File IO, NIO

1. try with (AutoClosable resource)

- JDK 1.7 이상 버전에서의 예외처리

```
try( Connection con = getConnection() ) {
    String sql = "select * from dual";
    Statement stmt = .....
    try( ResultSet rs = stmt.executeQuery(sql) ) {
        ...
    }
} catch(Exception e) {
    e.printStackTrace();
}
```

- finally 가 없어도 자동으로 close 처리해준다
- Connection 과 ResultSet이 내부적으로 AutoClosable interface를 구현하고 있기 때문에 가능하다

2. String 연산

- JDK 1.5 이상 버전에서의 String 연산

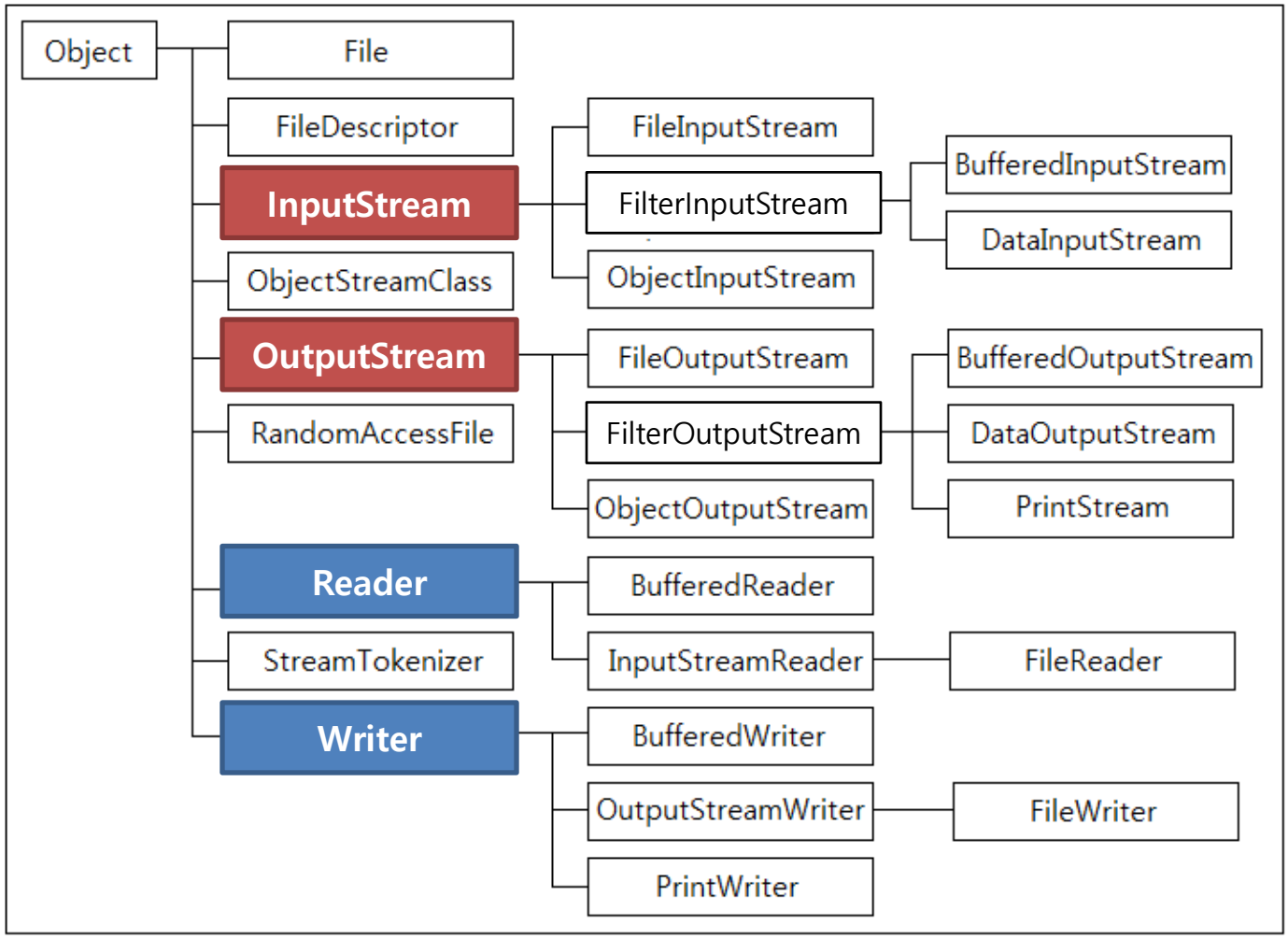
```
- 기존
StringBuffer sbf = new StringBuffer();
sbf.append("aaa");
sbf.append("bbb");

StringBuilder sbd = new StringBuilder();
sbd.append("aaa").append("bbb");

- Jdk 1.5 이상에서의 일반적인 String 연산시
String a = "aaa" + "bbb";
```

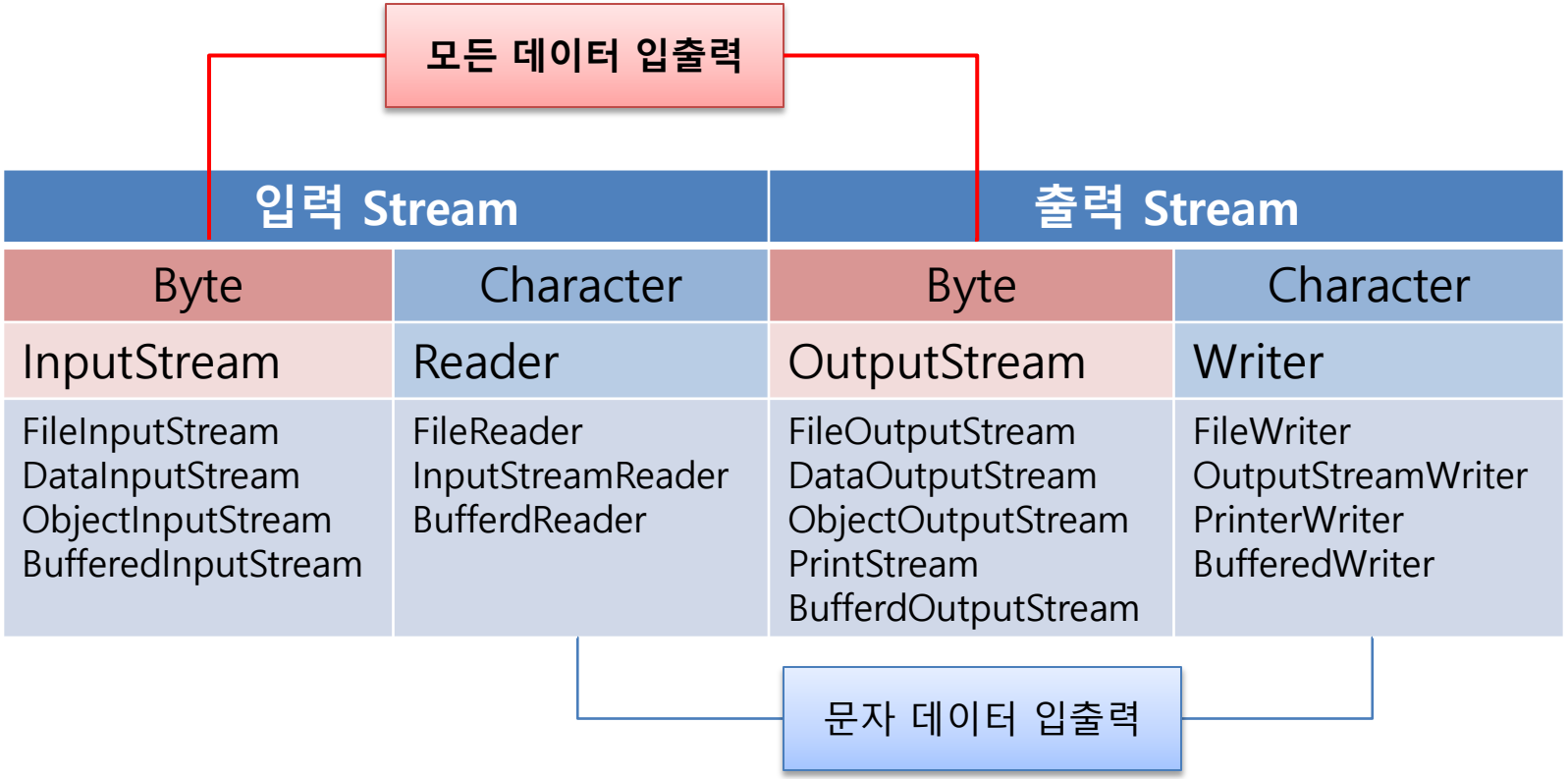
- JDK 5.0 이상부터는 String의 +연산을 컴파일러가 자동으로 StringBuilder 연산으로 바꿔주지만, 반복문 안에서의 +연산은 여전히 일반 +연산으로 수행하므로 가급적 StringBuffer나 StringBuilder를 사용한다.

3. Java IO Stream package



4. Java IO Stream 구성

- 아래 표와 같이 Stream 은 크게 입력, 출력 두가지 형태로 구분할 수 있는데 Byte 기반 스트림은 이미지, 동영상, 문자 등 모든 종류의 미디어를 주고받을 수 있는데 반해, Character 기반 스트림은 문자만 주고 받도록 설계되어 있다



5. Java IO Stream 예제

1. 파일 입출력 스트림 - 파일 읽고 쓰기

```
// 입력 스트림 생성 < image.jpg 파일을 읽어온다
InputStream fis = new FileInputStream("image.jpg");

// 출력 스트림 생성 > new_image.jpg 파일로 출력(생성)한다
OutputSteam fos = new FileOutputStream("new_image.jpg");

int readCount; // 읽어온 바이트의 개수
byte[ ] buffer = new byte[1024];
while((readCount = fis.read(buffer)) != -1){ //buffer단위로 읽는다
      // buffer 단위로 파일을 출력한다
      fos.write(buffer, 0, readCount);
}
fis.close(); // 입력과 출력 Stream 을 닫아서 자원을 반환한다
fos.close();
```


6. Reader 기준 파일처리 속도

- 아래는 읽기를 기준으로 한 파일처리 속도비교이다

1. Reader vs InputStream

사용클래스	처리시간(ms)
FileReader	18000
FileInputStream	800

2. BufferedReader vs BufferedInputStream

사용클래스	처리시간(ms)
BufferedReader	150
BufferedInutStream	80

7. Java NIO package

자바 4부터 새로운 입출력(NIO: New Input/Output)이라는 뜻에서 java.nio 패키지가 포함되었고, 자바 7로 버전업하면서 자바 IO와 자바 NIO 사이의 일관성 없는 클래스 설계를 바로 잡고, 비동기 채널 등의 네트워크 지원을 대폭 강화한 NIO.2 API가 추가되었다.

NIO.2는 java.nio2 패키지로 제공되지 않고 기존 java.nio의 하위 패키지(java.nio.channels, java.nio.charset, java.nio.file)에 통합되어 있다.

NIO 패키지	포함되어 있는 내용
java.nio	다양한 버퍼 클래스
java.nio.channels	파일 채널, TCP 채널, UDP 채널 등의 클래스
java.nio.channels.spi	java.nio.channels 패키지를 위한 서비스 제공자 클래스
java.nio.charset	문자셋, 인코더, 디코더 API
java.nio.charset.spi	java.nio.charset 패키지를 위한 서비스 제공자 클래스
java.nio.file	파일 및 파일 시스템에 접근하기 위한 클래스
java.nio.file.attribute	파일 및 파일 시스템의 속성에 접근하기 위한 클래스
java.nio.file.spi	java.nio.file 패키지를 위한 서비스 제공자 클래스

8. IO vs NIO

IO와 NIO는 데이터를 입출력한다는 목적은 동일하지만, 방식에 있어서는 아래와 같은 차이점이 있다.

구분	IO	NIO
입출력 방식	스트림(stream) 방식	채널(channel) 방식
버퍼 방식	넌버퍼(non-buffer)	버퍼(buffer)
비동기 방식	지원 안 함	지원
블로킹 / 넌블로킹 방식	블로킹 방식만 지원	블로킹 / 넌 블로킹 모두 지원

1. stream vs channel
   채널은 입출력시 하나만 생성하는 반면, 스트림은 입력과 출력 두개 필요
2. non-buffer vs buffer
   버퍼는 HW / SW간의 속도차이를 줄여준다. 빠르다.
   채널은 기본적으로 버퍼를 사용한다
3. blocking vs non-blocking
   스트림은 파일 읽기시 해당 thread 가 blocking 되고 interrupt 로 빠져나오는 것도 불가능하다. stream.close( ) 로만 blocking을 해제할 수 있다.

8. IO vs NIO 선택기준

- NIO 장점

불특정 다수의 클라이언트 연결 또는 멀티 파일들을 넌 블로킹이나 비동기로 처리할 수 있기 때문에 과도한 스레드 생성을 피하고 스레드를 효과적으로 재사용한다는 점에서 큰 장점이 있다.

- IO 장점

대용량 데이터를 처리할 경우에는 IO가 더 유리한데, NIO는 버퍼의 할당 크기도 문제가 되고, 모든 입출력 작업에 버퍼를 무조건 사용해야 하므로 받은 즉시 처리하는 IO 보다는 좀 더 복잡하다.

- NIO 사용

1. 연결 클라이언트 수가 많고
2. 하나의 입출력 처리작업이 오래 걸리지 않는 경우에 사용

- IO 사용

1. 연결 클라이언트 수가 적고,
2. 전송되는 데이터가 대용량이면서 순차적으로 처리될 필요성이 있는 경우

9. NIO Path 처리 (jdk 7 이상)

- 자바 8의 파일 처리를 위해서는 먼저 자바 7의 파일처리 변경사항을 이해해야 만 한다. 아래는 자바 7의 파일처리 변경사항 요약이다

1. 파일처리의 간소화를 위해 NIO2 도입 (NIO = new I/O)
2. 경로에 대한 처리를 위해 Path 가 추가 되었다

```
Path path1 = Paths.get("/", "절대경로");   // 앞에 /(루트) 포함시
Path path2 = Paths.get("상대경로","파일명");
```

3. 파일의 읽기와 쓰기를 위한 method 가 추가되었다

```
// 읽기
byte bytes[ ] = Files.readAllBytes(path);
String content = new String(bytes, StandardCharsets.UTF_8);
List<String> lines = Files.readAllLines(path);
// 쓰기
Files.write(path, content.getBytes(StandardCharsets.UTF_8));
```


10. NIO File 처리 (jdk 7 이상)

- 자바 8의 파일 처리를 위해서는 먼저 자바 7의 파일처리 변경사항을 이해해야 만 한다. 아래는 자바 7의 파일처리 변경사항 요약이다

4. 파일과 디렉토리 생성

```
// 경로상의 마지막 디렉토리만 생성
Files.createDirectory(path);
// 경로상의 모든 디렉토리 생성 (없을 경우)
Files.createDirectories(path);
```

5. 파일 복사, 이동, 삭제

```
Files.copy(fromPath, toPath);
Files.move(fromPath, toPath);
Files.delete(path);
// 삭제시 없는경우에 대한 예외처리
boolean deleted = Files.deleteIfExists(path);
```

