COMP2500 과제 4

목차

'바비 로즈 미술대학'(COMP1500의 과제 2 기억하시죠? ;))의 첫 졸업반인 여러분은 학우들과 함께 디지털 아티스트를 위한 앱, '아스키 아트 그리기 앱(AADA, ASCII Art Drawing App)'을 만들기로 결정했습니다. 예전에 2D 캔버스에 원, 삼각형 등을 그리는 코드를 작성했던 날을 떠올려보니 이건 프로그래머가 아닌 사람들에게 너무나 괴로운 일처럼 느껴졌거든요. 따라서 디지털 아티스트들이 코드 작성 없이 이런 일을 할 수 있는 앱을 만들게 된 겁니다. 이 앱은 코드를 작성할 줄을 모르지만 의욕이 넘치는 디지털 아티스트들이 쉽게 아스키 문자로 그림을 그릴 수 있게 해 줄 것입니다. 이 제품을 팔기 위해 동네 최고의 마케터도 구해놨으니 이제 만들기만 하면 되겠네요!

1. 프로젝트를 준비한다

1. `Assignment4` 프로젝트를 엽니다.
2. 다음의 `.java` 파일을 만듭니다.
    - `ICommand.java`
    - `Canvas.java`
    - `OverdrawAnalyzer.java`
    - `CommandHistoryManager.java`
3. 자, 이제 자리에서 일어나서 국민체조 한 번 하시고... 설계와 프로그래밍 시작하겠습니다. :)

2. 아스키 아트 그리기 앱 설계하기

전반적인 규칙

- 아래에 나와있는 제품 사양에 따라 마음껏 클래스를 설계하세요. 그러나 빌드봇이 여러분의 설계가 적절하지 못하다 생각하면 점수를 깎을 것입니다. :P
- 여러분이 작성하는 클래스들은 반드시 `academy.pocu.comp2500.assignment4` 패키지 안에 속해야 합니다. 그렇지 않으면 빌드봇이 그 클래스들을 무시합니다.
- `instanceof`, `Object.getClass()` 그리고 `Class<T>.cast()` 메서드를 사용할 수 없습니다. 사용하면 곧바로 0점이 뜨니 시도조차 하지 마세요. :)
- `private static final` 멤버 변수 외에 어떤 정적 멤버 변수도 사용할 수 없습니다.
- `public static`와 패키지 `static` 메서드를 사용할 수 없습니다.
- 내포(inner) 클래스 및 내포 열거형의 사용을 금합니다.
- 클래스 내부에 데이터를 저장할 때 사용하는 자료형은 직접 만들어 사용하세요. (예외: String과 HashMap, ArrayList, HashSet와 같은 컨테이너, 그리고 기본 자료형)
- 캔버스에 그림을 그릴 때 원점은 (0,0)이며 +x는 오른쪽, +y는 아래쪽입니다.
- `char`형 인자는 모두 화면에 출력 가능한 ASCII 문자라고 가정하세요.
- 출력 가능한 ASCII 문자의 범위는 32부터 126입니다. [32, 126]
- 캔버스는 어떤 경우에도 출력 불가능한 ASCII 문자를 가지고 있으면 안 됩니다.
- 어떤 클래스 대신 사용할 수 있는 기본 자료형이 존재한다면(예: Boolean/boolean, Integer/int) 그걸 대신 사용하세요. 기본 자료형 대신 클래스 버전이 허용되는 경우는 제네릭(generic) 클래스의 타입(type) 매개변수로 사용할 때 뿐입니다.

2.1 `Canvas` 클래스 구현하기

AADA의 첫 버전은 간단한 기능들만 지원할 예정입니다.

1. 픽셀 하나를 그린다
2. 수직선/수평선 하나를 그린다
3. 픽셀 하나를 변경하는 다양한 도우미 함수들

원, 삼각형, 사각형 등의 복잡한(?) 도형을 그리는 기능은 먼 훗날 언젠가 만들 계획이라죠.

여러분이 위 기능들을 `Canvas` 라는 이름의 클래스 하나에 모두 구현하면, 다른 동료가 각 메서드를 GUI 도구 키트(GUI toolkit)에 연결해 둔다고 하니 이 부분은 걱정하지 마세요.

Canvas 클래스를 구현할 때 한 가지 주의할 점! 별도의 public, 패키지, protected 메서드/멤버 변수를 추가하는 것을 불허합니다. 단, 다음은 허용됩니다.

1. `private` 메서드/멤버 변수
2. `public getWidth()` 메서드
3. `public getHeight()` 메서드

2.1.1 생성자를 구현한다

- 생성자는 다음의 인자를 받습니다.
  - 캔버스의 너비를 나타내는 `int`
  - 캔버스의 높이를 나타내는 `int`
- 너비와 높이는 음수가 아니라고 가정해도 좋습니다.
- 각 픽셀의 초기값은 `' '` 입니다.

2.1.2 `drawPixel()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 픽셀의 y 위치를 나타내는 `int`
  - 지정된 픽셀에 그릴 문자를 나타내는 `char`
- 지정된 픽셀에 지정된 문자를 그립니다.
- 이 메서드는 아무것도 반환하지 않습니다.

2.1.3 `getPixel()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 픽셀의 y 위치를 나타내는 `int`
- 이 메서드는 지정된 픽셀에 있는 문자를 반환합니다.

2.1.4 `increasePixel()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 픽셀의 y 위치를 나타내는 `int`
- 지정된 픽셀에 있는 문자의 아스키 값을 1만큼 증가시킵니다.
- 지정된 픽셀에 있는 문자의 아스키 값이 1만큼 증가되었다면 `true`를, 아니면 `false`를 반환합니다.

2.1.5 `decreasePixel()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 픽셀의 y 위치를 나타내는 `int`
- 지정된 픽셀에 있는 문자의 아스키 값을 1만큼 감소시킵니다.
- 지정된 픽셀에 있는 문자의 아스키 값이 1만큼 감소되었다면 `true`를, 아니면 `false`를 반환합니다.

2.1.6 `toUpper()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 픽셀의 y 위치를 나타내는 `int`
- 지정된 픽셀에 있는 문자를 대문자로 변경합니다.
- 이 메서드는 아무것도 반환하지 않습니다.

2.1.7 `toLower()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 픽셀의 y 위치를 나타내는 `int`
- 지정된 픽셀에 있는 문자를 소문자로 변경합니다.
- 이 메서드는 아무것도 반환하지 않습니다.

2.1.8 `fillHorizontalLine()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 y 위치를 나타내는 `int`
  - 지정된 행(row)에 있는 모든 픽셀에 그릴 문자를 나타내는 `char`
- 지정된 행 전체를 지정된 문자로 채워 넣습니다.
- 이 메서드는 아무것도 반환하지 않습니다.

2.1.9 `fillVerticalLine()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 지정된 열(column)에 있는 모든 픽셀에 그릴 문자를 나타내는 `char`
- 지정된 열 전체를 지정된 문자로 채워 넣습니다.
- 이 메서드는 아무것도 반환하지 않습니다.

2.1.10 `clear()` 메서드를 구현한다

- 이 메서드는 아무 인자도 받지 않습니다.
- 캔버스에 있는 모든 픽셀을 지웁니다.
- 이 메서드는 아무것도 반환하지 않습니다.

2.1.11 `getDrawing()` 메서드를 구현한다

- 이 메서드는 아무 인자도 받지 않습니다.

- 현재 캔버스에 그려있는 그림을 다음 포맷으로 반환합니다.

```
Canvas canvas = new Canvas(10, 10);

canvas.drawPixel(0, 0, '*');
canvas.drawPixel(1, 2, '$');
canvas.drawPixel(0, 1, '&');

System.out.println(canvas.getDrawing());
```

```
+----------+
|*         |
|&         |
| $        |
|          |
|          |
|          |
|          |
|          |
|          |
|          |
|          |
+----------+
```

** **주의**: 마지막 `+` 문자 이후에 줄 바꿈을 해야 합니다.

2.2 실행취소(undo)/다시실행(redo) 기능 구현하기

첫 버전 출시가 매우 성공적이었습니다. 하지만 불편함을 호소하는 최종 사용자들이 꽤 있네요. undo/redo 기능이 없어 너무 실수로 그림을 망치는 경우가 너무 많다고 합니다. 따라서 버전 2에서는 이 기능들을 넣기로 결정하였습니다. 커맨드(command) 패턴을 사용하면 각 연산을 개체(object)로 표현하고, 또 다른 클래스를 만들어 이들을 저장하고 관리할 수 있습니다. 이 패턴의 장점 중 하나는 최종 사용자가 이 앱에 내린 모든 커맨드의 히스토리를 기억할 수 있다는 거지요. 빈 캔버스에 이 커맨드들을 처음부터 쭈욱 재실행하기만 하면 똑같은 결과를 볼 수 있답니다!

2.2.1 `ICommand` 인터페이스를 구현한다

- `ICommand` 인터페이스에는 다음의 메서드가 있어야 합니다.
  - `execute()`
    - 유일한 인자로 `Canvas` 를 받습니다.
    - 본 커맨드를 실행했다면 `true` 를, 아니면 `false` 를 반환합니다.
  - `undo()`
    - 아무 인자도 받지 않습니다.
    - 본 커맨드의 실행을 취소했다면(undo) `true` 를, 아니면 `false` 를 반환합니다.
  - `redo()`
    - 아무 인자도 받지 않습니다.
    - 본 커맨드를 다시 실행했다면(redo) `true` 를, 아니면 `false` 를 반환합니다.
- `ICommand` 에 다른 메서드를 추가하는 것을 금합니다.
- 이미 `execute()` 메서드를 호출한 커맨드에서 다시 `execute()` 를 호출할 경우 커맨드는 처리가 되지 않아야 합니다. 예를 들어 'A' 캔버스에 어떤 커맨드 개체를 이미 실행(execute)했다면 이 개체의 `execute()` 메서드를 다시 호출해봐야 아무 소용이 없습니다.

2.2.2 구체 커맨드 클래스들을 구현한다

- 모든 구체 커맨드 클래스는 `ICommand` 인터페이스를 구현해야 합니다.

- 구현해야 할 커맨드 클래스는 총 8개입니다.

  - 캔버스에 있는 픽셀 하나에 지정된 문자를 그리라고 명령하는 커맨드
  - 캔버스에 있는 한 픽셀 저장된 아스키 값을 1만큼 증가시키라고 명령하는 커맨드
  - 캔버스에 있는 한 픽셀 저장된 아스키 값을 1만큼 감소시키라고 명령하는 커맨드
  - 캔버스에 있는 한 픽셀을 대문자로 변경하라고 명령하는 커맨드
  - 캔버스에 있는 한 픽셀을 소문자로 변경하라고 명령하는 커맨드
  - 캔버스에 있는 한 행을 모두 지정된 문자로 채우라고 명령하는 커맨드
  - 캔버스에 있는 한 열을 모두 지정된 문자로 채우라고 명령하는 커맨드
  - 캔버스를 깨끗이 지우라고 명령하는 커맨드
- 팀장님이 커맨드 개체를 생성할 때 쓸만한 매개변수명이 다음과 같다고 하십니다.

```
x
y
i
j
width
height
character
c
point
position
coordinate
vector
pixel
size
```

하지만 팀장님이 대충 뽑은 목록이니 이 중 일부는 적절하지 않을 수도 있다고 하네요. 잘 생각해서 매개변수명을 정하세요.

2.2.3 `CommandHistoryManager` 클래스 구현하기

`CommandHistoryManager`는 커맨드 실행과, 실행된 커맨드들의 히스토리 관리를 담당합니다.

2.2.3.1 생성자를 구현한다

- 이 생성자는 다음의 인자를 받습니다.
  - `Canvas`

2.2.3.2 `execute()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - `ICommand`
- 지정된 커맨드를 캔버스에 적용합니다.
- 커맨드가 성공적으로 실행되면 `true`를, 아니라 `false`를 반환합니다.

2.2.3.3 `canUndo()` 메서드를 구현한다

- 이 메서드는 아무 인자도 받지 않습니다.
- undo를 할 수 있는 커맨드가 있다면 `true`를, 아니면 `false`를 반환합니다.

2.2.3.4 `canRedo()` 메서드를 구현한다

- 이 메서드는 아무 인자도 받지 않습니다.
- redo를 할 수 있는 커맨드가 있다면 `true`를, 아니면 `false`를 반환합니다.

2.2.3.5 `undo()` 메서드를 구현한다

- 이 메서드는 아무 인자도 받지 않습니다.
- 가장 최근에 캔버스에 적용했던 커맨드의 실행을 취소합니다.
- 가장 최근에 적용했던 커맨드의 실행을 취소했다면(undo) `true`를, 아니면 `false`를 반환합니다.

2.2.3.6 `redo()` 메서드를 구현한다

- 이 메서드는 아무 인자도 받지 않습니다.
- 가장 최근에 실행 취소된 커맨드를 다시 캔버스에 적용합니다.
- 가장 최근에 실행 취소된 커맨드를 다시 실행했다면(redo) `true`를, 아니면 `false`를 반환합니다.
- 만약에 `redo()`를 호출하기 전에 완전히 새로운 커맨드가 캔버스에 적용되었다면 redo를 할 수 없습니다.

2.3 `OverdrawAnalyzer` 클래스 구현하기

이제 버전 2도 출시되었고, undo/redo 기능은 엄청난 히트였습니다. 그러나 이 앱을 오랫동안 실행하면 속도가 엄청 느려진다는 불평소리가 들리네요.

현재 추측으로는 undo/redo 할 때마다 픽셀 값을 바꿔서 인 것 같습니다. 화면에 그려지는 픽셀 값을 바꿀 때마다 어느 정도 하드웨어 성능이 소모되거든요. 이 문제를 분석하기 위해 `OverdrawAnalyzer` 클래스를 작성하기로 결정했습니다. 캔버스 대신 이 클래스를 사용하면 하드웨어 수준에서 각 픽셀이 업데이트되는 횟수를 알아낼 수 있습니다.

그러나 이 프로그램이 실행되는 하드웨어에는 한 가지 최적화 기능이 내장되어 있답니다. 픽셀의 값을 기존과 동일한 값으로 바꾸려 하면 하드웨어는 아무런 연산도 하지 않습니다. 즉, 이런 연산은 하드웨어 성능을 전혀 소모하지 않는다는 거죠. 이 클래스를 구현할 때 이런 점까지 감안하여 올바르게 성능을 분석하세요!

2.3.1 메서드를 오버라이딩한다

픽셀의 새 값이 기존 값과 동일한 경우 하드웨어는 이 그리기 연산을 무시합니다. 이런 동작이 픽셀 히스토리에 올바로 반영되도록 `Canvas` 클래스에 있는 몇몇 메서드들을 오버라이딩해야 할 수도 있습니다.

2.3.2 `getPixelHistory()` 메서드를 구현한다

- 이 메서드는 다음의 인자를 받습니다.
  - 픽셀의 x 위치를 나타내는 `int`
  - 픽셀의 y 위치를 나타내는 `int`
- 한 픽셀의 업데이트 히스토리를 `LinkedList<Character>`로 반환합니다. 이 픽셀이 가장 오래전에 그려진 문자가 첫 번째에, 가장 최근에 그려진 문자가 마지막에 위치합니다.

2.3.3 `getOverdrawCount()` 메서드를 구현한다

- 두 개의 오버로딩된 메서드가 있어야 합니다.
  1. 다음의 인자를 받으며, 지정된 픽셀이 덮어 쓰인(overdraw) 총횟수를 반환합니다.
     - 픽셀의 x 위치를 나타내는 `int`
     - 픽셀의 y 위치를 나타내는 `int`
  2. 아무 인자도 받지 않으며 캔버스 전체에서 픽셀이 덮어 쓰인 총횟수를 반환합니다.

- **주의**: 이 메서드는 하드웨어 수준에서 픽셀이 **덮어 쓰인** 횟수를 반환합니다. 픽셀에 값을 대입한 횟수가 아닙니다.

3. 본인 컴퓨터에서 테스트하는 법

이 과제는 여러분이 직접 설계하는 클래스들이 있기에 정형화된 테스트 코드를 제공해 드리는 것이 불가능합니다. 따라서 본 과제에서 작성한 코드를 테스트하는 것도 궁극적으로는 여러분의 몫입니다. 단, 몇 가지 테스트 케이스들을 아래에 글로 적어 둘 테니 테스트할 때 참고하세요. 이 외에도 더 많은 테스트를 해야 할 것입니다. 테스트를 많이 할수록 버그가 적어진다는 사실, 잘 아시죠? :)

3.1 기본 캔버스 테스트

1. `Canvas` 를 생성한다.
2. 한 픽셀에 문자 하나를 그린다. (`drawPixel()`)
3. 2의 픽셀 값을 읽어와 그 값이 올바른지 확인한다.
4. 다음의 연산들에 대해 1~3 단계를 반복한다.
    - `increasePixel()`
    - `decreasePixel()`
    - `toUpper()`
    - `toLower()`
5. 한 행을 동일한 문자로 채운다. (`fillHorizontalLine()`)
6. 그 행에 있는 모든 픽셀이 올바른 값을 가지는지 확인한다.
7. `fillVerticalLine()` 에 대해 5~6 단계를 반복한다.
8. 캔버스를 지우고 캔버스에 있는 모든 문자가 지워졌는지 확인한다.

3.2 실행취소(undo)/다시실행(redo) 테스트

1. `Canvas` 를 생성한다.
2. `CommandHistoryManager` 를 생성한다.
3. 픽셀 하나를 그리는 커맨드를 생성한다.
4. 3의 커맨드를 실행한다.
5. 4에서 실행한 커맨드를 undo 한다.
6. 픽셀이 빈 상태가 되었는지 확인한다.
7. 4의 커맨드를 redo 한다.
8. 픽셀이 올바른 문자로 바뀌었는지 확인한다.
9. 다른 커맨드에 대해 3~8 단계를 반복한다.

3.3 픽셀 덮어쓰기(overdraw) 히스토리 테스트

1. `OverdrawAnalyzer` 를 생성한다.
2. 픽셀 하나에 'c'를 그린다.
3. 동일한 픽셀에 'd'를 그린다.
4. 픽셀 히스토리를 구해온다. 예상 결과: 'c' -> 'd'
5. 동일한 픽셀에 'd'를 그린다.
6. 픽셀 히스토리를 구해온다. 예상 결과: 여전히 'c' -> 'd'
7. 다른 그리기 커맨드에 대해 2~6 단계를 반복한다.

3.4 덮어쓰기 횟수 테스트

1. `OverdrawAnalyzer` 를 생성한다.
2. 픽셀 하나에 'c'를 그린다.
3. 동일한 픽셀에 'd'를 그린다.
4. 덮어쓰기 횟수를 구해온다. 이 횟수가 2인지 확인한다.
5. 동일한 픽셀에 'd'를 그린다.
6. 덮어쓰기 횟수를 구해온다. 이 횟수가 여전히 2인지 확인한다.
7. 다른 그리기 커맨드에 대해 2~6 단계를 반복한다.

4. 클래스와 메서드 등록하기

본인 컴퓨터에서 프로그램 테스트를 마쳤다면 `academy.pocu.comp2500.assignment4.registry` 패키지 안에 있는 `Registry` 클래스에 여러분의 메서드들을 등록하세요. 그래야 빌드봇이 여러분의 코드를 테스트할 때 어떤 public 메서드들을 호출해야 되는지 알 수 있답니다. `Registry` 클래스에는 총 7개의 `registerXXX()` 메서드가 있습니다.

1. `registerDrawPixelCommandCreator`: 한 픽셀에 문자 하나를 그리는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
2. `registerIncreasePixelCommandCreator`: 한 픽셀에 있는 문자 값을 1만큼 증가시키는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.

3. `registerDecreasePixelCommandCreator`: 한 픽셀에 있는 문자 값을 1만큼 감소시키는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
4. `registerToUppercaseCommandCreator`: 한 픽셀에 있는 문자를 대문자로 변경하는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
5. `registerToLowercaseCommandCreator`: 한 픽셀에 있는 문자를 소문자로 변경하는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
6. `registerFillHorizontalLineCommandCreator`: 한 행을 같은 문자로 채우는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
7. `registerFillVerticalLineCommandCreator`: 한 열을 같은 문자로 채우는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.
8. `registerClearCommandCreator`: 캔버스를 지우는 커맨드를 만드는 생성자나 메서드를 등록한다. 2개의 오버로딩된 메서드가 있으니 본인의 설계에 적합한 버전을 사용할 것.

`academy.pocu.comp2500.assignment4.registry` 안에 들어있는 클래스들을 **변경하지 마세요**. 전혀 그럴 필요 없고 그냥 사용하시기만 하면 돼요!

`Registry` 클래스를 사용하는 방법을 예를 들어 설명하겠습니다. 여러분이 픽셀 하나를 그리는 커맨드를 나타내는 `Foo` 라는 클래스를 만들었다고 합시다. 그러면 `App` 클래스 생성자에서 다음과 같이 `registerDrawPixelCommandCreator()` 를 호출해 주세요.

```
package academy.pocu.comp2500.assignment4;

import academy.pocu.comp2500.assignment4.registry.Registry;

public class App {
    public App(Registry registry) {
        ...

        registry.registerDrawPixelCommandCreator("Foo");

        ...
    }
}
```

그게 아니라 `Foo` 클래스에 정의된 `bar()` 메서드가 그 커맨드를 만든다면 다음과 같이 해주세요.

```
package academy.pocu.comp2500.assignment4;

import academy.pocu.comp2500.assignment4.registry.Registry;

public class App {
    public App(Registry registry) {
        ...

        registry.registerDrawPixelCommandCreator("Foo", "bar");

        ...
    }
}
```

이러면 빌드봇이 픽셀 그리기 커맨드를 만드는 법을 알게 됩니다.

모든 메서드들을 등록한 뒤에는 `Program.java` 속에 아래의 코드를 추가한 뒤, 과제를 제출하기 전에 실행해보세요.

```
package academy.pocu.comp2500.assignment4.app;

import academy.pocu.comp2500.assignment4.App;
import academy.pocu.comp2500.assignment4.registry.Registry;

public class Program {

    public static void main(String[] args) {
        Registry registry = new Registry();
        App app = new App(registry);
        registry.validate();
    }
}
```

`validate()` 메서드는 `Registry` 클래스에 메서드를 등록할 때 오타 등을 내지 않았는지 확인해줍니다. 빌드봇이 메서드들을 찾지 못해 어쩔 수 없이 0점을 주기 전에 미리 확인해보는 게 좋겠죠? :) 참고로 이 메서드는 `assert` 키워드를 사용하니 적절한 VM 옵션을 지정해주는 것도 잊지 마세요.

5. 커밋, 푸시 그리고 빌드 요청

이건 어떻게 하는지 이제 다 아시죠? :)