5. File, Files

#1.인강/0.자바/6.자바-고급2편

File

자바에서 파일 또는 디렉토리를 다룰 때는 `File` 또는 `Files`, `Path` 클래스를 사용하면 된다. 이 클래스들을 사용하면 파일이나 폴더를 생성하고, 삭제하고, 또 정보를 확인할 수 있다.

먼저 `File` 클래스를 사용해보자.

```
package io.file;

import java.io.File;
import java.io.IOException;
import java.util.Date;

public class OldFileMain {

    public static void main(String[] args) throws IOException {
        File file = new File("temp/example.txt");
        File directory = new File("temp/exampleDir");

        // 1. exists(): 파일이나 디렉토리의 존재 여부를 확인
        System.out.println("File exists: " + file.exists());

        // 2. createNewFile(): 새 파일을 생성
        boolean created = file.createNewFile();
        System.out.println("File created: " + created);

        // 3. mkdir(): 새 디렉토리를 생성
        boolean dirCreated = directory.mkdir();
```

```
        System.out.println("Directory created: " + dirCreated);

        // 4. delete(): 파일이나 디렉토리를 삭제
        //boolean deleted = file.delete();
        //System.out.println("File deleted: " + deleted);

        // 5. isFile(): 파일인지 확인
        System.out.println("Is file: " + file.isFile());

        // 6. isDirectory(): 디렉토리인지 확인
        System.out.println("Is directory: " + directory.isDirectory());

        // 7. getName(): 파일이나 디렉토리의 이름을 반환
        System.out.println("File name: " + file.getName());

        // 8. length(): 파일의 크기를 바이트 단위로 반환
        System.out.println("File size: " + file.length() + " bytes");

        // 9. renameTo(File dest): 파일의 이름을 변경하거나 이동
        File newFile = new File("temp/newExample.txt");
        boolean renamed = file.renameTo(newFile);
        System.out.println("File renamed: " + renamed);

        // 10. lastModified(): 마지막으로 수정된 시간을 반환
        long lastModified = newFile.lastModified();
        System.out.println("Last modified: " + new Date(lastModified));
    }
}
```

실행 결과

```
File exists: false
File created: true
Directory created: true
Is file: true
Is directory: true
File name: example.txt
File size: 0 bytes
File renamed: true
Last modified: Tue Sep 03 15:17:18 KST 2024
```

- `temp/example.txt` 파일 생성

- `temp/exampleDir` 폴더 생성

`File`은 파일과 디렉토리를 둘 다 다룬다. 참고로 `File` 객체를 생성했다고 파일이나 디렉토리가 바로 만들어지는 것은 아니다. 메서드를 통해 생성해야 한다.

`File`과 같은 클래스들은 학습해야할 중요한 원리가 있는 것이 아니라, 다양한 기능의 모음을 제공한다. 이런 클래스의 기능들은 외우기 보다는 이런 것들이 있다 정도만 간단히 알아두고, 필요할 때 찾아서 사용하면 된다.

Files

File, Files의 역사

자바 1.0에서 `File` 클래스가 등장했다. 이후에 자바 1.7에서 `File` 클래스를 대체할 `Files`와 `Path`가 등장했다.

Files의 특징

- 성능과 편의성이 모두 개선되었다.
- `File`은 과거의 호환을 유지하기 위해 남겨둔 기능이다. 이제는 `Files` 사용을 먼저 고려하자.
- 여기에는 수 많은 유틸리티 기능이 있다. `File` 클래스는 물론이고, `File`과 관련된 스트림(`FileInputStream`, `FileWriter`)의 사용을 고민하기 전에 `Files`에 있는 기능을 먼저 찾아보자. 성능도 좋고, 사용하기도 더 편리하다.
- 기능이 너무 많기 때문에 주요 기능만 알아보고, 나머지는 필요할 때 검색하자.
- 이렇게 기능 위주의 클래스는 외우는 것이 아니다. 이런게 있다 정도의 주요 기능만 알아두고, 나머지는 필요할 때 검색하면 된다.

앞서 작성한 코드를 `Files`로 그대로 사용해보자.
참고로 `Files`를 사용할 때 파일이나, 디렉토리의 경로는 `Path` 클래스를 사용해야 한다.

```
package io.file;

import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

public class NewFilesMain {

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

```
        Path file = Path.of("temp/example.txt");
        Path directory = Path.of("temp/exampleDir");

        // 1. exists(): 파일이나 디렉토리의 존재 여부를 확인
        System.out.println("File exists: " + Files.exists(file));

        // 2. createFile(): 새 파일을 생성
        try {
            Files.createFile(file);
            System.out.println("File created");
        } catch (FileAlreadyExistsException e) {
            System.out.println(file + " File already exists");
        }

        // 3. createDirectory(): 새 디렉토리를 생성
        try {
            Files.createDirectory(directory);
            System.out.println("Directory created");
        } catch (FileAlreadyExistsException e) {
            System.out.println(directory + " Directory already exists");
        }

        // 4. delete(): 파일이나 디렉토리를 삭제
        //Files.delete(file);
        //System.out.println("File deleted");

        // 5. isRegularFile(): 일반 파일인지 확인
        System.out.println("Is regular file: " + Files.isRegularFile(file));

        // 6. isDirectory(): 디렉토리인지 확인
        System.out.println("Is directory: " + Files.isDirectory(directory));

        // 7. getFileName(): 파일이나 디렉토리의 이름을 반환
        System.out.println("File name: " + file.getFileName());

        // 8. size(): 파일의 크기를 바이트 단위로 반환
        System.out.println("File size: " + Files.size(file) + " bytes");

        // 9. move(): 파일의 이름을 변경하거나 이동
        Path newFile = Paths.get("temp/newExample.txt");
        Files.move(file, newFile, StandardCopyOption.REPLACE_EXISTING);
        System.out.println("File moved/renamed");
```

```
        // 10. getLastModifiedTime(): 마지막으로 수정된 시간을 반환
        System.out.println("Last modified: " +
Files.getLastModifiedTime(newFile));

        // 추가: readAttributes(): 파일의 기본 속성들을 한 번에 읽기
        BasicFileAttributes attrs = Files.readAttributes(newFile,
BasicFileAttributes.class);
        System.out.println("===== Attributes =====");
        System.out.println("Creation time: " + attrs.creationTime());
        System.out.println("Is directory: " + attrs.isDirectory());
        System.out.println("Is regular file: " + attrs.isRegularFile());
        System.out.println("Is symbolic link: " + attrs.isSymbolicLink());
        System.out.println("Size: " + attrs.size());
    }
}
```

- `Files`는 직접 생성할 수 없고, static 메서드를 통해 기능을 제공한다.

실행 결과

```
File exists: false
File created
temp/exampleDir Directory already exists
Is regular file: true
Is directory: true
File name: example.txt
File size: 0 bytes
File moved/renamed
Last modified: 2024-09-03T06:32:09.182412076Z
===== Attributes =====
Creation time: 2024-09-03T06:32:09Z
Is directory: false
Is regular io.file: true
Is symbolic link: false
Size: 0
```

경로 표시

파일이나 디렉토리가 있는 경로는 크게 절대 경로와 정규 경로로 나눌 수 있다.

File 경로 표시

```
package io.file;

import java.io.File;
import java.io.IOException;

public class OldFilePath {

    public static void main(String[] args) throws IOException {
        File file = new File("temp/..");
        System.out.println("path = " + file.getPath());
        // 절대 경로
        System.out.println("Absolute path = " + file.getAbsolutePath());
        // 정규 경로
        System.out.println("Canonical path = " + file.getCanonicalPath());

        File[] files = file.listFiles();
        for (File f : files) {
            System.out.println((f.isFile() ? "F" : "D") + " | " +
f.getName());
        }
    }
}
```

실행 결과

```
path = temp/..
Absolute path = /Users/yh/study/inflearn/java/java-adv2/temp/..
Canonical path = /Users/yh/study/inflearn/java/java-adv2
D | temp
F | java-adv2-v1.iml
D | out
F | .gitignore
D | .idea
D | src
```

- **절대 경로(Absolute path)**: 절대 경로는 경로의 처음부터 내가 입력한 모든 경로를 다 표현한다.
- **정규 경로(Canonical path)**: 경로의 계산이 모두 끝난 경로이다. 정규 경로는 하나만 존재한다.
  - 예제에서 `..`은 바로 위의 상위 디렉토리를 뜻한다. 이런 경로의 계산을 모두 처리하면 하나의 경로만 남는

다.

- 예를 들어 절대 경로는 다음 2가지 경로가 모두 가능하지만
  - `/Users/yh/study/inflearn/java/java-adv2`
  - `/Users/yh/study/inflearn/java/java-adv2/temp/..`
- 정규 경로는 다음 하나만 가능하다.
  - `/Users/yh/study/inflearn/java/java-adv2`

file.listFiles()

- 현재 경로에 있는 모든 파일 또는 디렉토리를 반환한다.
- 파일이면 F, 디렉토리면 D로 표현했다.

Files 경로 표시

앞의 예시를 `Files` 버전으로 작성한 코드이다.

```
package io.file;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
import java.util.stream.Stream;

public class NewFilesPath {

    public static void main(String[] args) throws IOException {
        Path path = Path.of("temp/..");
        System.out.println("path = " + path);
        // 절대 경로
        System.out.println("Absolute path = " + path.toAbsolutePath());
        // 정규 경로
        System.out.println("Canonical path = " + path.toRealPath());

        Stream<Path> pathStream = Files.list(path);
        List<Path> list = pathStream.toList();
        pathStream.close();

        for (Path p : list) {
            System.out.println((Files.isRegularFile(p) ? "F" : "D") + " | " +
p.getFileName());
```

```
        }
    }
}
```

실행 결과

```
path = temp/..
Absolute path = /Users/yh/study/inflearn/java/java-adv2/temp/..
Canonical path = /Users/yh/study/inflearn/java/java-adv2
D | temp
F | java-adv2-v1.iml
D | out
F | .gitignore
D | .idea
D | src
```

Files.list(path)

- 현재 경로에 있는 모든 파일 또는 디렉토리를 반환한다. `Stream`이라는 객체는 람다와 스트림에서 따로 학습한다. 여기서는 `toList()`를 통해 우리가 잘 아는 `List` 컬렉션으로 변경해서 사용했다.
- 파일이면 F, 디렉토리면 D로 표현했다.

Files로 문자 파일 읽기

문자로된 파일을 읽고 쓸 때 과거에는 `FileReader`, `FileWriter` 같은 복잡한 스트림 클래스를 사용해야 했다. 거기에 모든 문자를 읽으려면 반복문을 사용해서 파일의 끝까지 읽어야 하는 과정을 추가해야 한다. 또 한 줄 단위로 파일을 읽으려면 `BufferedReader`와 같은 스트림 클래스를 추가해야 했다.

`Files`는 이런 문제를 코드 한 줄로 깔끔하게 해결해준다.

Files - 모든 문자 읽기

```
package io.file.text;

import java.io.IOException;
```

```java
import java.nio.file.Files;
import java.nio.file.Path;

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

public class ReadTextFileV1 {

    private static final String PATH = "temp/hello2.txt";

    public static void main(String[] args) throws IOException {
        String writeString = "abc\n가나다";
        System.out.println("== Write String ==");
        System.out.println(writeString);

        Path path = Path.of(PATH);

        // 파일에 쓰기
        Files.writeString(path, writeString, UTF_8);
        // 파일에서 읽기
        String readString = Files.readString(path, UTF_8);

        System.out.println("== Read String ==");
        System.out.println(readString);
    }
}
```

실행 결과

```
== Write String ==
abc
가나다
== Read String ==
abc
가나다
```

- `Files.writeString()` : 파일에 쓰기
- `Files.readString()` : 파일에서 모든 문자 읽기

`Files`를 사용하면 아주 쉽게 파일에 문자를 쓰고 읽을 수 있다.

Files - 라인 단위로 읽기

```
package io.file.text;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

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

public class ReadTextFileV2 {

    public static final String PATH = "temp/hello2.txt";

    public static void main(String[] args) throws IOException {
        String writeString = "abc\n가나다";
        System.out.println("== Write String ==");
        System.out.println(writeString);

        Path path = Path.of(PATH);

        // 파일에 쓰기
        Files.writeString(path, writeString, UTF_8);
        // 파일에서 읽기

        System.out.println("== Read String ==");
        List<String> lines = Files.readAllLines(path, UTF_8);
        for (int i = 0; i < lines.size(); i++) {
            System.out.println((i + 1) + ": " + lines.get(i));
        }
    }
}
```

실행 결과

```
== Write String ==
abc
가나다
== Read String ==
1: abc
2: 가나다
```

Files.readAllLines(path)

- 파일을 한 번에 다 읽고, 라인 단위로 `List`에 나누어 저장하고 반환한다.

Files.lines(path)

- 파일을 한 줄 단위로 나누어 읽고, 메모리 사용량을 줄이고 싶다면 이 기능을 사용하면 된다.
- 다만 이 기능을 제대로 이해하려면 람다와 스트림을 알아야 한다. 지금은 이런 것이 있다 정도만 알아두자.

```
try(Stream<String> lineStream = Files.lines(path, UTF_8)){
    lineStream.forEach(line -> System.out.println(line));
}
```

- 파일을 스트림 단위로 나누어 조회한다. (I/O 스트림이 아니라, 람다와 스트림에서 사용하는 스트림이다)
- 파일이 아주 크다면 한 번에 모든 파일을 다 메모리에 올리는 것 보다, 파일을 부분 부분 나누어 메모리에 올리는 것이 더 나은 선택일 수 있다.
- 예를 들어 파일의 크기가 1000MB라면 한 번에 1000MB의 파일이 메모리에 불러진다.
  - 앞서 `Files.readAllLines`의 경우 `List`에 1000MB의 파일이 모두 올라간다.
- 이 기능을 사용하면 파일을 한 줄 단위로 메모리에 올릴 수 있다. 한 줄 당 1MB의 용량을 사용한다면 자바는 파일에서 한 번에 1MB의 데이터만 메모리에 올려 처리한다. 그리고 처리가 끝나면 다음 줄을 호출하고, 기존에 사용한 1M의 데이터는 GC한다.
- 용량이 아주 큰 파일을 처리해야 한다면 이런 방식으로 나누어 처리하는 것이 효과적이다.
  - 참고로 용량이 너무 커서 자바 메모리에 한 번에 불러오는 것이 불가능할 수 있다. 그때는 이런 방식으로 처리해야만 한다.
- 물론 `BufferedReader`를 통해서도 한 줄 단위로 이렇게 나누어 처리하는 것이 가능하다. 여기서 핵심은 매우 편리하게 문자를 나누어 처리하는 것이 가능하다는 점이다.
- 이 부분은 이후에 람다와 스트림을 학습해야 제대로 이해할 수 있다.

파일 복사 최적화

이번에는 파일을 복사하는 효율적인 방법에 대해서 알아보자.

예제를 위해서 200MB 임시 파일을 하나 만들어보자.

예제 파일 생성

```
package io.file.copy;

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

public class CreateCopyFile {
    private static final int FILE_SIZE = 200 * 1024 * 1024; // 200MB

    public static void main(String[] args) throws IOException {
        String fileName = "temp/copy.dat";
        long startTime = System.currentTimeMillis();

        FileOutputStream fos = new FileOutputStream(fileName);
        byte[] buffer = new byte[FILE_SIZE];
        fos.write(buffer);
        fos.close();

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

실행 결과

```
File created: temp/copy.dat
File size: 200MB
Time taken: 69ms
```

이렇게 만든 임시 파일을 다른 파일로 복사해보자.

파일 복사 예제1

```
package io.file.copy;

import java.io.FileInputStream;
```

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

public class FileCopyMainV1 {

    public static void main(String[] args) throws IOException {
        long startTime = System.currentTimeMillis();
        FileInputStream fis = new FileInputStream("temp/copy.dat");
        FileOutputStream fos = new FileOutputStream("temp/copy_new.dat");

        byte[] bytes = fis.readAllBytes();
        fos.write(bytes);
        fis.close();
        fos.close();
        long endTime = System.currentTimeMillis();
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
Time taken: 109ms
```

- `FileInputStream`에서 `readAllBytes`를 통해 한 번에 모든 데이터를 읽고 `write(bytes)`를 통해 한 번에 모든 데이터를 저장한다.
- 파일(copy.dat) ➜ 자바(byte) ➜ 파일(copy_new.dat)의 과정을 거친다.
- 자바가 `copy.dat` 파일의 데이터를 자바 프로세스가 사용하는 메모리에 불러온다. 그리고 메모리에 있는 데이터를 `copy_new.dat`에 전달한다.

파일 복사 예제2

```java
package io.file.copy;

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

public class FileCopyMainV2 {
```

```
    public static void main(String[] args) throws IOException {
        long startTime = System.currentTimeMillis();
        FileInputStream fis = new FileInputStream("temp/copy.dat");
        FileOutputStream fos = new FileOutputStream("temp/copy_new.dat");
        fis.transferTo(fos);
        fis.close();
        fos.close();
        long endTime = System.currentTimeMillis();
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
Time taken: 70ms
```

- `InputStream`에는 `transferTo()`라는 특별한 메서드가 있다. (자바 9)
- 이 메서드는 `InputStream`에서 읽은 데이터를 바로 `OutputStream`으로 출력한다.
- `transferTo()`는 성능 최적화가 되어 있기 때문에, 앞의 예제와 비슷하거나 조금 더 빠르다.
  - 상황에 따라 조금 더 느릴 수도 있다.
  - 참고로 디스크는 실행시 시간의 편차가 심하다는 점을 알아두자.
- 파일(copy.dat) ➜ 자바(byte) ➜ 파일(copy_new.dat)의 과정을 거친다.

`transferTo()` 덕분에 매우 편리하게 `InputStream`의 내용을 `OutputStream`으로 전달할 수 있었다.

파일 복사 예제3

```
package io.file.copy;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;

public class FileCopyMainV3 {

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

```
        Path source = Path.of("temp/copy.dat");
        Path target = Path.of("temp/copy_new.dat");
        Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);

        long endTime = System.currentTimeMillis();
        System.out.println("Time taken: " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
Time taken: 37ms
```

Files.copy()

앞의 예제들은 파일을 복사할 때 다음 과정을 거쳤다.

- 파일(copy.dat) ➜ 자바(byte) ➜ 파일(copy_new.dat)

이 과정들은 파일의 데이터를 자바로 불러오고 또 자바에서 읽은 데이터를 다시 파일에 전달해야 한다.

`Files.copy()` 는 자바에 파일 데이터를 불러오지 않고, 운영체제의 파일 복사 기능을 사용한다.
따라서 다음과 같이 중간 과정이 생략된다.

- 파일(copy.dat) ➜ 파일(copy_new.dat)

따라서 가장 빠르다. 파일을 다루어야 할 일이 있다면 항상 `Files` 의 기능을 먼저 찾아보자.
물론 이 기능은 파일에서 파일을 복사할 때만 유용하다. 만약 파일의 정보를 읽어서 처리해야 하거나, 스트림을 통해 네트워크에 전달해야 한다면 앞서 설명한 스트림을 직접 사용해야 한다.

정리