객체 지향

문제 1.

- 수강 신청 로직 구현
    - 학생(Student)이 여러 과목 수강 신청
    - 학생은 수강된 과목 확인
    - 과목(Course): 과목 이름
    - 여러 과목이 개설
- 객체 추출, 객체 간 관계

```
public class Course {
  private String name; // 과목명

  public Course() {
  }

  public Course(String name) {
    super();
    this.name = name;
  }

  public String getName() {
    return name;
  }

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

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

public class Student {
  private String name;
  private List<Course> courses; // 학생이 수강 신청한 과목들

  // 수강 신청 메서드
  // 파라미터로 course 객체가 넘어온다
  public void register(Course course) {
    courses.add(course);
  }

  // 수강 신청한 학생 목록 출력 메서드
  public void printMember() {
    System.out.println("학생 이름: " + name);
    // 과목의 이름 가져오기
    for(Course course: courses) {
      System.out.println("수강 과목: " + course.getName());
    }
  }

  public Student() {
  }

  // 코스 리스트 만들기
  public Student(String name) {
    super();
    this.name = name;
    // ArrayList 만들어 주기
    courses = new ArrayList<Course>();
  }

  public String getName() {
```

```
    return name;
  }

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

  public List<Course> getCourses() {
    return courses;
  }

  public void setCourses(List<Course> courses) {
    this.courses = courses;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Course c1 = new Course("심리학의 이해");
    Course c2 = new Course("발성");
    Course c3 = new Course("전산학개론");

    Student s1 = new Student("홍길동");
    Student s2 = new Student("장길산");

    s1.register(c1);
    s1.register(c2);

    s2.register(c2);
    s2.register(c3);
    s2.register(c1);

    s1.printMember();
    s2.printMember();
  }
}
```

문제 2.

- 학생은 수강 신청한 과목을 취소할 수 있다.
- 과목마다 수강 신청한 학생 목록을 확인할 수 있다.

```
Student.class

// 수강 신청 취소 메서드
  public void dropcourse(Course course) {
    // 해당 코스를 포함하고 있다면 삭제하기
    if(courses.contains(course)) {
      courses.remove(course);
      // 메서드 사용할 때 학생도 제거하기
```

```
      course.removeStudent(this);
    }
  }

  // 수강 신청 메서드 + 학생 포함
  // 파라미터로 코스 객체가 넘어온다
  public void register(Course course) {
    courses.add(course);
    // 현재 자기 자신의 객체인 this 넣어 주기
    course.addStudnt(this);
  }
```

```
package kosa.relation;

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

public class Course {
  private String name; // 과목명
  private List<Student> students; // 학생 리스트 만들어 주기

  public Course() {
  }

  public Course(String name) {
    super();
    this.name = name;
    // 학생 넣을 Arraylist
    students = new ArrayList<>();
  }

  // 학생 추가 메서드
  public void addStudnt(Student student) {
    students.add(student);
  }

  // 학생 삭제 메서드 (drop할 때 메서드 함께 호출)
  public void removeStudent(Student student) {
    students.remove(student);
  }

  // 학생 목록 출력
  public void printCourse() {
    System.out.println("과목명: " + name);
    for(Student student: students) {
      System.out.println("해당 과목 수강 학생: " + student.getName()+"\n");
    }
  }

  public String getName() {
    return name;
  }

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

```
  }

  public List<Student> getStudents() {
    return students;
  }

  public void setStudents(List<Student> students) {
    this.students = students;
  }
}
```

```
package kosa.relation;

public class Main {

  public static void main(String[] args) {
    Course c1 = new Course("심리학의 이해");
    Course c2 = new Course("발성");
    Course c3 = new Course("전산학개론");

    Student s1 = new Student("홍길동");
    Student s2 = new Student("장길산");

    s1.register(c1);
    s1.register(c2);
    s1.dropcourse(c2);

    s2.register(c2);
    s2.register(c3);
    s2.register(c1);
//    s1.printMember();
//    s2.printMember();

    // 해당 과목 수강한 학생들 출력
    c1.printCourse();
    c2.printCourse();
    c3.printCourse();
  }
}
```

상속

- 자식 클래스는 부모 클래스에 없는 필드와 메서드를 정의하여 기능을 추가할 수 있다.
- 오버라이딩: 상위 클래스에 정의된 메서드를 재정의하여 다르게 동작시킬 수 있다.
  - 오버라이딩을 하지 않으면 다형성을 구현할 수 없다.

super

- super: 부모 클래스.
- `super(name)`: super 함수 → 부모 생성자를 호출.
  - 생성자 첫 라인에서만 호출 가능하다.
- `super.` : 부모의 메서드 호출.

```
public Manager(String name) {
    super(name);
}

public double getsalary() {
    return super.getsalary() + bonus;
}
```

다형성

- 오버로딩: 메소드명이 동일하고 파라미터 값이 다른 것.
- 오버라이딩: 부모 클래스 메소드 재정의.

문제 1.

- 마이너스 통장 상속: 한도를 멤버 변수로 가짐.
- 한도+잔액에서 인출 금액 빼기.

```
package kosa.oop.ans;

public class Account {
  // 계좌 객체

  // 상태(필드): 계좌번호, 계좌주, 잔액 => 멤버 변수
  private String accountNo;
  private String ownerName;
  private int balance;

  // 기능(메서드): 입금하다, 출금하다
  public void deposit(int amount) {
    balance += amount;
  }

  // 기본 생성자 만들어 주는 습관 들이기~
  public Account() { // 기본 생성자
  }

  // 새로운 객체가 생성될 때마다 생성자 호출
  public Account(String accountNo, String ownerName, int balance) {
    super();
```

```
    this.accountNo = accountNo;
    this.ownerName = ownerName;
    this.balance = balance;
  }

  public int withdraw(int amount) throws Exception {
    if (balance < amount) {
      throw new Exception("잔액 부족");
    }
    balance -= amount;
    return amount;
  }

  public void printAccount() {
    System.out.println("계좌번호: " + accountNo);
    System.out.println("계좌 주인: " + ownerName);
    System.out.println("잔액: " + balance);
    System.out.println();
  }

  public String getAccountNo() {
    return accountNo;
  }

  public void setAccountNo(String accountNo) {
    this.accountNo = accountNo;
  }

  public String getOwnerName() {
    return ownerName;
  }

  public void setOwnerName(String ownerName) {
    this.ownerName = ownerName;
  }

  public int getBalance() {
    return balance;
  }

  public void setBalance(int balance) {
    this.balance = balance;
  }
}
```

```
package kosa.oop.ans;

public class MinusAccount extends Account {
  int limit;

  // 기본 생성자
  public MinusAccount() {
  }

  public MinusAccount(String accountNo, String ownerName, int balance, int limit) {
```

```
    super(accountNo, ownerName, balance);
    this.limit = limit;
  }

  public int getLimit() {
    return limit;
  }

  public void setLimit(int limit) {
    this.limit = limit;
  }

  @Override
  public int withdraw(int amount) throws Exception {
    if(limit + getBalance() < amount) {
      throw new Exception("잔액 부족");
    }
    // get set 메서드로 잔액 가져오기
    int balance = getBalance();
    setBalance(balance - amount);

    return amount;
  }

@Override
  public void printAccount() {
    super.printAccount();
    System.out.println("남은 한도액: " + limit);
  }
}
```

```
public class AccountMain {
  public static void main(String[] args) {
  Account account = new MinusAccount("1232434-232323", "김소현", 5000, 15000);
    try {
      account.withdraw(18000);
    } catch (Exception e) {
      e.printStackTrace();
    }
    account.printAccount();
  }
}
```

레퍼런스 타입

상속

- 대부분은 기존에 있는 JAVA API 활용한다.

```
예) class MyException extends Exception
    class MyServlet extends HttpServlet
```

- 부모의 데이터 타입으로 형 변환 가능하다.
- 부모가 자식으로 변환할 때는 캐스팅이 필요하다.
- 처음부터 부모 객체로 생성한 것을 자식으로 형 변환 할 수는 없다.

```
예) Car car = new Ambulance(1, blue, John, St.Mary hospital);
    Ambulance am = (Ambluance) car; => 가능

    Car car = new Car(2, red, Peter);
    Ambulance am = (Ambulance) car; => 불가능
```

instnaceof 연산자

- 형 변환 가능한지 확인하는 연산자.

```
Car car = new Ambulance(1, blue, John, St.Mary hospital);
if(car instanceof Ambulance)
  ~
else ~
```

문제 1.

- phone package에 추가
- Company.class → String dept, String position
- University.class → String major, int year
- 메뉴 1. 추가 → 1. 일반 2. 회사 3. 동창
- 메뉴 2. 출력 → 1. 전체 2. 회사 3. 동창

```
package kosa.phone;

public class Company extends PhoneInfo {
  String dept;
  String position;

  public Company() {
  }

  public Company(String name, String phoneNo, int birth, String dept, String position) {
```

```
    super(name, phoneNo, birth);
    this.dept = dept;
    this.position = position;
  }

  @Override
  public void show() {
    super.show();
    System.out.println("부서: " + dept);
    System.out.println("지위: " + position);
    System.out.println();
  }

  public String getDept() {
    return dept;
  }

  public void setDept(String dept) {
    this.dept = dept;
  }

  public String getPosition() {
    return position;
  }

  public void setPosition(String position) {
    this.position = position;
  }
}
```

```
package kosa.phone;

public class University extends PhoneInfo {
  String major;
  int year;

  public University() {
  }

  public University(String name, String phoneNo, int birth, String major, int year) {
    super(name, phoneNo, birth);
    this.major = major;
    this.year = year;
  }

  @Override
  public void show() {
    super.show();
    System.out.println("전공: " + major);
    System.out.println("학번: " + year);
  }

  public String getMajor() {
    return major;
  }
```

```
  public void setMajor(String major) {
    this.major = major;
  }

  public int getYear() {
    return year;
  }

  public void setYear(int year) {
    this.year = year;
  }
}
```

```
package kosa.phone;

import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    // 1. 추가 2. 출력 3. 검색 4. 종료
    Manager m = new Manager();
    Scanner sc = new Scanner(System.in);

    while (true) {
      System.out.println("기능을 선택해 주세요.");
      System.out.println("1. 추가  2. 출력  3. 검색  4. 종료");
      System.out.println("번호: ");
      String num = sc.nextLine();
      // case문 사용
      switch (num) {
      case "1":
        m.addPhoneInfo();
        break;
      case "2":
        m.listPhoneInfo();
        break;
      case "3":
        m.findPhoneInfo();
        break;
      case "4":
        System.out.println("기능을 종료합니다.");
        return;
      }
    }
  }
}
```

```
package kosa.phone;

import java.util.Scanner;

public class Manager {
```

```
PhoneInfo arr[] = new PhoneInfo[10];
static Scanner sc = new Scanner(System.in);
int count;

public void addPhoneInfo() {
  System.out.println("1.일반 2.회사 3.동창");
  System.out.print("선택: ");
  String menu = sc.nextLine();

  System.out.print("이름: ");
  String name = sc.nextLine();
  System.out.print("전화번호: ");
  String phoneNo = sc.nextLine();
  System.out.print("생년월일: ");
  String birth = sc.nextLine();

  // 여기까지는 코드 재활용

  switch (menu) {
  case "1":
    arr[count++] = new PhoneInfo(name, phoneNo, birth);
    break;
  case "2":
    System.out.print("부서: ");
    String dept = sc.nextLine();
    System.out.print("직책: ");
    String position = sc.nextLine();
    arr[count++] = new Company(name, phoneNo, birth, dept, position);
    break;
  case "3":
    System.out.print("학과: ");
    String major = sc.nextLine();
    System.out.print("학번: ");
    String year = sc.nextLine();
    arr[count++] = new University(name, phoneNo, birth, major, year);
    break;
  }
}

public void listPhoneInfo() {
  System.out.println("1.일반 2.회사 3.동창");
  System.out.print("선택: ");
  String menu = sc.nextLine();

  switch (menu) {
  case "1":
    for(int i=0;i<count;i++) {
      arr[i].show();
    }
    break;
  case "2":
    for(int i=0;i<count;i++) {
     // instancof로 판별
      if(arr[i] instanceof Company) {
        arr[i].show();
      }
    }
    break;
```

```
    case "3":
      for(int i=0;i<count;i++) {
        if(arr[i] instanceof University) {
          arr[i].show();
        }
      }
      break;
    }
  }

  public void searchPhoneInfo() {
    System.out.print("이름: ");
    String name = sc.nextLine();
    int idx = -1;

    for(int i=0;i<count;i++) {
      if(name.equals(arr[i].getName())) {
        arr[i].show();
        idx = i;
        break;
      }
    }
    if(idx == -1) {
      System.out.println("대상이 없습니다.");
    }
  }
}
```

추상 클래스(abstract class)

- 코드의 독립성과 일관성을 위해 사용한다.
- 중복 코드 최소화 목적.
- 추상 메서드를 하나 이상 가지고 있다.
- 객체를 생성할 수 없다.
    - 상속을 통하여 추상 메서드를 구현하여 객체를 생성할 수 있다.
- 추상 클래스를 상속받은 클래스는 추상 메서드를 반드시 구현해야 한다.

```
public class Person {
  private String name;
  private Role role;

  public Person() {
  }

  public Person(String name, Role role) {
    super();
    this.name = name;
    this.role = role;
```

```
  }

  // work 객체의 doing 메서드 호출
  public void doIt() {
    // 코드의 일관성 유지
    role.doing();
  }
}
```

```
public abstract class Role {
  // 반드시 doing이라는 메서드를 가지도록 정의
  // 추상 메서드: 설계도 역할
  public abstract void doing();
}
```

```
public class Study extends Role {
  @Override
  public void doing() {
    System.out.println("Studying");
  }
}
```

```
public class Work extends Role {
  @Override
  public void doing() {
    System.out.println("Working");
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    // Work => Study로 변경
    Work work = new Work();
    Study study = new Study();
    // 파라미터 값만 변경하면 쉽게 변경 가능
    // 코드의 일관성, 독립성 유지
    Person person = new Person("홍길동", work);
    person.doIt();
  }
}
```

문제 1.

- BoardService → insertBoard() { dao.insert() };

- OracleDao → insert()
- MySQLDao → insert()

인터페이스

- 인터페이스는 설계와 구현 완전히 분리

interface Speakable - String speak()//추상메소드 -- 1

class Man - String name - 생성자(String name) - getName() -- 2 (name 리턴)

class Reader extends Man implements Speakable -- 3
- speak(){getName + "자바화이팅"}

class Work extends Man implements Speakable -- 4
- speak(){getName + "열심히 일해야 한다."}

class Student extends Man -- 5
- speak(){getName + "공부나 열심히해"} //Student클래스의speak는 오버라이딩과 아무 관계가 없는 메소드

main() -- 6
Object obj[] = {new Reader("둘리"), new Work("길동"),
new Student("마이콜")};

speak() -> 호출결과 -> Student를 제외한 Reader와 Work 만 출력

```
public interface Speakable {
  public String speak();
}
```

```
public class Man {
  private String name;

  public Man(String name) {
    this.name = name;
  }

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

```
  }

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

```
public class Reader extends Man implements Speakable{
  public Reader(String name) {
    super(name);
  }

  @Override
  public String speak() {
    return getName() + ", 읽어라!";
  }
}
```

```
public class Work extends Man implements Speakable{
  public Work(String name) {
    super(name);
  }

  @Override
  public String speak() {
    return getName() + ", 일해라!";
  }
}
```

```
public class Student extends Man {
  public Student(String name) {
    super(name);
  }

  public String speak() {
    return getName() + ", 자바 파이팅";
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Object[] obj = { new Reader("둘리"), new Work("길동"), new Student("마이콜") };

    for (int i = 0; i < obj.length; i++) {
      if (obj[i] instanceof Speakable) {
        // 변환을 해 줘야 speak() 메서드가 사용 가능하다
        // Object는 해당 메서드 가지고 있지 않다
        Speakable sp = (Speakable) obj[i];
```

```
        System.out.println(sp.speak());
      }
    }
  }
}
```

객체 복제

- 메서드를 통해 복제하면 다른 인스턴스를 가리키게 만들 수 있다.
- Deep copy로 address도 객체 복사하면 같은 값을 참조하지 않게 된다.

내부 클래스

```
class A {
  class B {
  }
}

// B 선언
A.B b;

// 예시
Map map = new HashMap();
// <key, value> 여러 쌍을 넣을 수 있다
// 한 쌍만 가지고 있는 데이터 타입이 Map.Entry
// 즉, Map이라는 인터페이스 안에 Entry라는 내부 인터페이스가 있는 것
Map.Entry;
```

- B라는 클래스가 A 클래스 바깥에서는 의미가 없을 때 사용한다.

익명 내부 클래스

- 일반적인 경우

```
// 인터페이스 -> 객체 생성 -> 메서드 호출 이런 식으로 사용
class A implements 인터페이스
// 메서드 오버라이딩 위하여 객체 생성
A a = new A();
a.메서드();
```

- 익명 내부 클래스는 해당 인터페이스(추상 클래스)를 `1회만` 사용해야 하는 경우 사용한다.

- 매번 객체를 새로 생성하기 번거롭기 때문이다.

```
// 앞선 kosa.oop2 interface 예제 참고
// new Role()이 익명 내부 클래스 사용
Person p = new Person("장길산", new Role() {
      @Override
      public void doing() {
        System.out.println("Driver Role");
      }
    });
```

- 람다식은 추상 메서드가 하나일 때만 사용한다.

람다식

예제

- MyType을 익명 내부 클래스 및 람다식으로 구현

1. 파라미터 없는 경우

```
public interface MyType {
  public void hello();
}
```

```
public class LamdaExam {
  public static void main(String[] args) {
    // 익명 클래스 구현
    MyType m = new MyType() {
      @Override
      public void hello() {
        System.out.println("익명 클래스입니다.");
      }
    };
    m.hello();

    // 람다식 구현
    MyType m2 = () -> {
      System.out.println("람다식 형식입니다.");
    };
    m2.hello();

    // 향상된 람다식 구현
    MyType m3 = () -> System.out.println("향상된 람다식입니다.");
    m3.hello();
  }
}
```

2. 파라미터 하나 있는 경우

```
public interface YourType {
  public void talk(String message);
}
```

```
    // 파라미터 있는 경우의 람다식 구현
    YourType u2 = (String message) -> {
      System.out.println("메시지: " + message);
    };
    u2.talk("안녕하세요");

    // 파라미터 있는 경우의 향상된 람다식 구현 (괄호 제거)
    YourType u3 = message -> System.out.println("메시지: " + message);
    u3.talk("신기하당");
```

3. 파라미터 여러 개 있는 경우

```
public interface MyNumber {
  // 더 큰 값 return
  int getMax(int num1, int num2);
}
```

```
// 파라미터 여러 개 있는 경우
MyNumber mn = (x, y) -> (x > y) ? x : y;
System.out.println(mn.getMax(10, 15));
```

함수형 프로그래밍

- 함수를 정의하고 이 함수를 데이터 처리부로 보내 데이터를 처리하는 기법.
- java에서는 8부터 지원한다.

람다식: 객체 = (매개 변수, …) → {처리 내용}

- 데이터 처리부는 람다식을 받아 매개변수에 데이터를 대입하고 중괄호를 실행시켜 처리한다.
- 자바는 람다식을 익명 구현 객체(인터페이스 구현 객체)로 반환한다.

4. Runnable 인터페이스 이용

```
// Runnable 인터페이스 이용하여 1부터 10까지 람다식 출력
    Runnable r = () -> {
      for(int i = 0; i <= 10; i++) {
        try {
          Thread.sleep(1000);
        } catch (Exception e) {
          e.printStackTrace();
        }
        System.out.println(i);
      }
    };
    Thread t = new Thread(r);
    t.start();
```

오늘 한 것들

1. 수강 신청 예제 → 객체 간 관계
2. 상속, 오버라이딩, 형 변화, 다형성
    - MinusAccount
    - PhoneInfo
3. 추상클래스, 인터페이스 ⇒ 코드의 독립성, 일관성 보장
    - Role
    - Dao
4. 내부클래스, 람다식 ⇒ 코드 간결하게 만듦

과제

1. 주문 관리 프로그램
2. 문자열 과제
3. 야구 게임: JAVA, JAVASCRIPT 둘 다 해 보기, 객체 설계