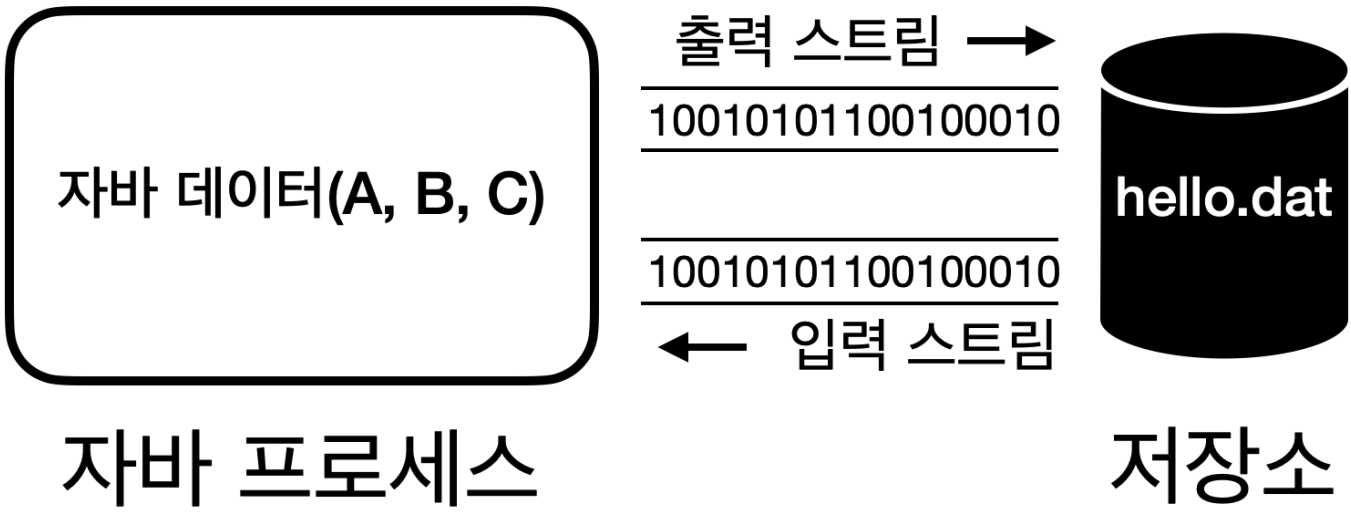
2. I/O 기본1

#0.강의/1.자바로드맵/6.자바-고급2편

스트림 시작1



자바가 가진 데이터를 `hello.dat` 라는 파일에 저장하려면 어떻게 해야할까?

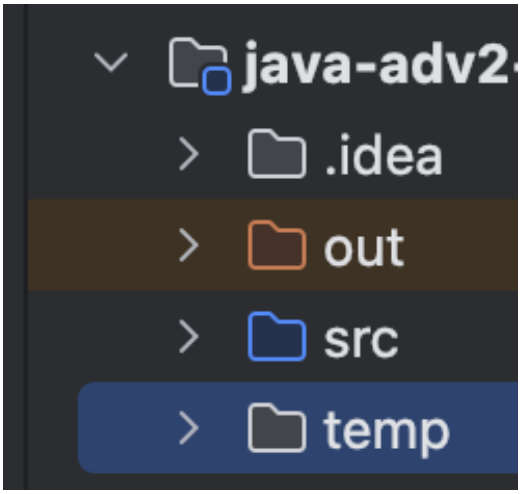
자바 프로세스가 가지고 있는 데이터를 밖으로 보내려면 출력 스트림을 사용하면 되고, 반대로 외부 데이터를 자바 프로세스 안으로 가져오려면 입력 스트림을 사용하면 된다.

참고로 각 스트림은 단방향으로 흐른다.

예제 코드를 통해 확인해보자.

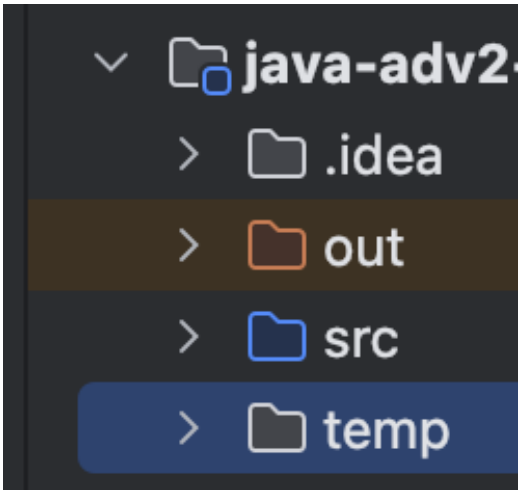
스트림 시작 - 예제1

주의!: 실행 전에 반드시 프로젝트 하위에 temp라는 폴더를 만들어야 한다. src 하위에 만들지 않도록 주의하자




```
package io.start;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class StreamStartMain1 {

    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("temp/hello.dat");
        fos.write(65);
        fos.write(66);
        fos.write(67);
        fos.close();

        FileInputStream fis = new FileInputStream("temp/hello.dat");
        System.out.println(fis.read());
        System.out.println(fis.read());
        System.out.println(fis.read());
        System.out.println(fis.read());
        fis.close();
    }
}
```

주의!: 다시 한번 확인하자. 실행 전에 반드시 프로젝트 하위에 temp라는 폴더를 만들어야 한다. src 하위에 만들지 않도록 주의하자



그렇지 않으면 `java.io.FileNotFoundException` 예외가 발생한다. 자바 코드로 폴더를 만드는 방법은 뒤에서 설명한다.

new FileOutputStream("temp/hello.dat")

- 파일에 데이터를 출력하는 스트림이다.
- 파일이 없으면 파일을 자동으로 만들고, 데이터를 해당 파일에 저장한다.
- 폴더를 만들지는 않기 때문에 폴더는 미리 만들어두어야 한다.

write()

- byte 단위로 값을 출력한다. 여기서는 65, 66, 67을 출력했다.
- 참고로 ASCII 코드 집합에서 65은 A, 66은 B, 67는 C이다.

new FileInputStream("temp/hello.dat")

- 파일에서 데이터를 읽어오는 스트림이다.

read()

- 파일에서 데이터를 byte 단위로 하나씩 읽어온다.
- 순서대로 65, 66, 67을 읽어온다.
- 파일의 끝에 도달해서 더는 읽을 내용이 없다면 -1을 반환한다.
  - 파일의 끝(EOF, End of File)

close()

- 파일에 접근하는 것은 자바 입장에서 외부 자원을 사용하는 것이다. 자바에서 내부 객체는 자동으로 GC가 되지만 외부 자원은 사용 후 반드시 닫아주어야 한다.

실행 결과

```
65
```

```
66
67
-1
```

- 입력한 순서대로 잘 출력되는 것을 확인할 수 있다. 마지막은 파일의 끝에 도달해서 -1이 출력된다.

실행 결과 - temp/hello.dat

```
ABC
```

- `hello.dat`에 분명 byte로 65, 66, 67을 저장했다. 그런데 왜 개발툴이나 텍스트 편집에서 열어보면 ABC라고 보이는 것일까?
- 앞서 자바에서 `read()`로 읽어서 출력한 경우에는 65, 66, 67이 정상 출력되었다.
- 우리가 사용하는 개발툴이나 텍스트 편집기는 UTF-8 또는 MS949 문자 집합을 사용해서 byte 단위의 데이터를 문자로 디코딩해서 보여준다. 따라서 65, 66, 67 byte를 ASCII 문자인 A, B, C로 인식해서 출력한 것이다.

참고: 파일 append 옵션

`FileOutputStream`의 생성자에는 `append`라는 옵션이 있다.

```
new FileOutputStream("temp/hello.dat", true);
```

- `true`: 기존 파일의 끝에 이어서 쓴다.
- `false`: 기존 파일의 데이터를 지우고 처음부터 다시 쓴다. (**기본값**)

스트림 시작 - 예제2

파일의 데이터를 읽을 때 파일의 끝까지 읽어야 한다면 다음과 같이 반복문을 사용하면 된다.

```
package io.start;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class StreamStartMain2 {

    public static void main(String[] args) throws IOException {
```

```
        FileOutputStream fos = new FileOutputStream("temp/hello.dat");
        fos.write(65);
        fos.write(66);
        fos.write(67);
        fos.close();

        FileInputStream fis = new FileInputStream("temp/hello.dat");
        int data;
        while ((data = fis.read()) != -1) {
            System.out.println(data);
        }
        fis.close();
    }
}
```

- 입력 스트림의 `read()` 메서드는 파일의 끝에 도달하면 -1을 반환하다. 따라서 -1을 반환할 때 까지 반복문을 사용하면 파일의 데이터를 모두 읽을 수 있다.

실행 결과

```
65
66
67
```

참고 - read()가 int를 반환하는 이유

이 부분은 크게 중요하지 않은 내용이니 참고만 해두자.

- 부호 없는 바이트 표현:
  - 자바에서 `byte`는 부호 있는 8비트 값(-128 ~ 127)이다.
  - `int`로 반환함으로써 0에서 255까지의 모든 가능한 바이트 값을 부호 없이 표현할 수 있다.
- EOF(End of File) 표시:
  - byte를 표현하려면 256 종류의 값을 모두 사용해야 한다.
  - 자바의 `byte`는 -128에서 127까지 256종류의 값만 가질 수 있어, EOF를 위한 특별한 값을 할당하기 어렵다.
  - `int`는 0~255까지 모든 가능한 바이트 값을 표현하고, 여기에 추가로 -1을 반환하여 스트림의 끝(EOF)을 나타낼 수 있다.
- 참고로 `write()`의 경우도 비슷한 이유로 `int` 타입을 입력 받는다.

스트림 시작2

스트림 시작 - 예제3

이번에는 `byte`를 하나씩 다루는 것이 아니라, `byte[]`을 사용해서 데이터를 원하는 크기 만큼 더 편리하게 저장하고 읽는 방법을 알아보자.

```
package io.start;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Arrays;

public class StreamStartMain3 {

    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("temp/hello.dat");
        byte[] input = {65, 66, 67};
        fos.write(input);
        fos.close();

        FileInputStream fis = new FileInputStream("temp/hello.dat");
        byte[] buffer = new byte[10];
        int readCount = fis.read(buffer, 0, 10);
        System.out.println("readCount = " + readCount);
        System.out.println(Arrays.toString(buffer));
        fis.close();
    }
}
```

실행 결과

```
readCount = 3
[65, 66, 67, 0, 0, 0, 0, 0, 0, 0]
```

출력 스트림

- `write(byte[])` : `byte[]`에 원하는 데이터를 담고 `write()`에 전달하면 해당 데이터를 한 번에 출력할 수

있다.

입력 스트림

- `read(byte[], offset, lentgh)` : `byte[]`을 미리 만들어두고, 만들어둔 `byte[]`에 한 번에 데이터를 읽어올 수 있다.
- `byte[]` : 데이터가 읽혀지는 버퍼
- `offset` : 데이터 기록되는 `byte[]`의 인덱스 시작 위치
- `length` : 읽어올 byte의 최대 길이
- **반환 값**: 버퍼에 읽은 총 바이트 수 여기서는 3byte를 읽었으므로 3이 반환된다. 스트림의 끝에 도달하여 더 이상 데이터가 없는 경우 -1을 반환

read(byte[])

- 참고로 `offset`, `length`를 생략한 `read(byte[])` 메서드도 있다. 이 메서드는 다음 값을 가진다.
  - `offset` : 0
  - `length` : `byte[].length`

스트림 시작 - 예제4

모든 byte 한 번에 읽기

```
package io.start;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Arrays;

public class StreamStartMain4 {

    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream("temp/hello.dat");
        byte[] input = {65, 66, 67};
        fos.write(input);
        fos.close();

        FileInputStream fis = new FileInputStream("temp/hello.dat");
        byte[] readBytes = fis.readAllBytes();
        System.out.println(Arrays.toString(readBytes));
```

```
            fis.close();
        }
    }
```

- `readAllBytes()`를 사용하면 스트림이 끝날 때 까지(파일의 끝에 도달할 때 까지) 모든 데이터를 한 번에 읽어올 수 있다.

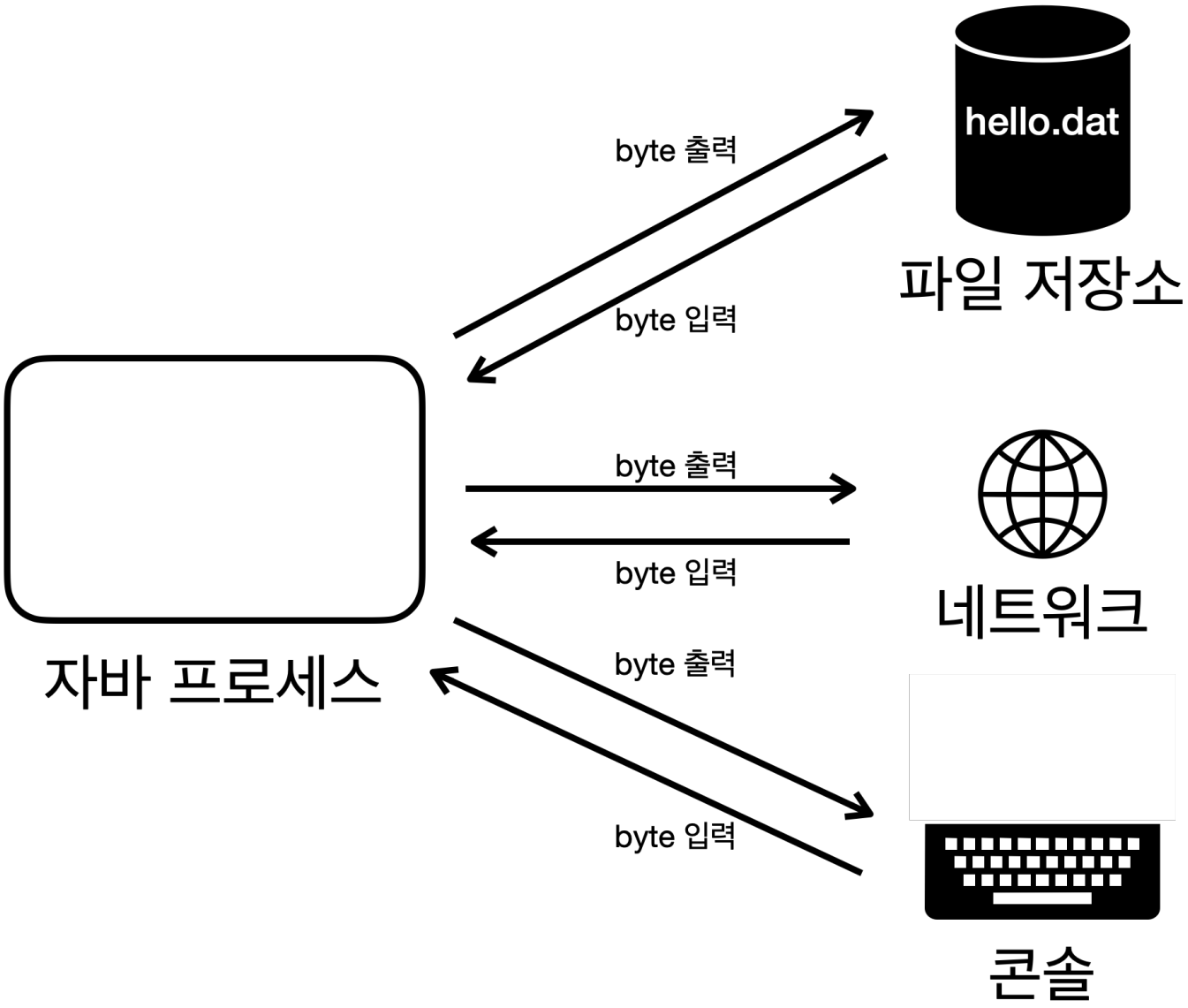
실행 결과

```
[65, 66, 67]
```

부분으로 나누어 읽기 vs 전체 읽기

- `read(byte[], offset, lentgh)`
  - 스트림의 내용을 부분적으로 읽거나, 읽은 내용을 처리하면서 스트림을 계속해서 읽어야 할 경우에 적합하다.
  - 메모리 사용량을 제어할 수 있다.
  - 예시) 파일이나 스트림에서 일정한 크기의 데이터를 반복적으로 읽어야 할 때 유용하다. 예를 들어, 대용량 파일을 처리할 때, 한 번에 메모리에 로드하기보다는 이 메서드를 사용하여 파일을 조각조각 읽어들일 수 있다.
  - 100M의 파일을 1M 단위로 나누어 읽고 처리하는 방식을 사용하면 한 번에 최대 1M의 메모리만 사용한다.
- `readAllBytes()`
  - 한 번의 호출로 모든 데이터를 읽을 수 있어 편리하다.
  - 작은 파일이나 메모리에 모든 내용을 올려서 처리해야 하는 경우에 적합하다.
  - 메모리 사용량을 제어할 수 없다.
  - 큰 파일의 경우 OutOfMemoryError가 발생할 수 있다.

InputStream, OutputStream



현대의 컴퓨터는 대부분 byte 단위로 데이터를 주고 받는다. 참고로 bit 단위는 너무 작기 때문에 byte 단위를 기본으로 사용한다.

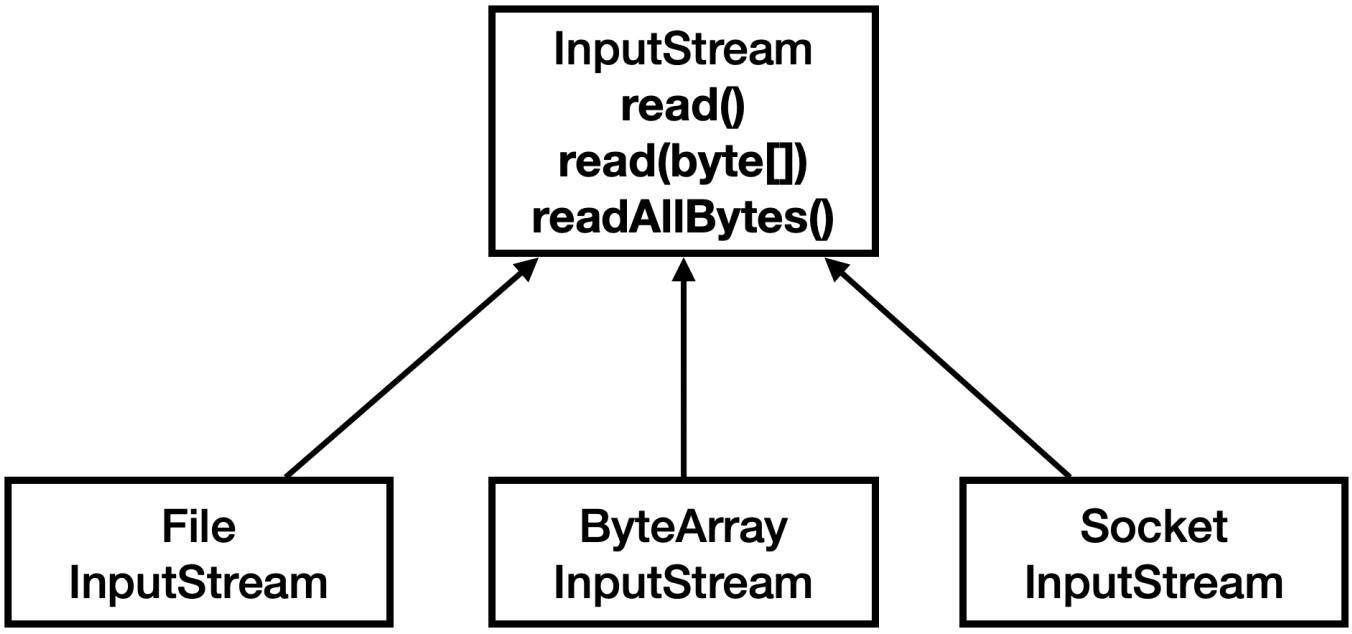
이렇게 데이터를 주고 받는 것을 **Input/Output(I/O)**라 한다.

자바 내부에 있는 데이터를 외부에 있는 파일에 저장하거나, 네트워크를 통해 전송하거나 콘솔에 출력할 때 모두 byte 단위로 데이터를 주고 받는다.

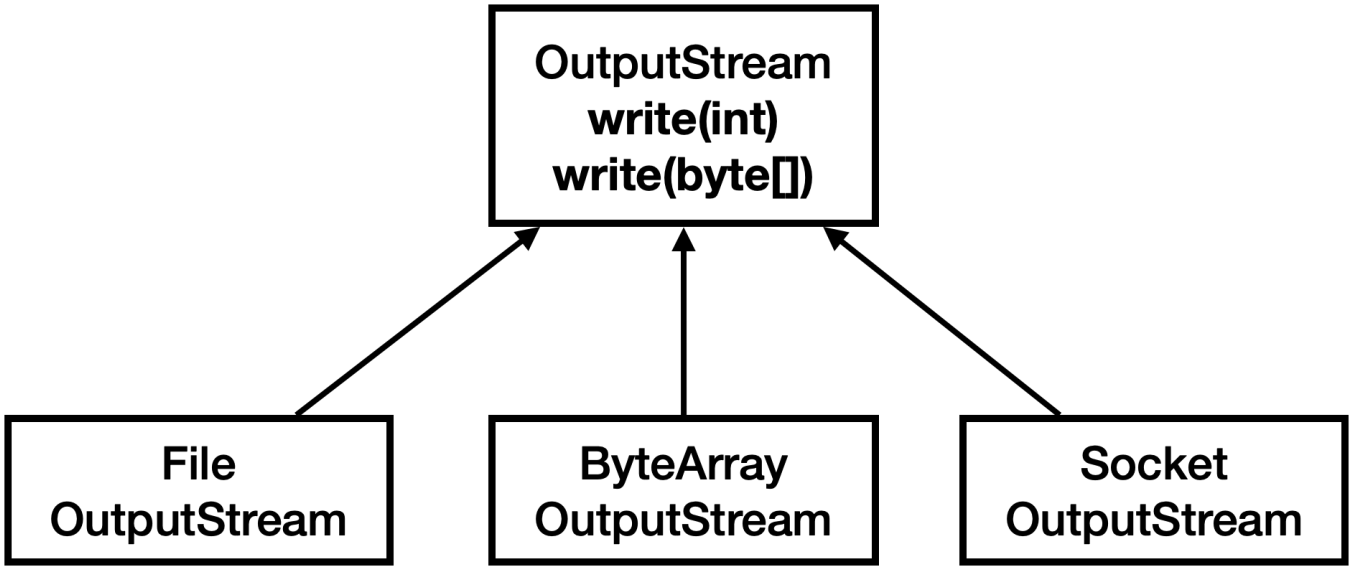
만약 파일, 네트워크, 콘솔 각각 데이터를 주고 받는 방식이 다르다면 상당히 불편할 것이다.

또한 파일에 저장하던 내용을 네트워크에 전달하거나 콘솔에 출력하도록 변경할 때 너무 많은 코드를 변경해야 할 수 있다.

이런 문제를 해결하기 위해 자바는 `InputStream`, `OutputStream`이라는 기본 추상 클래스를 제공한다.



- `InputStream`과 상속 클래스
- `read()`, `read(byte[])`, `readAllBytes()` 제공



- `OutputStream`과 상속 클래스
- `write(int)`, `write(byte[])` 제공

스트림을 사용하면 파일을 사용하든, 소켓을 통해 네트워크를 사용하든 모두 일관된 방식으로 데이터를 주고 받을 수 있다. 그리고 수 많은 기본 구현 클래스들도 제공한다.
물론 각각의 구현 클래스들은 자신에게 맞는 추가 기능도 함께 제공한다.

파일에 사용하는 `FileInputStream`, `FileOutputStream`은 앞서 알아보았다. 네트워크 관련 스트림은 이후에 네트워크를 다룰 때 알아보자. 여기서는 메모리와 콘솔에 사용하는 스트림을 사용해보자.

메모리 스트림

```
package io.start;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.util.Arrays;

public class ByteArrayStreamMain {

    public static void main(String[] args) throws IOException {
        byte[] input = {1, 2, 3};

        // 메모리에 쓰기
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        baos.write(input);

        // 메모리에서 읽기
        ByteArrayInputStream bais = new
ByteArrayInputStream(baos.toByteArray());
        byte[] bytes = bais.readAllBytes();
        System.out.println(Arrays.toString(bytes));
    }
}
```

실행 결과

```
[1, 2, 3]
```

`ByteArrayOutputStream`, `ByteArrayInputStream`을 사용하면 메모리에 스트림을 쓰고 읽을 수 있다.
이 클래스들은 `OutputStream`, `InputStream`을 상속받았기 때문에 부모의 기능을 모두 사용할 수 있다.
코드를 보면 파일 입출력과 매우 비슷한 것을 확인할 수 있다.
참고로 메모리에 어떤 데이터를 저장하고 읽을 때는 컬렉션이나 배열을 사용하면 되기 때문에, 이 기능은 잘 사용하지 않는다. 주로 스트림을 간단하게 테스트 하거나 스트림의 데이터를 확인하는 용도로 사용한다.

콘솔 스트림

```java
package io.start;

import java.io.IOException;
import java.io.PrintStream;

import static java.nio.charset.StandardCharsets.UTF_8;

public class PrintStreamMain {
    public static void main(String[] args) throws IOException {
        PrintStream printStream = System.out;

        byte[] bytes = "Hello!\n".getBytes(UTF_8);
        printStream.write(bytes);
        printStream.println("Print!");
    }
}
```

실행 결과

```
Hello!
Print!
```

우리가 자주 사용했던 `System.out`이 사실은 `PrintStream`이다. 이 스트림은 `OutputStream`를 상속받는다. 이 스트림은 자바가 시작될 때 자동으로 만들어진다. 따라서 우리가 직접 생성하지 않는다.

- `write(byte[])` : `OutputStream` 부모 클래스가 제공하는 기능이다.
- `println(String)` : `PrintStream`이 자체적으로 제공하는 추가 기능이다.

정리

`InputStream`과 `OutputStream`이 다양한 스트림들을 추상화하고 기본 기능에 대한 표준을 잡아둔 덕분에 개발자는 편리하게 입출력 작업을 수행할 수 있다. 이러한 추상화의 장점은 다음과 같다.

- **일관성**: 모든 종류의 입출력 작업에 대해 동일한 인터페이스(여기서는 부모의 메서드)를 사용할 수 있어, 코드의 일관성이 유지된다.
- **유연성**: 실제 데이터 소스나 목적지가 무엇인지에 관계없이 동일한 방식으로 코드를 작성할 수 있다. 예를 들어, 파일, 네트워크, 메모리 등 다양한 소스에 대해 동일한 메서드를 사용할 수 있다.
- **확장성**: 새로운 유형의 입출력 스트림을 쉽게 추가할 수 있다.
- **재사용성**: 다양한 스트림 클래스들을 조합하여 복잡한 입출력 작업을 수행할 수 있다. 예를 들어

`BufferedInputStream`을 사용하여 성능을 향상시키거나, `DataInputStream`을 사용하여 기본 데이터 타입을 쉽게 읽을 수 있다. 이 부분은 뒤에서 설명한다.

- **에러 처리**: 표준화된 예외 처리 메커니즘을 통해 일관된 방식으로 오류를 처리할 수 있다.

참고로 `InputStream`, `OutputStream`은 추상 클래스이다. 자바 1.0부터 제공되고, 일부 작동하는 코드도 들어있기 때문에 인터페이스가 아니라 추상 클래스로 제공된다.

파일 입출력과 성능 최적화1 - 하나씩 쓰기

파일을 효과적으로 더 빨리 읽고 쓰는 방법에 대해서 알아보자.

먼저 예제에서 공통으로 사용할 상수들을 정의하자.

```
package io.buffered;

public class BufferedConst {
    public static final String FILE_NAME = "temp/buffered.dat";
    public static final int FILE_SIZE = 10 * 1024 * 1024; // 10MB
    public static final int BUFFER_SIZE = 8192; // 8KB
}
```

- `FILE_NAME`: `temp/buffered.dat`라는 파일을 만들 예정이다.
- `FILE_SIZE`: 파일의 크기는 10MB이다.
- `BUFFER_SIZE`는 뒤에서 설명한다.

예제1 - 쓰기

먼저 가장 단순한 `FileOutputStream`의 `write()`를 사용해서 1byte씩 파일을 저장해보자.

그리고 10MB 파일을 만드는데 걸리는 시간을 확인해보자.

```
package io.buffered;

import java.io.FileOutputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.FILE_NAME;
import static io.buffered.BufferedConst.FILE_SIZE;
```

```
public class CreateFileV1 {

    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream(FILE_NAME);
        long startTime = System.currentTimeMillis();

        for (int i = 0; i < FILE_SIZE; i++) {
            fos.write(1);
        }
        fos.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File created: " + FILE_NAME);
        System.out.println("File size: " + FILE_SIZE / 1024 / 1024 + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }

}
```

- `fos.write(1)` : 파일의 내용은 중요하지 않기 때문에 여기서는 단순히 1이라는 값을 반복하며 계속 저장한다.
  - 한 번 호출에 1byte가 만들어진다.
  - 이 메서드를 약 1000만번(10 * 1024 * 1024) 호출하면 10MB의 파일이 만들어진다.

주의!

> 실행시 시스템에 따라서 1분 이상 걸리는 경우도 있으니 결과가 나올 때 까지 참고 기다리자.

실행 결과

```
File created: temp/buffered.dat
File size: 10MB
Time taken: 14092ms
```

- 실행을 하면 결과를 보는데 상당히 오랜 시간이 걸린다.
- M2 맥북 프로에서 총 실행 시간은 약 14초 정도가 걸렸다.

예제1 - 읽기

```
package io.buffered;
```

```java
import java.io.FileInputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.FILE_NAME;

public class ReadFileV1 {

    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream(FILE_NAME);
        long startTime = System.currentTimeMillis();

        int fileSize = 0;
        int data;
        while ((data = fis.read()) != -1) {
            fileSize++;
        }
        fis.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File name: " + FILE_NAME);
        System.out.println("File size: " + (fileSize / 1024 / 1024) + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

- `fis.read()`를 사용해서 앞서 만든 파일에서 1byte씩 데이터를 읽는다.
- 파일의 크기가 10MB이므로 `fis.read()` 메서드를 약 1000만번(10 * 1024 * 1024) 호출한다.

주의!

> 실행시 시스템에 따라서 1분 이상 걸리는 경우도 있으니 결과가 나올 때 까지 참고 기다리자.

실행 결과

```
File name: temp/buffered.dat
File size: 10MB
Time taken: 5003ms
```

- 실행을 하면 결과를 보는데 상당히 오랜 시간이 걸린다.
- M2 맥북 프로에서 총 실행 시간은 약 5초 정도가 걸렸다.

정리

10MB 파일 하나를 쓰는데 14초, 읽는데 5초라는 매우 오랜 시간이 걸렸다.
이렇게 오래 걸린 이유는 자바에서 1byte씩 디스크에 데이터를 전달하기 때문이다. 디스크는 1byte의 데이터를 받아서 1byte의 데이터를 쓴다. 이 과정을 무려 1000만 번 반복하는 것이다.

더 자세히 설명하면 다음 2가지 이유로 느려진다.

- `write()`나 `read()`를 호출할 때마다 OS의 시스템 콜을 통해 파일을 읽거나 쓰는 명령어를 전달한다. 이러한 시스템 콜은 상대적으로 무거운 작업이다.
- HDD, SSD 같은 장치들도 하나의 데이터를 읽고 쓸 때마다 필요한 시간이 있다. HDD의 경우 더욱 느린데, 물리적으로 디스크의 회전이 필요하다.
- 이러한 무거운 작업을 무려 1000만 번 반복한다.

비유를 하자면 창고에서 마트까지 상품을 전달해야 하는데, 화물차에 한 번에 하나의 물건만 가지고 이동하는 것이다.
화물차가 무려 1000만 번 이동을 반복해야 10MB의 파일이 만들어진다.
물론 반대로 데이터를 읽어 들일 때도 마찬가지이다.

이런 문제를 해결하려면 화물차에 더 많은 상품을 담아서 보내면 된다.

참고

이렇게 자바에서 운영 체제를 통해 디스크에 1byte씩 전달하면, 운영 체제나 하드웨어 레벨에서 여러가지 최적화가 발생한다. 따라서 실제로 디스크에 1byte씩 계속 쓰는 것은 아니다. 그렇다면 훨씬 더 느렸을 것이다. 하지만, 자바에서 1바이트씩 write()나 read()를 호출할 때마다 운영 체제로의 시스템 콜이 발생하고, 이 시스템 콜 자체가 상당한 오버헤드를 유발한다. 운영 체제와 하드웨어가 어느 정도 최적화를 제공하더라도, 자주 발생하는 시스템 콜로 인한 성능 저하는 피할 수 없다. 결국 자바에서 read(), write() 호출 횟수를 줄여서 시스템 콜 횟수도 줄여야 한다.

파일 입출력과 성능 최적화2 - 버퍼 활용

이번에는 1byte씩 데이터를 하나씩 전달하는 것이 아니라 `byte[]`을 통해 배열에 담아서 한 번에 여러 byte를 전달해보자.

예제2 - 쓰기

```
package io.buffered;
```

```java
import java.io.FileOutputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.*;

public class CreateFileV2 {
    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream(FILE_NAME);
        long startTime = System.currentTimeMillis();

        byte[] buffer = new byte[BUFFER_SIZE];
        int bufferIndex = 0;

        for (int i = 0; i < FILE_SIZE; i++) {
            buffer[bufferIndex++] = 1;

            // 버퍼가 가득 차면 쓰고, 버퍼를 비운다.
            if (bufferIndex == BUFFER_SIZE) {
                fos.write(buffer);
                bufferIndex = 0;
            }
        }

        // 끝 부분에 오면 버퍼가 가득차지 않고 남아있을 수 있다. 버퍼에 남은 부분 쓰기
        if (bufferIndex > 0) {
            fos.write(buffer, 0, bufferIndex);
        }

        fos.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File created: " + FILE_NAME);
        System.out.println("File size: " + FILE_SIZE / 1024 / 1024 + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

- 데이터를 먼저 `buffer`라는 `byte[]`에 담아둔다.
  - 이렇게 데이터를 모아서 전달하거나 모아서 전달받는 용도로 사용하는 것을 버퍼라 한다.
- 여기서는 `BUFFER_SIZE`만큼 데이터를 모아서 `write()`를 호출한다.
  - 예를 들어 `BUFFER_SIZE`가 10이라면 10만큼 모이면 `write()`를 호출해서 `10byte`를 한 번에 스트림

에 전달한다.

실행 결과

```
File created: temp/buffered.dat
File size: 10MB
Time taken: 14ms
```

- 실행 결과의 `BUFFER_SIZE`는 8192(8KB)이다.
- 실행 결과를 보면 이전 예제의 쓰기 결과인 14초 보다 약 1000배 정도 빠른 것을 확인할 수 있다.
  - 실행 시간은 시스템 환경에 따라 달라진다.

버퍼의 크기에 따른 쓰기 성능

`BUFFER_SIZE`에 따른 쓰기 성능

- **1**: 14368ms
- **2**: 7474ms
- **3**: 4829ms
- **10**: 1692ms
- **100**: 180ms
- **1000**: 28ms
- **2000**: 23ms
- **4000**: 16ms
- **8000**: 13ms
- **80000**: 12ms

많은 데이터를 한 번에 전달하면 성능을 최적화 할 수 있다. 이렇게 되면 시스템 콜도 줄어들고, HDD, SSD 같은 장치들의 작동 횟수도 줄어든다. 예를 들어 버퍼의 크기를 1 → 2로 변경하면 시스템 콜 횟수는 절반으로 줄어든다.
그런데 버퍼의 크기가 커진다고 해서 속도가 계속 줄어들지는 않는다.
왜냐하면 디스크나 파일 시스템에서 데이터를 읽고 쓰는 기본 단위가 보통 4KB 또는 8KB이기 때문이다.

- 4KB (**4096** byte)
- 8KB (**8192** byte)

결국 버퍼에 많은 데이터를 담아서 보내도 디스크나 파일 시스템에서 해당 단위로 나누어 저장하기 때문에 효율에는 한계가 있다.
따라서 버퍼의 크기는 보통 4KB, 8KB 정도로 잡는 것이 효율적이다.

예제2 - 읽기

```
package io.buffered;

import java.io.FileInputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.BUFFER_SIZE;
import static io.buffered.BufferedConst.FILE_NAME;

public class ReadFileV2 {

    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream(FILE_NAME);
        long startTime = System.currentTimeMillis();

        byte[] buffer = new byte[BUFFER_SIZE];
        int fileSize = 0;
        int size;
        while ((size = fis.read(buffer)) != -1) {
            fileSize += size;
        }
        fis.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File name: " + FILE_NAME);
        System.out.println("File size: " + (fileSize / 1024 / 1024) + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
File name: temp/buffered.dat
File size: 10MB
Time taken: 5ms
```

- 실행 결과의 `BUFFER_SIZE`는 8192(8KB)이다.
- 읽기의 경우에도 버퍼를 사용하면 약 1000배 정도의 성능 향상을 확인할 수 있다.
  - 실행 시간은 시스템 환경에 따라 달라진다.

버퍼를 사용하면 큰 성능 향상이 있다. 하지만 직접 버퍼를 만들고 관리해야 하는 번거로운 단점이 있다.
예제1과 같이 버퍼를 사용하지 않는 단순한 코드를 유지하면서, 버퍼를 사용할 때와 같은 성능의 이점을 누리는 방법은 없을까?

파일 입출력과 성능 최적화3 - Buffered 스트림 쓰기

`BufferedOutputStream`은 버퍼 기능을 내부에서 대신 처리해준다. 따라서 단순한 코드를 유지하면서 버퍼를 사용하는 이점도 함께 누릴 수 있다.
`BufferedOutputStream`을 사용해서 코드를 작성해보자.

예제3 - 쓰기

```
package io.buffered;

import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.*;

public class CreateFileV3 {

    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream(FILE_NAME);
        BufferedOutputStream bos = new BufferedOutputStream(fos, BUFFER_SIZE);
        long startTime = System.currentTimeMillis();

        for (int i = 0; i < FILE_SIZE; i++) {
            bos.write(1);
        }
        bos.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File created: " + FILE_NAME);
        System.out.println("File size: " + FILE_SIZE / 1024 / 1024 + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
```

```
        }
    }
```

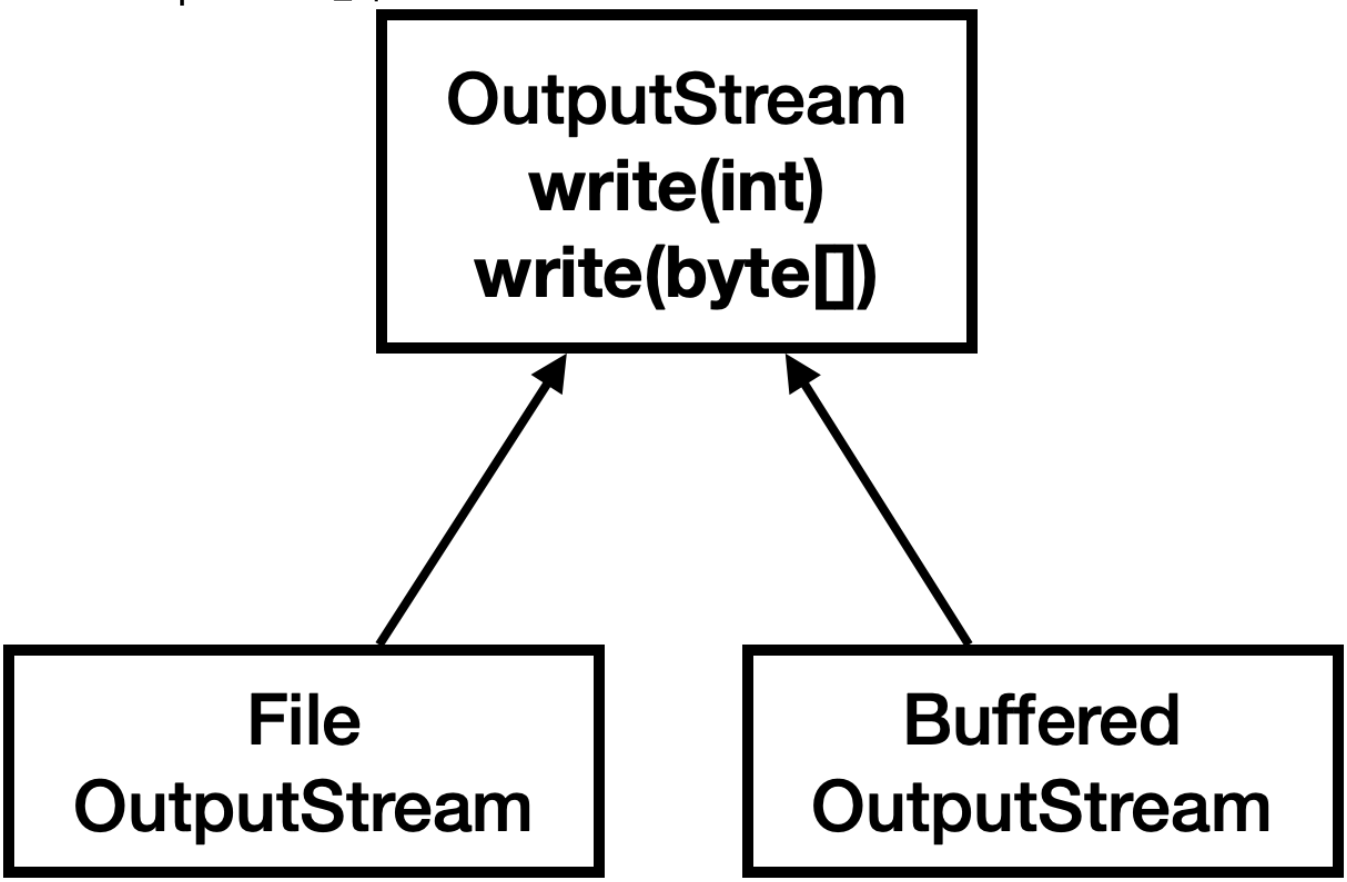
- `BufferedOutputStream`은 내부에서 단순히 버퍼 기능만 제공한다. 따라서 반드시 대상 `OutputStream`이 있어야 한다.
- 여기서는 `FileOutputStream` 객체를 생성자에 전달한다.
- 추가로 사용할 버퍼의 크기도 함께 전달할 수 있다.
- 코드를 보면 버퍼를 위한 `byte[]`을 직접 다루지 않고, 마치 예제1과 같이 단순하게 코드를 작성할 수 있다.

실행 결과

```
File created: temp/buffered.dat
File size: 10MB
Time taken: 102ms
```

- 성능도 예제1의 14초 보다 140배 빠른 0.1초에 처리되었다.
- 참고로 성능이 예제2보다는 다소 떨어지는데 그 이유는 뒤에서 설명하겠다.

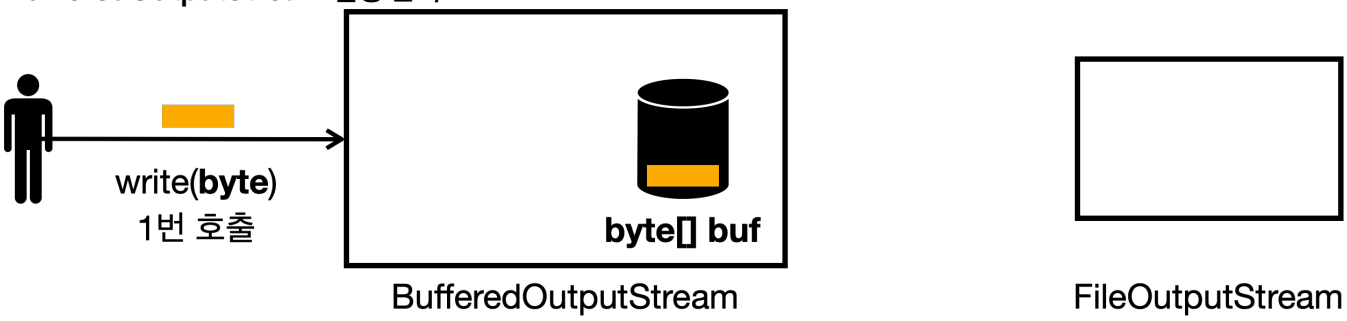
BufferedOutputStream 분석



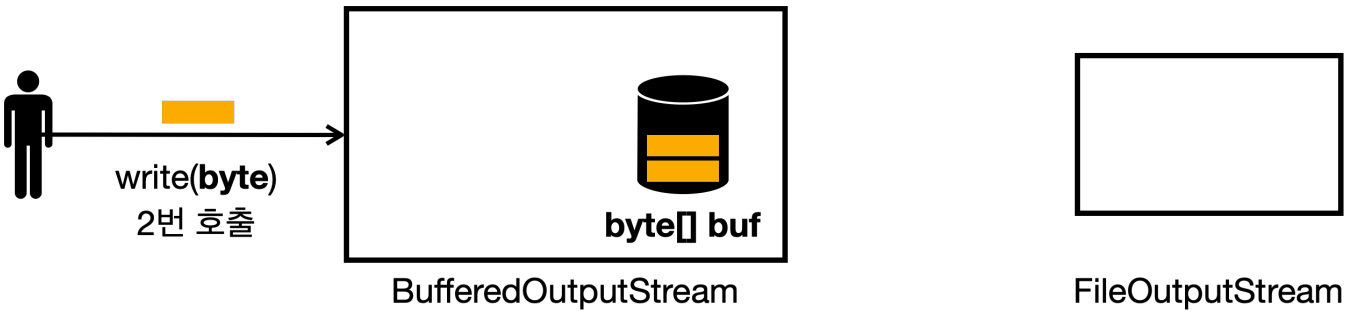
- `BufferdOutputStream`은 `OutputStream`을 상속받는다. 따라서 개발자 입장에서 보면 `OutputStream`

과 같은 기능을 그대로 사용할 수 있다. 예제에서는 `write()`를 사용했다.

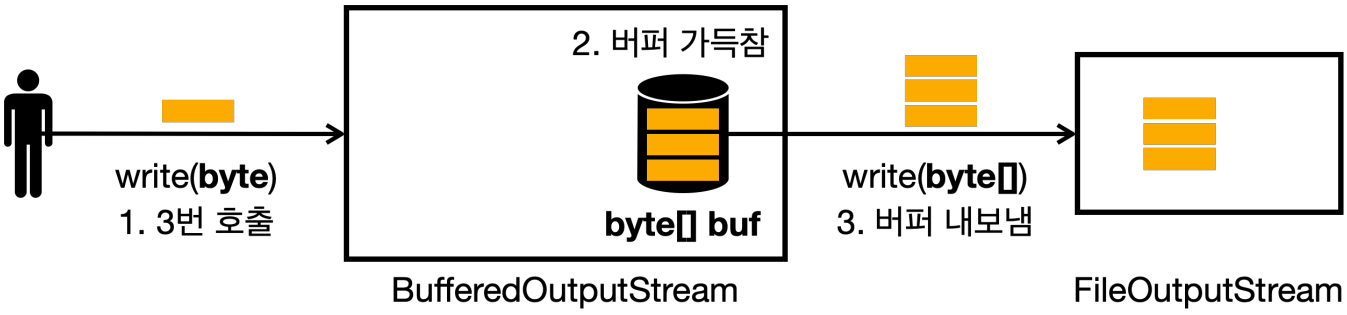
BufferedOutputStream 실행 순서



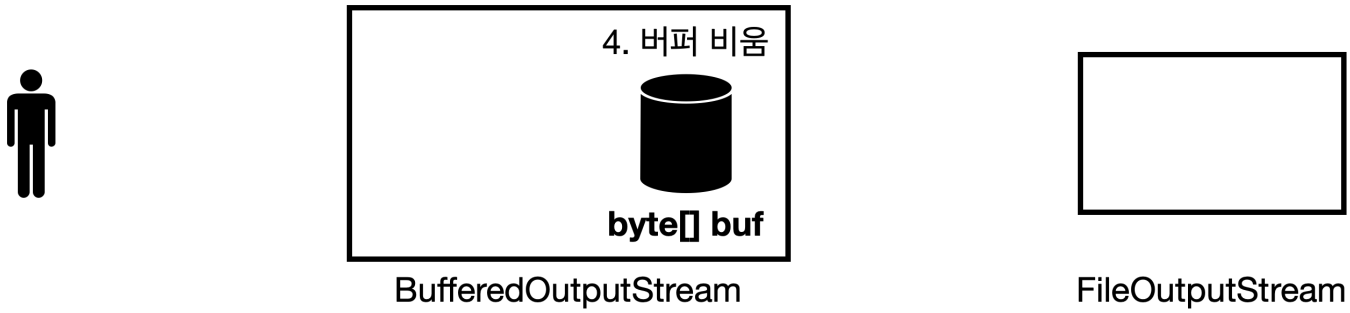
- `BufferedOutputStream`은 내부에 `byte[] buf`라는 버퍼를 가지고 있다.
- 여기서 버퍼의 크기는 3이라고 가정하겠다.
- `BufferedOutputStream`에 `write(byte)`를 통해 `byte` 하나를 전달하면 `byte[] buf`에 보관된다.
  - 참고로 실제로는 `write(int)` 타입이지만 쉽게 설명하기 위해 `write(byte)`로 그려두었다.



`write(byte)`를 2번 호출했다. 아직 버퍼가 가득차지 않았다.



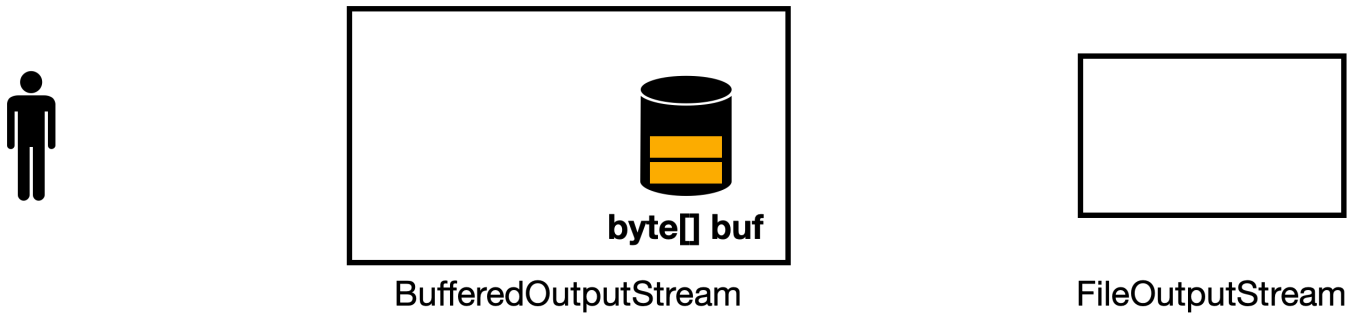
- `write(byte)`를 3번 호출하면 버퍼가 가득 찬다.
- 버퍼가 가득차면 `FileOutputStream`에 있는 `write(byte[])` 메서드를 호출한다.
  - 참고로 `BufferedOutputStream`의 생성자에서 `FileOutputStream`, `fos`를 전달해두었다.
- `FileOutputStream`의 `write(byte[])`을 호출하면, 전달된 모든 `byte[]`을 시스템 콜로 OS에 전달한다.



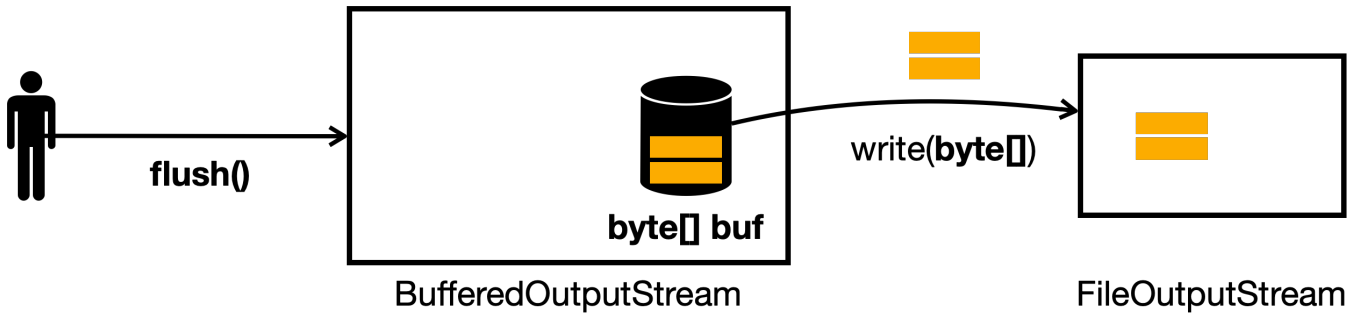
- 버퍼의 데이터를 모두 전달했기 때문에 버퍼의 내용을 비운다.
- 이후에 `write(byte)`가 호출되면 다시 버퍼를 채우는 식으로 반복한다.

flush()

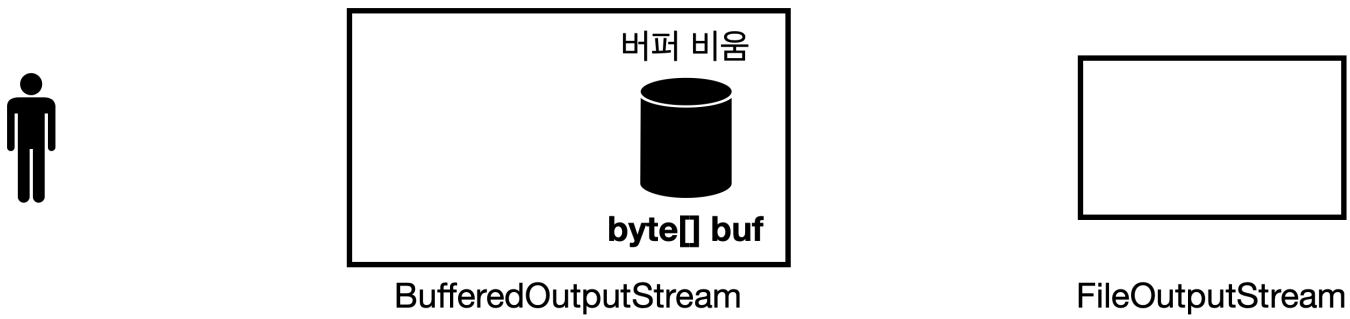
버퍼가 다 차지 않아도 버퍼에 남아있는 데이터를 전달하려면 `flush()`라는 메서드를 호출하면 된다.



- 버퍼에 2개의 데이터가 남아있음



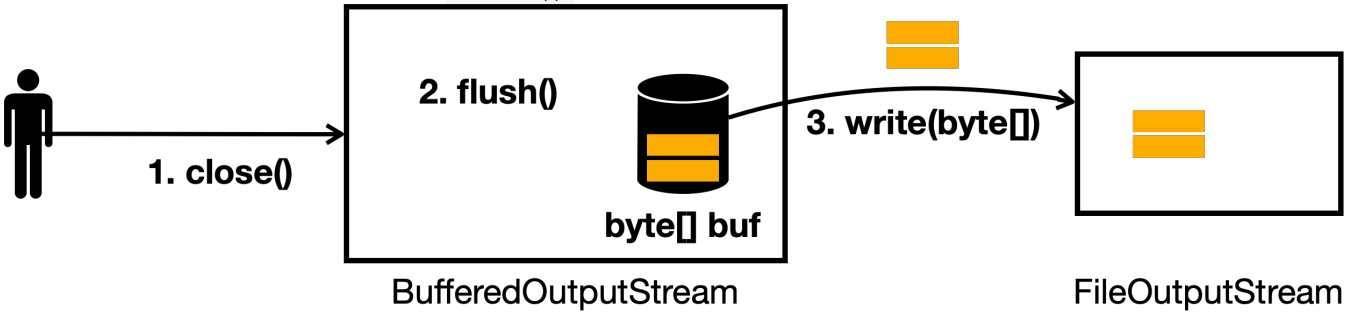
- `flush()` 호출
- 버퍼에 남은 데이터를 전달함



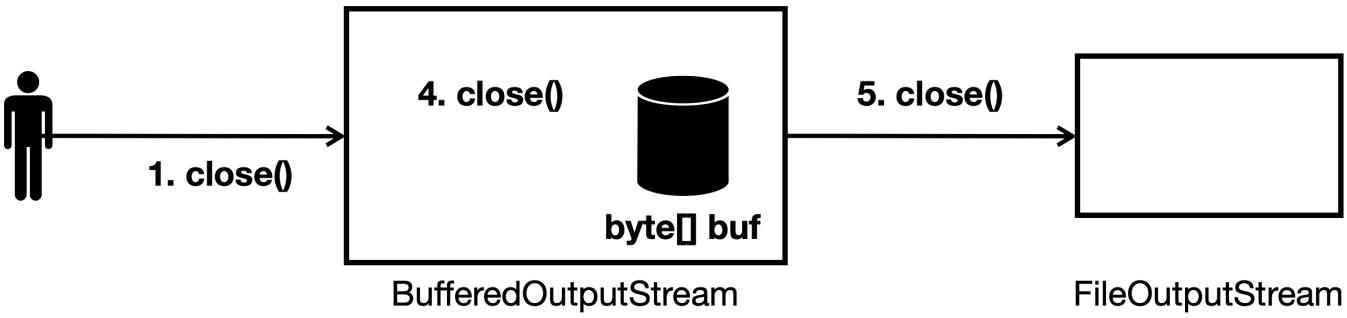
- 데이터를 전달하고 버퍼를 비움

close()

만약 버퍼에 데이터가 남아있는 상태로 `close()` 를 호출하면 어떻게 될까?

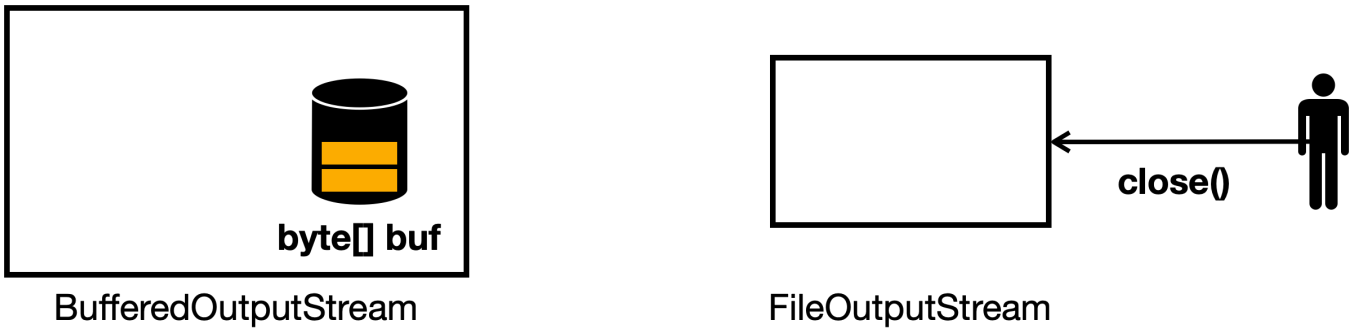


- `BufferedOutputStream` 을 `close()` 로 닫으면 먼저 내부에서 `flush()` 를 호출한다. 따라서 버퍼에 남아있는 데이터를 모두 전달하고 비운다.
- 따라서 `close()` 를 호출해도 남은 데이터를 안전하게 저장할 수 있다.



- 버퍼가 비워지고 나면 `close()` 로 `BufferedOutputStream` 의 자원을 정리한다.
- 그리고 나서 다음 연결된 스트림의 `close()` 를 호출한다. 여기서는 `FileOutputStream` 의 자원이 정리된다.
- 여기서 핵심은 `close()` 를 호출하면 `close()` 가 연쇄적으로 호출된다는 점이다. 따라서 마지막에 연결한 `BufferedOutputStream` 만 닫아주면 된다.

주의! - 반드시 마지막에 연결한 스트림을 닫아야 한다.



- 만약 `BufferedOutputStream` 을 닫지 않고, `FileOutputStream` 만 직접 닫으면 어떻게 될까?
- 이 경우 `BufferdOutputStream` 의 `flush()` 도 호출되지 않고, 자원도 정리되지 않는다. 따라서 남은

`byte`가 버퍼에 남아있게 되고, 파일에 저장되지 않는 심각한 문제가 발생한다.

- 따라서 지금과 같이 스트림을 연결해서 사용하는 경우에는 마지막에 연결한 스트림을 반드시 닫아주어야 한다.
  - 마지막에 연결한 스트림만 닫아주면 연쇄적으로 `close()`가 호출된다.

기본 스트림, 보조 스트림

- `FileOutputStream`과 같이 단독으로 사용할 수 있는 스트림을 기본 스트림이라 한다.
- `BufferedOutputStream`과 같이 단독으로 사용할 수 없고, 보조 기능을 제공하는 스트림을 보조 스트림이라 한다.

`BufferedOutputStream`은 `FileOutputStream`에 버퍼라는 보조 기능을 제공한다.
`BufferedOutputStream`의 생성자를 보면 알겠지만 반드시 `FileOutputStream` 같은 대상 `OutputStream`이 있어야 한다.

```
public BufferedOutputStream(OutputStream out) { ... }
public BufferedOutputStream(OutputStream out, int size) { ... }
```

- `BufferedOutputStream`은 버퍼라는 보조 기능을 제공한다. 그렇다면 누구에게 보조 기능을 제공할지 대상을 반드시 전달해야 한다.

정리

- `BufferedOutputStream`은 버퍼 기능을 제공하는 보조 스트림이다.
- `BufferedOutputStream`도 `OutputStream`의 자식이기 때문에 `OutputStream`의 기능을 그대로 사용할 수 있다.
  - 물론 대부분의 기능은 재정의 된다. `write()`의 경우 먼저 버퍼에 쌓도록 재정의 된다.
- 버퍼의 크기만큼 데이터를 모아서 전달하기 때문에 빠른 속도로 데이터를 처리할 수 있다.

파일 입출력과 성능 최적화4 - Buffered 스트림 읽기

예제3 - 읽기

```
package io.buffered;
```

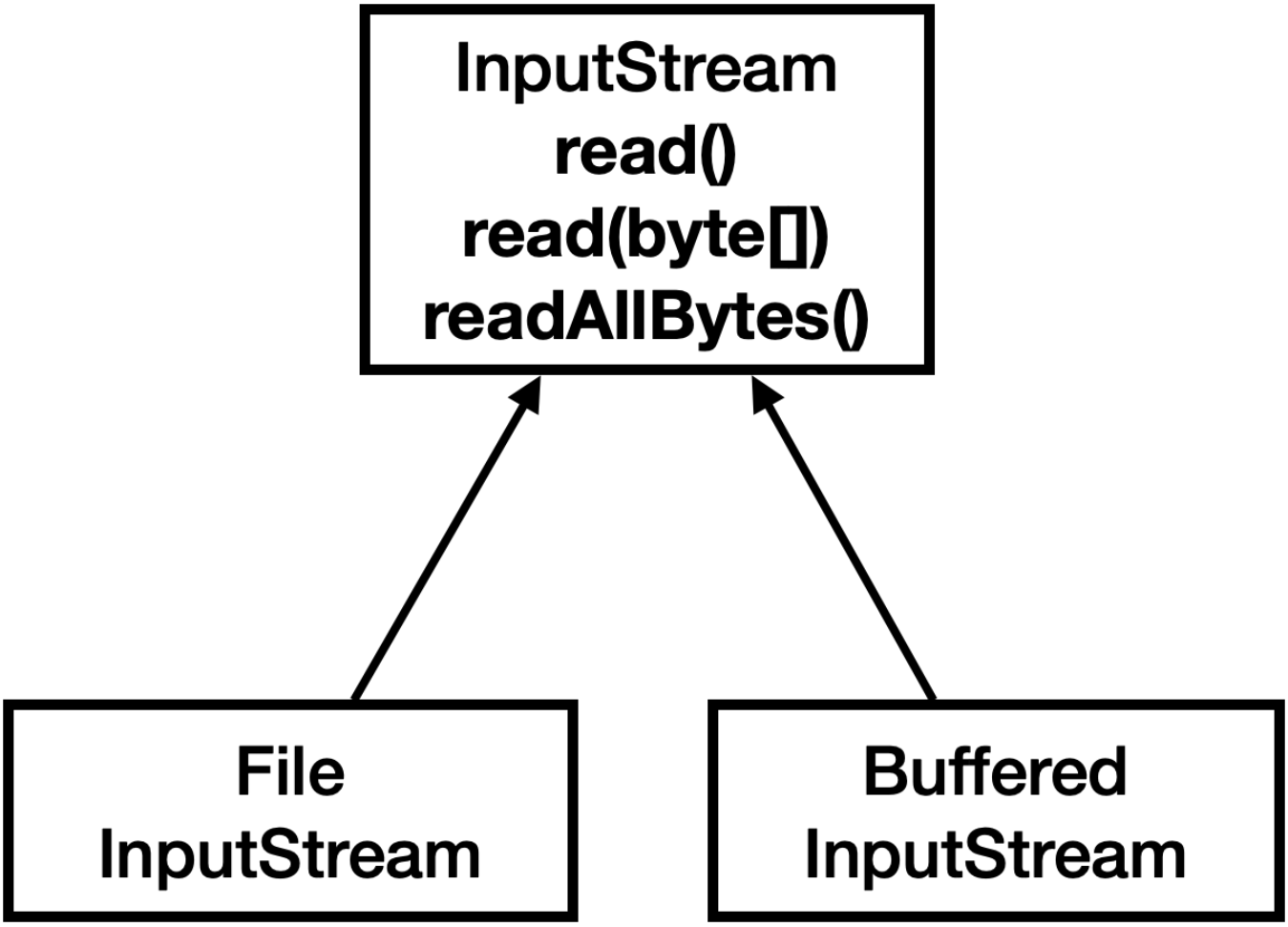
```java
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.BUFFER_SIZE;
import static io.buffered.BufferedConst.FILE_NAME;

public class ReadFileV3 {

    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream(FILE_NAME);
        BufferedInputStream bis = new BufferedInputStream(fis, BUFFER_SIZE);
        long startTime = System.currentTimeMillis();

        int fileSize = 0;
        int data;
        while ((data = bis.read()) != -1) {
            fileSize++;
        }
        bis.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File name: " + FILE_NAME);
        System.out.println("File size: " + (fileSize / 1024 / 1024) + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
File name: temp/buffered.dat
File size: 10MB
Time taken: 94ms
```

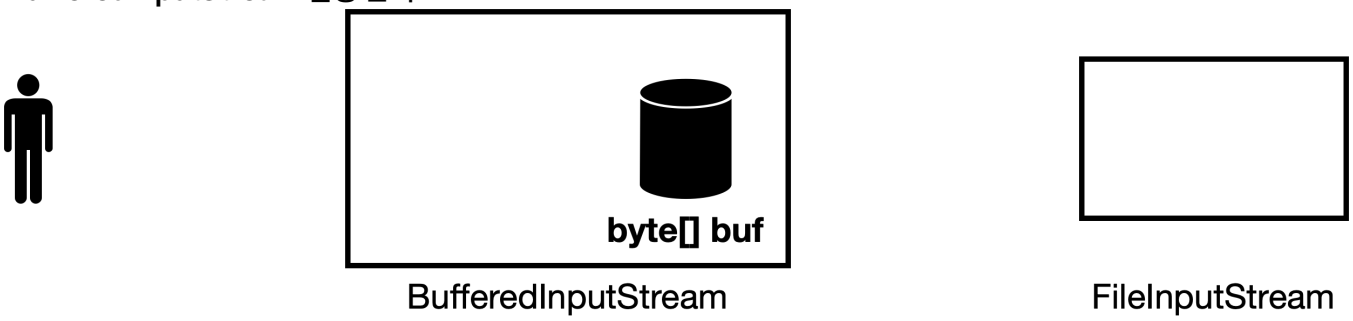
- 예제1이 약 5초 정도 걸렸는데, 약 50배 정도 빨라진 것을 확인할 수 있다.
- 예제2 보다는 느린데, 이 부분은 뒤에서 설명한다.

BufferedOutputStream 분석

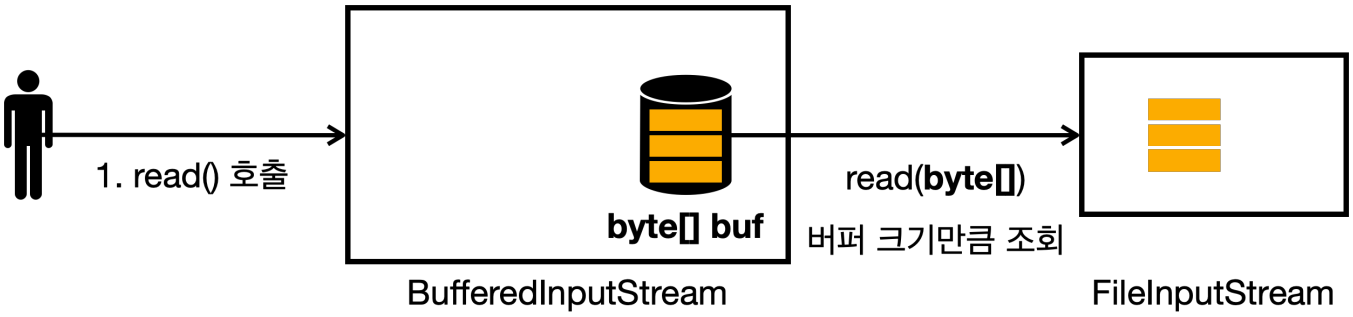


- `BufferdInputStream`은 `InputStream`을 상속받는다. 따라서 개발자 입장에서 보면 `InputStream`과 같은 기능을 그대로 사용할 수 있다. 예제에서는 `read()`를 사용했다.

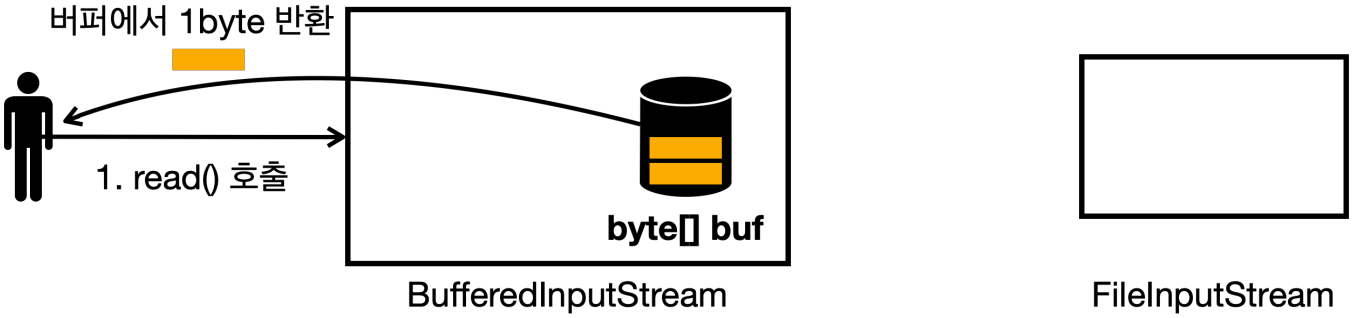
BufferedInputStream 실행 순서



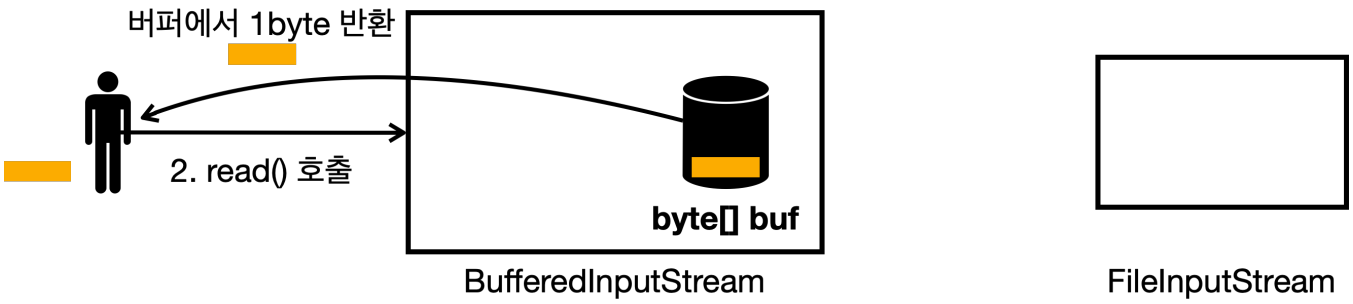
- `read()` 호출 전
- 버퍼의 크기는 3이라고 가정하겠다.



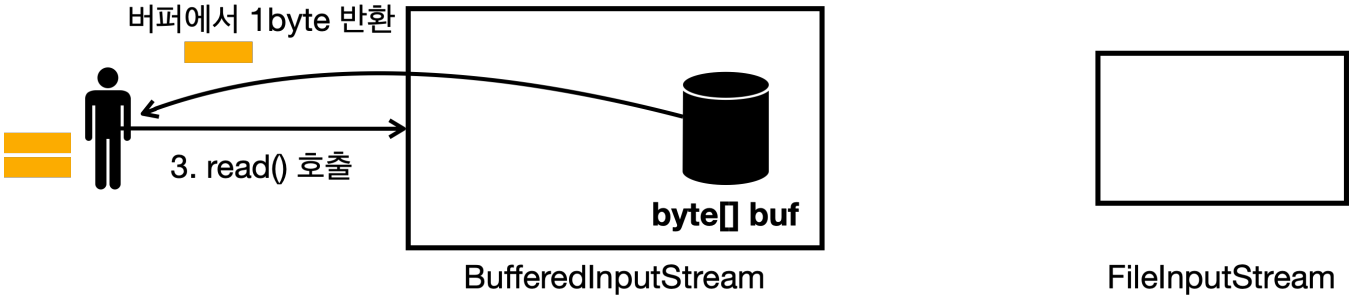
- `read()`는 1byte만 조회한다.
- `BufferedInputStream`은 먼저 버퍼를 확인한다. 버퍼에 데이터가 없으므로 데이터를 불러온다.
- `BufferedInputStream`은 `FileInputStream`에서 `read(byte[])`을 사용해서 버퍼의 크기인 3byte의 데이터를 불러온다.
- 불러온 데이터를 버퍼에 보관한다.



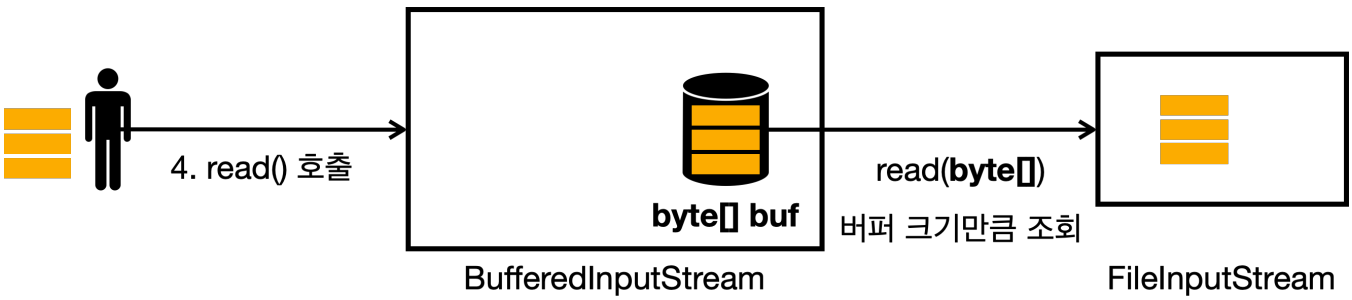
- 버퍼에 있는 데이터 중에 1byte를 반환한다.



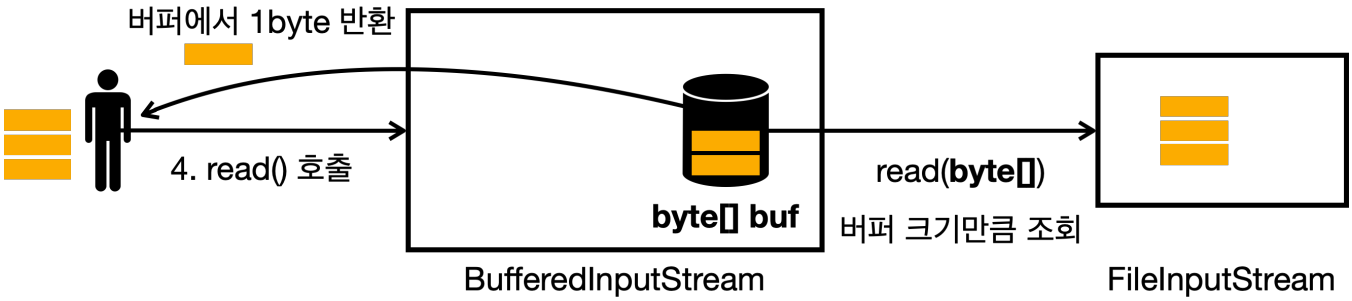
- `read()`를 또 호출하면 버퍼에 있는 데이터 중에 1byte를 반환한다.



- `read()`를 또 호출하면 버퍼에 있는 데이터 중에 1byte를 반환한다.



- `read()`를 호출하는데, 이번에는 버퍼가 비어있다.
- `FileInputStream`에서 버퍼 크기만큼 조회하고 버퍼에 담아둔다.



- 버퍼에 있는 데이터를 하나 반환한다.
- 이런 방식을 반복한다.

정리

- `BufferedInputStream`은 버퍼의 크기만큼 데이터를 미리 읽어서 버퍼에 보관해둔다. 따라서 `read()`를 통해 1byte씩 데이터를 조회해도, 성능이 최적화 된다.

버퍼를 직접 다루는 것 보다 BufferedXxx의 성능이 떨어지는 이유

- 예제1 쓰기: 14000ms (14초)
- 예제2 쓰기: 14ms (버퍼 직접 다룸)
- 예제3 쓰기: 102ms (BufferedXxx)

예제2는 버퍼를 직접 다루는 것이고, 예제3은 `BufferedXxx`라는 클래스가 대신 버퍼를 처리해준다. 버퍼를 사용하는 것은 같기 때문에 결과적으로 예제2와 예제3은 비슷한 성능이 나와야한다. 그런데 왜 예제2가 더 빠른 것일까?
이 이유는 바로 동기화 때문이다.

BufferedOutputStream.write()

```
@Override
public void write(int b) throws IOException {
    if (lock != null) {
        lock.lock();
        try {
            implWrite(b);
        } finally {
            lock.unlock();
        }
    } else {
        synchronized (this) {
            implWrite(b);
        }
    }
}
```

- `BufferedOutputStream`을 포함한 `BufferedXxx` 클래스는 모두 동기화 처리가 되어 있다.
- 이번 예제의 문제는 1byte씩 저장해서 총 10MB를 저장해야 하는데 이렇게 하려면 `write()`를 약 1000만 번 호출해야 한다.(10 * 1024 * 1024)
- 결과적으로 락을 걸고 푸는 코드도 1000만 번 호출된다는 뜻이다.

BufferedXxx 클래스의 특징

`BufferedXxx` 클래스는 자바 초창기에 만들이진 클래스인데, 처음부터 멀티 스레드를 고려해서 만든 클래스이다. 따라서 멀티 스레드에 안전하지만 락을 걸고 푸는 동기화 코드로 인해 성능이 약간 저하될 수 있다.
하지만 싱글 스레드 상황에서는 동기화 락이 필요하지 않기 때문에 직접 버퍼를 다룰 때와 비교해서 성능이 떨어진다.

일반적인 상황이라면 이 정도 성능은 크게 문제가 되지는 않기 때문에 싱글 스레드여도 `BufferedXxx`를 사용하면 충분하다.
물론 매우 큰 데이터를 다루어야 하고, 성능 최적화가 중요하다면 예제 2와 같이 직접 버퍼를 다루는 방법을 고려하자.
아쉽게도 동기화 락이 없는 `BufferedXxx` 클래스는 없다. 꼭 필요한 상황이라면 `BufferedXxx`를 참고해서 동기화 락 코드를 제거한 클래스를 직접 만들어 사용하면 된다.

파일 입출력과 성능 최적화5 - 한 번에 쓰기

파일의 크기가 크지 않다면 간단하게 한 번에 쓰고 읽는 것도 좋은 방법이다.
이 방법은 성능은 가장 빠르지만, 결과적으로 메모리를 한 번에 많이 사용하기 때문에 파일의 크기가 작아야 한다.

예제4 - 쓰기

```
package io.buffered;

import java.io.FileOutputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.FILE_NAME;
import static io.buffered.BufferedConst.FILE_SIZE;

public class CreateFileV4 {

    public static void main(String[] args) throws IOException {
        FileOutputStream fos = new FileOutputStream(FILE_NAME);
        long startTime = System.currentTimeMillis();

        byte[] buffer = new byte[FILE_SIZE];
        for (int i = 0; i < FILE_SIZE; i++) {
            buffer[i] = 1;
        }
        fos.write(buffer);
        fos.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File created: " + FILE_NAME);
        System.out.println("File size: " + FILE_SIZE / 1024 / 1024 + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
File created: temp/buffered.dat
File size: 10MB
Time taken: 15ms
```

- 실행 시간은 8KB의 버퍼를 직접 사용한 예제2와 오차 범위 정도로 거의 비슷하다.
- 디스크나 파일 시스템에서 데이터를 읽고 쓰는 기본 단위가 보통 4KB 또는 8KB이기 때문에, 한 번에 쓴다고해서 무작정 빠른 것은 아니다.

예제4 - 읽기

```
package io.buffered;

import java.io.FileInputStream;
import java.io.IOException;

import static io.buffered.BufferedConst.FILE_NAME;

public class ReadFileV4 {

    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream(FILE_NAME);
        long startTime = System.currentTimeMillis();

        byte[] bytes = fis.readAllBytes();
        fis.close();

        long endTime = System.currentTimeMillis();
        System.out.println("File name: " + FILE_NAME);
        System.out.println("File size: " + bytes.length / 1024 / 1024 + "MB");
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

- `readAllBytes()`를 사용하면 한번에 데이터를 다 읽을 수 있다.

실행 결과

```
File name: temp/buffered.dat
File size: 10MB
Time taken: 3ms
```

- 실행 시간은 8KB의 버퍼를 사용한 예제2와 오차 범위 정도로 거의 비슷하다.
- `readAllBytes()`는 자바 구현에 따라 다르지만 보통 4KB, 8KB, 16KB 단위로 데이터를 읽어들인다.

정리

정리

- 파일의 크기가 크지 않아서, 메모리 사용에 큰 영향을 주지 않는다면 쉽고 빠르게 한 번에 처리하자.
- 성능이 중요하고 큰 파일을 나누어 처리해야 한다면, 버퍼를 직접 다루자.
- 성능이 크게 중요하지 않고, 버퍼 기능이 필요하면 `BufferedXxx`를 사용하자.
  - `BufferedXxx`는 동기화 코드가 들어있어서 스레드 안전하지만, 약간의 성능 저하가 있다.