자바야 놀자 - 활용편

자바야 놀자 - 활용편

발행일	2016년 4월 16일
지은이	허진경
발행처	퍼플
출판등록	제300-2012-167호 (2012년 09월 07일)
주 소	서울시 종로구 종로1가 1번지
대표전화	1544-1900
홈페이지	www.kyobobook.co.kr

목 차

이 책을 보기 전에 자바에 대한 개념을 익히셔야 합니다. 자바야 놀자(기본편)에서는 자바의 주요 개념들을 더 쉽고 정확하게 배울 수 있습니다. 자바야 놀자(기본편)은 인터넷 교보문고[1]를 통해 구매하실 수 있습니다.

이 책의 소스코드는 http://javaspecialist.co.kr/board/629에서 다운로드 받을 수 있습니다.

1) http://pod.kyobobook.co.kr/newPODBookList/newPODBookDetailView.ink?barcode=1400000280836

1. 스레드 프로그래밍

이 장에서는 멀티스레드 프로그래밍에 대해 설명하기로 하겠습니다. 스레드는 프로그램 내에서 실행되는 흐름의 단위입니다. 스레드를 이용하면 동시에 여러 개 작업을 수행시킬 수 있습니다. 이 장에서는 스레드의 기본적인 사용법 및 스레드와 관련된 여러 메서드들의 사용법에 대해서 설명합니다. 또한 멀티스레드와 멀티스레드에서 주의해야 할 사항들에 대해 설명합니다.

주요 내용입니다.

- 스레드 코드 작성 및 실행
- run() 메서드
- 스레드 상태
- getPriority()와 setPriority()
- sleep()
- join()
- yield()
- synchronized(공유데이터 접근)
- wait()와 notify()

1.1. 스레드(Thread) 실행

스레드는 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위를 의미합니다. 일반적으로 한 프로그램은 하나의 스레드를 가지고 실행됩니다. 그러나 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행해야 할 수도 있습니다. 이러한 프로그램을 멀티스레드(multi-thread) 프로그램 이라고 합니다. 멀티스레드 프로그램의 가장 쉬운 예를 떠올린다면 채팅을 들 수 있습니다. 채팅은 사용자가 입력한 채팅내용을 상대방에게 보내기 위한 스레드와 상대방으로부터 전송된 내용을 화면에 보이게 하는 스레드가 동시에 실행되어 사용자가 채팅 내용을 입력하는 도중에도 상대방으로부터 전송된 내용이 화면에 보이는 것입니다. 사실 멀티스레드 프로그램의 스레드가 동시에 실행되는 것처럼 보이지만 CPU(코어가 1개일 경우)는 여러 개의 스레드가 실행되어야 할 경우 시분할 방식에 따라 어느 한 순간에는 한 개의 스레드만 실행시킵니다. 그런데 스레드에 할당된 CPU 사용시간 간격이 매우 작기 때문에 사용자는 동시에 진행되는 것처럼 느끼는 것입니다.

스레드는 플랫폼에 따라 약간씩의 차이가 있기 때문에 프로그래머가 이를 조정해 주어야 하며, 운영체제에 따라서도 처리방식의 차이가 있습니다. 프로세스는 각각 하나의 CPU 자원을 가지지만, 스레드는 하나의 자원을 공유할 수 있습니다. 멀티프로세스와 멀티스레드는 양쪽 모두 여러 흐름이 동시에 진행된다는 공통점을 가지고 있습니다. 하지만 멀티프로세스에서 각 프로세스는 독립적으로 실행되며 각각 별개의 메모리를 차지하고 있는 것과 달리 멀티스레드는 프로세스 내의 메모리를 공유해 사용할 수 있습니다. 또한 프로세스 간의 전환 속도에 비하여 스레드 간의 전환 속도가 더 빠릅니다.

멀티스레드의 다른 장점은 CPU가 여러 개[2]일 경우에 각각의 프로세서가 스레드 하나씩을 담당하는 방법으로 속도를 높일 수 있다는 것입니다. 이러한 시스템에서는 여러 스레드가 실제 시간상으로 동시에 수행될 수 있기 때문입니다. 그러나 멀티스레드 프로그램에서는 각각의 스레드 중 어떤 것이 먼저 실행될지 그 순서를 알 수 없기 때문에 실행 결과를 예측하기 어려운 단점이 있습니다.

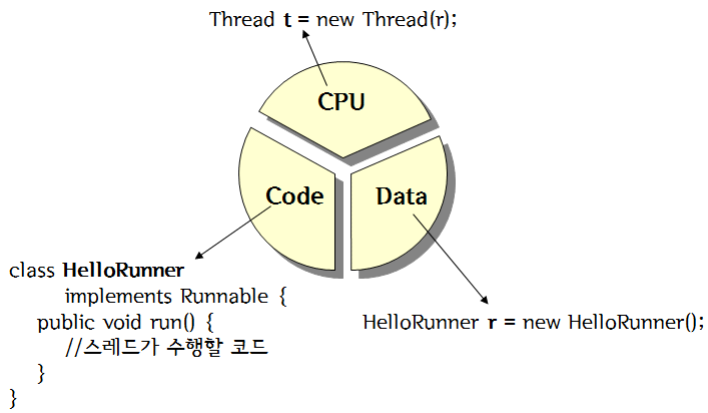
이 장에서는 용어 정의를 “스레드”는 실행환경을 의미하고, “실행환경(execution context)”은 프로그램과 데이터에 가상(Virtual) CPU를 함께 내장하고 있음을 나타내기로 하겠습니다. 그리고 "Thread"는 java.lang.Thread 클래스를 의미합니다.

1.1.1. 스레드와 run() 메서드

스레드를 실행되는 흐름의 단위라고 하였는데, 하나의 작업이 수행되기 위해서는 다음 그

2) CPU가 하나이더라도 코어(Core)가 둘 이상이라면 각각의 코어가 스레드 하나씩을 담당하여 처리합니다.

림에 보는 것처럼 3가지 요소를 필요로 합니다. 먼저 “Code”에 해당하는 부분에서 개발자는 스레드에 의해 수행될 작업 내용들을 run() 메서드에 구현해야 하는데 그림에서 Runnable 인터페이스를 구현한 HelloRunner 클래스입니다. 이러한 클래스들을 러너(runner)클래스라고 부릅니다. 그 다음에 “Data”부분이 있는데 스레드에 의해 수행될 Runner 객체에 해당하며 그림에서는 HelloRunner 클래스의 인스턴스 r에 해당합니다. 마지막으로 “Cpu”에 해당하는 부분으로 HelloRunner 클래스의 인스턴스 r을 인자로 하여 스레드 객체를 만드는 부분입니다. 이처럼 하나의 스레드를 만들려면 Runner 클래스, Runner 객체, 그리고 Thread 객체가 필요합니다.



다음 프로그램은 스레드를 이용하여 화면에 숫자를 출력하는 간단한 예제입니다.

HelloRunner.java

```
1:  public class HelloRunner implements Runnable {
2:      int i;
3:
4:      public void run() {
5:          i = 0;
6:          while (true) {
7:              System.out.println("Hello : " + i++);
8:              if ( i > 10 ) {
9:                  break;
10:             }
11:         }
12:     }//end run()
13: }//end class
```

HelloRunner 클래스를 만들기 위해서는 Runnable인터페이스를 implements합니다. HelloRunner 클래스처럼 쓰래드가 실행할 코드를 작성한 클래스를 러너(Runner) 클래스라고 부릅니다. 러너 클래스를 작성할 때 반드시 Runnable 인터페이스를 구현한 클래스가 아니어도 됩니다. Thread 클래스를 직접 상속받아 구현할 수 있습니다. 그러나 Runnable 인터페이스를 사용하여 러너 클래스를 구현하는 것을 권장합니다.

run() 메서드는 스레드가 수행할 부분입니다. run() 메서드는 Runnable 인터페이스의 추상(abstract) 메서드이기 때문에 Runnable 인터페이스를 구현(implements)하는 클래스에서는 반드시 run() 메서드를 만들어 줘야 합니다. run() 메서드는 인자가 없다는 것과 리턴타입이 void 임에 주의 하세요.

다음 코드는 스레드를 생성하고 시작시키는 예입니다.

ThreadExample.java

```
1:  public class ThreadExample {
2:     public static void main(String args[]) {
3:        System.out.println("main() 메서드 시작");
4:        HelloRunner r = new HelloRunner();
5:        Thread t = new Thread(r);
6:        t.start();
7:        System.out.println("main() 메서드 끝");
8:     }
9:  }
```

main 안에서 HelloRunner 클래스의 객체를 생성하고 Thread 클래스의 객체를 생성하고 있습니다. 이때, Thread 생성자의 인자로 HelloRunner 클래스의 객체를 사용하였습니다. 이 부분은 스레드에게 어떤 러너 클래스의 run() 메서드를 실행시킬 것인지를 알려줍니다. 생성된 스레드를 실행시키기 위해서는 반드시 start() 메서드를 호출해야 합니다. start() 메서드를 호출하면 스레드는 HelloRunner 클래스의 run() 메서드를 실행합니다.

이 예제를 실행시키면 7라인의 출력문장이 HelloRunner의 run() 메서드보다 먼저 실행될 것입니다. 그 이유는 ThreadExample 클래스를 실행시키면 main 스레드와 t 스레드가 실행되는데 t 스레드를 start() 시키면 t 스레드가 바로 실행되는 것이 아입니다. start() 한 스레드는 먼저 실행 가능한 상태(Runnable)가 된 다음 스레드 스케줄러로부터 프로세서를 할당받으면 그때 실행(Running)되기 때문입니다. t 스레드가 실행가능상 상태일 때에는 아마도 main 스레드는 이미 실행 중일 것입니다. 그래서 main스레드가 실행시키는 9라인이 먼저 실행되는 것입니다.

일반적으로 스레드를 만드는 방법은 Runnable인터페이스를 구현하여 사용하는 방법과 Thread 클래스를 상속받아 구현하는 방법이 있습니다. Thread 클래스를 상속받아 run() 메서드를 재정의 하는 방법도 가능하지만, 반드시 다른 클래스를 상속받아야 하는 상황이라면 Thread 클래스를 상속받을 수는 없을 것입니다. 앞의 예제와 같이 Runnable인터페이스를 구현하는 방법을 권장합니다. Thread 클래스는 java.lang 패키지에 있는 클래스이므로 import문이 필요 없습니다.

다음 프로그램 sleep() 메서드를 사용하여 main 스레드의 실행을 잠시 멈추게 하여 t 스레드를 먼저(?) 실행되게 하는 예입니다.

exam/java/chapter06/thread/HelloRunner2.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class HelloRunner2 implements Runnable {
4:     int i;
5:     public void run() {
6:        i = 0;
7:        while (true) {
8:           System.out.println("숫자 : " + i++);
9:           if ( i > 10 ) {
10:             break;
11:          }
12:       }
13:    }//end run()
14: }//end class
```

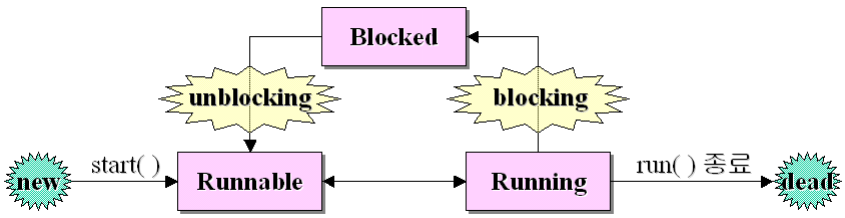
exam/java/chapter06/thread/ThreadExample2.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadExample2 {
4:     public static void main(String args[]) {
5:        System.out.println("main()의 시작");
6:        HelloRunner2 r = new HelloRunner2();
7:        Thread t = new Thread(r);
8:        t.start();
9:        try {
10:          Thread.sleep(10);
11:       } catch(InterruptedException e) {
12:          e.printStackTrace();
13:       }
14:       System.out.println("main()의 끝");
15:    }
16: }
```

이 예제가 이 전 프로그램과 다른 점은 TestThreadSleep.java 파일의 9라인에서 13라인까지 입니다. 7라인에서는 스레드를 생성합니다. 그리고 8라인에서는 start() 메서드를 사용하여 스레드를 실행시킵니다. 10라인의 Thread.sleep(10)은 현재 수행하고 있는 스레드의 수행을 잠깐 멈추게 하고 있습니다. Thread.sleep() 메서드는 밀리초(1/1000초) 단위로 스레드를 지연시킬 수 있습니다. sleep() 메서드는 InterruptedException이 발생할 가능성이 있는 메서드이므로 try~catch블록으로 예외처리를 해 주었습니다. 이때 sleep() 되는 스레드는 main 스레드일 경우도 있고 6라인에서 생성한 스레드 t 일 수도 있습니다. 이 예제에서는 확률적으로 sleep() 메서드에 의해 스레드 수행이 잠깐 멈추는 스레드는 main 스레드일 가능성이 높습니다. 그러므로 t 스레드를 실행시킨 다음 main 스레드에 의해 14라인에서 화면에 "main()의 끝"라는 문자열을 출력합니다.

1.1.2. 스레드 상태

스레드는 시작과 종료상태만 이는 것은 아닙니다. start()메서드에 의해 스레드가 실행되기 전에 스레드는 실행 가능상태가 됩니다. 다음 그림은 스레드의 기본적인 상태도를 나타낸 것입니다.



스레드 상태도를 살펴보면 먼저 하나의 스레드가 생성된 후 start() 메서드를 호출하면 곧바로 실행되지 않고 실행 가능한 상태가 되는 것을 알 수 있습니다. 스레드는 start() 메서드가 호출되면 실행가능(Runnable)한 상태로 대기하다가 스레드 스케줄러에 의해 프로세서를 할당받으면 스레드가 실행(Running)됩니다. 할당받은 시간 내에 run() 메서드 실행이 끝난다면 스레드는 종료되고, 그러지 못한다면 스레드는 다시 실행가능(runnable)한 상태로 돌아가 스레드 스케줄러에 의해 프로세서를 할당 받을 때까지 기다리게 된다. 또 스레드가 실행도중에 sleep(), join(), yield() 등과 같은 특정한 메서드의 Blocking 이벤트가 발생하면 봉쇄(Blocked)상태가 될 수도 있습니다.

앞에서 언급했지만 스레드를 생성하는 방법은 두 가지가 있는데 첫 번째 방법은 앞에서 설명한 것과 같이 Runnable 인터페이스를 implements하는 방법이었고, 이제 두 번째 방법에 대해 알아보겠습니다. 이 방법은 스레드 클래스를 직접 상속받아 구현하는 것입니다.

다음 프로그램은 앞의 예제를 스레드 클래스를 상속받아 구현한 것입니다.

exam/java/chapter06/thread/MyThread.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class MyThread extends Thread {
4:     public static void main(String args[]) throws Exception {
5:        Thread t = new MyThread();
6:        t.start();
7:        try {
8:           Thread.sleep(10);
9:        } catch(InterruptedException e) {
10:          e.printStackTrace();
11:       }
12:       System.out.println("main()의 끝");
13:    }//end main
14:    public void run() {
15:       int i = 0;
```

```
16:        while (true) {
17:           System.out.println("숫자 : " + i++);
18:           if ( i == 10 ) {
19:              break;
20:           }
21:        }
22:     }//end run
23: }//end class
```

Runnable 인터페이스를 implements하지 않고 Thread 클래스를 상속받아서 run() 메서드를 재정의해서 스레드를 만들었습니다. 스레드를 생성할 때 어떤 방식을 사용해도 되지만 일반적으로 Runnable 인터페이스를 구현하는 방법은 더 객체 지향적이고, 단일 상속의 문제를 해결할 수 있으며, 반드시 run() 메서드를 구현해야 하므로 일관성을 갖는 장점이 있습니다.

1.1.2.1. 세계 시간 출력 프로그램

다음 프로그램은 스레드를 이용하여 시계를 만드는 예제입니다. 시계는 현재 시간을 1초에 한 번씩 화면에 표시하면 됩니다.

exam/java/chapter06/thread/WorldClock.java

```
1:  package exam.java.chapter06.thread;
2:
3:  import java.util.*;
4:  import java.lang.Thread;
5:
6:  public class WorldClock implements Runnable {
7:
8:     Calendar calendar;
9:     String location;
10:
11:    public WorldClock(String city) {
12:       this.location = city;
13:    }
14:
15:    public void run() {
16:       while (true) {
17:          this.displayDate( );
18:          try {
19:             Thread.sleep(1000);
20:          } catch (InterruptedException e) {
21:          }//end try~catch
22:       }//end while
23:    }//end run()
24:
25:    public void displayDate() {
```

```
26:       String[] ids;
27:       SimpleTimeZone pdt = null;
28:
29:       if (this.location == "서울") {
30:          ids = TimeZone.getAvailableIDs(9*60*60*1000);
31:          if (ids.length == 0) System.exit(0);
32:          pdt = new SimpleTimeZone(9*60*60*1000, ids[0]);
33:       } else if (this.location == "도쿄") {
34:          ids = TimeZone.getAvailableIDs(9*60*60*1000);
35:          if (ids.length == 0) System.exit(0);
36:          pdt = new SimpleTimeZone(9*60*60*1000, ids[0]);
37:       } else if (this.location == "LA") {
38:          ids = TimeZone.getAvailableIDs(-8*60*60*1000);
39:          if (ids.length == 0) System.exit(0);
40:          pdt = new SimpleTimeZone(-8*60*60*1000, ids[0]);
41:       } else if (this.location == "뉴욕") {
42:          ids = TimeZone.getAvailableIDs(-5*60*60*1000);
43:          if (ids.length == 0) System.exit(0);
44:          pdt = new SimpleTimeZone(-5*60*60*1000, ids[0]);
45:       }
46:
47:       calendar = new GregorianCalendar(pdt);
48:       calendar.setTime(new Date());
49:       System.out.print("현재 " + location + "시각 :");
50:       System.out.print(calendar.get(Calendar.YEAR)+"년 ");
51:       System.out.print((calendar.get(Calendar.MONTH)+1)+"월 ");
52:       System.out.print(calendar.get(Calendar.DATE)+"일 :");
53:       System.out.print(calendar.get(Calendar.HOUR_OF_DAY)+"시 ");
54:       System.out.print(calendar.get(Calendar.MINUTE)+"분 ");
55:       System.out.print(calendar.get(Calendar.SECOND)+"초 ");
56:       System.out.println(" ZONE_OFFSET: " +
    (calendar.get(Calendar.ZONE_OFFSET)/(60*60*1000)));
57:    }
58: }//end class
```

exam/java/chapter06/thread/ThreadClockExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadClockExample {
4:     public static void main(String args[]) {
5:        WorldClock seoul = new WorldClock("서울");
6:        WorldClock tokyo = new WorldClock("도쿄");
7:        WorldClock la = new WorldClock("LA");
8:        WorldClock newyork = new WorldClock("뉴욕");
9:
10:       Thread seoulThread = new Thread(seoul);
11:       Thread tokyoThread = new Thread(tokyo);
12:       Thread laThread = new Thread(la);
13:       Thread newyorkThread = new Thread(newyork);
14:
```

```
15:       seoulThread.start();
16:       tokyoThread.start();
17:       laThread.start();
18:       newyorkThread.start();
19:    }
20: }
```

위의 코드는 main에서 seoulThread, tokyoThread, laThread, newyorkThread 4개의 스레드를 생성하고 각 스레드는 해당하는 지역의 시간을 출력합니다.

1.1.2.2. 폭탄 해체하기 프로그램

다음 예제는 멀티스레드를 이용하면 무엇을 할 수 있는지 보여주는 간단한 예입니다. 이 프로그램은 주어진 시간 안에 문제를 풀어야 합니다. 시간은 30초가 주어지며 30초 이내에 문제를 풀어야 합니다. 기회는 3번까지 주어지며 30초가 지나지 않더라도 3번의 기회를 모두 사용하면 실패합니다. 시간을 감소시키는 스레드와 문제를 출제하는 스레드가 동시에 실행됩니다. 코드에서 주의해서 봐야 할 부분은 스레드를 종료시키는 부분입니다. 스레드를 종료시키기 위해 플래그 값을 사용합니다.

스레드를 종료시키기 위해서는 스레드를 stop() 메서드를 사용해서 강제종료 시키지 마세요. stop() 메서드는 데드락(Dead Lock)이 발생할 가능성이 있어서 deprecated 됐습니다. 스레드를 종료시키기 위해서는 플래그 값 등을 이용해서 run() 메서드를 자연스럽게 종료시키는 것이 더 바람직합니다.

FlagData 클래스는 플래그 값을 저장하는 클래스입니다. 변수의 값을 변경시켜 run() 메서드가 자연스럽게 종료되도록 하기 위해 사용합니다.

exam/java/chapter06/thread/FlagData.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class FlagData {
4:     public static boolean isOK = false;
5:     public static boolean isFail = false;
6:  }//end FlagData class
```

CounterRunner 클래스는 시간을 감소시키면서 폭파장치 해체 여부를 확인하는 러너클래스입니다.

exam/java/chapter06/thread/CounterRunner.java

```
1:  package exam.java.chapter06.thread;
2:
```

```
3:  public class CounterRunner implements Runnable {
4:     public void run() {
5:        for(int i=30; i>0; i--) {
6:           //문제를 주어진 시간 안에 풀었는가? 확인
7:           //플래그 값으로 확인
8:           if(FlagData.isOK) {
9:              System.out.println("폭파장치가 해제되었습니다.");
10:             return;
11:          }
12:          if(FlagData.isFail) {
13:             System.out.println("폭파장치를 잘못 건드렸습니다.");
14:             break;
15:          }
16:
17:          System.out.println(i+"초 남았습니다.");
18:          //1초씩 sleep
19:          try {
20:             Thread.sleep(1000);
21:          } catch (InterruptedException e) {
22:             //nothing
23:          }
24:       }
25:       System.out.println("Boooooooooom!!!!!!!!!!!");
26:       System.exit(0);
27:    }//end run
28: }//end CounterRunner class
```

QuestionRunner 클래스는 문제를 출제하고 정답을 체크하는(폭탄을 해체하는) 러너클래스입니다.

exam/java/chapter06/thread/QuestionRunner.java

```
1:  package exam.java.chapter06.thread;
2:
3:  import javax.swing.JOptionPane;
4:
5:  public class QuestionRunner implements Runnable {
6:     String[] question = {  "3+5=",
7:                            "2349*245=",
8:                            "대한민국의 수도는?",
9:                            "200-199=",
10:                           "Runnable 인터페이스의 메서드는?",
11:                           "2의 16승은?"      };
12:    String[] answer = {    "8",
13:                           "575505",
14:                           "서울",
15:                           "1",
16:                           "run",
17:                           "65536"          };
18:    public void run() {
19:       System.out.println("주어진 시간 안에 문제를 풀어야 합니다.");
```

```
20:         System.out.println("기회는 3번까지 주어집니다.");
21:         int failCount = 3;
22:         while(true) {
23:            int index = (int)(Math.random()*6);
24:
25:            System.out.println(failCount +"번의 기회가 남았습니다.");
26:            String input = JOptionPane.showInputDialog(question[index]);
27:
28:            if(answer[index].equals(input)) {
29:               System.out.println("정답입니다.");
30:               FlagData.isOK = true;
31:               return;
32:            }else {
33:               failCount--;
34:               if(failCount <= 0) {
35:                  FlagData.isFail = true;
36:                  return;
37:               }
38:            }
39:         }//end while
40:      }//end run
41: }//end QuestionRunner class
```

다음 코드는 메인클래스입니다. 스레드를 생성하고 실행시키기 위한 코드입니다.

exam/java/chapter06/thread/ThreadBombExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadBombExample {
4:     public static void main(String[] args) {
5:        CounterRunner cr = new CounterRunner();
6:        QuestionRunner qr = new QuestionRunner();
7:
8:        Thread t1 = new Thread(cr);
9:        Thread t2 = new Thread(qr);
10:
11:       t1.start();
12:       t2.start();
13:    }
14: }
```

1.2. 스레드(Thread) 클래스의 메서드

다음 프로그램은 구구단의 출력부분을 스레드로 만들어 동시에 여러 개의 스레드가 실행된 후 출력하는 예제입니다. 다음 예를 통해 스레드 클래스가 제공하는 메서드들에 대하여 설명하겠습니다.

exam/java/chapter06/thread/GuGuRunner.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class GuGuRunner implements Runnable {
4:     private int dan;
5:     public GuGuRunner(int init_dan) {
6:        dan = init_dan;
7:     }
8:
9:     public void run() {
10:       for(int i=0; i<10; i++) {
11:          System.out.println(dan + "단: " + dan + "*" + i + "=" + dan*i);
12:       }
13:    }
14: }
```

exam/java/chapter06/thread/ThreadGuGuExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadGuGuExample {
4:     public static void main(String[] args) {
5:        Thread t2 = new Thread(new GuGuRunner(2));
6:        Thread t3 = new Thread(new GuGuRunner(3));
7:        Thread t4 = new Thread(new GuGuRunner(4));
8:        Thread t5 = new Thread(new GuGuRunner(5));
9:        Thread t6 = new Thread(new GuGuRunner(6));
10:       Thread t7 = new Thread(new GuGuRunner(7));
11:       Thread t8 = new Thread(new GuGuRunner(8));
12:       Thread t9 = new Thread(new GuGuRunner(9));
13:       t2.start();        t3.start();      t4.start();
14:       t5.start();        t6.start();      t7.start();
15:       t8.start();        t9.start();
16:    }
17: }
```

앞의 프로그램을 실행시키면 스레드 t2부터 t9까지 서로 경쟁하며 실행되는 것을 알 수 있습니다. 한순간에는 하나의 스레드만 수행되며 할당된 시간동안 스레드가 수행되다가, 다른 스레드에게 프로세서 사용권한이 넘어갑니다. 자바에서는 우선순위 값을 각각 스레드에 배정하고, 우선순위가 높은 스레드가 프로세서 사용권한을 할당받도록 하여 더 많은 프로

세스 사용 시간을 갖도록 하는 선점형(Preemptive) 방식을 사용합니다. 우선순위 값의 할당은 스레드 스케줄러(Thread Scheduler)가 담당합니다.

1.2.1. getPriority()와 setPriority()

자바의 스레드 우선순위는 1부터 10까지 가질 수 있습니다. 숫자가 클수록 우선순위는 높으며 기본 우선순위는 5입니다.

다음 프로그램은 setPriority() 메서드를 이용하여 스레드의 우선순위를 변경하고, getPriority() 메서드를 이용하여 스레드의 우선순위를 알아내는 예제입니다.

exam/java/chapter06/thread/GuGuRunner.java(이 코드는 앞에서 작성했던 코드입니다.)

```
1:  package exam.java.chapter06.thread;
2:
3:  public class GuGuRunner implements Runnable {
4:     private int dan;
5:     public GuGuRunner(int init_dan) {
6:        dan = init_dan;
7:     }
8:
9:     public void run() {
10:       for(int i=0; i<10; i++) {
11:          System.out.println(dan + "단: " + dan + "*" + i + "=" + dan*i);
12:       }
13:    }
14: }
```

exam/java/chapter06/thread/ThreadPriorityExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadPriorityExample {
4:     public static void main(String[] args) {
5:        Thread t2 = new Thread(new GuGuRunner(2));
6:        Thread t3 = new Thread(new GuGuRunner(3));
7:        Thread t4 = new Thread(new GuGuRunner(4));
8:        Thread t5 = new Thread(new GuGuRunner(5));
9:
10:       t2.setPriority(4);
11:       System.out.println( t3.getPriority() );
12:       t2.start();
13:       t3.start();
14:       t4.start();
15:       t5.start();
16:    }
17: }
```

자바에서 우선순위가 높은 스레드를 먼저 실행시키는 선점형 방식을 채택하고 있습니다. 그러므로 이 코드의 예상되는 실행 결과는 t2가 실행하는 2단이 가장 마지막에 실행되는 것입니다. 그러나 예제를 실행시키는 컴퓨터 환경에서 CPU의 코어가 두 개 이상이라면 결과에서 t2가 우선순위가 낮더라도 다른 스레드보다 먼저 실행될 수 있습니다. t2가 start된 후 t3가 runnable 상태가 되기 전에 t2가 running될 수도 있습니다.

1.2.2. sleep()

다음 프로그램은 sleep() 메서드를 이용하여 해당하는 스레드를 지정한 시간(mille second)동안 정지시키는 예제입니다. sleep() 메서드는 스레드의 실행을 지정한 시간동안 정지시킵니다. 지정한 시간이 지나면 스레드는 다시 실행상태로 되는 것이 아니고 실행 가능한 상태로 전이됩니다.

exam/java/chapter06/thread/GuGuSleepRunner.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class GuGuSleepRunner implements Runnable {
4:     private int dan;
5:     public GuGuSleepRunner(int init_dan) {
6:        dan = init_dan;
7:     }
8:     public void run() {
9:        long sleepTime = (long)(Math.random() * 500);
10:       System.out.println(dan + "단이" + sleepTime + "만큼 쉼");
11:       try{
12:          Thread.sleep(sleepTime);
13:       }catch (InterruptedException e) {
14:       }
15:       for(int i=0; i<10; i++) {
16:          System.out.println(dan + "단: " + dan + "*" + i + "=" + dan*i);
17:       }
18:    }
19: }
```

exam/java/chapter06/thread/ThreadSleepExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadSleepExample {
4:     public static void main(String[] args) {
5:        Thread t2 = new Thread(new GuGuSleepRunner(2));
6:        Thread t3 = new Thread(new GuGuSleepRunner(3));
7:        Thread t4 = new Thread(new GuGuSleepRunner(4));
8:        Thread t5 = new Thread(new GuGuSleepRunner(5));
9:        Thread t6 = new Thread(new GuGuSleepRunner(6));
10:
```

```
11:         t2.start();
12:         t3.start();
13:         t4.start();
14:         t5.start();
15:         t6.start();
16:     }
17: }
```

1.2.3. join()

다음 프로그램은 join() 메서드에 관한 예제입니다. join()메서드는 join된 스레드 이후에 실행(start)되는 스레드들은 join된 스레드가 실행을 종료한 다음 실행됩니다.

exam/java/chapter06/thread/GuGuRunner.java(이 코드는 앞에서 작성된 예제입니다.)

```
1:  package exam.java.chapter06.thread;
2:
3:  public class GuGuRunner implements Runnable {
4:      private int dan;
5:      public GuGuRunner(int init_dan) {
6:          dan = init_dan;
7:      }
8:      public void run() {
9:          for(int i=0; i<10; i++) {
10:             System.out.println(dan + "단: " + dan + "*" + i + "=" + dan*i);
11:         }
12:     }
13: }
```

exam/java/chapter06/thread/ThreadJoinExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadJoinEaxmple {
4:      public static void main(String[] args) {
5:          Thread t2 = new Thread(new GuGuRunner(2));
6:          Thread t3 = new Thread(new GuGuRunner(3));
7:          Thread t4 = new Thread(new GuGuRunner(4));
8:          Thread t5 = new Thread(new GuGuRunner(5));
9:          Thread t6 = new Thread(new GuGuRunner(6));
10:
11:         t2.setPriority(4);
12:         t3.setPriority(4);
13:         t4.setPriority(4);
14:         t5.setPriority(4);
15:         t6.setPriority(4);
16:
17:         t2.start();
18:         t3.start();
```

```
19:         t4.start();
20:         try {
21:            t4.join();       //t5와 t6은 t4의 실행이 종료돼야 실행됨
22:         }catch(InterruptedException e) {
23:         }
24:         t5.start();
25:         t6.start();
26:     }
27: }
```

1.2.4. yield()

다음 프로그램은 yield() 메서드에 관한 예제로 yield()는 동일순위의 스레드에게 프로세서의 사용을 양보하는 기능을 가지고 있습니다. 우선순위가 다른 스레드에게는 아무런 영향을 주지 않습니다.

exam/java/chapter06/thread/GuGuYieldRunner.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class GuGuYieldRunner implements Runnable {
4:     private int dan;
5:     public GuGuYieldRunner(int init_dan) {
6:        dan = init_dan;
7:     }
8:     public void run() {
9:        if(dan == 8) {
10:          System.out.println("8단이 9단에게 양보");
11:          Thread.yield();
12:       }
13:       for(int i=0; i<10; i++) {
14:          System.out.println(dan + "단: " + dan + "*" + i + "=" + dan*i);
15:       }
16:    }
17: }
```

exam/java/chapter06/thread/ThreadYieldExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class ThreadYieldExample {
4:     public static void main(String[] args) {
5:
6:        Thread t6 = new Thread(new GuGuYieldRunner(6));
7:        Thread t7 = new Thread(new GuGuYieldRunner(7));
8:        Thread t8 = new Thread(new GuGuYieldRunner(8));
9:        Thread t9 = new Thread(new GuGuYieldRunner(9));
10:
```

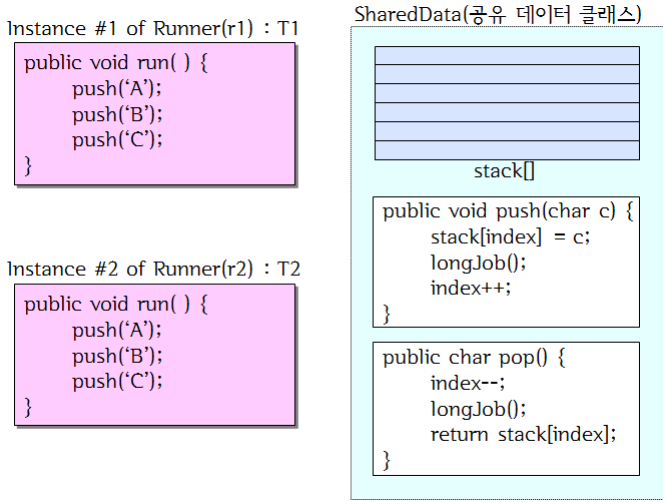
```
11:         t6.setPriority(4);
12:         t7.setPriority(4);
13:         t8.setPriority(5);
14:         t9.setPriority(5);
15:
16:         t6.start();
17:         t7.start();
18:         t8.start();
19:         t9.start();
20:     }
21: }
```

1.3. 공유데이터 접근

현 시점에서 여러 스레드 중 어떤 스레드가 수행될지 알 수 없고, 선정된 스레드가 얼마동안 수행될지 또 이 스레드가 끝난 후 다음에 어떤 스레드가 수행될지 알 수 가 없습니다. 따라서 프로그래머가 실행되는 스레드들을 관리 할 필요가 있습니다. 특히, 여러 스레드들이 공유 데이터(Shared Data)에 접근할 때는 스레드들의 관리가 더욱 필요합니다.

1.3.1. 공유데이터의 문제점

다음 예제는 반드시 이해해야하므로 충분한 시간을 가지고 공부하기 바랍니다. 먼저 이해를 돕기 위해 다음의 그림을 설명하겠습니다.



그림에서 오른쪽 공유데이터 클래스를 살펴보면 push()와 pop() 메서드가 있습니다. 이 메서드들은 스택에 데이터를 넣거(push)나 빼내(pop)는 일을 합니다.
첫 번째 스레드(T1)에서 스택에 데이터를 넣기 위해 push() 메서드를 호출합니다. 데이터를 받은 push() 메서드는 스택에 데이터를 넣습니다(①). 그 후 스택 데이터를 가리키는 포인터를 하나 증가시켜야 하는데, 그 전에 시간이 걸리는 작업(longJob())을 수행한다고 가정하면 중요한 문제가 발생할 수 있습니다. 그 이유는 첫 번째 스레드(T1) 수행 중에 스케줄러로부터 할당받은 프로세서시간을 사용하고 나면 T1 스레드는 Running상태에서 Runnable 상태로 되고, 이때 두 번째 스레드(T2)가 Running 상태가 될 수 있습니다. 시간이 오래 걸리는 일(longJob())을 수행하는 도중에(아직 포인터를 증가시키지 않았음), 우선권이 다른 스레드에게 넘어간 것입니다. 두 번째 스레드(T2)는 첫 번째 스레드에서 포인터가 증가하지 않았는데도 push() 메서드를 호출하게 되는데, 이때 포인터가 가리키는 곳에 데이터를 넣게 되므로 첫 번째 스레드 T1이 push한 데이터에 두 번째 스레드 T2가

push한 데이터가 덮어써지게 되는 것입니다.

지금까지 설명한 내용을 이해할 수 있도록 예제코드를 작성하겠습니다.
다음 코드는 공유데이터에 사용되는 클래스입니다. 푸시와 팝을 구현한 클래스입니다. 공유데이터 영역에 데이터를 푸시 할 때 어떤 데이터가 저장되는지 보여주기 위해서 push() 메서드에서 푸시 한 데이터를 반환하도록 구현했습니다. pop() 메서드의 synchronized 블록은 뒤에서 자세하게 설명됩니다.

exam/java/chapter06/thread/SharedData.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class SharedData {
4:
5:     int index=0;
6:     char[] stack = new char[6];
7:
8:     public char push(char c) {
9:        stack[index] = c;
10:       longJob();
11:       index++;
12:       return c;
13:    }
14:
15:    public char pop() {
16:       synchronized(this) {
17:          index--;
18:          longJob();
19:          return stack[index];
20:       }
21:    }
22:
23:    public void longJob() {
24:       for(int i=0; i<10; i++) {
25:          System.out.print(".");
26:       }
27:       System.out.println();
28:    }
29: }
```

아래의 코드는 스레드가 공유데이터 영역에 데이터를 푸시 합니다.

exam/java/chapter06/thread/PushRunner.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class PushRunner implements Runnable {
4:     String name;
5:     SharedData sd;
6:     PushRunner(String name, SharedData sd) {
```

```
7:         this.name = name;
8:         this.sd = sd;
9:      }
10:     public void run() {
11:        System.out.println(name + sd.push('A'));
12:        System.out.println(name + sd.push('B'));
13:        System.out.println(name + sd.push('C'));
14:     }
15: }
```

아래의 코드는 스레드가 공유데이터 영역에 저장되어 있는 데이터를 팝 합니다.

exam/java/chapter06/thread/PopRunner.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class PopRunner implements Runnable {
4:     String name;
5:     SharedData sd;
6:     PopRunner(String name, SharedData sd) {
7:        this.name = name;
8:        this.sd = sd;
9:     }
10:    public void run() {
11:       System.out.println(name + sd.pop());
12:       System.out.println(name + sd.pop());
13:       System.out.println(name + sd.pop());
14:    }
15: }
```

아래의 코드는 푸시 스레드와 팝 스레드를 각각 두 개씩 생성하여 실행시킵니다. 먼저 푸시 스레드가 데이터를 푸시한 후에 팝 스레드가 데이터를 팝 하도록 하기 위해 join() 메서드를 사용하였습니다.

exam/java/chapter06/thread/SharedDataExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class SharedDataExample {
4:     public static void main(String[] args) {
5:        SharedData sd = new SharedData();
6:
7:        PushRunner pushr1 = new PushRunner("Push-하나 : ", sd);
8:        PushRunner pushr2 = new PushRunner("Push-둘 : ", sd);
9:
10:       Thread pusht1 = new Thread(pushr1);
11:       Thread pusht2 = new Thread(pushr2);
12:
13:       pusht1.start();
14:       pusht2.start();
15:
```

```
16:         try {
17:            pusht1.join();
18:            pusht2.join();
19:         } catch (InterruptedException e) {
20:            //nothing...
21:         }
22:
23:         System.out.println("\n스택에 저장된 데이터는");
24:
25:         PopRunner popr1 = new PopRunner("Pop-하나 : ", sd);
26:         PopRunner popr2 = new PopRunner("Pop-둘 : ", sd);
27:
28:         Thread popt1 = new Thread(popr1);
29:         Thread popt2 = new Thread(popr2);
30:
31:         popt1.start();
32:         popt2.start();
33:
34:      }
35: }
```

예제를 실행시키면 스택에 데이터가 정상적으로 푸시 되지 않을 가능성이 있는 것을 알 수 있습니다. 만일 불특정 다수가 사용하는 인터넷 환경에서 멀티스레드로 인한 공유데이터의 손상은 매우 높은 확률을 갖게 될 것입니다. 지금까지의 상황에 대한 결론은 공유 데이터를 가지고 작업하는 스레드는 예측할 수 없는 순간에 제어권이 다른 스레드로 양도되어 데이터파손이나 손상을 일으킬 수 있음을 명심해야 한다는 것입니다. SharedDataExample 클래스를 실행시켰을 때의 결과는 항상 동일하지는 않습니다. pop() 메서드에 의해 데이터를 팝하는 부분에서 아무것도 출력이 되지 않을 수 있습니다.

실행 결과

```
.....................
Push-하나 : A
.....................
.....................
Push-하나 : B
Push-둘 : A
.....................
.....................
Push-하나 : C
Push-둘 : B
.....................
Push-둘 : C

스택에 저장된 데이터는
.....................
.....................
Pop-하나 : C
Pop-둘 :
.....................
Pop-하나 : B
.....................
Pop-둘 :
.....................
Pop-하나 : B
.....................
Pop-둘 : A
```

1.3.2. 공유데이터 문제점의 해결(synchronized)

이와 같은 공유데이터 문제는 매우 심각하기 때문에 해결방안을 모색해야 하는데 문제의 심각성에 비해 그렇게 어렵지만은 않습니다. 먼저 첫 번째 스레드가 push() 메서드나 pop() 메서드를 호출하여 수행되는 동안 두 번째 스레드로 제어권이 넘어가지 않아야 합니다. 또 다른 방법은 제어권이 두 번째 스레드로 넘어갔더라도, 첫 번째 스레드가 메서드를 완전히 수행하지 않은 상태라면, 제어를 첫 번째 스레드에게 다시 넘겨주면 됩니다. 자바에서는 두 번째 방법을 사용하는데 이때 사용하는 키워드가 synchronized 입니다.

다음 프로그램은 앞의 예제 프로그램에 synchronized를 추가하여 공유데이터의 문제점을 해결한 것입니다. 앞에서 작성된 SharedData 클래스 파일을 아래코드에서 진하게 된 부분만 수정한 다음 SharedDataExample 클래스로 실행시키세요.

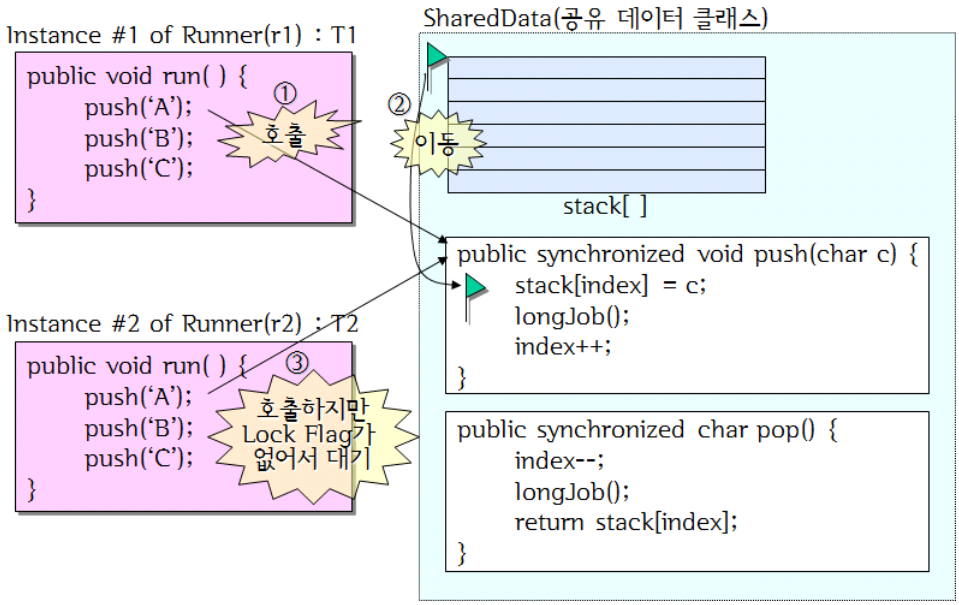
exam/java/chapter06/thread/SharedData.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class SharedData {
4:
```

```
5:      int index=0;
6:      char[] stack = new char[6];
7:
8:      public synchronized char push(char c) {
9:         stack[index] = c;
10:        longJob();
11:        index++;
12:        return c;
13:     }
14:
15:     public char pop() {
16:        synchronized(this) {
17:           index--;
18:           longJob();
19:           return stack[index];
20:        }
21:     }
22:
23:     public void longJob() {
24:        for(long i=0; i<20; i++) {
25:           System.out.print(".");
26:        }
27:        System.out.println();
28:     }
29: }
```

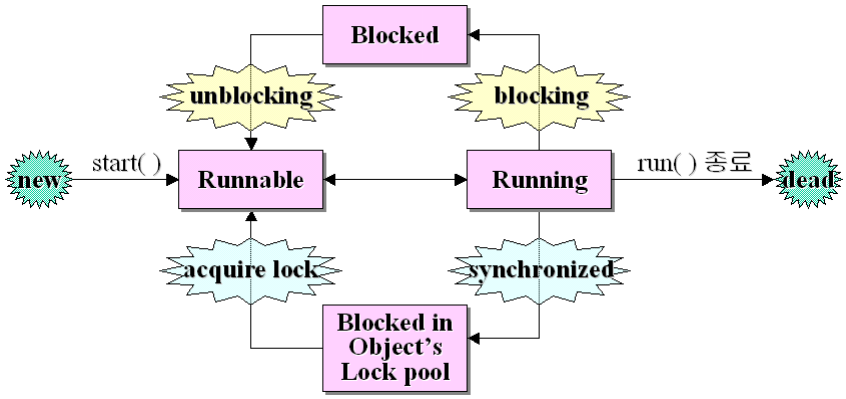
예제코드에서 push() 메서드와 pop() 메서드에서 synchronized를 사용했습니다. synchronized 키워드가 두 메서드에서 서로 다르게 사용되었는데, push()에서처럼 사용하는 것을 synchronized method라하고, pop() 메서드처럼 사용하는 것을 synchronized block이라고 합니다. lock flag를 필요로 하는 부분이 메서드 전체가 아니라면 필요한 부분만 블록으로 설정해서 사용하는 것이 더 바람직합니다. SharedDataExample 클래스를 통해서 실행시키면 실행 결과는 다르게 나타날 수 있지만 중요한 것은 스택에 데이터가 모두 푸시 되고 팝 된다는 것입니다.(pop()메서드에 의해 출력되는 데이터가 모두 존재합니다.)

synchronized를 설명하기 위해 Object Lock Flag를 먼저 설명하기로 하겠습니다. 생성된 모든 객체는 lock flag를 가지고 있는데 스레드가 synchronized 메서드나 블록을 수행하기 위해서는 반드시 lock flag를 가지고 있어야 합니다. 다음 그림을 살펴보세요.



앞의 그림에서처럼 push() 메서드를 호출하면 스레드는 공유객체의 lock flag가 있는지 확인하고, 있으면 lock flag를 가져간 후에 실행됩니다. 이렇게 lock flag를 소유한 스레드에 의해 push() 메서드가 수행되다가 다른 스레드에게 제어권이 넘어가면 첫 번째 스레드는 대기(block) 상태가 되고, 두 번째 스레드가 활동하기 시작할 것입니다. 그러나 두 번째 스레드에서도 push() 메서드를 호출하면 synchronized 키워드로 인해 공유객체의 lock flag를 가져와야 실행될 수 있습니다. 그런데, lock flag는 이미 첫 번째 스레드에서 소유하고 있기 때문에 두 번째 스레드에 의해 push() 메서드는 실행될 수가 없고. 두 번째 스레드는 대기 상태가 됩니다. 그동안 제어권은 다시 첫 번째 스레드로 넘어오게 되고, 멈춰있던 push() 메서드는 계속 수행하게 됩니다. 첫 번째 스레드에 의해 push() 메서드가 수행된 후에는 lock flag를 반납합니다. 그 후 lock flag가 반납되었기 때문에 대기하고 있던 두 번째 스레드가 제어권을 넘겨받아 수행될 수 있습니다. 이처럼 synchronized 메서드 또는 블록은 여러 스레드에 의해 공유데이터가 사용될 때 공유데이터가 스레드들로부터 데이터의 일관성을 갖도록 해 줄 수 있습니다.

다음 그림은 기본적인 스레드 상태도에 공유데이터 처리 부분을 추가한 것입니다.



앞의 그림을 보면 스레드 synchronized 메서드를 수행하면 lock flag를 받을 때까지 Lock pool에서 대기하게 되는 것을 볼 수 있습니다.

1.3.3. wait()와 notify()

synchronized로 스레드의 기본적인 문제는 처리되었지만 해결해야할 문제가 더 남아있습니다. 앞에서 설명한 스택 예제 프로그램을 다시 살펴보기로 하겠습니다. 만약 스택이 비어있는 상태에서 pop() 메서드가 수행된다면 어떻게 될까요? 이때에는 synchronized로도 해결할 수가 없습니다. 이 때 사용되는 메서드가 wait()와 notify()입니다.

앞에서 작성된 코드를 푸시와 팝이 동시에 실행되도록 join() 메서드 호출 부분을 삭제(주석처리해도 됩니다)한 다음 실행시켜보세요.

exam/java/chapter06/thread/SharedDataExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class SharedDataExample {
4:     public static void main(String[] args) {
5:        SharedData sd = new SharedData();
6:
7:        PushRunner pushr1 = new PushRunner("Push-하나 : ", sd);
8:        PushRunner pushr2 = new PushRunner("Push-둘 : ", sd);
9:
10:       Thread pusht1 = new Thread(pushr1);
11:       Thread pusht2 = new Thread(pushr2);
12:
13:       pusht1.start();
14:       pusht2.start();
15:
16: //    try {
17: //       pusht1.join();
```

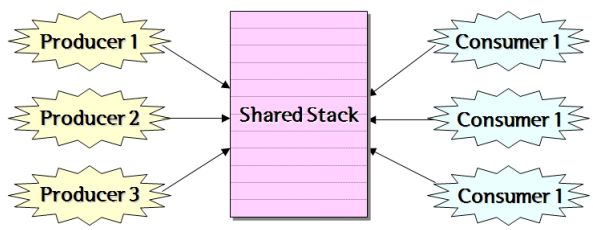
```
18: //        pusht2.join();
19: //    } catch (InterruptedException e) {
20: //        //nothing...
21: //    }
22:
23:        System.out.println("\n스택에 저장된 데이터는");
24:
25:        PopRunner popr1 = new PopRunner("Pop-하나 : ", sd);
26:        PopRunner popr2 = new PopRunner("Pop-둘 : ", sd);
27:
28:        Thread popt1 = new Thread(popr1);
29:        Thread popt2 = new Thread(popr2);
30:
31:        popt1.start();
32:        popt2.start();
33:
34:    }
35: }
```

위 코드는 스레드 4개를 실행시킵니다. 두 개의 스레드는 데이터를 푸시하고, 두 개의 스레드는 데이터를 팝하고 있습니다. 16라인부터 21라인까지 주석처리 한 다음 코드를 실행시키면 스레드 4개가 경쟁적으로 실행될 것입니다. 이 클래스를 실행시키면 아주 드물게 아래와 같은 예외가 발생하는 것을 볼 수 있습니다. 아래 결과는 이클립스에서 실행한 화면입니다.

```
Exception in thread "Thread-2" Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException: -1
        at exam.java.chapter06.thread.SharedData.push(SharedData.java:9)
        at exam.java.chapter06.thread.PushRunner.run(PushRunner.java:13)
        at java.lang.Thread.run(Unknown Source)
java.lang.ArrayIndexOutOfBoundsException: -1
        at exam.java.chapter06.thread.SharedData.pop(SharedData.java:19)
        at exam.java.chapter06.thread.PopRunner.run(PopRunner.java:13)
        at java.lang.Thread.run(Unknown Source)
```

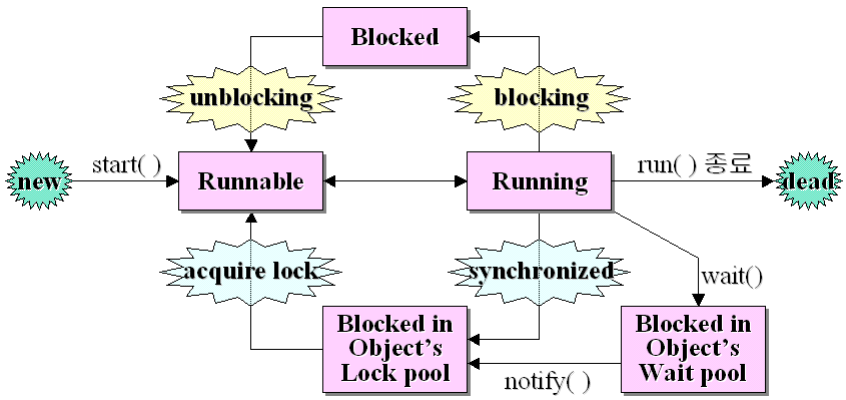
위와 같은 예외가 발생하는 이유는 스택에 데이터가 존재하지 않을 때 팝 스레드에 의해 데이터 팝이 일어날 경우입니다.

wait()와 notify() 에 대해 설명할 때에 생산자와 소비자 사이의 관계를 예로 설명하곤 합니다. 위의 예제를 수정하기 전에 생산자/소비자 문제에 대하여 설명하기 위해 다음 그림을 보면서 설명하겠습니다.



생성할 스레드의 수는 모두 6개(생산자 3개, 소비자 3개)입니다. 각각의 스레드가 공유데이터 영역(BakeStack)을 공유하고 있습니다. 생산자가 푸시 하여 데이터를 생산하면, 소비자는 팝 하여 데이터를 소비합니다. 그런데 생산자가 데이터를 푸시하기 전 또는 팝 할 데이터가 없을 때 소비자가 데이터를 팝 하려고 할 경우가 있습니다. 소비자 팝 할 데이터가 없을 때 팝을 하도록 한다면 예외가 발생할 수도 있습니다. 그래서 팝 할 데이터가 없다면 소비자에게 생산자가 데이터를 푸시 할 때까지 기다리도록 해야 할 것입니다(wait). 그리고 생산자는 데이터를 푸시하면 소비자에게 데이터가 푸시 되었음을 알려주어야 할 것입니다(notify).
이 6개의 스레드는 서로 경쟁하면서 실행되는데 공유데이터 영역인 스택을 깨뜨리지 않고, 푸시 된 데이터가 남아서도 안 되고 모자라서도 안 됩니다.

다음 그림은 스레드의 상태도에 wait()와 notify()메서드를 추가한 결과를 나타낸 것입니다.



다음은 앞에서 작성된 SharedData 클래스를 수정하여 푸시와 팝 스레드가 동시에 실행되어도 예외가 발생하지 않도록 한 코드입니다.

exam/java/chapter06/thread/SharedData.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class SharedData {
4:
5:     int index=0;
6:     char[] stack = new char[6];
7:
8:     public synchronized char push(char c) {
9:        notify();
10:       stack[index] = c;
11:       longJob();
12:       index++;
```

```
13:         return c;
14:     }
15:
16:     public char pop() {
17:         synchronized(this) {
18:             if(index==0) {
19:                 try {
20:                     wait();
21:                 } catch (InterruptedException e) {
22:                     System.out.println(e.getMessage());
23:                 }
24:             } else {
25:                 index--;
26:             }
27:             longJob();
28:             return stack[index];
29:         }
30:     }
31:
32:     public void longJob() {
33:         for(long i=0; i<20; i++) {
34:             System.out.print(".");
35:         }
36:         System.out.println();
37:     }
38: }
```

SharedData 클래스를 수정한 다음 SharedDataExample 클래스를 실행시켜 보세요. 이제 스택에 데이터가 없을 때 팝을 시도하더라도 예외가 발생하지 않습니다. pop() 메서드에서 스택에 데이터가 없을 경우(index가 0일 경우) wait() 메서드를 호출했습니다. 그리고 push() 메서드에서 notify() 메서드에 의해 데이터가 푸시 되었음을 알려주고 있습니다. 그런데 push() 메서드에서는 데이터를 푸시하기 전에 notify 하고 있습니다. 데이터를 푸시 한 다음 notify 해야 할 것 같지만 push() 메서는 synchronized 로 선언되어 있으므로 notify를 먼저 해도 wait 했던 스레드가 곧바로 실행되지 않습니다. 푸시하기 전에 notify 하면 wait하는 스레드는 바로 실행이 되는 것이 아니고 Lock flag를 다시 얻기 위해 Lock pool에서 대기합니다.

API문서에서 Thread 클래스를 보면 stop(), resume(), suspend() 등의 메서드가 deprecation되어 있는 것을 볼 수 있습니다. 그 이유는 이들 메서드는 lock flag를 가지고 있는 상태에서 봉쇄된 후 스레드가 비정상적으로 종료되었을 경우 lock flag를 반납하지 못하고 종료되므로 lock flag를 필요로 하는 다른 스레드가 lock flag를 얻을 수 없는 상태에 빠지는 데드락(Deadlock)이 발생할 수 있기 때문입니다.

앞의 wait()와 nitofy()를 설명하는 스레드 상태도 그림에서 wait()와 notify()에 의해 스레드의 상태를 분리하여 Wait pool로 표시한 이유는 wait() 메서드에 의해 스레드가 봉쇄될

때에는 lock flag를 반납하고 봉쇄되기 때문입니다. Wait pool에 있던 스레드는 notify() 메서드가 호출되면 반납했던 lock flag를 다시 얻기 위해 Lock pool로 빠지게 됩니다.

다음에 설명되는 예제들은 생산자/소비자 문제의 다른 예입니다.
모두 4개의 클래스(WaitNotifyExample, BakeStack, Baker, Customer)를 작성했습니다.
실행은 WaitNotifyExample 클래스를 이용합니다.

exam/java/chapter06/thread/WaitNotifyExample.java

```
1:  package exam.java.chapter06.thread;
2:
3:  public class WaitNotifyExample {
4:     public static void main(String[] args) {
5:        BakeStack bakeStack = new BakeStack();
6:
7:        Baker m1 = new Baker(bakeStack);
8:        Thread maker1 = new Thread (m1);
9:        maker1.start();
10:       Baker m2 = new Baker(bakeStack);
11:       Thread maker2 = new Thread (m2);
12:       maker2.start();
13:       Baker m3 = new Baker(bakeStack);
14:       Thread maker3 = new Thread (m3);
15:       maker3.start();
16:
17:       Customer c1 = new Customer(bakeStack);
18:       Thread customer1 = new Thread (c1);
19:       customer1.start();
20:       Customer c2 = new Customer(bakeStack);
21:       Thread customer2 = new Thread (c2);
22:       customer2.start();
23:       Customer c3 = new Customer(bakeStack);
24:       Thread customer3 = new Thread (c3);
25:       customer3.start();
26:    }
27: }
```

5라인은 공유데이터인 BakeStack 클래스의 객체(instance)를 만듭니다.
7라인부터 15라인은 3개의 Baker 클래스 객체를 만들고 이를 스레드화하여 실행시킵니다. 이들 스레드는 생산자 스레드입니다.
17라인부터 25라인은 3개의 Customer 클래스 객체를 만들고 이를 스레드화하여 실행시킵니다. 이들 스레드는 소비자 스레드입니다.
모두 6개의 스레드가 하나의 공유데이터(BakeStack)를 사용하고 있습니다.

이제 공유데이터인 BakeStack을 살펴보겠습니다.

exam/java/chapter06/thread/BakeStack.java

```
1:  package exam.java.chapter06.thread;
2:
```

```
3:  import java.util.Vector;
4:  public class BakeStack {
5:     private Vector buff = new Vector(10, 10);
6:
7:     public synchronized String pop() {
8:        String bread;
9:        while (buff.size() == 0) {
10:          try {
11:             this.wait();
12:          } catch (InterruptedException e) {
13:             e.printStackTrace();
14:          }
15:       }
16:       bread = (String)buff.remove(buff.size() - 1);
17:       return bread;
18:    }
19:    public synchronized void push(String bread) {
20:       this.notify();
21:       buff.addElement(bread);
22:    }
23: }
```

5라인에서는 스택을 좀 더 쉽게 구현하기 위해 Vector클래스를 사용했습니다.
9라인에서는 buff.size()가 0 이면 while문을 수행하는데 이는 스택이 비어 있음을 의미합니다. while문안에 wait() 메서드가 있는데 누군가 깨워줄 때까지 수행을 멈추겠다는 의미입니다. 여기서 synchronized를 사용해서 lock flag를 가져왔으나, wait()가 호출되면 lock flag는 반납됩니다. 그래야 다른 스레드가 pop()를 호출할 수 있기 때문입니다.
16라인은 스택이 비어있지 않으면 remove() 메서드를 호출해 빵을 스택에서 꺼내 넘겨줍니다.
20라인 notify() 메서드를 호출합니다. notify() 메서드는 대기 중인 스레드 중에서 하나를 임의로 선택하여 빵이 도착했다는 신호를 줍니다. 그러면 신호를 받은 스레드는 빵을 소비하게 됩니다. 이때 유의할 점은 notify() 메서드가 대기 중인 스레드에게 신호를 보내면 곧바로 대기 중인 스레드가 동작하지는 않는다는 것입니다. 물론 동작할 수도 있지만, 결정은 스레드 Scheduler에게 달려있으므로 push()가 몇 번 더 수행된 후에 대기 중인 스레드가 수행될 수 도 있습니다.
21라인은 addElement() 메서드를 호출하여 스택에 빵을 가져다 놓는 것입니다. 여기서의 의문사항은 addElement()보다 notify()를 먼저 실행하도록 했다는 것입니다. 즉, 빵을 가져다 놓기 전에 빵이 왔다는 신호를 한 것입니다. 이 의문점은 synchronized와 연관시켜 생각하면 쉽게 알 수 있을 것입니다.

이제 마지막으로 각각의 러너클래스들에 대해서 알아보기로 하겠습니다. 먼저 push()를 수행할 Baker 클래스입니다.

exam/java/chapter06/thread/Baker.java

```
1:  package exam.java.chapter06.thread;
2:
```

```
3:  public class Baker implements Runnable {
4:     private BakeStack bakeStack;
5:     private int num;
6:     private static int counter = 1;
7:     public Baker (BakeStack s) {
8:        bakeStack = s;
9:        num = counter++;
10:    }
11:    public void run() {
12:       String bread;
13:       for (int i = 0; i < 10; i++) {
14:          bread = getBread();
15:          bakeStack.push(bread);
16:          System.out.println("빵집" + num + " : " + bread);
17:          try {
18:             Thread.sleep((int)(Math.random()*300));
19:          } catch (InterruptedException e) {
20:             e.printStackTrace();
21:          }
22:       }
23:    }
24:    public String getBread() {
25:       String bread = null;
26:       switch ((int)(Math.random() * 3)) {
27:          case 0 :
28:             bread = "생크림 케익";
29:             break;
30:          case 1:
31:             bread = "식빵";
32:             break;
33:          case 2:
34:             bread = "고로케";
35:             break;
36:       }
37:       return bread;
38:    }
39: }
```

Runnable인터페이스를 implements한 전형적인 클래스이며, 11라인의 run() 메서드가 스레드에 의해 수행됩니다. 14라인에서는 getBread() 메서드를 호출하여 3개의 빵 중 하나를 반환합니다. 총 10 개의 빵을 만들어 냅니다. 18라인은 sleep() 메서드를 이용하여 스레드를 잠깐 정지시켰는데 컴퓨터 속도가 너무 빨라 순간적으로 처리되기 때문에 일부러 다른 스레드에게 기회를 주기 위해서 잠깐 쉬게 하는 것입니다.

다음은 Customer클래스입니다. 이 클래스도 앞의 Baker클래스와 거의 유사합니다.

exam/java/chapter06/thread/Customer.java

```
1:  package exam.java.chapter06.thread;
2:
```

```
3:  public class Customer implements Runnable {
4:     private BakeStack bakeStack;
5:     private int num;
6:     private static int counter = 1;
7:     public Customer (BakeStack s) {
8:        bakeStack = s;
9:        num = counter++;
10:    }
11:    public void run() {
12:       String bread;
13:       for (int i = 0; i < 10; i++) {
14:          bread = bakeStack.pop();
15:          System.out.println("손님" + num + " : " + bread);
16:          try {
17:             Thread.sleep((int)(Math.random()*300));
18:          } catch (InterruptedException e) {
19:             e.printStackTrace();
20:          }
21:       }
22:    }
23: }
```

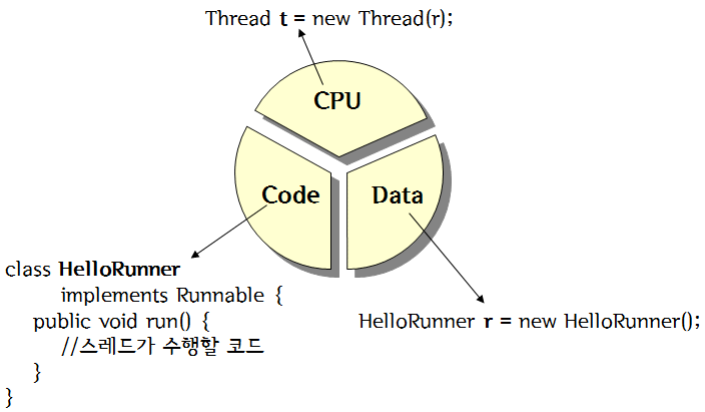
이 클래스는 공유 스택에서 빵을 10번 가져와 화면에 보여줍니다. Baker클래스와 같은 내용이므로 설명은 생략하겠습니다.

1.4. 요점 정리

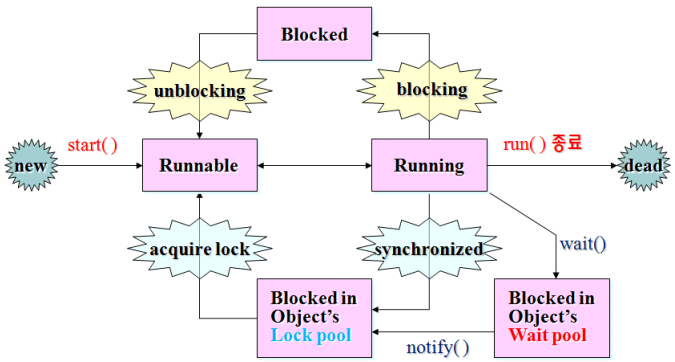
1. 스레드

최소 수행 단위

Runnable 인터페이스를 implements한 후 run()메서드에 스레드가 수행할 코드를 작성



2. 스레드 상태



3. 스레드 클래스 메서드

스레드 우선순위 : 1(가장 낮음)~10(가장 높음), 디폴트 5

setPriority(), getPriority() : 우선순위를 변경하거나 알아보는 메서드

sleep(ms) : 스레드의 실행을 잠시 멈추게 함

join() : join 이후 start 되는 스레드는 join 한 스레드가 종료해야 실행됨

yield() : 같은 우선순위를 가진 다른 스레드에게 먼저 프로세스를 점유하도록 함

4. 공유데이터

synchronized, wait() 와 notify()

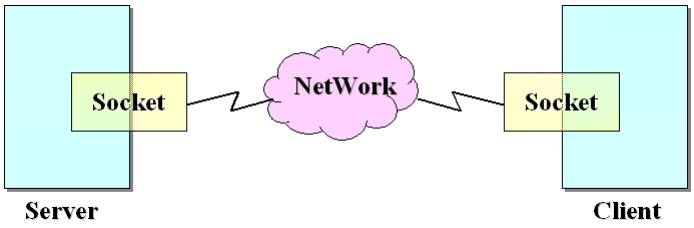
2. 네트워크 프로그래밍

이번 장에서는 TCP방식의 네트워크 프로그래밍과 UDP방식의 프로그램에 대하여 설명합니다. 통신을 하기 위한 기본적인 개념과 관련 클래스들에 대해 알아보겠습니다. 자실 자바로 네트워크 프로그램을 만들 일은 거의 없을지도 모릅니다. 왜냐하면 자바는 WAS(Web Application Server)를 이용해 대부분의 서비스를 제공하니까요. 그러나 최소한의 네트워크프로그래밍 관련 API는 알아두는 것이 좋습니다.

주요 내용입니다.

- TCP 네트워크 프로그래밍
- UDP 네트워크 프로그래밍
- 채팅 프로그램 샘플 1
- 채팅 프로그램 샘플 2
- 파일 서버프로그램 샘플

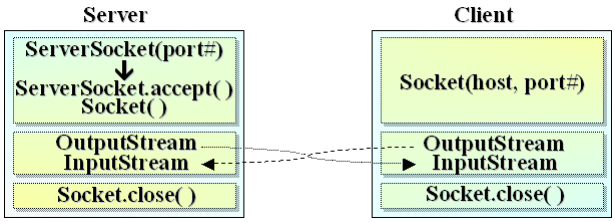
네트워크에는 소켓(Socket)이 있는데 소켓은 다음 그림에서처럼 어떤 프로그래밍 모델에서 프로세스 사이의 통신 종단점을 의미합니다.



통신이 이루어지려면 먼저 연결설정을 하고 주소를 지정해야 합니다. 연결을 설정하려면 한쪽 컴퓨터(서버)는 연결을 대기하는 프로그램을 실행해야 하고, 다른 쪽 컴퓨터(클라이언트)는 서버로 연결을 시도해야 합니다. 이때 클라이언트가 서버에 연결되려면 서버주소와 포트번호를 알아야 합니다(포트 : 하나의 서버에서 다른 네트워크 서비스를 제공하기 위해서 사용). 포트번호는 TCP/IP시스템에서는 16비트 크기를 가지며 범위는 0~65535 사이의 값을 가질 수 있지만 1023번 이하의 포트(0~1023) 번호는 시스템이 미리 지정된 서비스용(http : 80, ftp : 21 등)으로 사용하기 때문에 1023번 이하의 포트번호는 사용하지 않는 것이 좋습니다.
통신방법은 여러 가지가 있지만 자바에서는 크게 TCP 통신과 UDP통신으로 나눌 수 있습니다.

2.1. TCP 네트워크 프로그램

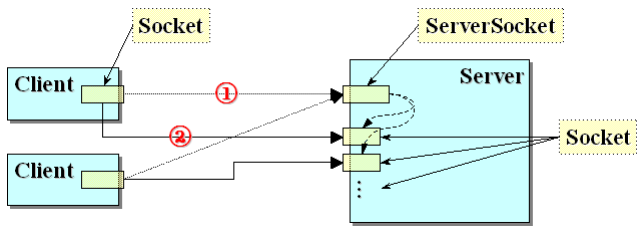
TCP 통신은 가장 많이 이용하는 통신 방식으로 “양방향의 스트림 통신을 제공하는, 신뢰성 있는 연결 지향형 통신방식”을 의미합니다. 양방향은 “클라이언트와 서버가 동시에 존재해야만 통신이 이루어짐”을 의미하고, 신뢰성은 “양단에서 데이터를 주고받을 때 데이터의 손실 없이 정확히 주고받을 수 있음”을 의미합니다. 연결 지향형은 “클라이언트와 서버가 상호 연결되어야 비로소 통신이 이루어짐”을 의미합니다. TCP/IP 소켓 연결은 java.net 패키지의 클래스를 사용하여 구현합니다.
다음 그림은 서버와 클라이언트에서 어떤 일이 일어나는 지를 보여주는 그림입니다. 그림에서 서버는 java.net 패키지의 ServerSocket클래스를 이용하여 포트 번호를 할당합니다. 클라이언트가 소켓객체를 생성하여 연결을 요청하면 서버는 accept() 메서드를 사용하여 소켓을 열어주고, 클라이언트는 서버주소의 포트번호로 연결을 요청합니다.



서버와 클라이언트의 프로세스가 정보를 교환할 때 스트림모델을 사용하는데, 이 때 소켓에는 두 개의 스트림 즉, 입력스트림(InputStream)과 출력스트림(OutputStream)이 들어있습니다. 임의의 프로세스가 다른 프로세스에게 데이터를 보내려면 소켓과 연관된 출력스트림에 기록하면 되고, 상대측 프로세서가 데이터를 읽을 때도 소켓과 연관된 입력 스트림을 읽기만하면 됩니다.

2.1.1. TCP 서버

TCP/IP 서버 응용 프로그램은 ServerSocket과 Socket 네트워크 클래스를 이용합니다. ServerSocket클래스는 서버를 설정하는 일을 합니다. 서버에서는 서버소켓을 생성한 다음 클라이언트의 접속을 대기해야 합니다. 그리고 클라이언트가 접속되면 서버에서는 임의의 소켓을 생성하여 클라이언트와 통신해야 합니다. 그러기 위해서 서버에서는 다른 두개의 소켓을 선언해야 하는데, 그 이유를 알아보기 위해 다음 그림을 참고로 설명하기로 하겠습니다.



서버는 동시에 여러 클라이언트를 수용할 수 있어야 합니다. 따라서 앞의 그림처럼 여러 클라이언트가 서버에 접속하고 이를 처리할 있도록 ServerSocket를 열어둡니다. 이것은 일종의 Listener역할을 수행합니다.(접수창구정도로 이해하면 됩니다.)

① ServerSocket은 클라이언트의 접속을 기다리고 있고 클라이언트는 ServerSocket으로 접속을 시도합니다.

② ServerSocket과 클라이언트 Socket의 접속이 이루어지면, ServerSocket은 새로운 Socket을 생성해서 클라이언트의 요청을 처리하도록 합니다. 그런 다음 ServerSocket은 다시 다른 클라이언트의 요청을 기다립니다.

이러한 방식을 이용하기 때문에 서버에서는 여러 클라이언트의 요청을 처리할 수 있는 것입니다.

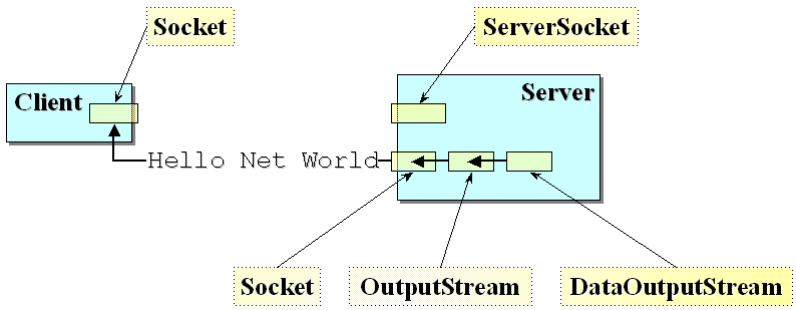
다음 코드는 통신에 필요한 서버의 역할을 작성한 예입니다.

exam/java/chapter09/tcp/SimpleServer.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.net.*;
4:  import java.io.*;
5:
6:  public class SimpleServer {
7:     public static void main(String args[]) {
8:        String[] messages = {
9:              "누가 당신을 시비거리에 올려놓고 있습니다. 강한 반발이 예상됩니다.",
10:             "새로운 일을 시작하기는 좋으나 처음부터 무리한 계획은 자제하세요.",
11:             "주변에 누군가가 당신을 좋아합니다. 주위를 천천히 돌아보세요.",
12:             "편안한 마음으로 생활할 수 있지만, 저녁에 사소한 고민거리가 생깁니다.",
13:             "어려운 일에 처한다고 당황하지 마세요. 곧 일의 실마리를 찾을 수 있습니다."
14:          };
15:
16:       ServerSocket serverSocket = null;
17:
18:       try {
19:          serverSocket = new ServerSocket(5432);
20:       } catch (IOException e) {
21:          e.printStackTrace();
22:       }//end try~catch
23:
24:       while (true) {
25:          int rand = (int)(Math.random() * messages.length);
26:          try {
27:             System.out.println("사용자의 접속을 기다립니다.");
28:             Socket newSocket = serverSocket.accept();
29:
30:             System.out.println(newSocket.getRemoteSocketAddress() + "에서
    접속.");
31:
32:             OutputStream os = newSocket.getOutputStream();
33:             DataOutputStream dos = new DataOutputStream(os);
34:             dos.writeUTF(messages[rand]);
35:
36:             dos.close();
37:             newSocket.close();
38:             System.out.println("사용자의 접속을 종료합니다.");
39:          } catch (IOException e) {
40:             e.printStackTrace();
41:          }//end try~catch
42:       }//end while
```

```
43:     }//end main()
44: }//end class
```

이 예제는 사용자와의 접속을 지속적으로 유지하지 않습니다. 클라이언트가 접속하면 서버에서는 메시지를 보내주고 클라이언트와의 연결을 닫습니다.
16라인에서 서버소켓을 선언하고(ServerSocket) 19라인에서 ServerSocket을 만들고, ServerSocket생성자의 인자는 포트번호를 부여합니다. 즉, "서버가 5432번 포트를 열고 클라이언트의 접속을 기다립니다."
25라인에서는 while 문이 무한 반복을 하는데, 이는 서버 프로그램을 계속 수행시키겠다는 의미입니다.
28라인에서는 serverSocket.accept()를 호출했는데, 이때 서버는 클라이언트가 접속할 때까지 대기하게 됩니다. 클라이언트가 접속하면, accept()메서드는 새로운 소켓이 생성하여 반환합니다.
32라인에서는 소켓으로부터 출력스트림을 얻어옵니다. 그리고 이를 이용해 33라인에서 필터스트림인 DataOutputStream을 생성하고, 이를 통해 34라인에서 writeUTF() 메서드를 호출해 정해진 메시지 중 하나를 소켓을 통해 출력합니다. 이렇게 저장한 문자열은 다음그림의 경로를 따라 클라이언트로 전송된다.



2.1.2. TCP 클라이언트

클라이언트 프로그램은 위의 서버처럼 복잡하지 않습니다. 그리고 서버에 접속한 후 전송되는 데이터는 모두 문자열 타입으로 전송된다는 사실을 기억하고 프로그래밍 해야 합니다. (이를 프로토콜이라고 하며 일종의 서버와 클라이언트간의 통신규약 즉, 약속을 의미합니다.)

다음 프로그램은 통신에 필요한 클라이언트의 역할을 작성한 예입니다.

exam/java/chapter09/tcp/SimpleClient.java

```
1:  package exam.java.chapter09.tcp;
2:
```

```
3:  import java.net.*;
4:  import java.io.*;
5:
6:  public class SimpleClient {
7:     public static void main(String args[]) {
8:        try {
9:           Socket newSocket = new Socket("127.0.0.1", 5432);
10:          InputStream is = newSocket.getInputStream();
11:          DataInputStream dis = new DataInputStream(is);
12:          System.out.println(dis.readUTF());
13:          dis.close();
14:          newSocket.close();
15:       } catch (ConnectException connExc) {
16:          System.err.println("서버연결 실패");
17:       } catch (IOException e) {
18:          e.printStackTrace();
19:       }
20:    }
21: }
```

9라인에서는 클라이언트도 소켓을 생성해야하므로 Socket객체를 만들었습니다. 이때 생성자의 인자로 서버의 IP 주소와 포트번호를 지정하고 있습니다. 127.0.0.1은 루프백(loopback) 주소를 의미하며, 자신 컴퓨터의 IP 주소를 가리킬 때 사용합니다(이렇게 지정한 이유는 서버측 프로그램도 자기 컴퓨터에서 수행되기 때문입니다.). 포트번호 5432를 기술한 이유는 서버에서 서버소켓을 오픈할 때 5432번으로 포트를 지정했기 때문입니다. 만약 다른 포트로 지정한다면 서버에 연결되지 않습니다. 9라인이 실행되면 서버에 연결됩니다. 작성된 코드대로라면 서버와 연결되면 연결과 동시에 서버는 문자열을 클라이언트 소켓으로 보냅니다. 즉, 접속하는 순간 서버는 문자열을 보내고 클라이언트는 이 문자열을 소켓에서 읽기만 하면 됩니다.
10라인에서는 소켓에서 InputStream 객체를 얻고, DataInputStream을 통해 문자열을 읽을 준비를 합니다.
12라인의 readUTF() 메서드를 통해 문자열을 읽어 화면에 출력합니다.

앞의 예제 프로그램을 실행시키려면 명령프롬프트 창에서 서버용 프로그램을 실행시킨 다음 다른 명령프롬프트 창에서 클라이언트를 실행하면 결과를 볼 수 있습니다.

```
java SimpleServer
```

```
java SimpleClient
```

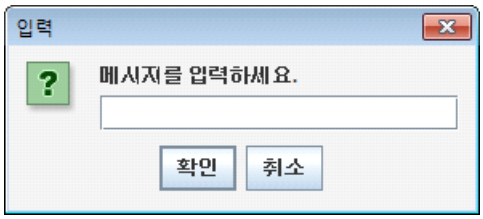
2.1.3. 간단한 채팅

다음 코드는 스레드를 사용하여 1대1 채팅하는 예입니다. 이 예제는 3개의 클래스 ChatServer, Sender, Receiver로 구성되어 있습니다. 그 중에서 Sender 클래스는 입력한 내용을 보내는 클래스이며, Receiver 클래스는 상대방으로부터 전달된 메시지를 화면에 뿌리는 클래스입니다.

exam/java/chapter09/tcp/SimpleChatServer.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.io.DataInputStream;
4:  import java.io.DataOutputStream;
5:  import java.io.IOException;
6:  import java.net.ServerSocket;
7:  import java.net.Socket;
8:
9:  import javax.swing.JOptionPane;
10:
11: public class SimpleChatServer {
12:     public static void main(String[] args) {
13:         ServerSocket server = null;
14:         int port = 7777;
15:
16:         try {
17:             server = new ServerSocket(port);
18:             Receiver receiver = new Receiver(server);
19:
20:             Sender sender = new Sender(port);
21:
22:             Thread rt = new Thread(receiver);
23:             Thread st = new Thread(sender);
24:
25:             rt.start();
26:             st.start();
27:         }catch(IOException e) {
28:             System.out.println(e.getMessage());
29:         }
30:     }
31: }
32:
33: class Sender implements Runnable {
34:     String ip = "127.0.0.1";
35:     int port;
36:
37:     public Sender(int port) {
38:         this.port = port;
39:     }
40:
41:     public void run() {
42:         while(true) {
```

```
43:           Socket socket = null;
44:           DataOutputStream dos = null;
45:           try {
46:              String message = JOptionPane.showInputDialog("메시지를 입력하세요.");
47:              if(message==null) {
48:                 System.exit(0);
49:              }
50:
51:              socket = new Socket(ip, port);
52:              dos = new DataOutputStream(socket.getOutputStream());
53:
54:              dos.writeUTF(message);
55:              System.out.println("보낸 메시지: " + message);
56:           } catch(Exception e) {
57:              e.printStackTrace();
58:           } finally {
59:              if(socket!=null) try{socket.close();}catch(Exception e) {}
60:              if(dos!=null) try{dos.close();}catch(Exception e) {}
61:           }
62:        }
63:     }//end run()
64: }//end Sender
65:
66: class Receiver implements Runnable {
67:     ServerSocket server;
68:     public Receiver(ServerSocket server) {
69:        this.server = server;
70:     }
71:     public void run() {
72:        while(true) {
73:           Socket socket = null;
74:           DataInputStream dis = null;
75:           try {
76:              socket = server.accept();
77:              dis = new DataInputStream(socket.getInputStream());
78:
79:              String message = dis.readUTF();
80:              System.out.println(socket.getInetAddress() + ":" + message);
81:           } catch(Exception e) {
82:              e.printStackTrace();
83:           } finally {
84:              if(socket!=null) try{socket.close();}catch(Exception e) {}
85:              if(dis!=null) try{dis.close();}catch(Exception e) {}
86:           }
87:        }
88:     }//end run()
89: }//end Receiver
```



상대방에게 전달하고 싶은 메시지를 입력한 다음 확인 버튼을 누르세요. 프로그램을 종료하고 싶으면 취소 버튼을 누르세요. 29라인의 아이피를 수정하면 실제 상대방과 대화를 할 수 있습니다. 대화 내용을 콘솔화면에 나타납니다. 이클립스에서 실행하면 Console 탭에 채팅 내용이 나타납니다.

2.2. UDP 네트워크 프로그램

TCP/IP가 연결 중심의 프로토콜이라면 UDP(User Datagram Protocol) 통신방식은 "비신뢰적, 비 연결지향형 통신"이라고 할 수 있습니다. TCP 통신과는 달리 우편과 유사합니다. TCP에서는 전화를 사용하는 것처럼 메시지를 순서대로 보내고 받을 수 있지만 UDP는 보낸 순서와 받는 순서가 다를 수 있습니다. 또 상대방의 주소가 잘못되면 데이터가 잘못 전달되거나 아예 데이터를 읽지 못할 수도 있습니다.
UDP는 DatagramSocket과 DatagramPacket이라는 두 개의 클래스를 지원합니다. 패킷은 송신자의 정보와 메시지길이, 메시지 등으로 구성되는 독립적인 메시지 단위입니다.

2.2.1. DatagramPacket

DatagramPacket에는 다음 네 가지의 생성자가 있는데, 2개는 데이터를 수신하는데 사용하고 나머지 2개는 데이터를 보내는데 사용합니다.

- DatagramPacket(byte[] buf, int length)
- DatagramPacket(byte[] buf, int offset, int length)
  - UDP 패킷을 수신할 수 있도록 바이트 배열을 설정합니다. 생성자로 사용되는 바이트 배열은 비워두고, 읽을 바이트의 크기를 설정합니다. 이때 배열의 크기보다 작게 지정합니다.
- DatagramPacket(byte[] buf, int length, InetAddress address, int port)
- DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)
  - 전송할 수 있도록 UDP 패킷을 설정하는 데 사용합니다.

2.2.2. DatagramSocket

DatagramSocket은 UDP 패킷을 읽고 쓰는데 사용합니다. 이 클래스에는 연결할 포트와 인터넷 주소를 지정하는데 사용하는 세 개의 생성자를 가지고 있습니다.

- DatagramSocket()
  - 로컬 호스트에서 이용할 수 있는 포트로 연결합니다.
- DatagramSocket(int port)
  - 로컬 호스트에서 지정된 포트로 연결합니다.
- DatagramSocket(int port, InetAddress laddr)
  - 지정된 주소의 지정된 포트 번호로 연결합니다.

다음 프로그램은 UDP방식을 이용하여 메신저 프로그램을 만든 예입니다.

exam/java/chapter09/udp/Messenger.java

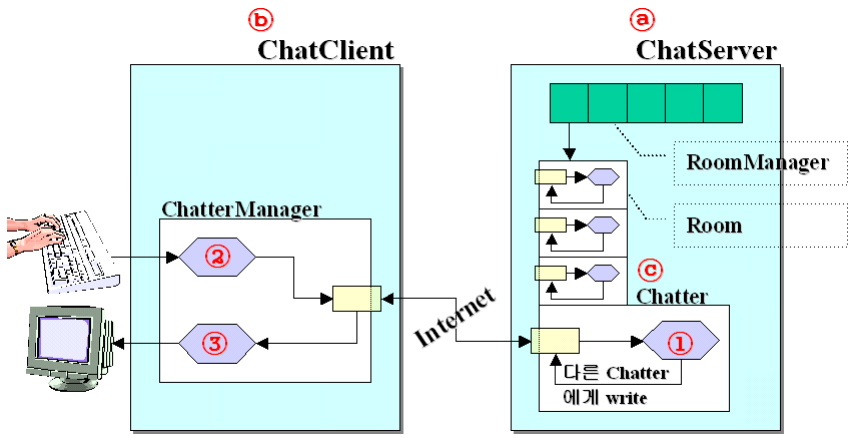
```
1:  package exam.java.chapter09.udp;
2:
3:  import java.awt.*;
4:  import java.awt.event.*;
5:  import java.io.*;
6:  import java.net.*;
7:
8:  public class Messenger implements Runnable, ActionListener {
9:
10:    private Frame f;
11:    private TextArea outputArea;
12:    private TextField addressField, inputField;
13:
14:    private DatagramSocket server, client;
15:    private DatagramPacket sinData, soutData;
16:
17:    private byte[] data = new byte[500];
18:
19:    public Messenger() {
20:       try {
21:          server = new DatagramSocket(8000);
22:          client = new DatagramSocket(7000, InetAddress.getLocalHost());
23:       }catch(IOException e) {
24:          e.printStackTrace();
25:       }
26:    } //end of constructor
27:
28:    public static void main(String[] args) {
29:       Messenger m = new Messenger();
30:       m.go();
31:       Thread t = new Thread(m);
32:       t.start();
33:    } //end of main
34:
35:    public void go() {
36:       try {
37:          f = new Frame(InetAddress.getLocalHost().getHostAddress());
38:       }catch(UnknownHostException e) {
39:          e.printStackTrace();
40: }
41:       f.addWindowListener(new WindowAdapter() {
42:          public void windowClosing(WindowEvent e) {
43:             System.exit(0);
44:          }
45:       });
46:
47:       outputArea = new TextArea();
```

```
48:        outputArea.setEditable(false);
49:
50:        addressField = new TextField();
51:        inputField = new TextField();
52:
53:        inputField.addActionListener(this);
54:
55:        Panel p1 = new Panel();
56:        p1.setLayout(new BorderLayout());
57:        p1.add(new Label("Address"), BorderLayout.WEST);
58:        p1.add(addressField, BorderLayout.CENTER);
59:
60:
61:
62:        Panel p2 = new Panel();
63:        p2.setLayout(new BorderLayout());
64:        p2.add(new Label("Message"), BorderLayout.WEST);
65:        p2.add(inputField, BorderLayout.CENTER);
66:
67:        f.add(p1, BorderLayout.NORTH);
68:        f.add(outputArea, BorderLayout.CENTER);
69:        f.add(p2, BorderLayout.SOUTH);
70:
71:        f.setSize(300,200);
72:        f.setVisible(true);
73:     } //end of go
74:
75:     public void actionPerformed(ActionEvent e) {
76:        String mssg = inputField.getText();
77:        String ip = addressField.getText();
78:
79:        outputArea.append(">> " + mssg+"\n");
80:
81:        InetAddress inet = null;
82:        soutData = null;
83:
84:        try {
85:           inet = InetAddress.getByName(ip);
86:
87:           soutData = new DatagramPacket(mssg.getBytes(),
    mssg.getBytes().length, inet, 8000);
88:
89:           client.send(soutData);
90:        } catch(Exception ex) {
91:           ex.printStackTrace();
92:        }
93:
94:        inputField.setText("");
95:
96:        if(ip==null) {
```

```
97:           try {
98:              ip = InetAddress.getLocalHost().getHostName();
99:           }catch(UnknownHostException ex) {
100:             ex.printStackTrace();
101:          }
102:       }
103:    } //end of actionPerformed
104:
105:    public void run() {
106:       while(true) {
107:          sinData = new DatagramPacket(data, data.length);
108:
109:          try {
110:             System.out.println("8000번 포트로 대기중...");
111:             server.receive(sinData);
112:          }catch(IOException e) {
113:             e.printStackTrace();
114:          }
115:
116:          String addr = sinData.getAddress().getHostName();
117:          String rsvData = new String(sinData.getData(), 0,
    sinData.getLength());
118:
119:          outputArea.append("[" + addr + "]" + rsvData + "\n");
120:       } //end of while
121:    } //end of run
122:} //end of class
```

2.3. TCP 채팅 프로그램 I

TCP를 이용하여 채팅 프로그램을 만들어 보겠습니다. 먼저 채팅 프로그램의 원리를 이해하기 위해 다음의 그림을 설명하기로 합니다.



약간 복잡한 그림으로 이 내용을 프로그램으로 옮겨도 상당히 복잡하기 때문에 그림을 충분히 이해를 한 후 프로그램을 이해하시기 바랍니다.

ⓐ의 ChatServer는 대화방(chatting room)을 관리할 수 있는 구조를 갖습니다.(물론 이 예제에서는 단순하게 대기실 한 곳에서 chatting을 하도록 구현했습니다. 이 코드를 더 발전시키면 방을 만들 수 있을 것입니다.) RoomManager라는 클래스가 방을 관리하는데 이를 위해 Vector 클래스를 사용하였습니다. 그 다음 Room이라는 클래스는 대화방에 속해 있는 대화자를 관리하는데, 역시 Vector 클래스를 통해 관리합니다.

ⓒ의 Chatter 클래스는 대화자가 접속하면 인스턴스가 서버에 자동으로 생성되며 접속된 대화자의 모든 정보를 관리합니다. Chatter 클래스에는 소켓을 가지고 있는데, 실제 ChatterClient(ⓑ)의 소켓과 연결되어 있습니다. ChatterClient가 메시지를 보내면, Chatter 클래스의 소켓에 전달되고, 이를 기다리던 스레드(①)는 데이터를 읽어서 대화방에 있는 다른 대화자에게 메시지를 전달하게 되는 것입니다.

ⓑ의의 ChatClient는 2개의 스레드가 있습니다. 먼저 ②의 스레드는 대화자로부터 메시지를 입력받습니다. 그런데 키보드에서 입력받고 있는 동안 다른 대화자가 메시지를 보내는 것을 수신하기 위해 스레드를 이용해서 해결해야 합니다. 이는 입력하는 동안 출력을 담당하는 스레드(③)가 소켓을 감시하면 해결할 수 있습니다. 소켓에 메시지가 도착하면 이를 화면에 출력하는데 스레드를 이용하지 않는다면 글을 쓰는 도중에 상대방이 보낸 메시지를 읽을 수 없을 것입니다.

2.3.1. ChatServer.java

어려운 코드는 아니지만, 그렇게 쉬운 코드도 아닙니다. 먼저 ChatServer부터 프로그램을 설명하기로 하겠습니다.

exam/java/chapter09/tcp/ChatServer.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import  java.io.*;
4:  import  java.net.*;
5:  import  java.util.*;
6:
7:  public class ChatServer {
8:
9:     public static void main(String [] args) {
10:
11:       System.out.println("Chatting Server Starting.");
12:
13:       int portNo = 5555;
14:
15:       if (args.length == 1 ) {
16:          portNo = Integer.parseInt(args[0]);
17:       }
18:
19:       ChatManager cm= new ChatManager(portNo);
20:    }
21:
22: }
```

앞의 ChatServer 프로그램이 전부는 아닙니다. 이 부분은 서버를 실행시키기 위해 포트번호를 입력받고, 서버의 모든 대화를 담당할 ChatManager 객체를 만듭니다. 그리고 CharManager의 인자로 서버의 포트번호를 명령행인자로 받아 넘겨줍니다.

이 프로그램은 다음에 있는 코드들을 모두 작성해야 실행이 가능합니다.

다음 프로그램은 채팅서버에 접속하는 채팅사용자 클래스입니다. Chatter 클래스는 실제 접속한 대화자의 정보를 가지고 있습니다.

```
23: class Chatter {
24:    private Socket clientSocket;
25:    private BufferedReader br;
26:    private PrintWriter pw;
27:    private ChatRoom chatRoom;
28:    private String chatterID;
29:
```

```
30:    Chatter(ChatRoom chatRoom,
31:       Socket clientSocket, String chatterID ) {
32:       System.out.println("Chatter 생성 : " + chatterID);
33:       try {
34:          this.chatRoom = chatRoom;
35:          this.clientSocket = clientSocket;
36:          this.chatterID = chatterID;
37:          br = new BufferedReader(new InputStreamReader(
    clientSocket.getInputStream()));
38:          pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(
    clientSocket.getOutputStream())));
39:          (new readSocketThread()).start();
40:       } catch (Exception e) {
41:          System.out.println(e.toString());
42:       }
43:    }
44:    public void sendMessage(String message) {
45:       pw.println(message);
46:       pw.flush();
47:    }
48:
49:    class readSocketThread extends Thread {
50:       String inputString = null;
51:       public void run() {
52:          try {
53:             while (true) {
54:                inputString = br.readLine();
55:                chatRoom.chatEveryChatter(inputString);
56:             }
57:          } catch ( Exception e ) {
58:             System.out.println(e.toString());
59:          }
60:       }
61:    }
62: }
```

49라인에서는 스레드를 통해 대화자가 서버로 전달하는 메시지를 감시합니다. 앞에서 설명한 그림의 ① 에 해당됩니다.

다음 프로그램은 대화방에 해당하는 클래스입니다.

```
63: class ChatRoom {
64:    private String roomName;
65:    private Vector joinChatters = new Vector();
66:    private Chatter roomMaker;
67:
68:    ChatRoom(String roomName) {
69:       System.out.println("채팅방 개설 : " + roomName);
70:       this.roomName = roomName;
```

```
71:        this.roomMaker = null;
72:     }
73:
74:     public synchronized void joinChatter(Chatter chatter) {
75:        joinChatters.add(chatter);
76:     }
77:
78:     public synchronized String getName() {
79:        return roomName;
80:     }
81:
82:     public synchronized void chatEveryChatter(String message) {
83:        for ( int i=0 ; i < joinChatters.size() ; i++) {
84:           ((Chatter)joinChatters.get(i)).sendMessage(message);
85:        }
86:     }
87:
88:     public int size() {
89:        return joinChatters.size();
90:     }
91:
92: }
```

82라인의 chatEveryChatter() 메서드는 방안에 있는 모든 대화자에게 메시지를 보낼 때 사용합니다.

다음 프로그램은 대화방을 관리하는 클래스입니다. ChatRoomManager 클래스는 생성과 동시에 "대기실"을 만듭니다.

```
93: class ChatRoomManager {
94:     private Vector chatRooms = new Vector();
95:
96:     ChatRoomManager() {
97:        System.out.println("ChatRoomManager Starting.");
98:        chatRooms.add(new ChatRoom("대기실"));
99:     }
100:
101:    public void makeRoom(String roomName) { }
102:
103:    public void deleteRoom(String roomName) { }
104:
105:    public void enterRoom(String roomName, Socket clientSocket) {
106:       Chatter chatter = null;
107:       ChatRoom tempRoom = null;
108:       boolean exitFor = false;
109:       int i;
110:       for(i=0; (exitFor == false)&&(i < chatRooms.size()); i++ ) {
111:          tempRoom = (ChatRoom)chatRooms.get(i);
112:          if (tempRoom.getName().equals(roomName)) {
```

```
113:             chatter = new Chatter(tempRoom, clientSocket,
    String.valueOf(tempRoom.size() + 1));
114:             tempRoom.joinChatter(chatter);
115:             exitFor = true;
116:          }
117:       }
118:    }
119:
120:    public void exitRoom(Chatter chatter) { }
121:}
```

105라인의 enterRoom() 메서드를 통해, 현재 존재하는 모든 대화방을 찾아서 해당 대화방을 찾고 그곳에 Chatter를 생성합니다. 이 클래스는 대화방을 관리합니다.

다음 코드는 채팅 서버를 전체적으로 관리하는 클래스입니다.

```
122:class ChatManager {
123:    private int serverPort;
124:    private ServerSocket serverSocket;
125:    private ChatRoomManager chatRoomManager;
126:
127:    ChatManager(int serverPort ) {
128:       System.out.println("Chatting Manager Starting");
129:
130:       try {
131:          this.serverPort = serverPort;
132:          chatRoomManager = new ChatRoomManager();
133:          serverSocket = new ServerSocket(serverPort);

134:          new listenerThread().start();
135:       } catch (Exception e) {
136:          System.out.println(e.toString());
137:       }
138:    }
139:
140:    class listenerThread extends Thread {
141:       private boolean stopListener = false;
142:       Socket clientSocket = null;
143:       public void run() {
144:          try {
145:             while ( !stopListener ) {
146:                System.out.println("Watting Client...");
147:                clientSocket = serverSocket.accept();
148:          chatRoomManager.enterRoom("대기실",clientSocket);
149:                System.out.println("Connection Established form:" +
    clientSocket.getInetAddress().getHostAddress());
150:             }
151:          } catch (Exception e ) {
152:             System.out.println(e.toString());
```

```
153:          }
154:       }
155:    }
156:}
```

123라인의 ChatManager 클래스는 133라인에서 ServerSocket 객체를 만듭니다. 그리고 134라인은 140라인에서 선언된 ListenerThread라는 내부 클래스의 객체를 생성합니다. 140라인에 선언된 내부 클래스가 스레드 클래스 입니다. 따라서 start() 메서드를 통해 스레드가 시작되면 run() 메서드는 루프를 돌면서 클라이언트의 접속을 기다리다 접속이 이루어지면 일반 소켓을 만든 후, "대기실"에 Chatter 클래스의 객체를 생성시키기 위해 ChatRoomManager의 enterRoom() 메서드를 호출합니다. 그리고 계속 루프를 돌면서 다른 클라이언트의 접속을 기다립니다.

복잡한 내용을 간단하게 설명했는데 이를 근거로 프로그램을 자세히 살펴보면서 직접 이해하기 바랍니다. 물론 이 프로그램에는 불필요한 부분도 있습니다. 그 이유는 GUI 버전으로 작성하기 위해 추가된 부분이 있기 때문입니다. 따라서 이 코드를 응용해서 GUI 용 채팅 애플리케이션을 만들 수도 있습니다.

2.3.2. ChatClient.java

다음 프로그램은 채팅에 필요한 ChatClient의 클래스입니다.

exam/java/chapter09/tcp/ChatClient.java

```
1: package exam.java.chapter09.tcp;
2:
3: import java.io.*;
4: import java.net.*;
5:
6: public class ChatClient {
7:
8:    public static void main (String [] args) {
9:
10:      String addr = "127.0.0.1";
11:      int portNo = 5555;
12:
13:      if (args.length == 2 ) {
14:         addr = args[0];
15:         portNo = Integer.parseInt(args[1]);
16:      }
17:
18:      ChatterManager cm = new ChatterManager(addr, portNo);
19:
```

```
20:    }
21: }
22:
```

클라이언트 프로그램은 비교적 짧습니다. 8라인의 main() 메서드에서 ChatterManager의 객체를 생성합니다. 이 프로그램 역시 아래 이어지는 코드를 모두 작성해야 실행이 가능합니다.

다음 코드는 ChatterManager클래스입니다. 이 클래스가 대부분의 클라이언트 채팅을 담당합니다.

```
23: class ChatterManager {
24:
25:    private String serverIP;
26:    private int serverPort;
27:    private Socket clientSocket;
28:    private BufferedReader br;
29:    private PrintWriter pw;
30:    private BufferedReader keyboard;
31:
32:    ChatterManager(String serverIP, int serverPort) {
33:       try {
34:
35:          this.serverIP = serverIP;
36:          this.serverPort = serverPort;
37:          clientSocket = new Socket(serverIP, serverPort);
38:
39:          br = new BufferedReader(
                    new InputStreamReader( clientSocket.getInputStream() ));
40:
41:          pw = new PrintWriter(new BufferedWriter(
                    new OutputStreamWriter(clientSocket.getOutputStream()) ));
42:
43:          keyboard = new BufferedReader(
                    new InputStreamReader(System.in, "KSC5601"));
44:
45:          (new readSocketThread()).start();
46:          (new writeSocketThread()).start();
47:
48:       } catch (Exception e) {
49:          System.out.println(e.toString());
50:       }
51:    }
52:
```

32라인의 생성자에서는 첫 번째 인자에 서버의 IP 주소가 전달되고, 두 번째 인자에 서버의 포트번호가 전달됩니다. 그리고 37라인에서는 서버에 접속을 시도합니다. 서버와의 접

속이 이루어지면 45라인과 46라인에서는 53라인과 65라인에 있는 스레드 클래스의 객체를 생성하고 실행시킵니다.

다음 스레드 클래스는 대화자가 키보드에 입력하는 문자열을 받아들입니다.

```
53:     class readSocketThread extends Thread {
54:        public void run() {
55:           try {
56:              while(true) {
57:                 System.out.println(br.readLine());
58:              }
59:           } catch (Exception e) {
60:              System.out.println(e.toString());
61:           }
62:        }
63:     }
64:
65:     class writeSocketThread extends Thread {
66:        String inputString = null;
67:        public void run() {
68:           try{
69:              while ((inputString = keyboard.readLine()) != null) {
70:                 pw.println(inputString);
71:                 pw.flush();
72:              }
73:           } catch (Exception e ) {
74:              System.out.println(e.toString());
75:           }
76:        }
77:     }
78: }
```

65라인의 스레드는 대화자가 키보드를 통해 입력하는 동안이라도 소켓에 메시지가 도착하면 이를 화면에 출력하는 역할을 합니다.

2.4. TCP 채팅 프로그램 II

다음 프로그램은 데이터를 이용한 채팅 예를 보인 것입니다. 화면 인터페이스를 CUI가 아닌 GUI로 작성했습니다.

exam/java/chapter09/tcp/ChatServer.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.io.*;
4:  import java.net.*;
5:  import java.util.*;
6:  public class ChatServer {
7:     Vector buffer;
8:     ServerSocket serverSocket;
9:     Socket socket;
10:    ObjectInputStream ois;
11:    ObjectOutputStream oos;
12:
13:    public void service() {
14:       try {
15:          System.out.println("접속 준비중");
16:          serverSocket = new ServerSocket(5555);
17:       } catch (IOException e) {
18:          System.err.println("서비스도중 IOException   발생!");
19:       }
20:
21:       while(true) {
22:          try {
23:             socket = serverSocket.accept();
24:             System.out.println(socket.getInetAddress()+"접속!");
25:             ois = new ObjectInputStream(socket.getInputStream());
26:             oos = new ObjectOutputStream(socket.getOutputStream());
27:             Thread t = new Thread(new ChatServerThread(buffer,ois,oos));
28:             t.start();
29:          } catch (IOException e) {
30:             System.err.println("IOException 발생!");
31:          }
32:       }
33:    }
34:    public static void main(String args[]) {
35:       System.out.println("Start Server Service...");
36:       ChatServer2 cs = new ChatServer2();
37:       cs.buffer = new Vector(5,1);
38:       cs.service();
39:    }
40: }
```

다음 프로그램은 클라이언트 하나에 하나씩 만들어질 스레드입니다. 클라이언트가 데이터

객체를 보내면 Vector에 저장된 ObjectOutputStream을 이용해서 알리게 됩니다. 클라이언트가 종료되면 스레드도 함께 종료됩니다.

exam/java/chapter09/tcp/ChatServerThread.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.io.*;
4:  import java.util.*;
5:
6:  public class ChatServerThread implements Runnable{
7:     Vector buffer;
8:     ObjectInputStream ois;
9:     ObjectOutputStream oos;
10:    Data d;
11:    boolean exit;
12:    String name;
13:
14:    public ChatServerThread(Vector v, ObjectInputStream ois ,
    ObjectOutputStream oos) {
15:       this.buffer = v;
16:       this.ois = ois;
17:       this.oos = oos;
18:       exit = false;
19:    }
20:
21:    public void run() {
22:       while(!exit) {
23:          try {
24:             d = (Data) ois.readObject();
25:          } catch (ClassNotFoundException e) {
26:             System.err.println("Data class를 찾을 수 없음!");
27:          } catch (OptionalDataException e1) {
28:             System.err.println("OptionalDataException 발생!");
29:          } catch (IOException e3) {
30:             System.err.println("IOExcdetion이 발생!");
31:          }
32:          int state = d.getState();
33:          if(state == Data.접속종료) {
34:             exit = true;
35:             d.setMessage("님이 종료하셨습니다.");
36:             name = d.getName();
37:             broadCasting();
38:             for(int i = 0 ; i <buffer.size() ; i++) {
39:                if( ((Data)buffer.elementAt(i)).getName().equals(name)) {
40:                   buffer.removeElementAt(i);
41:                   break;
42:                }
43:             }
44:             try{ ois.close();
45:                oos.close();
```

```
46:              }catch(IOException ex) {}
47:
48:           } else if(state == Data.처음접속) {
49:              Vector userName = new Vector(5,1);
50:              d.setOOS(oos);
51:              buffer.addElement(d);
52:              for(int i=0 ; i < buffer.size() ; i++) {
53:                 userName.addElement(((Data)buffer.elementAt(i)).getName());
54:              }
55:              d.setUserName(userName);
56:              System.out.println("broadCasting 시작");
57:              broadCasting();
58:           } else {
59:              broadCasting();
60:           }
61:        }
62:     }
63:
64:     public void broadCasting() {
65:        Vector v = (Vector)buffer.clone();
66:        for(int i = 0 ; i < v.size() ; i++ ) {
67:           try {
68:              ((Data)v.elementAt(i)).getOOS().writeObject(d);
69:              System.out.println("111");
70:           } catch (IOException e) {
71:              System.err.println("broadCasting method에서 예외 발생!");
72:              e.printStackTrace();
73:           }
74:        }
75:     }
76: }
```

다음 프로그램은 서버와 클라이언트 사이에 정보를 주고받기 위한 클래스의 예를 보인 것입니다.

exam/java/chapter09/tcp/Data.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.io.*;
4:  import java.util.*;
5:
6:  public class Data implements Serializable{
7:     private String message;
8:     private String name;
9:     private int state;
10:    private transient ObjectOutputStream oos;
11:    private Vector userName;
12:    public static final int 처음접속 = 0;
13:    public static final int 접속종료 = -1;
14:    public static final int 대화중 = 1;
15:
```

```
16:     public Data (String name, String message, int state, ObjectOutputStream o) {
17:        this.name = name;
18:        this.message = message;
19:        this.state = state;
20:        this.oos = o;
21:     }
22:     public Data (String name, String message, int state) {
23:        this(name , message, state, null);
24:     }
25:     public String getMessage() {
26:        return message;
27:     }
28:     public void setMessage(String s) {
29:        message = s;
30:     }
31:     public String getName() {
32:        return name;
33:     }
34:     public void setName(String s) {
35:     name = s;
36:     }
37:     public int getState() {
38:        return state;
39:     }
40:     public void setState(int i) {
41:        state = i;
42:     }
43:     public ObjectOutputStream getOOS() {
44:        return oos;
45:     }
46:     public void setOOS(ObjectOutputStream o) {
47:     oos = o;
48:     }
49:     public Vector getUserName() {
50:        return this.userName;
51:     }
52:     public void setUserName(Vector v) {
53:        this.userName = v;
54:     }
55: }
```

다음은 클라이언트 코드입니다.

exam/java/chapter09/tcp/ChatClient2.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.io.*;
4:  import java.net.*;
5:  import java.awt.*;
6:  import java.awt.event.*;
7:
```

```
8:  public class ChatClient2 {
9:     Socket socket;
10:    ObjectInputStream ois;
11:    ObjectOutputStream oos;
12:
13:    Thread t;
14:
15:    Frame first , second;
16:    Label state , commLabel , userLabel , serverLabel;
17:    Label IDLabel , userCount;
18:    TextArea commList;
19:    List userList;
20:    Button conn , transmission;
21:    TextField stateTextField , transTextField;
22:    TextField serverTextField , IDTextField;
23:
24:    String name;
25:    ChatClientThread cct;
26:
27:    public void connection(String serverName , int port) throws IOException{
28:       socket = new Socket(serverName,port);
29:       System.out.println(serverName + "  에 접속!");
30:
31:       oos = new ObjectOutputStream(socket.getOutputStream());
32:       ois = new ObjectInputStream(socket.getInputStream());
33:       System.out.println("OutputStream을 열었습니다. ");
34:       Data d = new Data(IDTextField.getText(),"님이 접.", Data.처음접속);
35:       System.out.println("Stream 연결에 성공하였습니다.");
36:       oos.writeObject(d) ;
37:
38:       System.out.println("InputStream을 열었습니다. ");
39:
40:       cct = new ChatClientThread(ois,this);
41:       t = new Thread(cct);
42:       t.start();
43:    }
44:
45:
46:    /**
47:     * 처음 보여질 GUI화면
48:     * Server의 이름과 User의 ID를 받아들여서,
49:     * Connection() 메서드를 호출합니다.
50:     */
51:    public void firstGo() {
52:       first = new Frame("Chat Browser");
53:       first.addWindowListener( new WindowAdapter() {
54:          public void windowClosing(WindowEvent e) {
55:             first.setVisible(false);
56:             first.dispose();
57:             System.exit(1);
```

```
58:           }
59:       });
60:
61:       Panel p = new Panel();
62:       p.setLayout(new GridLayout(2,2));
63:       serverLabel = new Label("Server",Label.CENTER);
64:       IDLabel = new Label(" I   D ",Label.CENTER);
65:       serverTextField = new TextField(0);
66:       IDTextField = new TextField(0);
67:       p.add(serverLabel);
68:       p.add(serverTextField);
69:       p.add(IDLabel);
70:       p.add(IDTextField);
71:
72:       conn = new Button("연결");
73:       conn.addActionListener( new ActionHandler());
74:       first.add(p,"Center");
75:       first.add(conn,"South");
76:       first.setSize(300,100);
77:
78:       /*
79:        * 화면의 중앙에 GUI가 보여질 수 있도록 합니다.
80:        * 현재의 Screen Size을 얻어온 후 중앙에 Dispaly 합니다.
81:        */
82:       Dimension d = first.getToolkit().getScreenSize();
83:       first.setLocation(d.width/2 - first.getWidth()/2 , d.height/2 -
    first.getHeight()/2);
84:       first.setResizable(false);
85:       first.setVisible(true);
86:    }
87:
88:    /**
89:     * 두번째로 보여질 GUI Chatting에 필요한 TextField와
90:     * 현재 접속되어 있는 사용자의 이름,
91:     * 그리고 대화 내용이 Display될 화면입니다.
92:     */
93:    public void secondGo() {
94:       second = new Frame("Chat v1.0 second");
95:       second.addWindowListener(new WindowAdapter() {
96:          public void windowClosing(WindowEvent e) {
97:             frameClose(e);
98:          }
99:       });
100:      state = new Label("접속중...");
101:      state.setBackground(Color.yellow);
102:      state.setForeground(Color.blue);
103:      second.add(state,"North");
104:
105:      Panel p1 = new Panel();
106:      p1.setLayout(new BorderLayout());
```

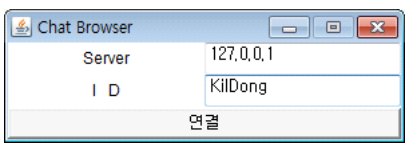
```
107:       commLabel = new Label("대화내용");
108:       commList = new TextArea();
109:       commList.setEditable(false);
110:       p1.add(commLabel,"North");
111:       p1.add(commList,"Center");
112:
113:       Panel p2 = new Panel();
114:       p2.setLayout(new BorderLayout());
115:       userLabel = new Label("사용자명",Label.CENTER);
116:       userList = new List();
117:       userCount = new Label("",Label.CENTER);
118:       p2.add(userLabel,"North");
119:       p2.add(userList,"Center");
120:       p2.add(userCount,"South");
121:
122:       Panel p3 = new Panel();
123:       transTextField = new TextField(50);
124:       //transTextField.requestFocus();  // TextField에 커서가 깜박이게 합니다.
125:       transTextField.addActionListener(new ActionHandler());
126:       transmission = new Button("전송");
127:       transmission.addActionListener(new ActionHandler());
128:       p3.add(transTextField);
129:       p3.add(transmission);
130:
131:       second.add(p3,"South");
132:       second.add(p1,"Center");
133:       second.add(p2,"West");
134:       second.setSize(600,500);
135:       Dimension d = second.getToolkit().getScreenSize();
136:       second.setLocation(d.width/2 - second.getWidth()/2 , d.height/2 -
    second.getHeight()/2);
137:       second.setResizable(false);
138:
139:       second.setVisible(true);
140:    }
141:
142:    public void frameClose(WindowEvent e) {
143:       Frame f = (Frame) e.getSource();
144:       f.setVisible(false);
145:       f.dispose();
146:       cct.exit = true;
147:       try {
148:          oos.writeObject(new Data(name,"님이 나가셨습니다.", Data.접속종료));
149:          oos.close();
150:       } catch (IOException e1) {
151:          System.err.println("종료중 IOExcpetion이 발생!");
152:       }
153:       System.exit(0);
154:    }
155:
```

```
156:    /**
157:     * Acrtion Event를 처리하는 Inner class입니다.
158:     */
159:    public class ActionHandler implements ActionListener{
160:       public void actionPerformed(ActionEvent e) {
161:          String actionCommand = e.getActionCommand();
162:          if(actionCommand.equals("연결")) {
163:             String server = serverTextField.getText();
164:             name = IDTextField.getText();
165:             first.setVisible(false);
166:             first.dispose();
167:             secondGo();
168:             try {
169:                connection(server,5555);
170:             } catch (IOException e1) {
171:                System.err.println("Connection 중 Exception이 발생하였습니다.");
172:             }
173:             state.setText("[ " + server +" ]" + " 에 접속됨. UserID : " + name);
174:          } else if ( actionCommand.equals("전송") ||
    e.getSource().equals(transTextField)) {
175:             try {
176:                oos.writeObject(new Data(name,transTextField.getText(),
    Data.대화중));
177:             } catch (IOException e2) {
178:                System.err.println("대화중 IOException이 발생하였습니다 ");
179:             }
180:             transTextField.setText("");
181:          }
182:       }
183:    }
184:
185:    public static void main(String args[]) {
186:       ChatClient2 cc = new ChatClient2();
187:          cc.firstGo();
188:    }
189:}//end class
```

exam/java/chapter09/tcp/ChatClientThread.java

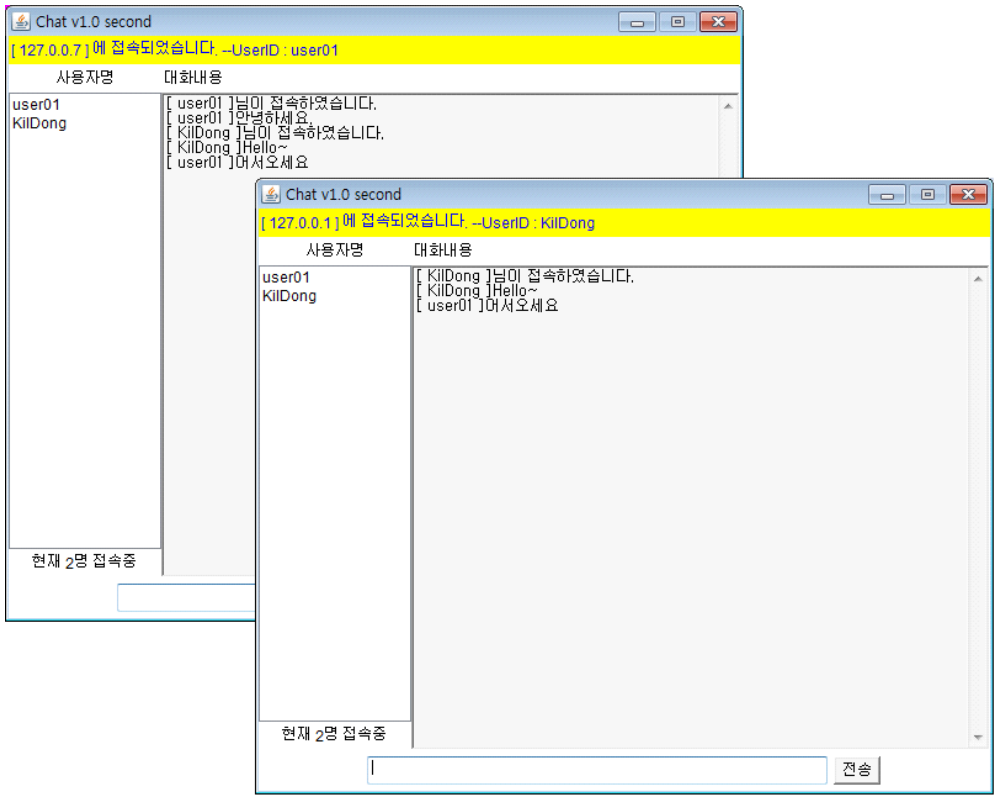
```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.io.*;
4:  import java.util.*;
5:
6:  public class ChatClientThread implements Runnable{
7:     ObjectInputStream ois;
8:     ChatClient2 cc;
9:     Data d;
10:    boolean exit = false;
11:
```

```
12:    public ChatClientThread(ObjectInputStream ois, ChatClient2 cc) {
13:       this.ois = ois;
14:       this.cc = cc;
15:    }
16:
17:    public void run() {
18:       while(!exit) {
19:          try {
20:             d = (Data) ois.readObject();
21:          } catch (IOException e) {
22:             System.err.println("run method IOException");
23:          } catch (ClassNotFoundException e1) {
24:             System.err.println("Data class NotFound");
25:          }
26:          int state = d.getState();
27:          String name = d.getName();
28:          if(state == Data.처음접속) {
29:             Vector userName = d.getUserName();
30:             cc.userList.removeAll();
31:             for(int i = 0 ; i < userName.size() ; i++) {
32:                cc.userList.add((String)userName.elementAt(i));
33:             }
34:                cc.userCount.setText("현재 " + cc.userList.getItemCount() +
35: "명 접속중");
36:          } else if (state == Data.접속종료) {
37:             cc.userList.remove(name);
38:             cc.userCount.setText("현재 " + cc.userList.getItemCount() +
39: "명 접속중");
40:          }
41:          cc.commList.append("[ " + name + " ]" + d.getMessage() + "\n");
42:       }
43:       try {
44:          ois.close();
45:       } catch (IOException e) {
46:          System.err.println(" ChatClientThread에의 ObjectOutputStream을
    Close하는 중에 IOException 발생!");
47:       }
48:    }//end run
49: }
```



채팅서버에 접속하기 위한 클라이언트 시작화면입니다. 서버의 아이피와 대화자의 아이디를 입력합니다.

다음 그림은 클라이언트 대화 화면입니다.



2.5. TCP 파일서버 프로그램

다음 프로그램은 간단한 파일서버 프로그램의 예를 보인 것입니다. 서버로부터 특정파일을 읽어 클라이언트 화면에 서버의 파일내용을 출력합니다.

exam/java/chapter09/tcp/FileServer.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.awt.*;
4:  import java.net.*;
5:  import java.io.*;
6:
7:  public class FileServer {
8:
9:     public static void main(String[] args) {
10:       ServerSocket s = null;
11:       Socket s1;
12:       byte[] intbuf = new byte[100];
13:       String fileName;
14:
15:       try {
16:          s = new ServerSocket(4321, 1);
17:       } catch (IOException e) {
18:          System.out.println("\nServer timed out!");
19:          System.exit(-1);
20:       }
21:
22:       while(true) {
23:          try {
24:             s1 = s.accept();
25:             fileName = getFileName(s1);
26:             sendFileToClient(s1, fileName);
27:             s1.close();
28:          } catch(IOException e) {
29:             System.out.println("Error - " + e.toString());
30:          }
31:       }
32:    }
33:
34:    public static String getFileName(Socket s1) throws IOException {
35:       InputStream s1in;
36:       DataInputStream d1In;
37:       String sfile;
38:       s1in = s1.getInputStream();
39:       d1In = new DataInputStream(s1in);
40:       sfile = d1In.readLine();
41:       System.out.println("File to open for reading : " + sfile);
```

```
42:         return(sfile);
43:     }
44:
45:     public static void sendFileToClient (Socket s1, String sfile) throws
    IOException {
46:         int c;
47:         FileInputStream fis;
48:         OutputStream s1out;
49:         s1out = s1.getOutputStream();
50:         File f = new File(sfile);
51:
52:         if(f.exists() != true) {
53:             String error = new String ("File " + sfile + "은(는) 존재하지 않습니다.\n");
54:             int len = error.length();
55:             for(int i=0; i<len; i++) {
56:                 s1out.write((int)error.charAt(i));
57:             }
58:             System.out.println(error);
59:             return;
60:         }
61:
62:         if(f.canRead()) {
63:             fis = new FileInputStream(sfile);
64:             System.out.println("Sending : " + sfile);
65:             while((c=fis.read()) != -1) {
66:                 s1out.write(c);
67:             }
68:             fis.close();
69:         } else {
70:             String error = new String ("Can't open " + sfile + "for reading...\n");
71:             int len = error.length();
72:             for(int i=0; i<len ; i++) {
73:                 s1out.write((int)error.charAt(i));
74:             }
75:             System.out.println(error);
76:         }
77:     }
78: }
```

다음은 클라이언트 측 코드입니다.

exam/java/chapter09/tcp/FileClient.java

```
1:  package exam.java.chapter09.tcp;
2:
3:  import java.net.*;
4:  import java.io.*;
5:
6:  import javax.swing.JOptionPane;
7:
```

```
8:  public class FileClient {
9:
10:    public static void main(String[] args) {
11:       Socket s;
12:       String server = JOptionPane.showInputDialog("서버의 주소를 입력하세요.");
13:       String fileName = JOptionPane.showInputDialog("파일명을 입력하세요.");
14:       int port = 4321;
15:
16:       try{
17:          s = new Socket (server, port);
18:          sendFileName (s,fileName);
19:          receiveFile (s);
20:          s.close();
21:       } catch (IOException e) {
22:          System.out.println("Connection failed");
23:       }
24:    }
25:
26:    public static void sendFileName (Socket s, String fileName) throws
    IOException {
27:       OutputStream  sOut;
28:       DataOutputStream dOut;
29:
30:       sOut = s.getOutputStream();
31:       dOut = new DataOutputStream(sOut);
32:       String sendString = new String(fileName + "\n");
33:       dOut.writeBytes(sendString);
34:    }
35:
36:    public static void receiveFile(Socket s) throws IOException {
37:       int c;
38:       InputStream sIn;
39:       sIn = s.getInputStream();
40:
41:       while ((c=sIn.read()) != -1) {
42:          System.out.print((char)c);
43:       }
44:    }
45: }
```

2.6. 요점 정리

1. TCP 네트워크 프로그래밍

소켓과 소켓 사이의 입출력

서버

```
serverSocket = new ServerSocket(5432);
socket = serverSocket.accept();
```

클라이언트

```
socket = new Socket("127.0.0.1", 5432);
```