를 직렬화해야 한다.
- 객체를 저장한다는 것은 객체의 모든 인스턴스변수의 값을 저장하는 것



ObjectInputStream & ObjectOutputStream

- **ObjectOutputStream 클래스의 메소드:인스턴스 저장**

  public final void **writeObject(Object obj)** throws IOException
- **ObjectInputStream 클래스의 메소드: 인스턴스 복원**

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

직렬화의 대상이 되는 인스턴스의 클래스는 java.io.Seiralizable 인터페이스를 구현 해야 한다.

이 인터페이스는 '직렬화의 대상임을 표시'하는 인터페이스 일뿐, 실제 구현해야 할 메소드가 존재하는 인터페이스는 아니다.

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

파일 대상의 인스턴스 저장과 복원의 예

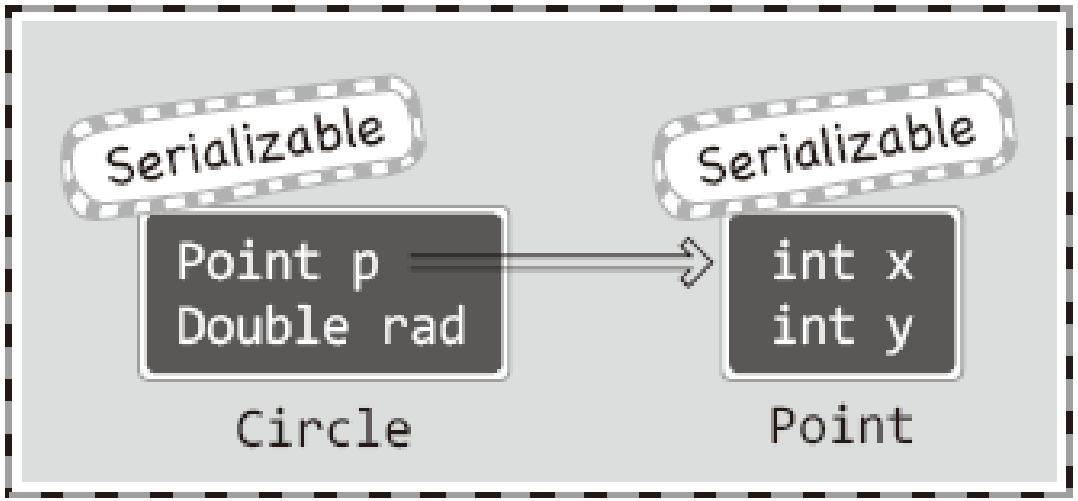
```
import java.io.*;

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);
	}
}
```

파일 대상의 인스턴스 저장과 복원의 예

```
class ObjectSerializable {
    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);
        }
}
```

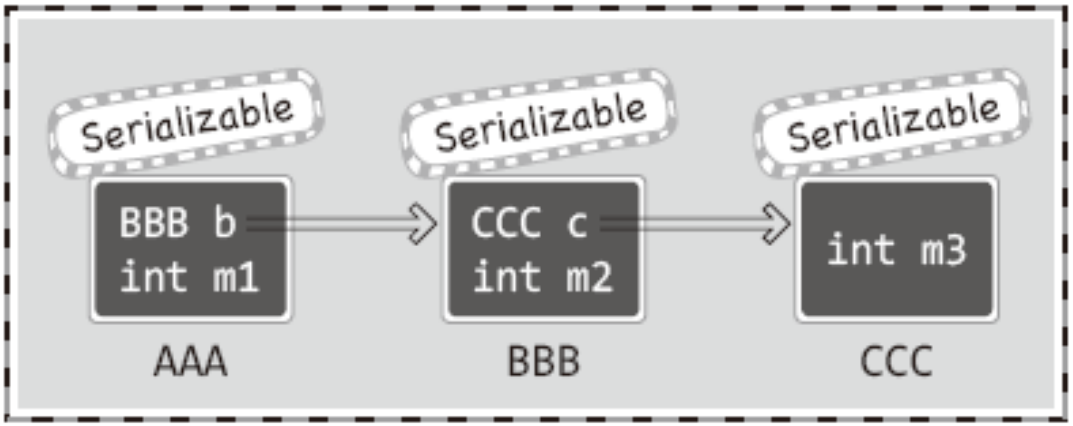
줄줄이 사탕으로 엮여 들어갑니다.

```
import java.io.*;

class Point implements Serializable
{
        int x, y;
        public Point(int x, int y) { this.x=x; this.y=y; }
}
```



줄줄이 사탕으로 엮여 들어갑니다.

```
class Circle implements Serializable
{
        Point p; double rad;
//멤버인 p가 참조하는 인스턴스도 직렬화 가능하다면(Serializable 인터페이스를 구현 하는 클래스의 인스턴스라면), p가 참조하는 인스턴스도 Circle 인스턴스가 직렬화 될 때 함께 직렬화가 된다.
        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);
        }
}
```



줄줄이 사탕으로 엮여 들어갑니다.

```
class SerializableInstMember {
    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);
    }
}
```

직렬화의 대상에서 제외! transient!

```
import java.io.*;
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("");
	}
}
```

직렬화의 대상에서 제외! transient!

```
class TransientMembers
{
    public static void main(String[] args)
                    throws IOException, ClassNotFoundException
    {
        /* 인스턴스 저장 */
        ObjectOutputStream out=
                    new ObjectOutputStream(new FileOutputStream("Personal.ser"));
        PersonalInfo info=new PersonalInfo("John", "baby", 3, 42);
        info.showCirlceInfo();
        out.writeObject(info);
        out.close();

        /* 인스턴스 복원 */
        ObjectInputStream in=
        new ObjectInputStream(new FileInputStream("Personal.ser"));
        PersonalInfo recovInfo=(PersonalInfo)in.readObject();
        in.close();

        /* 복원된 정보 출력 */
        recovInfo.showCirlceInfo();
        }
}
```

FILE 클래스

File 클래스

- **File 클래스**
  - 파일의 경로명을 다루는 클래스
  - 파일 이름 변경, 삭제, 디렉터리 생성, 크기 등 파일 관리
  - 파일 입출력은 파일 스트림을 이용

File 클래스

- **생성자와 경로관련 메서드**
  - 파일과 디렉토리를 다루는데 사용되는 클래스

생성자 / 메서드	설 명
File(String fileName)	주어진 문자열(fileName)을 이름으로 갖는 파일을 위한 File인스턴스를 생성한다. 파일 뿐만 아니라 디렉토리도 같은 방법으로 다룬다. 여기서 fileName은 주로 경로(path)를 포함해서 지정해주지만, 파일 이름만 사용해도 되는 데 이 경우 프로그램이 실행되는 위치가 경로(path)로 간주된다.
File(String pathName, String fileName) File(File pathName, String fileName)	파일의 경로와 이름을 따로 분리해서 지정할 수 있도록 한 생성자. 이 중 두 번째 것은 경로를 문자열이 아닌 File인스턴스인 경우를 위해서 제공된 것이다.
String getName()	파일이름을 String으로 반환한다.
String getPath()	파일의 경로(path)를 String으로 반환한다.
String getAbsolutePath() File getAbsoluteFile()	파일의 절대경로를 String으로 반환한다. 파일의 절대경로를 File로 반환한다.
String getParent() File getParentFile()	파일의 조상 디렉토리를 String으로 반환한다. 파일의 조상 디렉토리를 File로 반환한다.
String getCanonicalPath() File getCanonicalFile()	파일의 정규경로를 String으로 반환한다. 파일의 정규경로를 File로 반환한다.

멤버변수	설 명
static String pathSeparator	OS에서 사용하는 경로(path) 구분자. 윈도우 ";", 유닉스 ":"
static char pathSeparatorChar	OS에서 사용하는 경로(path) 구분자. 윈도우에서는 ';', 유닉스 ':'
static String separator	OS에서 사용하는 이름 구분자. 윈도우 "₩", 유닉스 "/"
static char separatorChar	OS에서 사용하는 이름 구분자. 윈도우 '₩', 유닉스 '/'

File 클래스

- ## 생성자와 경로관련 메서드

메서드	설 명
boolean canRead()	읽을 수 있는 파일인지 검사한다.
boolean canWrite()	쓸 수 있는 파일인지 검사한다.
boolean exists()	파일이 존재하는지 검사한다.
boolean isAbsolute()	파일 또는 디렉토리가 절대경로명으로 지정되었는지 확인한다.
boolean isDirectory()	디렉토리인지 확인한다.
boolean isFile()	파일인지 확인한다.
boolean isHidden()	파일의 속성이 '숨김(Hidden)'인지 확인한다. 또한 파일이 존재하지 않으면 false를 반환한다.
int compareTo(File pathname)	주어진 파일 또는 디렉토리를 비교한다. 같으면 0을 반환하며, 다르면 1 또는 -1을 반환한다. (Unix시스템에서는 대소문자를 구별하며, Windows에서는 구별하지 않는다.)
boolean createNewFile()	아무런 내용이 없는 새로운 파일을 생성한다.(파일이 이미 존재하면 생성되지 않는다.) File f = new File("c:₩₩jdk1.5₩₩work₩₩test3.java"); f.createNewFile();
static File createTempFile(String prefix, String suffix)	임시파일을 시스템의 임시 디렉토리에 생성한다. System.out.println(File.createTempFile("work", ".tmp")); 결과 : c:₩windows₩TEMP₩work14247.tmp
static File createTempFile(String prefix, String suffix, File directory)	임시파일을 시스템의 지정된 디렉토리에 생성한다.
boolean delete()	파일을 삭제한다.
void deleteOnExit()	응용 프로그램 종료시 파일을 삭제한다. 주로 임시파일을 삭제하는데 사용된다.
boolean equals(Object obj)	주어진 객체(주로 File인스턴스)가 같은 파일인지 비교한다. (Unix시스템에서는 대소문자를 구별하며, Windows에서는 구별하지 않는다.)
long length()	파일의 크기를 반환한다.
String[] list()	디렉토리의 파일목록(디렉토리 포함)을 String배열로 반환한다.
String[] list(FilenameFilter filter)	FilenameFilter인스턴스에 구현된 조건에 맞는 파일을 String배열로 반환한다.
File[] listFiles()	디렉토리의 파일목록(디렉토리 포함)을 File배열로 반환한다.
static File[] listRoots()	컴퓨터의 파일시스템의 root의 목록(floppy, CD-ROM, HDD drive)을 반환한다. (예: A:₩, C:₩, D:₩)

File 클래스

- **File 클래스가 지원하는 기능**

  → 디렉토리의 생성과 소멸

  → 파일의 소멸

  → 디렉토리 내에 존재하는 파일이름 출력

- **File 디렉토리 생성의 예**

```
File reDir=new File("C:₩₩YourJava");  //디렉토리 위치 정보
reDir.mkdir(); // 디렉토리 생성
```

- **File 파일 이동의 예**

```
File myFile=new File("C:₩₩MyJava₩₩my.bin");
File reFile=new File(reDir, "my.bin");
myFile.renameTo(reFile);  // 파일의 이동
```

File 클래스 예제

```
import java.io.File;
class FileMove
{
        public static void main(String[] args)
        {
            File myFile=new File("C:₩₩MyJava₩₩my.bin");
            if(myFile.exists()==false)
            {
                System.out.println("원본 파일이 준비되어 있지 않습니다.");
                return;
            }
            File reDir=new File("C:₩₩YourJava");
            reDir.mkdir();
            File reFile=new File(reDir, "my.bin");
            myFile.renameTo(reFile);
            if(reFile.exists()==true)
                System.out.println("파일 이동에 성공하였습니다.");
                else
                System.out.println("파일 이동에 실패하였습니다.");
        }
}
```

File 클래스 예제

```
import java.io.File;
class FileMoveOSIndepen
{
    public static void main(String[] args)
    {
        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 reDir=new File("C:"+File.separator+"YourJava");
        reDir.mkdir();
        File reFile=new File(reDir, "my.bin");
        myFile.renameTo(reFile);
        if(reFile.exists()==true)
            System.out.println("파일 이동에 성공하였습니다.");
        else
            System.out.println("파일 이동에 실패하였습니다.");
    }
}
```

상대경로 기반의 예제 작성

실제 프로그램 개발에서는 절대경로가 아닌 상대경로를 이용하는 것이 일반적이다.
그래야 실행 환경 및 실행위치의 변경에 따른 문제점을 최소화 할 수 있기 때문이다.

```
File subDir1=new File("AAA");
// 현재 디렉토리 기준으로...
File subDir2=new File("AAA₩₩BBB");
// 현재 디렉토리에 존재하는 AAA의 하위 디렉토리인 BBB...
File subDir3=new File("AAA"+File.separator+"BBB");
// AAA₩₩BBB의 운영체제 독립버전...
```

상대경로 기반의 예제

```
import java.io.File;

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());
	}
}
```

상대경로 기반의 예제 2

```
import java.io.File;

class ListFileDirectoryInfo
{
	public static void main(String[] args)
	{
		File myDir=new File("MyDir");
		File[] list=myDir.listFiles();

		for(int i=0; i<list.length; i++)
		{
			System.out.print(list[i].getName());
			if(list[i].isDirectory())
				System.out.println("₩t₩tDIR");
			else
				System.out.println("₩t₩tFILE");
		}
	}
}
```

File 클래스기반의 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);
```