프로덕션 코드 리뷰

code convention, format 맞추기

code convention, format을 지키는 것은 내 자신, 다른 개발자와의 소통을 위한 중요한 활동이다.

- 상수 값의 경우 static final, 변수 이름은 대문자

```
private static int RANGE = 9;
private final int MIN_COUNT = 1;
```

- 상수의 위치는? 상수, 클래스 변수, 인스턴스 변수, 생성자 순으로 위치한다.

```
public class Car {

    private int moveIndex;
    private int carNumber;
    private String carName;

    private static final int INIT_POSITION = 0;
    private static int autoIncrease = 0;

    [...]
}
```

- 공백 라인을 의미있게 사용해라. 문맥을 분리하는 부분에 사용한다.

```
public class Main {



    public static void main(String[] args) {


        Scanner scanner = new Scanner(System.in);
        System.out.println("자동차 대수는 몇 대 인가요?");
        [...]


        //경주 시작
        for(Car car: racingGame.getCarList()) {
            racingGame.racing(tryCount, car);
        }

        //결과 출력
        racingGame.printCarsDistance();

    }

}
```

- space도 고려한다.

```
        for (int i=10; i<1000; i++) {
            assertTrue(checkMove(2, i));
            assertTrue(checkMove(20, i));
            assertTrue(checkMove(200, i));
        }
```

- code format 기능은 Eclipse, Intellij 도구들의 formatting 기능을 활용할 것을 추천한다.

네이밍(이름 짓기)

내 자신, 다른 개발자와의 소통을 위해 가장 중요한 활동 중의 하나가 좋은 이름 짓기이다.

객체지향 생활 체조 규칙 5: 줄여쓰지 않는다(축약 금지)

> 누구나 실은 클래스, 메서드, 또는 변수의 이름을 줄이려는 유혹에 곧잘 빠지곤 한다. 그런 유혹을 뿌리쳐라. **축약은 혼란을 야기하며, 더 큰 문제를 숨기는 경향** 이 있다.
> 클래스와 메서드 이름을 한두 단어로 유지하려고 노력하고 문맥을 중복하는 이름을 자제하자.
> 클래스 이름이 Order라면 shipOrder라고 메서드 이름을 지을 필요가 없다.
> 짧게 ship()이라고 하면 클라이언트에서는 order.ship()라고 호출하며, 간결한 호출의 표현이 된다.

Code Complete 2 네이밍 참고

효과적인 이름짓기 문서에 Code Complete 2 네이밍 관련해 잘 정리해 되어 있다.

변수

변수의 이름을 짓는데 있어서 가장 중요한 고려사항은,
변수 이름이 변수가 표현하고 있는 것을 완벽하고 정확하게 설명해야 한다는 것이다.

이름은 가능한 구체적이어야 한다. 모호하거나 하나 이상의 목적으로 사용될 수 있는 일반적인 이름은 보통 나쁜 이름이다.

변수 이름의 길이가 평균적으로 **10~16**일 때 프로그램을 디버깅하기 위해서 들이는 노력을 최소화 할 수 있고, 변수의 평균 길이가 **8~20인** 프로그램은 디버깅하기가 쉽다.

- 너무 긴 이름
  numberOfPeopleOnTheUsOlympicTeam
  numberOfSeatsInTheStadium
  maximunNumberOfPointsInMordernOlympics
- 너무 짧은 이름
  n, np, ntm
  n, ns, nsisd
  m, mp, max, points
- 적당한 이름
  numTeamMembers, teamMemberCount
  numSeatsInStadium, seatCount
  teamPointsMax, pointsRecord

많은 프로그램들은 계산된 값(총계, 평균, 최대값 등)을 보관하는 변수들을 갖는다.
만약 변수의 이름에 **Total, Sub, Average, Max, Min, Record, String, Pointer 등의 한정자를 사용해야 한다면, 이름의 끝**이 이런 수정자를 입력하는 것이 좋다.

- 좋은 예
  revenueTotal
  expenseTotal
  revenueAverage
  expenseAverage
- 나쁜 예
  totalRevenue
  expenseTotal
  revenueAverage
  averageExpense

루틴(함수, 메소드)

1. 루틴이 하는 모든 것을 표현하라
2. 의미가 없거나 모호하거나 뚜렷한 특징이 없는 동사들을 피하라.
3. 루틴 이름을 숫자만으로 구분하지 말라.
4. 함수의 이름을 지을 때, 리턴 값에 대한 설명을 사용하라.
5. 프로시저의 이름을 지을 때, 확실한 의미를 갖는 동사 다음에 객체를 사용하라.
6. 공통적인 연산을 위한 규약을 만들어라.

효과적인 이름짓기 문서가 제안하는 체크리스트

- 변수가 표현하고자 하는 것을 이름이 완벽하고 정확하게 설명하는가?
- 변수가 프로그래밍 언어의 해결책보다는 실세계의 문제를 참조하고 있는가?
- 고민할 필요가 없을 만큼 긴가?
- 계산 값 한정자가 이름의 마지막에 있는가?
- Num 대신 Count나 Index를 사용하는가?
- 루프 인덱스의 이름이 의미가 있는가(루프가 한 줄 이상이거나 중첩되어 있다면 i,j,k가 아닌 다른 것)?
- 모든 '임시' 변수가 보다 의미 있는 이름으로 다시 명명되었는가?
- 불린 변수가 참일 때 그 의미가 분명하도록 명명되었는가?

좋은 이름 짓기 위한 연습 방법

- JDK, Spring 프레임워크와 같이 유명한 오픈소스 코드를 많이 읽는다.
- 동의어, 유사어 사전을 활용해 문맥에 맞는 이름을 찾으려 노력한다.
- 구현하는 프로그래밍 도메인에 대한 지식을 쌓기 위해 노력한다. 도메인 지식을 높아질 수록 좋은 이름을 지을 가능성이 높아진다.

final 사용해 값 변경 막기

- 최근에 등장하는 언어들은 변수의 기본이 불변
- 자바는 final을 활용해 값의 변경 막는 것이 가능

```
public class Car {
    private final String name;
    private int position;

    public RacingCar(final String name) {
        this.name = name;
    }

    [...]
}
```

```java
@Test
public void 경주_영_바퀴() {
    // given
    final RacingService racingService = new RacingService();
    final int numberOfCars = 3;
    final int time = 0;
    final InputView inputView = new InputView(numberOfCars, time);

    // when
    final List<ResultView> views = racingService.race(inputView);

    // then
    assertThat(views).hasSize(0);
}
```

객체를 객체스럽게 사용하도록 리팩토링해라

- Car 클래스를 추가한 후 name과 position을 상태 값으로 가지는 객체를 추가했다.
- 그런데 이 객체는 로직에 대한 구현은 하나도 없고, name과 position에 대한 setter와 getter 메소드만을 가진다.

```
public class Car {
  private String name;

  private int position = 1;

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

  public int getPosition() {
    return position;
  }

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

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

- 상태 값을 가지는 Car가 좀 더 객체다운 역할을 하도록 리팩토링한다.

```
public class Racing {

  [...]

  public int run(Car car) {
    int num = random.nextInt(11);
    int position = car.getPosition();

    if (num >= 4) {
      position++;
    }

    car.setPosition(position);
    position = car.getPosition();
    return position;
  }
}
```

객체의 상태 접근을 제한한다.

- 객체 인스턴스 변수의 접근 제어자는 private으로 구현한다.

```
public class Car {
    public String carName;
    public int move;

    public Car(String carName, int move) {
        this.carName = carName;
        this.move = move;
    }

}
```

단위 테스트하기 어려운 코드 단위 테스트하기

- 다음 코드는 Random 때문에 단위 테스트하기 힘들다. 단위 테스트가 가능하도록 리팩토링한다면 어떻게 리팩토링하는 것이 좋을까?

```
public class Car {
    private static final int FORWARD_NUM = 4;

    private int position;

    [...]

    public void move() {
        if (getRandomNo() >= 4)
            this.position++;
    }

    public int getRandomNo() {
        Random random = new Random();
        return random.nextInt(10);
    }
}
```

테스트 가능한 코드와 테스트하기 힘든 부분을 분리

```
    public void move(int number) {
        if (number >= FORWARD_NUM) {
            position++;
        }
    }
```

interface 사용해 해결

```
public interface MovableStrategy {
    boolean isMove();
}
```

인스턴스 변수의 수를 최소화한다.

- 인스턴스 변수의 수를 최소화할 수 있는 방법을 찾는다.
- 인스턴스 변수에 중복이 있는지를 확인하고 제거할 수 있는 방법을 찾는다.

```
public class RacingGame {
    private List<Car> cars;
    private List<String> winners;

    public List<Car> move(int time) {
        int curCountOfWin = 0;
        for(Car car : cars){
            countOfMove(time, car);
            curCountOfWin = max(car.getCarPosition(), curCountOfWin);
        }

        for(Car car : cars){
            setWinners(curCountOfWin, car);
        }

        return cars;
    }

    [...]
}
```

- 인스턴스 변수의 수를 최소화할 수 있는 방법을 찾는다.
- 다른 인스턴스 변수를 통해 생성할 수 있는 값을 굳이 인스턴스 변수로 구현하지 마라.

```java
public class GameResult {
    private List<Car> carList;
    private int topDistance;

    public List<Car> getCarList() {
        return carList;
    }

    public void setCarList(List<Car> carList) {
        this.carList = carList;
    }

    public void moveCar(int carIndex) {
        Car currentCar = carList.get(carIndex);
        currentCar.setMoveDistance(currentCar.getMoveDistance() + 1);
        topDistance = Math.max(topDistance, currentCar.getMoveDistance());
    }
}
```

setter 메소드 사용을 자제한다.

- 인스턴스를 초기화한 후에 값을 변경할 수 있는 setter 메소드를 생성하지 않는다. 가능하면 생성자를 사용해 초기화한다.

```
public class Car {
    private int carPosition;
    private String carName;

    public void setCarName(String carName){
        this.carName = carName;
    }

    public void movePosition(int ranNum) {
        if(ranNum>=4){
            carPosition++;
        }
    }
}
```

상태 데이터를 get하지 말고 메시지를 보내라.

- 객체의 데이터를 꺼내 로직을 구현하면 중복 코드가 발생한다.
- 객체에 메시지를 보내 상태 데이터를 가지는 객체가 일하도록 하라.

```
    private void addMaxCarPostion(GameResult result,
        int maxCarPosition, Car car) {
        if(maxCarPosition == car.getCarPostion()) {
            result.addWinner(car);
        }
    }
```

비지니스 로직과 UI 로직의 분리

- 비지니스 로직과 UI 로직을 한 클래스가 담당하지 않도록 한다.
- 단일 책임의 원칙에도 위배된다.

```
public class Car {
    private int position;

    [...]

    private void print(int position) {
        StringBuilder sb = new StringBuilder();
        for(int i=0; i<position; i++){
              sb.append("-");
        }
        System.out.println(sb.toString());
    }
}
```

- 현재 객체의 상태를 보기 위한 로그 메시지 성격이 강하다면 toString()을 통해 구현한다.
- View에서 사용할 데이터라면 getter 메소드를 통해 데이터를 전달한다.

Collection 활용 로직 처리

- 다음 코드는 Car 목록에서 최종 우승자를 구하는 로직이다. 이 코드를 Collection 기능을 사용해 어떻게 리팩토링할 것인가?

```java
public class ResultView {
  private Cars cars = null;

  private String getTopRankedCar(List<Car> carList) {
    String topCarString = "";
    cars = new Cars(carList);
    int maxPosition = getMaxPosition(carList);
    for(int i=0; i<carList.size(); i++) {
      if(cars.getPosition(i)==maxPosition) topCarString += cars.getCarName(i) + ", ";
    }
    return topCarString.substring(0, topCarString.length()-2);
  }

  private int getMaxPosition(List<Car> carList) {
    int maxPosition = 0;
    cars = new Cars(carList);
    for(int i=0; i<carList.size(); i++) {
      if(maxPosition < cars.getPosition(i)) maxPosition = cars.getPosition(i);
    }
    return maxPosition;
  }
}
```

- Collection을 활용해 로직을 구현할 때 직접 구현하려 하지 말고 먼저 Collection API를 통해 해결할 수 있는 방법이 있는지 찾는다.
- 방법을 찾았는데 해결 방법을 찾지 못하는 경우만 직접 구현한다.

테스트 코드 리뷰

어느 부분을 테스트할 것인가?

- 어느 정도의 테스트가 적정한가?
- 경계 값을 기준으로 테스트

```
    @Test
    public void 랜덤숫자가_4이상일때만_움직인다() {
        Car car = new Car();

        assertFalse(car.shouldMove(0));
        assertFalse(car.shouldMove(1));
        assertFalse(car.shouldMove(2));
        assertFalse(car.shouldMove(3));

        assertTrue(car.shouldMove(4));
        assertTrue(car.shouldMove(5));
        assertTrue(car.shouldMove(6));
        assertTrue(car.shouldMove(7));
        assertTrue(car.shouldMove(8));
        assertTrue(car.shouldMove(9));
    }
```

Test Fixture 생성

- Fixture란 테스트를 실행하기 위해 필요한 것으로 테스트를 실행하기 위해 준비해야할 것들을 의미한다.
- 테스트의 인스턴스 변수는 각 Test Case에서 공통으로 필요한 Fixture만 위치, 나머지는 각 Test Case에 로컬 변수로 구현한다.

```
public class RacingGameTest {
    private final String[] testNames = {"a", "b", "c"};
    private final int testPosition = 5;
    private final String resultSamePositionString = ", b";
    private final String resultWinnersString = "a, b";
    RacingGame racingGame;
    private final Car firstWinner = new Car(testPosition, "a");
    private Car secondWinner;
    private final List<Car> cars = new ArrayList<Car>();

    [...]
}
```

- @BeforeEach는 각 Test Case에서 중복으로 사용하는 Fixture만 초기화해야 한다.

```
public class StringCalculatorTest {
    private StringCalculator emptyInputStringCalculator;
    private StringCalculator whiteSpaceInputStringCalculator;
    private StringCalculator nullInputStringCalculator;
    private StringCalculator stringCalculator;
    private StringCalculator minArrayLengthStringCalculator;
    private StringCalculator evenArrayLengthStringCalculator;
    ...


    @BeforeEach
    void setUp() {
        System.out.println("StringCalculator Test setUp");
        emptyInputStringCalculator = new StringCalculator("");
        whiteSpaceInputStringCalculator = new StringCalculator(" ");
        nullInputStringCalculator = new StringCalculator(null);
        stringCalculator = new StringCalculator("2 + 3 * 4 / 2");
        minArrayLengthStringCalculator = new StringCalculator("2 +");
        minArrayLengthStringCalculator.splitInputString();
        evenArrayLengthStringCalculator= new StringCalculator("2 + 3 *");
        evenArrayLengthStringCalculator.splitInputString();
        ...
    }
}
```

특정 상태를 만들기 위한 반복 코드

- 우승자 구하는 로직을 테스트하기 위해 Test Fixture 준비

```
public class RacingGameResultTest {
    @Test
    public void check_ranking_if_correct() {
        List<Car> cars = new ArrayList<>();
        Car car1 = new Car("pobi");
        Car car2 = new Car("crong");
        Car car3 = new Car("honux");

        car1.move();
        car1.move();
        car2.move();
        car2.move();
        car2.move();
        car3.move();

        cars.add(car1);
        cars.add(car2);
        cars.add(car3);
        [...]
    }
}
```

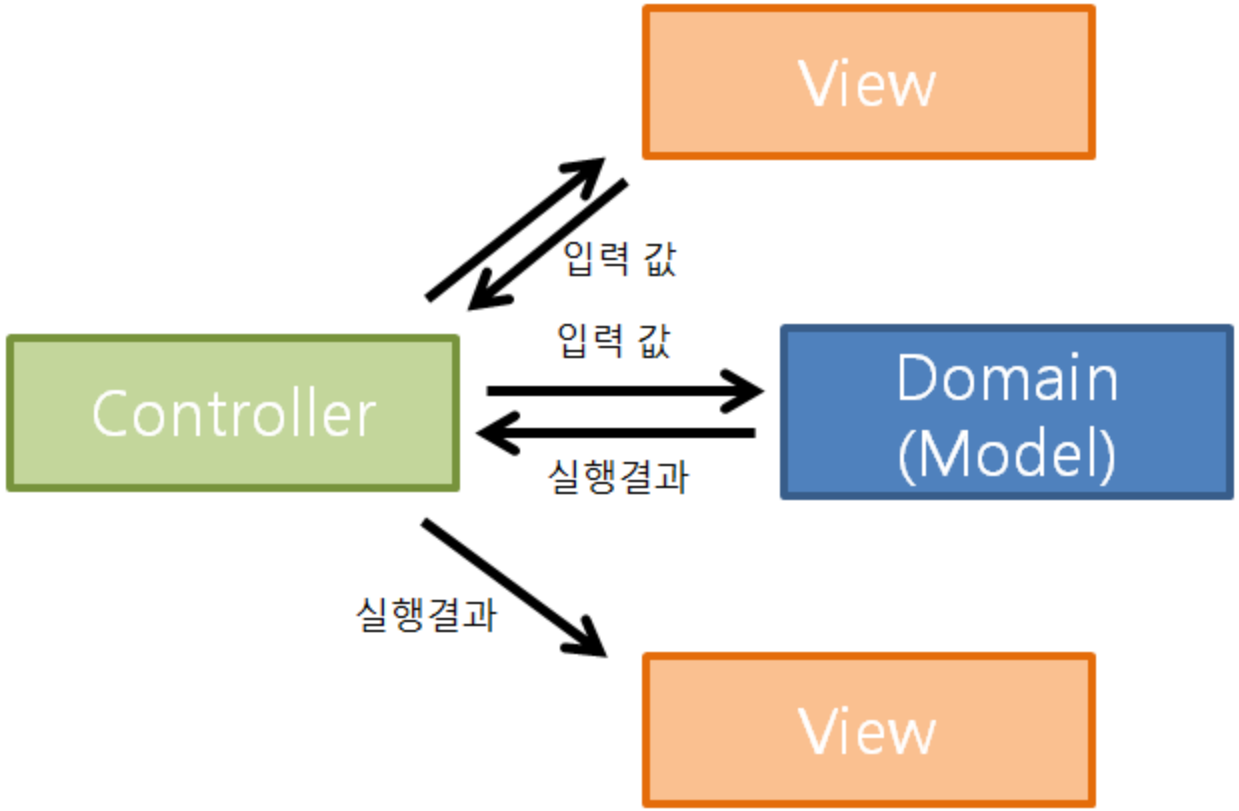
- Test Fixture를 위해 `Car(String name, int position)` 생성자를 추가한다면

```
public class RacingGameResultTest {
    @Test
    public void check_ranking_if_correct() {
        List<Car> cars = Arrays.asList(new Car("pobi", 2), new Car("crong", 3), new Car("honux", 1));

        [...]
    }
}
```

MVC

- 기본 패턴으로 MVC 패턴(Model View Controller) 구조를 유지하면서 구현하면 큰 틀에서의 분리가 가능함.



```
public class RacingMain {
    public static void main(String[] args) {
        String carNames = InputView.getCarNames();
        int tryNo = InputView.getTryNo();

        RacingGame racingGame = new RacingGame(carNames, tryNo);
        while(!racingGame.isEnd()) {
            racingGame.race();
            ResultView.printCars(racingGame.getCars());
        }
        ResultView.printWinners(racingGame.getWinners());
    }
}
```

FAQ

getter 메소드 없이 구현 가능한가?

- setter/getter 메소드를 사용하지 말라는 것은 핵심 비지니스 로직을 구현하는 도메인 객체를 의미한다.
- 도메인 Layer -> View Layer, View Layer -> 도메인 Layer로 데이터를 전달할 때 사용하는 DTO(data transfer object)의 경우 setter/getter를 허용한다.

Car의 생성자에 position을 추가하는 것은 테스트를 위한 변경이 아닌가?

- 맞다. Car가 도메인 객체와 DTO 두 가지 역할을 하기 때문이다.
- Car(도메인 객체)와 CarDto(DTO)로 역할을 분리한다면 position을 가지는 생성자는 CarDto에만 추가해도 된다.
- 두 가지 역할을 분리하지 않을 경우 Car 객체에 position을 가지는 생성자 추가 가능하다.