제 9 장 네트웍 통신

자바가 제공하는 네트웍 통신 기능은 크게 소켓을 사용하는 방식과 좀더 고수준의 프로토콜인 URL 프로토콜 연결을 사용하는 방식, 두 가지를 제공한다. 이들은 모두 스트림 방식의 통신을 제공한다.

이 장에서는 다음을 배운다.

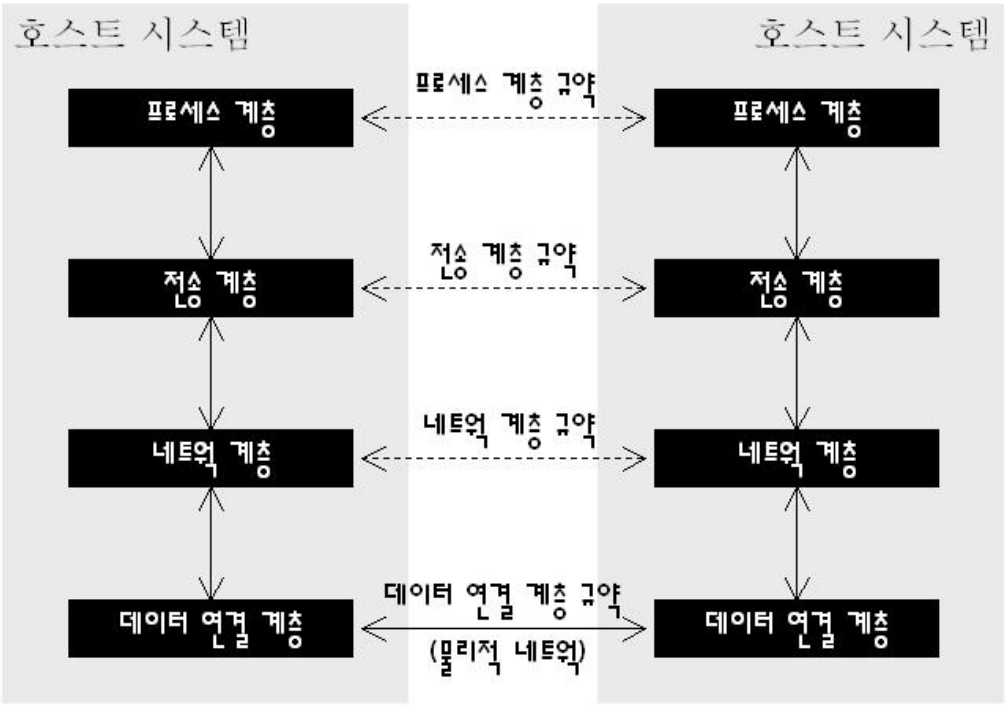
- URL 연결
- URL 프로토콜 핸들러
- 소켓 연결
- 소켓 프로그래밍 예제
- 네트웍 인터페이스 장치 정보

9.1. 네트웍

네트웍은 서로 다른 시스템을 연결하는 통신 체계를 뜻하는 말이다. 네트웍의 각 단말을 이루는 시스템을 보통 호스트라고 한다.

네트웍의 호스트들이 서로 통신하기 위해서는 미리 정의된 규약이 필요하다. 두 시스템을 연결하는 데 필요한 프로토콜을 각 역할에 따라 계층별로 분류하는데 보통 7 계층이나 4 계층으로 분류한다.

다음 그림은 7 계층 네트웍 모델을 단순화한 4 계층 네트웍 모델을 보여준다. 일반적으로 각 계층별로 별도의 규약이 정해져 있으며 상위 계층의 규약은 하위 계층 규약에 기반하여 정의된다.



[그림 1 – 4 계층 네트웍 모델]

각 계층을 간단히 살펴보면 다음과 같다.

(1) 데이터 연결 계층(data link layer) : 물리적 연결에 사용되는 계층이다. 이더넷, 토큰링, RS-232 직렬 케이블, 인공위성 연결, 무선 패킷 등등의 규약들이 이에 해당한다. 7 계층 모델에서는 이 계층을 데이터 연결 계층과 물리 계층으로 다시 분리한다.

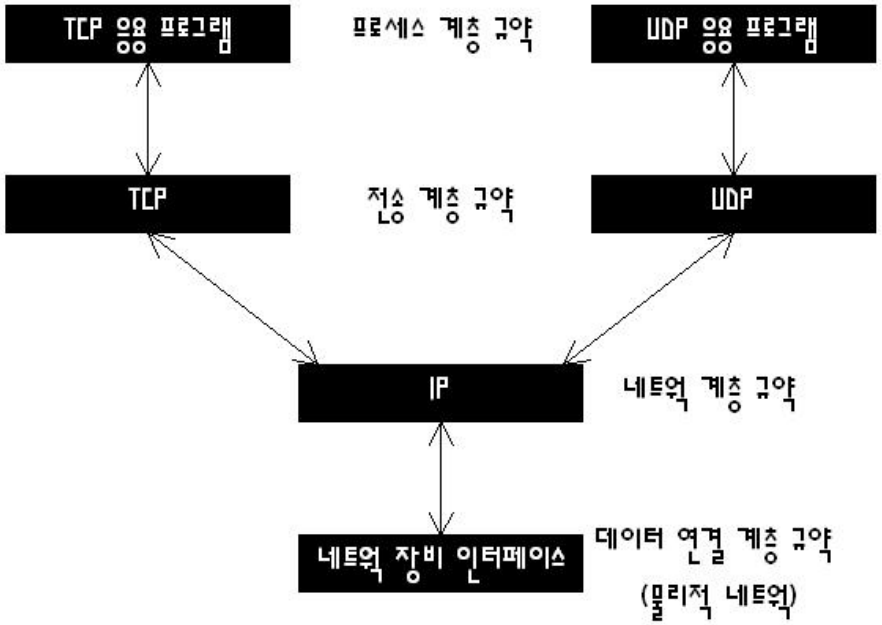
(2) 네트웍 계층 (network layer) : 원천 호스트에서 목적 호스트까지의 연결을 담당하는 계층이다. IP 규약의 경우 인터넷 주소를 사용한다.

(3) 전송 계층 (transport layer) : 원천 프로세스에서 목적 프로세스까지의 연결을 담당하는 계층이다. TCP, UDP 규약의 경우 프로세스를 찾기 위해 포트 번호를 사용한다.

(4) 프로세스 계층 (process layer) : 전송 계층을 활용하여 만드는 응용 프로그램 계층이다. http, ftp, telnet, smtp 등의 응용 프로그램 규약이 TCP/IP 규약 위에 만들어져 있다. 7 계층 모델에서는 이 계층을 세분하여 세션 계층, 표현 계층, 응용 프로그램 계층으로 나눈다.

전세계적으로 가장 많이 사용되는 네트웍 규약은 흔히 TCP/IP 라고 부르는 인터넷 규약이다.

다음 그림은 인터넷 규약을 4 계층 네트웍 모델로 표현하고 있다.



[그림 2 – TCP/IP 규약의 4 계층 네트웍 모델]

자바에서는 크게 두 가지 방식의 네트웍 연결을 제공한다. 하나는 전송 계층 규약과 프로세스 계층 규약 사이의 인터페이스인 소켓을 사용하여 TCP 응용 프로그램과 UDP 응용 프로그램을 만들 수 있게 하는 것이며, 또 다른 하나는 URL 을 사용하여 http, ftp 등의 응용 프로그램 규약에 따른 리소스를 접근할 수 있게 해준다.
네트웍 연결에 관련된 자바의 클래스들은 대부분 java.net 패키지에 정의되어 있다.

<용어 사전 시작 – TCP, UDP, IP>
TCP 는 전송 제어 규약(Transmission Control Protocol)의 약자로, 신뢰성 있는 완전 양방향의 바이트 스트림을 제공하는 연결형 규약이다. TCP 가 IP 를 사용하기 때문에 전체 인터넷 규약을 보통 TCP/IP 규약이라고 부른다.
UDP 는 사용자 데이터그램 규약(User Datagram Protocol)의 약자로 신뢰성 없는 비연결형 규약이다. UDP 는 TCP 와 달리 패킷의 전송 순서와 전송 여부 등을 보장하지 않는다.
IP 는 인터넷 규약(Internet Protocol)의 약자로 TCP, UDP 의 기반이 되는 패킷 전송 서비스를 제공하는 규약이다.

<용어 사전 끝 – TCP, UDP, IP>

9.1. URL 연결

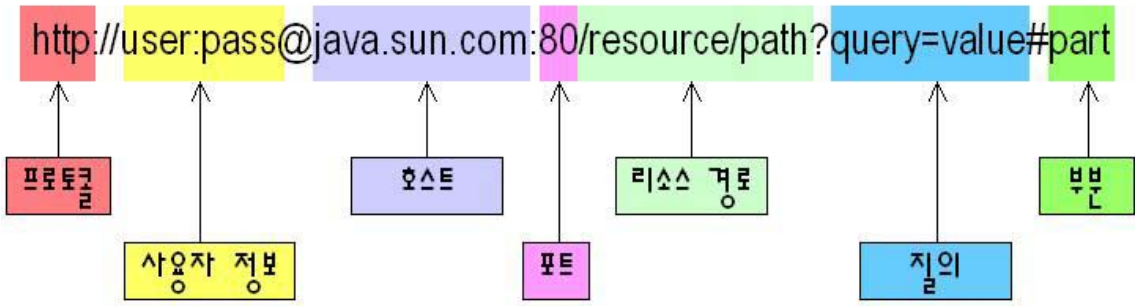
9.1.1. URL

URL 은 인터넷 상에 존재하는 리소스 위치를 나타내는 표준적인 방법이다. URL 연결은 URL 로 표현된 리소스를 연결하여 URL 에 표시된 규약에 따라 통신할 수 있는 방법을 제공한다.
각 URL 에는 URL 의 체계를 나타내는 규약이 있다. 가장 많이 사용되는 URL 규약으로는 http, ftp 등이 있으며 이들은 각각 통신 규약인 HTTP 와 FTP 를 나타낸다.

<용어 사전 시작 – URI, URL, URN>
URI (Uniform Resource Identifier) : URI 는 정형적으로 리소스를 나타내는 표준적인 방법을 뜻하며 구체적인 형태로 URL 과 URN 이 현재 정의되어 있다. 체계와 체계에 특정한 부분으로 구성된다. 즉, 체계:<체계에 특정한 내용>으로 표현한다.
URL (Uniform Resource Locator) : URL 은 인터넷 상에 존재하는 리소스의 주소를 나타내는 표준적인 방법이다.
URN (Uniform Resource Name) : URL 이 인터넷 상에 존재하는 리소스의 주소를 사용하여 표현되기 때문에 주소 변경 등의 사유가 발생하면 더 이상 리소스를 찾을 수 없는 문제점이 있다. URN 은 이런 문제를 해결하기 위한 것으로 urn:<이름 공간>:<이름 공간에서 구분되는 이름> 형식으로 정의된다. 책의 고유 번호인 ISBN 이 urn 을 사용하여 표현한다면 urn:ISBN:<ISBN 번호>와 같은 방식으로 나타낼 수 있다.

자바에서는 URI 와 URL 에 대응하는 클래스를 제공하지만, URN 에 대응하는 클래스는 제공하지 않는다. URI 클래스와 URL 클래스의 실질적인 큰 차이점은 URL 클래스는 지정된 규약에 맞는 스트림 핸들러를 처음 객체가 생성될 때 초기화한다는 것이다.
<용어 사전 끝 – URI, URL, URN>

URL 은 다음 그림과 같은 여러 부분들로 구성된다.



[그림 3 – URL 의 구성 요소]

URL 클래스는 여러 가지 생성자와 URL 구성 요소들을 알려주는 메소드들, 그리고 URL 연결을 생성하는 메소드들을 제공한다.
다음 프로그램은 명령 줄에서 입력된 URL 문자열의 각 부분을 알려준다.

<예제 1 시작 – URLInfo.java>

```
import java.net.*;

/**
 * URL 정보를 보여주는 프로그램
 */

public class URLInfo {
  public static void main(String[] args) {
    URL url = null;

    if (args.length < 1) {
```

```
      System.err.println("Usage : java URLInfo <url string>");
      System.exit(0);
    }

    try {
      url = new URL(args[0]);
    } catch (MalformedURLException e) {
      System.err.println("incorrect url - " + args[0]);
      System.exit(1);
    }

    System.out.println("Port    : " + url.getPort());
    System.out.println("Protocol  : " + url.getProtocol());
    System.out.println("Host    : " + url.getHost());
    System.out.println("File    : " + url.getFile());
    System.out.println("Ref     : " + url.getRef());
    System.out.println("Query   : " + url.getQuery());
    System.out.println("Path    : " + url.getPath());
    System.out.println("UserInfo  : " + url.getUserInfo());
    System.out.println("Authority : " + url.getAuthority());
  }
}
```

<예제 1 끝 – URLInfo.java>

URL 문자열을 인자로 받는 생성자를 사용하였고, URL 이 잘못 구성되어 있으면 MalformedURLException 예외를 던진다.
컴파일해서 실행하면 다음과 같은 결과를 볼 수 있다.

```
javac URLInfo.java <엔터>

java -cp . URLInfo
http://user:pass@java.sun.com:80/resource/path?query=value#part <엔터>
Port     : 80
Protocol  : http
Host     : java.sun.com
File     : /resource/path?query=value
```

```
Ref       : part
Query     : query=value
Path      : /resource/path
UserInfo  : user:pass
Authority : user:pass@java.sun.com:80
```

9.1.2. URL 연결의 사용

URL 연결을 표현하는 클래스는 URLConnection 이다. URL 연결을 사용하면 URL 로 표현되는 원격지 호스트의 리소스를 읽거나 쓸 수 있다.
규약과 내부 구현에 따라 입력 스트림 혹은 출력 스트림만 지원하는 연결도 있으며 둘 다 지원하기도 한다.

<요거 아세요 시작 – JDK 가 지원하는 URL 연결 규약>
선 마이크로시스템즈 사의 JDK 가 기본적으로 지원하는 URL 연결 규약으로는 http, ftp, file, mailto, jar 등이 있다.
http 와 ftp URL 연결은 입력 스트림과 출력 스트림을 모두 지원하며 HTTP 규약과 FTP 규약을 구현한다.
file URL 연결 규약은 같은 호스트 내의 파일 리소스를 경로를 사용하여 접근하는 연결 규약이다.
jar URL 연결 규약은 자바에서 클래스 파일이나 리소스 등을 배포할 때 사용하는 압축 파일 형식인 jar 파일에 포함된 리소스를 읽을 수 있는 자바 내부의 규약이다. 자세한 것은 java.net 패키지의 JarURLConnection 클래스를 참고하자.
mailto URL 연결은 SMTP 규약을 사용하여 메일을 보내는 일을 하므로 입력 스트림은 의미가 없고 출력 스트림만 지원한다.
참고로, mailto URL 연결의 경우 시스템 속성으로 세 가지를 참조한다. 먼저 user.fromaddr 는 보내는 이의 메일 주소이며 user.name 은 보내는 이의 이름, 그리고 mail.host 는 SMTP 메일 서버이다. 이 시스템 속성들이 지정되어 있지 않을 경우 메일 서버는 로컬 호스트를 사용하고 메일 주소는 현재 로그인 계정과 메일 서버를 참조하여 임시로 생성한다.
<요거 아세요 끝 – JDK 가 지원하는 URL 연결 규약>

URLConnection 객체를 생성하려면 URL 객체로부터 openConnection() 메소드를 호출하면 된다.
URLConnection 클래스는 기본적으로 월드와이드웹의 기반이 되는 HTTP 규약을 바탕으로 메소드들이 설계되어 있다. 따라서 다른 규약을 사용하는 경우 기능을 지원하지 않는 메소드들이 존재할 수 있다.
다음은 URLConnection 클래스의 주요 메소드들이다.

(1) public InputStream getInputStream();
URL 을 연결하여 입력 스트림을 되돌린다.
(2) public OutputStream getOutputStream();
URL 을 연결하여 출력 스트림을 되돌린다.
(3) public String getHeaderField(String name);
주어진 헤더 필드의 값을 되돌린다.
(4) public Map getHeaderFields();
헤더 필드를 모두 되돌린다.
(5) public Object getContent();
public Object getContent(Class[] classes);
URL 의 내용을 가장 적합한 자료형으로 되돌린다.
(6) public URL getURL();
URL 을 되돌린다.

대부분의 경우 위의 메소드들을 사용하여 입출력을 처리하는데, 연결의 속성을 지정하려면 URL 연결이 이루어지기 전에 다음 메소드들을 사용하면 된다.

(1) public boolean getUseCaches();
public void setUseCaches(boolean usecaches);
URL 연결의 캐시 사용 여부를 묻거나 지정한다.
(2) public boolean getDoInput();
public void setDoInput(boolean doinput);
URL 연결이 입력 스트림을 지원하는지 여부를 묻거나 지정한다. 기본값은 true 이다.
(3) public boolean getDoOutput();
public void setDoOutput(boolean dooutput);
URL 연결이 출력 스트림을 지원하는지 여부를 묻거나 지정한다. 기본값은 false 이다.

다음 예제는 각각 URL 에 지정된 리소스를 다운로드하는 프로그램과 해당 위치로 업로드하는 프로그램이다.
먼저 다운로드 프로그램부터 살펴보자.
URLDownloader 클래스의 download() 메소드를 보면 URL 객체로부터 openConnection() 메소드를 호출하여 URLConnection 객체를 얻은 다음에 getInputStream() 메소드를 호출하여 입력 스트림을 얻고 있다.
다른 부분은 스트림을 읽어서 파일을 저장하는 일반적인 코드들이다. URL 문자열이 슬래시('/')로 끝나는 경우에는 파일 이름이 아닌 디렉토리 이름으로 간주하여 입력 스트

림을 파일로 저장하지 않고 표준 출력으로 출력하였다. ftp 규약을 사용한 URL 일 경우 해당 디렉토리의 내용을 출력해줄 것이다.

이 프로그램은 간단한 하나의 기술을 더 사용하고 있는데 HTTP 규약의 경우 기본 인증이라는 사용자 id 와 암호를 사용한 인증 방식을 제공하고 있다. 이 기본 인증에 의해 보호되는 리소스에 접근하기 위해서는 사용자 id 와 암호를 인증받아야 하는데 그런 경우에 사용하는 것이 java.net 패키지의 Authenticator 클래스이다.

Authenticator 클래스의 setDefault() 메소드를 호출하면 그 이후로 기본 인증에 의해 보호되는 리소스에 접근하려고 하면 지정한 Authenticator 객체의 getPasswordAuthentication() 메소드가 호출된다. 이 메소드가 제대로 된 사용자 id 와 암호를 가진 PasswordAuthentication 객체를 되돌리면 해당 리소스 접근이 허가된다.

<예제 2 시작 – URLDownloader.java>

```
package yoonforh.net;

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

/**
 * URL 연결로부터 파일을 다운로드하는 프로그램
 */

public class URLDownloader {
  final static int BUFFER_SIZE = 2048;
  URL url = null;

  public static void main(String[] args) throws IOException {
    if (args.length == 0) {
      System.out.println("Usage : java URLDownloader <url>");
      System.exit(0);
    }

    new URLDownloader(args[0]).download();
  }

  public URLDownloader(String urlString) throws IOException {
    try {
```

```
      // 역 슬래시를 슬래시로 대체 후 URL 객체 생성
      url = new URL(urlString.replace('\\', '/'));

      // HTTP Basic Authentication 사용자 인증 처리
      Authenticator.setDefault(new Authenticator() {
          public PasswordAuthentication getPasswordAuthentication() {
            BufferedReader in = null;
            try {
              in = new BufferedReader(
                new InputStreamReader(System.in));
              System.out.println();
              System.out.println("-- Authentication Required --");
              System.out.print("user name : ");
              String user = in.readLine();
              System.out.print("password  : ");
              String pass = in.readLine();
              return           new           PasswordAuthentication(user,
pass.toCharArray());
            } catch (IOException e) {
              System.err.println("io error : " + e.getMessage());
            } finally {
              if (in != null) {
                try {
                  in.close();
                } catch (IOException e) { }
              }
            }
            return null;
          }
        });

    } catch (MalformedURLException e) {
      System.err.println("url is malformed : " + e.getMessage());
      throw e;
    } catch (IOException e) {
      System.err.println("io error : " + e.getMessage());
```

```
      throw e;
    }
  }

  public void download() throws IOException {
    boolean saveFile = true;
    String filename = url.getPath();

    filename = filename.substring(filename.lastIndexOf('/') + 1);
    saveFile = (filename.length() > 0);

    // 지정된 URL로부터 입력 스트림을 연다.
    BufferedInputStream in = null;
    BufferedOutputStream out = null;

    try {
      URLConnection connection = url.openConnection();
      in    =    new    BufferedInputStream(connection.getInputStream(),
BUFFER_SIZE);

      // 파일 이름이 없는 경우 즉, 디렉토리일 경우
      // 파일로 저장하지 않고 콘솔로 출력
      if (saveFile) {
        out  =  new  BufferedOutputStream(new  FileOutputStream(filename),
BUFFER_SIZE);
      } else {
        System.out.println("서버 응답 메시지 ---");
        out = new BufferedOutputStream(System.out, BUFFER_SIZE);
      }

      byte[] buffer = new byte[BUFFER_SIZE];
      int nRead = 0, nWrote = 0;
      if (saveFile) {
        System.out.println(filename + " 파일을 다운로드하고 있습니다.");
      }
```

```
      // 입력이 -1일 때까지 계속해서 읽어서 파일로 쓴다.
      while ((nRead = in.read(buffer, 0, 1024)) >= 0) {
        out.write(buffer, 0, nRead);
        nWrote += nRead;
        if (saveFile) {
          System.out.print("#"); // mark that it's under copying
        }
      }

      if (saveFile) {
        System.out.println();
        System.out.println(filename + " 파일에 " + nWrote + " 바이트를 썼습
니다.");
      }
    } finally {
      if (in != null) {
        in.close();  // 입력 스트림을 닫는다.
      }
      if (out != null) {
        out.flush();  // 출력 스트림을 플러시한다.
        out.close();  // 출력 스트림을 닫는다.
      }
    }
  }

}
```

<예제 2 끝 – URLDownloader.java>

위 프로그램을 컴파일하여 실행하여 보자.

```
javac yoonforh\net\URLDownloader.java <엔터>

java -cp . yoonforh.net.URLDownloader http://java.sun.com/index.html <
엔터>
```

index.html 파일을 다운로드하고 있습니다.

```
################################################
index.html 파일에 43498 바이트를 썼습니다.
```

다음은 이 프로그램에서 사용 가능한 몇 가지 URL 형태를 보여준다.

(1) http://some.host/protected.path/index.html
기본 인증이 적용된 사이트에 대해서 실행해보면 기본 인증이 제대로 동작하는지 여부를 확인해볼 수 있다. 기본 인증이 적용된 사이트일 경우 다음과 같이 인증을 위한 메시지가 나타나 사용자 id 와 암호를 묻는다.

```
-- Authentication Required --
user name : userid
password  : password
index.html 파일을 다운로드하고 있습니다.
#
index.html 파일에 836 바이트를 썼습니다.
```

(2) ftp://userid:password@host.name/resource/path
ftp 의 경우 사용자 id 와 암호를 지정하지 않으면 손님 권한으로 접근하게 된다.

(3) file:c:/autoexec.bat

다음 업로드 프로그램이다. URLConnection 은 기본값으로 출력 스트림을 지원하지 않는다. 따라서 setDoOutput(true);를 스트림을 열기 전에 호출해야 한다. 만약 URL 의 규약이 출력 스트림을 지원하지 않는다면 getOutputStream() 메소드를 호출할 때 예외가 발생한다. 나머지 부분은 다운로드 프로그램과 다를 것이 없다.

<예제 3 시작 – URLUploader.java>

```
package yoonforh.net;

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

/**
 * URL 연결로부터 파일을 업로드하는 프로그램
 */
```

```
public class URLUploader {
  final static int BUFFER_SIZE = 2048;
  URL url = null;

  public static void main(String[] args) throws IOException {
    if (args.length < 2) {
      System.out.println("Usage : java URLUploader <url> <filename>");
      System.exit(0);
    }

    new URLUploader(args[0]).upload(args[1]);
  }

  public URLUploader(String urlString) throws IOException {
    try {
      // 역 슬래시를 슬래시로 대체 후 URL 객체 생성
      url = new URL(urlString.replace('\\', '/'));
    } catch (MalformedURLException e) {
      System.err.println("url is malformed : " + e.getMessage());
      throw e;
    } catch (IOException e) {
      System.err.println("io error : " + e.getMessage());
      throw e;
    }
  }

  public void upload(String filename) throws IOException {
    // 지정된 URL로부터 입력 스트림을 연다.
    BufferedInputStream in = null;
    BufferedOutputStream out = null;

    try {
      URLConnection connection = url.openConnection();
      connection.setDoOutput(true);
```

```
      in = new BufferedInputStream(new FileInputStream(filename),
BUFFER_SIZE);
      out = new BufferedOutputStream(connection.getOutputStream(),
BUFFER_SIZE);

      byte[] buffer = new byte[BUFFER_SIZE];
      int nRead = 0, nWrote = 0;
      System.out.println(filename + " 파일을 업로드하고 있습니다.");

      // 입력이 -1일 때까지 계속해서 읽어서 파일로 쓴다.
      while ((nRead = in.read(buffer, 0, 1024)) >= 0) {
        out.write(buffer, 0, nRead);
        nWrote += nRead;
        System.out.print("#"); // mark that it's under copying
      }
      System.out.println();
      System.out.println(filename + " 파일에 " + nWrote + " 바이트를 썼습니
다.");
    } finally {
      if (in != null) {
        in.close();  // 입력 스트림을 닫는다.
      }
      if (out != null) {
        out.flush();  // 출력 스트림을 플러시한다.
        out.close();  // 출력 스트림을 닫는다.
      }
    }
  }
}
```

<예제 3 끝 – URLUploader.java>

다음은 업로드 프로그램에서 사용 가능한 몇 가지 URL 형태를 보여준다.
주의할 것은 http 규약의 경우, 출력 스트림으로 보낸다고 해서 서버쪽에 업로드되지 않는다. 좀더 복잡한 규약이 필요하다. 간단하게 업로드할 수 있는 방법으로 PUT 메소드가 있지만, 업로드 개념과는 다르므로 http 연결은 여기에서 고려하지 않는다. 여기에서 업로드라고 함은 주로 ftp 연결을 고려한 것이다.

(1) ftp://userid:password@host.name/resource/path
ftp 의 경우 사용자 id 와 암호를 지정하지 않으면 손님 권한으로 접근하게 된다.

(2) mailto:<받는 사람메일 주소>
mailto 규약은 SMTP 규약을 사용하여 메일을 보내는 일을 하므로 다음과 같은 방법으로 메일을 보낼 수 있다.

```
java -cp . -Duser.name="Yoon Kyung Koo" -Dmail.host=mail.kornet.net
yoonforh.net.URLUploader mailto:yoonforh@yahoo.com <보낼 파일 이름>
```

9.1.3. URL 스트림 핸들러

자바가 URL 연결을 처리하는 방식은 일반적으로 URL 객체의 생성자에서 규약 이름이 처음 사용될 때 해당 규약의 URL 연결을 처리할 스트림 핸들러를 찾아 적재하는 방식이다.
자바는 URL 에 사용되는 규약별로 서로 다른 URL 스트림 핸들러를 사용하는데 선 마이크로시스템즈 사의 JDK 에 포함된 기본 핸들러 클래스들은 sun.net.www.protocol.<규약>.Handler 라는 이름을 가진다. 예를 들어 http 규약을 처리하는 스트림 핸들러는 sun.net.www.protocol.http 패키지의 Handler 클래스이다. 참고로 sun.net.www.protocol 이라는 패키지 이름은 시스템 속성인 java.protocol.handler.pkgs 에서 정의한 값이며 자바 가상 머신에 따라 그 값이 다를 수 있다.
특정 규약에 해당하는 스트림 핸들러를 자바가 기본으로 제공하는 스트림 핸들러를 사용하지 않고 다른 스트림 핸들러를 사용하고 싶다거나, 새로운 규약의 스트림 핸들러를 사용하고자 할 경우에는 URL 클래스의 setURLStreamHandlerFactory() 메소드를 사용한다.

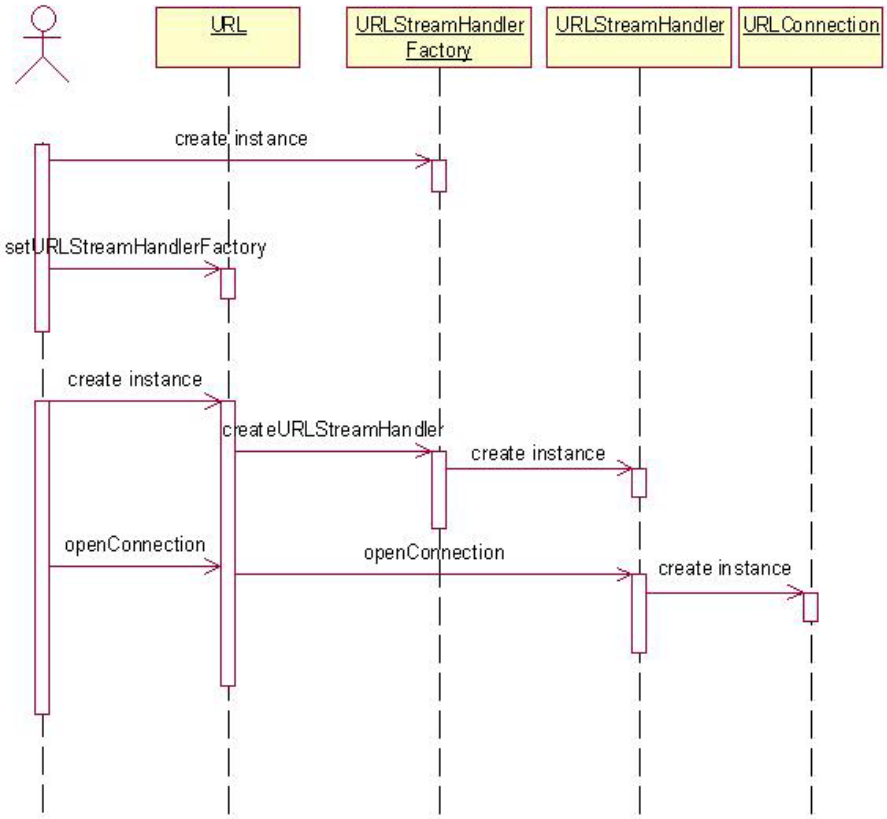
```
public static void setURLStreamHandlerFactory(URLStreamHandlerFactory
fac);
```

이 메소드는 자바 가상 머신이 실행되는 동안 한번만 실행할 수 있는데 URL 스트림 핸들러를 사용하면 사용자가 정의한 규약을 등록할 수 있기 때문에 편리한 URL 연결 방식의 프로그래밍을 할 수 있다.
URLStreamHandlerFactory 인터페이스는 이름 그대로 URL 스트림 핸들러를 만드는 클래스가 구현해야 하는 인터페이스이며 다음과 같이 URL 스트림 핸들러 객체를 만드는 메소드 하나를 선언하고 있다.

```
package java.net;
public interface URLStreamHandlerFactory {
```

```
    URLStreamHandler createURLStreamHandler(String protocol);
}
```

다음 그림은 사용자가 등록한 URL 스트림 핸들러를 사용하는 것을 보여준다.
먼저 URLStreamHandlerFactory 객체를 만들어 URL.setURLStreamHandlerFactory() 메소드를 호출하면 자바 가상 머신에 스트림 핸들러 팩토리 객체가 등록이 된다.
이후, 해당 팩토리의 규약을 사용하는 URL 이 생성될 때 핸들러 팩토리가 createURLStreamHandler() 메소드를 사용하여 스트림 핸들러를 생성한다.



[그림 4 – URL 스트림 핸들러의 사용]

자바 가상 머신에 URL 스트림 핸들러 팩토리를 등록하지 않고 URL 스트림 핸들러를 사용하는 방법은 매번 URL 객체를 생성할 때 생성자에서 인자로 URL 스트림 핸들러로 지정하는 방법이 있다. 이 방법은 특정 URL 객체에 대해서만 다른 스트림 핸들러를 사용할 수 있게 해준다.

```
public URL(String protocol, String host, nt port,
```

```
        String file, URLStreamHandler handler)
    throws MalformedURLException;
```

<따라하기 시작 – 사용자 URL 규약 정의>

규약을 간단하게 하나 만들어보도록 하자. 여기에서는 resource 라는 규약을 만들기로 한다. resource 규약은 스트림 핸들러 팩토리에서 지정한 경로를 루트 경로로 하고 URL 의 경로 부분을 상대 경로로 하여 지정된 경로에 위치한 파일을 읽는 규약이다. 변형된 형태의 file 규약이라고 생각할 수 있다.

이 프로그램은 yoonforh.net.handler 패키지에 위치한 다음 네 개의 자바 클래스로 구성된다.

(1) ResourceURLStreamHandlerFactory : resource 스트림 핸들러 팩토리

(2) ResourceURLStreamHandler : resource 스트림 핸들러

(3) ResourceURLConnection : resource URL 연결 클래스

(4) ResourceURLTest : 간단한 테스트 프로그램

1. 먼저 스트림 핸들러 팩토리 클래스를 구현해 보자.

스트림 핸들러 팩토리 클래스는 스트림 핸들러를 생성하는 메소드만 구현하면 된다. 생성자에서는 루트 경로를 받아 저장한다.

여기에서 유념할 것은 스트림 핸들러 팩토리는 자바 가상 머신에서 한번만 지정할 수 있기 때문에 여러 개의 스트림 핸들러를 등록하고자 하더라도 하나의 스트림 핸들러 팩토리에서 처리해야 한다는 것이다.

실제 구현은 간단하다. createURLStreamHandler() 메소드에서 규약을 비교하여 지원하는 규약이면 해당하는 스트림 핸들러 객체를 생성하고 아니면 null 을 리턴하여 기본 스트림 핸들러가 사용되도록 하면 된다.

<예제 4 시작 – ResourceURLStreamHandlerFactory.java>

```
package yoonforh.net.handler;

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

/**
 * resource 규약 스트림 핸들러 팩토리
 */

public class ResourceURLStreamHandlerFactory implements
```

```
URLStreamHandlerFactory {
  String rootPath = null;

  /**
   * 기본값으로 "/"를 루트 경로로 사용
   */
  public ResourceURLStreamHandlerFactory() {
    this("/");
  }

  /**
   * @param path 루트 경로
   */
  public ResourceURLStreamHandlerFactory(String path) {
    rootPath = path;
  }

  /**
   * 규약이 resource일 경우, 스트림 핸들러를 생성
   */
  public URLStreamHandler createURLStreamHandler(String protocol) {
    if (protocol.equals("resource")) {
      return new ResourceURLStreamHandler(rootPath);
    } else {
      /*
       * null을 리턴하는 규약에 대해서는 기본 스트림 핸들러가 사용된다.
       */
      return null;
    }
  }
}
```

<예제 4 끝 – ResourceURLStreamHandlerFactory.java>

2. 다음은 스트림 핸들러를 만든다.

스트림 핸들러 클래스는 URLStreamHandler 클래스를 상속하여 openConnection() 메소드를 구현해주면 된다.

여기에서는 openConnection() 메소드에서 URL 연결 객체를 생성하는 일만 한다.

<예제 5 시작 – ResourceURLStreamHandler.java>

```
package yoonforh.net.handler;

import java.net.*;
import java.io.IOException;

/**
 * resource URL 스트림 핸들러
 */

public class ResourceURLStreamHandler extends URLStreamHandler {
  String rootPath = null;

  /**
   * 생성자로 루트 경로를 받는다.
   */
  ResourceURLStreamHandler(String rootPath) {
    this.rootPath = rootPath;
  }

  /**
   * 지정된 URL의 리소스에 대한 연결을 설정한다.
   */
  public URLConnection openConnection(URL url) throws IOException {
    return new ResourceURLConnection(rootPath, url);
  }

}
```

<예제 5 끝 – ResourceURLStreamHandler.java>

3. 마지막으로 URL 연결 클래스를 구현하면 스트림 핸들러 설계가 끝난다. URL 연결 클래스는 URLConnection 클래스를 상속받아 connect() 메소드를 구현하고 입력을 지원할 경우 getInputStream() 메소드를 오버라이드, 출력을 지원할 경우

getOutputStream() 메소드를 오버라이드하면 된다. URLConnection 클래스의 getInputStream(), getOutputStream() 메소드는 기본적으로 해당 연산을 지원하지 않는다는 예외를 던지도록 구현되어 있다.

여기에서는 입력만 지원할 것이므로 getInputStream()만 오버라이드하였다. connect() 메소드의 경우에는 파일의 경우 별도 연결이 필요 없으므로 아무런 일도 하지 않는다.

<예제 6 시작 – ResourceURLConnection.java>

```
package yoonforh.net.handler;

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

/**
 * resource URL 연결 클래스
 */

public class ResourceURLConnection extends URLConnection {
  File file; // 리소스 파일

  ResourceURLConnection(String rootPath, URL url) {
    super(url);

    // path 부분을 제외한 나머지 URL 부분들은 무시한다.
    file = new File(rootPath, url.getPath());
  }

  /**
   * 별도의 connect 동작이 필요하지 않다.
   */
  public void connect() {
  }

  /**
   * 입력 스트림을 연다.
   */
  public InputStream getInputStream() throws IOException {
```

```
    return new FileInputStream(file);
  }

}
```

<예제 6 끝 – ResourceURLConnection.java>

4. 이제 스트림 핸들러 설계가 완성되었으므로 간단한 테스트 프로그램을 작성해 보자. 테스트 프로그램에서는 스트림 핸들러 팩토리를 등록하고 앞에서 만든 URLDownloader 프로그램을 사용하여 파일을 다운로드한다.

<예제 7 시작 – ResourceURLTest.java>

```
package yoonforh.net.handler;

import java.net.*;
import java.io.*;
import yoonforh.net.URLDownloader;

/**
 * resource URL 테스트 프로그램
 */

public class ResourceURLTest {
  public static void main(String[] args) {
    if (args.length < 1) {
      System.err.println("Usage : java ResourceURLTest <url>");
      System.exit(0);
    }

    // c:\를 루트 경로로 하는 resource 스트림 핸들러 생성
    ResourceURLStreamHandlerFactory factory = new
ResourceURLStreamHandlerFactory("c:\\");
    URL.setURLStreamHandlerFactory(factory);

    try {
      new URLDownloader(args[0]).download();
    } catch (IOException e) {
```

```
      System.err.println("입출력 에러 : " + e.getMessage());
    }
  }
}
```

<예제 7 끝 – ResourceURLTest.java>

5. 컴파일해서 resource URL 규약이 잘 동작하는지 확인해보자.

```
javac yoonforh\net\handler\*.java <엔터>
```

실행하기에 앞서 앞에서 만든 URLDownloader 프로그램의 클래스 파일들을 yoonforh\net 디렉토리에 위치하도록 복사하고 다음과 같이 resource URL 을 사용하여 실행해보자.

```
java -cp . yoonforh.net.handler.ResourceURLTest
resource:/windows/notepad.exe <엔터>
notepad.exe 파일을 다운로드하고 있습니다.
######################################################################
notepad.exe 파일에 66048 바이트를 썼습니다.
```

<따라하기 끝 – 사용자 URL 규약 정의>

9.2. 소켓 연결

소켓은 응용 프로그램이 TCP/IP 규약을 사용할 수 있는 API 를 제공하는 라이브러리이다. 버클리 대학에서 만들었기 때문에 흔히 버클리 소켓이라고 부른다.
소켓이란 이름은 전구가 전기 플러그를 꽂는 기구를 의미하는데 네트웍 연결에서 소켓은 네트웍 상에 위치한 두 개의 프로세스가 연결되기 위한 한쪽 끝을 뜻한다. 소켓을 사용한 네트웍 연결에 필요한 다섯 가지 요소는 다음과 같다.

(1) 통신 규약 : 연결될 두 프로그램이 서로 통신할 때 사용할 전송 계층 규약.
통신 프로토콜에는 먼저 연결을 구성한 다음 데이터를 주고받는 연결형 프로토콜(tcp 프로토콜)과 연결을 구성하지 않고 데이터를 주고받는 비연결형 프로토콜(udp 프로토콜)이 있다. 자바에서는 java.net 패키지의 Socket, ServerSocket 클래스를 사용하여 연결형 통신을 지원하고, DatagramSocket 과 MulticastSocket 클래스를 사용하여 비연결형 통신을 지원한다.
(2) 지역 호스트 주소
(3) 원격지 호스트 주소

호스트 주소는 보통 IP 주소라고 부르며 32 비트 혹은 128 비트의 정수로 표현된다. 인터넷 규약 버전 4(IPv4)를 지원하는 기존 네트웍은 32 비트로 호스트 주소를 표현하고 버전 6(IPv6)를 지원하는 새로운 네트웍은 128 비트로 호스트 주소를 표현한다.
버전 4 의 경우 정수를 문자열로 나타낼 때 마침표(.)로 구분되는 네 개의 8 비트 십진수로 흔히 나타내고, 버전 6 의 경우 콜론(:)으로 구분되는 8 개의 16 비트 16 진수로 흔히 나타낸다.
자바에서는 InetAddress 클래스가 나타내며 버전에 따라 Inet4Address 와 Inet6Address 자식 클래스가 대응된다. 정수로 나타내는 IP 주소 대신 기억하기 쉬운 호스트 이름을 사용하는데 InetAddress 클래스를 사용하여 호스트 이름과 IP 주소를 상호 변환할 수 있다.

(4) 지역 프로세스

(5) 원격지 프로세스

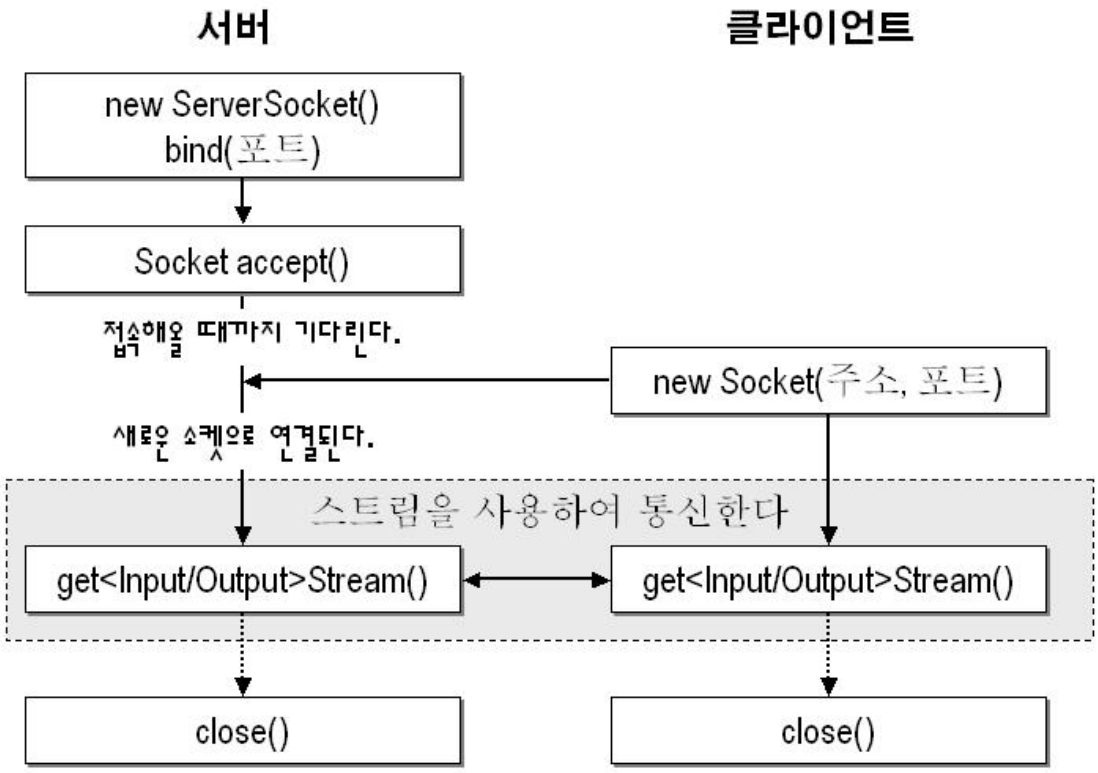
호스트 안에서 실행되는 프로세스를 지정하기 위해 보통 포트 번호를 사용한다. 보통 규약별로 서로 다른 포트를 사용하는데 http 규약은 80, ftp 규약은 21 과 같이 대부분의 알려진 규약들은 각 프로세스(서비스)가 사용할 포트 번호가 예약되어 있다. 사용자가 포트 번호를 자신의 프로그램에 부여할 때에는 보통 1024 이상의 값을 줘야 하고 해당 호스트에서 실행되는 다른 프로세스와 포트번호가 중복되지 않아야 한다.

여기에서 소켓은 한쪽 끝을 뜻하므로 통신 규약과 호스트 주소, 프로세스(포트 번호) 한 쌍을 나타낸다.

9.2.1. 소켓을 사용한 통신 방법

소켓을 사용할 경우 기반이 되는 전송 계층의 규약에 따라 프로그래밍 방식이 크게 달라진다.
udp 전송 규약을 사용하는 비연결형 통신의 경우, 데이터 전송을 신뢰할 수가 없고, 메시지의 전송 순서를 보장할 수 없다. 반면, 성능이나 네트웍 트래픽의 측면에서는 연결형 통신보다 훨씬 나은 장점이 있다.
tcp 전송 규약을 사용한 연결형 통신은 비연결형 통신의 문제점을 해결하기 위한 오버헤드가 있으나 프로그래밍 측면에서는 이러한 문제점을 고려하지 않아도 되므로 가장 많이 사용된다.

소켓을 사용할 때에는 보통 서버/클라이언트 모델이라고 하는 네트웍 통신 모델을 사용하게 되는데 서버는 네트웍 연결 요청을 기다리는 프로세스이며 클라이언트는 네트웍 연결을 요청하는 프로세스이다.
다음 그림은 각각 자바에서의 연결형과 비연결형 서버/클라이언트 통신 모델을 보여준다.



[그림 5 – 연결형 서버/클라이언트 통신 모델]

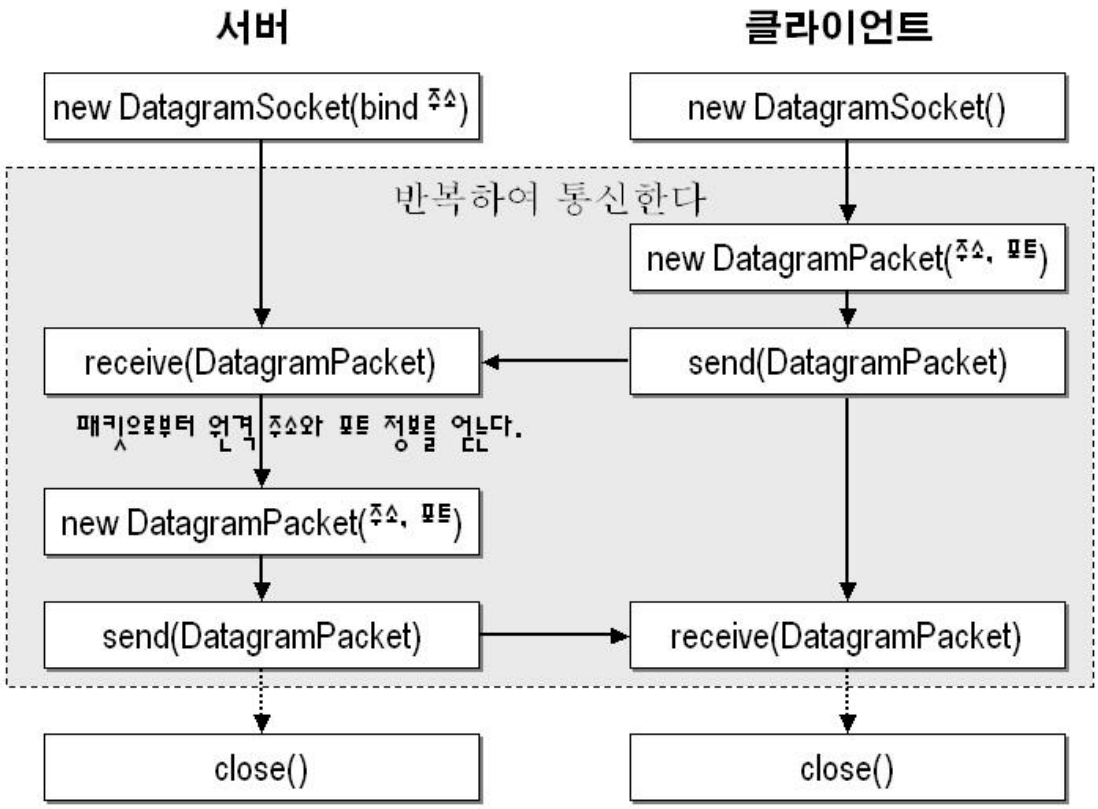
<요거 아세요 시작 – 연결형 통신에서 연결 시간 제한 두기>
연결형 서버/클라이언트 통신에서 클라이언트가 서버에 연결을 하는 방법은 생성자에서 직접 호스트와 포트를 지정하여 직접 연결하는 방법과 인자 없는 기본 생성자를 사용하여 나중에 명시적으로 연결하는 방법이 있다.
생성자에서 바로 서버 호스트로 연결을 시도하면, 연결 시간에 제한을 둘 수가 없다. 네트웍 상황에 따라 연결 시간은 상당히 길어질 수가 있으며, 연결의 결과를 알 수 있을 때까지 해당 쓰레드는 블로킹 상태에 있게 된다.
연결 시간에 제약을 두려면 클라이언트 소켓을 생성할 때 기본 생성자를 사용하고, 다음 connect() 메소드를 사용하여 연결하게 하면 된다.

```
public void connect(SocketAddress endpoint, int timeout) throws IOException;
```

이 메소드는 지정한 시간 내로 연결이 이루어지지 않으면 SocketTimeoutException 예외를 던진다.
예를 들어, 다음 코드는 연결 시간을 3 초로 제한한다.

```
Socket socket = new Socket();
try {
  socket.connect(new InetSocketAddress(host, port), 3000);
} catch (SocketTimeoutException e) {
  // 타임아웃 처리
} catch (IOException e) {
  // 기타 에러 처리
}
```

<요거 아세요 끝 – 연결형 통신에서 연결 시간 제한 두기>



[그림 6 – 비연결형 서버/클라이언트 통신 모델]

그림에서 볼 수 있듯이 연결형 통신 프로그램에서는 Socket 과 ServerSocket 클래스를 사용하며 스트림 방식으로 통신을 하며 비연결형 통신 프로그램 에서는 DatagramSocket 클래스를 사용하며 DatagramPacket 을 주고 받는 방식으로 통신을 한다.

9.2.2. 소켓 통신 예제

연결형 서버/클라이언트 프로그램을 작성해 보자. 여러 명의 클라이언트를 지원하는 간단한 채팅 서버/클라이언트 프로그램을 작성할 것이다.
서버가 여러 클라이언트와의 네트웍 연결을 동시에 처리하는 것을 동시성 서버(concurrent server)라고 부르는데 자바의 경우 여러 개의 쓰레드를 만들어서 클라이언트별로 처리하는 방법을 많이 사용한다.

<요거 아세요 시작 – 네트웍 연결 시 소켓의 값>
동일한 네트웍 연결이 동시에 생성될 수는 없기 때문에 네트웍 연결의 다섯 가지 요소 중 구분되는 값이 있어야만 새로운 네트웍 연결이 가능하다.
ServerSocket 이 accept()를 하면 새로운 Socket 이 생기는데 이 소켓은 사용하는 규약과 포트 번호는 ServerSocket 과 동일하고 호스트는 ServerSocket 이 실행된 호스트가 된다. ServerSocket 자체의 소켓 값은 임의의 클라이언트의 접속을 받기 위해 호스트가 와일드카드 값(C 소켓 API 에서는 INADDR_ANY 로 표현한다)이다.
ServerSocket 은 직접 연결을 이루지 않고 새로운 소켓을 생성하여 클라이언트와 네트웍 연결을 만들어주는데 새로 생성되는 서버쪽 소켓은 항상 호스트와 포트가 동일하므로 클라이언트쪽 소켓이 호스트나 포트 중 하나가 서로 달라야 구분되는 네트웍 연결이 된다.
일반적으로 클라이언트쪽 소켓은 포트를 임의로 할당받으므로 중복되지 않는다.
<요거 아세요 끝 – 네트웍 연결 시 소켓의 값>

<따라하기 시작 – 채팅 클라이언트 작성>
채팅 클라이언트는 간단한 그래픽 사용자 인터페이스를 지원한다.
대화 내용을 보여주는 텍스트에리어와 입력을 받는 텍스트필드 두 가지 컴포넌트로 구성되며 프로그램이 시작될 때 서버로 소켓 연결해서 서버가 보내는 메시지를 처리하는 전담 쓰레드가 실행된다.
먼저 공통 상수를 정의한 인터페이스를 보자. 서버 포트 번호를 7777 로 정하였고, 이 채팅 프로그램이 지원하는 두 가지 명령어 문자열을 정의하고 있다. 채팅 서버와 채팅 클라이언트는 모두 이 인터페이스를 구현하여 선언된 상수를 사용할 것이다.

<예제 8 시작 – ChatConstants.java>

```
package yoonforh.chat;

/**
 * 채팅 프로그램에서 사용하는 몇 가지 상수 선언
 */
```

```
public interface ChatConstants {
  int CHAT_PORT = 7777;
  String QUIT_COMMAND = "\\Q";
  String USERS_COMMAND = "\\USERS";
}
```

<예제 8 끝 – ChatConstants.java>

1. 먼저 채팅 클라이언트의 사용자 인터페이스 골격부터 작성하자. 텍스트에리어와 텍스트필드를 패널에 배치하고 프레임 윈도우에 패널을 추가한다.
배치하는 코드는 initUI() 메소드에 있다. 생성자는 컴포넌트를 배치한 후 서버에 연결을 시도하는 start() 메소드를 호출한다.

```
package yoonforh.chat;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

/**
 * 간단한 채팅 클라이언트
 */

public class ChatClient extends Panel implements ActionListener,
ChatConstants {
  final static int port = CHAT_PORT;

  String host; // 서버 호스트 이름
  TextField inputField; // 입력 필드
  TextArea outputArea;  // 대화 내용 창
  Socket socket; // 클라이언트 소켓
  BufferedReader in; // 소켓 입력 스트림
  PrintWriter out; // 소켓 출력 스트림
  Thread readerThread; // 서버 메시지 처리 쓰레드

  public ChatClient(String host) {
```

```
  this.host = host;

  // UI 배치
  initUI();

  // 서버에 연결을 시도
  start();
}

public static void main(String[] args) {
  if (args.length < 1) {
    System.err.println("Usage : java ChatClient <host name>");
    System.exit(0);
  }

  Frame f = new Frame("Chat Client");
  ChatClient panel = new ChatClient(args[0]);
  f.add(panel);
  f.setBounds(10, 10, 500, 300);
  f.addWindowListener(new WindowAdapter() {
     public void windowClosing(WindowEvent e) {
       System.exit(0);
     }
    });
  f.setVisible(true);
}

/**
 * UI 컴포넌트 배치
 */
void initUI() {
  // 입력 텍스트필드
  inputField = new TextField();
  // 대화 창
  outputArea = new TextArea();
  outputArea.setEditable(false); // 편집할 수 없게
```

```
    outputArea.setFocusable(false); // 포커스를 받지 않게

    setLayout(new BorderLayout());

    add(outputArea, BorderLayout.CENTER);
    add(inputField, BorderLayout.SOUTH);

    inputField.addActionListener(this);
    inputField.requestFocusInWindow(); // 입력 필드가 포커스를 받도록
  }

  /**
   * 대화 내용 창에 한 줄의 메시지 추가
   */
  public void appendLine(String message) {
    outputArea.append(message + '\n');
  }

}
```

2. 서버에 연결하고 소켓 연결로부터 입출력 스트림을 가져오는 부분을 구현한다.
readerThread 는 소켓 연결의 입력 스트림으로부터 메시지가 오기를 계속해서 기다리다가 메시지가 들어오면 텍스트에리어에 보여주는 일을 한다. 소켓 연결을 실제로 생성하는 connect() 메소드와 소켓 연결에 관련된 리소스를 해지하는 closeSocket() 메소드는 서로 socket 객체에 관하여 경쟁이 발생할 수 있으므로 메소드 차원에서 동기화하였다.
connect() 메소드에서 볼 수 있듯이 클라이언트쪽 소켓 사용 방법은 아주 간단하여 Socket 생성자에서 호스트와 포트를 지정하여 객체를 생성하면 이미 연결이 만들어지기 때문에 입출력 스트림을 소켓 객체로부터 구하여 그냥 사용하면 된다.

```
  /**
   * 서버에 연결하여 소켓 연결 생성
   */
  void start() {
    if (socket == null) {
      try {
```

```
      connect();
    } catch (IOException e) {
      appendLine("입출력 에러 : " + e.getMessage());
      closeSocket();
      return;
    }
  }

  /**
   * 서버의 데이터를 읽는 전담 쓰레드
   */
  readerThread = new Thread() {
      public void run() {
        while (socket != null
            && readerThread == Thread.currentThread()) {
          try {
            String line = in.readLine();
            if (line == null) { // null이면 스트림의 끝
              appendLine("연결이 끊어졌습니다.");
              closeSocket();
              break;
            }

            appendLine(line);
          } catch (IOException e) {
            appendLine("입출력 에러 : " + e.getMessage());
            closeSocket();
          }
        }
      }
    };

  // 쓰레드 실행
  readerThread.start();
}
```

```
  /**
   * 서버에 연결하여 입출력 스트림 생성
   */
  public synchronized void connect() throws IOException {
    socket = new Socket(host, port);

    in = new BufferedReader(
      new InputStreamReader(socket.getInputStream()));
    out = new PrintWriter(
      new OutputStreamWriter(socket.getOutputStream()),
      true /* auto flush */);
  }

  /**
   * 소켓을 닫는다.
   */
  public synchronized void closeSocket() {
    if (socket != null) {
      try { socket.close(); } catch (IOException e) {}
      socket = null;
    }
  }
```

3. 마지막으로 입력을 받는 텍스트필드의 액션 이벤트를 처리한다.
텍스트필드는 사용자가 엔터를 입력하면 액션 이벤트를 발생시킨다. 텍스트필드의 내용을 텍스트에리어에 디스플레이하고 서버로 보내는 일을 한다.
메소드 뒷 부분에 있는 \check 문자열을 검사하는 코드는 현재 소켓 상태를 검사하여 보여주는데 디버깅 목적으로 이와 비슷한 코드를 만들어 사용할 수 있을 것이다.

```
public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() == inputField) {
      String line = inputField.getText();
      inputField.setText(""); // clear text
      if (socket == null) {
        start();
      }
```

```
      // 빈 줄 처리
      if (line.length() == 0) {
        return;
      }

      appendLine(line);

      // 서버로 메시지 전송
      out.println(line);
      if (out.checkError()) {
        appendLine("서버로 메시지 전송에 실패했습니다.");
        closeSocket();
      }

      // 소켓 상태 디버깅하는 명령 처리
      if (line.startsWith("\\check")) {
        appendLine("Checking connection ---");
        if (socket == null) {
          appendLine("socket is null");
          return;
        }
        appendLine("socket connection : " + socket.isConnected());
        appendLine("socket input : " + !socket.isInputShutdown());
        appendLine("socket output : " + !socket.isOutputShutdown());
      }
    }
  }
```

<따라하기 끝 – 채팅 클라이언트 작성>

<따라하기 시작 – 채팅 서버 작성 및 실행>

채팅 서버의 경우, accept()에서 만들어진 각 클라이언트별 연결 소켓을 처리하는 쓰레드를 네트웍 연결마다 하나씩 전담하게 한다.

기본 동작은 한 클라이언트로부터 메시지를 받으면 해당 클라이언트를 제외한 나머지 클라이언트에게 모두 메시지를 보내는 일이다.
이러한 일을 위해 클라이언트 목록을 유지하고 있어야 한다.

1. 먼저 서버 프로그램 골격을 작성한다.
Vector 자료형으로 선언된 clients 필드는 각 클라이언트별로 생성될 쓰레드 인스턴스를 관리하기 위한 것이다.
ServerSocket 객체로부터 accept 한 후 생성된 각 연결 소켓 객체는 쓰레드 클래스인 Client 클래스의 생성자로 넘겨져 해당 쓰레드에서 입출력을 관리할 것이다. 이 채팅 프로그램은 사용자의 이름을 입력받아서 처리하는 부분이 빠져 있기 때문에 각 사용자별로 고유한 이름을 주기 위해 클라이언트 호스트의 주소와 각 소켓 객체의 해시 코드값을 합쳐서 적당한 이름을 만들었다.

```
package yoonforh.chat;

import java.io.*;
import java.net.*;
import java.util.*;

/**
 * 간단한 채팅 서버
 */

public class ChatServer implements ChatConstants {
  final static int port = CHAT_PORT;
  ServerSocket server;
  Vector clients = new Vector();

  public ChatServer() {
    // 먼저 서버 소켓을 생성
    createServerSocket();
  }

  void createServerSocket() {
    try {
      server = new ServerSocket(port);
```

```
    } catch (IOException e) {
      System.err.println("서버 소켓을 생성할 수 없습니다. : " +
e.getMessage());
      System.exit(1);
    }
  }

  /**
   * 클라이언트로부터 연결을 받아서 세션을 생성
   */
  void start() {
    while (true) {
      try {
        // 클라이언트가 접속할 때까지 블록 상태에 있다가
        // 접속하면 생성된 새로운 소켓을 새로 만든
        // Client 쓰레드 객체에게 넘겨줌
        Socket socket = server.accept();

        // 클라이언트의 이름을 적당히 정한다.
        // 원래 규약을 사용해서 이름을 클라이언트로부터 받아야 할 것이다.
        String name = makeName(socket);
        Client client = new Client(name, socket);

        // Client 쓰레드 객체를 목록에 등록
        clients.addElement(client);
      } catch (IOException e) {
        System.err.println("입출력 에러 : " + e.getMessage());
      }
    }
  }

  public static void main(String args[]) {
    new ChatServer().start();
  }

  /**
```

```
   * 적당히 클라이언트별로 고유한 이름을 만든다.
   */
  String makeName(Socket socket) {
    return socket.getInetAddress().getCanonicalHostName()
      + "-"
      + System.identityHashCode(socket);
  }

}
```

2. 각 클라이언트와의 연결을 담당할 쓰레드 클래스인 Client 클래스를 설계한다. 여기에서는 내부 클래스로 설계한다.
이 쓰레드는 입출력 스트림을 열고, 클라이언트로부터 입력을 받을 경우 다른 모든 클라이언트들에게 메시지를 브로드캐스트한다.
또, 이 프로그램이 지원하는 단 두 가지의 명령어 '\Q'와 '\USERS'를 처리한다.
그리고, 소켓 입출력 도중 예외가 발생하면 해당 연결을 종료한다.

```
  /**
   * 각 클라이언트별로 연결된 소켓을 전담하는 쓰레드
   */
  class Client extends Thread {
    Socket socket;
    BufferedReader in;
    PrintWriter out;

    Client (String name, Socket socket) throws IOException {
      super(name); // 쓰레드의 이름으로 된다.

      this.socket = socket;
      this.in
        = new BufferedReader(
          new InputStreamReader(socket.getInputStream()));
      this.out
        = new PrintWriter(
          new OutputStreamWriter(socket.getOutputStream()),
          true /* auto flush */);
```

```
  sendMessage("서버에 연결되었습니다. 당신의 이름은 " + name + "입니다.");
  ChatServer.this.broadcast(this, name + "이 새로 참가하였습니다.");
  start(); // 클라이언트 쓰레드 시작
}

/**
 * 클라이언트와의 소켓 연결을 처리한다.
 * 메시지를 받으면 이를 다른 모든 클라이언트에게 보낸다.
 */
public void run() {
  try {
    while (true) {
      String line = in.readLine();
      if (line == null) { // null이면 스트림의 끝
        closeSocket();
        break;
      }

      // 빈 줄일 경우
      if (line.length() == 0) {
        continue;
      }

      // 예약된 명령인지 검사
      // 모든 예약 명령은 역 슬래시로 시작하기로 약속되어있다.
      if (line.charAt(0) == '\\') {
        String upper = line.toUpperCase();
        if (upper.startsWith(QUIT_COMMAND)) {
          ChatServer.this.processQuitCommand(this);
          return;
        } else if (upper.startsWith(USERS_COMMAND)) {
          ChatServer.this.processUsersCommand(this);
          continue;
        }
      }
```

```
          // 입력 내용 앞에 클라이언트 이름을 붙인다.
          line = "[" + getName() + "] " + line;
          System.out.println(line);
          ChatServer.this.broadcast(this, line);
        }
      } catch (IOException e) {
        System.err.println("[" + getName() + "] 입출력 에러 : " +
e.getMessage());
        closeSocket();
      }
    }

    /**
     * 메시지를 보낸다.
     */
    public void sendMessage(String message) {
      out.println(message);
      if (out.checkError()) { // check error
        System.err.println("[" + getName() + "] 메시지 전송 에러");
        closeSocket();
      }
    }

    /**
     * 소켓 연결을 끊는다.
     */
    public void closeSocket() {
      try { if (socket != null) socket.close(); } catch (IOException e)
{}
      socket = null;
    }
  }
```

3. 전체 클라이언트에게 메시지를 보내는 메소드인 broadcast()와 두 개의 명령을 처리하는 메소드를 구현한다.

```
/**
 * 메시지를 모든 다른 클라이언트에게 보낸다.
 */
private void broadcast(Client from, String message) {
  // 연결이 끊겨진 클라이언트들을 잠시 저장하기 위해 사용
  Vector zombies = new Vector(5);

  Enumeration enum = clients.elements();
  while (enum.hasMoreElements()) {
    Client client = (Client) enum.nextElement();
    // 클라이언트 객체가 발송자인 경우 무시
    if (client == from) {
      continue;
    }

    if (client.socket == null) {
      // 소켓 연결이 끊어진 목록 작성
      zombies.addElement(client);
      continue;
    }

    client.sendMessage(message);
  }

  // 연결이 끊어진 클라이언트들을 삭제.
  // 모든 클라이언트가 메시지를 받을 수 있도록 하기 위해 별도로 처리
  enum = zombies.elements();
  while (enum.hasMoreElements()) {
    Client client = (Client) enum.nextElement();
    clients.removeElement(client);
  }
}

/**
 * quit 명령어 처리
```

```
   */
  private void processQuitCommand(Client client) {
    client.closeSocket();
    clients.removeElement(client);
  }

  /**
   * users 명령어 처리
   */
  private void processUsersCommand(Client from) {
    StringBuffer buffer = new StringBuffer(100);
    buffer.append("Current Users ---").append('\n');

    Enumeration enum = clients.elements();
    while (enum.hasMoreElements()) {
      Client client = (Client) enum.nextElement();
      if (client != null && client.socket != null) {
        buffer.append(client.getName()).append('\n');
      }
    }

    from.sendMessage(buffer.toString());
  }
```

4. 이제 구현이 완료되었으므로 컴파일하여 실행하는 일만 남았다.

```
javac yoonforh\chat\*.java <엔터>
```

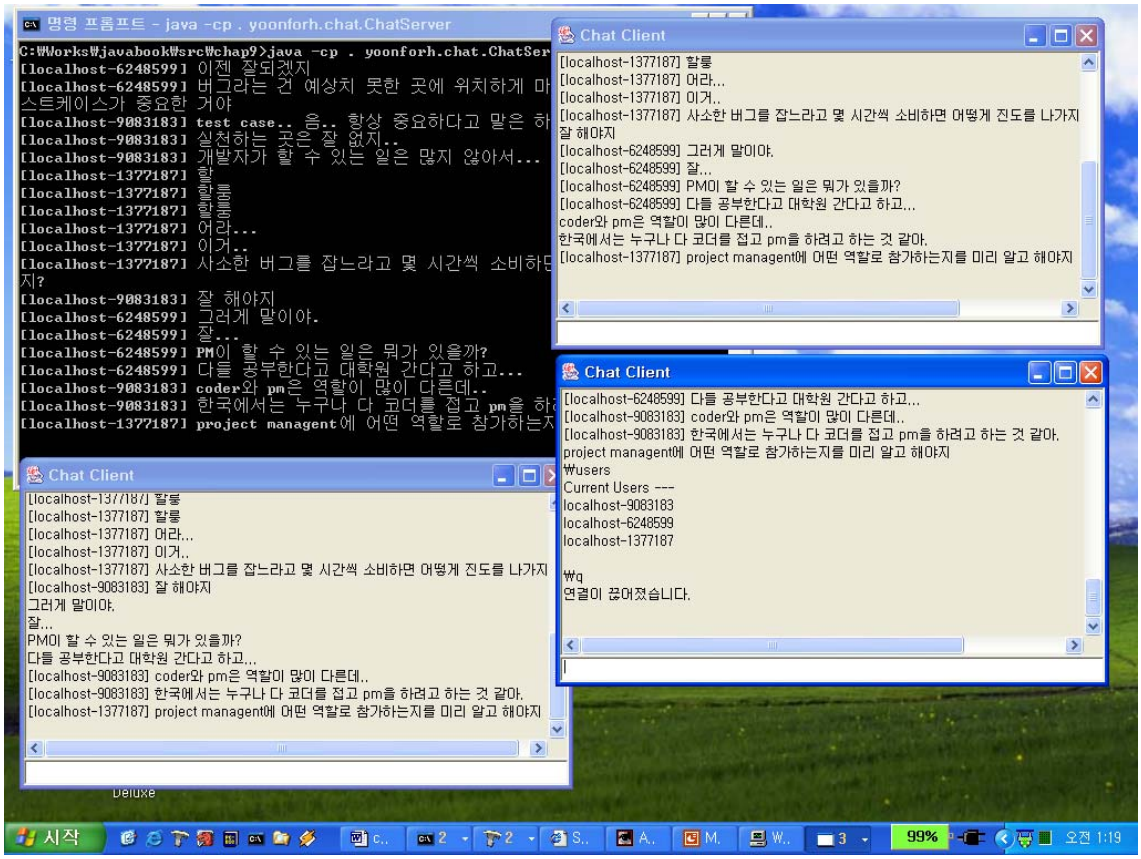
서버 프로그램은 다음과 같이 실행한다.

```
java -cp . yoonforh.chat.ChatServer <엔터>
```

클라이언트 프로그램은 명령줄 인자로 호스트 이름을 지정해줘야 한다. 테스트할 기계가 한 대뿐이라면 지역 호스트를 나타내는 localhost 를 지정해주면 된다.

```
java -cp . yoonforh.chat.ChatClient localhost <엔터>
```

다음 그림은 같은 PC 에서 세 개의 채팅 클라이언트를 띄워 실행 중인 모습니다.



[그림 7 – 채팅 서버와 채팅 클라이언트 실행 모습]

<따라하기 끝 – 채팅 서버 작성 및 실행>

<고급 팁 시작 – 동시성 서버의 쓰레드 개수 줄이기>

채팅 서버의 경우처럼 쓰레드별로 클라이언트의 소켓 연결을 전담해서 처리하면 프로그램은 아주 간단해진다. 하지만 쓰레드를 생성하는 것이 프로세스의 생성보다는 훨씬 가벼운 일이지만, 많은 클라이언트의 동시 접속을 처리하기 위해 네트웍 연결마다 쓰레드를 생성하게 되면 메모리 등 리소스 오버헤드가 커지게 된다. 이런 경우에는 쓰레드 하나에 몇 개의 네트웍 연결을 할당하는 방식으로 개선할 수 있다.

쓰레드 하나가 여러 개의 네트웍 연결을 처리하려면 이 채팅 서버처럼 readLine()과 같은 블로킹 메소드를 사용하면 메시지가 들어올 때까지 이 메소드에서 블록되어 버리기 때문에 그 동안 다른 네트웍 연결을 처리할 수가 없다. 따라서 이런 경우에는 넌블로킹 메소드인 InputStream 의 available() 메소드나 Reader 의 ready()를 사용하여 입력이 있는지 여부를 조사해보고, 입력이 있을 경우 현재 가능한 만큼만 읽어들이는 메소드인 read(byte[]) 혹은

read(char[]) 메소드를 사용해야 한다.
단일 쓰레드로 위의 채팅 서버를 고친다면 단일 쓰레드 부분은 다음과 같은 코드로 실행될 수 있다. 책의 부록에 포함된 NonBlockingChatServer.java 소스 코드를 참고하기 바란다.

```
/**
   * 각 클라이언트의 소켓 입력 상태를 검사한다.
   */
  public void run() {
    while (true) {
      Enumeration enum = clients.elements();
      while (enum.hasMoreElements()) {
        Client client = (Client) enum.nextElement();

        try {
          if (client.in.ready()) {
            client.getMessage();
          }
        } catch (IOException e) {
          System.err.println("[" + client.getName() + "] 입출력 에러 : " +
e.getMessage());
          client.closeSocket();
        }
      }

      try {
        Thread.sleep(10); // 잠깐씩 쉰다.
      } catch (InterruptedException e) { }
    }
  }
```

책의 뒷 부분에서 다룰 SocketChannel 이나 Selector 등을 사용하여 이와 같은 기법을 적용할 수 있다.
<고급 팁 끝 – 동시성 서버의 쓰레드 개수 줄이기>

9.2.3. C/C++ 프로그램과의 소켓 통신에서 주의할 점

자바를 사용하여 소켓 통신을 사용할 경우 많은 경우는 서버 혹은 클라이언트가 자바가 아

닌 C/C++ 등의 프로그램으로 작성된다.
특히 이미 서버가 존재하고 규약이 정해져 있을 경우, 자바로 클라이언트를 작성하려고 한다면 자바 프로그램 간의 통신과 다르게 입출력 스트림 사용 시에 많은 제약을 느끼게 된다. 이 중 빈번하게 발생하는 문제들을 짚어보도록 하자.

(1) 바이트 순서

가장 많이 실수를 범하는 중 하나가 바이트 순서 문제이다. C/C++ 프로그램에서도 소켓 통신을 제대로 하기 위해서는 바이트 순서를 htonl 혹은 htons 등의 매크로를 사용하여 네트웍 바이트 순서인 빅 엔디안(높은 바이트가 앞쪽에 나오는 바이트 순서)으로 숫자 값을 변환해서 내보내야 하지만, 리틀 엔디안을 사용하느 기계들 사이의 통신으로 급하게 만들어진 프로그램들은 그렇지 못한 경우가 많다.

(2) 자료형 크기의 차이

또 하나의 문제는 자바와 다른 언어들이 사용하는 자료형이 다르다는 것이다.
예를 들어, 자바는 byte 는 1 바이트, char 와 short 은 2 바이트, int 는 4 바이트, long 은 8 바이트로 수치를 나타내는 자료형의 크기가 정해져 있다. 하지만, C/C++의 경우에는 32 비트 기계와 64 비트 기계가 다르며, 또, C/C++의 char 는 자바의 byte 와 같은 자료형이며, C/C++의 int 와 long 은 32 비트 기계에서 같은 자료형을 나타내는 등의 차이가 있다. 따라서, 각 운영 체제에서 사용하는 자료형을 정확하게 이해하지 않으면 안된다.

(3) C/C++ 구조체의 바이트 정렬

또 하나의 문제는 C/C++의 struct 구조체의 바이트 정렬 문제이다.
자바에서 C/C++ 프로그램과 통신할 때에는 보통 바이트 스트림 방식으로 read(byte[]) 같은 메소드를 사용하게 될 것이다. 이때, 두 프로그램 간의 메시지 구조를 C/C++의 struct 구조체로 정의하는 경우가 대부분이다. struct 구조체는 바이트 정렬을 사용하기 때문에 자료형의 바이트 크기만으로 자바의 바이트 배열로 매핑하게 되면 전혀 엉뚱한 값들을 받게 된다.
예를 들어 32 비트 정수는 4 바이트 정렬 규칙을 따른다고 하면 다음 C 구조체의 크기는 5 바이트가 아니라 8 바이트가 된다. 두번째 필드인 i 가 4 번째 바이트에서 시작하기 때문에 첫번째 필드 c 다음에는 3 바이트가 채워지게 된다.

```
struct sample {
    char c;
    int i;
  };
```

(4) 문자열의 인코딩

C/C++의 문자열 표현은 자바에서 보면 바이트 배열이라고 볼 수 있다. 자바는 내부적으로

유니코드를 사용하고 char 자료형의 크기가 2 바이트이므로 C/C++로 문자열을 내보낼 때에는 String 클래스의 getBytes(encoding) 메소드를 사용하여 해당 문자셋 인코딩에 따른 바이트 배열로 변환한 다음에 내보내고 또, 받은 문자열을 String 객체로 변환하고자 할 때에도 먼저 바이트 배열에 데이터를 담은 다음, String 클래스의 인코딩을 지정한 생성자를 사용하여 변환하여야 한다.

(5) 버퍼링

일반적으로 C/C++에서 소켓을 사용하여 read()와 같은 블로킹 메소드를 사용하면 기반이 되는 운영 체제의 내부적인 버퍼 크기에 영향을 받게 된다. 따라서 버퍼 크기가 큰 운영 체제에서는 한 번의 read()에 원하는 양의 데이터를 읽어들이기 때문에 read() 함수의 리턴 값 즉, 실제 읽어들인 양을 제대로 확인하지 않는 코드들이 종종 있다.

자바의 경우에는 운영 체제의 버퍼와는 상관없이 구현되어 있기 때문에 네트웍을 통해서 C/C++이 보낸 데이터를 읽을 때 한 번에 읽을 수 있는 데이터 양이 불규칙적이고 일반적으로 적다. read(byte[]) 메소드는 바이트 배열이 채워지지 않더라도 읽어들일 수 있는 값이 한 바이트라도 존재하면 리턴한다.

따라서, 이러한 경우에는 반드시 read(byte[]) 메소드의 리턴값을 확인해야 하는데 다음과 같이 readn 이라는 메소드를 정의하여 사용하면 편리하다. 물론 이 메소드는 자바 프로그램 내부의 통신에서도 유용하게 사용할 수 있으며, C 프로그램에서도 이와 비슷한 기능을 하는 readn 함수가 많이 사용된다.

```
  /**
   * 지정한 바이트 크기를 읽을 때까지 기다리는 메소드
   * @return 읽어들인 바이트 크기
   */
  public static int readn(InputStream is, byte[] bytes, int offset, int
size)
        throws IOException {
    int savedOffset = offset;
    int left = size;

    while (true) {
      int nRead = is.read(bytes, offset, left);
      if (nRead < 0) { // 스트림의 끝에 도달
        break;
      }
      offset += nRead;
```

```
      left -= nRead;

      if (left <= 0) {
        break;
      }
    }

    return offset - savedOffset;
  }
```

9.3. 네트웍 인터페이스 장치 정보

TCP/IP 의 네트웍 계층에서 목적 호스트와의 전송 경로를 설정할 때, 지역 호스트에 랜(LAN) 카드와 같은 네트웍 인터페이스 장치가 여러 개 있을 경우, 목적 호스트의 주소에 따라 다른 장치를 사용하도록 설정이 가능하다. 흔히 이것을 라우팅(routing) 정보의 설정이라고 한다.

랜 카드와 같은 네트웍 인터페이스 장치가 여러 개 있을 경우, 그 정보를 알려면 java.net 패키지의 NetworkInterface 클래스를 사용할 수 있다.

```
public static Enumeration getNetworkInterfaces() throws SocketException;
```

다음 예제는 NetworkInterface 클래스를 사용하여 네트웍 인터페이스 정보를 보여준다. 윈도우 XP 와 같은 NT 계열의 ipconfig.exe 프로그램이나 유닉스 계열의 ifconfig 프로그램과 비슷한 역할을 수행한다.

<예제 9 시작 – GetIFConfig.java>

```
import java.net.*;
import java.util.Enumeration;

/**
 * 네트웍 인터페이스 정보를 사용하기
 */

public class GetIFConfig {
  public static void main(String[] args) {
    try {
```

```
      // 호스트의 네트웍 인터페이스 정보를 가져온다.
      Enumeration devices = NetworkInterface.getNetworkInterfaces();

      while (devices.hasMoreElements()) {
        // 각 네트웍 인터페이스의 정보를 표준 출력으로 내보낸다.
        NetworkInterface device = (NetworkInterface)
devices.nextElement();
        // 인터페이스의 자세한 이름
        System.out.println("device : " + device.getDisplayName());
        // 인터페이스의 간단한 이름
        System.out.println("  name : " + device.getName());

        // 각 인터페이스의 주소 정보
        Enumeration addresses = device.getInetAddresses();
        while (addresses.hasMoreElements()) {
          InetAddress address = (InetAddress) addresses.nextElement();
          System.out.println("  address : " + address.getHostAddress());
          System.out.println("   host name : " + address.getHostName());
          System.out.println("             canonical   name   :   "   +
address.getCanonicalHostName());
        }
        System.out.println();
      }
    } catch (SocketException e) {
      System.out.println("protocol error : " + e);
    }
  }
}
```

<예제 9 끝 – GetIFConfig.java>

컴파일하여 실행해보자. 실행 환경마다 다르지만, 대부분의 경우 지역 호스트로의 루프백 인터페이스와 랜 카드별로 설정된 인터페이스 정보들이 출력될 것이다.

```
javac GetIFConfig.java <엔터>

java -cp . GetIFConfig <엔터>
```

아래는 필자의 노트북에서 실행한 결과이다.

```
device : MS TCP Loopback interface
  name : lo
  address : 127.0.0.1
    host name : localhost
    canonical name : localhost

device : ACEMan-pro PPP over Ethernet Adapter
  name : eth0

device : Intel(R) PRO/100 VE Network Connection
  name : eth1
  address : 169.254.108.52
    host name : YOONFORH
    canonical name : YOONFORH
```

네트웍 인터페이스 정보를 활용하면 서버 소켓을 포트에 바인드할 때, 특정 인터페이스의 주소에 바인드하도록 할 수 있다. 주소를 지정하지 않고 포트만 지정하여 서버 소켓을 생성하면 지역 호스트의 모든 가능한 주소에 대하여 바인드를 한다. 즉, 위의 예라면 127.0.0.1 과 169.254.108.52 주소 모두에 대하여 바인드를 한다.
하지만, 서버 소켓을 생성할 때 다음 생성자를 사용하여 NetworkInteface 정보에서 가져온 InetAddress 객체를 지정하면 해당하는 주소에만 바인드를 할 것이다.

```
public ServerSocket(int port, int backlog, InetAddress bindAddr)
throws IOException;
```

맺음말

자바가 인터넷 시대의 산물이고 보면 네트웍 통신은 자바 프로그래머가 가장 쉽게 접하게 되는 분야 중의 하나일 것 같다. http 와 ftp 규약을 지원하는 URL 연결을 사용하면 간단하게 네트웍의 리소스에 접근할 수 있다.
소켓 연결과 관련한 좀더 고급 기능들은 이 책의 뒷부분에서 다시 다룰 것이다.
다음 장은 화려한 외양과 다양한 기능을 제공하는 새로운 그래픽 사용자 인터페이스인 스윙 컴포넌트를 다룬다.