<JSTORM>

파일처리와 네트워크 프로그래밍 - 1

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	파일처리와 네트웍 프로그래밍 - 1
Document file name:	파일처리와네트웍프로그래밍1_장대원.doc
Revision number:	<1.0>
Issued by:	<장대원> dwjang@jkdsoft.co.kr
	정리 : <윤준호> junoyoon@orgio.net
Issue Date:	<2000/8/23 >
Status:	final

Content Information

Audience	마이크로 소프트웨어 독자 자바의 네트웍 및 파일처리 프로그래밍에 대하여 알고 싶어하는 초보 프로그래머
Abstract	네트웍 프로그래밍 - 간단한 Telnet 서버, 클라이언트를 제작해 봄으로써 네트웍 프로그래밍의 기초를 다지고 스트리밍 클래스의 사용법을 익힌다.
Reference	
Benchmark information	

Document Approvals

고문	Signature	date
지위	Signature	date

.

Revision History

Revision	Date	Author	Description of change

Table of Contents

파일 처리와 네트웍 프로그래밍

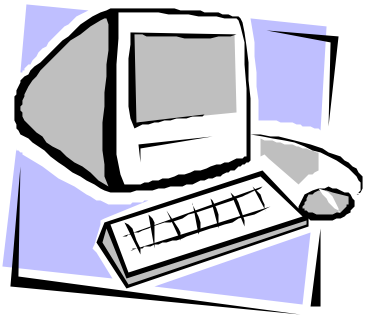
지난달까지 2회에 걸친 GUI 관련 기사로 독자 여러분들은 자바 프로그램을 작성하는 방법에 대해 어느 정도 감을 잡았으리라 생각됩니다. 필자 개인적으로는 독자 여러분들이 지난 기사에서 통해 주제로 선정된 GUI에 관한 내용보다도 프로그램이 어떻게 구상되어지며 또 어떤 과정을 거쳐 구현되어지는지에 대한 구체적인 느낌을 갖게되었다면 더 좋겠습니다.

어떤 클래스라이브러리나 몇몇 메쏘드의 사용법을 익히는 것보다는 프로그램의 설계능력이 훨씬 중요합니다. 게다가 그러한 능력은 쉽게 얻어지는 것도 아닙니다. 구체적인 API에 대한 지식이나 기술이 단편적인 이해 능력을 요구한다면 프로그램 설계 기술은 프로그래머의 종합적인 사고능력을 요구하기 때문입니다.

경험 있는 프로그래머들은 말합니다. “모르는 것은 찾아보면 된다.”고 말입니다. 사실 자바 API들은 문서화가 참 잘 되어 있습니다. 게다가 javadoc 이라는 문서화 툴을 이용하기 때문에 그 형식 또한 아주 일관된 방식을 유지하고 있습니다. 그래서 어느 정도 API 문서에 익숙해지고 나면 별도의 자바 책이 필요 없어질 정돕니다. 여기에 더하여 자바 튜토리얼을 손이 닫는 곳에 두고 있으면 아직 미지의 영역이 남아 있더라도 두둑한 배짱까지 생깁니다. 즉 자신이 모르는 지식이 있더라도 걱정할 필요가 없다는 것이지요.

그러나 실제로 프로그램을 만들어가다 보면 더 큰 벽이 있다는 것을 금새 알 수 있습니다. 특히 프로그래밍 경험이 부족한 초보자들의 경우엔 더욱 심각한 문제입니다. 어디서부터 시작을 해야하며 어떤 구성요소들이 필요하고 또 각 구성 요소들에게 어떠한 임무를 부여해야하는지 도무지 감이 잡히질 않는 것입니다. 초보 프로그래머들이 어느 정도 지식을 갖추고 난 후에도 쓸만한 프로그램을 선뜻 만들어내지 못하는 이유가 바로 그런 것입니다.

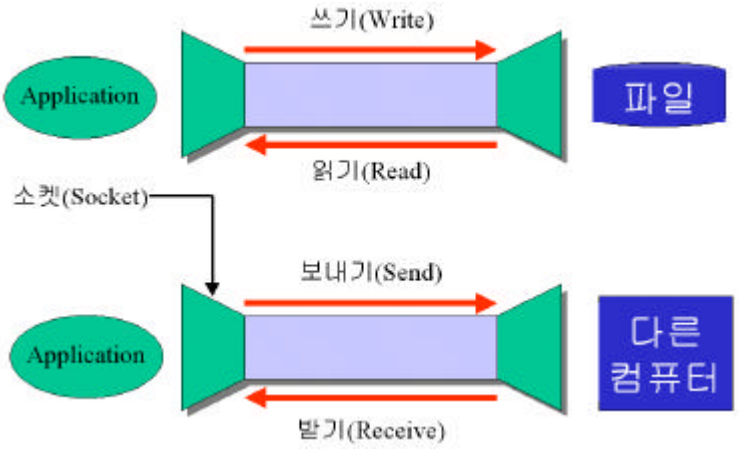
이번 2차 하반기 연재는 그러한 난점들을 해결하는데 가장 큰 목적이 있습니다. 따라서 독자 여러분들도 단순히 필자가 나열하는 지식을 머리 속에 담아두려고만 노력하지 말고 그러한 지식과 병행하여 독자들 나름의 작품을 하나씩 만들어보길 권합니다. 예제 수준을 벗어날 수 있는 정도면 어떤 작품이어도 좋습니다. 필자는 첫 작품으로 간단한 퍼즐 게임을 만들었던 것으로 기억되는군요. "백타가 불여 일작이라"라는 말이 있습니다. 손가락으로 예제 프로그램 백번 쳐보는 것보다 작품 하나를 끝까지 만들어보는 것이 낫다는 얘기입니다. 아직 시작하지 않은 독자가 있다면 지금이라도 당장 자신만의 작품에 도전해보시기 바랍니다.



파일/소켓/RMI (I)

네트웍은 I/O다!

I/O라 하면 파일, 키보드, 프린터 등만을 생각하는 사람들이 많은데 네트웍에 관련된 작업들도 바로 이 입출력이 그 대부분을 차지하고 있습니다. 파일 작업과 네트웍 작업의 공통점이 많다는 것을 눈치챈 선각자들이 이들을 하나로 묶어놓았기 때문입니다. 파일을 쓰고(Read) 읽는(Write) 작업이나 네트웍을 통해 데이터를 보내고(Send) 받는(Receive) 작업이 서로 거의 흡사하다는 이야기지요.([그림 1]) 그래서 자바로 네트웍 프로그래밍을 할 때 절대로 빠지지 않는 것이 바로 입출력입니다. 필자도 네트웍 이야기를 꺼내기 전에 파일 입출력 프로그래밍에 대한 예제를 맛보기로 준비해 두었습니다.



[그림 1] 네트웍은 입출력(I/O)이다

[리스트 1]의 OutputDemo 프로그램은 “Demo.dat”라는 파일에 "Today"라는 문자열을 기록한 다음 다시 현재의 날짜와 시간을 기록하는 프로그램입니다. Date 클래스는 생성자에 아무런 매개변수도 주어지지 않을 때 현재의 날짜와 시간으로 초기화되어 생성됩니

다. 그 외에도 몇몇 낯선 구문이 눈에 띄는군요. 그렇지만 걱정할 필요는 없습니다. 이 글 내내 다뤄질 내용이기 때문에 당장 이해가 가지 않더라도 좋습니다. 두 번째 프로그램인 InputDemo 프로그램은 OutputDemo 프로그램이 이전에 만들어낸 파일로부터 데이터를 읽어들여서 화면에 출력하는 프로그램입니다. 맨 아래쪽에서 출력 결과물을 볼 수 있습니다.

파일에 데이터를 쓰거나 파일로부터 데이터를 읽어들이기 위해서는 I/O를 담당하는 FileOutputStream 객체와 FileInputStream 객체를 사용해야 합니다. 이러한 객체들을 스트림 입출력 객체들이라고 하는데 파일을 다루는 입출력 객체라고 해서 File- 접두어가 붙게 됩니다. (스트림에 대해서는 "소켓이란?" 박스기사 참조)

그런데 추가적인 혹은 서로 다른 입출력 서비스를 사용하기 위해 스트림 객체들이 서로 중첩될 수 있습니다. 단 입력은 입력 계열끼리, 출력은 출력 계열끼리 중첩될 수 있는 것이죠. 예를 들어 [리스트 1]의 [1번]을 보면 바로 위에서 만들어진 FileOutputStream 객체에 ObjectOutputStream을 중첩하여 생성하는 부분이 있습니다. 이때 ObjectOutputStream은 파일에 String형 또는 Date 형 객체를 저장하기 위해 사용되어집니다. 병 속에 어떤 특별한 물질만 넣으려고 서로 다른 종류의 두 깔때기를 중첩해서 사용할 수 있는 것과 비슷한 이치입니다. java.io 패키지에는 이러한 깔때기가 무척 많습니다. 이들 중 몇몇은 이 글에서 다루게 되는데 잘 정리해 두었다가 자신의 입맛에 맞는 깔때기를 골라서 적소에 사용하기 바랍니다.

두 번째 프로그램의 입력 부분도 많이 다르지 않습니다. 잘 보면 두 개의 프로그램이 매우 유사한 구조로 되어 있음을 알 수 있습니다. 입출력이라는 것이 서로 대칭 구조를 가지고 있기 때문이죠. 그럼 이 정도 선에서 맛보기 프로그램에 대한 설명을 마치고 입출력과 네트웍에 관한 구체적인 이야기를 꺼내보겠습니다. 아직 풀리지 않는 의문들은 머리 속에 그대로 담아 두시고 계속해서 글을 읽어나가기 바랍니다.

[리스트 1] 파일 입출력 예제

```
// 파일 출력 예제
import java.io.*;
import java.util.*;

public class OutputDemo {

   public static void main(String args[]) {
      try {
      FileOutputStream fos = new FileOutputStream("Demo.dat");
      ObjectOutputStream oos = new ObjectOutputStream(fos);   <--- [1번]
```

```
        oos.writeObject("Today");
        oos.writeObject(new Date());

        oos.flush();
        fos.flush();
        oos.close();
        fos.close();
        } catch(IOException e) { e.printStackTrace(); }
    }
}
```

// 파일 입력 예제

```
import java.io.*;
import java.util.*;

public class InputDemo {

    public static void main(String args[]) {
        try {
        FileInputStream fis = new FileInputStream("Demo.dat");
        ObjectInputStream ois = new ObjectInputStream(fis);

        String today = (String)ois.readObject();
        Date date = (Date)ois.readObject();

        ois.close();
        fis.close();

        System.out.println(today);
        System.out.println(date);
        }
        catch(IOException e) { e.printStackTrace(); }
        catch(ClassNotFoundException e) { e.printStackTrace(); }
    }
}
```

// 출력 결과

```
Today
Tue Aug 03 18:06:07 GMT+09:00 1999
```

텔넷 서버를 만들자

이 달에 만들어 볼 프로그램은 아주 간단한 텔넷 프로그램입니다. 네트웍 강좌에 감초처럼 등장하는 프로그램이 텔넷 프로그램인데요, 사람들이 많이 사용하는 프로그램이고 또 구현하는데도 복잡하지 않기 때문에 자주 예제로 등장하는 것 같습니다. 하나의 프로그램이라고 하기엔 너무 규모가 작을지도 모르겠군요. 좀 더 큰 프로그램은 다음 달에 만날 수 있을 것입니다. 이번 달에 기초를 제대로 쌓아놓은 다음 다음달에는 썩 쓸만한 프로그램을 만들어보는 기회를 갖겠습니다.

텔넷 프로그램은 알다시피 서버와 클라이언트 이렇게 두 개의 영역으로 분리되어 있습니다. 따라서 우리는 두 개의 프로그램을 만들어야합니다. 먼저 서버를 만들어보도록 하겠습니다. 텔넷 서버 프로그램은 클라이언트가 보내오는 문자열을 받아서 클라이언트가 어떤 명령을 내렸는지 해석해야 합니다. 따라서 미리 어떤 문자열과 그 문자열에 해당하는 명령처리부분을 매핑(mapping)시켜 놓아야합니다. 매핑 과정은 단순성을 기하기 위해 if - else if - else 구문으로 처리하겠습니다. 그럼 우리의 텔넷 서버 프로그램이 어떤 명령어들을 지원해야 하는지 정의하고 넘어갈까요? 역시 프로그램의 단순성을 위해 텔넷 서버가 인식하는 명령어를 3개로 제한하겠습니다. ([표 1]) 그럼 텔넷 서버가 만들어지는 과정을 하나하나 살펴보겠습니다.

[표 1] 텔넷 서버가 인식하는 명령과 서버의 반응

서버가 인식하는 명령	서버의 반응
로그인	클라이언트의 ID와 비밀번호를 받아들인다.
로그아웃	클라이언트와의 접속을 끊는다.
인사	클라이언트에게 답례 인사를 한다.

[리스트 2] 텔넷 서버 프로그램

```
import java.io.*;
import java.net.*;

////////////////////////////////////////////////////////////////////
public class TelnetServer {

   private ServerSocket m_sSocket;

   // ----------------------------------------------------------------
   public TelnetServer() {
      try {
      // 23번 포트, 최대 10명의 클라이언트 수용
      m_sSocket = new ServerSocket(23, 10);
```

```
            while (true) {
                Socket cSocket = m_sSocket.accept();                    <--- [1번]
                TelnetConnection connection = new TelnetConnection(cSocket);
            }
            } catch (IOException ex) { ex.printStackTrace(); }
        }

        // -----------------------------------------------------------------
        public static void main(String[] args) {
            new TelnetServer();
        }
    }

    ////////////////////////////////////////////////////////////////
    class TelnetConnection extends Thread {

        private Socket m_cSocket;
        private DataInputStream m_fromClient;
        private PrintStream m_toClient;

        // -----------------------------------------------------------------
        public TelnetConnection(Socket socket) {
            m_cSocket = socket;

            try {
         // <--- [2번]
            m_fromClient = new DataInputStream(m_cSocket.getInputStream());
            m_toClient = new PrintStream(m_cSocket.getOutputStream());
            } catch (IOException ex) { ex.printStackTrace(); }

            start();
        }

        // -----------------------------------------------------------------
        public void run() {
            String clientID;
            String command;

            try {
            m_toClient.println("Welcome to Simple Telnet Service!!!");
            m_toClient.print("Login ID: ");
```

```
            clientID = m_fromClient.readLine();
            m_toClient.print("Password: ");
            m_fromClient.readLine();
            m_toClient.println("You're valid user!!!\n\n\n");

            while (true) {                                          <--- [3번]
                m_toClient.print("\nprompt> ");
                command = m_fromClient.readLine();

                if (command.equals("hi")) {
                    m_toClient.println("Hi~ Mr. " + clientID);
                    continue;
                }

                if (command.equals("bye")) {
                    m_toClient.println("See you again Mr. " + clientID);
                    break;
                }

                m_toClient.println("Invalid command! Type again");
            }
            } catch (IOException ex) { ex.printStackTrace(); }
            finally {
                try {
                    m_cSocket.close();
                }
                catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }

        // ------------------------------------------------------------
        public static void main(String[] args) {
            new TelnetServer();
        }
    }
```

텔넷 서버는 다시 크게 두 부분으로 나뉘어집니다. 클라이언트의 접속을 받아들이는 “접속 부분”과 클라이언트의 명령을 처리하는 “서비스 부분” 이렇게 두 부분으로 나뉘어지지요. 먼저 접속 부분을 살펴보겠습니다. 접속 부분은 매우 단순합니다. 단순히 클

라이언트가 접속할 때까지 대기(wait)하고 있다가 클라이언트가 접속하자마자 접속된 클라이언트를 서비스 부분에 연결시켜 주는 것이 전부입니다. ([리스트 2]의 [1번])

```
Socket cSocket = m_sSocket.accept();
```

바로 이 부분이 서버가 클라이언트의 접속을 대기하는 부분인데요, 서버측 소켓(ServerSocket)의 accept() 메쏘드는 클라이언트가 접속할 때까지 대기하고 있다가 클라이언트가 접속하면 접속한 클라이언트의 소켓을 리턴해줍니다. (소켓에 대한 내용은 박스기사를 참조하세요) 일단 클라이언트의 소켓을 가지고 있으면 클라이언트에게 메시지를 전송하거나 또는 클라이언트로부터 메시지를 받아들일 수 있습니다.

그런데 서버가 하나라고 해서 클라이언트도 하나만 접속하는 것은 아닙니다. 여러 클라이언트가 접속할 수도 있다는 가정 하에 프로그램을 작성하는 것이 바람직하겠지요. 게다가 여러 클라이언트들이 동시에 서비스를 요구할 수도 있다는 점을 잊어서는 안됩니다. 그럼 이 문제를 어떻게 해결해야 할까요?

이 문제는 멀티쓰레드를 사용하여 해결 할 수 있습니다. 각각의 클라이언트에 대해 따로따로 서비스 쓰레드를 할당해 주는 것이지요. (독자 여러분들이 쓰레드에 대한 기본 지식은 가지고 있다는 가정 하에 설명을 이어가겠습니다.) 이런 식으로 말입니다.

```
TelnetConnection connection = new TelnetConnection(cSocket);
```

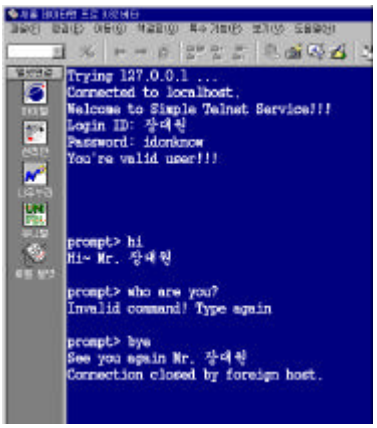
TelnetConnection 클래스가 바로 클라이언트의 요청에 응답하는 서비스 클래스입니다. 생성자는 클라이언트의 소켓 하나를 매개변수로 받아들입니다. TelnetConnection 객체가 생성된 후에는 그 클라이언트에게만 서비스를 제공하는 것이죠. 그런데 이 클래스의 정의 부분을 보면 Thread 클래스를 상속(extends) 하고 있다는 것을 알 수 있습니다. 따라서 각각의 클라이언트에 대해 동시에 여러 서비스 객체가 따로따로 서비스를 제공할 수 있는 것이지요.

지금까지 텔넷 서버의 골격을 살펴보았는데 이제는 실제로 어떻게 클라이언트의 요청을 받아들이고 명령을 처리하는지 살펴보겠습니다. 앞서 네트웍 작업이 파일 입출력과 다를 것이 없다는 이야기를 했었습니다.[리스트 1]에서는 파일에 데이터를 쓰거나 파일로부터 데이터를 읽어들이기 위해 I/O를 담당하는 FileOutputStream 객체와 FileInputStream 객체를 사용하였습니다. 네트웍을 통해 데이터를 보내거나(쓰거나) 받을(읽을)때에도 스트림 클래스들을 얻어오는 것이 가장 먼저 할 일입니다. 이러한 스트림 클래스들은 소켓으로부터 직접 얻어올 수 있습니다. ([리스트 2]의 [2번]) 소켓은 getInputStream()과 getOutputStream() 메쏘드의 반환 값으로 입출력 스트림 객체를 리턴합니다. 그리고 앞서 말했듯이 DataInputStream, PrintStream 클래스는 추가적인 입출력 서비스를 이용하기 위해 원래의 InputStream, OutputStream 객체에 중첩되었습니다.

이렇게 입출력 스트림을 얻어 왔고 또 추가로 필요한 서비스를 위해 스트림 객체를 중첩까지 해 두었으니 이제부터는 파일 작업을 하듯이 스트림 객체에 읽고 쓰면 됩니다. 이제 텔넷 서버가 할 일은 입력 스트림으로부터 문자열을 읽어들인 다음 문자열에 해당하는 명령을 처리하여 그 결과를 다시 클라이언트에게 출력 스트림을 통해 전달하는 것입니다.

[3번]의 while 루프 안에서 그러한 작업을 수행하게 되는데요, 사용자가 "hi" 라는 문자열을 입력했을 때는 "Hi~ Mr. + 클라이언트 ID" 이런 식으로 답례를 하게 했습니다. 이것은 서버와 클라이언트간의 일종의 약속이지요. 다들 아시겠지만 네트웍상에서 지켜지는 이러한 약속을 보통 프로토콜(protocol)이라고 하지요. 네트웍 프로그램에서 프로토콜은 아주 중요한 역할을 하게됩니다. 이 프로토콜에 맞춰서 실제의 통신이 이루어지기 때문이죠. 그럼 지금까지 살펴본 텔넷 서버 프로그램을 간단히 정리해 보겠습니다.

서버소켓(ServerSocket)을 하나 생성한 다음 클라이언트의 접속을 기다린다. 클라이언트가 접속하면 클라이언트에 대한 소켓 객체를 얻을 수 있는데 이 소켓으로부터 다시 입출력 스트림 객체를 얻어낸다. 파일 작업을 하듯이 입출력 스트림 객체를 통해 메시지를 보내거나 받는다.



[화면 1] 텔넷 서버와 새롬 데이터맨과의 통신

텔넷 클라이언트를 만들자

텔넷 클라이언트 프로그램도 서버와 거의 똑같은 구조를 이루고 있습니다. "접속 부분"과 "메시지를 주고받는 부분" 이렇게 두 파트로 이루어져 있지요.

```
m_sSocket = new Socket("localhost", 23);
```

[리스트 3]의 [1번]을 보면 서버에 연결하는 부분이 나오는데 Socket 클래스의 생성자는 서버의 주소와 포트번호를 매개변수로 받아서 서버에 연결한 다음 서버에 대한 소켓을 넘겨줍니다. 그 다음부터는 앞서 보았던 서버 프로그램과 거의 같습니다. 즉, 소켓으로부터 입출력 스트림 객체를 얻어온 다음 이 입출력 스트림 객체에 쓰거나 읽는 과정으로 통신이 이루어집니다.

클라이언트 측에서는 쓰는 작업과 읽는 작업이 동시에 이루어질 수 있기 때문에 두 작업에 대해 별도의 쓰레드가 생성될 필요가 있습니다. [리스트 3]에서는 쓰는 작업, 즉 서버에 명령을 전달하는 작업은 메인 쓰레드가 맡고, 읽는 작업, 즉 서버로부터의 메시지를 화면에 나타내는 작업은 별도의 쓰레드를 통해 이루어지도록 했습니다. [2번]의 ServerMessage 클래스가 바로 서버로부터의 메시지를 읽어들여 화면에 뿌리는 일을 하는 클래스인데 Thread 클래스를 상속받아서 별도의 쓰레드로 수행된다는 것을 알 수 있습니다.

서버와 클라이언트의 구조가 거의 같고 사용하는 클래스들과 API 메쏘드들도 모두 같기 때문에 더 이상의 설명은 생략하겠습니다. 한가지 클라이언트측 프로그램을 컴파일하면 deprecation 경고 메시지를 만나게 될텐데 DataInputStream 클래스의 readLine() 메쏘드가 소켓의 기본 단위인 byte 열을 character 열로 제대로 변환하지 못하기 때문에 그 사실을 경고하는 것입니다. 자바가 유니코드 체계를 사용하기 때문이죠. 한글을 제대로 보시기 위해서는 아래와 같이 바꾸어야 합니다.

```
DataInputStream d = new DataInputStream(in);를
BufferedReader d = new BufferedReader(new InputStreamReader(in));로 바꿔야합니다.
```

좀 더 자세히 알고 싶은 독자는 "초보 프로그래머의 디버깅 노트"를 참조하기 바랍니다.

[리스트 3] 텔넷 클라이언트 프로그램

```
import java.io.*;
import java.net.*;

////////////////////////////////////////////////////////////////
public class TelnetClient {

    private Socket m_sSocket = null;
    private DataInputStream m_fromServer = null;
    private PrintStream m_toServer = null;
    private DataInputStream m_fromUser = null;

    // ----------------------------------------------------------------
    public TelnetClient() {
        try {
        m_sSocket = new Socket("localhost", 23);              <-- [1번]
        m_fromServer = new DataInputStream(m_sSocket.getInputStream());
        m_toServer = new PrintStream(m_sSocket.getOutputStream());
        m_fromUser = new DataInputStream(System.in);
        } catch (IOException ex) { ex.printStackTrace(); }

        StartCommunication();
    }

    // ----------------------------------------------------------------
    private void StartCommunication() {
        (new ServerMessage()).start();

        String msg;
        try {
        while (true) {
            msg = m_fromUser.readLine();
            m_toServer.println(msg);
        }
        } catch (IOException ex) { ex.printStackTrace(); }
        finally {
            try {
                m_sSocket.close();
                System.exit(0);
            }
```

```
            catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }



    ////////////////////////////////////////////////////////////////
    class ServerMessage extends Thread {                    <--- [2번]

        // ------------------------------------------------------------
        public void run() {
            String msg;

            try {
            while (true) {
                msg = m_fromServer.readLine();
                System.out.println(msg);
            }
            } catch (IOException ex) { ex.printStackTrace(); }
            finally {
                try {
                    m_sSocket.close();
                    System.exit(0);
                }
                catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }
    }

    // ------------------------------------------------------------
    public static void main(String[] args) {
        new TelnetClient();
    }
}
```

I/O에 대해 체계적으로 정리하자.

이번 달에는 이론에 앞서 실질적인 예제를 먼저 살펴보았습니다. 내용 자체가 자칫 지루할 수 있는 것들이라 필자가 일부러 그렇게 구성해 보았습니다. 앞의 예제들을 보고 "프로그램이 그런 식으로 돌아가는구나!"라는 정도만 알고 있어도 괜찮은 성과라 생각됩니다. 이제 여기부터는 좀 더 체계적으로 자바의 입출력 클래스들을 살펴보겠습니다.

자바가 제공하는 입출력 클래스들은 그 종류가 참으로 다양합니다. 그리고 그 이름들도 서로 비슷비슷해서 처음 보는 사람들은 많이 헤매기도 합니다. 이게 그거 같고 그게 이거 같고 그렇거든요. 그렇지만 전체적인 체계를 이해하고 있으면 의외로 단순하다는 것을 느낄 수 있을 것입니다. 사실 양만 많지 서로 비슷비슷한 구조를 가지고 있기 때문에 전체 구조를 익힌 다음 구체적인 클래스들 중 하나 둘만 제대로 익히고 나면 의외로 쉽게 접근할 수 있습니다. 그럼 본격적으로 입출력 클래스들을 살펴볼까요?

입출력 클래스들은 크게 입력과 출력 이렇게 두 가지 부류로 나눌 수 있습니다. 그런데 이 것은 너무도 당연한 이야기니 특별한 설명은 하지 않겠습니다. 입출력 클래스들을 분류하는 또 다른 기준이 있습니다. 바로 바이트 스트림(Byte Stream)과 캐릭터 스트림(Character Stream) 이렇게 두 가지의 기준이죠. 바이트 스트림 ([그림 2]) 계열의 클래스들은 데이터를 입출력의 기본 단위인 바이트 단위로 작업하기 위한 것이고 캐릭터 스트림 ([그림 3]) 계열의 클래스들은 자바가 사용하는 문자 셋인 유니코드를 위해 개발된 것입니다. 자바에서는 하나의 문자를 2바이트로 처리하기 때문에 한 바이트씩 처리하는 바이트 스트림 계열의 클래스들은 문자열을 제대로 처리할 수가 없습니다. 그럼 이 두 부류의 스트림 클래스들을 좀 더 자세히 살펴볼까요?

바이트 스트림 클래스들

바이트 스트림 클래스들은 [그림 2]와 같은 상속 계층을 이루고 있습니다. 이 많은 클래스들 중에서 가장 기본이 되는 클래스는 상속 계층의 맨 위쪽에 자리하는 InputStream과 OutputStream 클래스입니다. 이 두 클래스는 추상 클래스(Abstract Class)인데 각각 [표 1]과 [표 2]와 같은 메쏘드들을 정의하고 있습니다. InputStream 클래스의 메쏘드들 중에서 조금 생소한 mark(), reset() 메쏘드에 대해서는 추가로 약간 더 설명 드리겠습니다.

스트림이라는 것은 일련의 바이트 배열입니다. 그리고 이 바이트 배열은 큐(Queue)에 들어 있다고 생각하면 이해하는데 편리할 것입니다. 바이트 스트림 클래스들은 큐에 들어 있는 바이트 열을 순서대로 하나씩 또는 몇 바이트씩 빼내는 역할을 하고 있죠. 큐라는 것에 주목할 필요가 있습니다. 큐에서는 한번 나온 데이터가 다시 원래의 위치에 들어가지 못한다는 사실을 독자 여러분들도 알고 있을 것입니다. 다시 데이터를 집어넣으려면 큐의 맨 꼬리부분에 추가시키는 방법 밖에는 없지요. 이러한 불편함을 해결하기 위해 마

크(mark), 리셋(reset) 메쏘드가 개발되었는데요, 일단 마크하고 나면 마크된 위치부터 앞으로 읽어들이는 바이트를 모두 어떤 버퍼에 저장해 놓습니다. 그런 다음 리셋을 호출하면 원래의 스트림에서는 뒤로 되돌아갈 수가 없기 때문에 앞서 저장해 두었던 버퍼로 작업을 하는 것이지요. 나머지 내용들도 표를 통해 정리해 두기 바랍니다.

[표 1] (바이트 스트림) InputStream 클래스의 메쏘드들

메쏘드	기능
int read()	한 바이트를 읽어 그 값을 리턴합니다. (0-255 사이 값)
int read(byte[] b)	읽어들일 수 있는 몇 바이트를 읽어들여 b에 저장하고 읽어들인 바이트 수를 리턴합니다. (단, b의 길이 이하로 읽어들입니다.)
int read(byte[] b, int off, int len)	읽어들일 수 있는 몇 바이트를 읽어들여 b에 저장하고 읽어들인 바이트 수를 리턴합니다. (단, b에 저장할 때 b의 off 위치부터 len 길이 만큼만 읽어들여 저장합니다.)
long skip(long n)	n개의 바이트를 무시하고 건너뛴 다음 실제로 건너뛴 바이트 수를 리턴합니다.
int available()	블로킹(blocking)되지 않고 읽을 수 있는 바이트 수를 리턴합니다.
void close()	스트림을 닫고 스트림에 관련된 자원들을 시스템에 반환합니다.
void mark(int readlimit)	스트림상의 현재 위치를 마크해 놓습니다. 나중에 reset() 메쏘드를 호출하면 스트림의 마크된 위치로 되돌아갑니다. 이렇게 하기 위해서는 마크한 다음부터 읽어들이는 바이트열을 어딘가에 저장해 놓아야하는데 그 버퍼의 크기를 매개변수 readlimit라 생각하면 되겠습니다. 따라서 마크한 다음 readlimit 이상의 바이트 수를 읽어들였다면 이전에 설정한 마크는 효력을 상실합니다.
void reset()	스트림 상에서 이전에 마크한 위치로 돌아갑니다.
boolean markSupported()	이(this) InputStream 객체가 마킹을 지원하는지 알아냅니다. 어떤 상황에서는 마킹할 수 없을 수도 있습니다.

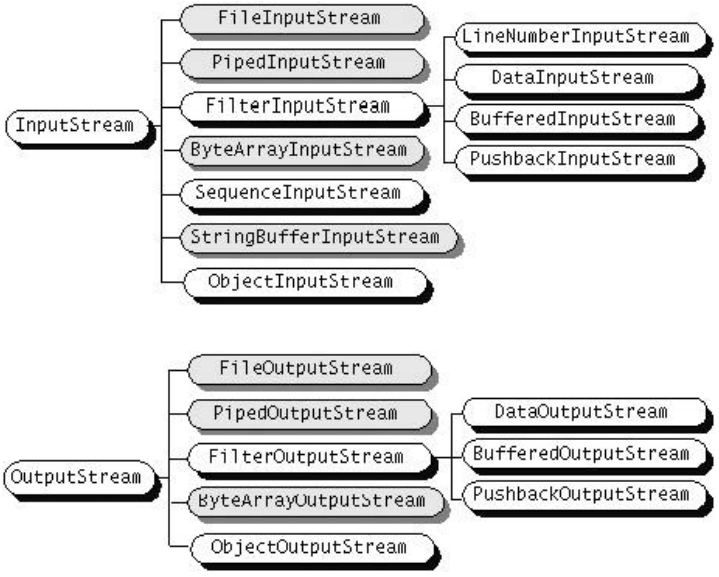
[표 2] (바이트 스트림) OutputStream 클래스의 메쏘드들

메쏘드	기능
void write(int b)	한 바이트를 씁니다.
void write(byte[] b)	b에 저장된 바이트 수만큼 씁니다.
void write(byte[] b, int off, int len)	b의 off번째 바이트부터 시작해서 len 수만큼 씁니다.
void flush()	버퍼에 아직 쓰지 않고 남아있는 모든 바이트를 일시에 씁니다.
void close()	스트림을 닫고 스트림에 관련된 자원들을 시스템에 반환합니다.

방금 InputStream 클래스와 OutputStream 클래스에 대해 살펴보았지만 독자 여러분들이 실제로 사용하는 클래스들은 두 클래스의 자식 클래스들입니다. 이 자식 클래스들은 모두 용도가 다른데 그 용도에 따라 추가적인 메쏘드들을 포함하거나 부모 클래스의 메쏘드를 오버라이드하고 있지요. 예를 파일 작업이 필요할 때는 FileInputStream, FileOutputStream 클래스를 사용하며 객체를 저장하거나 읽어들이기 위해서는 ObjectInputStream, ObjectOutputStream 클래스를 사용할 수 있습니다. 게다가 파일 작업과 객체 작업을 동시에 해야할 경우 두 스트림 객체를 중첩할 수 있다는 것도 필자가 글의 앞부분에서 언급했었지요.

이 글의 맨 앞 부분에서 보았던 예제([리스트 1])를 다시 한번 살펴보기 바랍니다. InputStream, OutputStream 클래스의 자식 클래스들을 어떤 식으로 사용할 수 있는지 좀 더 잘 알 수 있을 것입니다. 여기에 더하여 FileInputStream 객체나 ObjectInputStream 객체 모두 [표 1]의 메쏘드를 사용할 수 있다는 것을 염두에 두기 바랍니다. 자식 클래스들 하나하나에 대한 구체적인 내용은 생략하도록 하겠습니다. 대신 "마소 Jr 4월호 47쪽에 있는 [표 1], [표 2]"를 참조하여 간략히 정리해 두기 바랍니다. 물론 구체적인 메쏘드들에 대해서도 표준 API 도큐먼트 등을 통해 정리하는 것을 잊지 마시고요.

[그림 2] 바이트 스트림 계열의 입출력 클래스들



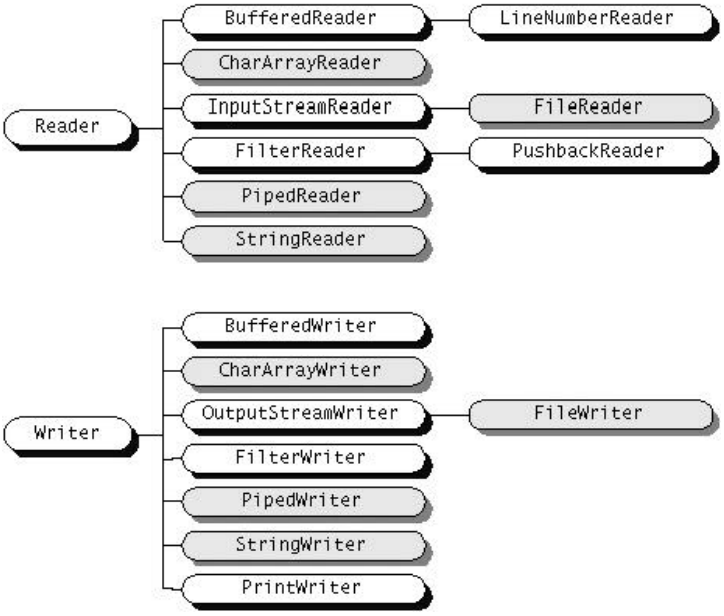
캐릭터 스트림 클래스들

캐릭터 스트림 클래스들은 2바이트의 문자를 다룬다는 것을 제외하고는 바이트 스트림 클래스들과 거의 비슷한 특성을 가지고 있습니다. 이름 또한 비슷한 클래스들이 많지요. ([그림 3]) 그래서 캐릭터 스트림에 대해서는 간략히만 살펴보고 넘어가도록 하겠습니다.

캐릭터 스트림의 계보에서도 가장 중요한 두 개의 최상위 부모 클래스가 있는데 Reader

와 Writer 클래스입니다. 이 두 클래스는 앞서 살펴보았던 InputStream, OutputStream 클래스와 거의 비슷합니다. 정의하고 있는 메쏘드도 거의 비슷해서 [표 1]과 [표 2]에서 보았던 메쏘드들을 고스란히 가지고 있다고 생각해도 좋습니다. 단 매개변수로 사용되는 데이터 타입이 byte가 아니라 char이라는 것만 다를 뿐입니다. 따라서 별도로 정리할 필요는 없을 것 같군요.

[그림 3] 캐릭터 스트림 계열의 입출력 클래스들



데이터 싱크 스트림(Data Sink Stream)

스트림 클래스들을 분류하는 또 다른 기준이 있습니다. 바로 "데이터 싱크 스트림"과 "데이터 프로세싱 스트림"으로 분류할 수 있지요. 데이터 싱크 스트림은 스트림 객체가 사용될 대상에 따라 분류하는 것입니다.([표 3]) 예를 들어 파일에 대한 스트림 클래스와 메모리에 대한 스트림 클래스 등을 서로 구분해 놓은 것이죠. 싱크라는 것은 메모리나 파이프 또는 파일과 같이 스트림 객체가 사용될 대상이라 생각하면 되겠습니다.

데이터 프로세싱 스트림(Data Processing Stream)

데이터 프로세싱이란 입출력되는 데이터에 대해 어떤 작용(Operation)을 가하는 것을 말합니다.([표 4]) 예를 들어 입출력 속도를 높이기 위해 버퍼링을 할 수 있겠지요. 혹은 데이터를 선별하기 위해 특별한 필터링을 할 수도 있습니다. 이렇게 입출력 데이터에 대해 어떤 작용이 필요할 때는 [표 4]에서 자신에게 필요한 스트림 클래스를 선택할 수 있습니다.

Sink Type	캐릭터 스트림 계열	바이트 스트림 계열
메모리	CharArrayReader, CharArrayWriter	ByteArrayInputStream, ByteArrayOutputStream
	StringReader, StringWriter	StringBufferInputStream
파이프(Pipe)	PipedReader, PipedWriter	PipedInputStream, PipedOutputStream
파일	FileReader, FileWriter	FileInputStream, FileOutputStream

[표 3] 데이터 싱크 스트림 클래스들

작업 내용(Process)	캐릭터 스트림 계열	바이트 스트림 계열
버퍼링	BufferedReader, BufferedWriter	BufferedInputStream, BufferedOutputStream
필터링	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream
바이트와 캐릭터의 상호 변환	InputStreamReader, OutputStreamWriter	
스트림들의 연결		SequenceInputStream
객체 시리얼라이즈		ObjectInputStream, ObjectOutputStream
데이터의 변환		DataInputStream, DataOutputStream
개수 세기	LineNumberReader	LineNumberInputStream
스택형 작업(Peeking Ahead)	PushbackReader	PushbackInputStream
출력	PrintWriter	PrintStream

[표 4] 데이터 프로세싱 스트림 클래스들

필터 스트림의 활용

그럼 마지막으로 [그림 2]와 [표 4]에 나와 있는 필터 스트림 클래스들을 활용한 예제를 하나 살펴보겠습니다. 스트림 클래스를 이용하는 주요 목적 중 하나가 필터링이므로 이 예제를 머리 속에 잘 담아두면 스트림 클래스에 대한 응용력이 한층 확고해 지리라 믿습니다.

이 프로그램은 키보드로(Standard input) 입력된 문자열을 모두 대문자로 바꿔서 출력하는 프로그램입니다. 이때 각 라인별로 줄 번호를 앞에 출력하도록 했습니다. ([화면 2]) 입

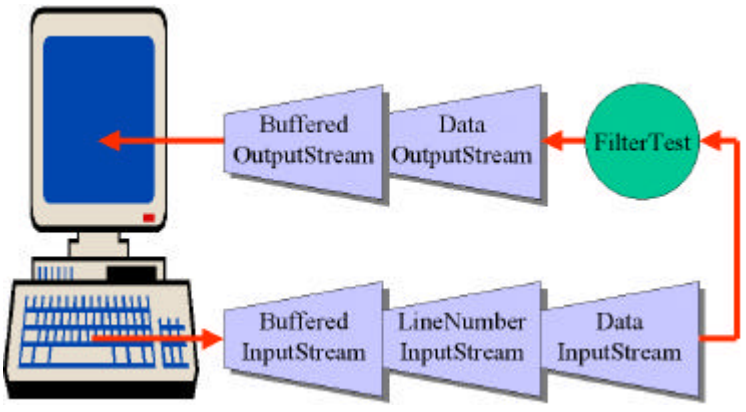
출력 속도의 향상을 위해 버퍼링을 사용하기도 합니다. 키보드로 입력하였을 때 입력을 종료하기 위해 Ctrl-Z(유닉스에서는 Ctrl-D)를 누르세요. 키보드로 입력하지 않을 경우엔 다음과 같이 실행해도 좋습니다.

```
java FilterTest < xxx.txt
```

이 프로그램은 필터 스트림 클래스의 활용에 관한 예제로서 세 종류의 필터 스트림 클래스를 사용합니다. ([리스트 4]) 맨 먼저 버퍼링을 위해 BufferedInputStream, BufferedOutputStream 이렇게 두 개의 클래스를 사용합니다. 그 다음 줄 번호를 매기기 위해 LineNumberInputStream 클래스를 사용하며 마지막으로 몇 가지 편리한 입출력 메쏘드를 사용하기 위해 DataInputStream, DataOutputStream 이렇게 두 개의 클래스를 사용합니다.

DataInputStream과 DataOutputStream 이 두 클래스는 자바의 기본 타입으로 데이터 입출력이 가능합니다. 예를 들어 readInt() 이런 메쏘드가 있어서 4바이트 정수 값을 한꺼번에 읽어올 수 있지요. 그 외에도 모든 기본 타입의 데이터 입출력이 가능하며 텍스트 한 줄의 입출력도 가능합니다. FilterTest 예제에서는 한 줄씩 텍스트를 읽어오는 목적으로 사용되었습니다. (readLine() 메쏘드 통해) 이 두 클래스는 이렇게 자바 기본 타입을 지원하기 때문에 그 활용 빈도가 무척 높습니다. 굳이 바이트 단위로만 작업해야할 일이 아니라면 이 클래스를 활용하여 쉽게 입출력 작업을 할 수 있습니다.

FilterTest 예제의 작동 방식은 [그림 4]에 보이는 바와 같습니다. 입력과 출력시에 여러 입출력 클래스들을 중첩하여 각 클래스에 맞는 필터링을 하는 것이 큰 줄기입니다. 소스 코드를 보면서 실제로 어떻게 필터 스트림들이 중첩되는지 잘 익혀 두기 바랍니다.



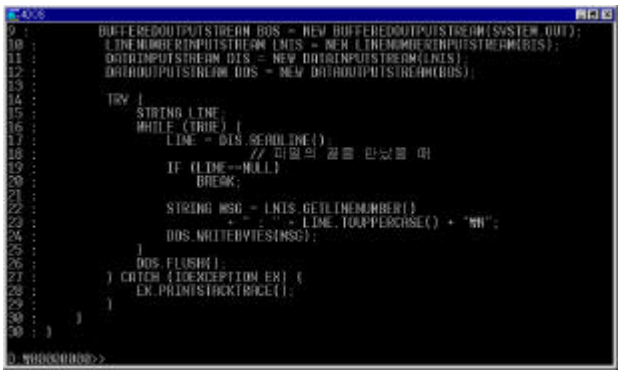
[그림 4] 필터링 스트림 테스트 예제의 작동 방식

[리스트 4] 필터링 스트림 테스트 예제

```
import java.io.*;

////////////////////////////////////////////////////////////////
public class FilterTest {

   // ---------------------------------------------------------
   public static void main(String args[]) {
      BufferedInputStream bis = new BufferedInputStream(System.in);
      BufferedOutputStream bos = new BufferedOutputStream(System.out);
      LineNumberInputStream lnis = new LineNumberInputStream(bis);
      DataInputStream dis = new DataInputStream(lnis);
      DataOutputStream dos = new DataOutputStream(bos);

      try {
         String line;
         while (true) {
            line = dis.readLine();
              // 파일의 끝을 만났을 때
            if (line==null)
               break;

            String msg = lnis.getLineNumber()
                  + " : " + line.toUpperCase() + "\n";
            dos.writeBytes(msg);
         }
         dos.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

필터 스트림에 관한 활용법을 살펴보았습니다만 어떤 경우엔 기존의 필터 스트림 클래스들이 지원하지 않는 필터링이 필요할 때도 있습니다. 이런 경우 사용자가 직접 필터 스트림을 구현할 수 있습니다. 자신만의 필터 스트림을 구현하고 싶은 독자는 다음 URL을 참조하면 방법을 찾을 수 있습니다.

http://java.sun.com/docs/books/tutorial/essential/io/writingFiltered.html



[화면 2] 필터링 스트림 테스트 예제의 실행화면

스트림 클래스들은 체계적으로 접근하자

자바의 입출력 패키지(java.io)가 제공하는 여러 클래스들은 체계적으로 접근하는 것이 매우 중요합니다. 워낙 다양하고 많은 클래스들이 있어서 자칫 그 속에서 헤맬 우려가 있기 때문입니다. 필자가 이번 글에서 모든 입출력 클래스들에 대한 설명을 자세히 할 수 없는 이유도 바로 양이 많다는 이유 때문입니다. 그렇지만 필자의 생각에는 양만 많지 그게 그거라는 생각이 들기도 합니다. 입출력 클래스들의 계층구조와 분류 구조만 머리 속에 뚜렷이 담아두고 있다면 의외로 단순하기 짝이 없는 패키지가 아닌가 싶은 생각도 듭니다. 어쨌든 본문에 나와 있는 스트림 클래스들의 상속 계층도와 분류 기준 표를 머리 속에 잘 그리면서 각 스트림 클래스들을 바라봐야겠습니다.

필자가 글을 모두 마치면서 좀 아쉬웠던 점이 있다면 스트림 클래스들의 각 분류별로 그 구체적인 사용 예를 충분히 보여주지 못했다는 것입니다. 그래서 독자들 스스로 다음 사이트를 방문하여 여러 예제들을 충분히 익히도록 하세요.

입출력에 대한 기본 적인 내용을 자세한 예제와 함께 잘 설명해 놓은 튜토리얼 사이트와 좀 더 고급스러운 테크닉이 소개되어 있는 JDC의 아티클입니다.

http://java.sun.com/docs/books/tutorial/essential/io/index.html
http://developer.java.sun.com/developer/technicalArticles/Streams/ProgIOStreams/index.html
http://developer.java.sun.com/developer/technicalArticles/Streams/WritingIOSC/index.html

다음 달에는 객체지향 통신 프레임웍인 RMI(Remote Method Invocation)에 대해 자세히 알아보겠습니다.

올바른 자바프로그래밍 학습을 위한 제안

1작이 관건이다. 그리고 좋아하는 것을 하자.

필자는 주위에서 프로그래밍 공부를 하고있는 친구들에게 항상 강조합니다. 쓸만한 프로그램을 자기 손으로 처음부터 끝까지 꼭 한번은 만들어 보라고 말입니다. 그리고 가능한 빨리 시작할수록 좋다는 얘기도 꼭 덧붙이죠. 이미 그런 경험이 있는 독자들은 공감하고 있겠지만 프로그램을 처음부터 끝까지 자기 손으로 한번 만들어보고 나면 프로그래밍에 대한 시야가 완전히 바뀝니다. 마치 장님이 눈을 뜨듯이 그 동안 뭔가 막연하기만 했던 것들이 한꺼번에 손에 잡히는 느낌을 가질 수 있을 것입니다. 게다가 그로 인해 얻게되는 자신감은 두고두고 큰 힘을 발휘하게 되죠.

필자도 컴퓨터공학을 전공하고 있지만 많은 전산 분야 전공생들이 전산 분야의 특이성을 재빨리 인식하지 못하고 타성에 젖어 살아가는 것 같아 안타까울 때가 많습니다. 학교에서 가르쳐주는 것들만 잘 알고 있으면 중간 이상은 간다고 생각하는 사람들이 매우 많은 것 같습니다. 그러나 같은 노력을 들이더라도 패러다임, 즉 프로그래밍에 대한 시각이 다른 상황이라면 그 능률이라는 것은 천지차이가 될 것입니다. 어떤 사람은 엄청난 노력을 들이고도 중간을 넘지 못하고 또 어떤 사람은 애초에 중간부터 시작할 수도 있다는 것이지요. 따라서 자신의 시각을 바꾸는 것이 급선무라 하겠습니다.

앞서 말했듯이 초보 프로그래머들이 자신의 시각을 바꿀 수 있는 가장 빠른 길은 자신의 작품을 처음부터 끝까지 자기 손으로 완성해보는 것입니다. 사실 그 후에도 넘어야할 또 다른 장벽들이 많이 있습니다만 일단 시작은 자신의 첫 작품을 완성하는 것입니다. 학교에서 과제로 제출하는 레포트 프로그램은 부족합니다. 처음 프로그램 구상부터 자신의 생각으로 시작한 프로그램이어야 합니다. 레포트물의 경우 프로그램의 종류와 스펙이 주어지기 때문에 자신의 생각이나 창의성이 제약을 받게 됩니다. 힘들게 완성을 하고도 별로 애착이 안가는 이유가 그 때문이죠. 어떤 프로그래머들은 자신의 작품이 마치 자신의 자식과 같은 생각이 든다는 말을 하는 합니다. 그만큼 자신의 프로그램에 애착이 있다는 이야기지요. 자신이 아끼는 프로그램일수록 좀 더 완벽한 프로그램이 되었으면 하는 마음은 당연한 것입니다. 그래서 그런 프로그래머들은 자신의 프로그램이 좀 더 좋은 프로그램이 되기 위해서는 어떻게 해야할까 하는 고민을 수없이 하게됩니다. 이런 고민들은 아주 소중한 것들입니다. 그런 고민들 속에서 프로그래머는 아주 많은 것을 얻을 수 있기 때문이죠. 바로 장인정신이 아닐까 생각합니다.

직업을 택할 때 가장 중요한 요소는 적성과 흥미라는 생각이 이제는 일반화되어 있습니다. 거창하게 직업까지 운운하지 않더라도 어떤 일이든 자신이 흥미 있어하는 일을 해야 능률이 오르게 마련입니다. 독자 여러분들이 프로그램을 짜려할 때도 똑같은 이치가 적용됩니다. 자신의 첫 작품을 고를 때 반드시 자신이 흥미 있어하는 것을 선택하길 바랍니다. 그리고 지금 당장 시작하세요.

소켓이란?

자바에서 소켓이란 어떤 데이터 통로의 입구나 출구라고 생각하면 쉽습니다.(본기사의 [그림 1] 참조) 소켓의 가장 중요한 역할은 실제의 네트웍으로부터 사용자를 격리시키는 것입니다. 좀 더 정확히 말한다면 소켓은 TCP 기반의 네트웍에 대한 추상화된 표현이라고 할 수 있습니다. 따라서 사용자는 실제의 네트웍에 대해 어떠한 가정이나 지식이 없이도 네트웍 프로그램을 만들어 낼 수 있습니다. 예를 들어 전송 속도나 네트웍의

트래픽, 에러 등에 대해 신경 쓰지 않고도 프로그램을 개발할 수 있다는 얘깁니다.

사용자가 알아야할 것은 오로지 소켓이 정의해 놓은 몇 가지 인터페이스뿐입니다. 본 기사에서 그러한 인터페이스 중 몇몇을 발견하게 되겠지만 가장 기본이 되는 내용은 소켓이 8비트의 바이트 단위로 데이터를 전송하고 받아들인다는 것입니다.

바이트 단위라고 해서 데이터가 한 바이트 가고 조금 쉬었다가 다시 한 바이트가 가는 식으로 전송되지는 않습니다. 데이터는 바이트들이 줄을 지어 간다고 생각하는 것이 맞습니다. 그것도 한 줄로 말입니다. 그래서 이러한 바이트 열을 특별히 스트림(Stream)이라는 용어를 사용하여 표현하기도 합니다. 자바에서는 바이트 단위뿐만 아니라 이렇게 스트림 단위로 데이터를 처리할 수 있는 여러 클래스들이 존재합니다.

초보 프로그래머의 디버깅 노트

바이트와 캐릭터

본문에도 어느 정도 언급이 되기 때문에 굳이 디버깅 노트에 올려놓지 않으려 했습니다만 한반도에 사는 한국인으로서 한글로 인해 야기되는 어려움이 많은 듯 해서 이번 디버깅 토트의 도마에는 한글과 같이 2바이트 문자를 사용하려할 때 발생하는 논리적인 에러를 올려놓겠습니다.

본문에 있는 [리스트 2] [리스트 3]의 텔넷 프로그램을 실행하다보면 한글이 제대로 나오지 못한다는 사실을 발견하게 될 것입니다. 영문은 매끄럽게 잘 나오지만 한글 부분에 가서는 외계인의 언어처럼 이상한 문자들이 나오는 것을 확인할 수 있습니다. 그렇게 된 이유는 한글을 읽어올 때 DataInputStream 클래스를 사용하기 때문입니다. 본문에서 DataInputStream 클래스는 바이트 스트림 계열의 입출력 클래스라고 했습니다. 따라서 한글과 같이 2바이트를 사용하는 문자는 제대로 출력될 리가 없지요. 예를 들어 '가'라는 문자를 표현하기 위해 1000100010001000 이렇게 2바이트를 사용한다고 생각해볼까요? 2바이트씩 구분하는 유니코드의 입장에서야 '가'로 보이겠지만 바이트별로 구분하는 바이트 스트림 클래스의 입장에서 보면 10001000과 10001000 이렇게 두 개의 문자로 보이게됩니다. 10001000이 아스키코드로 괴상한 문자라면 이 2바이트 열이 괴상한 문자 두 개로 보일 수 있다는 얘기죠. 따라서 2바이트 문자를 제대로 입출력하기 위해서는 캐릭터 스트림 계열의 클래스를 사용해야 합니다.

이제 "myfile.txt"라는 파일에 한글과 같이 2바이트를 사용하는 문자열이 포함되어 있다고 가정해볼까요? 이 파일에서 한 줄의 문자열을 읽어들여 화면에 출력하려면 다음과 같이 하면 되겠습니다.

```
FileInputStream fis = new FileInputStream("myfile.txt");
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
System.out.println(br.readLine());
```

본문에서 InputStreamReader 클래스는 캐릭터 스트림 클래스이며 또 바이트열과 캐릭터열을 상호 변환하는 역할을 하는 데이터 프로세싱 스트림이라고 했습니다. 바로 FileInputStream 객체가 읽어들이는 바이트열을 캐릭터 스트림으로 변환하는 일을 맡게됩니다. 따라서 2바이트를 사용하는 한글도 잘 읽어들일 수 있지요. (이 해결책은 네트웍 프로그래밍에도 그대로 적용된다는 사실을 잊지 마세요.) 추가로 사용된

BufferedReader 클래스는 InputStreamReader 클래스에 한 줄을 통째로 읽어오는 readLine()같은 메쏘드가 없기 때문에 중첩되어 사용되었습니다. 이와 같이 다양한 용도의 스트림 클래스들을 상황에 맞게 서로 중첩하여 사용하는 연습을 많이 하다보면 자바의 입출력 프레임웍에 대해 금방 익숙해질 수 있을 것입니다.