3주 차 공통 피드백

함수(메서드) 라인에 대한 기준

프로그래밍 요구사항을 보면 함수 15라인으로 제한하는 요구사항이 있다. 이 기준은 **`main()`** 함수에도 해당된다. 공백 라인도 한 라인에 해당한다. 15라인이 넘어간다면 함수 분리를 위한 고민을 한다.

발생할 수 있는 예외 상황에 대해 고민한다

정상적인 경우를 구현하는 것보다 예외 상황을 모두 고려해 프로그래밍하는 것이 더 어렵다. 예외 상황을 고려해 프로그래밍하는 습관을 들인다. 예를 들어 로또 미션의 경우 아래와 같은 예외 상황을 고민해 보고 해당 예외에 대해 처리를 할 수 있어야 한다.

- 로또 구입 금액에 1000 이하의 숫자를 입력
- 당첨 번호에 중복된 숫자를 입력
- 당첨 번호에 1~45 범위를 벗어나는 숫자를 입력
- 당첨 번호와 중복된 보너스 번호를 입력

비즈니스 로직과 UI 로직을 분리한다

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

```
public class Lotto {
    private List<Integer> numbers;

    // 로또 숫자가 포함되어 있는지 확인하는 비즈니스 로직
    public boolean contains(int number) {
        ...
    }

    // UI 로직
    private void print() {
        ...
    }
}
```

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

연관성이 있는 상수는 static final 대신 enum을 활용한다

```
public enum Rank {
    FIRST(6, 2_000_000_000),
    SECOND(5, 30_000_000),
    THIRD(5, 1_500_000),
    FOURTH(4, 50_000),
    FIFTH(3, 5_000),
    MISS(0, 0);

    private int countOfMatch;
    private int winningMoney;

    private Rank(int countOfMatch, int winningMoney) {
        this.countOfMatch = countOfMatch;
        this.winningMoney = winningMoney;
    }
}
```

final 키워드를 사용해 값의 변경을 막는다

최근에 등장하는 프로그래밍 언어들은 기본이 불변 값이다. 자바는 `final` 키워드를 활용해 값의 변경을 막을 수 있다.

```
public class Money {
    private final int amount;

    public Money(int amount) {
        ...
    }
}
```

객체의 상태 접근을 제한한다

인스턴스 변수의 접근 제어자는 private으로 구현한다.

```
public class WinningLotto {
    private Lotto lotto;
    private Integer bonusNumber;

    public WinningLotto(Lotto lotto, Integer bonusNumber) {
        this.lotto = lotto;
```

```
        this.bonusNumber = bonusNumber;
    }
}
```

객체는 객체스럽게 사용한다

Lotto 클래스는 **numbers**를 상태 값으로 가지는 객체이다. 그런데 이 객체는 로직에 대한 구현은 하나도 없고, **numbers**에 대한 getter 메서드만을 가진다.

```
public class Lotto {
    private final List<Integer> numbers;

    public Lotto(List<Integer> numbers) {
        this.numbers = numbers;
    }

    public int getNumbers() {
        return numbers;
    }
}

public class LottoGame {
    public void play() {
        Lotto lotto = new Lotto(...);

        // 숫자가 포함되어 있는지 확인한다.
        lotto.getNumbers().contains(number);

        // 당첨 번호와 몇 개가 일치하는지 확인한다.
        lotto.getNumbers().stream()...
    }
}
```

Lotto에서 데이터를 꺼내지(get) 말고 메시지를 던지도록 구조를 바꿔 데이터를 가지는 객체가 일하도록 한다.

```
public class Lotto {
    private final List<Integer> numbers;

    public boolean contains(int number) {
        // 숫자가 포함되어 있는지 확인한다.
        ...
    }
```

```
    public int matchCount(Lotto other) {
        // 당첨 번호와 몇 개가 일치하는지 확인한다.
        ...
    }
}

public class LottoGame {
    public void play() {
        Lotto lotto = new Lotto(...);
        lotto.contains(number);
        lotto.matchCount(...);
    }
}
```

(**참고**. getter를 사용하는 대신 객체에 메시지를 보내자)

필드(인스턴스 변수)의 수를 줄이기 위해 노력한다

필드(인스턴스 변수)의 수가 많은 것은 객체의 복잡도를 높이고, 버그 발생 가능성을 높일 수 있다.
필드에 중복이 있거나, 불필요한 필드가 없는지 확인해 필드의 수를 최소화한다.
예를 들어 총상금 및 수익률을 구하는 다음 객체를 보자.

```
public class LottoResult {
    private Map<Rank, Integer> result = new HashMap<>();
    private double profitRate;
    private int totalPrize;
}
```

위 객체의 **`profitRate`**와 **`totalPrize`**는 등수 별 당첨 내역(**`result`**)만 있어도 모두 구할 수 있는 값이다. 따라서 위 객체는 다음과 같이 하나의 필드만으로 구현할 수 있다.

```
public class LottoResult {
    private Map<Rank, Integer> result = new HashMap<>();

    public double calculateProfitRate() { ... }

    public int calculateTotalPrize() { ... }
}
```

성공하는 케이스 뿐만 아니라 예외에 대한 케이스도 테스트한다

테스트를 작성하면 성공하는 케이스에 대해서만 고민하는 경우가 있다. 하지만 예외에 대한 부분 또한 처리해야 한다. 특히 프로그램에서 결함이 자주 발생하는 부분 중 하나는 경계값이므로 이 부분을 꼼꼼하게 확인해야 한다.

```
@DisplayName("보너스 번호가 당첨 번호와 중복되는 경우에 대한 예외 처리")
@Test
void duplicateBonus() {
    assertThatThrownBy(() ->
            new WinningLotto(new Lotto(List.of(1, 2, 3, 4, 5, 6), 6))
    ).isInstanceOf(IllegalArgumentException.class);
}
```

테스트 코드도 코드다

테스트 코드도 코드이므로 리팩터링을 통해 개선해 나가야 한다. 특히 반복적으로 하는 부분을 중복되지 않게 만들어야 한다. 예를 들어 단순히 파라미터의 값만 바뀌는 경우라면 아래와 같이 테스트할 수 있다.

```
@DisplayName("천원 미만의 금액에 대한 예외 처리")
@ValueSource(strings = {"999", "0", "-123"})
@ParameterizedTest
void underLottoPrice(Integer input) {
    assertThatThrownBy(() -> new Money(input))
            .isInstanceOf(IllegalArgumentException.class);
}
```

테스트를 위한 코드는 구현 코드에서 분리되어야 한다

테스트를 위한 편의 메서드를 구현 코드에 구현하지 마라. 아래의 예시처럼 테스트를 통과하기 위해 구현 코드를 변경하거나 테스트에서만 사용되는 로직을 만들지 않는다.

- 테스트를 위해 접근 제어자를 바꾸는 경우
- 테스트 코드에서만 사용되는 메서드

단위 테스트하기 어려운 코드를 단위 테스트하기

아래 코드는 `Random` 때문에 `Lotto`에 대한 단위 테스트를 하기 힘들다. 단위 테스트가 가능하도록 리팩터링한다면 어떻게 하는 것이 좋을까?

```java
import camp.nextstep.edu.missionutils.Randoms;

public class Lotto {
    private List<Integer> numbers;

    public Lotto() {
        this.numbers = Randoms.pickUniqueNumbersInRange(1, 45, 6);
    }
}

------

public class LottoMachine {
    public void execute() {
        Lotto lotto = new Lotto();
    }
}
```

올바른 로또 번호가 생성되는 것을 테스트하기 어렵다. 테스트하기 어려운 것을 클래스 내부가 아닌 외부로 분리하는 시도를 해 본다.

```java
public class Lotto {
    private List<Integer> numbers;

    public Lotto(List<Integer> numbers) {
        this.numbers = numbers;
    }
}

------

import camp.nextstep.edu.missionutils.Randoms;

public class LottoMachine {
    public void execute() {
        List<Integer> numbers = Randoms
            .pickUniqueNumbersInRange(1, 45, 6);
        Lotto lotto = new Lotto(numbers);
    }
}
```

위 코드는 A 상황을 B로 바꾼 것이다.

A.

`Application`(테스트하기 어려움)
⬇️
`LottoMachine`(테스트하기 어려움)
⬇️
`Lotto`(테스트하기 어려움) ➡️ `Randoms`(테스트하기 어려움)

B.

`Application`(테스트하기 어려움)
⬇️
`LottoMachine`(테스트하기 어려움) ➡️ `Randoms`(테스트하기 어려움)
⬇️
`Lotto`(테스트하기 쉬움)

(**참고**. 메서드 시그니처를 수정하여 테스트하기 좋은 메서드로 만들기)

이처럼 단위 테스트를 할 때 테스트하기 어려운 부분은 분리하고 테스트 가능한 부분을 단위 테스트한다. 테스트하기 어려운 부분은 단위 테스트하지 않아도 된다. 남은 `LottoMachine`은 어떻게 테스트하기 쉽게 바꿀 수 있을지 고민해 본다.

private 함수를 테스트 하고 싶다면 클래스(객체) 분리를 고려한다

가독성의 이유만으로 분리한 private 함수의 경우 public으로도 검증 가능하다고 여겨질 수 있다. public 함수가 private 함수를 사용하고 있기 때문에 자연스럽게 테스트 범위에 포함된다. 하지만 가독성 이상의 역할을 하는 경우, 테스트하기 쉽게 구현하기 위해서는 해당 역할을 수행하는 다른 객체를 만들 타이밍이 아닐지 고민해 볼 수 있다. 다음 단계를 진행할 때에는 너무 많은 역할을 하고 있는 함수나 객체를 어떻게 의미 있는 단위로 분할할지에 초점을 맞춰 진행한다.