문제해결 및 실습 : Java

과제 5 보고서

세종대학교

제출일	2024.12.23	전공	소프트웨어학과
과목	문제해결및실습:Java	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. 블럭 격파 게임(Block Breaker)

a) 문제 분석

Java Swing을 이용하여 블럭을 격파하는 게임을 만드는 과제입니다. 주어진 예제를 참고하고, 추가적인 기능들, 사운드 등을 이용하여 보다 재미있는 게임을 만드는 것이 주 목적입니다. 각 씬들을 클래스로써 분리하고, 최대한 객체 지향적인 방식으로 설계하는 것이 추가 목적입니다. 유니티(Unity ; 게임 엔진)으로 게임을 제작해본 경험을 바탕으로 비슷한 설계 구조와 논리로 프로그램의 기틀을 잡아보았습니다.

가능하다면 오브젝트들은 GameObject 클래스를 상속받게 끔. 이는 최대한 객체 지향적인 방식으로 설계하는 목표에 해당합니다. 함수를 적절히 Overriding하고, 목적에 맞추어 변형시키려 했습니다. 추상 클래스가 GameObject에 해당하고, 이를 상속받는 것들이 모두 자식 클래스에 해당합니다.

게임의 재미도와 씬의 분리, 게임의 관리 등은 문제 풀이에서 설명하겠습니다.

b) 문제 풀이

먼저 씬의 분리입니다. 씬은 크게 StartScene, GameScene, GameOverScene 세 가지로 나뉩니다. 이들은 각각의 클래스로 구현되었고, JPanel을 상속받습니다.

StartScene의 경우 게임의 컨셉인 크리스마스를 연상케 하려 했습니다. 수업시간에 배운 Flickering Label을 응용하여 GameObject를 상속 받는 하나의 오브젝트로써 구현했습니다. 이들의 배치와 크기, 주기는 범위 내 랜덤으로 제한했습니다. 또한 이 Flickering Star들은 Vector에 저장시켜서 GameObject에서 Override한 draw() 메소드를 StartScene의 쓰레드에서 수행합니다. 모든 텍스트에 그림자 효과를 부여했고, Space Bar를 누르라 안내하는 텍스트는 주기에 따라 깜빡입니다. 이 씬에서 Space Bar를 누르면 GameScene으로 넘어가게 됩니다. 이번에 개발한 게임은 KeyListener 인터페이스를 사용하지 않고, 코드의 간결성을 위해 KeyAdapter를 사용합니다.

주기에 따라 깜빡이는 Space Bar 문구는 StartScene의 run() 메소드에서 elapsed라는 변수를 이용해 핸들링합니다. 굳이 elapsed를 사용한 이유는 텍스트가 보이는 시간과 보이지 않는 시간이 동일하게 되면 텍스트의 가독성이 떨어진다고 생각했기 때문입니다. 다음은 그 코드입니다.

```
@Override
public void run() {
    int elapsed = 0;
    while (true) {
        try {
            repaint();
```

```
            Thread.sleep(10);
            elapsed += 10;

            if (elapsed > 700 && elapsed < 1000) {
                spaceTextVisible = false;
            }
            else if (elapsed >= 1000) {
                spaceTextVisible = true;
                elapsed = 0;
            }
        }
        catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Elapsed로 핸들링하기 위해 별도의 Boolean 변수를 선언하여 사용했습니다./

다음은 GameOverScene입니다. GameOverScene도 StartScene의 기능과 동일하게 구현되어 있습니다. 하지만 다른 점은 플레이어의 점수와 최고 기록을 보여주는 것입니다. 이를 구현하기 위해 DataManager라는 별도의 클래스를 생성해서 구현했습니다. DataManager의 점수를 저장하는 Vector를 두고, 최대값과 마지막에 저장된 값을 반환하는 함수를 이용했습니다. DataManager에 대한 자세한 설명은 관리용 클래스에 대해 설명할 때 보충하겠습니다.

추가적으로 디자인을 위해 눈이 내리는 이펙트를 주었습니다. 이 FallSnow 클래스로 FlickeringStar 클래스와 동일하게 GameObject를 상속 받습니다. 이들은 y값은 0으로 고정되고, x값과 y축으로 이동하는 속도, update할 때 필요한 delta Time 값, 반지름을 범위 내 랜덤으로 갖게 됩니다. 이들은 0.5초 간격으로 20개 씩 랜덤으로 생성되고 Vector에 저장됩니다. 또한 바닥에 내려 앉는 것을 판정하여, Vector에서 삭제합니다. 하지만 이를 구현할 때 저장, 삭제 작업을 반복하다보니 iterator 관련한 예외가 자주 발생하여, 이를 방지하기 위해 추가한 객체들은 toAddSnow라는 별도의 Vector에 저장했습니다. 후에 이들을 적절한 타이밍에 본 Vector에 추가하는 방식으로 알고리즘을 구성했습니다. 이 적절한 타이밍은 그려지는 시점 바로 직전으로 선정했습니다.

```
snows.addAll(toAddSnow);
toAddSnow.clear();
if(toAddSnow.isEmpty()) {
    var it = snows.iterator();
    while (it.hasNext()) {
        it.next().draw(g2);
    }
}
```

GameOverScene의 paintComponent 메소드 내에 작성된 코드 중 일부입니다.

이번에는 GameScene를 살펴보겠습니다. 이 씬에서의 모든 객체들은 objects라는 Vector에 추가됩니다. 또한 GameOverScene에서 FallSnow 객체들을 담을 때 사용했던 로직과 마찬가지로, toAddObject라는 Buff Vector를 두어 iterator 순회에 Exception이 생기는 것을 최대한 방지하고자 했습니다. 다른 주요한 구성요소를 더 보자면, 사용자가 직접적으로 조종하는 Player, 깨뜨려야하

는 블럭인 Block과 이들이 저장되는 BlockPanel이 있습니다. 또 GameObject를 상속받은 공의 이탈을 막아주는 Wall과 공인 Ball이 있습니다.

이 씬에서는 좌우 방향키 입력을 통해 플레이어를 이동시킵니다. 다른 씬들과 마찬가지로 KeyListener가 아닌 KeyAdapter를 사용하여 구현했습니다. 또한 게임을 초기화하는 initGame()과 reset() 메소드를 만들어서, 현재 스테이지에 맞는 블럭을 생성하고, 공이 존재한다면 삭제하고, 플레이어의 위치를 원상 복구 시키기 등의 게임 진행을 위한 작업을 수행합니다. 더불어 관리용 클래스들의 변수를 초기화하거나 변경하는 작업도 병행합니다.

추가적인 사항으로 사용자의 경험성을 증대시키기 위해 현재 점수와 현재 최고 점수를 넘었는지 확인할 수 있는 Text를 화면 하단에 배치합니다. 만일 현재 점수가 최고 점수라면 점수 텍스트 위에 "Best Score"라는 문구를 띄웁니다. 이에 대한 구현은 DataManager의 도움을 받습니다. 또 스테이지를 클리어하게 되면 2.5초간 게임이 정지되며 "Stage N Clear!"라는 문구가 화면 가운데에서 등장합니다. 이 문구는 boolean 변수를 toggle하는 방식을 이용해서 0.5초 간격으로 깜빡입니다.

이번에는 게임을 관리하는 Management 클래스를 보겠습니다. 먼저 모든 게임 내용을 관장하는 GameManager 클래스입니다. 이 클래스는 Singleton으로 설계되어 항상 존재하고, 하나만 존재하고, 어디서든 접근 가능합니다. 현재 씬을 번호로써 관리하는데 sceneNumber라는 변수를 이용해 0일 때는 StartScene, 1일 때는 GameScene, 2일 때는 GameOverScene을 뜻합니다. 이를 통해 JFrame을 상속받는 Main 클래스의 setScene() 메소드를 통해 씬 전환을 이루게 됩니다. 그리고 현재 존재하는 공의 숫자인 ballCount, 블럭의 남은 개수인 blockNum, 현재 점수인 score, 스테이지 레벨인 level과 같은 정수형 변수들이 존재하고, JFrame을 저장하는 frame, 현재 인 게임 상태인지 판단하는 isInGame 변수 등이 존재합니다. 보시면 알다시피 게임에서 핵심 요소들을 다루는 것을 알 수 있습니다. 메소드 같은 경우도 객체지향적인 프로그램을 설계하기 위해 getter와 setter를 구성했고, 게임 오버의 판정을 내리는 gameOver(), 스테이지 클리어의 판정을 내리는 stageClear()가 존재합니다.

주요 메소드인 gameOver()를 살펴보면, 인 게임 상황에서 공의 개수가 0이하로 떨어지면 DataManager에 현재 점수를 저장하고 true를 반환하는 방식으로 동작합니다. 또한 다른 변수들에 대한 초기화도 병행되어 수행됩니다. stageClear()같은 경우 인 게임인 상황에서 블럭의 숫자가 0이하로 떨어지면 레벨을 증가시키고 true를 반환하는 방식으로 작동합니다. 다음은 그 코드입니다.

```
public boolean gameOver() {
    if (isInGame) {
        if (ballCount <= 0) {
            setInGame(false);
            level = 1;
            DataManager.addScore(score);
```

```
            score = 0;
            return true;
        }
    }
    return false;
}

public boolean stageClear() {
    if (isInGame) {
        if (blockNum <= 0) {
            setInGame(false);
            level++;
            return true;
        }
    }
    return false;
}
```

이번에는 DataManager입니다. 이 클래스는 Static 클래스로써 사용됩니다. 모든 변수와 메소드가 static으로써, 마찬가지로 어디서든 접근할 수 있습니다. 프로그램이 실행되는 동안 Vector에 점수를 저장하며, 주요 메소드로는 최고 점수를 알 수 있는 getMaxScore(), 가장 최근 점수를 확인할 수 있는 getLastScore()가 있습니다. 가장 최근 점수를 반환하는 메소드를 만든 이유는 게임 오버 될 시 마지막 점수를 Vector에 추가하고 현재 점수를 초기화하기 때문입니다.

관리용 클래스의 마지막은 AudioManager입니다. 이 클래스도 GameManager처럼 Singleton으로 설계되어 기민하게 사용할 수 있습니다. 또한 enum 자료형을 사용해서 이름만으로 직관적이게 사용할 수 있게 설계했습니다. 모든 음악 파일은 URL 배열에 저장되고, 이후 Clip 배열에 등록해 두었습니다.

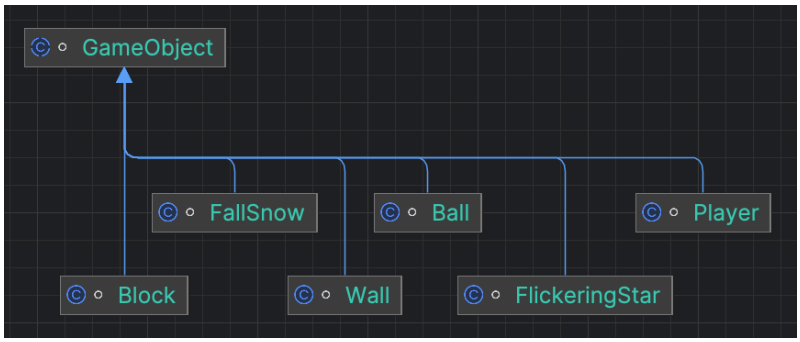
주요 메소드로는 Background Music을 재생하는 startBGM()과 SoundEffect를 재생하는 startSFX()로 구성되었습니다. 두 메소드 모두 필요한 곳에서 트랙의 이름만 넣으면 사용할 수 있습니다. startBGM() 같은 경우 모든 음악을 정지시키고, 현재 선택된 음악의 framePosition을 0으로 초기화 시킨 후 loop으로 재생하도록 설계했습니다. startSFX()는 현재 재생되고 있지 않은 모든 음원의 framePosition을 0으로 초기화시키고, 현재 선택된 음원을 재생하도록 작동합니다. 아래는 그 코드입니다.

```
public void startBGM(musicList track) {
    for (Clip clip : clips) {
        clip.stop();
    }

    int tmp = track.ordinal();
    clips[tmp].setFramePosition(0); // 위치 초기화
    clips[tmp].start();

    if (track != musicList.GameOver) {
        clips[tmp].loop(Clip.LOOP_CONTINUOUSLY);
    }
}
```

```
public void startSFX(musicList track) {
    for (Clip clip : clips) {
        if (!clip.isRunning()) {
            clip.setFramePosition(0);
        }
    }
    int tmp = track.ordinal();
    clips[tmp].start();
}
```

이러한 방식을 통해 필요한 곳에서 적절한 음악이나 효과음을 주어 더 실감하는 게임을 만들 수 있습니다.

이번에는 GameObject를 상속받는 클래스들에 대해 알아보겠습니다. 먼저 코드가 상대적으로 긴 Ball과 Player를 제외한 클래스들은 따로 GameObject 클래스 파일 아래에 묶어 두었습니다. 아래는 GameObject의 상속관계를 보여주는 Diagram입니다.



보시다시피 게임에서 사용되는 Ball, Wall, Player, Block과 장식용으로 사용되는 FallSnow, FlickeringStar가 모두 추상 클래스인 GameObject의 자식 클래스임을 확인할 수 있습니다.

가장 먼저 벽의 역할을 하는 Wall 클래스는 세로인지, 가로인지를 isHorizontal이라는 Boolean 변수를 통해 정의합니다. 또한 칠해지는 과정은 Graphics2D의 GradientPaint를 이용해서 가로 벽과 세로 벽이 이어지는 듯한 효과를 주고자 했습니다.

다음은 Player입니다. 사용자가 직접 조작하는 플레이어는 GameScene으로 들어온 입력을 move() 함수를 통해 처리합니다. 이 메소드의 인자로 key 값이 들어오게 되고 이를 판별하고 update를 핸들링하는 변수인 isLeft와 isRight의 값을 변경합니다. 또한 벽의 범위를 벗어나지 못하도록 하는 제한도 이 메소드 내에서 수행됩니다. Update() 메소드는 GameObject 클래스의 메소드를 Overriding한 형태로, isLeft와 isRight을 판별해서 x에 VX * dt를 더해주거나 빼줍니다. 또한 만일 벽과 충돌이 발생할 것 같다면 저장해두었던 prev_x를 통해 x값을 되돌려 줍니다. 디자인적인 부분은 그림자가 지는 효과를 연출하도록 했고, 모양도 RoundRect로 생성했습니다.

다음은 Block입니다. 이 블럭 클래스에서 이미지 사용과 차별점을 주고자 했습니다. 먼저 블럭은 총 3 종류로 blockType을 통해 관리합니다. 이 blockType은 Math.Random()을 통해 정해진 확률

0 ~ 0.65 (0 : 기본 블럭), 0.65 ~ 0.9 (1 : 하드 블럭), else (1 : 스페셜 블럭)로 정해집니다. 각각의 이미지 또한 다르게 구현되었고, 특성또한 다릅니다. 먼저 기본 블럭은 눈모양을 하고 있고, 특성은 따로 존재하지 않습니다. 스페셜 블럭인 경우에는 문제의 조건에 있었던 것처럼 부수게 되면 공이 3개로 분할되어 날아오게 되는 기능을 가지고 있습니다. 디자인으로는 깜빡거리며 시선을 끄는 효과를 부여했습니다. 마지막 하드 블럭의 경우에는 hp가 존재해서 3번을 때려야 블럭이 부서지도록 했습니다. 또한 투명도가 변경되면서 이 블럭의 체력이 얼마나 남았는지 확인할 수 있습니다. 아래는 각각의 블럭의 타입의 draw() 메소드의 일부입니다.

```
g2.setColor(Color.DARK_GRAY);
g2.fillRect((int) x + 2, (int) y + 2, (int) (w), (int) (h));

if (blockType == 0) {
    g2.drawImage(img, (int) x, (int) y, (int) w, (int) h, null);
}
else if (blockType == 1) {
    g2.drawImage(img, (int) x, (int) y, (int) w, (int) h, null);

    Composite originalComposite = g2.getComposite();

    g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, alpha));
    g2.setColor(Color.WHITE);
    g2.fillRect((int) x, (int) y, (int) w, (int) h);

    g2.setComposite(originalComposite);
}
else {
    Composite originalComposite = g2.getComposite();

    g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, hp / 3f));
    g2.drawImage(img, (int) x, (int) y, (int) w, (int) h, null);

    g2.setComposite(originalComposite);
}
```

모든 블럭이 칠해지기전 먼저 그림자 효과를 주기 위해서 DarkGray 색으로 칠합니다. 이후에 각각의 blockType에 맞는 형태로 이미지를 칠합니다. 스페셜 블럭의 alpha값은 run() 메소드에서 변화됩니다.

가장 중요한 클래스인 Ball 클래스입니다. 초반에 설계 방향과는 다르게 구현의 용이성에 의해 모든 충돌 처리는 이 Ball 클래스에서 작동합니다. 이 충돌 해결을 수행하는 collisionResolution() 메소드는 Wall, Player, Block과의 충돌을 각각 처리합니다.

먼저 벽과 충돌했을 때, 벽의 네 변의 위치와 좌표를 알아내고 만일 그 내부에 들어갔다면 밖으로 다시 밀어내는 형식의 코드를 사용합니다. 이는 수업 때 다룬 내용이기에 생략하겠습니다. 다음은 Player와 충돌할 때 처리입니다. 플레이어와 충돌할 때 만일 계속 반대 방향으로만 진행하게 된다면, 공을 의도한 대로 움직이지 못할 것입니다. 이를 해결하기 위해 플레이어의 중앙과 왼쪽 끝, 오른쪽 끝을 맞을 때 각각 다르게 튕겨내도록 해야 할 것입니다. 이를 통해 공이 맞은 위치와 가운데, 오른쪽, 왼쪽의 거리를 측정해서 먼저 가운데에 가장 가깝게 맞은 경우 상향으로 튕겨나

가도록 합니다. 우측에 가깝게 맞은 경우 우상향으로 튕겨나가게 하고, 좌측의 경우 좌상향으로 튕겨나가게 합니다. 현재 속도 벡터의 원소의 양음 판별이 안되므로 절댓값을 이용합니다. 이후에 맞았던 자리를 왼쪽 -1 ~ 오른쪽 +1의 형태로 변환하여 속도 벡터를 회전시킵니다. 이는 벡터에 회전 행렬을 곱하는 형태로 연산됩니다. 회전각도는 -25 ~ +25도로 제한시켰습니다. 이를 적용한 후에 원래의 속력을 유지하기 위해 비율을 scale의 형태로 곱해줍니다. 다음은 그 코드입니다.

```
// 충돌 판정
if (x > left && x < right && y > top && y < bottom) {
    float overlapLeft = Math.abs(x - left);
    float overlapRight = Math.abs(x - right);
    float overlapTop = Math.abs(y - top);

    //   기존 속력 저장
    float originalScalar = (float) Math.sqrt(vx * vx + vy * vy);

    if (overlapTop < overlapLeft && overlapTop < overlapRight) {
        //  위쪽 충돌
        y = top - r;
        vy = -Math.abs(vy);
        isCollision = true;
    } else if (overlapLeft < overlapRight) {
        //  왼쪽 충돌
        x = left - r;
        vx = -Math.abs(vx);
        vy = -Math.abs(vy);
        isCollision = true;
    } else {
        //  오른쪽 충돌
        x = right + r;
        vx = Math.abs(vx);
        vy = -Math.abs(vy);
        isCollision = true;
    }

    if (isCollision) {
        float relativeX = (x - p.x) / (p.w / 2); // -1 (왼쪽) ~ 1 (오른쪽)

        float angleAdjustment = relativeX * (float) Math.toRadians(25);

        //  속도 벡터 회전 (행렬 곱)
        float newVx = (float) (vx * Math.cos(angleAdjustment) - vy *
Math.sin(angleAdjustment));
        float newVy = (float) (vx * Math.sin(angleAdjustment) + vy *
Math.cos(angleAdjustment));

        vx = newVx;
        vy = newVy;

        //  원래 속력 유지
        float newScalar = (float) Math.sqrt(vx * vx + vy * vy);

        float scale = originalScalar / newScalar;
        vx *= scale;
        vy *= scale;
    }
}
```

마지막으로 블럭과 충돌했을 시를 보겠습니다. 기본적인 충돌은 Wall과 동일합니다. 하지만 블럭에 대한 효과들을 처리해야 합니다. blockType이 0인 경우 점수를 20점 더해줍니다. 1인 경우 duplicateBall() 메소드를 실행하고 점수를 20점 더해줍니다. 2인 경우 블럭의 hp를 1 감소시키고

점수를 10점 더해줍니다. 만일 hp가 0이하라면 블럭이 사라지게 합니다.

공을 3개로 늘려주는 메소드인 duplicateBall()은 속력을 동일하게 유지하고 퍼져나가는 효과를 주는 공 2개를 추가로 생성해야 합니다. 그러면 현재 공의 각도와 시작 각도를 정하고, 추가되는 공의 개수 만큼 반복문을 순회하면 됩니다. 아래는 그 코드입니다.

```
void duplicateBall() {

    AudioManager.getInstance().startSFX(AudioManager.musicList.Duplicate);

    float baseAngle = (float) Math.toDegrees(Math.atan2(vy, vx));
    float dAngle = spreadAngle / (bullets - 1);

    float startAngle = baseAngle - spreadAngle;

    for (int i = 0; i < bullets - 1; i++) {
        float curAngle = startAngle + dAngle * i;
        float radianAngle = (float) Math.toRadians(curAngle);

        Ball newBall = new Ball();

        //  속력 동일하게 해줘야 함
        newBall.vx = (float) (240 * Math.cos(radianAngle));
        newBall.vy = (float) (240 * Math.sin(radianAngle));
        newBall.x = x;
        newBall.y = y;

        GameManager.getInstance().frame.gameScene.toAddObjects.add(newBall);
    }

    GameManager.getInstance().ballCount+=2;
}
```

c) 시행 착오 및 느낀 점

이번 과제는 제가 흥미를 가지고 있는 게임 제작이었기에, 나름 흥미를 가지고 접근하였습니다. 하지만 Swing을 이용해서 게임을 만드는 것이 생각보다 더 원초적인 부분이 많았기에 마치 C언어로 테트리스를 개발하는 것과 같이 많은 시행 착오가 있었습니다. 그 중 기억에 남는 부분들에 대해서 언급하겠습니다.

첫 번째, 오디오 관련입니다. 충돌이라던지 다양한 효과음을 각각의 객체가 분담하여 발생시키면 가장 일관성있는 방식이 될 것이라고 생각했습니다. 하지만 공이 많아져서 효과음이 각각 발생하게 되면 소음에 가깝게 소리가 들려 차라리 재생 시간이 짧은 효과음을 가능할 때, 한 곳에서만 발생시키자는 생각을 했습니다. 이로써 오디오를 전체적으로 관리하는 AudioManager를 생성했습니다.

두 번째, 플레이어와 공의 충돌 판정이었습니다. 사실 처음에는 벡터를 회전시킨다는 생각을 못하고 있었습니다. 하지만 지속된 테스트 끝에 공이 의도한 방향으로 가면 어떨까라는 생각을 하

였고, 벡터의 회전을 배웠던 것이 생각나서 그대로 구현해보려 했습니다. 하지만 꺾어지는 양을 정량적으로 정할 수 없어 이 부분에 대해 수많은 테스트를 해야 했습니다. 그나마 최적으로 찾아낸 값을 25도로 판단했는데, 조금 더 좋은 방법이 있으면 그 부분에 대해 공부해보고 싶다는 생각도 했습니다.

세 번째, 플레이어의 움직임입니다. 플레이어가 좌우로 움직이는 것을 기존에는 vx 값을 이용해서만 핸들링 했었습니다. 그러나 방향전환을 할 때 딜레이가 발생했고, 이를 해결하고자 isLeft, isRight 변수를 도입해서 속력이 0이 되는 지점이 존재하지 않도록 하여 이 문제를 해결했습니다.

네 번째, 개발 환경 문제입니다. 제가 개발을 한 환경은 Mac에서 인텔리제이를 통해 개발했습니다. 프레임의 크기는 800*800으로 설정해두었습니다. 이후 추출한 jar 파일을 윈도우에서 가동해보니 우측이 잘리는 현상이 발생했습니다. 이를 해결하기 위해 웹서핑, GPT 등을 이용해서 마구 찾아보았지만, 명쾌한 답은 없었고 DPI 스케일링 처리 차이로 인해 GUI가 어긋나는 현상이 발생할 수 있다는 것뿐이었습니다. 따라서 보통 윈도우로 채점하실 것 같다는 생각에 윈도우 용으로 Frame을 재설정하였습니다.

2학년 2학기. 가장 배우고 싶었고, 알아야 한다는 Java라는 언어와 객체지향에 대해 깊게 배울 수 있는 시간이었습니다. 더군다나 Swing과 같은 GUI 프레임워크로 게임을 만들어보는 경험은 객체지향을 몸에 익힐 수 있었습니다. C++부터 Java로 이어지는 수많은 지식들은 후에 제 프로그래밍 역량과 수학적 코딩 실력을 크게 향상 시켜줄 것이라고 생각합니다. 좋은 과제로 한 학기를 마무리 함과 항상 귀중한 수업에 감사드립니다.

이상입니다.