Left 4 Grade 2023 Document #3

~ developer's guide ~

목적

이 문서에는 Left 4 Grade 2023(이하 L4G) 프로젝트를 수행하는 여러분이 플레이어 코드를 작성할 때 **알아 두어야 하는**, 또는 **사용할 수 있을 만한** 내용들이 담겨 있습니다.

주요 파일들 소개

L4G2023 Java project에 들어 있는 일부 .java 파일의 내용을 간략하게 설명합니다.

Classroom.java (`l4g` package)

L4G의 '강의실'을 나타내는 클래스입니다. 매 게임마다 새로운 강의실 인스턴스를 만들어 사용합니다. L4G 참조 문서에 적힌 모든 규칙들이 작 적용되어 있으므로, 코드 내용을 직접 읽기보다는 Game Concepts 문서를 다시 한 번 구경해 보는 것을 권장합니다.

Constants.java (`l4g.common` package)

게임 내에서 사용하는 여러 값들을 미리 각각의 static 필드에 담아 두었습니다. 반복 흐름을 구성할 때, 아니면 단순히 강의실 가로 길이가 얼마나 긴 지 확인하고 싶을 때 자동 완성 기능을 사용해 가며 각 필드들을 확인해 주세요.

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

여러분의 플레이어가 게임을 진행하면서 매 턴 받게 될 정보들을 다루기 위한 클래스들입니다. 자세한 내용은 다음 페이지를 참고해 주세요.

Player.java (`l4g` package)

모든 플레이어들이 '공통적으로' 갖추어야 할 요소들에 대한 선언 및 정의가 적혀 있습니다. 여러분은 `Player class`의 '일종'인 새로운 클래스에 대한 클래스 정의를 구성하여 L4G 프로젝트에 참여하게 됩니다.

Player_YOURNAMEHERE.java (`l4g.customplayers` package)

미리 적어 둔, 여러분의 플레이어를 구성하기 위한 파일입니다.

Program.java ('default' package)

L4G '프로그램'을 구성하기 위한 파일입니다. 여러분의 플레이어를 등록하거나 게임 번호와 같은 각종 설정을 변경할 수 있습니다.

여러분의 플레이어 인스턴스에 기본으로 포함되어 있는 필드들

Player.java에는 모든 플레이어 인스턴스들이 각자 포함하게 될 필드들이 다수 선언되어 있으며, 이들은 여러분이 자주 사용할 예정이므로 각각의 의미를 파란 주석으로 기록해 두었습니다. 여기서는 그러한 주요 필드들에 대해 요약하면서 여러분이 각종 Data들을 어떻게 획득할 수 있는지 설명합니다.

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

- `String name`
  플레이어의 '이름'입니다. 여기서의 이름은 게임 밖 사람에게 무언가를 보여줄 때 출력을 위해 사용됩니다. 여러분의 플레이어는 다른 플레이어의 ID는 볼 수 있지만 이름은 볼 수 없습니다.

- `boolean trigger_acceptDirectInfection`
  '직접 감염'을 수락하려는 경우 미리 이 필드 자리에 `true`를 담아 두세요.

- `long gameNumber`
  현재 진행중인 게임의 '게임 번호'입니다. '엄청 큰 숫자 하나'가 필요할 때 사용할 수 있습니다. Bot들은 이 값과 자신의 ID 값을 사용하여 이동 또는 배치 의사 결정을 수행합니다.

- `TurnInfo turnInfo`
  이번 턴에 대한 정보가 들어 있습니다. `TurnInfo` 인스턴스에는 다음 필드들이 포함되어 있습니다:

  - `int turnNumber` – 이번 턴이 몇 번째 턴인지(첫 턴은 0턴)
  - `boolean isDirectInfectionChoosingTurn` – 이번 턴이 '직접 감염 여부'를 선택하는 턴인지

- `PlayerInfo myInfo`
  여러분 플레이어 자신의 현재 상태에 대한 정보가 들어 있습니다. `PlayerInfo` 인스턴스에는 다음 필드들이 포함되어 있습니다:

  - `int ID` - 플레이어마다 고유한 번호
  - `StateCode state` - 플레이어의 현재 상태
  - `int HP` - 현재 체력(시체, 감염체일 때 유효)
  - `int transition_cooldown` - 다음 상태로 전이하기까지 남은 턴 수(시체, 감염체, 영혼일 때 유효)
  - `Point_Immutable position` - 플레이어의 현재 위치

- `Score myScore`
  여러분의 플레이어가 현재 기록중인, 여섯 가지 평가 기준별 점수 값의 사본이 담겨 있습니다. (여기에 담겨 있는 점수 값은 부문별 기록을 의미하며, 이는 게임이 끝난 다음 산출되는 '학점'과는 다릅니다)

- `CellInfo[][] cells`
  현재 플레이어의 시야 범위를 감안한 각 칸별 정보들이 들어 있습니다. 아쉽지만 각 정보의 원본은 여러분이 직접 다룰 수 없으며, 대신 여러분은 별도로 마련해 둔 Data 분석용 메서드들을 사용하여 필요한 정보를 얻을 수 있습니다:

  - `Count_`계열 메서드들 – 현재 이 칸에 있는 요소(플레이어 / 행동 / 사건)의 수를 return합니다
  - `CountIf_`계열 – 여러분이 지정한 판정용 Code를 통해 해당 조건에 맞는 요소 수를 return합니다
  - `ForEach_`계열 – 여러분이 지정한 작업용 Code를 각 요소들에 대해 한 번씩 실행합니다
  - `Select_`계열 – 여러분이 지정한 판정용 Code를 통해 해당 조건을 만족하는 요소 목록을 만들어 return합니다

  이러한 메서드들을 사용하는 방법은 뒷 페이지에서 자세히 설명하고 있습니다. 위에서 언급한 '요소'란 `PlayerInfo class`, `Action class`, `Reaction class`의 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` 인스턴스에는 다음 필드들이 포함되어 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` 인스턴스에는 다음 필드들이 포함되어 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(사건을 일으킨 플레이어의 위치)입니다

좌표 다루기 – **Point class와 Point_Immutable class**

프로그래밍 동네에서 단어 'immutable'은 '변경 불가능한'을 의미합니다. 강의실 내의 각 칸에는 모두 고유한 Point_Immutable 인스턴스가 부여되어 있습니다. 그리고 이들을 통해 '특정 칸의 왼쪽 칸에 대한 좌표'와 같은 새로운 좌표를 만들어 다룰 때는 Point_Immutable class가 아닌 Point class의 인스턴스를 사용하게 됩니다. Point 인스턴스는 여러분이 자유롭게 새로 만들거나 편집할 수 있으며 Soul_Spawn()의 return값으로도 활용됩니다.

요약하면, 강의실이 여러분에게 나누어 주는 좌표들은 모두 immutable합니다. 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 칸이 내가 현재 있는 칸에서 **2**칸 떨어져 있는지 확인하기

```
int row = 3;      // 확인하려는 칸의 세로 좌표
int column = 4;   // 확인하려는 칸의 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 좌표 값은 위에서 아래로 증가
int offset_column = -1;   // 가로 좌표 값은 왼쪽에서 오른쪽으로 증가
Point dest = myInfo.position.Offset(offset_row, offset_column);
```

위의 예시들처럼, 여러분은 보통 '내 위치'를 기준으로 좌표 관련 연산을 수행할 것입니다. 만약 자신이 직접 만든 Point 인스턴스를 기준으로 연산을 수행하고 싶은 경우에도, 그냥 해당 그 인스턴스에 .을 찍고 동일한 메서드를 호출해 쓰면 됩니다(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`의 내 왼쪽 칸 좌표에 해당하는 곳에 들어 있습니다.

이 때, `cells[-1][-1]`처럼 '강의실 바깥'에 해당하는 Data를 얻으려 시도하면 그 즉시 붙잡혀 10턴 동안 영혼 상태가 되는 페널티를 받게 됩니다. 그러므로 `cells`를 사용할 때는 좌표의 유효성을 항상 보장해 주어야 합니다.

대충 경각심을 가지게 되었다면, 이제 `cells`에 담긴 Data를 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 세어 return해 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 인수로 '요소 하나를 인수로 받고 `boolean` 형식 값을 return하는 Code 덩어리' 하나를 받습니다. 위에서는 '어떤 플레이어의 상태가 감염체인지 여부를 return하는' 람다 식을 적어 판정용 Code로써 집어넣고 있습니다(람다 식에 대해서는 이 문서의 후반부에서 다시 설명하고 있습니다). 이렇게 `CountIf_`계열 메서드를 호출하면, 인수로 지정된 Code 덩어리로 각 요소를 한 번씩 검사하여 `true`가 몇 번 return되었는지를 집계해 return합니다

  - 위의 코드에서 `Infected` 자리에 `Survivor`를 적으면 생존자 수를 집계할 수 있습니다. 사실 이 Code 덩어리는 여러분이 가장 빈번하게 사용할 것이라 말해도 과언이 아닐 것입니다. 따라서 후반부 설명을 봐도 이해가 잘 안 되면 반드시 강사에게 문의해 주세요!

- 각 칸별 시체 수를 별도로 세어 내가 직접 선언해 둔 `int[][]` 필드 자리에 담기

```
// 주의: 이 아래에 있는 선언은 counts가 필드 이름이 되도록 적절한 자리에 적어 주세요.
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 필드를 선언해 두고, 중첩된 for문을 적어 강의실 내 각 칸에 대해 CountIf_계열 메서드를 적절히 호출하도록 Code 흐름을 구성할 수 있습니다

- 사전 답사에서 본 것처럼, 내 시야 범위 바깥에 있는 칸의 경우 거기에 실제 누군가 있다 하더라도 내 관점에서는 아무도 없는 것으로 간주될 수 있으며, 그러한 칸에 대해서도 오류 없이 Data 획득용 메서드를 사용할 수 있습니다. 각 상태별로 보유한 추가적인 능력(생존자가 발견 정보를 공유받음, 시체가 자신의 위에 누가 있는지 감지, 감염체가 모든 시체의 존재를 감지) 또한 각 칸의 Data를 확인하며 사용할 수 있으며, 이 때 시야 범위 바깥의 플레이어 정보는 확인 가능하나 해당 칸에서 발생한 행동 및 사건 정보는 확인할 수 없습니다(아무 일도 일어나지 않았던 것으로 간주됩니다)

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

```
// 주의: 이 아래에 있는 두 선언은 각 이름들이 필드 이름이 되도록 적절한 자리에 적어 주세요.
Pos_Immutable pos_max = null;
int HP_max = -1;

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 담겨 있음!
```

- ForEach_계열 메서드들은 인수로 '요소 하나를 인수로 받고 값을 return하지 않는 Code 덩어리' 하나를 받습니다. ForEach_계열 메서드를 호출하면 각 요소를 한 번씩 담아 인수로 지정된 Code 덩어리를 실행해 줍니다. 위에서는 각 감염체의 HP를 현재까지 찾아 둔 HP 최대값과 비교함으로써 가장 HP가 많은 감염체가 서 있는 칸의 좌표를 pos_max에 담아 내고 있습니다

이외에도 '조건에 맞는 요소 목록'을 만들어 return하는 Select_계열 메서드가 존재하지만, 여러분이 몹시 복잡한 플레이어를 추구하지 않는 이상 해당 메서드는 여간해서는 쓸 일이 없을 것입니다. Select_계열 메서드는 CountIf_계열 메서드처럼 판정용 Code 덩어리를 인수로 받고, 해당 조건에 맞는 요소만 모아 둔 새로운 ArrayList<> 인스턴스를 return해 줍니다. 여러분은 return된 인스턴스를 어딘가에 담아 둔 다음 적절한 반복 흐름을 구성하여 각 요소의 내용을 더 정밀하게 관찰할 수 있습니다.

'람다 식'에 대해

람다 식(lambda expression)은 '수식 적는 자리에 적을 수 있는 Code 덩어리' 입니다. '어떤 플레이어가 생존자라면 `true`를 return'과 같은 Code를 클래스 정의 안 다른 곳에 메서드 정의로서 적어 둘 수 있겠지만, 람다 식을 사용하면 그 Code를 실제로 다루게 될 문장 근처에 Code 본문을 적어 둘 수 있습니다. 람다 식은 우리가 지금 이해하기에는 조금 하지만, 일단 L4G에서는 그냥 이 정도로만 이해하고 사용해도 충분할 것입니다.

람다 식은 보통 아래와 같이 생겼습니다:

```
인수_선언 -> Code 본문
```

람다 식의 계산 결과값은 메서드와 같이 '지금 본문을 적어 두고 나중에 인수를 담아 호출 가능'하며, 그렇다 보니 람다 식의 생김새는 메서드 정의와 비슷합니다. 람다 식의 `->` 왼쪽 부분은 메서드 정의 첫 줄의 인수 선언 부분에 해당하며, 오른쪽 부분은 메서드 정의 본문(내용물)에 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 인수 값을 더해 return)를 갖습니다. 축약한 버전의 경우 얼핏 보기에 '너무 많이 축약한 듯' 보이지만, 적절한 위치에 적어 두는 경우 오류 없이 원래 버전과 동일한 의미를 갖게 됩니다(달리 말하면 Java에서 람다 식은 아무 곳에나 적어 둘 수 없도록 되어 있습니다).

- 람다 식의 인수 형식은, 직접 지정하지 않는다면, 그 람다 식 주변 Code의 내용에 따라 결정됩니다. 예를 들어, `CellInfo.CountIf_Players()` 호출식 안에 적어 둔 람다 식은, `CountIf_Players()`의 정의에 따라 '`PlayerInfo` 하나를 받는 Code 덩어리'로 간주될 수 있습니다. 만약 인수를 하나만 다루는 경우에는 `a -> a + 3`처럼 람다 식 인수 부분의 괄호를 생략해 적을 수 있습니다.
- 람다 식의 본문을 수식 하나(내지는 그 수식을 포함하는 return문)만 사용해서 적을 수 있다면, 중괄호와 return문을 생략하고 그냥 그 수식만 적어도 됩니다. 그렇지 않은 경우, 일반 메서드 정의를 적을 때처럼 중괄호를 써서 전체 내용을 문장들의 나열로 적어 주어야 합니다.

일반 메서드 정의를 적을 때 그 메서드가 소속되어 있는 클래스의 이름들을 사용할 수 있었듯, 람다 식 또한 비슷한 느낌으로 scope 관점에서 각종 이름들을 사용할 수 있습니다. 다만, 우리가 상상하는 (그리고 다른 프로그래밍 언어들의 람다 식 기능에 모두 포함되어 있는) 것과는 달리, Java의 람다 식을 적을 때는 '그 람다 식을 적고 있는 메서드 정의 안에서 선언한 지역 변수'를 거의 사용할 수 없습니다. 따라서 **람다 식을 통해 내 어떤 Data 이름 자리에 담긴 값을 읽거나 바꾸고 싶을 때는, 반드시 그 이름을 (지역 변수가 아닌) 필드로서 선언해** 두고 사용해야 합니다. 한숨이 나오지만 Java의 람다 식은 2015년에 들어서야 겨우 도입된 기능이니 넓은 마음으로 이해해 주도록 합시다. 람다 식에 대한 더 자세한 내용이 궁금한 친구들은 강사에게 개인적으로 문의해 주세요.

코드 작성시 유의할 사항 모음

여러분이 플레이어 코드를 작성할 때 특히 유념해 두어야 할 내용들을 적어 두었습니다.

- (중요)L4G의 정규 게임 진행 결과는 수강생 여러분의 실제 학점에 반영됩니다. 따라서 객관성을 도모하기 위해 같은 조건에서 진행한 게임은 항상 같은 결과를 내도록 구성되어야 합니다. 이러한 목표를 달성하기 위해, 여러분의 플레이어는 절대 프로그램 밖에서 Data를 가져와 사용하지 않아야 합니다. 예를 들어, 키보드 입력을 받거나, 현재 시각 값을 계산에 활용하거나, 파일에 무언가를 담아 두었다가 읽어오거나 해서는 안 됩니다. Java 동네에서 랜덤 값을 다룰 때 사용하는 `java.util.Random`도, 기본적으로 매 실행마다 서로 다른 값을 제공하도록 되어 있으니 사용하지 말아주세요. 그렇기는 하지만, 게임 번호와 같이 게임 규칙 범위 안에서 여러분의 플레이어가 얻을 수 있는 Data들을 조합하여 새로운 Data를 만들어 의사 결정에 사용할 수는 있습니다. Bot들은 모두 이러한 방식을 사용하여 서로 다른 의사 결정을 수행할 수 있도록 구성되어 있습니다.

- (중요)학점에 반영될 정규 게임은 총 100,000번의 게임을 수행하게 되며 따라서 원활하게 정규 게임을 진행하기 위해서는 성능 관련 이슈를 중요하게 고려해야 합니다. 자세한 설명은 생략하고, **여러분의 .java 파일에는 절대 'static'이 적혀 있어서는 안 됩니다!**

- 여러분이 지금 시점에 사용할 수 있을 만한 모든 이름에 파란 주석으로 설명을 달아 두었습니다. 주석 내용과 함께 수업시간에 다루었던 것들, 이 문서에 적혀 있는 내용을 확인하면서 천천히 진행해 보세요.

- 자꾸 내 플레이어가 영혼 페널티를 받는다면, 아래 내용을 체크해 보세요:
  - 부등호 연산자를 사용할 때는 '두 값이 같을 때' 어떻게 할 것인지 잘 정해야 합니다
  - 어느 방향으로 이동하든 상관없을 것 같을 때 무조건 Right로만 가게 만들면 플레이어가 강의실 가장 오른쪽 칸에 있을 때 강의실을 나가 버릴 가능성이 있습니다
  - 내 플레이어가 어떤 게임의 몇 턴 째에 페널티를 받는지 확인하고, 수업시간에 같이 진행해 본 테스트 방법을 활용하여 해당 상황을 재연해 보세요. Code 흐름의 허점을 발견하고 보완하기 위한 실마리를 찾을 수 있을 것입니다

- 아이디어는 있는데 어떻게 코드로 옮길 지 모르겠다 싶을 때는 언제든 강사의 도움을 요청할 수 있습니다. 종이 등에 그려 둔 스케치를 지참하여 연락해 주세요.

Bot 플레이어 코드

이번 프로젝트의 목적은 '생각을 코드로 옮겨 보는 것'이며 이는 '코드를 직접 적어 보는 것'과는 크게 다릅니다. `14g.bots package`에는 여러분이 이미 사전 답사에서 겨루어 본 몇 가지 Bot들에 대한 클래스들이 소속되어 있으며, `Bot_HornDone`을 제외한 다른 Bot들은 각각 여섯 가지 평가 기준들 중 하나를 극단적으로 추구하도록 구성되어 있습니다.

Bot용 코드는 여러분이 자유롭게 복붙하여 사용할 수 있습니다. 단, Bot의 특정 기능을 그대로 가져와 사용할 때는 그 Code 덩어리(예를 들면 메서드 정의)만 잘라 오는 것이 아니라 그 안에서 사용할 Data를 다루기 위한 별도의 선언들, 첫 턴에 각종 필드들에 적절한 값을 담아 두기 위한 `Soul_Stay()` 정의 안 코드들도 함께 가져와 추가해 주어야 합니다. 이러한 주의 사항들은 각 .java 파일들에 주석으로 자세히 설명해 두었으니, 꼼꼼하게 읽어 보고 적절히 복붙해 사용하면 되겠습니다.

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