



7.4.1 결론



- ❖ 귀찮은 에러처리
 - ❖ 에러처리를 하면 에러구문을 포함시켜야 하는 수고는 있다.
- ❖ 에러처리의 유용함
 - ❖ 에러처리의 사용으로 프로그램이 동작하는 동안에 발생하는 에러를 찾을 수 있다면 10번이라도 사용하겠다.
 - ❖ 비록 소스는 훨씬 길어지겠지만!



7. Java Stream





INDEX



- 9.1 배경
- 9.2 STREAM
- 9.3 표준 스트림과 FILE 클래스
- 9.4 FILE 스트림
- 9.5 MEMORY 스트림
- 9.6 2차 스트림
- 9.7 마무리



9.1 배경



- 9.1.1 개요





9.1.1 개요



- ❖ 입출력을 위한 공통된 방법
 - ❖ 다양한 장소에 존재하는 데이터들을 핸들하기 위해서는 입출력 데이터를 처리하는 공통된 방법이 있어야 한다.
- ❖ 스트림(Stream)의 정의
 - ❖ 자료의 입출력을 도와주는 중간 매개체
- ❖ 입출력 장치
 - ❖ 파일 디스크(파일)
 - ❖ 키보드, 모니터, 마우스
 - ❖ 메모리
 - ❖ 네트워크



9.2 Stream



- 9.2.1 스트림이란?
- 9.2.2 스트림의 원리
- 9.2.3 스트림의 종류
- 9.2.4 문자 스트림과 바이트 스트림
- 9.2.5 결론





9.2.1 스트림이란?



- ❖ 스트림(Stream)
 - ❖ 장치(Device- 하드웨어 장치)로부터 데이터를 읽거나 기록할 때 사용하는 **중간 매개체 역할을** 하는 놈
- ❖ 입출력 장치의 일반적인 특징
 - ❖ 일반적인 입출력 장치는 대부분 데이터를 **읽고 기록**한다는 특징이 있다.
- ❖ 스트림의 역할
 - ❖ 스트림은 데이터를 **읽고 쓰기** 위한 **공통된 방법을 제공**



9.2.1 스트림이란?(계속)



- ❖ 장치(Device)와 스트림(Stream)
 - ❖ 장치마다 연결할 수 있는 각각의 **스트림이 존재**
- ❖ 스트림의 내부 동작원리는 몰라도 된다. **사용할 줄만** 알면 된다.
 - ❖ 사용자는 스트림이 어떻게 장치와 연결되고 작업이 되는지 몰라도 된다.
 - ❖ 단지 스트림을 어떻게 생성하며 어떻게 **데이터를 읽고 쓰는 지만 알면 된다.**





9.2.2 스트림의 원리



- ❖ 스트림이란?
 - ❖ 스트림이란 **빨대다.**
- ❖ 스트림으로부터 데이터 읽기
 - ❖ 목표지점에 적절한 입력용 **스트림을 생성한다.**
 - ❖ 스트림으로부터 데이터를 읽는다.
 - ❖ 스트림으로부터 필요한 만큼 계속해서 데이터를 읽는다.
 - ❖ 스트림을 **닫는다.**
- ❖ 빨대로부터 음료수 마시기
 - ❖ 적절한 빨대를 음료수에 **꽂는다.**
 - ❖ 빨대로부터 음료수를 흡입한다.
 - ❖ 빨대로부터 마시고 싶은 만큼 계속해서 음료수를 흡입한다.
 - ❖ 빨대를 **제거한다.**



9.2.2 스트림의 원리



- ❖ 스트림에 데이터 기록하기
 - ❖ 목표지점에 적절한 출력용 **스트림을 생성한다.**
 - ❖ 스트림에 데이터를 기록한다.
 - ❖ 스트림에 필요한 만큼 계속해서 데이터를 기록한다.
 - ❖ 스트림을 **닫는다.**
- ❖ 빨대로 음료수 내뱉기
 - ❖ 적절한 빨대를 **컵에 꽂는다.**
 - ❖ 빨대로 입안에 있는 음료수를 내뱉는다.
 - ❖ 빨대로 내뱉고 싶은 만큼 음료수를 내뱉는다.
 - ❖ 빨대로 **제거한다.**
- ❖ 스트림의 기본적인 종류
 - ❖ **입력용** 스트림과 **출력용** 스트림이 있다.
 - ❖ 음료수를 내뱉고 다시 마시려면 **빨대가 2개** 필요하다.

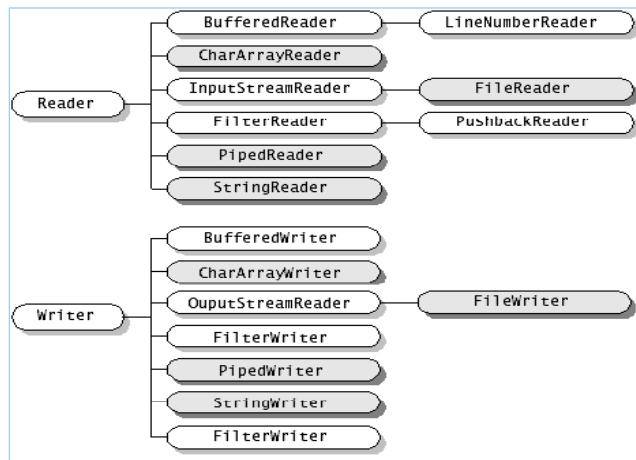




9.2.3 스트림의 종류



그림 9-1 문자 스트림의 구성도



❖ 문자 스트림의 패턴

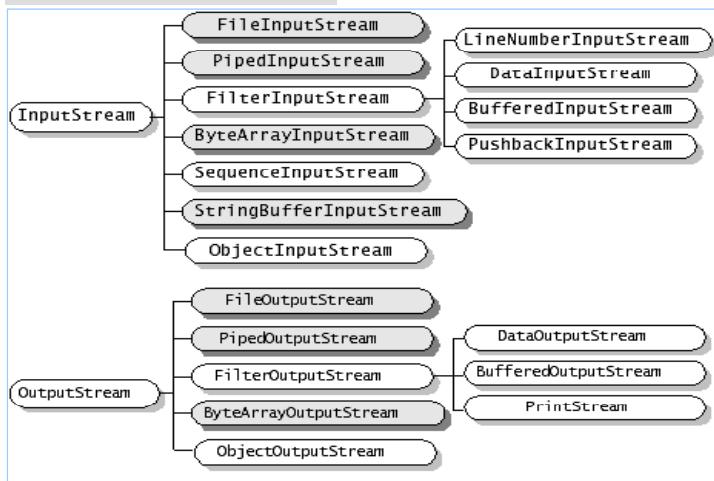
- ❖ Reader는 **입력용 문자 스트림**이다.
- ❖ Writer는 **출력용 문자 스트림**이다.
- ❖ Reader나 Writer가 붙는다면 **문자 스트림**이다.



9.2.3 스트림의 종류(계속)



그림 9-2 바이트 스트림의 구성도



❖ 바이트 스트림의 패턴

- ❖ InputStream은 **입력용 바이트 스트림**이다.
- ❖ OutputStream은 **출력용 바이트 스트림**이다.
- ❖ 'InputStream'나 'OutputStream'이라는 단어가 붙어 있다면 **바이트 스트림**이다.
- ❖ PrintStream은 출력용만 존재하기 때문에 약간 특이하다.





9.2.3 스트림의 종류(계속)



- ❖ 스트림의 종류 I
 - ❖ 문자 스트림: Reader, Writer(문자 단위로 처리)
 - ❖ 바이트 스트림: InputStream, OutputStream(바이트 단위로 처리)
- ❖ 스트림의 종류 II
 - ❖ 입력 스트림: Reader나 InputStream
 - ❖ 출력 스트림: Writer나 OutputStream
- ❖ 스트림의 규칙
 - ❖ InputStream과 Reader는 [읽어 들이는 메서드를 포함하고 있어야 한다.](#)
 - ❖ OutputStream과 Writer는 [기록하는 메서드를 포함하고 있어야 한다.](#)



9.2.3 스트림의 종류(계속)



- ❖ **바이트** 단위의 **read()**메서드
 - ❖ int read()
 - ❖ int read(byte cbuf[])
 - ❖ int read(byte cbuf[], int offset, int length)
- ❖ **문자** 단위의 **read()**메서드
 - ❖ int read()
 - ❖ int read(char cbuf[])
 - ❖ int read(char cbuf[], int offset, int length)
- ❖ **바이트** 단위의 **writer()**메서드
 - ❖ int writer(int c)
 - ❖ int writer(byte cbuf[])
 - ❖ int writer(byte cbuf[], int offset, int length)
- ❖ **문자** 단위의 **writer()**메서드
 - ❖ int writer()
 - ❖ int writer(char cbuf[])
 - ❖ int writer(char cbuf[], int offset, int length)



9.2.4 문자 스트림과 바이트 스트림

- ❖ 스트림을 바이트로 핸들
 - ❖ 바이트 스트림 : **바이트** 단위
 - ❖ 원시 바이트를 그대로 주고 받겠다는 의미
- ❖ 스트림을 문자로 핸들
 - ❖ 문자 스트림: **문자** 단위
 - ❖ 원시 바이트를 2바이트씩 묶어서 사용할 수도 있고, 1바이트 단위로도 사용할 수 있다.
 - ❖ 자바에서 사용하는 문자방식은 **유니코드(Unicode)** 방식
 - ❖ 그래서 바이트로 전송되어지는 것을 **스트림에서 재해석한 후 유니코드 문자로 변환하게** 된다.
 - ❖ 결과적으로 바이트를 문자로 가공을 하는 것이며, 문자의 인코딩은 문자 스트림에서 자동으로 해석하게 된다.



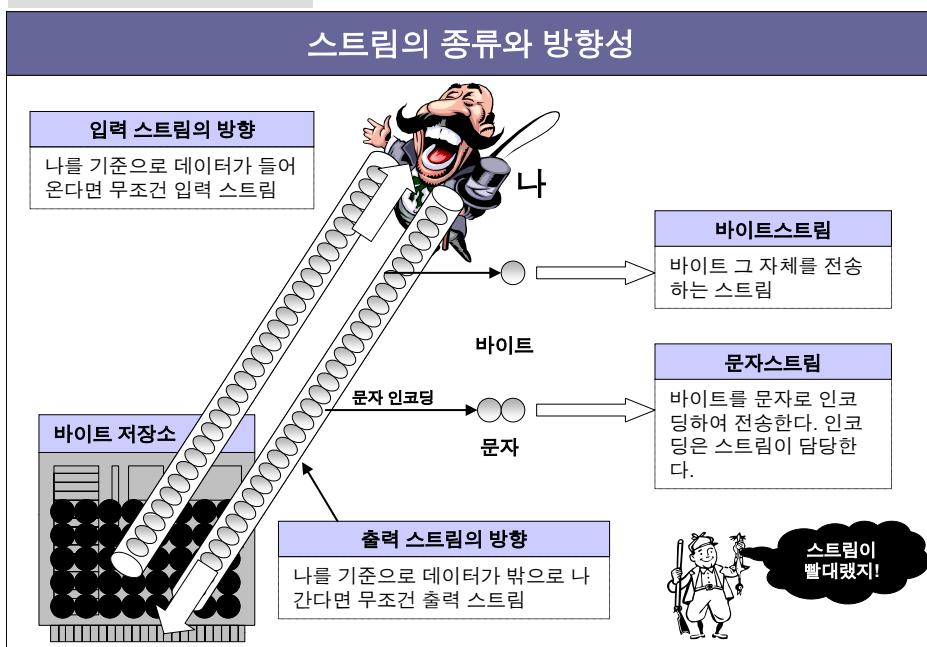
9.2.4 문자 스트림과 바이트 스트림(계속)

- ❖ 바이트 스트림(Byte Stream)
 - ❖ 데이터를 **바이트 단위로** 주고 받는 것을 말한다.
 - ❖ 대표적인 바이트 스트림
 - ❖ **InputStream** 과 **OutputStream**
 - ❖ 8bit의 이진 비트를 묶으면 바이트가 된다.
 - ❖ 원래 데이터는 모두 바이트이다.
 - ❖ zip이나 jar 압축파일도 바이트로 되어 있다.
 - ❖ 이 바이트들이 적절하게 변환되면 의미 있는 데이터가 된다.
- ❖ 문자 스트림(Char Stream)
 - ❖ 바이트들을 2바이트씩 묶어서 사용할 수도 있고, 1바이트 단위로도 사용할 수 있다.
 - ❖ 문자 인코딩에 따라서 다르게 사용된다.
 - ❖ 자바에서 사용하는 문자 방식은 **유니코드(Unicode)** 방식이다.
 - ❖ 바이트를 문자로 가공을 하는 것이며, 문자의 인코딩은 문자 스트림에서 자동으로 해석하게 된다.



9.2.4 문자 스트림과 바이트 스트림(계속)

그림 9-3 스트림의 입출력 원리



9.2.4 문자 스트림과 바이트 스트림(계속)

- ❖ **바이트 스트림들 (InputStream, OutputStream)**
 - ❖ `AudioInputStream`, `ByteArrayInputStream`, `FileInputStream`, `FilterInputStream`, `InputStream`
 - ❖ `ObjectInputStream`, `PipedInputStream`, `SequenceInputStream`, `StringBufferInputStream`
 - ❖ `BufferedInputStream`, `ByteArrayInputStream`, `DataInputStream`, `FilterInputStream`
 - ❖ `PushbackInputStream`, `ByteArrayOutputStream`, `FileOutputStream`, `FilterOutputStream`
 - ❖ `ObjectOutputStream`, `OutputStream`, `PipedOutputStream`, `BufferedOutputStream`
 - ❖ `ByteArrayOutputStream`, `DataOutputStream`, `FilterOutputStream`
- ❖ **문자 스트림들 (Reader, Writer)**
 - ❖ `Writer`, `BufferedWriter`, `CharArrayWriter`, `FilterWriter`, `OutputStreamWriter`, `FileWriter`, `PipedWriter`
 - ❖ `PrintWriter`, `StringWriter`, `BufferedReader`, `LineNumberReader`, `CharArrayReader`
 - ❖ `InputStreamReader`, `FileReader`, `FilterReader`, `PushbackReader`, `PipedReader`, `StringReader`



9.2.5 결론



- ❖ 스트림이란?
 - ❖ 데이터를 읽고 기록하기 위한 **중간 매개체 역할을** 담당
- ❖ 스트림 생성
 - ❖ 스트림의 **목표지점이** 있어야 한다.
 - ❖ 해당 목표지점에 스트림을 생성했다면 **데이터를 읽거나 기록할** 수 있다.
- ❖ 스트림의 공통된 특징
 - ❖ 입력 스트림의 경우 : **read()** 계열의 메서드를 사용
 - ❖ 출력 스트림의 경우 : **write()** 계열의 메서드를 사용해서 데이터를 기록



9.3 표준 스트림과 File 클래스



- 9.3.1 표준 출력
- 9.3.2 System.in
- 9.3.3 File 클래스
- 9.3.4 파일 목록 출력하기
- 9.3.5 디렉토리 생성, 삭제, 파일이름 변경





9.3.1 표준 출력



- ❖ 표준 입출력 스트림
 - ❖ 스트림 중에서 기본적으로 가장 많이 사용되는 스트림
 - ❖ java.lang 패키지의 System 클래스에 스태틱 멤버로 선언 되어 있음
 - ❖ 자동으로 초기화된다.
- ❖ System 클래스의 표준 입출력 스트림 멤버
 - ❖ package java.lang;
 - ❖ public class System{
 - ❖ public static PrintStream out;
 - ❖ public static InputStream in;
 - ❖ public static PrintStream err;
 - ❖ //중간 생략
 - ❖ }



9.3.1 표준 출력(출력)



- ❖ System.out
 - ❖ System.out.println("에러메시지");
 - ❖ 콘솔 화면에 문자열을 출력하기 위한 용도
- ❖ System.in
 - ❖ int d = System.in.read(); //한 바이트 읽어내기
 - ❖ 키보드의 입력을 받아들이기 위해서 사용하는 표준 입력 스트림
 - ❖ 키보드에 문자를 입력한 후 엔터키를 눌렀을 때 System.in으로 문자들이 들어오며 read()를 이용해서 한 바이트씩 읽어낸다.
- ❖ System.err
 - ❖ System.err.println("데이터")
 - ❖ out과 같은 PrintStream이지만 에러 메시지의 출력을 위해서 사용하는 표준 에러 스트림(Standard Error Stream)
- ❖ in, out, err라는 3개의 스트림은 미리 생성되어 있으며, 여러분들이 System 클래스의 **이름으로 접근해서 사용할 수 있다.**





9.3.2 System.in



§9-1 SystemInMain.java

```
01: /**
02:  System.in을 테스트하기 위한 예제
03: */
04: import java.io.*;
05: public class SystemInMain{
06:     public static void main(String[] args) throws IOException{
07:         System.out.print("엔터를 누르세요");
08:         int i = System.in.read();
09:         System.out.println(i);
10:     } //end of main
11: } //end of SystemInMain class
```

```
C:\javasrc\chap09>javac SystemInMain.java
```

```
C:\javasrc\chap09>java SystemInMain
```

```
엔터를 누르세요
```

```
13
```



9.3.2 System.in(계속)



§9-2 SystemInMain2.java

```
01: /**
02:  2개의 문자를 읽어 내는 System.in
03: */
04: import java.io.*;
05: public class SystemInMain2{
06:     public static void main(String[] args) throws IOException{
07:         System.out.print("엔터를 누르세요");
08:         int i = System.in.read();
09:         System.out.println(i);
10:         i = System.in.read();
11:         System.out.println(i);
12:     } //end of main
13: } //end of SystemInMain2 class
```

```
C:\javasrc\chap09>javac SystemInMain2.java
```

```
C:\javasrc\chap09>java SystemInMain2
```

```
엔터를 누르세요
```

```
13
```

```
10
```





9.3.2 System.in(계속)



§9-3 SystemInMain3.java

```
01:  /**
02:   한 라인을 읽어내는 System.in
03:  */
04:  import java.io.*;
05:  public class SystemInMain3{
06:      public static void main(String[] args) throws IOException{
07:          System.out.print("문자를 입력한 후 엔터를 누르세요? ");
08:          int i;
09:          while( (i = System.in.read()) != '\n' ){
10:              System.out.println((char)i + ":" + i);
11:          }
12:      } //end of main
13:  } //end of SystemInMain3 class
```

```
C:\javasrc\chap09>javac SystemInMain3.java
C:\javasrc\chap09>java SystemInMain3
H:72
e:101
l:108
l:108
o:111
:13
```



9.3.2 System.in(계속)



- ❖ 문자와 수의 출력
 - ❖ System.out.println((char)i + ":" + i);
- ❖ 참고 |
 - ❖ '\n'은 문자입니다.
 - ❖ 하지만 프로그래머라면 이것을 **숫자로** 보아야 한다.
 - ❖ 프로그램에서는 문자에 해당하는 숫자를 문자표에서 찾아서 숫자로 **해석하기 때문에**, 프로그래머 입장에서는 문자라고 생각하기보다는 아예 **처음부터 숫자로** 봐야 한다.
- ❖ 참고 ||
 - ❖ 위의 예제에서 **throws IOException**을 사용했다.
 - ❖ System.in.read()를 사용하면 **IOException 처리를** 해주어야 하지만 가상머신에게 **에러처리를 미루고** 있다.
 - ❖ 이것에 대한 자세한 사항은 7장의 에러처리 미루기 부분을 참고





9.3.3 File 클래스



- ❖ File 클래스
 - ❖ 파일과 디렉토리에 대한 정보를 제공하고 관리할 수 있게 해준다.
 - ❖ File 클래스 자체적으로는 파일을 상대로 데이터를 읽어내고 기록할 수 없다.
- ❖ File 객체 생성의 예
 - ❖ File f1 = new File("파일이름");
 - ❖ File f2 = new File("디렉토리경로");
- ❖ 참고
 - ❖ 파일과 디렉토리에 관련된 정보 추출과 관리를 하기 위해서 파일명과 디렉토리명을 이용해서 File 객체를 생성할 수 있다.



§ 9-4 FileMain.java



```
01: /**
02:  파일의 정보를 출력하는 예제
03: */
04: import java.io.*;
05: import java.net.*;
06: import java.util.*;
07: public class FileMain {
08:  public static void main(String[] args) throws MalformedURLException{
09:   File f = new File("FileMain.java");
10:   PrintStream out = System.out;
11:   out.println("isFile(): " + f.isFile()); //파일인지 아닌지
12:   out.println("isDirectory(): " + f.isDirectory()); //디렉토리인지 아닌지
13:   out.println("isHidden(): " + f.isHidden()); //숨김 파일인지
14:   out.println("lastModified(): " + f.lastModified()); //마지막에 수정된 날짜
15:   out.println("canRead(): " + f.canRead()); //읽기 속성을 가진 파일인지
16:   out.println("canWrite(): " + f.canWrite()); //쓰기 속성을 가진 파일인지
17:   out.println("getPath(): " + f.getPath()); //상대 경로
18:   out.println("getAbsolutePath(): " + f.getAbsolutePath()); //절대 경로
19:   out.println("getName(): " + f.getName()); //디렉토리 또는 파일의 이름
20:   out.println("toURL(): " + f.toURL()); //URL형식의 경로
21:   out.println("exists(): " + f.exists()); //파일이 존재하는지
22:   out.println("length(): " + f.length()); //파일의 길이
23:  } //end of main
24: } //end of FileMain class
```





9.3.3 File 클래스(계속)



```
C:\javasrc\chap09>javac FileMain.java
C:\javasrc\chap09>java FileMain
isFile():
true
isDirectory():
false
isHidden():
false
lastModified():
1087998966206
canRead():
true
canWrite():
true
getPath():
FileMain.java
getAbsolutePath():
C:\javasrc\chap09\FileMain.java
getName():
FileMain.java
toURL():
file:/C:/javasrc/chap09/FileMain.java
exists():
true
length():
1253
```

- ❖ File 객체로 출력할 수 있는 정보들
 - ❖ f.isFile(): 파일인지 아닌지
 - ❖ f.isDirectory(): 디렉토리인지 아닌지
 - ❖ f.isHidden(): 숨김 파일인지
 - ❖ f.lastModified(): 마지막에 수정된 날짜
 - ❖ f.canRead(): 읽기 속성을 가진 파일인지
 - ❖ f.canWrite(): 쓰기 속성을 가진 파일인지
 - ❖ f.getPath(): 상대 경로
 - ❖ f.getAbsolutePath(): 절대 경로
 - ❖ f.getName(): 디렉토리 또는 파일의 이름
 - ❖ f.toURL(): URL형식의 경로
 - ❖ f.exists(): 파일이 존재하는지
 - ❖ f.length(): 파일의 길이



9.3.4 파일 목록 출력하기



- ❖ 현재 디렉토리의 File 객체 생성
 - ❖ File dir1 = new File(".");
 - ❖ 현재 디렉토리의 정보를 얻기 위해서 사용
- ❖ 상위 디렉토리의 File 객체 생성
 - ❖ File dir2 = new File("../");
 - ❖ 상위 디렉토리의 정보를 얻기 위해서 사용



§9-5 CurrentDirMain.java

```
01:  /**
02:   현재 디렉토리와 상위 디렉토리의 경로 출력
03:  */
04: import java.io.*;
05:public class CurrentDirMain {
06: public static void main (String args[]) throws IOException{
07:     File dir1 = new File(".");
08:     File dir2 = new File("../");
09:     //AbsolutePath
10:    System.out.println("AbsolutePath");
11:    System.out.println ("Current dir-> " + dir1.getAbsolutePath());
12:    System.out.println ("Parent dir-> " + dir2.getAbsolutePath());
13:    //CanonicalPath
14:    System.out.println("CanonicalPath");
15:    System.out.println ("Current dir-> " + dir1.getCanonicalPath());
16:    System.out.println ("Parent dir-> " + dir2.getCanonicalPath());
17:    //절대경로지정
18:    File dir3 = new File("c:\\\\");
19:    System.out.println ("c:\\\\ -> " + dir3.getAbsolutePath());
20: } //end of main
21: } //end of CurrentDirMain class
```

```
C:\javasrc\chap09>javac CurrentDirMain.java
```

```
C:\javasrc\chap09>java CurrentDirMain
```

```
AbsolutePath
Current dir-> C:\javasrc\chap09\
Parent dir-> C:\javasrc\chap09\..
CanonicalPath
Current dir-> C:\javasrc\chap09
Parent dir-> C:\javasrc
c:\\ -> c:\\
```

디렉토리를 이용한 `File` 객체를 생성하는 방법을 배웠으니 이제 파일 목록을 출력해 보는 예제를 해보자

§9-6 FileListMain.java

```
01:  /**
02:   File 클래스의 listFiles() - 디렉토리 목록 보기
03:  */
04: import java.io.*;
05: public class FileListMain{
06:     public static void main(String[] args){
07:         File f = new File("c:\\\\");
08:         File[] fs = f.listFiles();
09:         for(int i=0; i<fs.length; i++){
10:             System.out.println(fs[i].getName());
11:         }
12:     } //end of main
13: } //end of FileListMain class
```

```
C:\javasrc\chap09>javac FileListMain.java
```

```
C:\javasrc\chap09>java FileListMain
```

```
AUTOEXEC.BAT
boot.ini
bootfont.bin
CONFIG.SYS
Documents and Settings
Inetpub
IO.SYS
jakarta
javasrc
MSDOS.SYS
Program Files
RECYCLER
System Volume Information
WINDOWS
```

9.3.4 파일 목록 출력하기(계속)

- ❖ 디렉토리 내에 존재하는 File 목록 얻어내기
 - ❖ File f = new File("C:\\\\");
 - ❖ File[] fs = f.listFiles(); //파일과 디렉토리 목록을 포함한 배열
 - ❖ 파일과 디렉토리 목록의 이름을 출력하기 위해서 다음과 같이 File 클래스의 멤버 메서드인 **getName()**을 호출하고 있다.
- ❖ 파일과 디렉토리 목록의 이름을 출력
 - ❖ for(int i=0; i<fs.length; i++){
 - ❖ System.out.println(fs[i].getName());
 - ❖ }
- ❖ 파일과 디렉토리를 구분하는 메서드
 - ❖ boolean isDirectory()
 - ❖ boolean isFile()



9.3.5 디렉토리 생성, 삭제, 파일이름 변경

§9-7 FileMakeMain.java

```
01:  /**
02:  디렉토리 생성
03:  */
04:  import java.io.*;
05:  public class FileMakeMain{
06:      public static void main(String[] args) throws IOException{
07:          File f = new File("NewFolder");
08:          if(!f.exists()){
09:              f.mkdir();
10:              System.out.println("NewFolder 디렉토리를 생성 완료");
11:          }
12:      } //end of main
13:  } //end of FileMakeMain class
```

C:\javasrc\chap09>javac FileMakeMain.java
C:\javasrc\chap09>java FileMakeMain
NewFolder 디렉토리를 생성 완료

```
C:\javasrc\chap09>dir N*
08-06 오후 04:19    <DIR>          NewFolder
0개 파일           0 바이트
```

- ❖ 디렉토리의 생성
 - ❖ File f = new File("디렉토리명");
 - ❖ if(!f.exists()){
 - ❖ f.mkdir();
 - ❖ }



9.3.5 디렉토리 생성, 삭제, 파일이름 변경(계속)

§9-8 FileDeleteMain.java

```
01:  /**
02:  디렉토리 삭제
03:  */
04:  import java.io.*;
05:  public class FileDeleteMain{
06:      public static void main(String[] args) throws IOException{
07:          File f = new File("NewFolder");
08:          if(f.exists()){
09:              f.delete();
10:              System.out.println("NewFolder 디렉토리를 삭제 완료");
11:          }
12:      } //end of main
13:  } //end of FileDeleteMain class
```

```
C:\javasrc\chap09>javac FileDeleteMain.java
C:\javasrc\chap09>java FileDeleteMain
NewFolder 디렉토리를 삭제 완료
```

```
C:\javasrc\chap09>dir N*
파일을 찾을 수 없습니다.
```

- ❖ File f = new File("NewFolder");
 - ❖ //디렉토리의 경우 비어 있어야 한다.
 - ❖ if(f.exists()){ //디렉토리가 존재하는지 확인
 - ❖ f.delete(); //디렉토리 삭제
 - ❖ }

9.3.5 디렉토리 생성, 삭제, 파일이름 변경(계속)

§9-9 FileRenameMain.java

```
01:  /**
02:  파일이름 변경
03:  */
04:  import java.io.*;
05:  public class FileRenameMain{
06:      public static void main(String[] args) throws IOException{
07:          File f = new File("FileRenameMain.class");
08:          File t = new File("FileRenameMain_backup.class");
09:          if(f.exists()){
10:              f.renameTo(t);
11:              System.out.println("이름바꾸기 완료");
12:              System.out.print("FileRenameMain.class->");
13:              System.out.println("FileRenameMain_backup.class");
14:          }
15:      } //end of main
16:  } //end of FileRenameMain class
```

```
C:\javasrc\chap09>javac FileRenameMain.java
C:\javasrc\chap09>java FileRenameMain
이름바꾸기 완료
FileRenameMain.class->FileRenameMain_backup.class
```

```
C:\javasrc\chap09>dir FileRenameMain*
07-04 오후 03:49    764 FileRenameMain.class
07-04 오후 03:49    864 FileRenameMain.java
01-05 오후 07:35    547 FileRenameMain_backup.class
                    3개 파일     2,175 바이트
```

- ❖ 파일의 이름을 변경하는 예
 - ❖ File f = new File("원본파일명");
 - ❖ File t = new File("변경할파일명");
 - ❖ if(f.exists()){
 - ❖ f.renameTo(t);
 - ❖ }

§9-21 BufferedFileCopy.java

```
01:  /**
02:  Buffered 스트림을 이용한 파일 복사 예제(한 바이트 단위)
03:  */
04:  import java.io.*;
05:  public class BufferedFileCopy {
06:      //BufferedFileCopy 프로그램의 실행
07:      //java BufferedFileCopy 원본파일이름 목표파일이름
08:      //예) java BufferedFileCopy s.exe t.exe
09:      //s.exe는 같은 디렉토리 상에 존재해야 한다.
10:      public static void main(String[] args) throws IOException{
11:          int i, len=0;
12:          FileInputStream fis = new FileInputStream(args[0]);
13:          FileOutputStream fos = new FileOutputStream(args[1]);
14:          BufferedInputStream bis = new BufferedInputStream(fis);
15:          BufferedOutputStream bos = new BufferedOutputStream(fos);
16:          long psecond = System.currentTimeMillis();
17:          while((i=bis.read()) != -1){
18:              bos.write(i);
19:              len++;
20:          }
21:          bis.close();
22:          bos.close();
23:          psecond = System.currentTimeMillis() - psecond;
24:          System.out.println(len + " bytes " + psecond +" milliseconds");
25:      } //end of main
26:  } //end of BufferedFileCopy class
```

```
C:\javasrc\chap09>javac BufferedFileCopy.java
C:\javasrc\chap09>java BufferedFileCopy s.exe t.exe
6592232 bytes 563 milliseconds
```

```
C:\javasrc\chap09>dir t.exe
오후 04:14           6,592,232 t.exe
```

❖ 스트림의 필수

❖ 데이터의 용량이 큰 경우 무조건 Buffered 스트림으로 변환해서 사용한다.

§9-26 ByteToCharMain.java

```
01:  /**
02:  바이트 스트림을 문자 스트림으로 변환하는 예제
03:  */
04:  import java.io.*;
05:  public class ByteToCharMain{
06:      //ByteToCharMain 프로그램의 실행
07:      //java ByteToCharMain 원본파일이름 목표파일이름
08:      //예) java ByteToCharMain ByteToCharMain.java tmp.txt
09:      public static void main (String[] args) throws IOException {
10:          FileInputStream fis = new FileInputStream (args[0]);
11:          FileOutputStream fos = new FileOutputStream (args[1]);
12:          InputStreamReader isr = new InputStreamReader (fis);
13:          OutputStreamWriter osw = new OutputStreamWriter (fos);
14:          int i;
15:          while((i = isr.read()) != -1){
16:              osw.write (i);
17:          }
18:          osw.close();
19:          isr.close();
20:          System.out.println("작업완료");
21:      } //end of main
22:  } //end of ByteToCharMain class
```

```
C:\javasrc\chap09>javac ByteToCharMain.java
C:\javasrc\chap09>java ByteToCharMain ByteToCharMain.java tmp.txt
작업완료
```

```
C:\javasrc\chap09>dir tmp.txt
오후 04:29           867 tmp.txt
```



9.4 File 스트림



- 9.4.1 파일 입출력의 종류
- 9.4.2 바이트 단위의 출력 파일 스트림
- 9.4.3 바이트 단위의 입력 파일 스트림
- 9.4.4 문자 단위의 출력 파일 스트림
- 9.4.5 문자 단위의 입력 파일 스트림
- 9.4.6 파일복사
- 9.4.7 RandomAccessFile
- 9.4.8 결론



9.4.1 파일 입출력의 종류



- ❖ 파일 입출력을 위한 스트립
 - ❖ **FileInputStream**, **FileOutputStream** // **바이트** 단위
 - ❖ FileInputStream : 읽기
 - ❖ FileOutputStream : 쓰기
 - ❖ **FileReader**, **FileWriter** //**문자** 단위
 - ❖ FileReader : 읽기
 - ❖ FileWriter : 쓰기



9.4.2 바이트 단위의 출력 파일 스트림

- ❖ 바이트 단위의 File 출력 스트림의 생성
 - ❖ `FileOutputStream fos = new FileOutputStream("a.dat");`
 - ❖ `//fos로 쓰기 작업`
 - ❖ `fos.close(); //스트림 닫기`
- ❖ 바이트 단위의 File 입력 스트림의 생성
 - ❖ `FileInputStream fis = new FileInputStream("b.dat");`
 - ❖ `//fis로 읽기 작업`
 - ❖ `fis.close(); //스트림 닫기`

■ 스트림을 생성할 때 반드시 목표지점을 생각하면서 프로그램을 해야 한다.

9.4.2 바이트 단위의 출력 파일 스트림 (계속)

§9-10 FileStreamMain.java

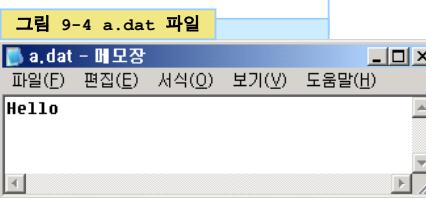
```
01: /**
02:  FileOutputStream으로 데이터 기록하기
03: */
04: import java.io.*;
05: public class FileStreamMain{
06:     public static void main(String[] args) throws IOException{
07:         FileOutputStream fos = new FileOutputStream("a.dat");
08:         fos.write(72);
09:         fos.write(101);
10:         fos.write(108);
11:         fos.write(108);
12:         fos.write(111);
13:         fos.close();
14:         System.out.println("a.dat 파일 기록완료");
15:     } //end of main
16: } // end of FileStreamMain class
```

```
C:\javasrc\chap09>javac FileStreamMain.java
C:\javasrc\chap09>java FileStreamMain
```

```
a.dat 파일 기록완료
```

```
C:\javasrc\chap09>type a.dat
Hello
```

- ❖ `FileOutputStream의 write()` 메서드
 - ❖ `void write(byte[] b)`
 - ❖ `void write(byte[] b, int off, int len)`
 - ❖ `void write(int b) //사용`



9.4.2 바이트 단위의 출력 파일 스트림 (계속)

§ 9-11 FileStreamMain2.java

```
01: /**
02: FileOutputStream으로 데이터 기록하기 II
03: */
04: import java.io.*;
05: public class FileStreamMain2{
06:     public static void main(String[] args) throws IOException{
07:         FileOutputStream fos = new FileOutputStream("a1.dat");
08:         byte[] b = new byte[]{72, 101, 108, 108, 111};
09:         fos.write(b);
10:         fos.close();
11:         System.out.println("a1.dat 파일 기록완료");
12:     } //end of main
13: } // end of FileStreamMain2 class
```

```
C:\javasrc\chap09>javac FileStreamMain2.java
C:\javasrc\chap09>java FileStreamMain2
a1.dat 파일 기록완료
```

```
C:\javasrc\chap09>type a1.dat
Hello
```

- ❖ FileOutputStream의 write() 메서드
 - ❖ void write(byte[] b)//사용
 - ❖ void write(byte[] b, int off, int len)
 - ❖ void write(int b)

9.4.2 바이트 단위의 출력 파일 스트림 (계속)

- ❖ byte 배열 생성
 - ❖ byte[] b = new byte[]{72, 101, 108, 108, 111};
- ❖ write(byte[] b, int off, int len)
 - ❖ 배열 b에서 인덱스가 off인 위치부터 len 개를 기록하는 메서드이다. 즉 배열 b에서 일부분만을 기록하는 메서드이다.
- ❖ 파일의 존재여부 확인
 - ❖ File f = new File("a.dat");
 - ❖ boolean b = f.exists();
- ❖ 덮어쓰기 형태로 파일 출력 스트림 생성
 - ❖ FileOutputStream fos = new FileOutputStream("a.dat", false);
- ❖ 존재하는 파일에 덧붙여쓰기 형태로 파일 출력 스트림 생성
 - ❖ FileOutputStream fos = new FileOutputStream("a.dat", true);

9.4.2 바이트 단위의 출력 파일 스트림 (계속)

§ 9-12 FileStreamMain3.java

```
01:  /**
02:   * 존재하는 파일의 끝에 덧붙여쓰기
03:  */
04:  import java.io.*;
05:  public class FileStreamMain3{
06:      public static void main(String[] args) throws IOException{
07:          File f = new File("a.dat");
08:          if(f.exists()){
09:              FileOutputStream fos = new FileOutputStream("a.dat", true);
10:              byte[] b = new byte[]{72, 101, 108, 108, 111};
11:              fos.write(b);
12:              fos.close();
13:              System.out.println("a.dat 파일의 끝부분에 데이터 추가하기 완료");
14:          }
15:      } //end of main
16:  } // end of FileStreamMain3 class
```

C:\javasrc\chap09>javac FileStreamMain3.java
C:\javasrc\chap09>java FileStreamMain3
a.dat 파일의 끝부분에 데이터 추가하기 완료

C:\javasrc\chap09>type a.dat
HelloHello



9.4.3. 바이트 단위의 입력 파일 스트림

- ❖ FileInputStream의 생성
 - ❖ FileInputStream fis = new FileInputStream("a.dat");
 - ❖ //fis로 읽기 작업
 - ❖ fis.close(); //스트림 닫기
- ❖ FileInputStream의 read() 계열의 메서드
 - ❖ int read()
 - ❖ int read(byte[] b)
 - ❖ int read(byte[] b, int off, int len)
- ❖ 스트림으로부터 한 바이트씩 끝까지 읽어내기
 - ❖ while(i=fis.read()) != -1{
 - ❖ System.out.print((char)i);
 - ❖ }



9.4.3. 바이트 단위의 입력 파일 스트림 (계속)

§9-13 FileStreamMain4.java

```
01: /**
02:  FileInputStream을 이용해서 파일 읽기
03: */
04: import java.io.*;
05: public class FileStreamMain4{
06:     public static void main(String[] args) throws IOException{
07:         FileInputStream fis = new FileInputStream("a.dat");
08:         int i;
09:         while( (i=fis.read()) != -1 ){
10:             System.out.print((char)i);
11:         }
12:         fis.close();
13:     } //end of main
14: } // end of FileStreamMain4 class
```

❖ 숫자를 문자로 출력하기
❖ System.out.print((char)i);

```
C:\javasrc\chap09>javac FileStreamMain4.java
C:\javasrc\chap09>java FileStreamMain4
HelloHello
```



9.4.3. 바이트 단위의 입력 파일 스트림 (계속)

§9-14 FileStreamMain5.java

```
01: /**
02:  FileInputStream의 read(byte[] b)
03:  FileInputStream을 이용한 여러 바이트 읽기
04: */
05: import java.io.*;
06: public class FileStreamMain5{
07:     public static void main(String[] args) throws IOException{
08:         FileInputStream fis = new FileInputStream("FileStreamMain5.java");
09:         int count;
10:         byte[] b = new byte[10];
11:         while( (count=fis.read(b)) != -1 ){
12:             for(int i=0; i<count; i++){
13:                 System.out.print((char)b[i]);
14:             }
15:         }
16:         fis.close();
17:     } //end of main
18: } // end of FileStreamMain5 class
```

```
C:\javasrc\chap09>javac FileStreamMain5.java
C:\javasrc\chap09>java FileStreamMain5
파일 출력 생략
FileStreamMain5.java가 출력된다.
```



§9-15 FileStreamMain6.java

```
01:  /**
02:   * 하나의 배열로 파일에 존재하는 모든 데이터 읽어내기
03:  */
04:  import java.io.*;
05:  public class FileStreamMain6{
06:      public static void main(String[] args) throws IOException{
07:          //1. 파일 사이즈 알아내기
08:          File f = new File("FileStreamMain6.java");
09:          int fileSize = (int)f.length();
10:          System.out.println("파일의 사이즈:" + fileSize);
11:          //2. 파일 사이즈에 해당하는 배열 만들기
12:          byte[] b = new byte[fileSize];
13:          //3. 스트림을 이용해서 배열에 데이터 채우기
14:          FileInputStream fis = new FileInputStream("FileStreamMain6.java");
15:          int pos = 0;int size = 10 ;int temp;
16:          while( (size=fis.read(b, pos, size)) > 0 ){
17:              pos += size;
18:              temp = b.length - pos;
19:              if(temp < 10){
20:                  size = temp;
21:              }
22:          }
23:          fis.close();
24:          System.out.println("읽은 바이트수:" + pos);
25:          //4. 배열을 통째로 파일에 기록하기
26:          FileOutputStream fos = new FileOutputStream("test.txt");
27:          fos.write(b);
28:          fos.close();
29:      } //end of main
30:  } // end of FileStreamMain6 class
C:\javasrc\chap09>javac FileStreamMain6.java
C:\javasrc\chap09>java FileStreamMain6
파일의 사이즈:1148
읽은 바이트수:1148
```



9.4.3. 바이트 단위의 입력 파일 스트림 (계속)

- ❖ 오프셋의 위치를 표시하기 위해서 pos라는 변수를 두고 0에서 시작, 한번에 읽을 바이트 수를 size에 명시
 - ❖ int pos = 0;
 - ❖ int size = 10;
- ❖ pos는 읽은 바이트 수들의 합계를 계산하고, 전체 배열의 오프셋을 계산
 - ❖ **pos += size;**
- ❖ 남은 바이트 수를 계산하기 위해 전체 배열의 크기에서 읽은 바이트 수를 빼고 있다.
 - ❖ **temp = b.length - pos;**



9.4.3. 바이트 단위의 입력 파일 스트림 (계속)

- ❖ 남은 바이트 수가 0보다 작다면 while문을 종료
 - ❖ `while((size=fis.read(b, pos, size)) > 0){`
- ❖ 남은 바이트 수가 한번에 읽을 바이트 수보다 작다면 b.length - pos 만큼만 읽어냄으로써 정확한 크기의 데이터를 읽어내고 있다.
 - ❖ `if(temp < 10){`
 - ❖ `size = temp;`
 - ❖ `}`
- ❖ 전체 데이터를 byte[] b에 읽었다면 이 데이터를 하나의 파일로 저장하기 위해서 다음과 같이 FileOutputStream을 생성한 후 바로 기록
 - ❖ `FileOutputStream fos = new FileOutputStream("test.txt");`
 - ❖ `fos.write(b);`
 - ❖ `fos.close();`



9.4.4 문자 단위의 출력 파일 스트림

- ❖ 문자 스트림을 관리하는 최상위 스트림
 - ❖ `public abstract class Reader extends Object`
 - ❖ `public abstract class Writer extends Object`
- ❖ Reader와 Writer 클래스
 - ❖ **모든 문자 입력 스트림의 최상위 클래스**이다.
- ❖ **Reader, Writer** 클래스 VS InputStream, OutputStream 클래스의 차이
 - ❖ **Reader, Writer**
 - ❖ 수행하는 역할과 비슷하지만 바이트가 아닌 **문자들을 입출력 하는** 데 차이가 있다.



9.4.4 문자 단위의 출력 파일 스트림(계속)

- ❖ `FileWriter`를 생성하는 법
 - ❖ `FileWriter fw = new FileWriter("파일명");`
 - ❖ `//fw을 이용해서 문자 기록하기`
 - ❖ `fw.close(); //스트림 닫기`
- ❖ `FileWriter`의 `write()` 계열의 메서드
 - ❖ `void write(String str)`
 - ❖ `void write(String str, int off, int len)`
- ❖ `Writer`의 `writer()` 계열의 메서드
 - ❖ `void write(int c)`
 - ❖ `void write(char[] cbuf)`
 - ❖ `void write(char[] cbuf, int off, int len)`



9.4.4 문자 단위의 출력 파일 스트림(계속)

§9-16 `FileWriterMain.java`

```
01: /**
02:  * FileWriter로 문자 기록하기
03: */
04: import java.io.*;
05: public class FileWriterMain{
06:     public static void main(String[] args) throws IOException{
07:         FileWriter fos = new FileWriter("writer.dat");
08:         fos.write(72);
09:         fos.write(101);
10:         fos.write(108);
11:         fos.write(108);
12:         fos.write(111);
13:         fos.close();
14:         System.out.println("writer.dat 파일 기록 완료");
15:     } //end of main
16: } // end of FileWriterMain class
```

```
C:\javasrc\chap09>javac FileWriterMain.java
C:\javasrc\chap09>java FileWriterMain
writer.dat 파일 기록 완료
```

```
C:\javasrc\chap09>type writer.dat
Hello
```



9.4.4 문자 단위의 출력 파일 스트림(계속)

§ 9-17 FileWriterMain2.java

```
01: /**
02:  * FileWriter에 문자열과 문자배열 기록하기
03: */
04: import java.io.*;
05: public class FileWriterMain2{
06:     public static void main(String[] args) throws IOException{
07:         char[] content = new char[]{72, 101, 108, 108, 111};
08:         String str = new String("Hello World!");
09:         FileWriter fos = new FileWriter("writer2.dat");
10:         fos.write(content); //문자배열의 기록
11:         fos.write(str); //문자열의 기록
12:         fos.close();
13:         System.out.println("writer2.dat 파일 기록 완료");
14:     } //end of main
15: } // end of FileWriterMain2 class
```

```
C:\javasrc\chap09>javac FileWriterMain2.java
```

```
C:\javasrc\chap09>java FileWriterMain2
```

```
writer2.dat 파일 기록 완료
```

```
C:\javasrc\chap09>type writer2.dat
```

```
HelloHello World!
```



9.4.4 문자 단위의 출력 파일 스트림(계속)

- ❖ 데이터로 사용할 문자배열과 문자열
 - ❖ char[] content = new char[]{72, 101, 108, 108, 111}; //문자배열
 - ❖ String str = new String("Hello World!"); //문자열
- ❖ FileWriter에 데이터 기록하기
 - ❖ fos.write(content);
 - ❖ fos.write(str)
- ❖ 존재하는 파일에 덮어쓰기
 - ❖ FileWriter fw = new FileWriter("writer2.dat", false);
- ❖ 존재하는 파일의 마지막 부분에 덧붙여 넣기
 - ❖ FileWriter fw = new FileWriter("writer2.dat", true);



9.4.5 문자 단위의 입력 파일 스트림

- ❖ FileReader를 생성하는 방법
 - ❖ FileReader `fr = new FileReader("파일명");`
 - ❖ //fr을 이용해서 문자 읽기
 - ❖ `fr.close();`//스트림 닫기
- ❖ FileReader의 `read()` 계열의 메서드
 - ❖ `int read()`
 - ❖ 하나의 문자를 읽어내기 위한 메소드
 - ❖ `int read(char[] cbuf, int offset, int length)`
 - ❖ 배열의 특정 부분으로 읽어 들이고자 한다면 이 메소드를 사용하면 된다.

9.4.5 문자 단위의 입력 파일 스트림(계속)

§ 9-19 FileReaderMain.java

```
01:  /**
02:  * FileReader를 이용해서 파일 읽기
03:  */
04: import java.io.*;
05: public class FileReaderMain{
06:     public static void main(String[] args) throws IOException{
07:         FileReader fr = new FileReader("writer.dat");
08:         int i;
09:         while( (i=fr.read()) != -1 ){
10:             System.out.print((char)i);
11:         }
12:         fr.close();
13:     } //end of main
14: } // end of FileReaderMain class
```

```
C:\javasrc\chap09>javac FileReaderMain.java
```

```
C:\javasrc\chap09>java FileReaderMain
```

```
Hello
```

9-20 FileCopy.java

```
01:  /**
02:   * File 스트림을 이용한 파일 복사 예제(한 바이트 단위)
03:   */
04:  import java.io.*;
05:  public class FileCopy {
06:      //FileCopy 프로그램의 실행
07:      //java FileCopy 원본파일이름 목표파일이름
08:      //예) java FileCopy s.exe t.exe
09:      //s.exe는 같은 디렉토리 상에 존재해야 한다.
10:     public static void main(String[] args) throws IOException{
11:         int i, len=0;
12:         FileInputStream fis = new FileInputStream(args[0]);
13:         FileOutputStream fos = new FileOutputStream(args[1]);
14:         long psecond = System.currentTimeMillis();
15:         while((i=fis.read()) != -1){
16:             fos.write(i);
17:             len++;
18:         }
19:         fis.close();
20:         fos.close();
21:         psecond = System.currentTimeMillis() - psecond;
22:
23:         System.out.println(len + " bytes " + psecond + " miliseconds");
24:     } //end of main
25: } //end of FileCopy class
```

```
C:\javasrc\chap09>javac FileCopy.java
C:\javasrc\chap09>java FileCopy s.exe t.exe
6592232 bytes 41546 miliseconds
```

```
C:\javasrc\chap09>dir t.exe
오후 04:10           6,592,232 t.exe
```

- ❖ 테스트할 파일 복사 예제
 - ❖ File 스트림을 이용한 파일 복사
 - ❖ Bufferd 스트림을 이용한 파일 복사

9.4.5 문자 단위의 입력 파일 스트림(계속)

- ❖ 입력 스트림에서 한 바이트씩 읽고 출력 스트림으로 바로 내보내기
 - ❖ while((i=fis.read()) != -1){
 - ❖ fos.write(i);
 - ❖ }
- ❖ System.currentTimeMillis()
 - ❖ 1970년 1월 1일부터 계산된 1/1000초 단위의 값
- ❖ ■ 파일 복사에 걸린 시간 측정
 - ❖ long psecond = System.currentTimeMillis();
 - ❖ //작업
 - ❖ psecond = psecond - System.currentTimeMillis();

9.4.5 문자 단위의 입력 파일 스트림(계속)

❖ **BufferedOutputStream**

- ❖ 파일 스트림을 이용한 파일 복사의 경우 버퍼링 기능을 지원하지 않아 복사하는데 상당한 시간이 필요하다.
- ❖ **BufferedOutputStream**으로 변환해서 사용하면 된다.

❖ File 스트림을 Buffered 스트림으로 변환

- ❖ `FileInputStream fis = new FileInputStream("원본파일명");`
- ❖ `FileOutputStream fos = new FileOutputStream("복사파일명");`
- ❖ `BufferedInputStream bis = new BufferedInputStream(fis);`
- ❖ `BufferedOutputStream bos = new BufferedOutputStream(fos);`



§9-21 **BufferedFileCopy.java**

```
01: /**
02:  Buffered 스트림을 이용한 파일 복사 예제(한 바이트 단위)
03: */
04: import java.io.*;
05: public class BufferedFileCopy {
06:     //BufferedFileCopy 프로그램의 실행
07:     //java BufferedFileCopy 원본파일이름 목표파일이름
08:     //예) java BufferedFileCopy s.exe t.exe
09:     //s.exe는 같은 디렉토리 상에 존재해야 한다.
10:     public static void main(String[] args) throws IOException{
11:         int i, len=0;
12:         FileInputStream fis = new FileInputStream(args[0]);
13:         FileOutputStream fos = new FileOutputStream(args[1]);
14:         BufferedInputStream bis = new BufferedInputStream(fis);
15:         BufferedOutputStream bos = new BufferedOutputStream(fos);
16:         long psecond = System.currentTimeMillis();
17:         while((i=bis.read()) != -1){
18:             bos.write(i);
19:             len++;
20:         }
21:         bis.close();
22:         bos.close();
23:         psecond = System.currentTimeMillis() - psecond;
24:         System.out.println(len + " bytes " + psecond + " milliseconds");
25:     } //end of main
26: } //end of BufferedFileCopy class
```

```
C:\javasrc\chap09>javac BufferedFileCopy.java
C:\javasrc\chap09>java BufferedFileCopy s.exe t.exe
6592232 bytes 563 milliseconds
```

```
C:\javasrc\chap09>dir t.exe
오후 04:14           6,592,232 t.exe
```



❖ 스트림의 필수

- ❖ 데이터의 용량이 큰 경우 무조건 Buffered 스트림으로 변환해서 사용한다.





9.4.7 RandomAccessFile



- ❖ 임의의 접근(Random Access)
 - ❖ 스트림으로 사용할 때 순차적으로 read()을 호출하게 되며, reset()을 이용하면 처음부터 다시 읽을 수 있다. 하지만 사용자가 원하는 위치로 이동하거나 되돌릴 수는 없다. 그렇기 때문에 일반적인 스트림에서는 데이터를 랜덤하게 접근할 수 없다.
- ❖ RandomAccessFile의 생성과 옵셋(Offset- 위치) 이동
 - ❖ RandomAccessFile rf = new RandomAccessFile(" 파일명 ", " 모드 ");
 - ❖ rf.seek(10); //옵셋이 10인 위치로 이동



§9-22 RandomAccessFileMain.java

```
01: /**
02:  임의 접근방식으로 파일 접근-RandomAccessFile을 테스트하는 예제
03: */
04: import java.io.*;
05: public class RandomAccessFileMain{
06:     public static void main(String[] args) throws IOException{
07:         String s = "I love normal java";
08:         String q = "jabook";
09:         RandomAccessFile rf = new RandomAccessFile("raccess.txt", "rw");
10:         rf.writeBytes(s);
11:         rf.close();
12:         RandomAccessFile rf2 = new RandomAccessFile("raccess.txt", "rw");
13:         rf2.seek(7);
14:         rf2.writeBytes(q);
15:         rf2.close();
16:         RandomAccessFile rf3 = new RandomAccessFile("raccess.txt", "r");
17:         System.out.println(rf3.readLine());
18:         rf3.close();
19:     } //end of main
20: } //end of RandomAccessFileMain class
```

```
C:\javasrc\chap09>javac RandomAccessFileMain.java
C:\javasrc\chap09>java RandomAccessFileMain
I love jabook java
```



9.4.7 RandomAccessFile(계속)

- ❖ 읽기 또는 쓰기 속성으로 RandomAccessFile 생성
 - ❖ RandomAccessFile rf = new RandomAccessFile("raccess.txt", "rw");
 - ❖ rf.writeBytes(s); //데이터 기록
 - ❖ rf.close(); //RandomAccessFile 닫기
- ❖ rf2생성
 - ❖ RandomAccessFile rf2 = new RandomAccessFile("raccess.txt", "rw");
 - ❖ rf2.seek(7);
 - ❖ rf2.writeBytes(q);
 - ❖ rf2.close();
- ❖ seek() 메서드
 - ❖ 파일 포인터를 특정한 절대위치로 이동시킬 수 있다.
- ❖ rf생성
 - ❖ RandomAccessFile rf3 = new RandomAccessFile("raccess.txt", "r");
 - ❖ System.out.println("글내용은" + rf3.readLine());
 - ❖ rf3.close()

9.4.8 결론

- ❖ 자바의 파일 스트림
 - ❖ 바이트 : **FileInputStream, FileOutputStream**
 - ❖ 문자 : **FileReader, FileWriter**



8. Thread and Synchronization



INDEX



- 8.1 배경
- 8.2 쓰레드의 기본
- 8.3 쓰레드의 제어
- 8.4 동기화
- 8.5 마무리





8.1 배경



8.1.1 개요



8.1.1 개요



- ❖ 프로세스(Process)
 - ❖ 하나의 프로그램(Program)은 하나의 프로세스(Process)에 해당한다.
- ❖ 멀티 태스킹(Multi- Tasking)
 - ❖ 프로세스(Process)의 경우 운영체제에서 자동으로 관리해 준다.
 - ❖ 운영체제 차원의 프로세스(Process) 관리를 멀티 태스킹(Multi-Tasking)이라고 한다.





8.1.1 개요(계속)



- ❖ 프로세스(Process)와 스레드(thread)
 - ❖ 하나의 프로세스(Process) 내에는 여러 개의 스레드(Thread)가 존재할 있다.
- ❖ 스레드(Thread)란?
 - ❖ 하나의 프로그램 내에서 실행되는 메서드
 - ❖ 같은 순간에 두 개의 메서드가 동시에 실행되면 두 개의 스레드가 동작하는 것이다.
- ❖ 이장에서 공부할 것
 - ❖ 두 개의 메서드를 동시에 실행시키는 방법에 대해서 배우게 될 것이다.
 - ❖ 사실 하나의 메서드는 이미 실행된 상태이다.
 - ❖ main()도 하나의 메서드이기 때문이다.



8.1.1 개요(계속)



- ❖ 스레드 프로그램의 할 때 주의해야 할 사항
 - ❖ 우선권(Priority)
 - ❖ 여러 개의 메서드가 실행되기 때문에 어느 메서드에게 작업할 권한을 많이 줄 것인지
 - ❖ 동기화(Synchronization)
 - ❖ 공유자원을 상대로 순서대로 작업이 이루어지는 것을 ‘동기화(Synchronization)’가 보장된다’라고 한다.
 - ❖ ex) 은행의 입금 출금 문제





8.2 스레드의 기본



- 8.2.1 스레드란
- 8.2.2 Runnable로 스레드 만들기
- 8.2.3 Thread 상속으로 스레드 만들기
- 8.2.4 Runnable을 사용하는 이유
- 8.2.5 스레드의 상태
- 8.2.6 결론



8.2.1 스레드란



- ❖ 일반적인 메서드
 - ❖ main()을 실행시키면 순서대로 작업이 진행된다.
 - ❖ 한마디로 ‘시퀀셜(Sequential)’ 하다라는 의미이다.
 - ❖ 단 한 순간에 하나의 메서드만이 동작하는 것
- ❖ 우리의 목표
 - ❖ **두 개의 메서드를 동시에 실행하는 것**
- ❖ 스레드의 기본
 - ❖ **한 순간에 두 개의 메서드가 동시에 실행되었을 때, 실행된 메서드를 스레드(Thread)라고 한다.**





8.2.1 스레드란(계속)



- ❖ 생각
 - ❖ 어떻게 동시에 두 개의 메서드를 실행시키는가?
 - ❖ 프로그램 언어마다 라이브러리를 제공한다.
- ❖ 스레드란?
 - ❖ 프로그램에서 **독립적으로 실행되는 메서드**
- ❖ 스레드의 조건
 - ❖ 하나의 메서드가 실행되는 것도 하나의 스레드가 실행되는 것이지만, 진정한 스레드는 동시에 **두 개 이상의 메서드가 실행되는 것을 의미한다.**
 - ❖ 두 개의 메서드가 존재하고 **서로 독립된 메서드로써 동시에 작업을 진행할 수 있으면**, 이 때 이 메서드들을 스레드라고 부른다.



8.2.2 Runnable로 스레드 만들기

- ❖ 스레드란 메서드다.
- ❖ 스레드로 사용할 메서드를 만들어야 한다.
 - ❖ 스레드의 메서드 이름은 정해져 있다.
 - ❖ **public void run()**
- ❖ 스레드의 메서드를 구현하기 위한 인터페이스
 - ❖

```
public interface Runnable{  
    void run();  
}
```
- ❖ Runnable 인터페이스의 구현
 - ❖

```
public class Top implements Runnable{  
    public void run(){  
        //.....작업 내용  
    }  
}
```



2.2 Runnable로 스레드 만들기(계속)

- ❖ Runnable 인터페이스를 구현하고 있는 Top형의 객체 생성
 - ❖ Top t = new Top();
- ❖ 진짜 스레드에 Runnable 장착
 - ❖ Top t = new Top();
 - ❖ Thread thd = new Thread(t); // import java.lang.*; 자동 임포트

- ❖ 스레드 동작시키기
 - ❖ Top t = new Top(); // 가짜 스레드
 - ❖ Thread thd = new Thread(t); // 진짜 스레드
 - ❖ thd.start(); // 스레드 동작시키기



8-1 ThreadMain.java

```
l1:  /**
l2:  Runnable을 구현해서 만드는 스레드(스레드를 생성하는 방법)
l3: */
l4: class Top implements Runnable{
l5:     public void run(){
l6:         for(int i=0; i<50; i++)
l7:             System.out.print(i+"\t");
l8:     }
l9: } //end of Top class
l0:
l1: public class ThreadMain{
l2:     public static void main(String[] args){
l3:         System.out.println("프로그램 시작");
l4:         Top t = new Top();
l5:         Thread thd = new Thread(t);
l6:         thd.start();
l7:         System.out.println("프로그램 종료");
l8:     } //end of main
l9: } //end of ThreadMain class
```

```
C:\javasrc\chap08>javac ThreadMain.java
```

```
C:\javasrc\chap08>java ThreadMain
```

프로그램 시작

프로그램 종료

1	2	3	4	5	6	7	8	9
11	12	13	14	15	16	17	18	19
21	22	23	24	25	26	27	28	29
31	32	33	34	35	36	37	38	39
41	42	43	44	45	46	47	48	49



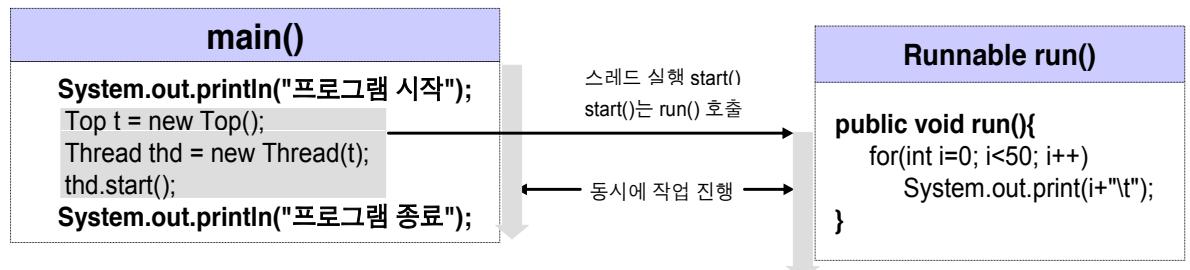
2.2 Runnable로 스레드 만들기(계속)

❖ 예제설명

- ❖ '프로그램 시작'과 '프로그램 종료'가 출력되고 난 뒤에 스레드의 작업이 진행된다.
- ❖ main()이 작업을 끝낸 후 동작중인 스레드가 작업을 끝낼 때까지 대기하기 때문이다. 이러한 원리는 다음과 같이 표현할 수 있다.

그림 8-1 스레드의 생성과 실행

스레드의 실행 순서



§8-2 ThreadMain2.java

```
01:  /**
02:  두 개의 스레드를 생성한 후 실행하는 예(Runnable 구현)
03:  */
04:  class Top implements Runnable{
05:      public void run(){
06:          for(int i=0; i<50; i++)
07:              System.out.print(i+"\t");
08:      }
09:  } //end of Top class
10:
11: public class ThreadMain2{
12:     public static void main(String[] args){
13:         System.out.println("프로그램 시작");
14:         //1. Runnable을 구현하는 객체 만들기
15:         Top t = new Top();
16:         //2. Runnable을 장착한 후 진짜 스레드 만들기
17:         Thread thd1 = new Thread(t);
18:         Thread thd2 = new Thread(t);
19:         //3. 스레드 동작 시키기
20:         thd1.start();
21:         thd2.start();
22:         System.out.println("프로그램 종료");
23:     } //end of main
24: } //end of ThreadMain2 class
```

C:\javasrc\chap08>javac ThreadMain2.java

C:\javasrc\chap08>java ThreadMain2

프로그램 시작

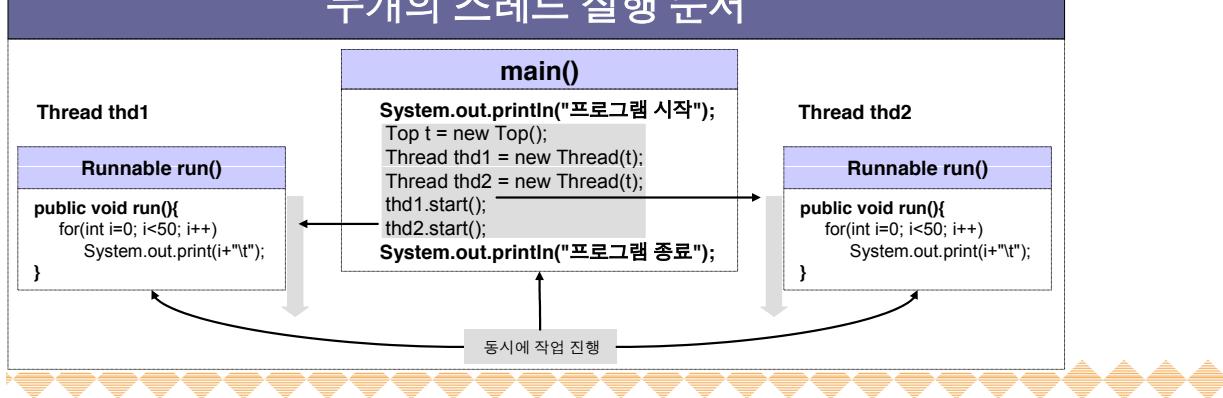
프로그램 종료

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	0	1	2	3
4	5	6	7	8	9	10	11	12	13
14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	16
17	18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45	46
47	48	49	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49



그림 8-2 다중 스레드의 생성과 실행

두개의 스레드 실행 순서



2.2 Runnable로 스레드 만들기 (계속)

- ❖ Top t = new Top();
- ❖ Thread thd1 = new Thread(t);
- ❖ Thread thd2 = new Thread(t);

- ❖ 참고
 - ❖ C나 C++에서는 스레드를 구현하기 위해서 함수 포인터(Function Pointer)라는 것을 이용한다.
 - ❖ 하지만 자바에서는 함수 포인터의 개념이 없기 때문에 Runnable 인터페이스를 이용하는 것이다.
 - ❖ 혹시 함수 포인터의 개념을 알고 있다면 Runnable 인터페이스가 하는 역할과 비교해 보라.



8.2.3 Thread 상속으로 스레드 만들기

- Thread를 상속받는 방법(상속을 통해서 스레드를 만드는 방법)

```
public class DerivedThread extends Thread{  
    public void run(){  
        //...스레드 내의 작업  
    }  
}
```

- DerivedThread d = new DerivedThread();

- d.start();

§8-3 DerivedThreadMain.java

```
01:  /**  
02:   Thread를 상속하는 스레드 만들기 (Thread 상속)  
03:  **/  
04:  class DerivedThread extends Thread{  
05:      public void run(){  
06:          for(int i=0; i<50; i++)  
07:              System.out.print(i+"\t");  
08:      }  
09:  } //end of DerivedThread class  
10:  
11: public class DerivedThreadMain{  
12:     public static void main(String[] args){  
13:         System.out.println("프로그램 시작");  
14:         DerivedThread d = new DerivedThread();  
15:         d.start();  
16:         System.out.println("프로그램 종료");  
17:     } //end of main  
18: } //end of DerivedThreadMain class
```

```
C:\javasrc\chap08>javac DerivedThreadMain.java
```

```
C:\javasrc\chap08>java DerivedThreadMain
```

```
프로그램 시작
```

```
프로그램 종료
```

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49



8-4 DerivedThreadMain2.java

```
01:  /**
02:   Thread를 상속하는 두 개의 스레드를 생성한 후 실행시키기 (Thread 상속)
03:  */
04:  class DerivedThread extends Thread{
05:      public void run(){
06:          for(int i=0; i<50; i++)
07:              System.out.print(i+"\t");
08:      }
09:  } //end of DerivedThread class
10:
11: public class DerivedThreadMain2{
12:     public static void main(String[] args){
13:         System.out.println("프로그램 시작");
14:         DerivedThread d1 = new DerivedThread();
15:         DerivedThread d2 = new DerivedThread();
16:         d1.start();
17:         d2.start();
18:         System.out.println("프로그램 종료");
19:     } //end of main
20: } //end of DerivedThreadMain2 class
```



```
C:\javasrc\chap08>javac DerivedThreadMain2.java
C:\javasrc\chap08>java DerivedThreadMain2
프로그램 시작
프로그램 종료
0      1      2      3      4      5      6      7      8      9
10     11     12     13     14     15     0      1      2      3
4      5      6      7      8      9      10     11     12     13
14     15     16     17     18     19     20     21     22     23
24     25     26     27     28     29     30     31     32     16
17     18     19     20     21     22     23     24     25     26
27     28     29     30     31     32     33     34     35     36
37     38     39     40     41     42     43     44     45     46
47     48     49     33     34     35     36     37     38     39
40     41     42     43     44     45     46     47     48     49
```



8.2.4 Runnable을 사용하는 이유

- ❖ extends Frame을 상속했기 때문에 Thread를 상속할 수 없는 경우

```
❖ class RunFrame extends Frame{  
❖     //...작업  
❖ }
```

- ❖ Frame은 상속을 하고 Runnable은 구현을 한 상태

```
❖ class RunFrame extends Frame implements Runnable{  
❖     public void run(){  
❖         //...스레드 내의 작업  
❖     }  
❖ }
```



8.2.4 Runnable을 사용하는 이유(사용)

- ❖ Runnable을 사용하는 이유

- ❖ Thread를 상속하기 이전에 다른 클래스를 상속한 경우 Runnable을 이용해서 스레드를 만든다.

- ❖ 창 띄우기와 스레드 동작 시키기

- ❖ RunFrame r = new RunFrame();
 - ❖ r.setSize(300, 100);
 - ❖ r.show();
 - ❖ Thread t = new Thread(r);
 - ❖ t.start();

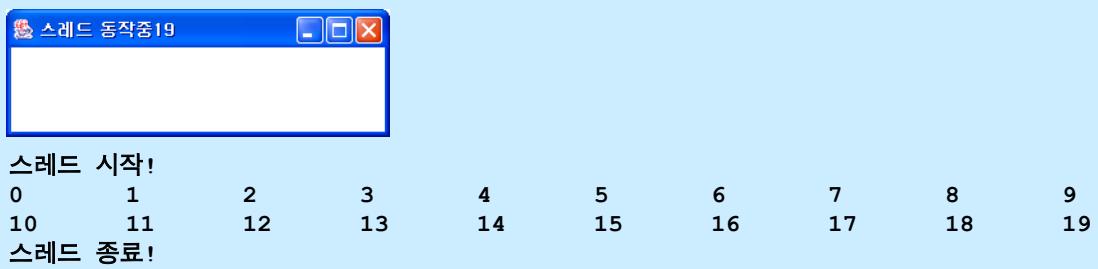


8-5 RunFrameMain.java

```
01:  /**
02:   Frame의 상속과 Runnable의 구현
03:   Thread를 상속하지 못하는 경우 Runnable로 구현
04: */
05: import java.awt.*;
06: class RunFrame extends Frame implements Runnable {
07:     public void run(){
08:         int i = 0;
09:         System.out.println("스레드 시작!");
10:         while(i<20){
11:             System.out.print(i + "\t");
12:             this.setTitle("스레드 동작중" + i++);
13:             try{
14:                 Thread.sleep(300);
15:             }catch(InterruptedException e){System.out.println(e);}
16:         }
17:         System.out.println("스레드 종료!");
18:     }
19: } //end of RunFrame class
20:
21: public class RunFrameMain{
22:     public static void main(String args[]){
23:         RunFrame r = new RunFrame();
24:         r.setSize(300, 100);
25:         r.show();
26:         Thread t = new Thread(r);
27:         t.start();
28:     } //end of main
29: } //end of RunFrameMain class
```

```
C:\javasrc\chap08>javac RunFrameMain.java
C:\javasrc\chap08>java RunFrameMain
```

그림 8-3



◆ 제어권

- ❖ 만약 위의 예제를 스레드로 구현하지 않고 일반 메서드를 호출하는 방식으로 구현한다면 while문이 끝날 때까지 Frame의 다른 작업을 할 수가 없다.
- ❖ 스레드로 구현하지 않으면 while문이 끝나야만 다른 작업을 할 수 있다.

2.4 Runnable을 사용하는 이유(사용)

- ❖ RunFrame으로 가능한 업캐스팅
 - ❖ Frame f = new RunFrame(); //Frame으로의 캐스팅
 - ❖ Runnable r = new RunFrame()://Runnable으로의 캐스팅
- ❖ RunFrame의 할일
 - ❖ 창만들고 띄우기
 - ❖ 스레드 동작 시키기

■창 띄우기

```
◆RunFrame = new RunFrame();
◆r.setSize(300,300);
◆r.show();
```

■스레드 동작시키기

```
◆Thread t = new Thread(r);
◆t.start();
```



2.4 Runnable을 사용하는 이유(사용)

- ❖ RunFrame r을 Thread의 매개변수로 줄 수 있는 이유
 - ❖ RunFrame이 Runnable로 업캐스팅이 가능하기 때문
- ❖ Thread 클래스의 생성자
 - ❖ public Thread(Runable target)
 - ❖ 여기서 Runnable을 매개변수로 주어야 하지만 RunFrame을 매개변수로 주어도 상관없다
 - ❖ 그것은 매개변수로 삽입될 때 다음과 같은 업캐스팅 현상이 일어나기 때문이다.
 - ❖ Thread 객체를 생성할 때 매개변수의 업캐스팅
 - ❖ Runable target = r; //은 RunFrame의 객체

■스레드 동작시키기

```
◆Thread t = new Thread(r);
◆t.start();
```



2.4 Runnable을 사용하는 이유(사용)

- ❖ 클래스 내부에서 스레드 구동 시키기

```
❖ class RunnableFrame extends Frame implements Runnable {  
❖     public RunnableFrame(){  
❖         new Thread(this).start(); //Runnable을 이용한 스레드의 구동  
❖     }  
❖     //생략...  
❖ }
```

- ❖ 이와 같은 구문이 가능한 이유 또한 업캐스팅 때문

- ❖ this는 언젠가 생성될 메모리를 의미
- ❖ this 자체는 Runnable로 캐스팅될 수 있기 때문에 위와 같은 구문이 가능하다.



2.4 Runnable을 사용하는 이유(사용)

§8-6 RunnableFrameMain.java

```
01:  /**  
02:   Frame 내부에서 스레드 동작 시키기  
03: */  
04: import java.awt.*;  
05: class RunnableFrame extends Frame implements Runnable {  
06:     public RunnableFrame(){  
07:         new Thread(this).start();  
08:     }  
09:     public void run(){  
10:         int i = 0;  
11:         System.out.println("스레드 시작!");  
12:         while(i<20){  
13:             System.out.print(i + "\t");  
14:             this.setTitle("스레드 동작중" + i++);  
15:             try{  
16:                 Thread.sleep(300);  
17:             }catch(InterruptedException e){System.out.println(e);}  
18:         }  
19:         System.out.println("스레드 종료!");  
20:     }  
21: } //end of RunnableFrame class
```



```

23: public class RunnableFrameMain{
24:     public static void main(String args[]){
25:         RunnableFrame r = new RunnableFrame();
26:         r.setSize(300, 100);
27:         r.show();
28:     } //end of main
29: } //end of RunnableFrameMain class

```

C:\javasrc\chap08>javac RunnableFrameMain.java

C:\javasrc\chap08>java RunnableFrameMain

그림 8-4



스레드 시작!

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19

스레드 종료!

- ❖ RunnableFrame 클래스 내에서 스레드 동작 시키기
 - ❖ new Thread(this).start(); //this는 RunnableFrame형이다.



§8-7 SoloFrameMain.java

```

01: /**
02:  Frame과 Thread 분리시켜서 구현
03: */
04: import java.awt.*;
05: class SoloFrame extends Frame{
06:     public SoloFrame(){
07:         SoloThread t = new SoloThread(this);
08:         t.start();
09:     }
10: } //end of SoloFrame class
11:
12: class SoloThread extends Thread{
13:     private Frame f = null;
14:     public SoloThread(Frame f){
15:         this.f = f; //SoloFrame의 참조값 챙겨두기
16:     }
17:     public void run(){
18:         int i = 0;
19:         System.out.println("스레드 시작!");
20:         while(i<20){
21:             System.out.print(i + "\t");
22:             f.setTitle("스레드 동작중" + i++);
23:             try{
24:                 this.sleep(300);
25:             }catch(InterruptedException e){System.out.println(e);}
26:         }
27:         System.out.println("스레드 종료!");
28:     }
29: } //end of SoloThread class

```



```
31: public class SoloFrameMain{
32:     public static void main(String args[]){
33:         SoloFrame s = new SoloFrame();
34:         s.setSize(300, 100);
35:         s.show();
36:     } //end of main
37: } //end of SoloFrameMain class
```

```
C:\javasrc\chap08>javac SoloFrameMain.java
C:\javasrc\chap08>java SoloFrameMain
```

그림 8-5



스레드 시작!

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19

스레드 종료!

- ❖ 프레임과 스레드를 분리시켜서 구현
 - ❖ class SoloFrame extends Frame{...}
 - ❖ class SoloThread extends Thread{...}



2.4 Runnable을 사용하는 이유(사용)

- ❖ 스레드에 프레임의 참조값 설정
 - ❖ private Frame f = null;
 - ❖ public SoloThread(Frame f){
 - ❖ this.f = f; //참조값복사
 - ❖ }
- ❖ sleep() 메서드
 - ❖ sleep() 메서드는 현재의 스레드를 일정시간 동안 대기상태로 만든다.
- ❖ SoloThread 시작
 - ❖ SoloThread t = new SoloThread(this); //this는 SoloFrame형
 - ❖ t.start();





8.2.5 스레드의 상태



- ❖ 스레드의 제어
 - ❖ 스레드를 만들고 시작시키는 것보다 **스레드를 제어하는 것**이 더 중요하다.
- ❖ 스레드의 상태
 - ❖ 시작 상태(**start** 상태)
 - ❖ 동작할 수 있는 상태(**Runnable** 상태)
 - ❖ 동작 상태(**Run** 상태)
 - ❖ 한 순간에 단 하나의 스레드만이 Run 상태가 된다.
 - ❖ 대기 상태(**NotRunnable** 상태)
 - ❖ 종료 상태(**Dead** 상태)



8.2.5 스레드의 상태(계속)



- ❖ Runnable 상태
 - ❖ **스레드가 실행되면** 일반적으로 Runnable 상태가 된다.
 - ❖ Runnable 상태의 스레드는 여러 개 존재할 수 있다.
- ❖ Run 상태
 - ❖ Runnable 상태에서만 Run 상태가 될 수 있다.
 - ❖ **한 순간 단 하나의 스레드만이 Run 상태가 된다.**
- ❖ Runnable과 Run의 상태 변화
 - ❖ Runnable 상태에 있는 **스레드들끼리 번갈아 가면서** Run 상태가 된다.
 - ❖ Run 상태가 될 때 **스레드는 작업을 진행할 수 있다.**
- ❖ Dead 상태(스레드의 종료)
 - ❖ run() 메서드의 **종료는 스레드의 종료를** 의미한다.
- ❖ NotRunnable 상태
 - ❖ NotRunnable 상태는 Run 상태로 진입할 수는 없지만 Dead 상태는 아닌 대기 상태

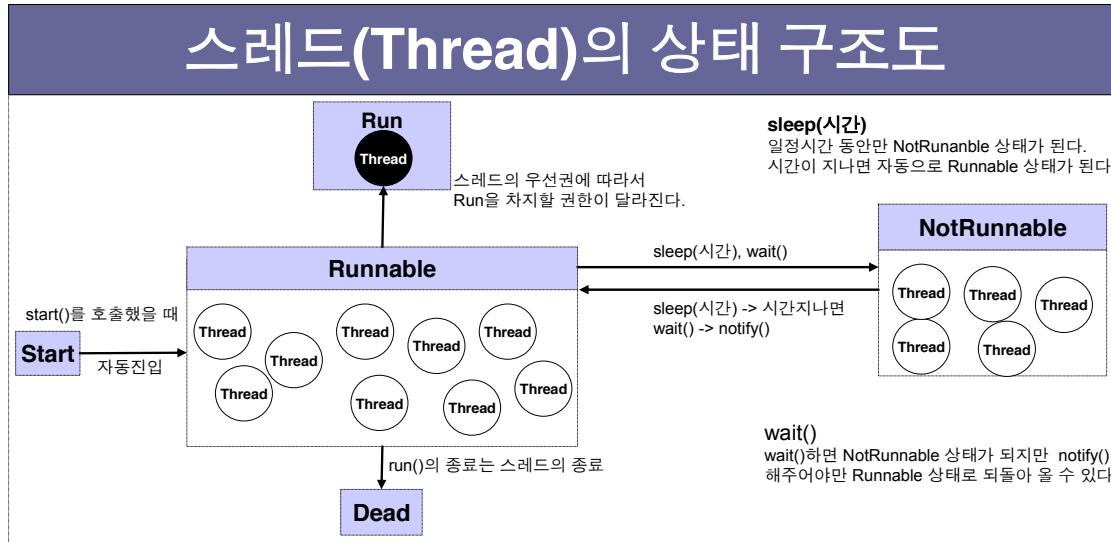




8.2.5 스레드의 상태(계속)



그림 8-6 스레드의 상태



8.2.5 스레드의 상태(계속)



- ❖ 5초 동안 NotRunnable 상태 만들기
 - ❖ `Thread.sleep(5000);`
- ❖ `sleep()` 메서드
 - ❖ `Thread`의 스태틱 메서드이기 때문에 **프로그램 어디에서나** 사용 가능
- ❖ NotRunnable 상태를 만드는 방법
 - ❖ `sleep()`은 **일정 시간 동안만 NotRunnable 상태**로 만든다.
 - ❖ `wait()`와 `notify()`는 **수동으로 NotRunnable 상태로** 들어가고 나오는 것 을 제어할 수 있다.





8.3 스레드의 제어



- 8.3.1 스레드의 우선권
- 8.3.2 NotRunnable 상태 만들기
- 8.3.3 스레드 죽이기
- 8.3.4 스레드의 Resume, Suspend
- 8.3.5 결론



8.3.1 스레드의 우선권



- ❖ Thread 클래스의 스태틱 우선권 상수
 - ❖ public static final int MIN_PRIORITY = 1;
 - ❖ public static final int NORM_PRIORITY = 5;
 - ❖ public static final int MAX_PRIORITY = 10;
- ❖ 우선권 문제
 - ❖ 어떠한 스레드가 **Run 상태를 많이** 차지할 것인가의 문제이다





8.3.1 스레드의 우선권(계속)



- ❖ 스레드의 상태 설정하기
 - ❖ PriorityThread t = new PriorityThread();
 - ❖ t.setPriority(1);
 - ❖ //t.setPriority(Thread.MIN_PRIORITY);
 - ❖ //우선권이 가장 낮은 상태
- ❖ ◆ t.setPriority(5);
 - ❖ //t.setPriority(Thread.NORM_PRIORITY);
 - ❖ //일반적인 스레드가 갖는 우선권
- ❖ ◆ t.setPriority(10);
 - ❖ //t.setPriority(Thread.MAX_PRIORITY);
 - ❖ //우선권이 가장 높은 상태



8.3.1 스레드의 우선권(계속)



- ❖ 스레드에 설정된 우선권 얻어내기
 - ❖ int p = t.getPriority()
- ❖ 우선권(Priority)
 - ❖ 스레드의 우선권을 설정하기 위해서 **setPriority()** 메서드를 사용하고, 현재 설정되어 있는 우선권을 얻어내기 위해서 **getPriority()**를 사용한다.



§8-8 PriorityThreadMain.java

```
01:  /**
02:   스레드의 우선권을 테스트하는 예제
03:  */
04:  class PriorityThread extends Thread {
05:      public void run(){
06:          int i = 0;
07:          System.out.print(this.getName()); //스레드의 이름 출력
08:          System.out.println("[우선권:" + this.getPriority() + "] 시작\t");
09:          while(i < 10000){
10:              i = i + 1;
11:              try{
12:                  this.sleep(1);
13:              }catch(Exception e){System.out.println(e);}
14:          }
15:          System.out.print(this.getName()); //스레드의 이름 출력
16:          System.out.println("[우선권:" + this.getPriority() + "] 종료\t");
17:      }
18:  } //end of PriorityThread class
19:
20: public class PriorityThreadMain {
21:     public static void main(String[] args){
22:         System.out.println("Main메서드 시작");
23:         for(int i=1; i<=10; i++){
24:             //for(int i=Thread.MIN_PRIORITY; i<=Thread.MAX_PRIORITY; i++) {
25:                 PriorityThread s = new PriorityThread();
26:                 s.setPriority(i);
27:                 s.start();
28:             }
29:             System.out.println("Main메서드종료");
30:         } //end of main
31:     } //end of PriorityThreadMain class
```

C:\javasrc\chap08>javac PriorityThreadMain.java
C:\javasrc\chap08>java PriorityThreadMain

Main메서드 시작
Thread- 6[우선권:6] 시작
Thread- 7[우선권:7] 시작
Thread- 8[우선권:8] 시작
Thread- 9[우선권:9] 시작
Thread- 5[우선권:5] 시작
Thread- 10[우선권:10] 시작
Main메서드종료
Thread- 3[우선권:3] 시작
Thread- 4[우선권:4] 시작
Thread- 1[우선권:1] 시작
Thread- 2[우선권:2] 시작
Thread- 10[우선권:10] 종료
Thread- 8[우선권:8] 종료
Thread- 9[우선권:9] 종료
Thread- 6[우선권:6] 종료
Thread- 7[우선권:7] 종료
Thread- 5[우선권:5] 종료
Thread- 3[우선권:3] 종료
Thread- 4[우선권:4] 종료
Thread- 1[우선권:1] 종료
Thread- 2[우선권:2] 종료

8.3.2 NotRunnable 상태 만들기

❖ 스레드를 NotRunnable 상태로 만드는 방법

- ❖ sleep()을 이용해서 일정시간 동안만 대기시키는 방법(자동)
 - ❖ 주어진 시간만큼만 NotRunnable 상태로 보내진다.
 - ❖ 시간이 지나면 자동으로 Runnable 상태로 복귀
- ❖ wait()와 notify()를 이용해서 대기와 복귀를 제어하는 방법(수동)
 - ❖ 사용자가 직접 wait() 호출해서 NotRunnable 상태로 만든다.
 - ❖ notify()를 호출해 주어야만 Runnable 상태로 복귀

❖ sleep()의 사용

- ❖ try{
 - ❖ Thread.sleep(1000); //시간의 단위는 1/1000초
- ❖ }catch(InterruptedException e){e.printStackTrace();}



8.3.2 NotRunnable 상태 만들기

§8-9 NotRunnableMain.java

```
01:  /**
02:   sleep()을 이용한 작업의 일시 중단 - main()에서의 Thread.sleep()
03:  */
04:public class NotRunnableMain{
05: public static void main(String[] args){
06: long current = System.currentTimeMillis();
07: System.out.println("프로그램 시작");
08: try{
09:     Thread.sleep(5000);
10: }catch(InterruptedException e){e.printStackTrace();}
11: System.out.println("프로그램 종료");
12: System.out.println("시간: " + (System.currentTimeMillis()-current));
13: } //end of main
14: } //end of NotRunnableMain
```

```
C:\javasrc\chap08>javac NotRunnableMain.java
C:\javasrc\chap08>java NotRunnableMain
프로그램 시작
프로그램 종료
시간: 5000
```



§8-10 NotRunnableThreadMain.java

```
01:  /**
02:   * 스레드에서 sleep()의 사용
03:  */
04:  import java.util.*;
05:  class NotRunnableThread extends Thread {
06:      public void run(){
07:          int i = 0;
08:          while(i < 10){
09:              System.out.println(i + "회:" + System.currentTimeMillis() + "\t");
10:              i = i + 1;
11:              try{
12:                  this.sleep(1000);
13:              }catch(Exception e){System.out.println(e);}
14:          }
15:      }
16:  } //end of NotRunnableThread class
17:
18: public class NotRunnableThreadMain {
19:     public static void main(String args[] ) {
20:         NotRunnableThread s = new NotRunnableThread();
21:         s.start();
22:     } //end of main
23: } //end of NotRunnableThreadMain class
```



```
C:\javasrc\chap08>javac NotRunnableThreadMain.java
C:\javasrc\chap08>java NotRunnableThreadMain
0회:1087996213815
1회:1087996214815
2회:1087996215815
3회:1087996216815
4회:1087996217815
5회:1087996218815
6회:1087996219815
7회:1087996220815
8회:1087996221815
9회:1087996222815
```

- ❖ Thread를 상속한 경우 sleep()의 사용
 - ❖ `this.sleep(1000);`
 - ❖ 그렇지 않으면 Thread.sleep(1000)을 사용해야 한다.
- ❖ public static void sleep (long millis) throws InterruptedException
 - ❖ millis 시간만큼 작업을 멈추게 되며, 시간이 경과했을 때 다시 작업을 재개하게 된다.
 - ❖ millis는 1/1000초 단위를 사용한다.





8.3.3 스레드 죽이기



- ❖ 스레드의 종료
 - ❖ **run()** 메서드의 종료는 스레드의 종료를 의미한다.
 - ❖ 일반적으로 지속적인 작업을 하기 위해 run() 내에 while문을 포함하고 있으며, 이 while문이 끝나면 스레드가 종료되는 경우가 많다.
- ❖ Deprecated된 Thread의 stop() 메서드
 - ❖ public final void stop() Deprecated
 - ❖ public final void stop(Throwable obj) Deprecated
- ❖ Deprecated된 메서드
 - ❖ Deprecated된 메서드는 안전하지 못하며 권장하지 않는 구버전의 메서드라는 의미
 - ❖ Java Doc API를 보면 Deprecated된 메서드는 짙은 글씨체로 정확하게 표시하고 있다.



8.3.3 스레드 죽이기(계속)



- ❖ java에서 일반적인 스레드 죽이는 법
 - ❖ while 문의 조건을 제어함으로써 run()을 빠져 나오게 하는 방법을 주로 사용한다.
- ❖ run() 메서드의 일반적인 모델
 - ❖ while문의 조건에 따라 run()의 종료를 제어
 - ❖ public void run(){
 while(조건){
 //작업
 }
 }



08-11 TerminateThreadMain.java

```
01:  /**
02:   while문의 조건을 이용한 스레드의 종료를 테스트하는 예
03:  */
04:  class TerminateThread extends Thread {
05:      //스레드의 종료를 제어하는 플래그
06:      private boolean flag = false;
07:      public void run(){
08:          int count = 0;
09:          System.out.println(this.getName() +"시작");
10:          while(!flag){
11:              try {
12:                  //작업
13:                  this.sleep(100);
14:              } catch(InterruptedException e) { }
15:          }
16:          System.out.println(this.getName() +"종료");
17:      }
18:      public void setFlag(boolean flag){
19:          this.flag = flag;
20:      }
21:  } //end of TerminateThread class
22:
23: public class TerminateThreadMain {
24:     public static void main(String args[])throws Exception{
25:         System.out.println("작업시작");
26:         TerminateThread a = new TerminateThread();
27:         TerminateThread b = new TerminateThread();
28:         TerminateThread c = new TerminateThread();
29:         a.start();
30:         b.start();
31:         c.start();
32:         int i;
33:         System.out.print("종료할 스레드를 입력하시오! A, B, C, M?\n");
```



```
34:         while(true){
35:             i = System.in.read();
36:             if(i == 'A'){
37:                 a.setFlag(true);
38:             }else if(i == 'B'){
39:                 b.setFlag(true);
40:             }else if(i == 'C'){
41:                 c.setFlag(true);
42:             }else if(i == 'M'){
43:                 a.setFlag(true);
44:                 b.setFlag(true);
45:                 c.setFlag(true);
46:                 System.out.println("main종료");
47:                 break;
48:             }
49:         }
50:     } //end of main
51: } //end of TerminateThreadMain class
```

```
C:\javasrc\chap08>javac TerminateThreadMain.java
C:\javasrc\chap08>java TerminateThreadMain
```

작업시작

종료할 스레드를 입력하시오! A, B, C, M?

Thread-1시작

Thread-2시작

Thread-3시작

A

Thread-1종료

B

Thread-2종료

C

Thread-3종료

M

main종료

```
C:\javasrc\chap08>java TerminateThreadMain
```

작업시작

종료할 스레드를 입력하시오! A, B, C, M?

Thread-1시작

Thread-2시작

Thread-3시작

M

main종료

Thread-1종료

Thread-2종료

Thread-3종료





8.3.3 스레드 죽이기(계속)



- ❖ 스레드 제어를 위한 flag 설정하기
 - ❖ private boolean flag = false;
 - ❖ //flag가 true로 설정되면 while문이 끝난다.
 - ❖ public void setFlag(boolean flag){
 - ❖ this.flag = flag;
 - ❖ }
- ❖ flag를 이용한 run() 내의 while문 제어
 - ❖ while(!flag){ // 그리고 스레드를 제어하기 위해서 while문의 조건에 flag를 이용하고 있습니다.
 - ❖ //작업
 - ❖ }
- ❖ 스레드 Stop
 - ❖ flag가 true가 된다
 - ❖ 그러면 while문은 동작을 멈추게 되며 run()을 빠져 나오게 된다.



30-12 ControlThread.java

```
01:  /**
02:  두 개의 조건을 이용한 스레드의 종료
03: */
04: class ControlThread extends Thread {
05:     //모든 스레드의 종료를 제어하는 플래그
06:     public static boolean all_exit = false;
07:     //스레드의 종료를 제어하는 플래그
08:     private boolean flag = false;
09:     public void run(){
10:         int count = 0;
11:         System.out.println(this.getName() +"시작");
12:         //flag나 all_exit 둘 중 하나만 true이면 while문이 끝난다.
13:         while(!flag && !all_exit){
14:             try {
15:                 //작업
16:                 this.sleep(100);
17:             } catch(InterruptedException e) { }
18:         }
19:         System.out.println(this.getName() +"종료");
20:     }
21:     public void setFlag(boolean flag){
22:         this.flag = flag;
23:     }
24: } //end of ControlThread class
```



```

20:  public class ControlThreadMain {
21:      public static void main(String args[]) throws Exception{
22:          System.out.println("작업시작");
23:          ControlThread a = new ControlThread();
24:          ControlThread b = new ControlThread();
25:          ControlThread c = new ControlThread();
26:          a.start();
27:          b.start();
28:          c.start();
29:          Thread.sleep(100);
30:          int i;
31:          System.out.print("종료할 스레드를 입력하시오! A, B, C, M?\n");
32:          while(true){
33:              i = System.in.read(); //문자를 입력받기 위해서.
34:              if(i == 'A'){
35:                  a.setFlag(true);
36:              }else if(i == 'B'){
37:                  b.setFlag(true);
38:              }else if(i == 'C'){
39:                  c.setFlag(true);
40:              }else if(i == 'M'){
41:                  //모든 스레드를 종료시킨다.
42:                  ControlThread.all_exit = true;
43:                  System.out.println("main종료");
44:                  break;
45:              }
46:          }
47:      }
48:  } //end of main
49: } //end of ControlThreadMain class

```



```

C:\javasrc\chap08>javac ControlThreadMain.java
C:\javasrc\chap08>java ControlThreadMain
작업시작

```



종료할 스레드를 입력하시오! A, B, C, M?

Thread-1시작	
Thread-2시작	
Thread-3시작	
A	
Thread-1종료	C:\javasrc\chap08>java ControlThreadMain
B	작업시작
Thread-2종료	종료할 스레드를 입력하시오! A, B, C, M?
C	Thread-1시작
Thread-3종료	Thread-2시작
M	Thread-3시작
main종료	M //스레드 한번에 제어

```

C:\javasrc\chap08>java ControlThreadMain
작업시작
종료할 스레드를 입력하시오! A, B, C, M?
Thread-1시작
Thread-2시작
Thread-3시작
M //스레드 한번에 제어
main종료
Thread-1종료
Thread-2종료
Thread-3종료

```



8.3.4 스레드의 Resume, Suspend

- ❖ public final void suspend() Deprecated
 - ❖ 스레드의 작업을 대기시킨다.
- ❖ public final void resume() Deprecated
 - ❖ 스레드의 대기 상태를 해제하고 작업을 재개한다.
- ❖ Deprecated언급
 - ❖ 더 이상 **사용하지 않는 메서드를** 의미합니다.

8.3.5 결론

- ❖ 스레드의 제어를 위한 도구
 - ❖ setPriority()
 - ❖ setPriority()는 스레드가 Run 상태에 들어갈 수 있는 우선권을 결정하게 된다.
 - ❖ sleep()
 - ❖ sleep()은 일정 시간 동안 작업을 멈추게 하는 기능이 있다.
 - ❖ wait()
 - ❖ wait()는 스레드를 대기상태(NotRunnable)로 보내게 된다.
 - ❖ notify()
 - ❖ notify()는 대기상태에 있는 스레드를 Runnable 상태로 복귀시켜서 작업을 재개하게 한다.



8.4 동기화



8.4.1 멀티스레드와 문제점

8.4.2 공유자원의 접근

8.4.3 synchronized

8.4.4 synchronized의 활용

8.4.5 synchronized의 한계



8.4.1 멀티스레드와 문제점



- ❖ A, B, C 세 사람이 있다.
 - ❖ 화장실이 3개 있다.
 - ❖ 문제될 것이 없다.
 - ❖ 화장실이 하나 (공유 자원(Shared Resource)) 밖에 없다.
 - ❖ 한 사람이 쓰면, 다른 사람들은 화장실을 쓰면 안 된다.
 - ❖ 대기해야 한다.
- ❖ 동기화(Synchronization)의 정의
 - ❖ **줄서기**(번갈아 가면서 순서대로 공유자원 사용하기)
- ❖ 동기화(Synchronization)의 기법
 - ❖ synchronization 블록(자원을 사용할 때 자원에 락(Lock)을 거는 방식)
 - ❖ **wait()와 notify()**





8.4.2 공유자원의 접근



- ❖ 공유자원이 문제가 되는 이유
 - ❖ 동시에 작업을 진행하는 스레드의 특성 때문
- ❖ Bank 클래스
 - ❖ class Bank{
 - ❖ //...
 - ❖ }
- ❖ NotSynMain 클래스의 스태틱 멤버
 - ❖ class NotSynMain{
 - ❖ public static Bank MyBank = new Bank();
 - ❖ //....작업
 - ❖ }



8.4.2 공유자원의 접근(계속)



■ Park 스레드 클래스

```
◆ class Park extends Thread{  
    ◆ public void run(){  
        ◆     // NotSyncMain.myBank 사용  
    ◆ }  
    ◆ }  
}
```

■ ParkWife 스레드 클래스

```
◆ class ParkWife extends Thread{  
    ◆ public void run(){  
        ◆     // NotSyncMain.myBank 사용  
    ◆ }  
    ◆ }  
}
```

- ❖ 두 개의 스레드가 다음과 같이 동시에 실행된다면 공유자원의 문제가 발생
- ❖ 스레드의 생성 및 실행
 - ❖ Park p = new Park(); //Park 스레드 생성
 - ❖ ParkWife w = new ParkWife(); //ParkWife 스레드 생성
 - ❖ p.start(); //스레드 시작
 - ❖ w.start(); //스레드 시작



§8-13 NotSyncMain.java

```
01:  /**
02:   * 동기화의 문제 발생
03:  */
04:  class Bank{
05:      private int money = 10000; //예금 잔액
06:      public int getMoney(){
07:          return this.money;
08:      }
09:      public void setMoney(int money){
10:          this.money = money;
11:      }
12:      public void saveMoney(int save){
13:          int m = this.getMoney();
14:          try{
15:              Thread.sleep(3000);
16:          }catch(InterruptedException e){e.printStackTrace();}
17:          this.setMoney(m + save);
18:      }
19:      public void minusMoney(int minus){
20:          int m = this.money;
21:          try{
22:              Thread.sleep(200);
23:          }catch(InterruptedException e){e.printStackTrace();}
24:          this.setMoney(m - minus);
25:      }
26:  } //end of Bank class
```



```
28: class Park extends Thread{
29:     public void run(){
30:         NotSyncMain.myBank.saveMoney(3000);
31:         System.out.println("saveMoney(3000) :" + NotSyncMain.myBank.getMoney());
32:     }
33: } //end of Park class
34:
35: class ParkWife extends Thread{
36:     public void run(){
37:         NotSyncMain.myBank.minusMoney(1000);
38:         System.out.println("minusMoney(1000) :" + NotSyncMain.myBank.getMoney());
39:     }
40: } //end of ParkWife class
41:
42: public class NotSyncMain{
43:     public static Bank myBank = new Bank();
44:     public static void main(String[] args) throws Exception{
45:         System.out.println("원금 :" + myBank.getMoney());
46:         Park p = new Park();
47:         ParkWife w = new ParkWife();
48:         p.start();
49:         try{
50:             Thread.sleep(200);
51:         }catch(InterruptedException e){e.printStackTrace();} 62
52:         w.start();
53:     } //end of main
54: } //end of NotSyncMain class
```

C:\javasrc\chap08>javac NotSyncMain.java
C:\javasrc\chap08>java NotSyncMain

원금:10000
minusMoney(1000) :9000
saveMoney(3000) :13000 //잘못 나왔다. 12000이 나와야 맞는것이다. 공유자원 실패



8.4.2 공유자원의 접근(계속)



- ❖ 공유자원 Bank myBank의 이용
 - ❖ NotSyncMain.myBank.saveMoney(3000); //3000원 입금
 - ❖ NotSyncMain.myBank.minusMoney(1000); //1000원 출금
- ❖ 동기화가 **보장된 상태에서의** 입출금 금액
 - ❖ 12000(결과금액) = 10000(원금) + 3000(입금) - 1000(출금)
- ❖ 하지만 위의 **예제의 실행결과는 13000원이** 출력된다.
 - ❖ 그 이유는 Park의 run()에서 예금된 금액을 가져온 후 3초의 시간이 지연된 후 입금이 처리되기 때문
 - ❖ public void saveMoney(int save){
 - ❖ int m = this.getMoney(); //예금되어 있는 금액 확인
 - ❖ try{
 - ❖ Thread.sleep(3000);
 - ❖ }catch(InterruptedException e){e.printStackTrace();}
 - ❖ this.setMoney(m + save); //입금처리
 - ❖ }



8.4.2 공유자원의 접근(계속)



- ❖ 동기화가 유지되지 않은 경우 입출금의 계산 절차
 - ❖ Park이 10000을 읽어감
 - ❖ Park는 3초 대기
 - ❖ Park이 대기하는 동안 ParkWife 또한 10000을 읽어감
 - ❖ Park이 대기하는 동안 ParkWife 0.2초 대기
 - ❖ Park이 대기하는 동안 ParkWife는 1000원을 출금
 - ❖ Park이 대기하는 동안 ParkWife는 작업 완료(남은 돈은 9000원)
 - ❖ Park은 3초 대기한 후 읽어온 10000으로 3000원 입금
 - ❖ 결과는 13000원
- ❖ 동기화의 유지
 - ❖ Park이 Bank myBank를 사용할 때 ParkWife는 Bank myBank를 사용하기 위해서 **대기해야 한다.**





8.4.3 synchronized



- ❖ synchronized
 - ❖ 메서드와 블록 형태로 사용할 수 있다.
 - ❖ synchronized 메서드로 사용될 경우 해당 메서드 내에서 사용되는 모든 멤버 변수들은 락(Lock)이 걸리게 된다.
- ❖ synchronized 메서드의 예
 - ❖ public synchronized void saveMoney(int save){
 //....공유자원 - 멤버의 사용
}
- ❖ synchronized 블록의 예
 - ❖ public void saveMoney(int save){
 synchronized(this){
 //....공유자원 - 멤버의 사용
 }
}

 //synchronized 메서드나 블록 내에서 사용되는 공유자원은 무조건 동기화가 보장된다.



§8-14 SyncMain.java

```
01:  /**
02:  synchronized를 이용한 동기화의 보장
03:  */
04:  class Bank{
05:      private int money = 10000; //예금 잔액
06:      public int getMoney(){
07:          return this.money;
08:      }
09:      public void setMoney(int money){
10:          this.money = money;
11:      }
12:      public synchronized void saveMoney(int save){
13:          int m = this.getMoney();
14:          try{
15:              Thread.sleep(3000);
16:          }catch(InterruptedException e){e.printStackTrace();}
17:          this.setMoney(m + save);
18:      }
19:      public void minusMoney(int minus){
20:          synchronized(this){
21:              int m = this.money;
22:              try{
23:                  Thread.sleep(200);
24:              }catch(InterruptedException e){e.printStackTrace();}
25:              this.setMoney(m - minus);
26:          }
27:      }
28:  } //end of Bank class
```



```

28: class Park extends Thread{
29:     public void run(){
30:         SyncMain.myBank.saveMoney(3000);
31:         System.out.println("saveMoney(3000) :" + SyncMain.myBank.getMoney());
32:     }
33: } //end of Park class
34:
35: class ParkWife extends Thread{
36:     public void run(){
37:         SyncMain.myBank.minusMoney(1000);
38:         System.out.println("minusMoney(1000) :" + SyncMain.myBank.getMoney());
39:     }
40: } //end of ParkWife class
41:
42: public class SyncMain{
43:     public static Bank myBank = new Bank();
44:     public static void main(String[] args) throws Exception{
45:         System.out.println("원금 :" + myBank.getMoney());
46:         Park p = new Park();
47:         ParkWife w = new ParkWife();
48:         p.start();
49:         try{
50:             Thread.sleep(200);
51:         }catch(InterruptedException e){e.printStackTrace();} 62
52:         w.start();
53:     } //end of main
54: } //end of NotSyncMain class
C:\javasrc\chap08>javac SyncMain.java
C:\javasrc\chap08>java SyncMain
원금:10000
saveMoney(3000):13000
minusMoney(1000):12000

```



8.4.3 synchronized(계속)



- ❖ synchronized 메서드
 - ❖ public **synchronized** void saveMoney(int save){
 - ❖ //메서드 내에 사용된 **멤버 변수**들의 동기화가 보장된다.
 - ❖ }

- ❖ public **synchronized** void minusMoney(int minus){
 - ❖ //메서드 내에 사용된 **멤버 변수**들의 동기화가 보장된다.
 - ❖ }





8.4.3 synchronized(계속)



- ❖ synchronized 블록을 사용한 saveMoney() 메서드
 - ❖ public void saveMoney(int save){
 - ❖ synchronized(**this**){
 - ❖ //블록 내에 사용된 this 즉 현재 클래스의 멤버 변수들에 대한 동기화가 보장된다.
 - ❖ }
 - ❖ }
- ❖ synchronized 블록을 사용한 minusMoney() 메서드
 - ❖ public void minusMoney(int minus){
 - ❖ synchronized(**this**){
 - ❖ //블록 내에 사용된 this 즉 현재 클래스의 멤버 변수들에 대한 동기화가 보장된다.
 - ❖ }
 - ❖ }



8.4.3 synchronized(계속)



- ❖ synchronized 블록 사용시 주의
 - ❖ synchronized의 괄호에 들어가는 **this**이다.
 - ❖ synchronized 블록을 사용할 때 괄호 안에 사용된 객체의 의미는 공유자원이 어디에 존재하는가의 문제이다.
 - ❖ 공유자원이 **현재의 클래스에 존재하기 때문에** synchronized 괄호 안에 this가 사용된 것이다.
 - ❖ 현재 클래스의 멤버가 공유자원이면 현재 클래스를 의미하는 this를 사용하면 된다.





8.4.4 synchronized의 활용



```
❖ public void run(){
❖     synchronized(SyncMain2.myBank){
❖         SyncMain2.myBank.saveMoney(3000);
❖     }
❖ }
```



```
❖ public void run(){
❖     synchronized(SyncMain2.myBank){
❖         SyncMain2.myBank.minusMoney(1000);
❖     }
❖ }
```



§8-15 SyncMain2.java

```
01:  /**
02:  synchronized 블록을 동기화 예제(또 다른 방법)
03:  */
04:  class Bank{
05:      private int money = 10000; //예금 잔액
06:      public int getMoney(){
07:          return this.money;
08:      }
09:      public void setMoney(int money){
10:          this.money = money;
11:      }
12:      public void saveMoney(int save){
13:          int m = this.getMoney();
14:          try{
15:              Thread.sleep(3000);
16:          }catch(InterruptedException e){e.printStackTrace();}
17:          this.setMoney(m + save);
18:      }
19:      public void minusMoney(int minus){
20:          int m = this.money;
21:          try{
22:              Thread.sleep(200);
23:          }catch(InterruptedException e){e.printStackTrace();}
24:          this.setMoney(m - minus);
25:      }
26:  } //end of Bank class
```



```

28: class Park extends Thread{
29:     public void run(){
30:         synchronized(SyncMain2.myBank){
31:             SyncMain2.myBank.saveMoney(3000);
32:         }
33:         System.out.println("saveMoney(3000) :" + SyncMain2.myBank.getMoney());
34:     }
35: }
36: } //end of Park class
37:
38: class ParkWife extends Thread{
39:     public void run(){
40:         synchronized(SyncMain2.myBank){
41:             SyncMain2.myBank.minusMoney(1000);
42:         }
43:         System.out.println("minusMoney(1000) :" + SyncMain2.myBank.getMoney());
44:     }
45: }
46: } //end of ParkWife class
47: public class SyncMain2{
48:     public static Bank myBank = new Bank();
49:     public static void main(String[] args) throws Exception{
50:         System.out.println("원금 :" + myBank.getMoney());
51:         Park p = new Park();
52:         ParkWife w = new ParkWife();
53:         p.start();
54:         try{
55:             Thread.sleep(200);
56:         }catch(InterruptedException e){e.printStackTrace();}
57:         w.start();
58:     } //end of main
59: } //end of SyncMain2 class

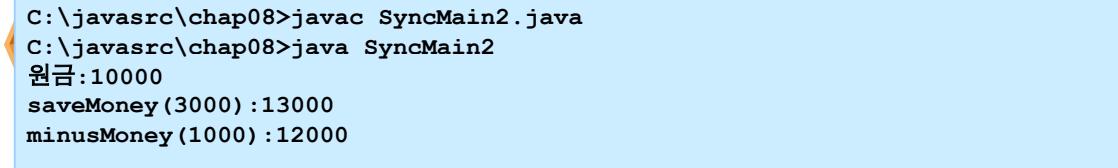
```



```

C:\javasrc\chap08>javac SyncMain2.java
C:\javasrc\chap08>java SyncMain2
원금:10000
saveMoney(3000) :13000
minusMoney(1000) :12000

```



- ❖ synchronized 블록을 이용한 동기화
 - ❖ synchronized(SyncMain2.myBank){
 - ❖ //SyncMain2.myBank 멤버의 동기화 보장
 - ❖ }
- 



8.4.5 synchronized 한계



- ❖ 비디오 가게를 만들자
 - ❖ 비디오 가게에 비디오 테이프 : 5 개
 - ❖ 손님들이 비디오를 빌려 간다
 - ❖ 동기화가 보장되어야 한다.
 - ❖ 두 사람이 동시에 같은 비디오를 빌려 가면 안 된다.



§8-16 VideoShopMain.java

```
01:  /**
02:  잘못된 동기화의 예
03: */
04:  import java.util.*;
05:  class VideoShop{
06:      private Vector buffer = new Vector();
07:      public VideoShop(){
08:          buffer.addElement("은하철도999-0");
09:          buffer.addElement("은하철도999-1");
10:          buffer.addElement("은하철도999-2");
11:          buffer.addElement("은하철도999-3");
12:      }
13:      public String lendVideo(){
14:          String v = (String) this.buffer.remove(buffer.size()-1);
15:          return v;
16:      }
17:      public void returnVideo(String video){
18:          this.buffer.addElement(video);
19:      }
20:  } //end of VideoShop class
22:  class Person extends Thread{
23:      public void run(){
24:          synchronized(VideoShopMain.vShop){
25:              //5초 동안 VideoShopMain.vShop은 락(Lock)에 걸리게 된다.
26:              try{
27:                  String v = VideoShopMain.vShop.lendVideo();
28:                  System.out.println(this.getName() + ":" + v + " 대여");
29:              } catch(InterruptedException e){}
```



```

29:             System.out.println(this.getName() + ":" + v + " 보는중");
30:             this.sleep(5000);
31:             System.out.println(this.getName() + ":" + v + " 반납");
32:             VideoShopMain.vShop.returnVideo(v);
33:         }catch(InterruptedException e){e.printStackTrace();}
34:     }
35: }
36: } //end of Person class
37:
38: class VideoShopMain{
39:     public static VideoShop vShop = new VideoShop();
40:     public static void main(String[] args){
41:         System.out.println("프로그램 시작");
42:         Person p1 = new Person();
43:         Person p2 = new Person();
44:         Person p3 = new Person();
45:         Person p4 = new Person();
46:         p1.start();
47:         p2.start();
48:         p3.start();
49:         p4.start();
50:         System.out.println("프로그램 종료");
51:     }
52: } //end of VideoShopMain class

```



```

C:\javasrc\chap08>javac VideoShopMain.java
C:\javasrc\chap08>java VideoShopMain
프로그램 시작
프로그램 종료
Thread-1:은하철도999-3 대여
Thread-1:은하철도999-3 보는중
Thread-1:은하철도999-3 반납
Thread-2:은하철도999-3 대여
Thread-2:은하철도999-3 보는중
Thread-2:은하철도999-3 반납
Thread-3:은하철도999-3 대여
Thread-3:은하철도999-3 보는중
Thread-3:은하철도999-3 반납
Thread-4:은하철도999-3 대여
Thread-4:은하철도999-3 보는중
Thread-4:은하철도999-3 반납

```



8.4.5 synchronized 한계(계속)

- ❖ synchronized 블록
 - ❖ synchronized(VideoShopMain.vShop){
 - ❖ String v = VideoShopMain.vShop.lendVideo(); //비디오를 빌린다.
 - ❖ try{
 - ❖ this.sleep(5000); //비디오를 보는 시간 5초
 - ❖ }catch(InterruptedException e){e.printStackTrace();}
 - ❖ VideoShopMain.vShop.returnVideo(v); //비디오를 반납한다.
 - ❖ }
- ❖ 비디오를 보는 5초 동안 VideoShop vShop은 synchronized로 인해 락(Lock)이 걸린 상태
 - ❖ 이럴 경우 비디오 가게에서는 단 하나의 테이프만을 운영하는 것이 된다.
 - ❖ 결과를 보시면 '은하철도999- 3' 비디오 테이프만을 대여하고 있는 것을 확인 할 수 있다.
 - ❖ 이것은 일반적인 메서드 호출이나 별반 다른 것이 없다.
 - ❖ 즉 비디오 가게에서는 4개의 테이프를 동시에 빌려 주어야만 장사를 잘하는 것이다.



8.4.5 synchronized 한계(계속)

- ❖ synchronized를 다시 적용하기
 - ❖ 테이프를 빌려주는 곳과 반환하는 곳에 synchronized를 거는 것이 더 정확한 사용 방법입니다.
- ❖ synchronized를 걸어야 하는 곳
 - ❖ public synchronized String lendVideo(){...}
 - ❖ public synchronized void returnVideo(String video){...}



§8-17 VideoShopMain2.java

```
01:  /**
02:   * 데이터를 집어넣고 추출할 때에만 동기화를 보장
03:   */
04:  import java.util.*;
05:  class VideoShop{
06:      private Vector buffer = new Vector();
07:      public VideoShop(){
08:          buffer.addElement("은하철도999-0");
09:          buffer.addElement("은하철도999-1");
10:          buffer.addElement("은하철도999-2");
11:          buffer.addElement("은하철도999-3");
12:      }
13:      public synchronized String lendVideo(){
14:          String v = (String) this.buffer.remove(buffer.size()-1);
15:          return v;
16:      }
17:      public synchronized void returnVideo(String video){
18:          this.buffer.addElement(video);
19:      }
20:  } //end of VideoShop class
22:  class Person extends Thread{
23:      public void run(){
24:          try{
25:              String v = VideoShopMain2.vShop.lendVideo();
26:              System.out.println(this.getName() + ":" + v + " 대여");
27:              System.out.println(this.getName() + ":" + v + " 보는중");
28:              this.sleep(5000);
29:              System.out.println(this.getName() + ":" + v + " 반납");
30:              VideoShopMain2.vShop.returnVideo(v);
31:          }catch(InterruptedException e){e.printStackTrace();}
32:      }
33:  } //end of Person class
35:  class VideoShopMain2{
36:      public static VideoShop vShop = new VideoShop();
37:      public static void main(String[] args){
38:          System.out.println("프로그램 시작");
39:          Person p1 = new Person();
40:          Person p2 = new Person();
41:          Person p3 = new Person();
42:          Person p4 = new Person();
43:
44:          p1.start();
45:          p2.start();
46:          p3.start();
47:          p4.start();
48:          System.out.println("프로그램 종료");
49:      } //end of main
50:  } //end of VideoShopMain2 class
```



```
C:\javasrc\chap08>javac VideoShopMain2.java
C:\javasrc\chap08>java VideoShopMain2
프로그램 시작
프로그램 종료
Thread-1: 은하철도999-3 대여
Thread-2: 은하철도999-2 대여
Thread-3: 은하철도999-1 대여
Thread-4: 은하철도999-0 대여
Thread-1: 은하철도999-3 보는중
Thread-2: 은하철도999-2 보는중
Thread-3: 은하철도999-1 보는중
Thread-4: 은하철도999-0 보는중
Thread-1: 은하철도999-3 반납
Thread-2: 은하철도999-2 반납
Thread-3: 은하철도999-1 반납
Thread-4: 은하철도999-0 반납
```



8.4.5 synchronized 한계(계속)

- ❖ 위의 프로그램은 synchronized를 `lendVideo()`와 `returnVideo()`에 적용
 - ❖ 4개의 비디오를 **동시**에 대여할 수 있다.
 - ❖ 비디오를 보는 동안 synchronized 처리하는 것보다는 훨씬 유연하게 동작하는 것을 확인할 수 있다.
- ❖ **하지만** 이 경우에는 또 다른 문제가 제기
 - ❖ 즉 비디오 테이프를 보는 곳에 동기화를 걸지 않았기 때문에 5번째 비디오 테이프를 빌리면 심각한 문제가 발생할 것
 - ❖ 위의 예에서 다음과 같이 5번째 비디오를 빌리는 구문을 삽입하면 에러가 발생합니다.
- ❖ 문제발생
 - ❖ `Person p5 = new Person();`
 - ❖ `p5.start();`



§8-18 VideoShopMain3.java

```
01:  /**
02:   * 여유분이 있을 때만 비디오 테이프를 빌려주기
03:  */
04:  import java.util.*;
05:  class VideoShop{
06:      private Vector buffer = new Vector();
07:      public VideoShop(){
08:          buffer.addElement("은하철도999-0");
09:          buffer.addElement("은하철도999-1");
10:          buffer.addElement("은하철도999-2");
11:          buffer.addElement("은하철도999-3");
12:      }
13:      public synchronized String lendVideo(){
14:          if(buffer.size()>0){
15:              String v = (String) this.buffer.remove(buffer.size()-1);
16:              return v;
17:          }else{
18:              return null;
19:          }
20:      }
21:      public synchronized void returnVideo(String video){
22:          this.buffer.addElement(video);
23:      }
24:  } //end of VideoShop class
25:  class Person extends Thread{
26:      public void run(){
27:          String v = VideoShopMain3.vShop.lendVideo();
28:      }

```

```
9:          if( v == null){
0:              System.out.println(this.getName() + "비디오가 없군요. 안봅니다.");
1:              return;
2:          }
3:          try{
4:              System.out.println(this.getName() + ":" + v + " 대여");
5:              System.out.println(this.getName() + ":" + v + " 보는중\n");
6:              this.sleep(5000);
7:              System.out.println(this.getName() + ":" + v + " 반납");
8:              VideoShopMain3.vShop.returnVideo(v);
9:          }catch(InterruptedException e){e.printStackTrace();}
0:      }
1:  } //end of Person class
2:  public class VideoShopMain3{
3:      public static VideoShop vShop = new VideoShop();
4:      public static void main(String[] args){
5:          System.out.println("프로그램 시작");
6:          Person p1 = new Person();
7:          Person p2 = new Person();
8:          Person p3 = new Person();
9:          Person p4 = new Person();
0:          Person p5 = new Person();
1:          p1.start();
2:          p2.start();
3:          p3.start();
4:          p4.start();
5:          p5.start();
6:          System.out.println("프로그램 종료");
7:      }
8:  } //end of main
9: } //end of VideoShopMain3 class
```



```
C:\javasrc\chap08>javac VideoShopMain3.java
C:\javasrc\chap08>java VideoShopMain3
프로그램 시작
프로그램 종료
Thread-1: 은하철도999-3 대여
Thread-1: 은하철도999-3 보는중

Thread-2: 은하철도999-2 대여
Thread-2: 은하철도999-2 보는중

Thread-3: 은하철도999-1 대여
Thread-3: 은하철도999-1 보는중

Thread-4: 은하철도999-0 대여
Thread-4: 은하철도999-0 보는중

Thread-5비디오가 없군요. 안봅니다.
Thread-1: 은하철도999-3 반납
Thread-2: 은하철도999-2 반납
Thread-3: 은하철도999-1 반납
Thread-4: 은하철도999-0 반납
```



8.4.5 synchronized 한계(계속)

- ❖ 비디오 테이프가 없을 때 null을 리턴
 - ❖ if(buffer.size()>0){
 - ❖ String v = (String)this.buffer.remove(buffer.size()- 1);
 - ❖ return v;
 - ❖ }else{
 - ❖ return null;
 - ❖ }
- ❖ Person에서는 비디오 테이프가 없다면 다음과 같이 바로 스레드를 종료하게 됩니다.
- ❖ if(v == null){
 - ❖ System.out.println(this.getName() + "비디오가 없군요. 안봅니다.");
 - ❖ return ; //비디오 테이프를 빌리는 것 자체를 포기한다.
 - ❖ }
 - ❖ }





8.4.5 synchronized 한계(계속)

- ❖ 이렇게 한다면 여러 발생은 막을 수는 있다.
 - ❖ 하지만 자원을 요청한 사람은 대기하는 것이 아니라 아예 **포기**하는 것이다.
 - ❖ 즉 이 비디오 가게는 장사를 잘 못하는 것이다.
 - ❖ 만약 제대로 장사를 한다면 잠깐 기다리라고 한 뒤 비디오 테이프가 들어오면 바로 빌려주면 될 것이다.

- ❖ wait()와 notify()를 사용하면 된다.
 - ❖ 예제는 10장에서 다루게 될 것.



8.5 마무리



8.5.1 결론





8.5.1 결론



- ❖ 이 장에서 배운 것
 - ❖ 스레드를 여러 개 생성하는 방법과 우선권을 제어하는 방법
 - ❖ 멀티 스레드를 사용할 때 발생하는 문제점
- ❖ 실제로 프로그램에서는 위의 예제들과 같이 간단하게 사용되지 않음
 - ❖ 오히려 많은 생각을 해야만 스레드들의 제어와 동기화 문제를 해결할 수 있다..



10. Object Class

