4. I/O 활용

#1.인강/0.자바/6.자바-고급2편

회원 관리 예제1 - 메모리

I/O를 사용해서 회원 데이터를 관리하는 예제를 만들어보자.

요구사항

회원 관리 프로그램을 작성해라.

회원의 속성은 다음과 같다.

- ID
- Name
- Age

회원을 등록하고, 등록한 회원의 목록을 조회할 수 있어야 한다.

프로그램 작동 예시

```
1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 1
ID 입력: id1
Name 입력: name1
Age 입력: 20
회원이 성공적으로 등록되었습니다.

1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 1
ID 입력: id2
Name 입력: name2
Age 입력: 30
회원이 성공적으로 등록되었습니다.
```

```
1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 2
회원 목록:
[ID: id1, Name: name1, Age: 20]
[ID: id2, Name: name2, Age: 30]

1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 3
프로그램을 종료합니다.
```

회원 클래스

```java
package io.member;

public class Member {
    private String id;
    private String name;
    private Integer age;

    public Member() {
    }

    public Member(String id, String name, Integer age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Member{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

회원을 저장하고 관리하는 인터페이스

```
package io.member;

import java.util.List;

public interface MemberRepository {
    void add(Member member);

    List<Member> findAll();
}
```

- `add()` : 회원 객체를 저장한다.
- `findAll()` : 저장한 회원 객체를 `List`로 모두 조회한다.
- Repository는 저장소라는 뜻이다.

메모리에 회원을 저장하고 관리하는 구현체

```java
package io.member.impl;

import io.member.Member;
import io.member.MemberRepository;

import java.util.ArrayList;
import java.util.List;

public class MemoryMemberRepository implements MemberRepository {

    private final List<Member> members = new ArrayList<>();

    @Override
    public void add(Member member) {
        members.add(member);
    }

    @Override
    public List<Member> findAll() {
        return members;
    }
}
```

- 간단하게 메모리에 회원을 저장하고 관리하자.
- 회원을 저장하면 내부에 있는 `members` 리스트에 회원이 저장된다.
- 회원을 조회하면 `members` 리스트가 반환된다.

프로그램 main

```java
package io.member;

import io.member.impl.MemoryMemberRepository;

import java.util.List;
import java.util.Scanner;

public class MemberConsoleMain {

    private static final MemberRepository repository = new
MemoryMemberRepository();
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("1.회원 등록 | 2.회원 목록 조회 | 3.종료");
            System.out.print("선택: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // newline 제거

            switch (choice) {
                case 1:
                    registerMember(scanner);
                    break;
                case 2:
                    // 회원 목록 조회
                    displayMembers();
                    break;
                case 3:
                    System.out.println("프로그램을 종료합니다.");
                    return;
                default:
                    System.out.println("잘못된 선택입니다. 다시 입력하세요.");
            }

        }
    }

    private static void registerMember(Scanner scanner) {
        System.out.print("ID 입력: ");
        String id = scanner.nextLine();

        System.out.print("Name 입력: ");
        String name = scanner.nextLine();

        System.out.print("Age 입력: ");
        int age = scanner.nextInt();
        scanner.nextLine(); // newline 제거

        Member newMember = new Member(id, name, age);
        repository.add(newMember);
        System.out.println("회원이 성공적으로 등록되었습니다.");
    }

    private static void displayMembers() {
```

```
            List<Member> members = repository.findAll();
            System.out.println("회원 목록:");
            for (Member member : members) {
                System.out.printf("[ID: %s, Name: %s, Age: %d]\n", member.getId(),
member.getName(), member.getAge());
            }
        }

}
```

- 콘솔을 통해 회원 등록, 목록 조회 기능을 제공한다.

각 실행 결과는 다음과 같다.

회원 등록

```
1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 1
ID 입력: id1
Name 입력: name1
Age 입력: 20
회원이 성공적으로 등록되었습니다.
```

목록 조회

```
1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 2
회원 목록:
[ID: id1, Name: name1, Age: 20]
```

종료

```
1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 3
프로그램을 종료합니다.
```

문제

이 프로그램은 잘 작동하지만, 데이터를 메모리에 보관하기 때문에, 자바를 종료하면 모든 회원 정보가 사라진다. 따라

서 프로그램을 다시 실행하면 모든 회원 데이터가 사라진다.

프로그램을 종료하고 다시 실행해도 회원 데이터가 영구 보존되어야 한다.

회원 관리 예제2 - 파일에 보관

회원 데이터를 영구 보존하려면 파일에 저장하면 된다.

다음과 같이 한 줄 단위로 회원 데이터를 파일에 저장해보자.

temp/members-txt.dat 예시

```
id1,member1,20
id2,member2,30
```

- 여기서는 문자를 파일에 저장한다. 문자를 다루므로 `Reader`, `Writer`를 사용하는 것이 편리하다.
- 한 줄 단위로 처리할 때는 `BufferedReader`가 유용하므로 `BufferedReader`, `BufferedWriter`를 사용해보자.

```
package io.member.impl;

import io.member.Member;
import io.member.MemberRepository;

import java.io.*;
import java.util.ArrayList;
import java.util.List;

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

public class FileMemberRepository implements MemberRepository {

    private static final String FILE_PATH = "temp/members-txt.dat";
    private static final String DELIMITER = ",";

    @Override
    public void add(Member member) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter(FILE_PATH,
UTF_8, true))) {
```

```
            bw.write(member.getId() + DELIMITER + member.getName() + DELIMITER
+ member.getAge());
            bw.newLine();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public List<Member> findAll() {
        List<Member> members = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(FILE_PATH,
UTF_8))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] memberData = line.split(DELIMITER);
                members.add(new Member(memberData[0], memberData[1],
Integer.valueOf(memberData[2])));
            }
            return members;
        } catch (FileNotFoundException e) {
            return new ArrayList<>();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

- `MemberRepository` 인터페이스가 잘 정의되어 있으므로, 이 인터페이스를 기반으로 파일에 회원 데이터를 보관하는 구현체를 만들면 된다.
- `DELIMITER`: 회원 데이터는 `id1,member1,20`와 같이 `,`(쉼표)로 구분한다.

참고: 빈 컬렉션 반환

- 빈 컬렉션을 반환할 때는 `new ArrayList()` 보다는 `List.of()`를 사용하는 것이 좋다.
- 자세한 내용은 자바 중급 2편 - 10 컬렉션 프레임워크 - 컬렉션 유틸 - 빈 리스트 생성 부분을 참고하자.
- 이번 예제에서도 `List.of()`를 사용하는 것이 좋지만, 뒤에 나오는 `ObjectStream` 부분과 내용을 맞추기 위해 빈 컬렉션에 `new ArrayList()`를 사용했다.

회원 저장

```
bw.write(member.getId() + DELIMITER + member.getName() + DELIMITER +
```

```
member.getAge());
bw.newLine();
```

- 회원 객체의 데이터를 읽어서 `String` 문자로 변경한다. 여기서 `write()` 는 `String` 을 입력으로 받는다. 그리고 `DELIMITER` 를 구분자로 사용한다.
- `bw.write("id1,member1,20")` 를 통해 저장할 문자가 전달된다.
- 회원 데이터를 문자로 변경한 다음에 파일에 보관한 것이다.
- 각 회원을 구분하기 위해 `newLine()` 을 통해 다음 줄로 이동한다.

회원 조회

- `line = br.readLine()` 을 통해 각 회원 하나하나를 불러온다.
  - `line = "id1,member1,20"` 와 같이 하나의 회원 정보가 담긴 한 줄 문자가 입력된다.
- `String[] memberData = line.split(DELIMITER)`
  - 회원 데이터를 `DELIMITER` 구분자로 구분해서 배열에 담는다.
- `members.add(new Member(memberData[0], memberData[1], Integer.valueOf(memberData[2])`
  - 파일에 읽은 데이터를 기반으로 회원 객체를 생성한다.
  - `id` , `name` 은 `String` 이기 때문에 타입이 같다. `age` 의 경우 문자(`String`) 20으로 조회했기 때문에 숫자인 `Integer` 로 변경해야 한다.
- `FileNotFoundException e`
  - 회원 데이터가 하나도 없을 때는 `temp/members-txt.dat` 파일이 존재하지 않는다. 따라서 해당 예외가 발생한다. 이 경우 회원 데이터가 하나도 없는 것으로 보고 빈 리스트를 반환한다.

try-with-resources

`try-with-resources` 구문을 사용해서 자동으로 자원을 정리한다. `try` 코드 블록이 끝나면 자동으로 `close()` 가 호출되면서 자원을 정리한다.

```
try (BufferedWriter bw = new BufferedWriter(new FileWriter(FILE_PATH, UTF_8,
true))) {...}
try (BufferedReader br = new BufferedReader(new FileReader(FILE_PATH, UTF_8)))
{...}
```

- 참고: `try-with-resources` 구분에 대한 자세한 내용은 **자바-중급1편 - 예외 처리2 - 실습** 내용을 참고하자.

MemberConsoleMain 수정 - FileMemberRepository 사용

```
public class MemberConsoleMain {
    //private static final MemberRepository repository = new
MemoryMemberRepository();
```

```
    private static final MemberRepository repository = new
FileMemberRepository();
    ...
}
```

- `MemoryMemberRepository` 대신에 `FileMemberRepository`를 사용하도록 코드를 수정하자.
- `MemberRepository` 인터페이스를 사용한 덕분에 구현체가 변경되더라도 클라이언트의 다른 코드들은 변경하지 않아도 된다.

실행

```
1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 1
ID 입력: id1
Name 입력: name1
Age 입력: 20
회원이 성공적으로 등록되었습니다.

1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 1
ID 입력: id2
Name 입력: name2
Age 입력: 30
회원이 성공적으로 등록되었습니다.

1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 2
회원 목록:
[ID: id1, Name: name1, Age: 20]
[ID: id2, Name: name2, Age: 30]

1.회원 등록 | 2.회원 목록 조회 | 3.종료
선택: 3
프로그램을 종료합니다.
```

결과 - temp/members-txt.dat

```
id1,member1,20
id2,member2,30
```

파일에 회원 데이터를 저장한 덕분에, 자바를 다시 실행해도 저장한 회원이 잘 조회되는 것을 확인할 수 있다.

문제

모든 타입을 문자로 저장하는 문제

```
public class Member {
    private String id;
    private String name;
    private Integer age;
}
```

- 회원 객체는 `String`, `Integer` 같은 자바의 다양한 타입을 사용한다.
- 그런데 이런 타입을 무시하고 모든 데이터를 문자로 변경해서 저장하는 부분이 아쉽다.
- `age`의 경우 문자를 숫자로 변경하기 위한 코드도 따로 작성해야 한다.
  - `Integer.valueOf(memberData[2])`

구분자(`DELIMITER`)를 사용하는 문제

- `id`, `name`, `age` 각 필드를 구분하기 위해 구분자를 넣어서 저장하고, 또 조회할 때도 구분자를 사용해서 각 필드를 구분해야 한다.

회원 관리 예제3 - DataStream

앞서 배운 예시 중에 `DataOutputStream`, `DataInputStream`을 떠올려보자. 이 스트림들은 자바의 데이터 타입을 그대로 사용할 수 있다. 따라서 자바의 타입을 그대로 사용하면서 파일에 데이터를 저장하고 불러올 수 있고, 구분자도 사용하지 않아도 된다.

```
package io.member.impl;

import io.member.Member;
import io.member.MemberRepository;

import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class DataMemberRepository implements MemberRepository {
```

```
    private static final String FILE_PATH = "temp/members-data.dat";

    @Override
    public void add(Member member) {
        try (DataOutputStream dos = new DataOutputStream(new
FileOutputStream(FILE_PATH, true))) {
            dos.writeUTF(member.getId());
            dos.writeUTF(member.getName());
            dos.writeInt(member.getAge());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public List<Member> findAll() {
        List<Member> members = new ArrayList<>();
        try (DataInputStream dis = new DataInputStream(new
FileInputStream(FILE_PATH))) {
            while (dis.available() > 0) {
                members.add(new Member(dis.readUTF(), dis.readUTF(),
dis.readInt()));
            }
            return members;

        } catch (FileNotFoundException e) {
            return new ArrayList<>();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

회원 저장

```
dos.writeUTF(member.getId());
dos.writeUTF(member.getName());
dos.writeInt(member.getAge());
```

- 회원을 저장할 때는 회원 필드의 타입에 맞는 메서드를 호출하면 된다.
- 이전 예제에서는 각 회원을 한 줄 단위로 구분했는데, 여기서는 그런 구분이 필요없다.

회원 조회

```
new Member(dis.readUTF(), dis.readUTF(), dis.readInt()
```

- 회원 데이터를 조회할 때는 회원 필드의 각 타입에 맞는 메서드를 사용해서 조회하면 된다.

MemberConsoleMain 수정 - DataMemberRepository 사용

```
public class MemberConsoleMain {
    //private static final MemberRepository repository = new
MemoryMemberRepository();
    //private static final MemberRepository repository = new
FileMemberRepository();
    private static final MemberRepository repository = new
DataMemberRepository();
    ...
}
```

실행 결과 - temp/members-data.dat

파일이 정상 보관된다. 문자와 byte가 섞여 있다.

DataStream 원리

`DataStream`은 어떤 원리로 구분자나 한 줄 라인 없이 데이터를 저장하고 조회할 수 있는 것일까?
다음과 같이 회원 데이터를 저장한다고 가정해보자.

String

```
dos.writeUTF("id1"); // 저장
dis.readUTF(); // 조회 id1
```

- `readUTF()`로 문자를 읽어올 때 어떻게 `id1`이라는 3글자만 정확하게 읽어올 수 있는 것일까?
- `writeUTF()`은 UTF-8 형식으로 문자를 저장하는데, 저장할 때 2byte를 추가로 사용해서 앞에 글자의 길이를 저장해둔다. (65535 길이까지만 사용 가능)

```
3id1(2byte(문자 길이) + 3byte(실제 문자 데이터))
```

- 따라서 `readUTF()`로 읽어들일 때 먼저 앞의 2byte로 글자의 길이를 확인하고 해당 길이 만큼 글자를 읽어들인다.
- 이 경우 2byte를 사용해서 3이라는 문자의 길이를 숫자로 보관하고, 나머지 3byte로 실제 문자 데이터를 보관한다.

기타 타입

```
dos.writeInt(20);
dis.readInt()
```

- 자바의 `Int(Integer)`는 4byte를 사용하기 때문에 4byte를 사용해서 파일을 저장하고, 읽을 때도 4byte를 읽어서 복원한다.

저장 예시

```
dos.writeUTF("id1");
dos.writeUTF("name1");
dos.writeInt(20);
dos.writeUTF("id2");
dos.writeUTF("name2");
dos.writeInt(30);
```

저장된 파일 예시

```
3id1(2byte(문자 길이) + 3byte)
5name1(2byte(문자 길이) + 5byte)
20(4byte)
3id2(2byte(문자 길이) + 3byte)
5name2(2byte(문자 길이) + 5byte)
30(4byte)
```

- 여기서는 이해를 돕기 위해 각 필드를 엔터로 구분했다. 실제로는 엔터 없이 한 줄로 연결되어 있다.
- 저장된 파일은 실제로는 문자와 byte가 섞여있다.

정리

`DataStream` 덕분에 자바의 타입도 그대로 사용하고, 구분자도 제거할 수 있었다.
추가로 모든 데이터를 문자로 저장할 때 보다 저장 용량도 더 최적화 할 수 있다.

예를 들어서 숫자의 1,000,000,000(10억)을 문자로 저장하게 되면 총 10byte가 사용된다.
왜냐하면 숫자 1 0 0 0 0 0 0 0 0 0 각각 하나하나를 문자로 저장해야 하기 때문에 ASCII 인코딩을 해도 각각 1byte가 사용된다.
하지만 이것을 자바의 int와 같이 4byte를 사용해서 저장한다면 4byte만 사용하게 된다. 여기서는 `writeInt()`를 사용하면 4byte를 사용해서 저장한다.
물론 이렇게 byte를 직접 저장하면, 문서 파일을 열어서 확인하고 수정하는 것이 어렵다는 단점도 있다.

문제

`DataStream` 덕분에 회원 데이터를 더 편리하게 저장할 수 있는 것은 맞지만, 회원의 필드 하나하나를 다 조회해서 각 타입에 맞도록 따로따로 저장해야 한다. 이것은 회원 객체를 저장한다기 보다는 회원 데이터를 하나하나 분류해서 따로 저장한 것이다.

```
dos.writeUTF(member.getId());
dos.writeUTF(member.getName());
dos.writeInt(member.getAge());
```

다시 처음으로 돌아와서 회원 객체를 자바 컬렉션에 저장하는 예를 보자.

```
public class MemoryMemberRepository implements MemberRepository {

    private final List<Member> members = new ArrayList<>();

    @Override
    public void add(Member member) {
        members.add(member);
    }

    @Override
    public List<Member> findAll() {
        return members;
    }
}
```

자바 컬렉션에 회원 객체를 저장할 때는 복잡하게 회원의 필드를 하나하나 꺼내서 저장할 필요가 없었다. 단순하게 회원 객체를 그대로 자바 컬렉션에 보관하면 된다. 조회할 때도 마찬가지이다.
이렇게 편리하게 회원 객체를 저장할 수 있는 방법은 없을까?

회원 관리 예제4 - ObjectStream

회원 인스턴스도 생각해보면 메모리 어딘가에 보관되어 있다. 이렇게 메모리에 있는 보관되어 있는 객체를 읽어서 파일에 저장하기만 하면 아주 간단하게 회원 인스턴스를 저장할 수 있을 것 같다.
`ObjectStream`을 사용하면 이렇게 메모리에 보관되어 있는 회원 인스턴스를 파일에 편리하게 저장할 수 있다.
마치 자바 컬렉션에 회원 객체를 보관하듯이 말이다.

객체 직렬화

자바 객체 직렬화(Serialization)는 메모리에 있는 객체 인스턴스를 바이트 스트림으로 변환하여 파일에 저장하거나 네트워크를 통해 전송할 수 있도록 하는 기능이다. 이 과정에서 객체의 상태를 유지하여 나중에 역직렬화(Deserialization)를 통해 원래의 객체로 복원할 수 있다.

객체 직렬화를 사용하려면 직렬화하려는 클래스는 반드시 `Serializable` 인터페이스를 구현해야 한다.

Serializable 인터페이스

```
package java.io;

public interface Serializable {
}
```

- 이 인터페이스에는 아무런 기능이 없다. 단지 직렬화 가능한 클래스라는 것을 표시하기 위한 인터페이스일 뿐이다.
- 메서드 없이 단지 표시가 목적인 인터페이스를 마커 인터페이스라 한다.

Member - Serializable 추가

```
package io.member;

import java.io.Serializable;

public class Member implements Serializable {
    private String id;
    private String name;
    private Integer age;
    ...
}
```

- `Member` 클래스에 `Serializable`을 구현한다.
- 이제 이 클래스의 인스턴스는 직렬화 될 수 있다.

만약 해당 인터페이스가 없는 객체를 직렬화 하면 다음과 같은 예외가 발생한다.

```
java.io.NotSerializableException: io.member.Member
```

```
package io.member.impl;

import io.member.Member;
import io.member.MemberRepository;

import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class ObjectMemberRepository implements MemberRepository {

    private static final String FILE_PATH = "temp/members-obj.dat";

    @Override
    public void add(Member member) {
        List<Member> members = findAll();
        members.add(member);

        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(FILE_PATH))) {
            oos.writeObject(members);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public List<Member> findAll() {
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(FILE_PATH))) {
            Object findObject = ois.readObject();
            return (List<Member>) findObject;
```

```
        } catch (FileNotFoundException e) {
            return new ArrayList<>();
        } catch (IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }

}
```

MemberConsoleMain 수정 - ObjectMemberRepository 사용

```
public class MemberConsoleMain {
    //private static final MemberRepository repository = new
MemoryMemberRepository();
    //private static final MemberRepository repository = new
FileMemberRepository();
    //private static final MemberRepository repository = new
DataMemberRepository();
    private static final MemberRepository repository = new
ObjectMemberRepository();
    ...
}
```

실행 결과 - temp/members-obj.dat

파일이 정상 보관된다. 문자와 byte가 섞여 있다.

주의!

만약 예외가 발생하고, 실행이 잘 되지 않는다면 `members-obj.dat` 파일을 삭제하고 다시 시도하자. 잘못된 정보가 저장되어 있을 수 있다.

직렬화

- `ObjectOutputStream`를 사용하면 객체 인스턴스를 직렬화해서 byte로 변경할 수 있다.
- 우리는 회원 객체 하나가 아니라 회원 목록 전체를 파일에 저장해야 하므로 `members` 컬렉션을 직렬화 해야한다.
- `oos.writeObject(members)`를 호출하면 `members` 컬렉션과 그 안에 포함된 `Member`를 모두 직렬화해서 byte로 변경한다. 그리고 `oos`와 연결되어 있는 `FileOutputStream`에 결과를 출력한다.

참고로 `ArrayList`도 `java.io.Serializable`을 구현하고 있어서 직렬화 할 수 있다.

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{...}
```

temp/members-obj.dat 결과

```
??srjava.util.ArrayListx????a?Isizexpwsrio.member.Member????0RMLagetLjava/
lang/Integer;LidtLjava/lang/String;Lnameq~xpsrjava.lang.Integer..???
8Ivaluexrjava.lang.Number???
???xptid1tname1sq~sq~tid2tname2x
```

직렬화를 하면 문자와 byte 정보가 섞여 있다. 잘 보면 `ArrayList`, `Member` 같은 클래스 정보도 함께 저장되는 것을 확인할 수 있다.

역직렬화

- `ObjectInputStream`을 사용하면 byte를 역직렬화 해서 객체 인스턴스로 만들어 수 있다.
- `Object findObject = ois.readObject()`를 사용하면 역직렬화가 된다. 이때 반환 타입이 `Object`이므로 캐스팅해서 사용해야 한다.

정리

객체 직렬화 덕분에 객체를 매우 편리하게 저장하고 불러올 수 있었다.
객체 직렬화를 사용하면 객체를 바이트로 변환할 수 있어, 모든 종류의 스트림에 전달할 수 있다. 이는 파일에 저장하는 것은 물론, 네트워크를 통해 객체를 전송하는 것도 가능하게 한다. 이러한 특성 때문에 초기에는 분산 시스템에서 활용되었다.
그러나 객체 직렬화는 1990년대에 등장한 기술로, 초창기에는 인기가 있었지만 시간이 지나면서 여러 단점이 드러났다. 또한 대안 기술이 등장하면서 점점 그 사용이 줄어들게 되었다.
현재는 객체 직렬화를 거의 사용하지 않는다.

참고로 객체 직렬화 관련해서 다음과 같이 더 학습할 내용들이 있다. 하지만 현대에는 객체 직렬화를 잘 사용하지 않으므로 이런 것이 있다는 것 정도만 알아두고 넘어가자

- `serialVersionUID`: 객체 직렬화 버전을 관리한다.
- `transient` 키워드: `transient`가 붙어있는 필드는 직렬화 하지 않고 무시한다.

XML, JSON, 데이터베이스

회원 객체와 같은 구조화된 데이터를 컴퓨터 간에 서로 주고 받을 때 사용하는 데이터 형식이 어떻게 발전해왔는지 알아보자.

객체 직렬화의 한계

객체 직렬화를 사용하지 않는 이유

- 버전 관리의 어려움
  - 클래스 구조가 변경되면 이전에 직렬화된 객체와의 호환성 문제가 발생한다.
  - serialVersionUID 관리가 복잡하다.
- 플랫폼 종속성
  - 자바 직렬화는 자바 플랫폼에 종속적이어서 다른 언어나 시스템과의 상호 운용성이 떨어진다.
- 성능 이슈
  - 직렬화/역직렬화 과정이 상대적으로 느리고 리소스를 많이 사용한다.
- 유연성 부족
  - 직렬화된 형식을 커스터마이즈하기 어렵다.
- 크기 효율성
  - 직렬화된 데이터의 크기가 상대적으로 크다.

객체 직렬화의 대안1 - XML

```
<member>
  <id>id1</id>
  <name>name1</name>
  <age>20</age>
</member>
```

플랫폼 종속성 문제를 해결하기 위해 2000년대 초반에 XML이라는 기술이 인기를 끌었다.
하지만 XML은 매우 유연하고 강력했지만, **복잡성**과 **무거움**이라는 문제가 있었다. 태그를 포함한 XML 문서의 크기가 커서 네트워크 전송 비용도 증가했다.

객체 직렬화의 대안2 - JSON

```
{ "member": { "id": "id1", "name": "name1", "age": 20 } }
```

JSON은 **가볍고 간결하며**, 자바스크립트와의 자연스러운 호환성 덕분에 웹 개발자들 사이에서 빠르게 확산되었다.
2000년대 후반, **웹 API와 RESTful 서비스**가 대중화되면서 JSON은 표준 데이터 교환 포맷으로 자리 잡았다.

XML은 데이터 구조의 복잡성과 엄격한 스키마 정의가 필요한 초기 웹 서비스와 엔터프라이즈 환경에서 중요한 역할을 했지만, 시간이 지나면서 JSON과 같은 가볍고 효율적인 데이터 형식이 더 많이 채택되었다. JSON은 웹과 모바일 애플리케이션의 발전과 함께 급속히 인기를 얻었으며, 현재는 대부분의 데이터 교환에서 기본적인 포맷으로 사용되고 있다. XML은 특정 영역에서 여전히 사용되지만, JSON이 현대 소프트웨어 개발의 주류로 자리 잡았다.
지금은 웹 환경에서 데이터를 교환할 때 JSON이 사실상 표준 기술이다.

객체 직렬화의 대안3 - Protobuf, Avro - 더 적은 용량, 더 빠른 성능

- JSON은 거의 모든 곳에서 호환이 가능하고, 사람이 읽고 쓰기 쉬운 텍스트 기반 포맷이어서 디버깅과 개발이 쉽다.
- 만약 매우 작은 용량으로 더 빠른 속도가 필요하다면 Protobuf, Avro 같은 대안 기술이 있다.
- 이런 기술은 호환성은 떨어지지만 byte 기반에, 용량과 성능 최적화가 되어 있으므로 매우 빠르다.
- 다만 byte 기반이므로 JSON처럼 사람이 직접 읽기는 어렵다.

정리

- 자바 객체 직렬화는 대부분 사용하지 않는다.
- JSON이 사실상 표준이다. JSON을 먼저 고려하자.
- 성능 최적화가 매우 중요하다면 Protobuf, Avro 같은 기술을 고려하자. (대부분 JSON만 사용해도 충분하다)

참고: JSON을 생성하고 사용하는 내용은 스프링 MVC 강의에서 다룬다.

데이터베이스

이번에는 회원과 같은 구조화된 데이터를 보관하는 방법에 대해서 고민해보자.
앞서 설명한 것 처럼 회원 객체 같은 구조화된 데이터를 주고 받을 때는 JSON 형식을 주로 사용한다.
하지만 어떤 형식이든 데이터를 저장할 때, 파일에 데이터를 직접 저장하는 방식은 몇 가지 큰 한계가 있다.

- 첫째, 데이터의 무결성을 보장하기 어렵다. 여러 사용자가 동시에 파일을 수정하거나 접근하려고 할 때, 데이터의 충돌이나 손상 가능성이 높아진다. 이러한 경우, 데이터의 일관성을 유지하는 것이 매우 어렵다.
- 둘째, 데이터 검색과 관리의 비효율성이다. 파일에 저장된 데이터는 특정 형식 없이 단순히 저장될 수 있기 때문에, 필요한 데이터를 빠르게 찾는 데 많은 시간이 소요될 수 있다. 특히, 데이터의 양이 방대해질수록 검색 속도는 급격히 저하된다.
- 셋째, 보안 문제이다. 파일 기반 시스템에서는 민감한 데이터를 안전하게 보호하기 위한 접근 제어와 암호화 등이 충분히 구현되지 않을 수 있다. 결과적으로, 데이터 유출이나 무단 접근의 위험이 커질 수 있다.
- 넷째, 대규모 데이터의 효율적인 백업과 복구가 필요하다.

이러한 문제점들을 하나하나 해결하면서 발전한 서버 프로그램이 바로 데이터베이스다.

이러한 문제점들 때문에 대부분의 현대 애플리케이션에서는 데이터베이스를 사용한다. 데이터베이스는 위의 한계들을 극복하고, 대량의 데이터를 효율적으로 저장, 관리, 검색할 수 있는 강력한 도구를 제공한다.
예를 들어서, 특정 ID를 가진 회원을 찾는다고 가정해보자. 파일에 직접 저장하는 경우 파일의 모든 내용을 다 찾아볼 것이다. 데이터베이스는 인덱스라는 것만 간단하게 지정하면 매우 빠른 속도로 회원을 찾을 수 있다.

이런 이유로 실무에서는 대부분의 데이터를 파일에 저장하지 않고 데이터베이스에 저장한다.
참고로 이미지, 영상처럼 큰 데이터는 파일로 보관하고, 그 외의 데이터는 거의 모두 데이터베이스에 보관한다고 생각하면 된다. 고객 정보, 주문 내역, 결제 내역 등등 거의 대부분의 데이터는 데이터베이스에 보관한다.

데이터베이스만 해도 하나의 큰 분야이다. 백엔드 개발자가 목표라면 데이터베이스는 반드시 깊이있게 학습해야 한다.

참고: 데이터베이스는 자바 로드맵 이후에 다룰 예정이다.

정리