Left 4 Grade 2 Episode 2 – 100명 Edition Document #4

~ developer's guide ~

목적

이 문서는 Left 4 Grade 2 Episode 2 – 100명 Edition(이하 L4G) 프로젝트를 수행하는 학생 여러분이 플레이어 코드를 작성할 때 **꼭 알아 두어야 하는** 내용들을 설명하고 있습니다. 또한, 여러분이 실제로 import하여 사용할 project 코드에 대한 간략한 소개와 함께 Player class를 작성할 때 사용하게 될 다른 클래스들을 설명합니다.

시작하기 전에

업데이트된 project를 import하기 전에 아래 내용들을 먼저 읽어주세요:

- 이 project에는 무려 20여 개의 코드 파일이 들어 있습니다(심지어 아직 여러분의 project에 추가해 두지 않은 파일들도 남아 있습니다). 하지만 여러분이 직접 작성하게 될 부분은 **Player_YOURNAMEHERE.java** 뿐이며 **Program.java에 있는 게임 설정 부분**을 제외한 다른 코드는 L4G 내부에서 사용하는 내용들이므로 신경 쓰지 않아도 좋습니다. 이 프로젝트는 **여러분이 아직 Java 및 프로그래밍 기본을 익히지 않은 상태에서 참여하는 것을 가정**하고 있습니다. 그러므로, 여러분이 보고 사용할 수 있는 모든 요소들은 주석 설명을 읽어 가며 쉽게 이해할 수 있도록 마련되어 있습니다.

- L4G의 정규 게임 진행 결과는 수강생 여러분의 실제 학점에 반영됩니다. 따라서 객관성을 도모하기 위해 같은 조건에서 진행한 게임은 항상 같은 결과를 내도록 구성되어야 합니다. 다시 말하면, **같은 플레이어 구성 아래 같은 게임 번호를 사용하면 항상 같은 결과를 내도록** 플레이어 코드가 **시간 / 날씨 / 컴퓨터 사양 등과 같은 외부 요인에 의해 의사 결정을 수행해서는 안 된다**는 것을 의미합니다. 결론적으로, **여러분의 플레이어는 Random class를 사용할 수 없습니다!** 적어도 해당 플레이어를 만든 여러분 자신은, 게임이 진행될 때 **자신의 플레이어가 어떤 상황에서 어떻게 행동할 것인지를 유추할 수 있어야 합니다.**
  다만, 위에서 언급한 조건이 충족된다면(외부 요인 없음, Random class 안 씀, 본인이 유추 가능), 여러분의 플레이어가 받을 수 있는 정보를 토대로 임의의 수를 만들어 의사 결정에 사용할 수는 있습니다. Project에 동봉되어 있는 Bot 플레이어 클래스들은 모두 이러한 방식을 사용하여 같은 클래스의 인스턴스라도 서로 다른 의사 결정을 수행하도록 구성되어 있습니다.

- 학점에 반영될 정규 게임은 총 1,000,000번의 게임을 수행하게 되며 따라서 원활하게 정규 게임을 진행하기 위해서는 성능 관련 이슈를 중요하게 고려해야 합니다. 자세한 설명은 생략하고, **여러분의 코드에는 절대 'static'이 있어서는 안 됩니다!**

- L4G 프로젝트에서 코드 작성은 여러분의 생각을 한데 모으기 위한 과정에 불과하며, 프로젝트 자체에서 코드 작성을 최우선 목표로 삼고 있는 것은 아닙니다. 그러므로 코드를 작성하다 **난관에 봉착하면 언제나 조교와 접촉을 시도하여 도움을 받을 수 있습니다**. 단, 도움을 받기 위해서는 여러분의 생각을 표현한 글, 그림 등이 적힌 노트를 꼭 지참하여야 합니다.

위의 내용을 충분히 읽고 이해했다면 project를 import하고 코드 작성을 시작해보도록 합시다!

프로젝트 외관

L4G2EP2-100ME Java project에 들어 있는 각 패키지와 일부 java 파일의 내용을 간략하게 설명합니다.

패키지들

L4G를 구성하는 요소들은 총 다섯 가지 패키지에 나뉘어 담겨 있습니다. l4g package에는 강의실 자체, 그리고 게임 내부에서 사용하는 주요 클래스들이 정의되어 있으며 여러분의 플레이어는 여기 있는 클래스들을 전혀 고려하지 않아도 됩니다. l4g.common package에는 '좌표'와 같은 기초 데이터 형식, 상수 집합, 게임 번호 목록과 같이 자유롭게 사용 가능한 클래스들이 들어 있으며 여러분은 l4g.customplayers package에 있는 Player_YOURNAMEHERE.java 파일의 이름을 바꾸는 것으로 코드 작성을 시작한 다음 (default package)의 Program class 안에 있는 main()에서 설정 및 플레이어 등록을 함으로써 이를 테스트해 볼 수 있습니다.

l4g.bots package에는 여러분이 작성한 플레이어를 테스트하기 위해 게임에 같이 참여할 Bot 플레이어 클래스들이 들어 있습니다. 이들은 player's guide 문서에 적혀 있던 여섯 가지 역할들 중 그나마 쉬운 세 가지를 나름 중점적으로 수행하도록 구성되어 있으며 여러분은 플레이어 코드를 작성할 때 이 패키지에 있는 내용을 참고하며 자신만의 플레이어를 만들어 나아가면 되겠습니다. **단, Bot_HornDone.java에 정의된 무법자 Bot 플레이어는 강의실의 엔트로피를 증가시키기 위한 테스트 전용 Bot 플레이어입니다.** 무법자 Bot 플레이어는 정규 게임에서 사용되지 않으며, **여러분은 무법자 Bot 플레이어가 사용하는 코드를 가져가 사용할 수 없습니다**.

Classroom.java (l4g package)

L4G의 '강의실'을 나타내는 클래스입니다. 이 클래스는 L4G 참조 문서에 적힌 모든 규칙들을 담고 있으며, 게임 한 판을 진행하기 위해 사용됩니다. 따라서 코드가 몹시 길고 어려우니 가급적 읽지 않는 것을 권장합니다.

Constants.java (l4g.common package)

게임 내에서 사용하는 여러 상수들이 들어 있습니다. 정규 게임에서는 미리 정해진 상수들을 사용하여 게임을 진행하지만 심심할 때 재미로 여기 있는 값을 바꾼 다음 게임을 진행해 볼 수도 있겠습니다(4x4칸 강의실에서 200명의 플레이어가 게임을 진행하도록 설정하는 것도 가능합니다).

여러분이 for문 등을 작성할 때 '강의실의 가로 길이'와 같은 값을 사용해야 한다면 직접 숫자 8을 쓰기보다는 여기 지정된 `Constants.Classroom_Width` 상수 필드를 사용하는 것이 오류를 줄일 수 있는 좋은 방법입니다.

PlayerInfo.java, CellInfo.java, Action.java, Reaction.java, TurnInfo.java (모두 l4g.data package)

여러분의 플레이어가 게임을 진행하면서 매 턴 받게 될 정보를 나타내는 클래스들이 들어 있습니다. 이 클래스들은 모두 '읽기 전용 필드'만 가지고 있으며, 여러분의 의사 결정 메서드가 호출되기 직전에 강의실이 이들을 적절히 생성하여 여러분의 플레이어 인스턴스에 있는 적절한 필드에 담아 줍니다.

Player.java (l4g package)

Player class는 모든 플레이어들이 공통적으로 갖게 될 요소들을 정의하며(Java에서는 이를 superclass라 부릅니다) 여러분은 실질적으로 Player class의 의사 결정 부분을 직접 구현한, 여러분의 이름이 담긴 새로운 class를 만드는 것으로 L4G 게임에 참여하게 됩니다. 플레이어는 매 턴마다 자신이 구현한 여러 의사 결정 메서드들 중 하나 이상을 호출받으며 이동할 방향, 또는 배치할 좌표를 반환함으로써 게임에 참여하게 됩니다.

Player_YOURNAMEHERE.java (l4g.customplayers package)

미리 만들어져 있는, 여러분의 플레이어 클래스를 구현하기 위한 파일입니다. 여러분은 Alt + Shift + R 또는 F2를 통해 이 클래스(이 파일)의 이름을 바꿈으로써 자신만의 새로운 플레이어를 만들기 시작하게 됩니다!

여러분의 플레이어 클래스에 기본으로 들어 있는 필드들

Player class에는 모든 플레이어들이 각자 보유하게 될 필드들이 다수 정의되어 있으며 Player.java에는 각 필드에 대한 자세한 설명이 주석으로 기록되어 있습니다. 여기서는 Player class에 정의되어 있는 필드들에 대해 요약하면서 여러분이 사용하게 될 데이터 클래스들의 구성을 설명합니다.

- int ID
  플레이어마다 고유한 일련 번호입니다. 정규 게임에서 일련 번호는 '제출 시간'순서대로 매겨집니다. 여러분은 이 값을 통해 '저기 있는 시체가 누구인지', '내가 방금 누구를 죽였는지' 등을 식별할 수 있습니다. 사실, 최고의 플레이어나 한 명만 패는 플레이어가 되고 싶지 않은 이상 남의 ID는 큰 의미를 갖지 않습니다.

- String name
  플레이어의 '이름'입니다. 이 필드는 생성자의 super(); 부분에서 초기화하며 각종 정보를 '사람에게 보여줄 때'는 ID 대신 이 이름을 사용하여 출력해 줍니다. 여러분의 플레이어는 다른 플레이어의 ID는 볼 수 있지만 이름은 볼 수 없습니다. 이름 비교를 통해 팀 워크를 하려는 친구들은, 얼른 포기하세요.

- boolean trigger_acceptDirectInfection
  '직접 감염'을 수락하려는 경우 이 필드를 true로 설정해 두면 됩니다.
  이 필드의 사용은 보통 생성자에서 한 번 초기화해 두는 것으로 충분합니다.

- long gameNumber
  현재 진행중인 게임의 '게임 번호'입니다. '엄청 큰 숫자 하나'가 필요할 때 이용할 수 있습니다. Bot 플레이어들은 이 값과 자신의 ID 값을 의사 결정에 활용합니다.

- TurnInfo turnInfo
  이번 턴에 대한 정보가 들어 있습니다. TurnInfo class는 다음 정보들을 담고 있습니다:

  - int turnNumber – 이번 턴이 몇 턴인지
  - boolean isDirectInfectionChoosingTurn – 이번 턴이 '직접 감염 여부'를 선택하는 턴인지

- PlayerInfo myInfo
  플레이어의 현재 상태에 대한 정보가 들어 있습니다. PlayerInfo class는 다음 정보들을 담고 있습니다:

  - int ID - 플레이어마다 고유한 번호
  - StateCode state - 플레이어의 현재 상태
  - int HP - 플레이어가 시체 또는 감염체인 경우 현재 체력
  - int transition_cooldown - 다음 상태로 전이하기까지 남은 턴 수(시체, 감염체, 영혼일 때 적용)
  - Point_Immutable position - 플레이어의 현재 위치

- Score myScore
  여러분의 플레이어가 현재 기록중인 각 점수 값의 사본이 들어 있습니다. 점수 값은 말 그대로 '원점수'를 의미하며, 이는 게임이 끝나고 산출되는 '학점'과는 다릅니다. 따라서 이 필드의 내용은 '목표 점수에 도달했나'를 판단하기 위한 자료로서 사용하는 것이 가장 유용합니다.

- CellInfo[][] cells
  현재 플레이어의 시야 범위를 감안한 각 칸별 정보들이 들어 있습니다. CellInfo class는 중요 데이터를 숨겨 놓고 있으며 대신 여러분은 아래에 있는 public 필드 / 메서드들을 보고 사용할 수 있습니다:

  - static CellInfo Blank – '빈 칸'을 의미하는 static instance
  - Count_계열 메서드들 – 현재 이 칸에 있는 요소(플레이어 / 행동 / 사건)의 수를 반환합니다.
  - CountIf_계열 – 주어진 판정 메서드를 통해 해당 조건에 맞는 요소 수를 반환합니다.
  - ForEach_계열 – 주어진 작업 메서드를 모든 요소에 대해 호출하여 수행합니다.
  - Select_계열 – 주어진 판정 메서드를 통해 해당 조건에 맞는 요소 목록을 만들어 반환합니다.

  이러한 메서드들의 사용법은 이 문서 후반에서 자세히 설명하고 있습니다. 위에서 언급한 '요소'란 PlayerInfo class, Action class, Reaction class의 인스턴스를 말하며 이들은 각각 '해당 칸에 현재 위치한 플레이어', '해당 칸에서 방금 전에 수행된 행동(도착 위치 기준)', '해당 칸에서 방금 전에 발생한 사건(발견 사건 제외, 행위 주체의 위치 기준)' 정보를 담고 있습니다.

  이들 중 Action class는 다음 정보들을 담고 있습니다:

  - int actorID – 이 행동을 수행한 플레이어의 ID입니다.
  - Action.TypeCode type – 이 행동이 '이동'인지 '배치'인지를 나타냅니다.
  - Point_Immutable location_from – 이 행동이 수행된 시작 위치 좌표입니다.
  - Point_Immutable location_to – 이 행동이 수행된 도착 위치 좌표입니다.

  그리고, Reaction class는 다음 정보들을 담고 있습니다:

  - int subjectID – 이 사건을 일으킨 플레이어(행위의 주체)의 ID입니다.
  - Reaction.TypeCode type – 이 사건의 형식을 나타냅니다.
  - int objectID – 이 사건을 당한 플레이어(행위의 대상)의 ID입니다.
  - Point_Immutable location – 이 사건이 발생한 위치(subject의 위치)입니다.

좌표 다루기 – Point class와 Point_Immutable class

Java에서 Immutable은 '내부 값을 바꿀 수 없는'을 의미하는 형용사입니다. 강의실 내의 각 칸은 모두 고유하고 영속적인 Point_Immutable 좌표를 가지고 있습니다. 이들을 통해 '특정 칸의 왼쪽 칸을 가리키는 좌표'와 같은 새로운 좌표값을 임시로 만들 때는 Point_Immutable이 아닌 Point class의 인스턴스를 사용하게 됩니다. Point 좌표는 여러분이 자유롭게 생성 / 편집할 수 있으며 Soul_Spawn()의 반환 형식으로도 활용됩니다.

요약하면, 강의실이 여러분에게 나누어 주는 좌표들은 모두 Immutable입니다. 그리고 여러분이 이들의 옆 칸, 이들의 근처 칸 등을 계산한 결과는 Immutable이 아닌 좌표, 즉 Point class의 인스턴스로 표현됩니다.

여러분이 사용할 수 있는 좌표 연산을 예로 들면 다음과 같습니다(여기서 사용되는 각 이름의 의미는 해당 코드를 친 다음 마우스 포인터를 갖다 대면 자세히 나옵니다):

- 내가 현재 있는 칸의 **왼쪽** 칸 좌표 만들기

```
//Point_Immutable class 또는 Point class의 인스턴스에 점을 찍어서 다양한 좌표 연산을 할 수 있음
Point pos_left = myInfo.position.GetAdjacentPoint(DirectionCode.Left);
```

- 특정 좌표가 내 위치에서 **2**칸 떨어져 있는지 확인하기

```
int row = 3;        // 내가 원하는 위치의 세로 좌표
int column = 4;     // 내가 원하는 위치의 가로 좌표
boolean result = myInfo.position.GetDistance(row, column) == 2;
```

- 내 위치에서 **위로 한 칸, 왼쪽으로 한 칸** 떨어진 칸을 가리키는 좌표 만들기

```
int offset_row = -1;// 세로 좌표는 위에서 아래로 +
int offset_column = -1;     // 가로 좌표는 왼쪽에서 오른쪽으로 +
Point dest = myInfo.position.Offset(offset_row, offset_column);
```

위의 예시들처럼, 여러분은 보통 '내 위치'를 기준으로 좌표 연산을 수행할 것입니다. 만약 자신이 직접 만든 좌표를 기준으로 연산을 수행하고 싶은 경우에도, 그냥 해당 좌표 변수에 점을 찍고 동일한 메서드를 호출해 쓰면 되겠습니다(Point_Immutable class나 Point class나 사용 방법은 동일합니다).

'칸 정보' 활용하기 – CellInfo class

여러분에게 제공되는 대부분의 정보들은 특정 '칸'에 귀속되어 있습니다. 예를 들어, 내 왼쪽 칸에 있는 플레이어에 대한 정보는 cells 2차원 배열에서 내 왼쪽 칸 좌표에 해당하는 곳에 들어 있습니다. 따라서 여러분은 먼저 적절한 좌표값을 만들어 낸 다음 cells 2차원 배열을 조심스레 탐색하여 적절한 정보를 읽어 오게 됩니다.

그래서, 조심해야 합니다. cells[-1][-1]처럼 배열의 범위를 벗어나는(강의실 밖을 나타내는) 좌표를 사용하면 그 즉시 붙잡혀 20턴 동안 영혼 상태가 되는 패널티를 받게 됩니다. cells 2차원 배열을 쓸 때는 좌표의 유효성을 항상 보장해 주어야 합니다.

대충 경각심을 가지게 되었다면, 이제 cells 2차원 배열을 사용하는 방법을 확인해 봅시다:

- 나와 같은 칸에 있는 총 **플레이어 수** 확인하기

```
int count = cells[myInfo.position.row][myInfo.position.column].Count_Players();
```

  - Count_계열 메서드들은 무식하게 전체 숫자를 반환해 줍니다.

- 나와 같은 칸에 있는 **감염체** 수 확인하기

```
int count = cells[myInfo.position.row][myInfo.position.column].CountIf_Players(
                    player -> player.state == StateCode.Infected );
```

  - CountIf_계열 메서드들은 argument로 메서드 하나를 받습니다. 위에서는 '이 플레이어의 상태가 감염체인지 여부를 true/false로 반환하는' 람다 식을 집어 넣고 있습니다(람다 식에 대해서는 이 문서의 후반부에서 다시 설명하고 있습니다). 여러분이 이렇게 조건 검사를 위한 메서드를 넣어 주면 CountIf_계열 메서드들은 해당 메서드로 각 요소를 검사하여 true가 몇 번 나왔는지를 집계해 반환합니다.

  - 위의 코드에서 Infected 자리에 Survivor를 넣으면 생존자 수를 집계할 수 있습니다. 사실 이 메서드는 여러분이 가장 빈번하게 사용할 메서드라 해도 과언이 아닐 것입니다. 따라서 후반부 설명을 봐도 이해가 잘 안 되면 반드시 조교에게 문의해 주세요!

- 각 칸별 시체 수를 별도로 만든 int 형식 2차원 배열에 저장하기

```
/* 주의: 아래의 배열은 임시 변수가 아닌 field로 선언하여 쓰는 것이 좋습니다. */
int[][] counts = new int[Constants.Classroom_Height][Constants.Classroom_Width];

for ( int row = 0; row < Constants.Classroom_Height; ++row )
    for ( int column = 0; column < Constants.Classroom_Width; ++column )
            counts[row][column] = cells[row][column].CountIf_Players(
                                        player -> player.state == StateCode.Corpse );
```

- 여러분은 종종, 칸 하나에 대한 정보가 아닌 전체 강의실에 대한 간략한 통계 데이터를 필요로 하게 될 것입니다. 이를 위해 여러분은 능숙한 솜씨로 자신만의 2차원 배열 field을 선언 / 초기화하고, 이중 for문을 써서 전체 칸에 대해 CountIf_계열 메서드를 적절히 호출해 가며 배열의 내용을 기입해 주면 되겠습니다.

- 현재 시야 범위 내에서 '가장 HP가 많은 감염체가 서 있는 칸의 좌표' 확인하기

```
/* 주의: 아래의 두 변수는 임시 변수가 아닌 field로 선언하여 써야만 합니다. */
Pos_Immutable pos_max = null;
int HP_max = -1;

for ( int row = 0; row < Constants.Classroom_Height; ++row )
    for ( int column = 0; column < Constants.Classroom_Width; ++column )
            cells[row][column].ForEach_Players(player ->
            {
                    if ( HP_max < player.HP && player.state == StateCode.Infected )
                    {
                            pos_max = player.position;
                            HP_max = player.HP;
                    }
            });

//원하는 좌표는 pos_max에 있음!
```

- ForEach_계열 메서드들은 argument로 '요소 하나를 받아 작업을 수행하며 반환은 안 하는' 메서드를 받습니다. 위에서는 각 감염체의 HP를 현재까지 찾은 HP 최대값과 비교함으로써 가장 HP가 많은 감염체가 서 있는 칸의 좌표를 pos_max에 담아 내고 있습니다.

이외에도 '조건에 맞는 요소 목록'을 만들어 반환하는 Select_계열 메서드가 존재하지만, 여러분이 몹시 복잡한 플레이어를 추구하지 않는 이상 해당 메서드는 그리 쓸 일이 없을 것입니다. Select_계열 메서드는 CountIf_계열 메서드처럼 조건 검사를 위한 메서드를 argument로 받고, 해당 조건에 맞는 요소만 모아 둔 새로운 ArrayList<> 인스턴스를 반환해 줍니다. 여러분은 반환된 인스턴스를 자신의 field에 담은 다음 적절한 foreach문을 통해 각 요소의 내용을 더 정밀하게 관찰할 수 있습니다.

cells 2차원 배열에서 '모르는 칸'은 null이 아닌 '빈 칸'으로 채워집니다. 빈 칸에는 말 그대로 아무도 없으며, 따라서 Count_계열 메서드를 쓴 결과 또한 '오류'가 아닌 정상적인 '0'이 나오게 됩니다. 쉽게 말하면, 대부분의 경우 여러분이 자신의 시야 범위를 직접 고려할 필요는 없습니다.

'람다 식'에 대해

람다 식(lambda expression)은 쉽게 말하면 '임시 메서드'입니다. 원래 메서드는 클래스 안에 정의해야 하지만, 람다 식은, 마치 여러분이 메서드 안에서 field 가 아닌 임시 변수를 선언해 사용하듯, 자신이 원하는 위치에서 직접 정의해 쓸 수 있습니다. 사실 람다 식은 우리가 지금 이해하기에는 조금 복잡합니다. 그러니 L4G 에서는 그냥 이 정도로만 이해하고 사용하면 되겠습니다.

람다 식의 기본 문법은 다음과 같습니다:

```
인수_선언 -> 몸통
```

람다 식은 기본적으로 '메서드'입니다. 람다 식의 왼쪽 부분은 메서드의 argument 선언 부분에 해당하며, 오른쪽 부분은 메서드의 몸통에 해당합니다. 아래의 예시를 보면 이해가 좀 더 쉬울 것 같군요:

```
// 일반 메서드 버전
static int Add(int a, int b) { return a + b; }

// 람다 식 버전
(int a, int b) -> { return a + b; }

// 위의 람다 식을 축약한 버전
(a, b) -> a + b;
```

위의 예시에서 두 람다 식은 Add()와 동일한 의미(두 argument 를 더해 반환)를 갖습니다. 축약한 버전의 경우 얼핏 보기에 '너무 많이 축약한 듯' 보이지만, 적절한 위치에서 사용하는 경우 오류 없이 의도대로 원래 버전과 동일한 의미를 갖게 됩니다.

- 람다 식의 argument 형식은, 그 람다 식 주변 코드의 내용에 따라 자동적으로 결정됩니다. 예를 들어, CellInfo.CountIf_Players() 호출식 안에 들어가는 람다 식은, CountIf_Players()의 정의에 따라 그 argument 하나의 형식을 자동으로 PlayerInfo 형식으로 간주할 수 있습니다. 만약 Argument 가 하나만 있는 경우 a -> a + 3 처럼 argument 부분의 괄호도 생략할 수 있습니다.
- 만약 메서드 몸통이 return 문 하나만 필요하다면, 람다 식에서는 중괄호와 return 문을 생략하고 그냥 그 식만 적을 수 있습니다. 그렇지 않은 경우, 일반 메서드 몸통처럼 중괄호를 써서 전체 내용을 적어 주어야 합니다.

일반 메서드가 자신이 정의된 클래스의 field 를 볼 수 있었듯, 람다 식 또한 그러합니다. 다만, 우리가 상상하는 (그리고 다른 언어들의 람다 식이 모두 제공하는) 것과는 달리, Java 의 람다 식은 '그 람다 식이 정의된 메서드의 임시 변수'를 거의 사용할 수 없습니다. 따라서 **람다 식을 통해 내 어떤 변수의 값을 읽거나 바꾸고 싶을 때는, 반드시 그 변수를 (임시 변수가 아닌) field 로 만들어** 두고 사용해야 합니다. 한숨이 나오지만 이 기능은 2015 년에 들어서야 겨우 Java 에 도입된 기능이니 넓은 마음으로 이해해 주도록 합시다. 람다 식에 대한 더 자세한 내용이 궁금한 친구들은 조교에게 개인적으로 문의해 주세요.

코드 작성시 유의할 사항 모음

이 단원에서는 여러분이 플레이어 코드를 작성할 때 특히 유념해 두어야 할 내용들을 한 번 더 안내합니다.

- 분기문을 작성할 때는 '논리적 구멍'이 생기지 않도록 각별히 주의하세요!
  - 부등호를 쓸 때는 '두 값이 같을 때'를 잘 처리해 주어야 합니다.
  - 어느 방향을 골라야 할 지 모르겠을 때 무조건 Right로만 가게 만들면 플레이어가 강의실 오른쪽 끝에 있을 때 강의실을 나가 버릴 가능성이 큽니다.

- 배열을 쓰기 전에 대괄호 안에 넣을 숫자가 범위를 벗어나지는 않는지 꼭 확인하세요!
  - 배열 범위 오류는 '강의실 파괴 시도'로 간주되며 즉시 잡혀가게 됩니다.

- Random class를 사용하거나 static 키워드를 사용하면 안 됩니다!
  - 이 두 가지 조건은 몹시 중요하기 때문에, 여러분이 제출한 코드에서 이 요소들이 발견되면 해당 코드를 제거하고 '미제출'로 간주해**야만** 합니다. 꼭 지켜 주세요.

- 거듭 강조하지만, 플레이어 코드를 짜다 막히는 경우 꼭 조교의 도움을 요청하세요!

Bot 플레이어 코드

이번 프로젝트의 목적은 '생각을 코드로 옮겨 보는 것'이며 이는 '코드를 직접 적어 보는 것'과는 크게 다릅니다. l4g.bots package에는 여러분이 참고 가능한 몇 가지 Bot 플레이어 클래스(Bot_HornDone 제외)가 들어 있으며 이들은 각각 한 가지 목표를 가지고 의사 결정을 수행하도록 구성되어 있습니다.

Bot 플레이어 코드는 여러분이 자유롭게 가져와 사용할 수 있습니다(아예 코드 파일 내용 자체를 자유롭게 복붙해 올 수 있습니다). 단, Bot 플레이어의 특정 기능을 가져와 사용할 때는 반드시 해당 플레이어가 새로 정의한 필드들, 해당 필드를 초기화하기 위한 Soul_Stay() 내의 코드들도 함께 가져와 추가해 주어야 합니다. 이러한 주의사항들은 각 코드에 주석으로 자세히 설명되어 있으니, 꼼꼼하게 읽어 보고 적절히 복붙해 사용하면 되겠습니다.

모르는 부분이 생기면 반드시 조교에게 도움을 요청하세요. 그럼, 행운을 빕니다!