**객체지향프로그래밍 실습과제**

**제13주 품질 좋은 소프트웨어 만들기 I I**

제13주 품질 좋은 소프트웨어 만들기 I에 이어 World of Zuul이라는 이름의 게임 프로그램을 가지고 실습을 합니다. 엉성하게 만들어진 초기 버전을 조금씩 개선해 나가면서 품질 좋은 소프트웨어를 만들려면 어떻게 해야 하는지 학습하고 있습니다. 지난 주에는 아래와 같은 내용을 학습했습니다.

설명에 14주라고 표현되는 부분이 여러 군데 있는데, 원래 14주에 해야 할 실습을 13주에 하게 되어 주차 번호가 그렇게 된 점을 이해해주세요.

* 반복되는 기능을 메소드로 추상화함으로써 코드 중복 피하기

Printlocationinfo( )

* 클래스 내부 구현이 밖에서 보이지 않게 함, 정보 은닉

Room의 출구 정보, northExit, eastExit, southExit, westExit을 private으로 선언하고 대신 public getExit(direction) 메소드를 제공

* 책임분담원칙 준수 – 데이터를 갖고 있는 클래스가 그 데이터를 이용하는 작업을 담당히게 함

Room에 대한 정보를 요약해 알려주는 기능을 Game이 아닌 Room이 담당하게 함

* Room 클래스에 public getExitString 메소드를 구현함
* 미래 변경에 대비하여 프로그램 구조를 유연하게 설계함

북동남서 외의 다른 방향이 추가되더라도 Room 클래스에 필드를 추가하지 않아도 되도록 함

* northExit, eastExit, southExit, westExit 필드를 없애고 대신 Map<String, Room>을 필드로 둠
* 게임이 바뀌더라도 클래스 사용법은 가능한 한 바뀌지 않게 함

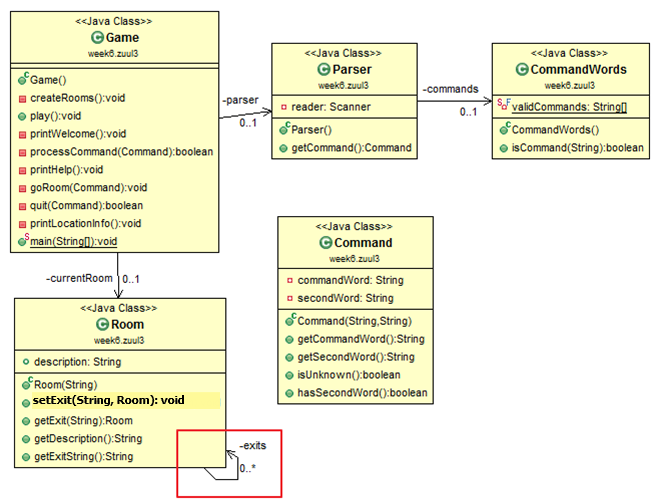
방향이 추가되면 Room 클래스의 아래 메소드(클래스 사용법)도 따라서 바뀌어야 함

public void setExits(Room north, Room east, Room south, Room west)

이를 방지하기 위해 위 메소드를 아래와 같이 수정함

public void setExit(String direction, Room neighbor)

이렇게 해서 얻어진 클래스 다이어그램은 아래와 같습니다.



이번 주는 아직도 부족한 점들을 추가로 개선해 나갑니다.

# **응집성**

지난 주에 스스로 작성한 w13.zuul5 패키지를 "w14.zuul6"라는 이름의 패키지로 복사하여 저장하고 w14.zuul6에서 아래 작업을 진행하세요.

지난 주 모범답안인 [w13.zuul5](src/zuul5.zip) 패키지를 이클립스에 복사해 넣은 후 이것을 "w14.zuul6"라는 이름의 패키지로 복사하여 저장하고 아래 작업을 진행해도 좋습니다.

**간접 결합 Implicit coupling**

맨 처음, Game 클래스가 Room 클래스의 public 필드를 직접 이용하던 시절에는 Game과 Room이 강하게 결합되어 있었습니다. 이 때는 Room의 필드에 변경을 가하면 Game 클래스가 컴파일되지도 않았습니다. 이렇게 표면적으로 드러나는 결합은 더 이상 없도록 지난 번 실습에서 수정을 했습니다.

그러나 이런 표면적으로 드러나는 결합 외에, 겉으로는 잘 드러나지 않는 간접적 결합이 있습니다. 이런 결합이 있는 경우에도 한 클래스를 수정하면 다른 클래스도 수정을 해야 합니다. 우리는 되도록 한 클래스를 수정하더라도 다른 클래스는 수정하지 않아도 되는 설계를 추구합니다. 이렇게 하는 것을 “변화를 지역화한다(localize change)"고 합니다.

1. 우리의 게임은 앞으로도 진화를 거듭해 갈 것입니다. 방을 이리 저리 옮겨 다니는 것만으로는 재미 있는 게임이 될 수 없습니다. 여러 가지 아이템을 방에 배치하고 사용자가 아이템을 이용하게 하거나, 방에 여러 가지 캐릭터를 두어 상호작용을 하도록 하는 등 앞으로 많은 변화가 필요합니다.

Game 클래스의 printLocationInfo( ) 메소드를 살펴보세요. 이 메소드는 Room에게 getDescription 메소드와 getExitString 메소드를 호출하고 그 반환값을 출력합니다. 만약 우리가 Room 클래스에 아이템을 추가한다면 Game 클래스의 printLocationInfo 메소드도 getDescription 메소드와 getExitString 메소드 외에, 가령, getItemString 같은 메소드를 추가로 호출하도록 수정해 줘야 할 것입니다. 즉 Room의 내부 정보가 Game 클래스의 printLocationInfo( ) 메소드 구현에 영향을 미칩니다. 이것이 바로 겉으로 잘 드러나지 않는 **간접 결합**입니다. **이 결합을 없애려면 Room이 자기 책임을 더 잘 하도록 하면 됩니다.**

Room클래스에 getLongDescription() 메소드를 추가하세요. 이 메소드는 Room에 관한 **모든** 정보를 하나의 String으로 반환합니다. 또, Game 클래스의 printLocationInfo( ) 메소드가 이 새로 만든 메소드를 사용하도록 수정하세요.

Room 클래스의 메소드

Text

Description automatically generated

Game 클래스의 메소드

A picture containing graphical user interface

Description automatically generated

실행 예는 아래와 같습니다.

Text

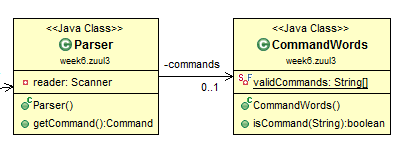
Description automatically generated

1. 이 애플리케이션에는 Parser와 CommandWords 클래스가 있습니다. 두 클래스의 소스 코드를 읽어보세요.

Parser는 사용자 입력을 분석하고, 입력에 해당하는 Command 객체를 만들어주는 일을 합니다. 사용자 입력의 첫 단어가 go, help, quit 등의 명령어가 아니라면 명령어가 틀렸다는 판단을 해야 하고, 올바른 명령어 중 하나라면 그에 해당하는 Command 객체를 생성해야 합니다. 이 일을 하기 위해 Parser는 어떤 명령어들이 가능한지를 알아야 합니다. 그러니까 어떤 명령어들이 정의되어 있는지를 Parser가 자기 **내부에** 데이터로 가지고 있게 클래스를 설계할 **수도** **있었습니다.**

**응집성 (Cohesion)**

그러나, 우리는 처음부터 commandWords라는 클래스를 별도로 두었습니다. 이 클래스는 명령어 목록을 관리하는 역할만을 하며, 어떤 명령어가 있는지 알아보는 데 사용됩니다. Parser는 어떤 단어가 올바른 명령어인지 여부를 판별하기 위해 CommandWords에게 물어봅니다. 만약 “help”가 올바른 명령어인지 여부를 알려면, Parser는 CommandWords에게 inCommand(“help”)를 호출합니다. 그러면 true 혹은 false가 반환됩니다. 이래 그림을 보세요. 녹색 동그라미 오른쪽 위에 “C” 표시가 붙은 것은 Constructor(생성이자)라는 의미입니다. 녹색은 public을 의미합니다. (빨간색은 private입니다.)



Parser는 문장을 분석하는 기능을 하는 놈이고, CommandWords는 명령어 세트를 관리하고 단어가 주어졌을 때 그 단어가 유효한 명령어인지 여부를 판별해주는 기능만을 합니다. 각각의 기능을 위해 별도의 클래스를 둔 것입니다. 이렇게 한 클래스가 하나의 사물 또는 개념만을 나타내도록, 한 메소드가 하나의 기능만을 맡도록 설계하는 것이 좋습니다. 이렇게 설계된 클래스나 메소드는 응집성(cohesion)이 높다고 합니다.

응집성이 높은 설계를 하면 클래스나 메소드를 다른 곳에서 재사용할 수 있을 가능성이 높아집니다. 여러 가지 기능이 합쳐진 엔진은 특정 자동차 모델에서만 사용될 수 있지만, 엔진에 사용되는 (잘 굴러가게 하는 한 가지 일만 하는) 쇠구슬은 여러 가지 자동차 혹은 여러 가지 기계의 부품으로 사용될 수 있습니다.

우리 게임은 현재 세 가지 명령만 인식합니다. 그런데 나중에 명령어를 추가해야 할 경우가 생길 수 있습니다. 가령 “look”라는 명령을 추가해야 한다고 해 봅시다. 다른 방으로 이동하면 바로 그 방 정보가 출력되지만, 이런 저런 다른 일을 하고 나면 화면이 밀려 올라가 방 정보를 더 이상 볼 수 없게 됩니다. 이런 때 “look” 명령어를 입력하면 현재 방의 정보가 다시 보이도록 하고 싶은 것입니다.

“look” 명령어를 추가하려면 당연히 CommandWords 클래스를 수정해야 할 것입니다. 명령어 집합을 관리하는 기능을 CommandWords라는 클래스로 분리하여 구현했기 때문에 Parser와 서로 엉키지 않고 CommandWords 클래스만 수정하여 명령어를 추가할 수 있습니다. 명령어가 저장되어 있는 CommandWords 배열에 “look”라는 명령어 하나를 추가하기만 하면 됩니다. 이렇게 코드를 수정해 보세요.

명령어를 추가하는 일은 여러 클래스에 영향을 주는 아주 복잡한 변경이라고 생각되지만 의외로 간단합니다. CommandWords 내의 배열에 명령어를 추가하고 다른 곳은 건드리지 않았는데도 전체 프로그램이 아무 문제없이 컴파일이 됩니다. 클래스간 결합성이 매우 낮아졌다는 표시입니다. 한 군데를 고쳐도 다른 곳에 영향을 주지 않습니다. 전체 프로그램이 바람직한 방향으로 발전하고 있는 것입니다.

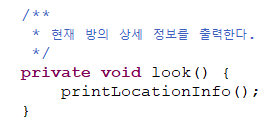
이제 게임을 시작하고 look 명령을 이용해보세요. 명령어를 추가하기만 했지, 아직 그에 해당하는 작업을 하는 코드는 작성하지는 않았기 때문에, “look” 명령에 대해 아무런 반응이 없을 것입니다. 명령어가 아닌 단어를 입력하면 “Unknown command...” 메세지가 나옵니다. look에 대해 이런 메세지가 나오지 않는다는 것은 매우 좋은 징조입니다. “look” 기능이 작동하도록 구현해 주기만 하면 잘 될 것이기 때문입니다.

“look” 명령어가 제대로 작동하게 하려면 Game 클래스의 processCommand 메소드를 수정해야 합니다. 명령을 실행하는 작업이 이 곳에서 이루어지기 때문입니다. 아래 코드를 보세요. 아래 그림의 화살표 부분에 look 명령을 처리하는 구절을 추가해야 합니다.

Text

Description automatically generated

명령이 help일 때는 (이 클래스 내에서만 이용되는 private 메소드인) printHelp 메소드를 호출하고, 명령이 go 일 때는 private goRoom 메소드를 호출하고, 명령이 quit일 때는 private quit 메소드를 호출합니다. 실제 작업은 이 printHelp, goRoom, quit 메소드들이 수행합니다. 이와 마찬가지로, 명령어가 look일 때는 private look 메소드를 호출하도록 하면 됩니다. 당연히 아래 그림과 같은 private look 메소드를 추가로 작성해야 합니다.



컴파일 에러가 없도록 프로그램을 작성하고 게임을 실행하여 look 명령을 이용해 보세요.

Graphical user interface, text, application

Description automatically generated

1. eat 명령을 추가하고 사용자가 “eat”를 입력하면 “Delicious!”라는 메세지가 출력되도록 하세요.

게임을 더 재미 있게 하려면, 음식 아이템을 찾아 먹으면 그 때로부터 일정 시간 동안 배가 고프지 않도록 할 수도 있을 것입니다. 지금은 우선 간단히 명령어 추가가 쉽다는 것만 확인하고 넘어갑니다. 프로그램을 완성하고 look, eat 명령어를 사용해 보세요.

Text

Description automatically generated

1. Game, Parser, CommandWords 클래스 사이의 결합성은 이제 매우 낮아 (좋아) 보입니다 - 확장을 하려고 할 때 매우 쉽게, 빠르게 할 수 있습니다. 그러나 위에서 말한 “간접 결합” 문제가 아직 남아 있는 곳이 있습니다. 게임을 실행하고 help 명령어를 입력해 보세요.

Graphical user interface, text

Description automatically generated

새로 추가한 look, eat 명령어가 보이지 않습니다.

표면에 드러나 보이지는 않지만 Game 클래스의 printHelp 메소드가 간접적으로 CommandWords 클래스의 내부 정보에 연관되어 있는 것입니다. 물론 이 문제는 간단히 해결할 수 있습니다. 아래 Game.printHelp 메소드의 두 번째 프린트 문장에 두 명령어를 추가해 주기만 하면 됩니다.

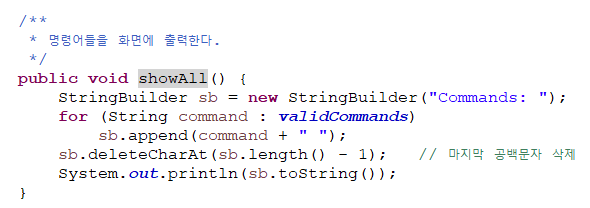
Text

Description automatically generated

그러나 자칫 이런 문제가 있다는 사실 자체를 알아채지 못하고 넘어갈 수가 있습니다. 이런 문제는 생각보다 심각한 문제입니다. 개발 과정에서 미처 발견되지 못하고 현장에 배치될 때까지 이 오류가 남아 있을 가능성이 매우 높기 때문입니다. 사소한 오류라고 하더라도 일단 현장 배치가 되고 난 후 발견되면 이것을 고치기 위해 매우 큰 비용을 치르게 됩니다. 개발 과정에서 철저한 테스트를 함으로써 이런 오류를 사전에 발견하도록 노력해야 하겠지만, 되도록 이런 “간접 결합”이 존재하지 않도록 클래스를 설계하는 게 좋습니다.

여기에서의 간접 결합은 앞서 말한 **책임 분담 원칙 (Responsibility-driven design)**을 지키지 않아서 나타난 문제입니다. help 명령이 들어오면 어떤 명령어들이 가능한지 알려 줘야 합니다. **명령어들을 데이터로 가지고 있으면서 명령어들을 관리하는 놈은 바로 CommandWords 클래스입니다. 그러니까 이 일(어떤 명령어들이 가능한지 알려 주는 일)에 대한 책임도 CommandWords가 져야 옳습니다.** 그런데 지금은 Game 클래스의 printHelp 메소드가 이 일을 스스로 맡아 처리하고 있습니다. 이렇게 하지 말고 Game 클래스의 printHelp 메소드가 CommandWords에게 그 일을 하도록 지시해야 합니다. 그러면 나중에 다른 명령어가 추가되더라도 Game.printHelp 메소드의 **코드 변경 없이** 새로 추가된 명령어를 포함하여 모든 유효한 명령어들을 출력하게 할 수 있게 됩니다.

Game 클래스의 printHelp 메소드가 CommandWords에게 어떤 명령어들이 가능한지 출력하도록 지시하는 데 사용할 아래 메소드를 CommandWords 클래스에 추가하세요. 명령어들을 공백문자(space)로 분리하여 한줄로 출력하고 마지막에 줄바꿈문자를 출력하게 하세요.



이제 Game 클래스의 printHelp 메소드를 수정해야 합니다. 명령어들을 직접 출력하는 대신 CommandWords에게 showAll 메소드를 호출하도록 코드를 수정해야 하겠습니다. 코드를 잘 보고 수정해 보세요. 아래로 내려가지 말고 여기 멈춰서 스스로 수정해 보세요. 소스코드를 메모장에 백업해 놓고 수정을 해 보세요. 잘 안되면 다시 복원하면 됩니다.

?

?

?

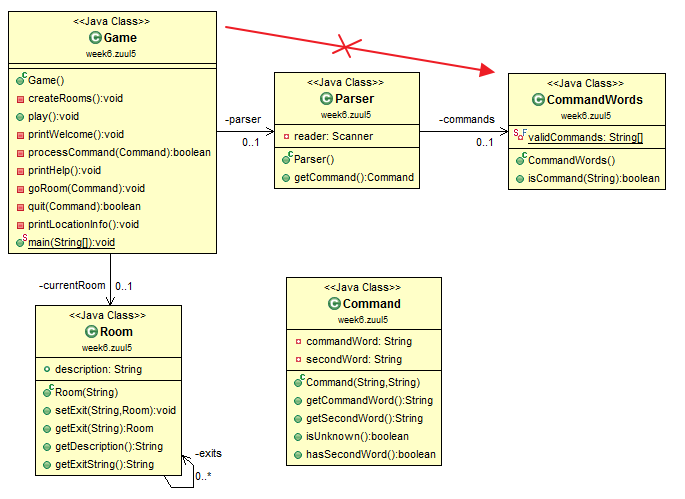
?

?

?

잘 안 되나요? 무엇이 문제입니까?

Game 클래스의 printHelp 메소드에서 CommandWords에게 메소드를 호출해야 하는데, Game은 CommandWords를 가리키는 참조를 가지고 있지 않습니다. CommandWords를 가리키는 참조가 c라면 "c.showAll()" 이렇게 호출해야 하는데, 이 c를 Game 클래스는 가지고 있지 않습니다. 아래 그림을 보면 명확히 보입니다. Game은 CommandWords와 연계(associate)되어 있지 않습니다.



어떻게 하면 될까요? 두 가지 방법이 가능합니다.

1. CommandWords를 가리키는 참조변수를 Game 클래스의 필드(상태변수)로 추가한다.
2. Game은 이미 Parser를 필드로 가지고 있고 Parser는 CommandWords를 사용하고 있으므로 Parser를 통해서 간접적으로 CommandWords의 메소드를 호출한다.

첫 방법을 쓰면 클래스 다이어그램이 바뀝니다. Game으로부터 CommandWords 클래스로 화살표가 새로 생깁니다. 즉, 새로운 직접적 의존관계가 만들어집니다. 클래스 다이어그램에서 화살표가 많아지는 것은 결합도가 높아지는 것을 의미하므로 일반적으로 별로 바람직하지 않습니다. 또, 현재 Parser만이 CommandWords와 연계(associate)되어 있으므로, Parser만이 CommandWords를 가리키는 참조(reference)를 갖고 있습니다. 그러므로 Game이 Parser로부터 이 (CommandWords를 가리키는) 참조를 얻어 오는 방법도 생각해야 합니다.

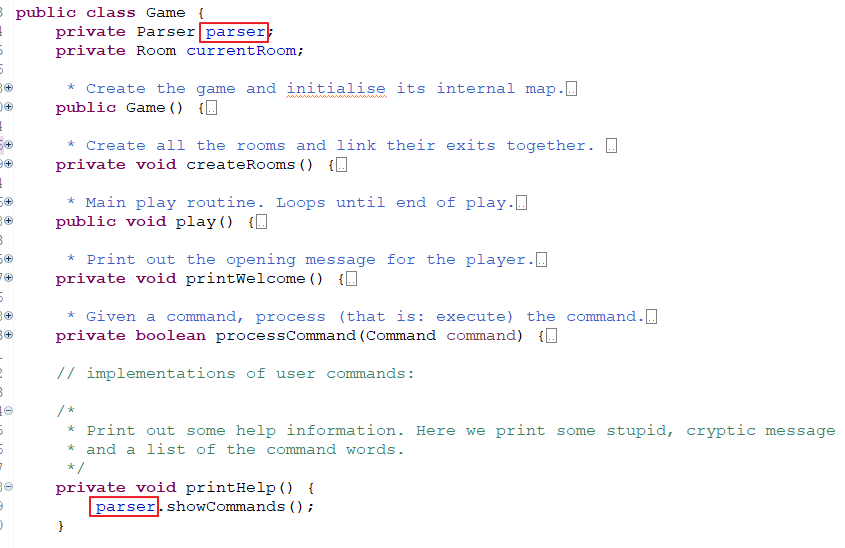
그러니까, 우리는 두 번째 방법을 사용합시다. Game 클래스의 printHelp 메소드는 Parser에게 showCommands라는 메소드를 호출하고, Parser의 showCommands 메소드는 다시 CommandWords의 showAll 메소드를 호출하도록 합니다.

Parser클래스에 showCommands 메소드 추가.

A screenshot of a computer program

Description automatically generated

Game클래스 printHelp 메소드 수정



위와 같이 수정하고 나서 게임을 실행해 보세요. help 명령어가 제대로 작동하나요?

Graphical user interface, text, application

Description automatically generated

이제 어떤 명령어를 CommandWords에 추가하면 그 명령어가 자동으로 help 응답 메세지에 반영됩니다. 음~~ 좋군... 역시 디자인이 중요해!!!

CommandWords.java에 있는 명령어 배열 validCommands에 들어 있는 명령어들의 순서를 위 그림과 같이 go, quit, help, look, eat로 맞춰 주세요(자동채점을 위해).

아래 그림과 같이 작동해야 합니다.

Graphical user interface, text, application

Description automatically generated

**완성된 클래스들을 아래와 같이 강온실에 제출하세요. 패키지 선언문은 넣지 마세요. Import 문장들도 넣지 마세요.**

**강온실 w14p1\_zuul6 문제의 답으로 Parser 클래스를 제출하세요.**

**강온실 w14p2zuul6 문제의 답으로 Game 클래스를 제출하세요.**

# **사용자 인터페이스 집중화**

w14.zuul6 패키지를 "w14.zuul7"라는 이름의 패키지로 복사하여 저장하고 w14.zuul7에서 아래 작업을 진행하세요.

지금은 콘솔 입출력만 가능하지만, 미래에 네트워크를 통한 원격 입출력이 가능하도록 할 수도 있을 것이고, 그래픽 인터페이스를 구현하게 될 지도 모릅니다. **그래서 사용자 인터페이스는 한 클래스 혹은 소수의 클래스 그룹에서만 수행하도록 집중할 필요가 있습니다.** 이렇게 사용자 인터페이스를 한 군데 모아 놓는 것은 애플리케이션을 구현할 때 일반적으로 적용하는 지침입니다.

우리의 경우 사용자가 help 명령을 입력하면 CommandWords 클래스가 직접 화면에 어떤 명령어들이 있는지 **출력**하도록 구현했습니다. 위 지침에 따르면 그렇게 하지 말아야 합니다. 출력은 Game 클래스가 담당하고 나머지 **클래스들은 출력할 내용을 만들어 내어 Game에게 넘겨주도록** 하면 좋겠습니다.

1. CommandWords 클래스의 showAll( ) 메소드가 명령어들을 출력하는 대신 명령어들을 String 타입으로 반환하게 하세요. 메소드 이름도 getCommandList( )로 변경하세요. getCommandList( )는 명령어들을 공백문자로 분리하여 한줄의 String으로 반환하게 하세요. String의 맨 앞에 “Commands: “를 포함시키지 말고, String의 끝에 공백문자나 줄문자를 포함시키지 마세요.
2. Parser 클래스의 showCommands( ) 메소드도 getCommandList( )로 변경하세요. 이 메소드는 CommandWords.getCommandList 메소드로부터 받은 String을 그대로 반환하면 됩니다.
3. Game 클래스의 printHelp( ) 메소드가 Parser의 getCommandList( ) 메소드를 사용하도록 수정하세요. 우선 “Commands: “를 출력하고 그 후에 Parser의 getCommandList( ) 반환값을 출력하게 하세요.

프로그램을 컴파일하고 실행해 보세요. help 명령어를 사용해 보세요. 애플리케이션의 작동에는 변화가 없어야 합니다.

Text

Description automatically generated

**강온실 w14p3\_zuul7 문제의 답으로 Parser와 CommandWords 클래스를 제출하세요. import 문장은 제외하세요.**

# **뒤로 돌아가기**

w14.zuul7 패키지를 "w14.zuul8"라는 이름의 패키지로 복사하여 저장하고 w14.zuul8에서 아래 작업을 진행하세요.

“back” 명령어를 추가하세요. 이 명령을 입력하면 최근에 있던 방으로 돌아가야 합니다. “back” 명령어가 한 단계만 뒤로 갈 수 있게 하면 됩니다. “back”을 하여 한 단계 이전에 있던 방으로 들어간 후 다시 back을 하면 그냥 그 자리에 있도록 하세요. 게임을 시작하자 마자 바로 “back”을 입력하면 그 자리에 있도록 하세요.

“back” 명령어는 한 단계만 뒤로 갈 수 있습니다. 그런데 사용자는 이 명령어를 이용하여 임의의 장소로 돌아갈 수 있다고 생각할 지도 모릅니다. 가령, 현재 Office에 있는데 몇 단계 전에 갔었던 Hall로 돌아가고 싶으면 “back Hall” 이렇게 입력할 지도 모릅니다. 이럴 때 아래와 같은 안내 메시지가 나타나게 하세요.

한 단계 전으로만 돌아갈 수 있습니다.

back 명령어는 두 번째 단어를 가질 수 없습니다.

물론 장소 이동도 없어야 합니다. 실행 예는 아래와 같습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

<힌트>

최근에 있던 방으로 되돌아가려면 최근에 어느 방에 있었는지 기억하고 있어야 합니다. 이를 위해 Game 클래스에 필드를 하나 추가합니다. 현재 있는 방을 가리키는 currentRoom이라는 필드가 있는데 이에 추가로 recentRoom(최근에 있던 방)을 필드로 추가합니다. 방을 이동할 때마다 currentRoom을 recentRoom에 복사하여 저장하고 currentRoom은 새 방을 가리키게 하면 됩니다.

private back 메소드를 새로 만들고, back 명령이 들어오면 back 메소드를 호출하게 합니다. back 명령이 두 번째 단어를 포함하고 있는지 여부를 판별해야 하므로 back 메소드는 Command 타입 파라미터를 갖도록 합니다. back 메소드는 Command를 살펴보아 두 번째 단어가 있으면 오류 메시지를 출력하고 반환합니다. 그렇지 않은 경우에는 recentRoom이 null인지 살펴봅니다. null이 아닌 경우에만 recentRoom을 currentRoom에 넣어 줍니다. (초기에는 recentRoom에 null이 들어있습니다.) 그리고 최종적으로 printLocationInfo를 호출합니다.

back 명령을 한 번 더 내리면 recentRoom과 currentRoom이 같아집니다. 따라서, 연이어 back을 하면 더 이상 움직이지 않게 됩니다.

**강온실 w14p4\_zuul8 문제의 답으로 Game 클래스를 제출하세요.**

-----------------------------------

이상으로, 그다지 좋지 않은 설계를 더 좋은 설계로 개선하는 예를 보면서, 설계할 때 고려해야 할 일반적인 지침들을 살펴보았습니다. 이제 우리는 꽤 잘 설계된, 그래서 미래에 새로운 기능을 추가하더라도 쉽게 적응할 수 있는, 단단하고도 유연한 설계를 갖게 되었습니다. 그러나 이것이 다는 아닙니다. 아직도 약간의 개선이 있을 수 있습니다.

# **아이템 추가**

w14.zuul8 패키지를 "w14.zuul9"라는 이름의 패키지로 복사하여 저장하고 w14.zuul9에서 아래 작업을 진행하세요.

이제 우리는 게임에 아이템을 추가하려고 합니다. 각 방에는 아이템이 하나 있을 수 있습니다. 아이템은 이름(name), 설명(description), 무게(weight) 등의 속성을 갖습니다. 아이템의 무게는 나중에 사용될 것입니다. 플레이어가 들 수 있는 총 무게를 정해 놓고 그 무게 이상의 아이템을 들고 갈 수 없게 할 것입니다. 플레이어가 강해지면 더 무거운 짐을 들 수 있도록 할 수도 있을 것입니다.

게임에 아이템을 추가하기 위해 Room에 아래와 같은 필드를 추가하는 방법이 있습니다.

public class Room {  
 private String description; // 방에 대한 설명  
 private Map<String, Room> exits; // 출구  
 private String itemName; // 이 방에 있는 아이템 이름  
 private String itemDescription; // 이 방에 있는 아이템에 대한 설명  
 private String itemweight; // 이 방에 있는 아이템 무게  
 . . .  
}

그러나, 이렇게 하는 것은 좋은 설계가 아닙니다. **응집성**이 낮아지기 때문입니다. Room 클래스가 방과 아이템 등 두 가지를 모두 나타냅니다. 응집성이 높으려면 한 클래스는 한 가지 사물이나 개념만을 나타내야 합니다.

위 코드와 같이 작성하면 어떤 나쁜 점이 있을까요? 어떤 아이템이 특정 방에 묶여 버립니다. 아이템은 경우에 따라 플레이어가 들고 가야 하는데, 위와 같은 설계에서는 처리가 매우 복잡해집니다. 기존 방의 아이템 관련 필드를 null로 적어주고 새 방의 아이템 관련 필드에 값을 적어줘야 합니다. 또, 나중에 한 방에 여러 개의 아이템이 있게 하려면 필드를 그만큼 추가해야 합니다.

올바른 설계는 Item 클래스를 새로 선언하는 것입니다. 이 클래스는 이름(name), 설명(description), 무게(weight) 등의 필드를 가지면 될 것입니다. Room 클래스에는 Item 필드를 추가하면 됩니다. 아래와 같이 코드를 작성하세요.

1. Item 클래스를 작성한다. Item 클래스는 세 개의 private 필드를 가지며 이 필드들을 읽는 데 사용되는 세 개의 public get 메소드를 갖는다(getName, getDescription, getWeight). 또 세 개의 파라미터를 갖는 생성자 하나를 둔다.
2. Room 클래스에 Item 타입의 private 필드와 public void setItem(Item item) 메소드를 추가한다. setItem 메소드는 Room에 아이템을 놓는데 사용한다.
3. Game 클래스의 createRooms( ) 메소드에서 방들을 만들 때 아래와 같이 아이템들을 추가한다. 
4. 플레이어가 어떤 방에 들어가면 방에 대한 설명, 출구 정보와 함께 그 방에 있는 아이템에 대한 설명이 나타나도록 필요한 부분을 수정한다. 단, 지금까지 공부한 설계의 기본 원칙에 맞게 설계해야 한다.

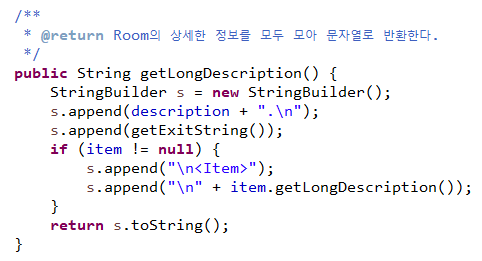
Room에 있는 Item에 대한 설명문을 어떻게 만들어 내야 할까요? 어느 클래스가 만들어 내야 할까요?

==> Item이 만들어 내야 합니다. (그러니까 Item 클래스에 getLongDescription 메소드가 추가로 있어야 합니다.)

Graphical user interface, text

Description automatically generated

또, Room의 getLongDescription 메소드도 수정되어야 합니다. Room에 대한 설명, 출구에 대한 설명에 추가로 아이템에 대한 설명이 포함되도록 해야 합니다. 단, 아이템이 있을 수도 있고 없을 수도 있으니 있을 때만 아이템에 대한 설명이 포함되도록 해야 합니다.



프로그램을 시험해 보세요. 제대로 잘 작동하는지 확인하세요. 예를 들면 아래와 같이 작동해야 합니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**강온실 w14p5\_zuul9 문제의 답으로 Game과 Item 클래스를 제출하세요.**

# **여러 개의 아이템**

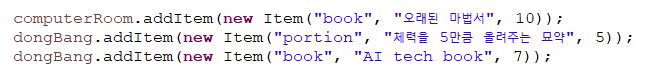
w14.zuul9 패키지를 "w14.zuul10"라는 이름의 패키지로 복사하여 저장하고 w14.zuul10에서 아래 작업을 진행하세요.

Room에 여러 개의 아이템이 있을 수 있게 프로그램을 수정하세요. Item을 별도의 클래스로 분리하여 설계한 가치는 이 때 잘 나타납니다. Item을 별도의 클래스로 선언했기 때문에 이 요구에 쉽게 대응할 수 있습니다. Room의 필드로 Item을 두는 대신 적절한 List of Items(가령, ArrayList<Item>)를 두면 됩니다. 그리고 여러 아이템들을 추가할 수 있도록 **set**Item을 **add**Item으로 변경합니다 (setItem은 아이템을 설정한다는 의미이므로 방에 하나의 아이템만을 둘 수 있다는 의미가 내포되어 있습니다. 반면, addItem은 이미 다른 Item이 있을 때도 Item을 추가할 수 있으므로 여러 개의 Item이 있을 수 있다는 의미가 포함되어 있습니다.) 이에 맞춰 Room 클래스들의 getLongDescription 메소드도 적절히 수정하면 됩니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그리고 아래 예와 같이 한 방에 여러 개의 아이템을 두도록 코드를 수정하세요.



게임이 제대로 잘 작동하는지 확인하세요. 예를 들면 아래 그림과 같습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**강온실 w14p6\_zuul10 문제의 답으로 Game과 Room 클래스 등 두 개의 클래스를 제출하세요. Import 문장들은 제외하고 제출하세요.**

# **Refactoring**

w14.zuul10 패키지를 "w14.zuul11"라는 이름의 패키지로 복사하여 저장하고 w14.zuul11에서 아래 작업을 진행하세요.

**Refactoring**

클래스와 메소드 구조를 다시 생각하고 다시 설계하는 작업 (설계를 뒤집어 엎는 작업).

사용법이나 작동의 변경 없이 코드의 구조만 바꾸는 작업

이제 우리는 아래와 같이 작동하도록 게임을 발전시키려고 합니다.

* 선수(player)가 현재 방에 있는 아이템을 집어들 수 있다.
* 선수가 여러 아이템을 갖고 다닐 수 있지만 들 수 있는 무게에 제한이 있다.
* 어떤 아이템은 집어 들 수 없다.
* 선수는 자기가 갖고 있는 아이템을 현재 방에 내려 놓을 수 있다.

이렇게 하기 위해서는 선수가 현재 갖고 있는 아이템이 무엇인지, 선수는 최대 얼마까지의 무게를 갖고 갈 수 있는지 등을 메모리에 저장해 기억해 두고 있어야 합니다. 이 메모리는 어디에 두어야 할까요?

Command, CommandWords, Parser 등은 당연히 아닙니다. 기능적으로 아무런 관련이 없습니다.

Room은 그 인스턴스가 여러 개이기 때문에 그 중 어느 한 Room에 이런 정보를 저장하는 건 이상합니다.

지금 선수는 한 사람뿐입니다. 선수가 최근에 있었던 방을 나타내는 recentRoom, 선수가 현재 있는 방을 나타내는 currentRoom 등의 메모리를 Game 클래스에 두었듯이, 이 정보(선수가 현재 갖고 있는 아이템이 무엇인지, 선수는 최대 얼마까지의 무게를 갖고 갈 수 있는지 등)도 Game 클래스에 두는 것이 좋겠습니다.

그런데 Game 클래스는 이미 너무 크고 너무 복잡합니다!!!

그래서 별도의 클래스를 새롭게 도입하는 편이 좋겠습니다.

선수! 바로 Player입니다. 이 클래스를 새로 두면 나중에 여러 명의 선수가 게임에 참여하게 할 수도 있을 것입니다. 선수 하나 하나와 관련된 정보는 각 선수 인스턴스가 스스로 기억하도록 하는 게 좋겠습니다.

Game은 Player 객체를 생성하고 전체 게임의 흐름을 관리하는 일만 하면 좋겠습니다. 선수와 관련된 정보는 모두 Player 클래스로 모아야 하겠지요. 현재 Game 클래스의 필드로 선언되어 있는 currentRoom, recentRoom은 모두 선수와 관련된 정보(선수가 현재 어디 있는지, 과거 어디에 있었는지)이므로 이들을 Player로 옮기면 좋겠습니다. 선수가 여러 명인 경우 각 선수마다 나름대로의 현재 위치, 최근 위치를 가지게 될 것입니다.

Refactoring은 꽤 많은 노력과 시간을 요구하지만 적절한 refactoring은 전체 프로젝트 일정을 줄여 줍니다. 좋지 않은 프로그램 구조를 끝까지 끌고 가면 뒤로 갈수록 문제가 많아집니다.

Refactoring은 두 단계로 진행합니다.

1. 기능 변경 없이 구조 개선만 한다. 구조개선 후에는 예전처럼 잘 작동하는지 철저히 테스트한다.
2. 새 구조에 추가로 요구되는 기능을 구현한다.

우선, 위 1단계에 해당하는 작업을 진행합니다.

기능 변경 없이 Game에서 Player를 분리하세요. 기능 개선을 위한 작업은 다음 단계에서 수행합니다. Multi-Player 기능은 앞으로도 상당 기간 동안 필요가 없을 것이라고 가정하세요. Multi-Player 기능까지 고려하면 문제가 지나치게 복잡해집니다.

<힌트>

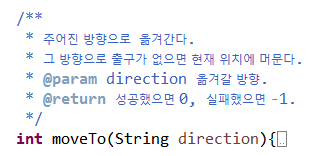
게임 프로그램이 어떻게 실행되는지 정리해 봅시다. Game의 play 메소드가 "키보드 입력을 읽어 명령을 해석하고, 명령에 따른 작업을 수행하는 일"을 반복합니다. 명령 중에 player와 관련된 것으로는 “go”와 “back”이 있습니다. 올바른 “go” 명령이 입력되면 player의 위치가 바뀌어야 합니다. Player의 위치를 바꾸는 일은 player이 스스로 하게 해야 합니다(책임분담원칙). 그러니까 Game클래스가 Player에게 “어느 방향으로 이동해라” 이런 명령을 내려야 합니다. 가령, player.moveTo("north") 와 같은 메소드 호출을 해야 합니다. 그러니까 Player 클래스가 아래와 같은 형식의 메소드를 지원해야 합니다.

moveTo(String direction)

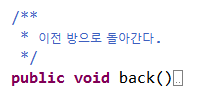
어느 방으로 갈지는 Game이 알고 있습니다. 사용자 명령을 해석한 결과를 갖고 있는 놈이 Game이니까요. 그래서 Game 클래스는 player.moveTo("north")와 같이 메소드를 호출할 수 있습니다.

moveTo 메소드의 반환값은 어떤 타입이어야 할까요?

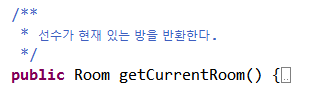
moveTo 메소드는 파라미터로 주어진 direction이 무엇이고 Player가 현재 어느 방에 있느냐에 따라 주어진 direction으로 이동할 수도 있고 그렇지 않을 수도 있습니다. 그러니까 moveTo가 실행한 후 그 결과가 성공이었는지 실패였는지를 메소드를 호출한 놈에게 알려주어야 합니다. 실패했으면 -1을, 성공했으면 0을 반환하는 것으로 약속하면 되겠습니다. 그러니까 moveTo 메소드의 형식은 아래와 같습니다.



“back” 명령이 입력되면 player가 최근에 있던 방으로 돌아가야 합니다. Player가 최근에 있던 방은 Player 스스로가 알고 있습니다. 그러니까 Player가 직전 방으로 돌아가도록 하려면, 인자 없이 player.back( )과 같은 형식으로 메소드를 호출하면 됩니다. 그러니까 Player 클래스는 아래와 같은 형식의, 파라미터 없는 메소드도 지원해야 합니다.



Game 클래스가 작업을 하다 보면 Player가 현재 어느 방에 있는지 알 필요가 생길 것입니다. 가령, 사용자가 “look” 명령을 입력한 경우, 명령을 해석하고 실행하는 Game 클래스는 player가 현재 어느 방에 있는지 알아야 그 방에게 정보를 요청하고 반환되는 정보를 출력할 수가 있습니다. 그런데 Game에 있던 currentRoom이라는 필드가 Player 클래스로 옮겨갔기 때문에 Game은 Player가 현재 어느 방에 있는지의 정보를 평소에는 갖고 있지 않습니다. 그래서 Game은 Player의 현재 위치를 알고 싶으면 Player에 물어봐야 합니다. 가령, player.getCurrentRoom( ) 이렇게 메소드를 호출해야 합니다. 이 메소드를 실행하면 무엇이 반환되도록 해야 할까요? 바로 Room입니다. hall이 반환되거나 computerRoom이 반환되거나, 아무튼 Room 타입 객체가 반환되어야 합니다. 그러니까 Player는 아래와 같은 메소드를 지원해야 합니다.



이상을 정리하여 Player 클래스를 작성하되 각 메소드 헤더와 메소드 주석만 작성하고(몸체 없이), javadoc을 돌려 api 문서를 만들어 냅니다. 그러면, 예를 들어, 아래와 같은 문서가 만들어집니다. (아래 링크를 클릭해서 열리는 문서에서 Player를 한 번 더 클릭하세요.)

[Player 클래스의 인터페이스](doc/index.html)

그 이후에 각 메소드 몸체에 들어갈 코드를 작성해 넣으면 됩니다.

구체적인 수정 사항을 정리하면 아래와 같습니다.

1. Game에 있던 두 개의 필드가 Player 클래스로 이동합니다. 그리고 아래와 같이 Player의 생성자를 작성합니다. 생성자를 아래 그림과 같은 형식으로 작성하면, Player를 생성할 때 시작 위치를 지정해 주어야 합니다. 가령 new Player(hall).

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. Game 클래스는 Player 필드를 가져야 하며 Game 클래스 생성자에서 Player 인스턴스를 하나 생성해 주어야 합니다. 물론 player 필드가 이 인스턴스를 가리키도록 설정해야 합니다.

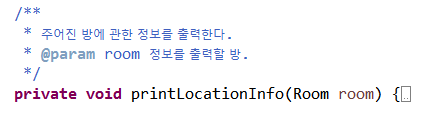
Game 클래스 생성자에서 Player를 생성할 때는 Player의 초기 위치를 지정해 주어야 합니다(가령 hall). 그런데 지금까지 방을 가리키는 **hall**, **lectureRoom**, **dongBang** 등 변수들이, 아래 그림에서 보듯, createRooms메소드의 지역변수로 선언되어 있었습니다.

Graphical user interface, text, application

Description automatically generated

위 그림처럼 지역변수로 선언되어 있으면 이 변수들을 createRooms 메소드 외부에서는 사용할 수가 없습니다. 그러니까, Game의 생성자 내에서 new Player(**hall**) 혹은 new Player(**lectureRoom**)과 같이 쓸 수 없습니다. 그래서 이 지역변수들을 Game 클래스의 필드로 바꿔주어야 합니다. 필드는 클래스 전체에서 통용됩니다. createRooms 메소드 내에 있는 **Room 객체를 생성하는 코드는 그대로 둔 채** Room 타입 변수 선언문만 메소드 밖으로 꺼내어 인스턴스 필드로 선언해 주세요. 특별한 이유가 있지 않은 한 private 필드로 선언합니다.

1. Game 클래스의 printLocationInfo( ) 메소드도 변경해야 합니다. 이 메소드는 선수가 있는 방의 정보를 출력해 줍니다. 이 메소드는 지금까지 파라미터를 갖지 않는 메소드였습니다. Game 클래스 내에 currentRoom이라는 필드가 있고 Game 클래스 내에 있는 모든 메소드는 이 필드에 접근할 수 있었습니다. 그러니까 printLocationInfo( ) 메소드는 currentRoom의 정보를 출력하면 됐습니다. 그러나, 이제 Game 클래스가 currentRoom이라는 필드를 갖지 않게 됐기 때문에, printLocationInfo( ) 메소드는 선수가 현재 어느 방에 있는지 스스로는 알지 못합니다. 그래서 printLocationInfo( ) 메소드를 호출할 때 현재 선수가 있는 방을 알려주어야 printLocationInfo( )메소드가 그 방에 대한 정보를 출력할 수 있습니다. 만약 선수가 현재 lectureRoom에 있다면 printLocationInfo(lectureRoom)과 같이 호출해야 합니다. 그러므로 이 메소드의 형식이 아래와 같이 바뀌어야 합니다.



Game 클래스 내의 다른 메소드에서 위 메소드를 호출할 때는 아래와 같이 해야 합니다. player에게 현재 어느 방에 있는지 물어보고 그 방에 대한 정보를 출력하도록 하는 것입니다.

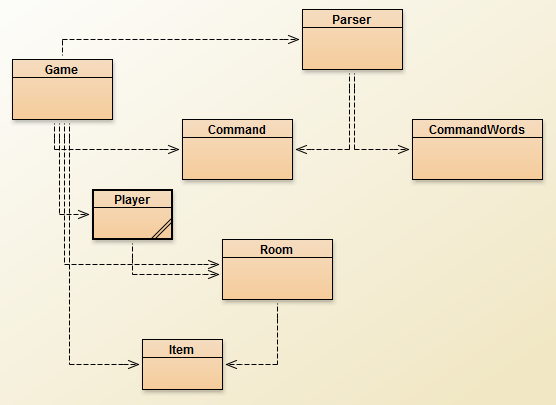
printLocationInfo(player.getCurrentRoom());

1. Game 클래스의 back과 goRoom 메소드는 모두 방을 이동하는 기능과 새 방의 정보를 출력하는 두 가지 기능을 수행했습니다. 그 중 방을 이동하는 기능은 Player의 back과 moveTo 메소드로 옮겨 갑니다. 이 메소드들은 선수에 관한 정보를 변경하는 기능이므로 Player 클래스에 두는 것이 좋습니다. 새 방의 정보를 출력하는 기능은 game 클래스의 back과 goRoom 메소드에 그대로 놔 둡니다. 입출력 등 사용자 인터페이스 기능은 Game 클래스에 집중시키기로 했기 때문입니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

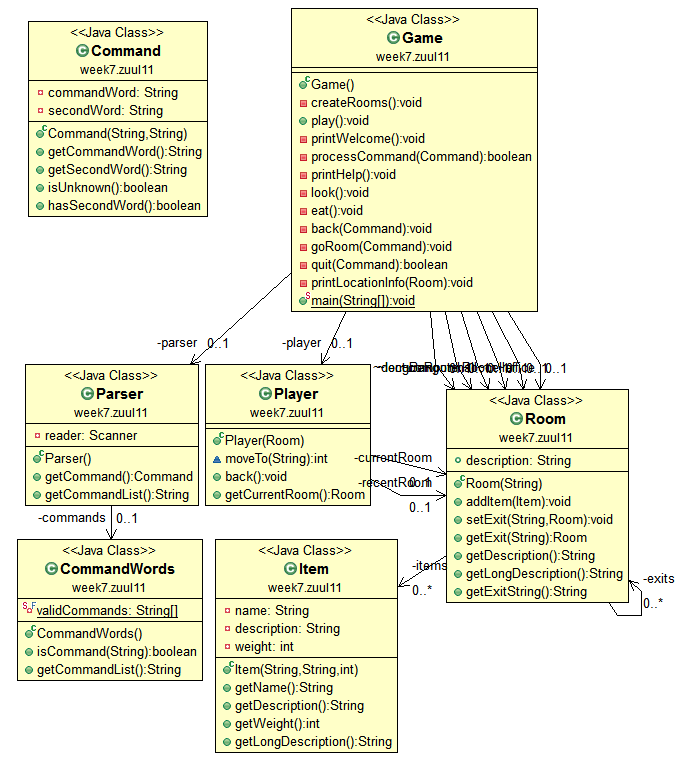
완성된 애플리케이션의 클래스 다이어그램은 아래와 같습니다.



위 다이어그램에서 화살표가 실선이 아니고 점선이라는 점에 주의하세요. **점선 화살표**는 한 클래스가 다른 클래스에 **의존(depend)**한다는 의미입니다. A 클래스의 코드에 B 클래스 이름이 나타나면 A가 B에 의존하는 것입니다. B가 없으면 A는 컴파일도 안 되고 실행되지도 않습니다. 위 그림을 보면 Game과 Parser가 Command의 의존하고 있습니다. (Game과 Parser 코드에 Command라는 클래스가 사용되고 있습니다.)

앞에서 다른 클래스 다이어그램을 볼 때는 화살표가 실선 화살표였습니다. **실선 화살표**는 **연계(associate)**되어 있다는 의미입니다. A 클래스가 B 클래스 객체를 가리키는 필드를 가지면 (그러니까 A가 B를 **항상** 알고 있으면) A는 B에 연계되어 (associate) 있다고 말합니다. A가 B에 의존한다고 해서 A와 B가 연계되어 있다고 말할 수는 없습니다. 반면, A가 B에 연계되어 있다면 A는 B에 의존합니다.

의존관계가 아닌 **연계관계**를 보여주는 클래스 다이어그램은 아래와 같습니다.



위 그림을 보면 Game과 Parser가 Command에 연계되어 있지 않습니다. (Game과 Parser객체는 Command객체를 가리키는 필드를 갖지 않습니다.) 그래서 평소에는 (스스로는) Command 객체에 접근할 수 없습니다.

코드를 모두 수정한 후, 예전처럼 잘 작동하는지 철저히 테스트하세요. go, back, look, help 등의 기능이 모두 잘 작동하는지 확인하세요.

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

**강온실 w14p7\_zuul11 문제의 답으로 Player 클래스를 제출하세요.**

**강온실 w14p8\_zuul11 문제의 답으로 Game 클래스를 제출하세요.**

# **아이템 이동**

w14.zuul11 패키지를 "w14.zuul12"라는 이름의 패키지로 복사하여 저장하고 w14.zuul12에서 아래 작업을 진행하세요.

더 좋은 방향으로 (기능 추가가 쉽도록) 구조를 변경했으니 (refactoring), 이제 새로운 기능을 추가합니다. 선수가 아이템을 집어 들거나 놓는 기능을 구현하려고 합니다. 선수는 여러 개의 아이템을 들고 다닐 수 있지만 자신이 들고 다닐 수 있는 최대 무게보다 많이 들고 다니지는 못합니다.

1. 아래와 같은 두 개의 명령어를 추가합니다. 아이템을 들거나 내려 놓을 때 사용할 명령어입니다.

take, drop

두 명령어 모두 인자(두 번째 단어)를 주어야 합니다. 인자는 아이템 이름(가령 book, portion 등)입니다.

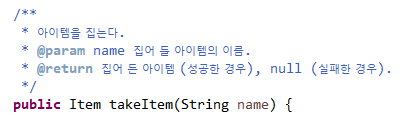
즉, take book, drop book 같은 형식으로 명령을 입력하게 됩니다.

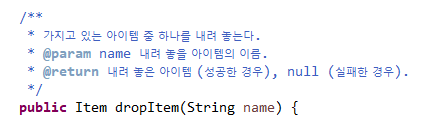
take와 drop 명령이 내려지면 그에 따라 아이템을 집거나 내려놓는 일을 한 후, player가 지니고 있는 아이템들에는 어떤 것들이 있는지를 화면에 출력해 줍니다. 출력 형식은 예를 들면 아래와 같습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

1. 선수가 들 수 있는 최대무게 (maxWeight), 현재 가지고 있는 아이템의 List (가령 ArrayList<Item>) 등을 저장할 메모리가 필요합니다. 이 메모리들은 당연히 Player 클래스의 필드로 넣어야 할 것이고 Player 생성자에서 이들과 관련된 적절한 작업을 해 주어야 합니다.   
   아이템을 들거나 내려놓는 작업도 Player이 해야 합니다. 그러니까 takeItem, dropItem 같은 메소드가 Player에 추가되어야 합니다. 이 메소드들은 String 타입 파라미터를 가져야 하겠지요. “어떤 이름의 Item을 집어라.”, “어떤 이름의 Item을 내려 놓아라.” 이런 식으로 사용할테니까요. 가령 “book”을 집어라, takeItem(“book”) 이렇게 사용될 것입니다.





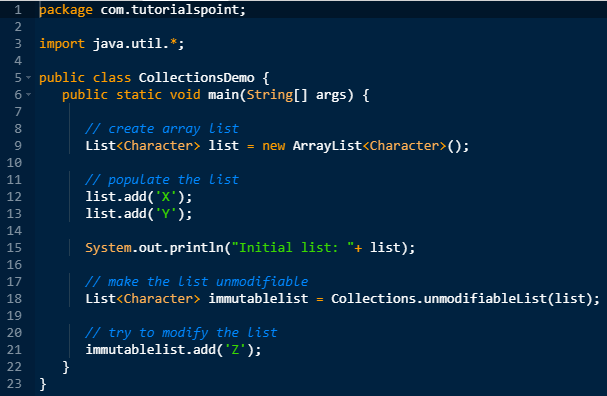
1. Player에게 어떤 아이템들을 가지고 있는지 물어보면 이를 알려주는 getItems 같은 메소드도 필요할 것입니다. 이 메소드는 Player이 가지고 있는 아이템의 리스트를 반환합니다. 그런데 리스트 자체를 반환하면 그것을 받은 자가 리스트를 임의로 변경할 수도 있으니 안전성 측면에서 좋지 않습니다. 그래서 리스트 자체를 반환하는 대신 그 리스트의 unmodifiableView를 반환합니다. 그렇게 하면 getItems를 호출한 자가 getItems 메소드의 반환값으로 받은 리스트를 함부로 변경하지 못합니다.

아래는 unmodifiableView에 대한 설명입니다.

자바 표준라이브러리의 **Collections 클래스는 static 메소드**인 unmodifiableList 메소드를 갖습니다.

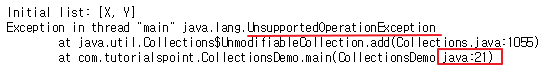


List<T> 타입 리스트인 myList가 있을 때 Collections.unmodifiableList(myList)를 호출하면 myList의 수정할 수 없는 view가 반환됩니다. View를 반환한다는 말은 myList의 복사본을 만들지 않는다는 의미입니다. 아래 예를 보세요.



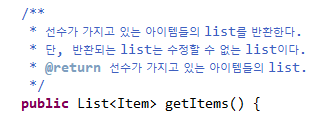
<https://www.tutorialspoint.com/java/util/collections_unmodifiablelist.htm>

위 프로그램을 실행하면 아래와 같이 예외가 발생합니다. UnmodifiableList에 원소를 추가하려고 시도했기 때문입니다.

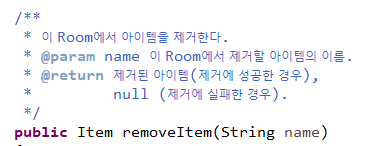


위 코드에서 list와 immutablelist는 하나의 리스트를 가리키는 참조입니다. 그러나 list를 통해서는 리스트의 내용을 변경할 수 있는 반면, immutablelist를 통해서는 그렇게 할 수 없습니다.

getItems 메소드 헤더는 아래와 같습니다.



1. 선수가 아이템을 들고 가면 방에서 그 아이템이 없어져야 합니다. 그러니까 Room클래스에 아이템을 제거하는 메소드가 추가되어야 합니다. 메소드 이름은 가령 removeItem 정도면 되겠습니다. 이 메소드는 String 타입 파라미터를 가져야 하겠지요. “어떤 이름의 Item을 없애라” 이런 식으로 사용할 것입니다.



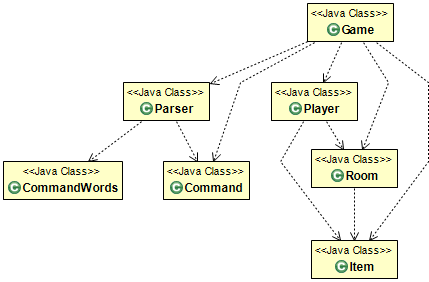
1. 어떤 아이템은 성격상 들고 갈 수 없습니다. 이런 아이템은 어떻게 구현하는 게 좋을까요? 무게를 아주 무겁게 설정할 수도 있고, 아이템에 canBePickedUp 같은 이름의 boolean 타입 필드를 두는 방법도 있겠습니다. 우리는 간단히 전자로 하기로 합시다. int 타입 값 중 가장 큰 값은 Integer.MAX\_VALUE입니다. Integer 클래스에 선언되어 있는 static constant(상수)입니다. 들고 갈 수 없는 아이템으로 설정하려면 그 무게를 Integer.MAX\_VALUE로 지정하면 됩니다.
2. items 라는 명령어를 추가합니다. 이 명령어를 입력하면 현재 선수가 가지고 있는 아이템을 모두 보여주고 그 무게의 합을 보여줍니다.
3. take와 drop 명령이 내려지면 그에 따라 아이템을 집거나 내려놓는 일을 한 후, player가 지니고 있는 아이템들에는 어떤 것들이 있는지를 화면에 출력해 줍니다. items 명령이 내려질 때도 player가 지니고 있는 아이템들에는 어떤 것들이 있는지를 화면에 출력해 주어야 합니다. 이 때의 출력 형식은 예를 들면 아래와 같습니다.

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

자동 생성된 설명

아이템들을 출력한 후 현재 가지고 있는 무게 총 합과 함께 선수가 들 수 있는 최대 무게가 함께 출력되도록 했습니다.

위 기능들을 구현하세요. Game, Player, CommandWords, Room 클래스가 변경되어야 할 것입니다. Item 클래스는 바뀌지 않습니다. 바뀌는 클래스들 중에는 Room 클래스를 제일 먼저 수정하는 게 좋겠습니다. Room 클래스는 Item 외의 다른 클래스에 의존하지 않기 때문에, 다른 클래스의 변경 내용을 고려하지 않고 독립적으로 수정할 수 있습니다. (다른 클래스들은 Room에 의존하기 때문에 Room이 바뀌면 그 바뀐 내용을 반영하여 수정해야 합니다.)



그 다음으로는 Player를 변경하는 편이 좋겠습니다. Player 클래스는 바뀌지 않는 Item과 이미 바꾼 Room에만 의존하기 때문입니다. Game과 CommandWords에는 의존하지 않습니다. Player를 먼저 고치고 그에 맞춰 Game을 수정하는 게 순서입니다. CommandWords는 다른 클래스와 관계없이 언제든 수정할 수 있습니다.

<힌트>

**Room 클래스:**

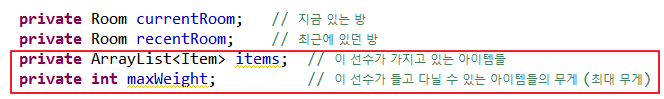
추가될 removeItem 메소드 구현 예

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

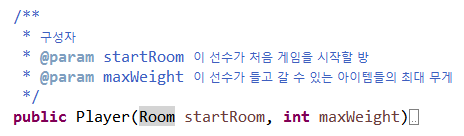
**Player 클래스**

필드 추가



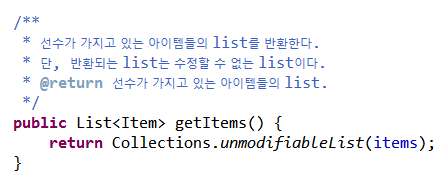
생성자 변경

추가된 두 개의 필드를 적절히 초기화해 준다.



메소드 추가

① getItems



1. takeItem

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위 코드에서 아이템을 집을 수 있는지 없는지를 판별해야 하는데 그 로직을 이곳에 직접 구현하면 코드가 너무 길어집니다. 그러니까 아이템을 집을 수 있는지 없는지를 판별해 주는 아래와 같은 메소드를 하나 작성하고 위 코드에서는 이 메소드를 이용하는 편이 좋습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위 메소드는 이 클래스 내부에서만 사용할 것이므로 private 메소드로 선언합니다.

위 메소드를 구현할 때 이 Player가 현재 가지고 있는 아이템들의 무게가 얼마인지 알아야 하는데 이 기능도 위 메소드에 직접 적어 넣으면 코드가 너무 길어집니다. 그래서 아래와 같은 메소드를 작성하고 위 코드에서는 이 메소드를 이용하는 편이 좋습니다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

1. dropItem

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

이상의 변화가 반영된 Player 클래스의 api는 아래와 같습니다.

[인터페이스](doc12\index.html)

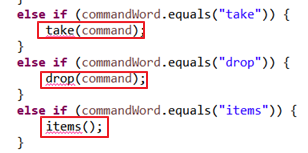
**CommandWords 클래스**

세 개의 명령어 추가 - 이 문제의 (가)와 (바) 참고

**Game 클래스**

Player 생성자 파라미터가 변경되었으므로 Game 생성자에서 Player 인스턴스를 생성할 때 새 형식에 맞도록 수정합니다. Player가 들 수 있는 최대 무게는 임의로 정합니다 (가령 20).

세 개의 명령어를 처리하는 코드를 추가합니다. take와 drop은 인자(second word)가 필요한 명령입니다. 그래서 아래 코드에서 take 메소드와 drop 메소드를 호출할 때는 Command 객체를 인자로 넘겨줍니다. (Command 객체 속에 second word가 들어 있으므로). items는 인자가 없는 명령입니다. 그래서 items 메소드를 호출할 때는 Command 객체를 인자로 제공해 주지 않아도 됩니다.



그리고 명령에 따른 작업을 실제로 수행하는 take, drop, items 메소드를 작성합니다. 이들은 모두 private 메소드입니다.

1. take

텍스트, 스크린샷, 폰트, 문서이(가) 표시된 사진

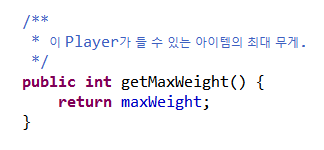
자동 생성된 설명

위 코드에서 리스트에 있는 모든 아이템을 출력하는 작업을 직접 적으면 위 코드가 너무 길어집니다. 그래서 아래와 같은 메소드를 작성하고 위 코드에서는 이 메소드를 호출하도록 합니다. 출력의 형식은 이 문제 사)에 그 예시가 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

위 코드에서 선수가 들 수 있는 최대 무게가 얼마인지 Game 클래스는 알지 못하므로 Player에게 물어봐야 합니다. 그러니까 Player 클래스가 이 목적의 메소드를 추가로 지원해야 합니다. 아래와 같은 메소드를 Player 클래스에 추가하고 위 코드에서는 선수가 들 수 있는 최대 무게가 얼마인지 알아내기 위해 이 메소드를 호출합니다.



1. drop

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

1. items

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

컴파일 에러가 없도록 하고 새 기능뿐 아니라 기존 기능까지도 철저히 테스트합니다. 아래는 실행 예입니다.

텍스트, 스크린샷, 문서이(가) 표시된 사진

자동 생성된 설명

**강온실 w14p9\_zuul12 문제의 답으로 Room 클래스를 제출하세요. Import 문장은 포함시키지 마세요.**

**강온실 w14p10\_zuul12 문제의 답으로 Player 클래스를 제출하세요. Import 문장은 포함시키지 마세요.**

**강온실 w14p11\_zuul12 문제의 답으로 Game 클래스를 제출하세요. Import 문장은 포함시키지 마세요.**

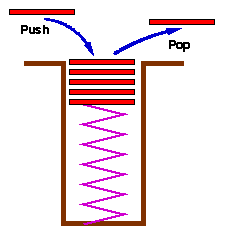
# **여러 단계 back**

w14.zuul12 패키지를 "w14.zuul13"라는 이름의 패키지로 복사하여 저장하고 w14.zuul13에서 아래 작업을 진행하세요.

이번에는 “back” 기능을 확장합니다. 여러 번 “back”을 호출함으로써 여러 단계 전으로 갈 수 있게 합니다.

이 기능을 가장 편리하게 구현하는 방법은 [java.util.Stack](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Stack.html)을 이용하는 것입니다. 방을 거쳐갈 때마다 방을 Stack에 넣어줍니다. 뒤로 가고 싶을 때는 Stack에서 최근에 넣은 방을 꺼내어 그 곳으로 이동하면 됩니다. 한 번 더 뒤로 가로 싶어도 마찬가지입니다.

Stack은 ArrayList나 HashSet같이 여러 개의 객체들을 저장하는 그릇입니다. Stack의 특징은 그 사용법에 있습니다. 아래 그림에서 보듯 Stack에 객체를 넣을 때는 push(객체) 메소드를, 꺼낼 때는 pop() 메소드를 사용합니다. 어떤 객체를 push해라, 맨 위에 있는 객체를 pop해라, 이런 의미의 명령입니다. 여러 개를 push한 후 여러 번 pop을 하면, 제일 나중에 들어간 객체가 제일 먼저 나옵니다. (Last-In First-Out).

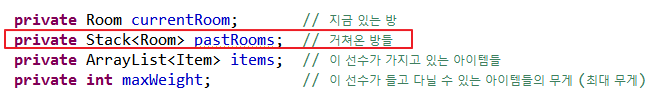


Player의 위치는 Player 스스로가 관리하므로 이 기능을 구현하기 위해서는 Player 클래스가 주로 변경되어야 할 것입니다. 현재 Player에는 최근에 있던 방을 기억하기 위해 recentRoom 필드가 있는데 이것을 Stack으로 바꾸어 과거에 거쳐온 모든 방들을 기억할 수 있게 하고 관련되는 메소드들을 수정하면 됩니다.

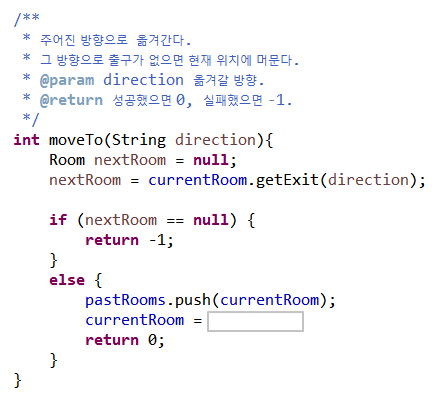
Player가 최근 있었던 방을 저장하는 데 사용했던 recentRoom 대신 Stack<Room> 을 필드로 둡니다. Stack은 generic class이므로 타입 파라미터를 갖습니다. Stack<Room>은 Room들을 저장하기 위한 Stack입니다. 방을 옮겨 갈 때마다 지금까지 있었던 방을 recentRoom에 저장해 두었듯이, 방을 옮겨갈 때마다 최근에 있었던 방을 Stack에 push해 둡니다. 무한히 많은 방을 push할 수 있습니다. back할 때는 Stack을 pop하면 최근에 있던 방이 반환되니까 그리로 들어가면 됩니다. pop 할 때마다 그 이전 방들이 차례로 반환됩니다. (방을 push하거나 방을 pop한다는 것은 방을 가리키는 참조를 push하거나 pop한다는 말입니다!)

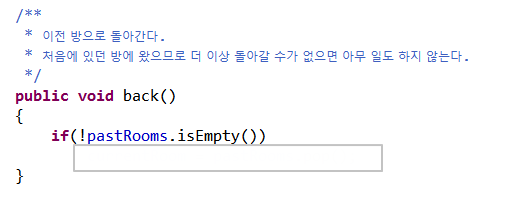
back을 거듭하여 맨 처음 위치로 돌아간 상태에서 back 명령을 내리면 그 자리에 그대로 머물러 있도록 하세요. 별다른 안내 메세지는 나타나지 않아도 좋습니다. back을 거듭하여 맨 처음 위치로 돌아간 상태에서는 stack이 비어 있을 것입니다. 그러니까 평소에 pop을 할 때마다 stack이 비어 있는지 확인하고 비어 있지 않은 경우에만 pop을 해야 합니다. 비어 있다면 아무 일도 하지 않으면 됩니다.

여러 단계 back 기능을 구현하기 위해서는 Player 클래스만 변경하면 됩니다! 다른 클래스는 바꾸지 않아도 됩니다. 우리 프로그램 구조가 상당히 좋다는 의미입니다.









프로그램 실행 예는 아래와 같습니다.

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 문서, 폰트이(가) 표시된 사진

자동 생성된 설명

**강온실 w14p12\_zuul13 문제의 답으로 Game, Item, Player, Room, Command, CommandWords, Parser 클래스 등 애플리케이션 전체를 제출하세요. Game 클래스만 public으로 선언하고 나머지 클래스들은 public으로 선언하지 마세요. 패키지 문장은 없어야 합니다. 모든 import 문장을 포함하여 제출하세요. import 문장은 프로그램의 맨 위에 있어야 합니다.**

**import java.util.Scanner;**

**import java.util.List;**

**import java.util.ArrayList;**

**import java.util.Collections;**

**import java.util.Iterator;**

**import java.util.Stack;**

**import java.util.HashMap;**

**import java.util.Map;**

**import java.util.Set;**

우리의 프로젝트는 여기까지입니다. 우리는 주로 클래스 설계를 어떻게 하는 것이 좋은지 예를 보기 위해 이 프로젝트를 수행했습니다. 이 프로젝트가 더 재미 있는 게임으로 진화하려면 프로그래밍 기술 외에 스토리와 상상력이 추가되어야 합니다. 지금까지 우리가 한 프로젝트에 상상력이 더해지면 제법 재미 있는 게임으로 진화할 수 있을 것입니다. 여러분들의 과감한 상상력이 필요한 시점입니다. 학생 개개인의 몫입니다.

상황 설정 예:

* 인체 내에서 바이러스 공격에 대응하기 위해 혈관을 여행하는 백혈구, 놀이공원, 굴 속 애벌레, 몬스터가 가득한 지하 동굴

기능 예:

* 주어진 시간 내에 임무 완수하기 (시간을 명령어 입력 횟수로 대신할 수 있다.)
* 한쪽으로만 갈 수 있고 반대 방향으로는 갈 수 없는 출구
* 열쇠를 찾아야만 들어갈 수 있는 방
* 공간이동
* 캐릭터 추가 (캐릭터에게 특정 아이템을 보여주면 캐릭터가 힌트를 말해주는 등 여러 가지 캐릭터들)
* 장소를 옮겨 다니는 캐릭터

끝.