7. NIO 기반 입출력

1. NIO 소개
2. 파일과 디렉토리
3. 버퍼(Buffer)
4. 파일 채널(FileChannel)

1. NIO 소개

❖ NIO(New Input/Output)

- 기존 java.io API와 다른 새로운 입출력 API
- 자바4부터 추가 → 자바7부터 네트워크 지원 강화된 NIO.2 API 추가
- 관련 패키지

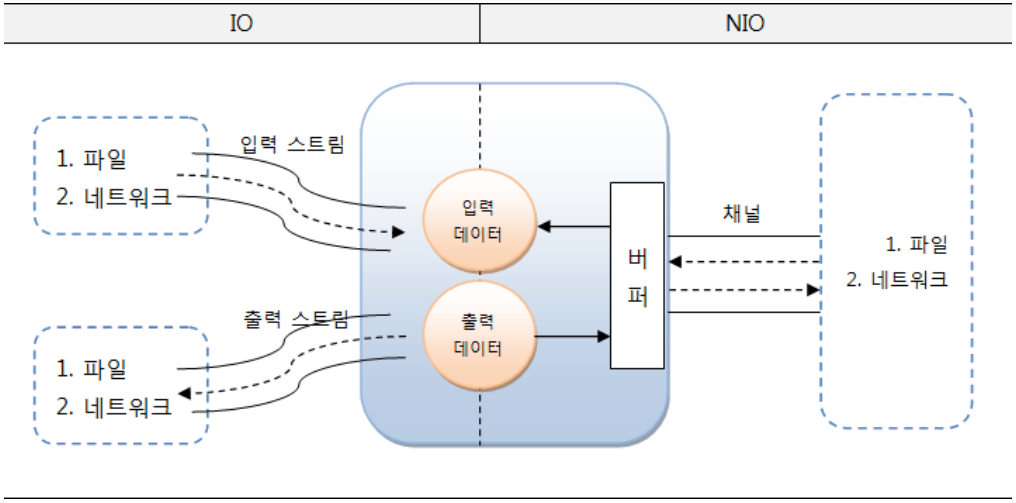
NIO 패키지	포함되어 있는 내용
java.nio	다양한 버퍼 클래스
java.nio.channels	파일 채널, TCP 채널, UDP 채널등의 클래스
java.nio.channels.spi	java.nio.channels 패키지를 위한 서비스 제공자 클래스
java.nio.charset	문자셋, 인코더, 디코더 API
java.nio.charset.spi	java.nio.charset 패키지를 위한 서비스 제공자 클래스
java.nio.file	파일 및 파일 시스템에 접근하기 위한 클래스
java.nio.file.attribute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

1. NIO 소개

❖ IO와 NIO의 차이점

구분	IO	NIO
입출력 방식	스트림 방식	채널 방식
버퍼 방식	넌버퍼(non-buffer)	버퍼(buffer)
동기 / 비동기 방식	동기 방식	동기 / 비동기 방식 모두 지원
블로킹 / 넌블로킹 방식	블로킹 방식	블로킹 / 넌블로킹 방식 모두 지원

- 스트림 vs. 채널
  - IO 스트림: 입력 스트림과 출력 스트림으로 구분되어 별도 생성
  - NIO 채널: 양방향으로 입출력이 가능하므로 하나만 생성



1. NIO 소개

❖ IO와 NIO의 차이점

- 넌버퍼 vs. 버퍼
  - IO 스트림 - 넌버퍼(non-buffer)
    - IO에서는 1바이트씩 읽고 출력 – 느림
    - 보조 스트림인 BufferedInputStream, BufferedOutputStream를 사용해 버퍼 제공 가능
    - 스트림으로부터 입력된 전체 데이터를 별도로 저장해야
      - 저장 후 입력 데이터의 위치 이동해가면서 자유롭게 이용 가능
  - NIO 채널 - 버퍼(buffer)
    - 기본적으로 버퍼 사용해 입출력 - 성능 좋음
    - 읽은 데이터를 무조건 버퍼(Butter: 메모리 저장소)에 저장
      - 버퍼 내에서 데이터 위치 이동해 가며 필요한 부분만 읽고 쓸 수 있음

1. NIO 소개

❖ IO와 NIO의 차이점

- 블로킹 vs 넌블로킹
  - IO 스트림 - 블로킹
    - 입력 스트림의 read() 메소드 호출
      - 데이터 입력 전까지 스레드는 블로킹(대기상태)
    - 출력 스트림의 write() 메소드 호출
      - 데이터 출력 전까지 스레드는 블로킹
    - 스레드 블로킹 -다른 일을 할 수가 없고 interrupt 해 블로킹 빠져나올 수도 없음
    - 블로킹을 빠져 나오는 유일한 방법 - 스트림을 닫는 것
  - NIO 채널 – 블로킹, 넌블로킹
    - NIO 블로킹은 스레드를 interrupt 함으로써 빠져나올 수 있음
    - NIO는 넌블로킹 지원
      - 입출력 작업 시 스레드가 블로킹되지 않음
    - 준비 완료된 채널을 선택하는 기능!!!

1. NIO 소개

❖ 네트워크 프로그램 개발 시 IO와 NIO의 선택

- IO 방식 선택하는 경우
  - 연결 클라이언트의 수가 적고,
  - 전송되는 데이터가 대용량이면서
  - 순차적으로 처리될 필요성 있을 경우

- NIO 방식 선택하는 경우
  - 연결 클라이언트의 수가 많고
  - 전송되는 데이터 용량이 적으면서,
  - 입출력 작업 처리가 빨리 끝나는 경우

2. 파일과 디렉토리

❖ 파일 관련 패키지

- IO는 파일의 속성 정보 읽기 위해 File 클래스만 제공
- NIO는 좀 더 다양한 파일의 속성 정보 제공
  - 클래스와 인터페이스를 java.nio.file, java.nio.file.attribute 패키지에서 제공

❖ 경로 정의(Path)

- java.nio.file.Path 인터페이스
  - IO의 java.io.File 클래스에 대응
  - NIO의 여러 곳에서 파일 경로 지정 위해 Path 사용

- Path 구현 객체
  - java.nio.file.Paths 클래스의 정적 메소드인 get() 메소드로 얻음
  - Path path=Paths.get(String first, String… more);
  - Path path=Paths.getURI uri);

2. 파일과 디렉토리

❖ Path 주요 메소드

리턴타입	메소드 (매개변수)	설명
int	compareTo(Path other)	파일 경로가 동일하면 0, 상위면 -1, 하위면 양수리턴,
Path	getFileName()	부모경로를 제외한 파일 또는 디렉토리 이름만 가진 Path 리턴
FileSystem	getFileSystm()	FileSystme 객체 리턴
Path	getName(int index)	C:₩Temp₩dir₩file.txt 일 경우, 인덱스가 0이면 Temp, 1이면 dir , 인덱스가 2이면 file.txt리턴
int	getNameCount()	중첩 경로 수 리턴, C:₩Teml₩dir₩file.txt이면 3
Path	getParent()	바로 위 부모 폴더 리턴
Path	getRoot()	루트 디렉토리 의 Path 리턴
Iterator(Path)	iterator()	경로에 있는 모든 디렉토리와 파일을 Path객체로 생성하고 반복자를 리턴
Path	normalize()	상대경로 표기할때 불필료한 요소 제거 C:₩Temp₩dir1₩...₩dir2₩file.txt -> C:₩Temp₩dir2₩file.txt
Watchkey	register()	WatchService를 등록
File	toFile	java.io.File 객체 리턴
String	toString()	파일 경로를 문자열로 리턴
URI	toLine()	파일 경로를 URI로 리턴

2. 파일과 디렉토리

❖ 파일 시스템 정보(FileSystem)

- 운영체제의 파일 시스템은 FileSystem인터페이스 통해 접근
- FileSystem 구현 객체는 FileSystems의 정적 메소드인 getDefault()로 얻을 수 있음

```
FileSystem fileSystem = FileSystems.getDefault();
```

- FileSystem에서 제공하는 메소드

리턴타입	메소드(매개변수)	설명
Iterable<FileStore>	getFileStores()	드라이버 정보를 가진 FileStore 객체들을 리턴
Iterable<Path>	getRootDirectories()	루트 디렉토리 정보를 가진 Path 객체들을 리턴
String	getSeparator()	디렉토리 구분자 리턴

2. 파일과 디렉토리

❖ FileStore

- 드라이버를 표현하는 객체
- 주요 메소드

리턴타입	메소드(매개변수)	설명
long	getTotalSpace()	드라이버 전체 공간 크기(바이트단위) 리턴
long	getUnallocatedSpace()	할당되지 않은 공간크기(바이트단위)리턴
long	getUsableSpace()	사용가능한 공간 크기 getUnallocatedSpacee()와 동일한 값
boolean	isReadOnly()	읽기 전용
String	name()	드라이브명 리턴
String	type	파일 시스템 종료

2. 파일과 디렉토리

❖ 파일 및 디렉토리 생성/삭제 및 속성 읽기(Files)

- java.nio.file.Files 클래스
  - 파일과 디렉토리의 생성 및 삭제, 이들의 속성을 읽는 메소드 제공
    - 속성: 숨김, 디렉토리 여부, 크기, 소유자
  - 제공 메소드

리턴 타입	메소드 매개변수	설명
long 또는 Path	copy(...)	복사
Path	createDirectoris(...)	모든 부모 디렉토리 생성
Path	createDirectory(...)	경로의 마지막 디렉토리 생성
Path	createFile(...)	파일 생성
void	delete(...)	삭제
boolean	deleteExists(...)	존재한다면 삭제
boolena	exists(...)	존재 여부
FileStore	getFileStore(...)	파일이 위치한 드라이브 리턴
FileTime	getLastModifiedTime(...)	마지막 수정 시간을 리턴

2. 파일과 디렉토리

❖ 와치 서비스(WatchService)

- 디렉토리의 내용 변화 감시
- 자바 7에서 처음 소개
  - 디렉토리내의 파일의 생성, 삭제, 수정 감시
  - 적용 사례
    - 에디터 바깥에서 파일 내용 수정하면 파일 내용이 변경됐으니 파일을 다시 불러올 것인지 묻는 대화상자 띄움
- 와치 생성

```
WatchService watchService = FileSystems.getDefault().newWatchService();
```

- WatchService 등록

```
directory.register(watchService,
                   StandardWatchEventKinds.ENTRY_CREATE,
                   StandardWatchEventKinds.ENTRY_DELETE,
                   StandardWatchEventKinds.ENTRY_MODIFY);
```

2. 파일과 디렉토리

❖ 와치 서비스(WatchService)

- 와치 키 생성

```
While(true){
        WatchKey watchKey = watchService.take();
}
```

- WatchEvent 리스트 얻기

```
List<WatchEvent<?>> list = watchKey.pollEvents();
```

- WatchEvent 처리

2. 파일과 디렉토리

❖ 와치 서비스(WatchService)

- 와치 이벤트 처리

```
for(WatchEvent watchEvent : list) {
   //이벤트 종류 얻기
   Kind kind = watchEvent.kind();
   //감지된 Path 얻기
   Path path = (Path)watchEvent.context();
   if(kind == StandardWatchEventKinds.ENTRY_CREATE) {
      //생성되었을 경우, 실행할 코드
      Platform.runLater(()->textArea.appendText("파일 생성됨 -> " + path.getFileName() + "₩n"));
   } else if(kind == StandardWatchEventKinds.ENTRY_DELETE) {
      //삭제되었을 경우, 실행할 코드
      Platform.runLater(()->textArea.appendText("파일 삭제됨 -> " + path.getFileName() + "₩n"));
   } else if(kind == StandardWatchEventKinds.ENTRY_MODIFY) {
      //변경되었을 경우, 실행할 코드
      Platform.runLater(()->textArea.appendText("파일 변경됨 -> " + path.getFileName() + "₩n"));
   } else if(kind == StandardWatchEventKinds.OVERFLOW) {
   }
}
```

2. 파일과 디렉토리

❖ 와치 서비스(WatchService)

- 와치 종료

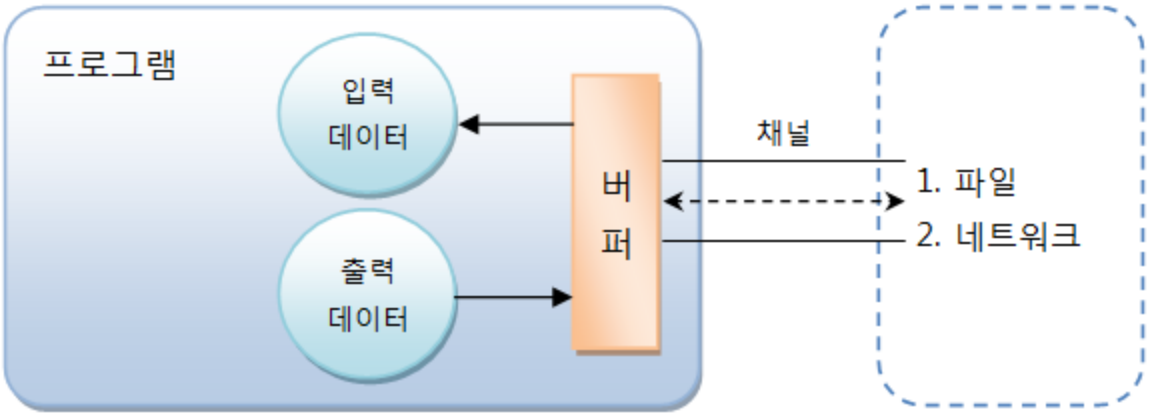
```
WatchService watchService = FileSystems.getDefault().newWatchService();
directory.register(watchService,
                   StandardWatchEventKinds.ENTRY_CREATE,
                   StandardWatchEventKinds.ENTRY_DELETE,
                   StandardWatchEventKinds.ENTRY_MODIFY);

while(true) {
    WatchKey watchKey = watchService.take();
    List<WatchEvent<?>> list = watchKey.pollEvents();
    for(WatchEvent watchEvent : list) {
        ...
    }
    boolean valid = watchKey.reset();
    if(!valid) { break; }
}
watchService.close()
```

3절. 버퍼

❖ 버퍼(Buffer)

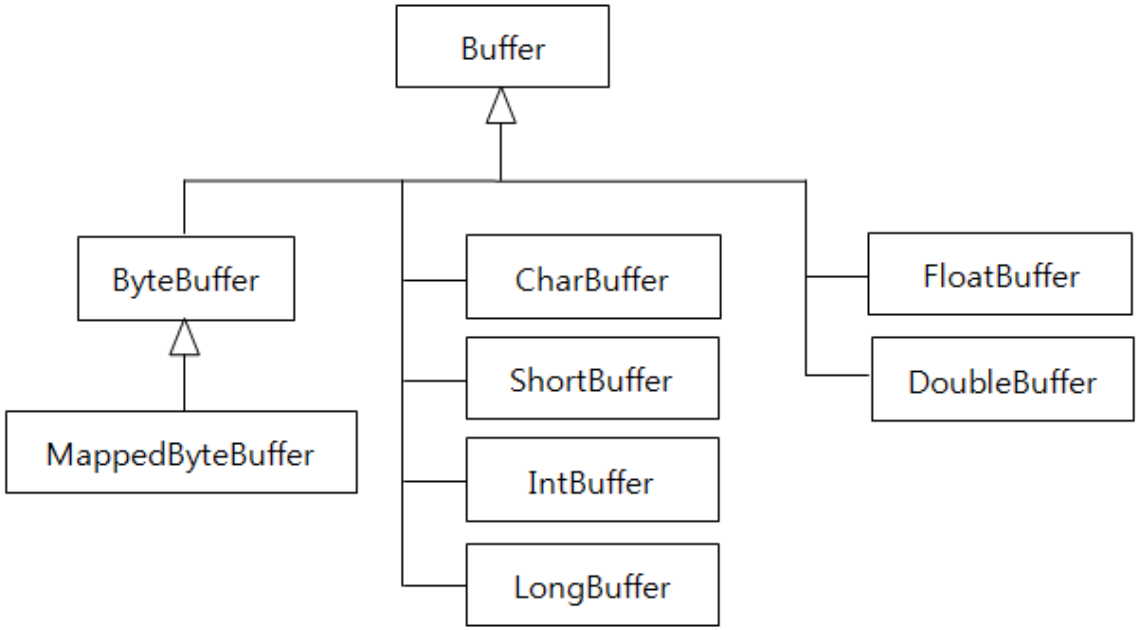
- 버퍼는 읽고 쓰기 가능한 메모리 배열
- NIO에서는 데이터를 입출력을 하기 위해서는 항상 버퍼 사용



3절. 버퍼

❖ Buffer 종류

- 분류 기준
  - 저장되는 데이터 타입에 따라 분류
    - ByteBuffer, CharBuffer, IntBuffer, DoubleBuffer
  - NIO 버퍼는 저장되는 데이터 타입에 따라 별도의 클래스로 제공
  - 이들 버퍼 클래스들은 Buffer 추상 클래스 모두 상속
  - MappedByteBuffer는 파일의 내용에 랜덤하게 접근하기 위해서 파일의 내용을 메모리와 맵핑시킨 버퍼



3절. 버퍼

❖ Buffer 종류

- 분류 기준
  - 어떤 메모리를 사용하느냐에 따른 분류
    - 다이렉트(Direct)와, 넌다이렉트(NonDirect) 버퍼
    - P.1115~1118 예제에서 버퍼 속도 차이만 확실히 이해

구분	넌다이렉트 버퍼	다이렉트 버퍼
사용하는 메모리 공간	JVM 의 힙 메모리	운영체제의 메모리
버퍼 생성 시간	버퍼 생성이 빠르다.	버퍼 생성이 느리다.
버퍼의 크기	작다.	크다.(큰 데이터를 처리할 때 유리)
입출력 성능	낮다.	높다.(입출력이 빈번할 경우 유리)

3절. 버퍼

❖ Buffer 생성

- allocate() 메소드(넌다이렉트 버퍼 생성)
  - 각 데이터 타입 별 넌다이렉트 버퍼 생성
  - 매개값 - 해당 데이터 타입의 저장 개수
  - Ex)

```
ByteBuffer byteBuffer = ByteBuffer.allocate(100);
CharBuffer charBuffer = CharBuffer.allocate(100);
```

- wrap() 메소드(넌다이렉트 버퍼 생성)
  - 이미 생성되어 있는 타입 별 배열을 래핑해 버퍼 생성
  - 일부 데이터만 가지고도 버퍼 생성 가능!

```
byte[] byteArray = new byte[100];
ByteBuffer byteBuffer = ByteBuffer.wrap(byteArray);
char[] charArray = new char[100];
CharBuffer charBuffer = CharBuffer.wrap(charArray);
CharBuffer charBuffer = CharBuffer.wrap("NIO 입출력은 버퍼를 이용합니다.");
```

3절. 버퍼

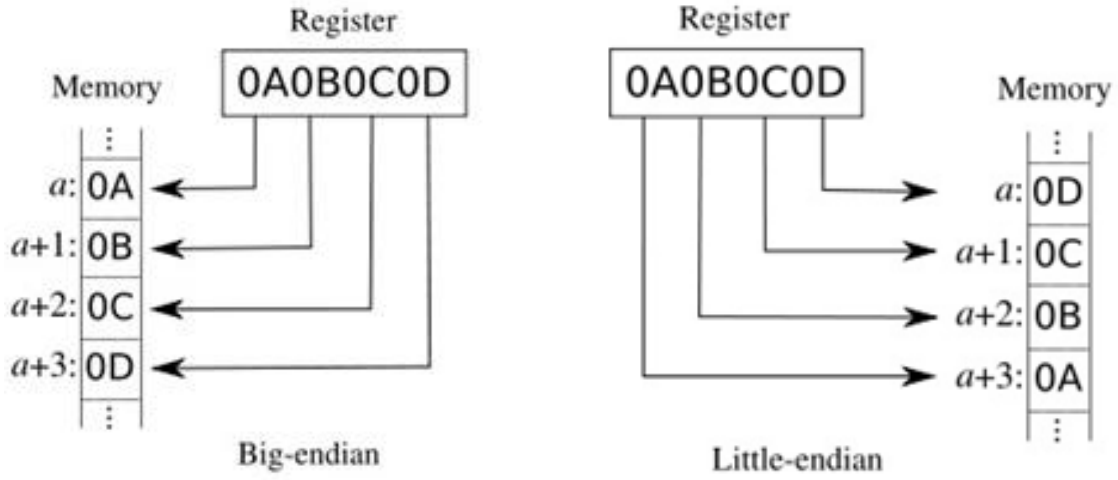
- allocateDirect() 메소드(다이렉트 버퍼 생성)
  - JVM 힙 메모리 바깥쪽 (운영체제가 관리하는 메모리)에 다이렉트 버퍼 생성
  - 각 타입 별 Buffer 클래스에는 없고 ByteBuffer 에서만 제공

  - asXXXBuffer() 메소드(각 타입 별 다이렉트 버퍼 생성)
    - asCharBuffer(), asShortBuffer(), asIntBuffer(), asLongBuffer(), asFloatBuffer(), asDoubleBuffer()
    - 우선 다이렉트 ByteBuffer를 생성하고 호출
    - 초기 다이렉트 ByteBuffer 생성 크기에 따라 저장 용량 결정

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(100);                 //100 개의 byte 값 저장
CharBuffer charBuffer = ByteBuffer.allocateDirect(100).asCharBuffer();  //50 개의 char 값 저장
IntBuffer intBuffer = ByteBuffer.allocateDirect(100).asIntBuffer();     //25 개의 int 값 저장
```

3절. 버퍼

- byte 해석 순서(ByteOrder)
  - 운영체제는 두 바이트 이상을 처리할 때 처리 효율이나 CPU 디자인 상의 문제로 바이트 해석 순서를 정함
  - 데이터를 외부로 보내거나 외부에서 받을 때도 영향 미치기 때문에 바이트 데이터를 다루는 버퍼도 이를 고려해야 !
  - Big endian: 앞 바이트부터 먼저 처리
  - Little endian: 뒤 바이트부터 먼저 처리



3절. 버퍼

❖ byte 해석 순서(ByteOrder)

- 운영 체제가 사용하는 바이트 해석 순서 확인 방법

```
System.out.println("네이티브의 바이트 해석 순서 :"+ByteOrder.nativeOrder());
```

- JVM은 동일한 조건으로 클래스 실행해야 하므로 무조건 Big endian
- Little endian으로 동작하는 운영체제에서 만든 데이터 파일을 Big endian로 동작하는 운영체제에서 읽어 들여야 한다면 ByteOrder 클래스로 데이터 순서를 맞춰야
- 운영체제와 JVM의 바이트 해석 순서가 다를 경우
  - JVM이 운영체제와 데이터 교환 할 때 자동 처리
- 다이렉트 버퍼를 이용할 경우 운영체제의 native I/O를 사용
  - 운영체제의 기본 해석 순서로 JVM의 해석 순서를 맞추는 것이 성능에 도움

3절. 버퍼

❖ Buffer의 위치 속성(position, limit, capacity, mark)

- 사용 전 위치 속성의 개념, 속성이 언제 변경되는지 알고 있어야

속성	설명
position	현재 읽거나 쓰는 위치값이다. 인덱스 값이기 때문에 0 부터 시작하며, 음수를 가지거나 limit 보다 큰값을 가질 수 없다. 만약 position 과 limit 의 값이 같아진다면 더이상 데이터를 쓰거나 읽을 수 없다는 뜻이 된다.
limit	버퍼에서 읽거나 쓸 수 있는 위치의 한계를 나타낸다. 이 값은 capacity 보다 작거나 같은 값을 가지며 음수 값은 가지지 않는다. 인덱스 값이기 때문에 0 부터 시작하며 최초 버퍼를 만들었을때는 capacity 와 같은 값을 가진다.
capacity	버퍼의 최대 데이터 개수(메모리 크기)를 나타낸다. 크기는 음수가 되지 않으며 한번 만들어지면 절대 변하지 않는다. 인덱스 값이 아니라 수량임을 주의하자.
mark	reset() 메소드를 실행했을 때에 돌아오는 위치를 지정하는 인덱스로서 mark() 메소드로 지정할 있다. 주의할 점은 반드시 position 이하의 값으로 지정해 주어야 한다. position 나 limit 의 값이 mark 값보다 작은 경우, mark 는 자동 제거된다. mark 없는 상태에서 reset() 메소드를 호출하면 InvalidMarkException 이 발생한다.

- position, limit, capacity, mark 속성의 크기 관계

$$0 \le mark \le position \le limit \le capacity$$

3절. 버퍼

- 위치 속성을 이해를 위한 쓰기 모드 예 (p.1123~1126)
  - 먼저 배열에 데이터 저장
  - 읽기 모드로 변경하기 위해 flip() 메소드 호출
  - 현재 position의 위치를 기억시키기 위해 mark() 호출
  - position을 mark 위치로 이동하기 위해 reset() 호출
  - 버퍼를 되감아 처음부터 읽기 위해 rewind() 호출
  - 버퍼의 위치 속성을 초기화하기 위해 clear() 호출

3절. 버퍼

❖ Buffer 메소드

- 공통 메소드: Buffer 추상 클래스에 정의된 메소드

리턴타입	메소드(매개변수)	설명
Object	array()	버퍼가 래핑(wrap)한 배열을 리턴
int	arrayOffset()	버퍼의 첫번째 요소가 있는 내부 배열의 인덱스를 리턴
int	capacity()	버퍼의 전체 크기를 리턴.
Buffer	clear()	버퍼의 위치 속성을 초기화(position=0, limit=capacity)
Buffer	flip()	limit 을 position 으로, position 을 0 인덱스로 이동
boolean	hasArray()	버퍼가 래핑(wrap)한 배열을 가지고 있는지 여부
boolean	hasRemaining()	position 과 limit 사이에 요소가 있는지 여부(position<limit)
boolean	isDirect()	운영체제의 버퍼를 사용하는지 여부
boolean	isReadOnly()	버퍼가 읽기 전용인지 여부
int	limit()	limit 위치를 리턴
Buffer	limit(int newLimit)	newLimit 으로 limit 위치를 설정
Buffer	mark()	현재 위치를 mark 로 표시
int	position()	position 위치를 리턴
Buffer	position(int newPosition)	newPosition 으로 position 위치를 설정
int	remaining()	position 과 limit 사이의 요소의 개수
Buffer	reset()	position 을 mark 위치로 이동
Buffer	rewind()	position 을 0 인덱스로 이동

3절. 버퍼

❖ 데이터를 읽고 저장하는 메소드

- 데이터 읽기: get(…)
- 데이터 저장: put(…)
  - Buffer 추상 클래스에는 없고, 각 타입 별 하위 Buffer 클래스가 가짐

- 상대적(Relative) 메소드
  - 현재 위치 속성인 position에서 데이터를 읽고 저장
  - 상대적 get()과 put() 메소드 호출하면 position 값 증가

  - position 값이 limit 값까지 증가한 상태
    - 상대적 get() 사용 - BufferUnderflowException 예외 발생
    - 상대적 put() 사용 - BufferOverflowException 예외 발생

3절. 버퍼

❖ 데이터를 읽고 저장하는 메소드

- 절대적(Absolute) 메소드
  - position과 상관없이 주어진 인덱스에서 데이터 읽고 저장
  - 절대적 get()과 put() 메소드를 호출하면 position의 값은 증가되지 않음

- 상대적과 절대적 메소드 구분 방법
  - 상대적 메소드: index 매개값이 없는 메소드
  - 절대적 메소드: index 매개값이 있는 메소드

- 상세 메소드는 1127~1128 페이지 참조

3절. 버퍼

❖ 버퍼 예외의 종류

- 버퍼 예외 발생 주요 원인
  - 버퍼가 다 찼을 때, 데이터를 저장하려는 경우
  - 버퍼에서 더 이상 읽어올 데이터가 없을 때 데이터를 읽으려는 경우

- 버퍼와 관련된 예외 클래스

예외	설명
BufferOverflowException	position 이 limit 에 도달했을 때 put() 호출하면 발생
BufferUnderflowException	position 이 limit 에 도달했을 때 get() 호출하면 발생
InvalidMarkException	mark 가 없는 상태에서 reset() 메소드를 호출하면 발생
ReadOnlyBufferException	읽기 전용 버퍼에서 put() 또는 compact() 메소드를 호출하면 발생

- 데이터 위치 속성 값의 변화 예제 통해 예외 발생 상황 이해
  - P.1129~1132

3절. 버퍼

❖ Buffer 변환

- 채널이 데이터를 저장하고 읽는 버퍼는 모두 ByteBuffer

- 프로그램 목적 맞게 ByteBuffer를 다른 기본 타입 버퍼로 변환 필요
  - P. 1132~1136

  - Ex 1): ByteBuffer에 특정 문자셋으로 인코딩 된 바이트들이 저장
    - 디코딩 된 CharBuffer로 변환 후 문자열을 얻어야

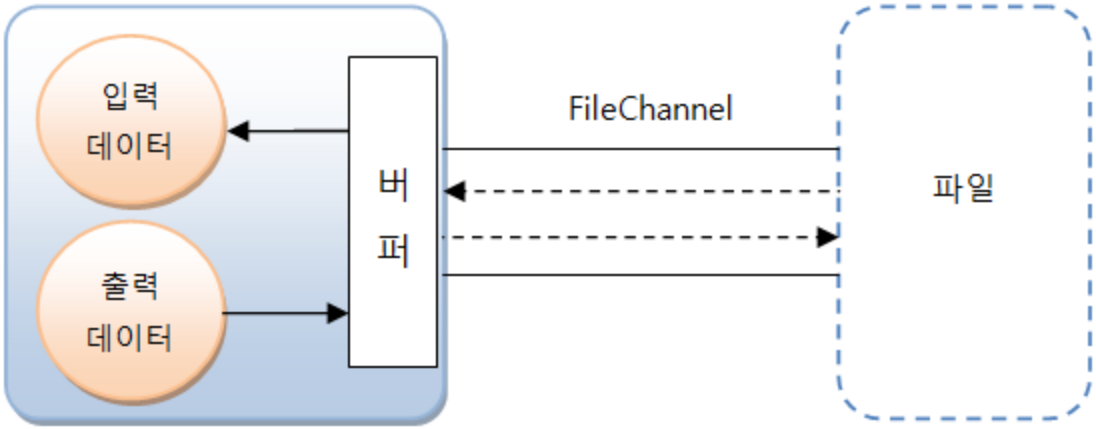
  - Ex 2): ByteBuffer에 정수 바이트들이 저장되어 있을 경우
    - IntBuffer로 변환해서 읽어야

  - Ex 3): CharBuffer와 IntBuffer의 내용을 채널로 출력할 경우
    - ByteBuffer로 변환해야

4절. 파일 채널

❖ 파일 채널(FileChannel)

- 파일 읽기와 쓰기 가능하게 해주는 역할
- 동기화 처리가 되어 있기 때문에 멀티 스레드 환경에서 사용해도 안전



4절. 파일 채널

❖ FileChannel 생성과 닫기

- FileInputStream, FileOutputStream의 getChannel() 메소드 호출
- FileChannel.open()

```
FileChannel fileChannel = FileChannel.open(Path path, OpenOption... options);
```

  - 첫 번째 path 매개값 - 열거나, 생성하고자 하는 파일 경로
  - 두 번째 options 매개값 - 열기 옵션 값
    - StandardOpenOption 의 다음 열거 상수

열거 상수	설명
READ	읽기용으로 파일을 연다.
WRITE	쓰기용으로 파일을 연다.
CREATE	파일이 없다면 새 파일을 생성한다.
CREATE_NEW	새 파일을 만든다. 파일이 이미 있으면 예외와 함께 실패한다.
APPEND	파일 끝에 데이터를 추가한다(WRITE 나 CREATE 와 함께 사용됨)
DELETE_ON_CLOSE	스트림을 닫을 때 파일을 삭제한다(임시파일을 삭제할 때 사용)
TRUNCATE_EXISTING	파일을 0 바이트로 잘라낸다(WRITE 옵션과 함께 사용됨)

- 채널 닫기

```
fileChannel.close();
```

4절. 파일 채널

❖ 파일 쓰기와 읽기

- 파일 쓰기

```
int bytesCount = fileChannel.write(ByteBuffer src);
```

  - 파일에 쓰여지는 바이트는 ByteBuffer의 position 부터 limit 까지

- 파일 읽기

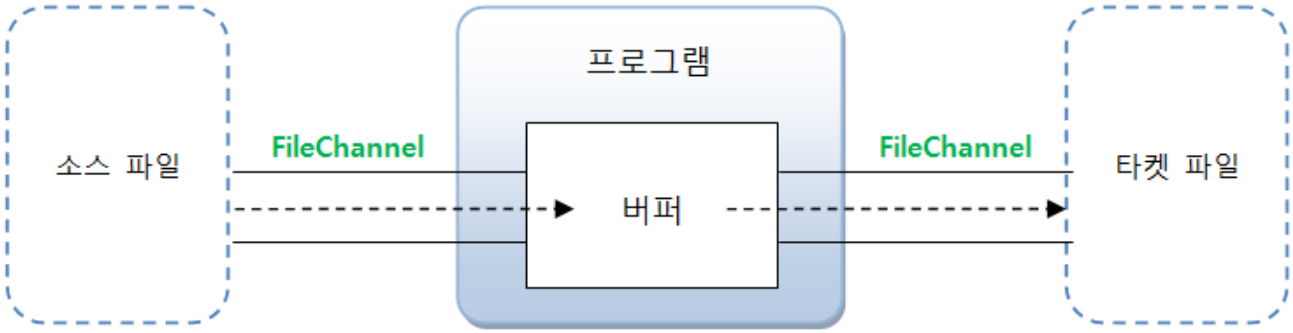
```
int bytesCount = fileChannel.read(ByteBuffer dst);
```

  - 파일에서 읽혀지는 바이트는 ByteBuffer의 position부터 저장
    - 버퍼에 한 바이트를 저장할 때마다 position이 1씩 증가
    - 버퍼에 저장한 마지막 바이트의 위치는 position-1 인덱스까지
    - 한 번 읽을 수 있는 최대 바이트 수는 position부터 ByteBuffer의 capacity까지
  - 리턴값은 파일에서 ByteBuffer로 읽혀진 바이트 수
    - 0~(position-1)까지의 바이트 수
    - 더 이상 읽을 바이트가 없다면 read() 메소드는 -1 리턴

4절. 파일 채널

❖ 파일 복사

- 두 개의 FileChannel 이용



- Files.copy() 메소드 이용

```
Path targetPath = Files.copy(Path source, Path target, CopyOption... options);
```

  - 첫 번째 source 매개값에는 원본 파일의 Path 객체 지정
  - 두 번째 target 매개값에는 타겟 파일의 Path 객체 지정
  - 세 번째 매개값은 StandardCopyOptoin 열거 상수를 목적에 맞게 나열

열거 상수	설명
REPLACE_EXISTING	타겟 파일이 존재하면 대체한다.
COPY_ATTRIBUTES	파일의 속성까지도 복사한다.
NOFOLLOW_LINKS	링크 파일일 경우 링크 파일만 복사하고 링크된 파일은 복사안한다.