파일과 I/O 스트림

☞ 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 크지않다. 기본적인 입출력의 형태는 동일하다. 그리고 이것이 JAVA의 I/O 스트림이 갖는 장점이다.

자바 스트림의 큰 분류

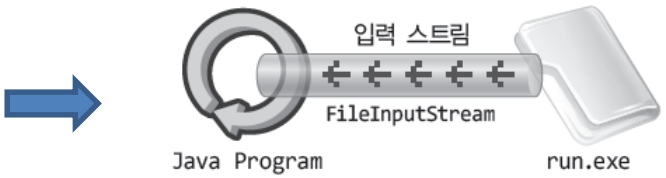
• 입력 스트림(Input Stream)	프로그램으로 데이터를 읽어 들이는 스트림
• 출력 스트림(Output Stream)	프로그램으로부터 데이터를 내보내는 스트림

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

☞ 파일 기반의 입력 스트림 형성

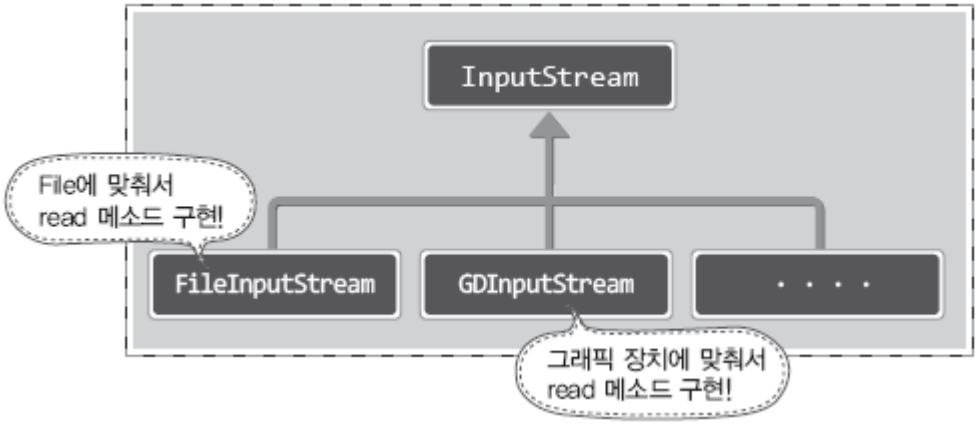
파일 run.exe 대상의 입력 스트림 생성

```
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 대상의 출력 스트림 형성

InputStream	↔	OutputStream
FileInputStream	↔	FileOutputStream

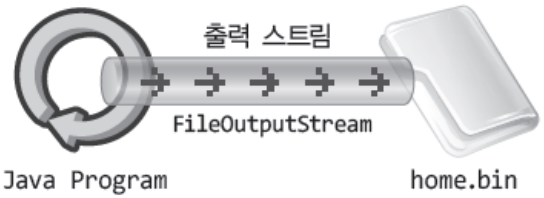
입출력 스트림은 대부분 쌍(Pair)을 이룬다.

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

```
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);
}
```

```
복사된 바이트 크기 85528870
```

위 예제에서는 바이트 단위 복사(1바이트씩 복사)가 진행된다. 따라서 50MB가 넘는 크기의 파일 복사에는 오랜 시간이 걸린다.

☞ 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

```
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);
}
```

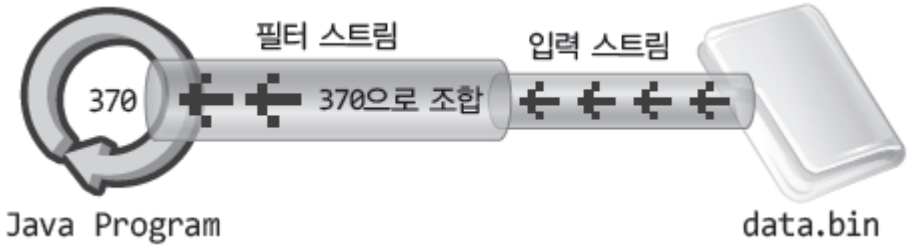
이전 예제와의 가장 큰 차이점은 1KB 크기의 버퍼를 이용해서 데이터를 입출력한다는 점이다.
실제로 속도의 향상을 느낄 수 있다.

☞ 필터 스트림

```
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 대신 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.

☞ 기본 자료형 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 필터스트림이다.

```
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!

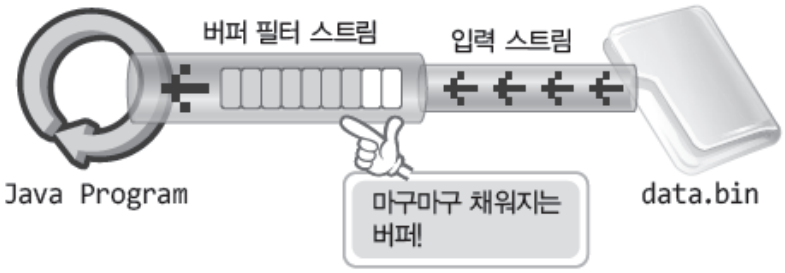
- 필터 입력 스트림 클래스 — FilterInputStream 클래스를 상속한다.
- 필터 출력 스트림 클래스 — FilterOutputStream 클래스를 상속한다.

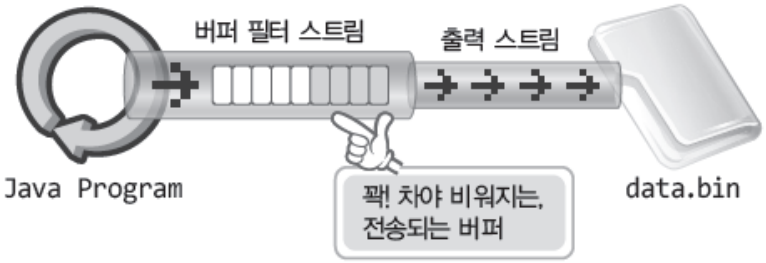
필터 입출력 스트림의 특징, 이를 통해서 필터 스트림 클래스의 구분이 가능

☞ 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 됨으로,
read, write 함수의 호출이 빠르게 진행된다.





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

BufferedInputStream은 입력 버퍼,
BufferedOutputStream은 출력 버퍼 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);
```

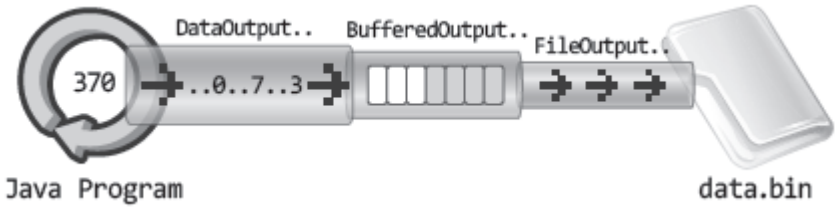
버퍼 스트림

```
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.



생성자와 상속의 관계를 통해서 스트림의 연결 가능성을 확인해야 한다!

☞ 문자 스트림

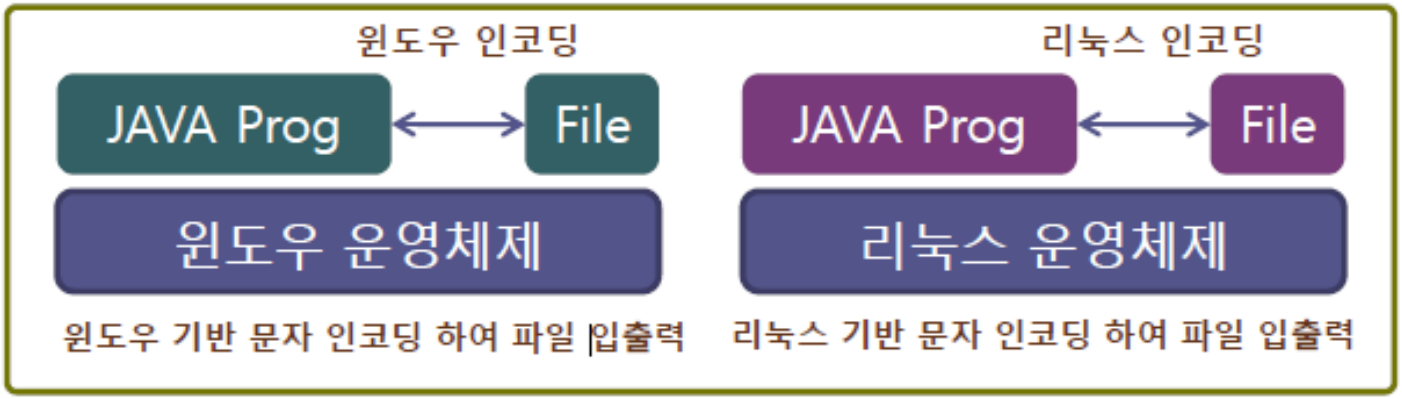
바이트 스트림의 데이터 송수신 특성

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

바이트 스트림을 이용한 파일 대상의 문자 저장의 문제점

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

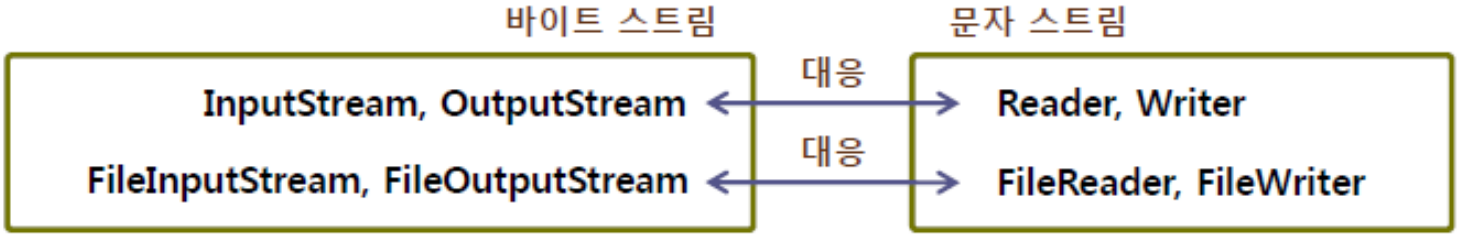
문자 스트림은 해당 운영체제에 따른 인코딩 방식을 지원



문자 스트림은 해당 운영체제의 문자 인코딩 기준을 따라서 데이터입출력 진행

대부분의 문자 스트림은 바이트 스트림과 1대1의 대응을 이룬다!

☞ 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!

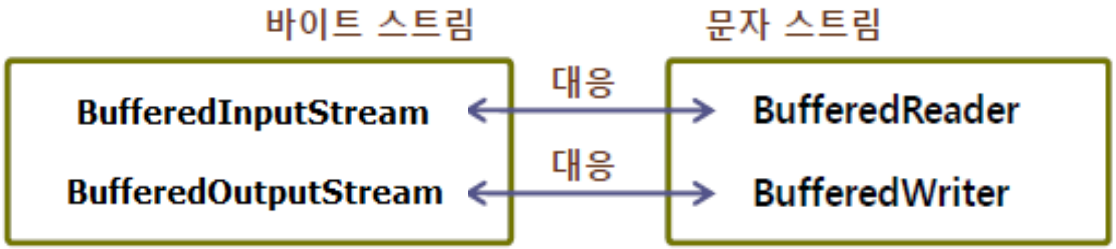
```
public static void main(String[] args) throws IOException
{
    char[] cbuf=new char[10];
    int readCnt;

    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.

☞ 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"));
```

☞ 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

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

실행결과

```
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
```

☞ PrintWriter

- System.out이 PrintStream임을 기억하고, 이 이상으로 PrintStream을 활용하지 않는다.
- printf, println등 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.

```
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();
}
```

문자열에 ₩n이 삽입되었다고 해서 파일 내에서 개행이 이뤄지지는 않는다. 그러나 println 호출 시 시스템에 따른 개행정보가 적절히 삽입된다.

☞ 스트림을 통한 인스턴스의 저장

=> 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);
}
```

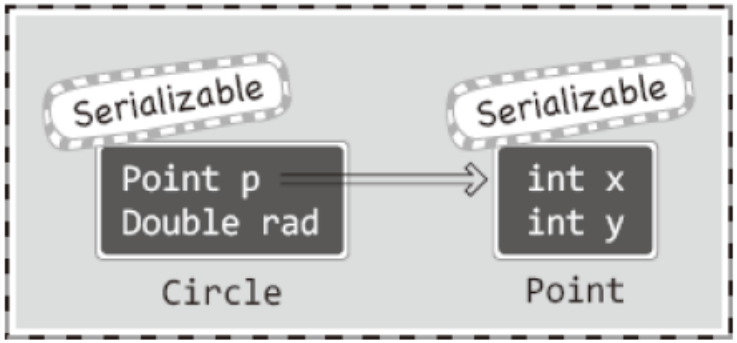
실행결과

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

직렬화 가능!

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

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

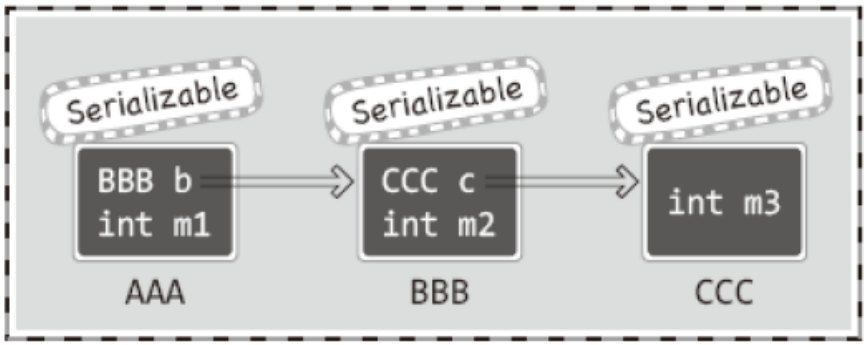


직렬화 가능!

```
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);
    }
}
```

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
```

생성 모드

"r"	읽기 위한 용도
"rw"	읽고 쓰기 위한 용도

```
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();
}
```

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.

☞ 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.