Design Rationale for Ending

I have chosen to implement an abstract class for `Ending` class, so that it provides a module that is open for additions and closed for modifications, thus following the Open/Closed Principle. Furthermore, all methods within the abstract class are usable by `GoodEnding` and `BadEnding`, thus following the Interface Segregation Principle.

Abstraction is followed by information hiding, by declaring attributes `private` or `protected`, and providing getter and setter methods. I have also prevented privacy leaks in getter methods such as `getAllZombies` and `getAllHumans` in `NewWorld` class, to return a shallow copy of the `ArrayList`.

I have also ensure that the subclass is able to do everything the base class can do, so the `GoodEnding` and `BadEnding` class both can for instance, terminate the world running by the `run()` method, since both endings have a common functionality which is to terminate the game (since endings are made to end the game). Thus, this will follow Liskov Substitution Principle. Thus, the following statement is valid:

```
    private Ending goodEnding = new GoodEnding(this);
    private Ending badEnding = new BadEnding(this);
```

DRY principle is followed by encapsulating common functions into methods so that it is reusable to prevent code duplication. Example is `checkAllZombies`, `checkAllHumans`, in `GoodEnding` and `BadEnding`.

Code should have a high usability with the documentation provided in `Ending`, `GoodEnding`, `BadEnding`, `QuitAction`, `NewWorld`, `NewMap`, etc.

The Open/Closed Principle is also applied by introducing the `Ending` base abstract class, where the abstract class is closed for modifications, while open for modifications by extending the `Ending` base class such as with `GoodEnding` and `BadEnding` class.

Finally, I have made sure that queries such as getter methods do not mutate the state, thus following the Command Query Separation.