<JSTORM>

자바 I/O성능 향상 (Tuning Java I/O Performance)

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

Document Information

Document title:	자바 I/O성능 향상
Document file name:	ioTunning_박준용_final.doc
Revision number:	<1.0>
Issued by:	<JSTORM>
Issue Date:	<2001/08/08 >
Status:	final

Content Information

Audience	자바 초, 중급
Abstract	이 문서는 자바 I/O 성능 향상을 위한 다양한 기술들을 설명한다
Reference	(http://developer.java.sun.com/developer/technicalArticles/Programming/PerfTuning/index.html) 를 번역
Benchmark information	

Table of Contents

자바 I/O 성능 향상

(Tuning Java™ I/O Performance)

본 문서는 자바 I/O 성능 향상을 위한 다양한 기술들을 설명한다. 언급하고 있는 기술 중 대부분은 디스크 파일 I/O에 관한 것이고 몇몇은 네트웍 I/O와 화면 출력에 적용할 수 있다. 첫번째로 low-level I/O 이슈들을 다루고 그 후에 압축(compression), 형식화(formatting), 직렬화(serialization)와 같은 high-level 이슈들을 다룬다. 그러나 여기서 설명하는 것들은 탐색 알고리즘의 선택이나 자료 구조 같은 어플리케이션 설계 이슈들을 다루지는 않고 파일 캐싱 같은 system-level 이슈들도 설명하지 않는다.

자바 I/O에 대해 이야기할 때 자바 프로그래밍 언어가 가정하는 두가지 다른 형태의 디스크 파일 구성은 의미가 없다. 하나는 바이트 스트림 기반이고 다른 하나는 캐릭터 순서(sequences) 기반이다. 자바 언어에서 캐릭터는 C와 같은 다른 일반 언어처럼 1 바이트가 아닌 2 바이트를 사용하여 나타낸다. 이와 같은 이유 때문에 파일에서 캐릭터를 읽을 때 변환 작업이 필요하다. 이제 설명할 예제에서처럼 이러한 다른 언어와의 차이점은 종종 중요한 문제가 되기도 한다.

Low-Level I/O Issues

- I/O 속도 향상을 위한 기본 법칙
- 버퍼링
- 텍스트 파일 읽기/쓰기
- 형식화 비용(Formatting Cost)
- 랜덤 접근(Random Access)

High-Level I/O Issues

- 압축
- 캐싱
- Tokenizatio n
- 직렬화
- 파일에 대한 정보 포함하기
- 추가 정보

I/O 속도 향상을 위한 기본 법칙

설명을 시작하면서 여기 I/O 속도 향상을 위한 몇가지 기본 법칙이 있다.

1. 디스크 접근을 피하라.
2. OS 접근을 피하라.
3. 메소드 호출을 피하라.
4. 바이트와 캐릭터를 각각 처리하는 것을 피하라.

어떠한 I/O도 일어나지 않을수 있기 때문에 이러한 규칙들은 포괄적인 방법으로 적용할 수 없다. 그러나 파일에서 뉴라인 바이트('\n')의 숫자를 세는 다음 3개의 예제를 고려한다면 이 규칙들을 적용하는 방법을 알 수 있을 것이다.

접근방법 1 : 읽기 메소드

첫번째 접근방법은 FileInputStream으로 간단히 read() 메소드를 사용한다.

```
import java.io.*;

 public class intro1 {
   public static void main(String args[]) {
     if (args.length != 1) {
       System.err.println("missing
filename");
       System.exit(1);
     }
     try {
       FileInputStream fis =
           new FileInputStream(args[0]);
       int cnt = 0;
       int b;
       while ((b = fis.read()) != -1) {
         if (b == '\n')
           cnt++;
       }
       fis.close();
       System.out.println(cnt);
     }
     catch (IOException e) {
       System.err.println(e);
     }
   }
 }
```

그러나 이런 접근 방법은 FileInputStream.read가 파일의 다음 바이트를 리턴하는 네이티브 메소드라서 런타임 시스템에 많은 호출을 발생시킨다.

접근방법 2 : 대용량 버퍼 사용하기

두번째 접근 방법은 대용량 버퍼를 사용함으로써 위의 문제를 회피한다.

```
import java.io.*;

 public class intro2 {
   public static void main(String args[]) {
    if (args.length != 1) {
      System.err.println("missing filename");
      System.exit(1);
    }
    try {
      FileInputStream fis =
          new FileInputStream(args[0]);
      BufferedInputStream bis =
          new BufferedInputStream(fis);
      int cnt = 0;
      int b;
      while ((b = bis.read()) != -1) {
        if (b == '\n')
          cnt++;
        }
      bis.close();
      System.out.println(cnt);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
 }
```

BufferedInputStream.read는 입력 버퍼에서 다음 바이트를 얻고 시스템에는 거의 접근하지 않는다.

접근방법 3 : 직접적인 버퍼링(Direct Buffering)

세번째 접근 방법은 read 메소드 호출을 없애기 때문에 BufferedInputStream을 피하고 직접 버퍼링하는 것이다.

```
import java.io.*;

   public class intro3 {
```

```
    public static void main(String args[]) {
      if (args.length != 1) {
        System.err.println("missing
filename");
        System.exit(1);
      }
      try {
        FileInputStream fis =
            new FileInputStream(args[0]);
        byte buf[] = new byte[2048];
        int cnt = 0;
        int n;
        while ((n = fis.read(buf)) != -1) {
          for (int i = 0; i < n; i++) {
            if (buf[i] == '\n')
              cnt++;
          }
        }
        fis.close();
        System.out.println(cnt);
      }
      catch (IOException e) {
        System.err.println(e);
      }
    }
}
```

1MB 입력 파일의 경우 프로그램의 초단위 실행시간은 다음과 같다.

intro1 6.9(초)

intro2 0.9(초)

intro3 0.4(초)

가장 느린경우와 가장 빠른 경우 약 17배의 차이가 난다.

이렇게 큰 속도향상은 언제나 자신만의 버퍼링 방법인 3번째 접근방법으로 사용하기를 증명하지는 않는다. 이러한 접근방법은 신중히 구현하지 않는다면 파일의 끝에 도달했을 때 이벤트 처리 같은 곳에서 에러가 발생하기 쉽다. 또한 다른 방법에 비해 가독성이

떨어진다. 그러나 머리속에 기억하고 있다가 필요할 때 고쳐 쓸줄 안다면 유용할것이다.

아마 대부분의 어플리케이션에서는 접근방법 2 가 적당할 것이다.

버퍼링

접근 방법 2 와 3은 디스크에서 파일의 큰 덩어리를 읽어들이고 이후 한번에 한 바이트나 한 캐릭터에 접근하는 버퍼링 기술을 사용한다. 버퍼링은 I/O 속도 향상에 기본적이며 중요한 기술이고 많은 자바 클래스들이 버퍼링을 지원한다. (바이트를 위한 BufferedInputStream 과 캐릭터를 위한 BufferedReader)

분명한 질문 하나는 버퍼가 클수록 I/O가 빨라지느냐 하는 것이다. 일반적으로 자바 버퍼는 기본 크기가 1024 혹은 2048 바이트 이다. 더 큰 버퍼가 I/O 속도 향상에 도움이 될수 있지만 겨우 5 내지 10% 정도이다.

접근방법 4 : 전체 파일

버퍼링의 극단적인 경우는 먼저 파일의 길이를 알아내어 전체 파일을 읽어들이는 것이다.

```
import java.io.*;

public class readfile {
  public static void main(String args[]) {
    if (args.length != 1) {
      System.err.println("missing filename");
      System.exit(1);
    }
    try {
      int len = (int)(new File(args[0]).length());
      FileInputStream fis =
          new FileInputStream(args[0]);
      byte buf[] = new byte[len];
```

```
      fis.read(buf);
      fis.close();
      int cnt = 0;
      for (int i = 0; i < len; i++) {
        if (buf[i] == '\n')
          cnt++;
      }
      System.out.println(cnt);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

이러한 접근방법은 파일이 바이트들의 배열처럼 다루어지므로 편리하다. 그러나 매우 큰 파일인 경우 읽어들일만한 충분한 메모리가 없을 수 있다는 명백한 문제가 있다.

버퍼링의 다른 관점에서 터미널 윈도우에 텍스트 출력을 생각해보자. 기본적으로 System.out(a PrintStream)은 라인 버퍼이다. 라인 버퍼는 뉴라인 캐릭터를 만날때마다 출력 버퍼를 비운다는 뜻이다. 이것은 실제 입력이 일어나기 전 입력 프롬프트가 화면에 나타나야 하는 상호대화방식(interactivity)에서 중요한 문제이다.

접근방법 5 : 라인 버퍼링 사용하지 않기

이번 예제에서 처럼 라인 버퍼링을 사용하지 않을수 있다.

```
import java.io.*;

public class bufout {
  public static void main(String args[]) {
    FileOutputStream fdout =
        new FileOutputStream(FileDescriptor.out);
    BufferedOutputStream bos =
        new BufferedOutputStream(fdout, 1024);
    PrintStream ps =
        new PrintStream(bos, false);

    System.setOut(ps);
```

```
    final int N = 100000;

    for (int i = 1; i <= N; i++)
      System.out.println(i);

    ps.close();
  }
}
```

이 프로그램은 출력으로 1..100000 의 정수를 쓰고 기본적인 라인 버퍼링을 사용한것보다 약 3배 정도 빠르다.

버퍼링은 아래에 보여지는 예제중 하나로 중요한 부분이고 버퍼는 랜덤 파일 접근(random file access)의 속도 향상에 사용된다.

텍스트 파일 읽기/쓰기

이전에 파일에서 캐릭터를 읽을 때 메소드 호출 오버헤드는 명백해진다고 언급했었다. 이것의 다른 예제로 텍스트 파일에서 라인의 수를 세는 프로그램을 들 수 있다.

```
import java.io.*;

 public class line1 {
   public static void main(String args[]) {
     if (args.length != 1) {
       System.err.println("missing filename");
       System.exit(1);
     }
     try {
       FileInputStream fis =
           new FileInputStream(args[0]);
       BufferedInputStream bis =
           new BufferedInputStream(fis);
       DataInputStream dis =
           new DataInputStream(bis);
       int cnt = 0;
       while (dis.readLine() != null)
         cnt++;
```

```
        dis.close();
        System.out.println(cnt);
      }
      catch (IOException e) {
        System.err.println(e);
      }
    }
  }
```

이 프로그램은 각 캐릭터를 얻기 위해 read 메소드 호출을 사용해서 구현된 DataInputStream.readLine을 사용한다. 새로운 접근방법은 다음과 같다.

```
import java.io.*;

 public class line2 {
   public static void main(String args[]) {
     if (args.length != 1) {
       System.err.println("missing filename");
       System.exit(1);
     }
     try {
       FileReader fr = new FileReader(args[0]);
       BufferedReader br = new BufferedReader(fr);
       int cnt = 0;
       while (br.readLine() != null)
         cnt++;
       br.close();
       System.out.println(cnt);
     }
     catch (IOException e) {
       System.err.println(e);
     }
   }
 }
```

이 접근방법이 더 빠르다. 예를들어 , 200,000 라인의 6MB 파일의 경우 두번째 프로그램은 첫번째에 비해 20%정도 빠르다.

그러나 두번째 프로그램이 더 빠르지 않다면 기억할만한 중요한 문제가 있다. DataInputStream.readLine이 쓰이지 않게 되었기 때문에 첫번째 프로그램에서 자바2 컴파일러는 deprecation 경고를 발생시킨다. 아마도 그것은 바이트를 캐릭터로 변경시키지 않을 것이고

ASCII 텍스트 바이트 스트림외에 다른 것들을 포함한 텍스트 파일을 다루는데는 적절한 선택이 아닐것이다.

이것이 이전에 언급했었던 바이트 스트림과 캐릭터 스트림 사이의 차이점이다. 다음과 같은 프로그램을 보자.

```
import java.io.*;

public class conv1 {
  public static void main(String args[]) {
    try {
      FileOutputStream fos =
         new FileOutputStream("out1");
      PrintStream ps =
         new PrintStream(fos);
      ps.println("\uffff\u4321\u1234");
      ps.close();
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }
}
```

이 프로그램은 결과 파일을 쓰지만 유니코드 캐릭터들을 저장하지 않은 실제적인 결과이다. 읽기/쓰기 I/O 클래스들은 캐릭터 기반이고 이런 이슈들을 해결하기 위해 설계되었다. OutputStreamWriter는 캐릭터를 바이트로의 인코딩이 적용되었다.

유니코드 캐릭터를 출력하기 위해 PrintWriter를 사용하는 프로그램은 다음과 비슷해보인다.

```
import java.io.*;

public class conv2 {
  public static void main(String args[]) {
    try {
      FileOutputStream fos =
          new FileOutputStream("out2");
```

```
        OutputStreamWriter osw =
            new OutputStreamWriter(fos, "UTF8");
        PrintWriter pw =
            new PrintWriter(osw);
        pw.println("\uffff\u4321\u1234");
        pw.close();
      }
      catch (IOException e) {
        System.err.println(e);
      }
    }
  }
```

이 프로그램은 아스키 텍스트 인코딩 프로퍼티를 가지고 다른 캐릭터들은 2 혹은 3바이트로 인코딩되는 UTF-8 인코딩을 사용한다.

형식화 비용(Formatting Costs)

실제로 파일에 데이터를 쓰는 것은 출력 비용의 일부이다. 다른 명백한 비용은 데이터 형식화(formatting)이다. 다음 처럼 출력하는 3가지 예제를 생각해보자.

The square of 5 is 25

접근방법 1

첫번째 접근방법은 고유한 I/O 비용의 생각을 갖고 조합된 스트링으로 간단히 출력하는 것이다.

```
public class format1 {
  public static void main(String args[]) {
    final int COUNT = 25000;

    for (int i = 1; i <= COUNT; i++) {
      String s = "The square of 5 is 25\n";
      System.out.print(s);
    }
  }
}
```

접근방법 2

두번째 접근방법은 "+"를 사용한 간단한 형식화이다.

```
public class format2 {
   public static void main(String args[]) {
     int n = 5;

     final int COUNT = 25000;


     for (int i = 1; i <= COUNT; i++) {
       String s = "The square of " + n + " is " +
           n * n + "\n";
       System.out.print(s);
     }
   }
 }
```

접근방법 3

세번째 접근방법은 java.text 패키지의 MessageFormat 클래스를 사용한다.

```
import java.text.*;

public class format3 {
  public static void main(String args[]) {
    MessageFormat fmt =
     new MessageFormat("The square of {0} is
{1}\n");
     Object values[] = new Object[2];


   int n = 5;


   values[0] = new Integer(n);
   values[1] = new Integer(n * n);
```

```
    final int COUNT = 25000;


    for (int i = 1; i <= COUNT; i++) {
      String s = fmt.format(values);
      System.out.print(s);
     }
    }
  }
```

이 프로그램들은 동일한 결과를 출력한다. 실행시간을 보면

format1 1.3

format2 1.8

format3 7.8

가장 빠른 프로그램과 가장 느린 프로그램은 약 6배의 차이가 난다. 세번째 프로그램은 형식이 미리 컴파일되지 않고 대신 편리한 정적 메소드를 사용한다면 훨씬 더 느려질 것이다.

접근방법 4

MessageFormat.format(String , Object[])

```
import java.text.*;

public class format4 {
  public static void main(String args[]) {
    String fmt = "The square of {0} is {1}\n";
    Object values[] = new Object[2];


    int n = 5;


    values[0] = new Integer(n);
```

```
        values[1] = new Integer(n * n);

        final int COUNT = 25000;


        for (int i = 1; i <= COUNT; i++) {
          String s =
              MessageFormat.format(fmt, values);
          System.out.print(s);
        }
      }
    }
```

이전의 예제보다 지금 예제는 시간이 1/3 정도 더 오래 걸린다.

접근방법 3은 접근방법 1 과 2 보다 훨씬 더 느리지만 접근방법 3을 사용하지 말라는 뜻은 아니다. 그러나 시간 비용을 생각해야 한다.

메시지 형식화는 국제화 컨텍스트에서 중요하고 어플리케이션은 리소스 번들에서 형식을 읽으므로 이 문제에 관해 생각하고 사용해야 한다.

랜덤 접근(Random Access)

RandonAccessFile은 파일에서 랜덤 I/O 접근(바이트 레벨에서)을 수행하는 자바 클래스이다. 이 클래스는 C/C++에서 찾을 수 있는것과 비슷한 현재 읽거나 쓰는 지점에서 다른 위치로 파일 포인터를 옮기는 seek 메소드를 제공한다.

seek 메소드는 런타임 시스템에 접근하므로 값비싼 경향이 있다. 하나의 대안으로는 RandomAccessFile의 상위에서 버퍼링을 하는 방법이 있고 직접 바이트를 읽는 read 메소드를 구현하는 방법이 있다. read를 수행하기 위한 파라미터는 원하는 바이트 크기를 나타내는 0 보다 크거나 같은 바이트 오프셋이다. 이것이 어떻게 사용되는가는 다음 예제를 보자.

```
import java.io.*;

  public class ReadRandom {
    private static final int DEFAULT_BUFSIZE =
4096;

    private RandomAccessFile raf;
    private byte inbuf[];
    private long startpos = -1;
    private long endpos = -1;
    private int bufsize;

    public ReadRandom(String name)
     throws FileNotFoundException {
      this(name, DEFAULT_BUFSIZE);
    }

    public ReadRandom(String name, int b)
        throws FileNotFoundException {
      raf = new RandomAccessFile(name, "r");
      bufsize = b;
      inbuf = new byte[bufsize];
    }

    public int read(long pos) {
      if (pos < startpos || pos > endpos) {
        long blockstart = (pos / bufsize) *
bufsize;
        int n;
        try {
          raf.seek(blockstart);
          n = raf.read(inbuf);
        }
        catch (IOException e) {
          return -1;
        }
        startpos = blockstart;
        endpos = blockstart + n - 1;
        if (pos < startpos || pos > endpos)
          return -1;
      }

      return inbuf[(int)(pos - startpos)] & 0xffff;
    }
```

```
    public void close() throws IOException {
      raf.close();
    }

    public static void main(String args[]) {
      if (args.length != 1) {
        System.err.println("missing filename");
        System.exit(1);
      }

      try {
        ReadRandom rr = new ReadRandom(args[0]);
        long pos = 0;
        int c;
        byte buf[] = new byte[1];
        while ((c = rr.read(pos)) != -1) {
          pos++;
          buf[0] = (byte)c;
          System.out.write(buf, 0, 1);
        }
        rr.close();
      }
      catch (IOException e) {
        System.err.println(e);
      }
    }
  }
```

이 프로그램은 순차적으로 바이트들을 읽고 읽은 바이트들을 출력한다.

이 기술은 파일에서 동시에 근처에 있는 바이트들을 읽는 접근 위치를 가지고 있을 때 유용하다. 예를들어 , 정열된 파일에서 바이너리 서치 스키마를 구현한다면 이러한 접근방법은 유용하다. 큰 파일의 임의의 지점에서 랜덤하게 접근하는 것이 비용이 적게 든다.

압축(Compression)

자바는 바이트 스트림들을 압축하고 압축을 푸는 클래스들을 제공한다. 이들은 java.util.zip 패키지에서 찾을 수 있고 jar 파일을 위해 기본적으로 제공된다. (Jar 파일은 추가 목록을 가지고 있는 Zip 파일이다.)

다음 프로그램은 하나의 입력 파일을 얻고 입력 파일을 표현하는 하나의 엔트리를 가진 압축된 Zip파일에 출력을 쓴다.

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

  public class compress {
    public static void doit(
                            String filein,
                            String fileout
                            ) {
      FileInputStream fis = null;
      FileOutputStream fos = null;
      try {
        fis = new FileInputStream(filein);
        fos = new FileOutputStream(fileout);
        ZipOutputStream zos =
            new ZipOutputStream(fos);
        ZipEntry ze = new ZipEntry(filein);
        zos.putNextEntry(ze);
        final int BUFSIZ = 4096;
        byte inbuf[] = new byte[BUFSIZ];
        int n;
        while ((n = fis.read(inbuf)) != -1)
          zos.write(inbuf, 0, n);
        fis.close();
        fis = null;
        zos.close();
        fos = null;
      }
      catch (IOException e) {
        System.err.println(e);
      }
      finally {
        try {
          if (fis != null)
            fis.close();
          if (fos != null)
            fos.close();
        }
        catch (IOException e) {
        }
      }
```

```
    }
  public static void main(String args[]) {
    if (args.length != 2) {
     System.err.println("missing filenames");
     System.exit(1);
    }
   if (args[0].equals(args[1])) {
     System.err.println("filenames are identical");
     System.exit(1);
      }
      doit(args[0], args[1]);
    }
  }
```

다음 프로그램은 이전 처리과정과 반대로 입력으로 하나의 Zip 파일(Zip 파일 안에 하나의 엔트리만 있다고 가정한)을 받아서 엔트리를 출력 파일로 압축을 푼다.

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

  public class uncompress {
    public static void doit(
                             String filein,
                             String fileout
                             ) {
      FileInputStream fis = null;
      FileOutputStream fos = null;
      try {
        fis = new FileInputStream(filein);
        fos = new FileOutputStream(fileout);
        ZipInputStream zis = new
ZipInputStream(fis);
        ZipEntry ze = zis.getNextEntry();
        final int BUFSIZ = 4096;
        byte inbuf[] = new byte[BUFSIZ];
        int n;
        while ((n = zis.read(inbuf, 0, BUFSIZ)) !=
-1)
          fos.write(inbuf, 0, n);
        zis.close();
        fis = null;
        fos.close();
```

```
        fos = null;
      }
      catch (IOException e) {
        System.err.println(e);
      }
      finally {
        try {
          if (fis != null)
            fis.close();
          if (fos != null)
            fos.close();
        }
        catch (IOException e) {
        }
      }
    }
    public static void main(String args[]) {
      if (args.length != 2) {
     System.err.println("missing filenames");
     System.exit(1);
      }
    if (args[0].equals(args[1])) {
     System.err.println("filenames are identical");
     System.exit(1);
      }
      doit(args[0], args[1]);
    }
  }
```

압축이 I/O 성능에 도움을 주는가 아닌가는 로컬 하드웨어
설정(프로세서와 디스크 드라이브의 상대적인 속도)에 달려있다.
Zip기술을 사용하는 압축은 보통 데이터 크기를 50% 정도 줄이지만
압축하고 압축을 푸는데 약간의 시간 비용이 든다. 300Mhz의 CPU와 IDE
디스크 드라이브를 사용해서 5~10MB의 압축된 텍스트 파일로
실험해보면 압축되지 않은 파일을 읽을 때 보다 압축된 파일을 읽을 때
약 1/3 정도 속도증가를 보인다.

플로피 디스크 같은 매우 느린 미디어에 데이터를 쓸 경우 압축을
사용하면 훨씬 유용하다. 300Mhz 펜티엄 프로세서와 느린
플로피(일반적인 PC에 달려있는) 를 사용해서 테스트 해본 결과 압축된
텍스트 파일을 플로피 파일에 쓰는 것이 직접 플로피 디스켓에
복사하는 것보다 약 50% 정도 속도 향상이 있었다.

캐싱(Caching)

하드웨어 캐싱에 대한 세부적인 사항은 여기에서 다루지 않는다. 그러나 때때로 소프트웨어 캐싱은 I/O 속도 향상에 쓰일 수 있다. 텍스트파일에서 무작위 순서로 라인들을 읽기를 원할 경우를 생각해보자. 이런 일을 하는 한가지 방법은 모든 라인들을 읽어서 ArrayList(Vector와 비슷한 콜렉션 클래스)에 저장하는 것이다.

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

 public class LineCache {
   private ArrayList list = new ArrayList();

   public LineCache(String fn) throws IOException
{
     FileReader fr = new FileReader(fn);
     BufferedReader br = new BufferedReader(fr);
     String ln;
     while ((ln = br.readLine()) != null)
       list.add(ln);
     br.close();
   }

   public String getLine(int n) {
     if (n < 0)
       throw new IllegalArgumentException();

     return (n < list.size() ?
      (String)list.get(n) : null);
   }

   public static void main(String args[]) {
     if (args.length != 1) {
       System.err.println("missing filename");
       System.exit(1);
     }
     try {
       LineCache lc = new LineCache(args[0]);
       int i = 0;
       String ln;
       while ((ln = lc.getLine(i++)) != null)
```

```
            System.out.println(ln);
          }
          catch (IOException e) {
            System.err.println(e);
          }
        }
    }
```

getLine 메소드는 특정위치에서 원하는 라인을 찾을 때 사용된다. 이 기술은 매우 유용하지만 큰 파일인 경우 많은 메모리를 필요로 하므로 제한사항이 있다. 다른 대안은 요청됐었던 마지막 100 라인 정도를 기억하고 다른 요청이 올 경우 디스크에서 읽어들이는 것이다. 이러한 계획은 라인 접근의 지역성(locality)이 있다면 잘 동작하지만 라인 요청이 무작위라면 좋은 방법이 아니다.

Tokenization

Tokenization 은 논리적인 덩어리(예를 들면 단어 같은)로 바이트나 캐릭터 시퀀스를 자르는 과정이다. 자바는 이러한 기능을 수행하기 위해StreamTokenizer 클래스를 제공한다.

```
import java.io.*;

 public class token1 {
   public static void main(String args[]) {
    if (args.length != 1) {
      System.err.println("missing filename");
      System.exit(1);
    }
    try {
      FileReader fr = new FileReader(args[0]);
      BufferedReader br = new BufferedReader(fr);
      StreamTokenizer st = new
StreamTokenizer(br);
      st.resetSyntax();
      st.wordChars('a', 'z');
      int tok;
      while ((tok = st.nextToken()) !=
          StreamTokenizer.TT_EOF) {
        if (tok == StreamTokenizer.TT_WORD)
```

```
            ;// st.sval has token
          }
          br.close();
        }
        catch (IOException e) {
          System.err.println(e);
        }
      }
  }
```

이 예제는 소문자 단어(a ? z)를 기준으로 토큰화 한다. 이와 동등하게 구현하려면 다음과 같을 것이다.

```
import java.io.*;

 public class token2 {
   public static void main(String args[]) {
     if (args.length != 1) {
       System.err.println("missing filename");
       System.exit(1);
     }
     try {
       FileReader fr = new FileReader(args[0]);
       BufferedReader br = new BufferedReader(fr);
       int maxlen = 256;
       int currlen = 0;
       char wordbuf[] = new char[maxlen];
       int c;
       do {
         c = br.read();
         if (c >= 'a' && c <= 'z') {
           if (currlen == maxlen) {
             maxlen *= 1.5;
             char xbuf[] =
                 new char[maxlen];
             System.arraycopy(
                 wordbuf, 0,
                 xbuf, 0, currlen);
             wordbuf = xbuf;
           }
           wordbuf[currlen++] = (char)c;
         }
         else if (currlen > 0) {
           String s = new String(wordbuf,
```

```
                0, currlen);
            // do something with s
              currlen = 0;
            }
          } while (c != -1);
          br.close();
        }
        catch (IOException e) {
          System.err.println(e);
        }
      }
    }
```

두번째 프로그램은 약간의 저레벨 코드를 사용함으로써 첫번째 것보다 20% 정도가 빠르다.

말하자면 StreamTokenizer는 캐릭터 기반 스트림(BufferedReader 같은)에서 읽기를 수행하는 혼합 클래스이지만 동시에 바이트의 관점에서 알파벳 캐릭터인 경우 모든 캐릭터들이 2 바이트 값(0xff 보다 큰)으로 이루어졌다고 생각한다.

직렬화(Serialization)

직렬화는 자바 자료 구조를 표준 형식을 사용해서 바이트 스트림으로 변경시키는데 사용된다. 예를 들어 다음 프로그램은 랜덤 정수형의 배열을 출력한다.

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

public class serial1 {
  public static void main(String args[]) {
    ArrayList al = new ArrayList();
    Random rn = new Random();
    final int N = 100000;

    for (int i = 1; i <= N; i++)
      al.add(new Integer(rn.nextInt()));

    try {
      FileOutputStream fos =
```

```
            new FileOutputStream("test.ser");
          BufferedOutputStream bos =
            new BufferedOutputStream(fos);
          ObjectOutputStream oos =
            new ObjectOutputStream(bos);
          oos.writeObject(al);
          oos.close();
        }
        catch (Throwable e) {
          System.err.println(e);
        }
      }
    }
```

그리고 다음 프로그램은 반대로 배열에서 읽기를 수행한다.

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

 public class serial2 {
   public static void main(String args[]) {
     ArrayList al = null;

     try {
       FileInputStream fis =
           new FileInputStream("test.ser");
       BufferedInputStream bis =
           new BufferedInputStream(fis);
       ObjectInputStream ois =
           new ObjectInputStream(bis);
       al = (ArrayList)ois.readObject();
       ois.close();
     }
     catch (Throwable e) {
       System.err.println(e);
     }
   }
 }
```

I/O 기능 수행시 속도 향상을 위해 버퍼링을 사용했다는 것을 주목하라.

크기가 큰 데이터를 쓰거나 읽을 때 직렬화 보다 더 빠른 방법이 있을까? 아마도 특별한 클래스들을 제외한다면 없을것이다. 예를 들어 8바이트의 집합 대신에 텍스트로 64비트 long integer를 출력해야 한다고 생각해보자. long integer의 최대 크기는 텍스트로 20 캐릭터 정도이고 바이너리 표현으로는 2.5배 정도 더 길다. 그러므로 , 이러한 형식(format)으로는 더 빠르지 않을것처럼 보인다. 그러나 비트맵 같은 몇몇 경우에 특별한 형식(format)은 성능향상이 있을지 모른다. 그러나 직렬화에 의해 제공되는 표준방법외에 자신만의 고유한 방법을 사용한다면 약간의 트레이드오프를 포함할 것이다.

실제 I/O와 직렬화 형식화 비용(DataInputStream과 DataOutputStream 사용) 너머에 역직렬화시에 새로운 객체 생성 같은 다른 비용이 발생한다.

DataInputStream 의 메소드들은 다음 예처럼 반-사용자정의(semi-custom) 데이터 형식화에 사용할 수 있다는것에 주목하라.

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

 public class binary1 {
   public static void main(String args[]) {
     try {
       FileOutputStream fos =
           new FileOutputStream("outdata");
       BufferedOutputStream bos =
           new BufferedOutputStream(fos);
       DataOutputStream dos =
           new DataOutputStream(bos);
       Random rn = new Random();
       final int N = 10;
       dos.writeInt(N);
       for (int i = 1; i <= N; i++) {
         int r = rn.nextInt();
         System.out.println(r);
         dos.writeInt(r);
       }
       dos.close();
     }
     catch (IOException e) {
       System.err.println(e);
```

```
      }
    }
  }
```

그리고

```
import java.io.*;

 public class binary2 {
   public static void main(String args[]) {
     try {
       FileInputStream fis =
           new FileInputStream("outdata");
       BufferedInputStream bis =
           new BufferedInputStream(fis);
       DataInputStream dis =
           new DataInputStream(bis);
       int N = dis.readInt();
       for (int i = 1; i <= N; i++) {
         int r = dis.readInt();
         System.out.println(r);
       }
       dis.close();
     }
     catch (IOException e) {
       System.err.println(e);
     }
   }
 }
```

이 프로그램들은 파일에 10개의 정수를 쓰고 다시 읽어 들인다.

파일에 대한 정보 포함하기(Obtaining Information About Files)

여태껏 우리의 토론은 각 파일에 대한 입력과 출력에 집중되었다. 그러나 I/O 성능 향상에 다른 관점에서 본다면 파일들의 프로퍼티들도 관련된다는 것을 알수 있다. 예를 들어 파일이름의 길이를 출력하는 작은 프로그램을 생각해보자.

```
import java.io.*;

 public class length1 {
   public static void main(String args[]) {
     if (args.length != 1) {
       System.err.println("missing filename");
       System.exit(1);
     }
     File f = new File(args[0]);
     long len = f.length();
     System.out.println(len);
   }
 }
```

자바 런타임 시스템은 스스로 파일의 길이를 알수가 없어서 이러한 정보를 가지고 있는 OS에게 요청을 해야 한다. 파일이 디렉토리 인지 아닌지 파일이 마지막으로 변경된 시간 등과 같은 파일 정보들도 가지고 있다. java.io 패키지의 File 클래스는 이러한 정보를 요청하는 메소드들의 집합을 제공한다. 시간의 관점에서 이러한 요청은 보통 매우 비싸므로 가능한 적게 사용해야 한다.

파일 정보를 요청하는 좀 더 긴 예제를 보면 다음에 보이는 것처럼 시스템에서 모든 파일 패스네임을 얻기 위해 파일 시스템에 재귀적으로 접근한다.

```
import java.io.*;

 public class roots {
   public static void visit(File f) {
     System.out.println(f);
   }

   public static void walk(File f) {
     visit(f);
     if (f.isDirectory()) {
       String list[] = f.list();
       for (int i = 0; i < list.length; i++)
         walk(new File(f, list[i]));
     }
   }
```

```
    public static void main(String args[]) {
      File list[] = File.listRoots();
      for (int i = 0; i < list.length; i++) {
        if (list[i].exists())
          walk(list[i]);
        else
          System.err.println("not accessible: "
              + list[i]);
      }
    }
  }
```

이 예제는 디렉토리 구조를 알아내기 위해 isDirectory 와 exists 같은 File 메소드를 사용한다. 각 파일은 형태(plain 파일 혹은 디렉토리)에 따라 정확히 한번 호출된다.

추가 정보(Further Information)

기사 JDC Performance Tips and Firewall Tunneling Techniques 에서 자바 어플리케이션 성능향상을 위한 일반적인 방법을 논의한다.

Don Knuth의 책 *The Art of Computer Programming* , Volume 3 에서 B-Tree사용법과 같은 외부 소팅과 서치 알고리즘에 대해 논의한다.