기능 목록 구현을 재검토한다

기능 목록을 클래스 설계와 구현, 함수(메소드) 설계와 구현과 같이 너무 상세하게 작성하지 않는다.
클래스 이름, 함수(메소드) input/output은 언제든지 변경될 수 있기 때문이다.
너무 세세한 부분까지 정리하기 보다 구현해야할 기능 목록을 정리하는데 집중한다.
정상적인 경우도 중요하지만 예외적인 상황도 기능 목록에 정리한다.
특히 예외 상황은 시작단계에서 모두 찾기 힘들기 때문에 기능을 구현하면서 계속해서 추가해 나간다.

기능 목록 작성 예시

- 사용자가 입력한 이름을 쉼표 기준으로 분리해야 한다.
- 사용자 이름이 5자 이하인지 검증해야 한다.
- 자동차는 4이상인 경우 전진한다.
- 자동차는 4미만인 경우 정지한다.
- 0 ~ 9 사이의 임의의 값을 생성한다.
- 여러 대의 자동차 중 최대 위치 값을 구한다.
- 최대 위치 값에 해당하는 자동차 목록을 구한다.
- ...

값을 하드코딩하지 마라.

문자열, 숫자 등의 값을 하드코딩하지 마라.
상수(static final)를 만들고 이름을 부여해 이 변수의 역할이 무엇인지 의도를 드러내라.
구글에서 "java 상수"와 같은 키워드로 검색해 상수 구현 방법을 학습하고 적용해 본다.

자동차 경주 게임 미션에도 값을 하드코딩하는 경우가 많아 다시 한번 피드백 항목에 넣었다.
자동차 경주 게임에서 이동/정지 기준인 4, Random 생성 기준이라 할 수 있는 10등이 상수로 구현할 수 있는 대표적인 예다.

축약하지 마라

의도를 드러낼 수 있다면 이름이 길어져도 괜찮다.

코드 리뷰를 하다보니 아직까지 변수 이름, 함수 이름을 축약하는 경우를 종종 볼 수 있었다.
축약하기보다 의도를 드러낼 수 있는 이름을 사용한다.

java api를 적극 활용한다

메소드를 직접 구현하기 전에 java api에서 제공하는 기능인지 검색을 먼저 해본다.
java api에서 제공하는 않을 경우 직접 구현한다.

예를 들어 우승자를 출력할 때 우승자가 2명 이상이면 쉼표(,) 기준으로 출력을 위한 문자열은 다음과 같이 구현 가능하다.

```
List<String> winners = Arrays.asList("pobi", "jason");
String result = String.join(",", winners);
```

배열 대신 java collection을 사용하라

java collection 자료구조(List, Set, Map 등)를 사용하면 데이터를 조작할 때 다양한 api를 사용할 수 있다.

예를 들어 `List<String>` 에 "pobi"라는 값이 포함되어 있는지는 다음과 같이 확인할 수 있다.

```
List<String> winners = Arrays.asList("pobi", "jason");
boolean result = winners.contains("pobi");
```

객체에 메시지를 보내라

상태 데이터를 가지는 객체에서 데이터를 꺼내려(get)하지 말고 객체에 메시지를 보내라.

예를 들어 Car가 우승자인지를 판단하기 위해 최대 이동 거리 값을 가지는 Car인지 판단 기능은?

```
private boolean isMaxPosition(Car car) {
  return car.getPosition() == maxDistance;
}
```

위와 같이 구현하지 않고 다음과 같이 Car에게 메시지를 보내 구현한다.

```
car.isMaxPosition(maxDistance);
```

필드(인스턴스 변수)의 수를 줄이기 위해 노력한다

필드(인스턴스 변수)의 수가 많은 것은 객체의 복잡도를 높이고, 버그 발생 가능성을 높일 수 있다.
필드(인스턴스 변수)에 중복이 있거나, 불필요한 필드가 없는지 확인해 필드의 수를 최소화한다.

예를 들어 우승자를 구하는 다음 객체를 보자.

```
public class Winner {
    private List<Car> cars;
    private List<String> winnerList;
    private int maxDistance;

}
```

위 객체의 maxDistance와 winnerList는 자동차 목록(cars)만 있어도 모두 구할 수 있는 값이다.
따라서 위 객체는 다음과 같이 하나의 인스턴스 변수만으로 구현할 수 있다.

```
public class Winner {
    private List<Car> cars;

    private int getMaxDistance() { ... }

    public List<String> getWinners() { ... }
}
```

비지니스 로직과 UI 로직을 분리해라.

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

```
public class Car {
    private int position;

    // 자동차 이동 여부를 결정하는 비지니스 로직
    public void move(int randomValue) {
    }

    // UI 로직
    private void print(int position) {
        StringBuilder sb = new StringBuilder();
        ...
    }
}
```

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

함수(메소드) 라인에 대한 기준

프로그래밍 요구사항을 보면 함수 15라인으로 제안하는 요구사항이 있다.
이 기준은 main() 함수에도 해당된다.
주석은 가능하면 함수 밖 또는 코드 우측에 추가하는 방식으로 구현한다.
공백 라인도 한 라인에 해당한다.

commit 메시지에 "#번호"를 추가하지 않는다.

github에서 #번호는 다른 이슈 또는 Pull Request를 참조할 때 사용한다.
현재 미션에서는 참조할 이슈 또는 PR이 없기 때문에 commit 메시지에 "#번호"를 추가하지 않는다.

발생할 수 있는 예외케이스에 대해 고민한다

정상적인 경우를 구현하는 것보다 예외 상황을 모두 고려해 프로그래밍하는 것이 더 어렵다.
예외 상황을 고려해 프로그래밍하는 습관을 들인다.

예를 들어 자동차경주게임 미션의 경우...
만약 진행할 게임의 횟수에 음수를 입력한다면
만약 이름에 빈 값을 넣는다면
만약 쉼표를 이름으로 넣는다면과 같은 예외 케이스를 고민해보고 해당 에러에 대해 처리를 할 수 있어야 한다.

주석은 꼭 필요한 경우만 남긴다

메서드의 역할이 무엇인지는 메서드의 이름으로,
변수의 역할이 무엇인지는 변수의 이름으로 알 수 있어야 한다.
주석은 꼭 필요한 것이 아니면 남기지 않는다.

git을 통해 관리할 자원에 대해서도 고려한다.

.class 파일은 java 코드가 있으면 생성할 수 있다. 따라서 .class 파일은 굳이 git을 통해 관리하지 않아도 된다.
intellij의 .idea 폴더, eclipse의 .metadata 폴더 또한 개발 도구가 자동으로 생성하는 폴더이기 때문에 굳이 git으로 관리하지 않아도 된다.
앞으로 git에 코드를 추가할 때는 git을 통해 관리할 필요가 있는지 여부를 고려해볼 것을 추천한다.

Pull Request를 보내기전 브랜치를 확인한다

기능 구현 작업을 fork된 Repository의 master branch가 아닌,
기능 구현을 위해 새로 만든 브랜치에서 작업한 후 PR을 보낸다.