자료구조 실습 보고서

[제 1주] : 마방진

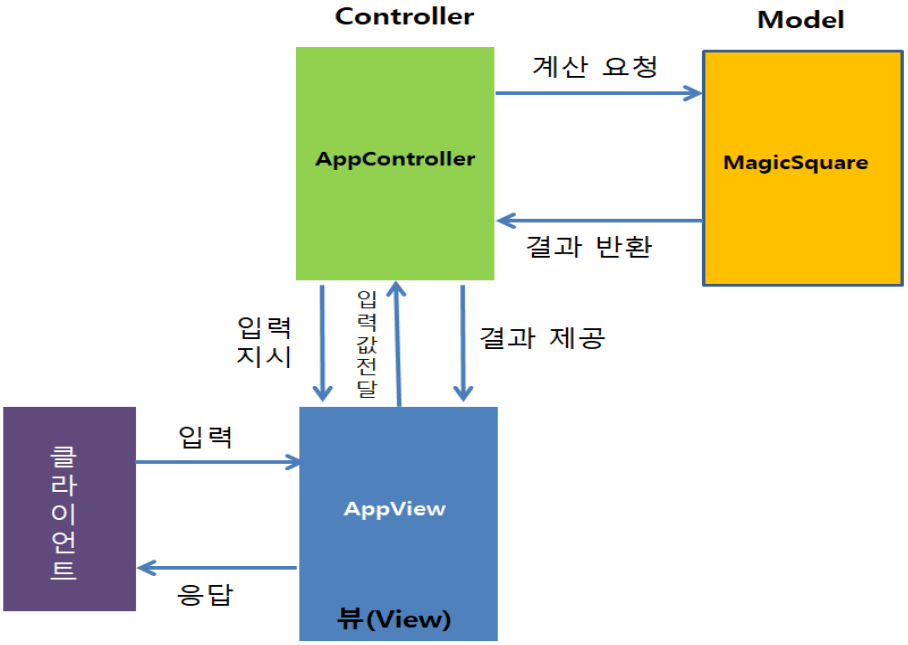
제출일: 2022-03-04(금)

201902708 신동훈

프로그램 설명

1. 프로그램의 전체 설계 구조

간단하게 표현하면 다음과 같다.



MVC(Model - View - Controller)구조를 따르며, 각각의 역할은 다음과 같다.

AppController

AppView에게 입력 요청을 보내며, 그에 대한 응답으로 반환된 입력값을 받아온다.
MagicSquare에게 마방진 계산을 요청하며, 그에 대한 응답으로 반환된 마방진의 결과를 받아온다.
AppView에게 출력 정보를 제공하며, 출력을 요청한다.

AppView

화면(콘솔)에 메시지를 출력하는 역할과, 화면으로부터 입력값을 받아와 다른 객체에게 전달하는 역할을 담당한다.

MagicSquare

마방진을 계산하고, 계산된 마방진을 반환하는 역할을 수행한다.

2. 함수 설명(주요 알고리즘/ 자료구조의 설명 포함)

마방진을 구하는데 사용되는 자료구조는 int형의 2차원 배열을 사용하며, 알고리즘은 다음과 같다.

1. 마방진을 구할 때 시작 위치는, 마방진의 맨 윗 줄의 한 가운데, 즉 2차원 배열로 표현하자면 [0][차수/2]이다.
2. 마방진은 항상 오른쪽 위칸을 채우려 시도한다. 즉 현재 마지막으로 채워진 마방진의 row와 column을 각각 row, col이라 한다면, 오른쪽 윗 칸은 col+1(오른쪽), row-1(위)이 된다.
2-1. 이때 만약 col+1이 마방진의 크기를 벗어난다면 맨 왼쪽으로 이동해야 하기에, 다음 col의 인덱스는 0이 된다.
2-2. row - 1도 마찬가지로 마방진의 크기를 벗어난다면, 마방진의 맨 아래로 이동해야 하기에 row의 인덱스는 (마방진의 차수 -1)이 된다.
3. 계속해서 2번의 과정을 반복하여 마방진을 채워나간다. 그러나 위 과정만 반복하다 보면 마방진의 우측 위가 이미 채워져 있는 경우가 발생한다. 이 경우에는 오른쪽 위가 아닌 바로 한 칸 아래(row + 1)를 채우도록 한다.
3-1. 이때도 마찬가지로 row + 1이 마방진의 범위를 벗어나는 경우, 맨 위로 이동해야 하기에 row의 인덱스는 0이 된다.

위의 2~3 과정을 반복하면 마방진이 완성된다.
아래는 마방진을 구하는 함수의 구현이다. 주석을 통해 위의 과정을 자세히 설명해두었다.

MagicSquare의 solve() : 마방진을 계산하는 함수이다.

```
public Board solve(int anOrder) {//차수를 받아온다.
        if ( OrderValidity.validityOf(anOrder) != OrderValidity.Valid ) {
        //차수(Order)가 유효하지 않다면 계산하지 않는다.
             return null ;
         }
```

```
else {
    Board board = new Board (anOrder) ;
   // 차수와 함께 Board 객체 생성자를 call 하여, Board 객체를 생성한다.

    CellLocation currentLoc = new CellLocation (0, anOrder/2) ;
   // 출발위치(보드의 맨윗줄 한가운데)를 현재의 위치로 설정한다.

    CellLocation nextLoc = new CellLocation();
   // 다음위치를 설정할 CellLocation 객체를 생성해둔다.

    board.setCellValue(currentLoc, 1) ;// 보드의<출발위치>에1 을채운다.

    int lastValue = anOrder * anOrder ;
    //차수 * 차수 = 마방진의 전체 칸의 수 = lastValue

    for ( int cellValue = 2 ; cellValue <= lastValue ; cellValue++ ) {

        // 단계1: <현재위치>로부터<다음위치>인"오른쪽위" 위치를계산한다
        final int currentRow = currentLoc.row();//현재 Row를 받아온다
        final int currentCol = currentLoc.col();//현재 Col을 받아온다

        /*
          row 는 위칸으로 갈수록 인덱스가 작아진다.
          즉 위로 한 칸 이동하기 위해서는 row - 1을 해주어야 한다.
          currentRow - 1 이 -1인 경우에는, 범위를 초과하였으므로
          마방진의 맨 아래 (anOrder - 1)로 이동해야 한다.
         */
        int nextRow = (currentRow - 1 == -1) ? anOrder -1 : currentRow - 1;

        /*
          col 은 오른쪽으로 이동하기 위해서는 인덱스를 키워주어야 한다.
          즉 오른쪽으로 한 칸 이동하기 위해서는 col + 1을 해주어야 한다.
          currentCol + 1 이 anOrder 인 경우에는, 범위를 초과하였으므로
          마방진의 맨 왼쪽(0)으로 이동해야 한다.
         */
        int nextCol = (currentCol + 1 == anOrder) ? 0 : currentCol + 1;


        nextLoc.setRow(nextRow);//<다음위치>의 Row 를 설정한다.
        nextLoc.setCol(nextCol);//<다음위치>의 Col 을 설정한다.
```

```
            // 단계2: <다음위치> 가채워져있으면
            // <다음위치>를 <현재위치>의 바로 한 줄 아래칸 위치로 수정한다.
            if( ! board.cellIsEmpty (nextLoc) ) {


                /*
                  row 는 아래칸으로 갈수록 인덱스가 커진다.
                  즉 아래로 한 칸 이동하기 위해서는 row + 1을 해주어야 한다.
                  currentRow + 1 이 anOrder 인 경우에는, 범위를 초과하였으므로
                  마방진의 맨 위래(0)로 이동해야 한다.
                */
                nextRow = (currentRow + 1 == anOrder) ? 0 : currentRow + 1;


                /*
                바로 한 줄 아래칸 위치이므로 Col의 변화는 없다.
                 */
                nextCol = currentCol;



                nextLoc.setRow(nextRow);//<다음위치>의 Row 를 설정한다.
                nextLoc.setCol(nextCol);//<다음위치>의 Col 을 설정한다.
            }


            // 단계3: <다음위치>를새로운<현재위치>로한다.
            currentLoc.setRow (nextLoc.row()) ;
            currentLoc.setCol (nextLoc.col()) ;


            // 단계4: 새로운<현재위치>에 cellValue 값을넣는다.
            board.setCellValue (currentLoc, cellValue) ;
        }
        return board ;
    }
}
```

3. 종합 설명

해당 프로그램은 정말 단순히 사용자로부터 숫차(차수) 하나를 입력받은 후, 해당 숫자를 차수로 하는 마방진을 출력하는 프로그램이다.
앱을 실행하기 위해서는 '_DS01_Main_201902708_신동훈' 이라는 이름의 클래스에 속한 main 메서드를 실행시켜주면 된다.

사용자는 원하는 마방진의 차수를 3~99 사이의 홀수 중에서 입력을 할 수 있으며, 올바른 값을 입력했다면 마방진을 반환하고, 그렇지 않다면 다시 입력하라는 메시지가 출력된다.
main 메서드를 실행하면 AppController의 run 메서드를 통해서 프로그램이 제어된다.
AppController는 run이 실행되면 먼저 AppView에게 "마방진 풀이를 시작합니다"라는 메시지를 출력하도록 요청하고, 그 이후 AppView의 inputOrder() 메서드를 통해 사용자로부터 차수를 입력받아 그 값을 반환해달라는 요청을 한다.

이후 AppController에서는 해당 요청받은 값이 음수인지 음수가 아닌지 판단한 후, 음수가 아닌 경우 3~99 사이의 홀수인지 확인한다. 올바른 값이 입력되었을 경우 프로그램의 흐름은 다음과 같다.

AppController에서 MagicSquare의 solve 메서드를 호출하며, 인자로는 사용자로부터 입력받은 마방진의 차수를 전달하여 해당 차수의 마방진 풀이를 요청한다.
MagicSquare는 solve메서드가 호출되면 위에서 살펴본 알고리즘을 따라 마방진을 구한 후, 해당 마방진을 Board 타입의 객체로 AppController에게 반환한다.

AppController는 전달받은 Board타입의 객체를 적절한 형태로 출력하도록 row와 column을 순차적으로 탐색하며, 해당 row와 column의 값을 AppView에게 전달하여 화면에 출력하도록 요청한다.

이후 다시 사용자의 입력을 받아오도록 AppView에게 요청하며 위의 과정을 반복한다.

프로그램 장단점/특이점

1. 프로그램의 장점

‘유지보수성과 재사용성이 용이.’

마방진을 구하여 사용자에게 반환한다는 목표를 이루기 위해, 적절한 책임을 가진 객체들을 정의하였고, 각각의 객체들은 자신에 책임을 다하기 위해 필요한 행동들을 수행하도록 구성하였다. 예를 들면 MagicSquare은 주어진 차수의 마방진을 구하는 solve라는 행동을 가지고 있는 식이다.

위와 같이 프로그램은 객체지향적으로 작성되었기에 객체지향 프로그래밍의 장점(대표적으로 유지보수성의 증가 등)을 모두 갖는다.

또한 MVC패턴을 적용하여, 화면에 출력되는 부분의 수정이 필요할 경우 View만, 비즈니스 로직에 변화가 생긴다면 Model만 변경할 수 있게끔 변경으로 인한 Side Effect를 최소한의 범위로 줄였기에, 이 또한 프로그램의 유지보수성을 증가시킨다.

2 프로그램의 단점

마방진을 구하는 프로그램이지만 마방진의 차수는 3~99사이의 홀수로 제한된다. 99를 초과하는 더 큰 값이나, 짝수에 대해서는 마방진을 구할 수 없다.

실행 결과 분석

1. 입력과 출력

1-1. 잘못된 입력 - 음수를 입력한 경우

```
<<< 마방진풀이를시작합니다>>>

? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다): -1

<<< 마방진풀이를종료합니다>>>
```

예시로는 -1만 입력하였으나, 다른 음수를 입력해도 정상적으로 종료된다.

1-2. 잘못된 입력 - 짝수를 입력한 경우

```
<<< 마방진풀이를시작합니다>>>

? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다): 4
[오류]차수가 짝수입니다. 홀수이어야 합니다.
? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다): 80
[오류]차수가 짝수입니다. 홀수이어야 합니다.
? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다):
```

3~99 범위의 짝수를 입력한 경우 적절한 메시지와 함께 다시 값을 입력하도록 한다.

1-3. 잘못된 입력 - 범위를 벗어난 입력의 경우

```
? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다): 1
[오류]차수가 너무 작습니다. 3 보다 크거나 같아야 합니다.
? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다): 100
[오류] 차수가 너무 큽니다. 99 보다 작거나 같아야 합니다.
? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다):
```

3~99사이의 범위를 벗어난 경우 적절한 메시지와 함께 값을 다시 입력받도록 한다.

1-4. 적절한 입력

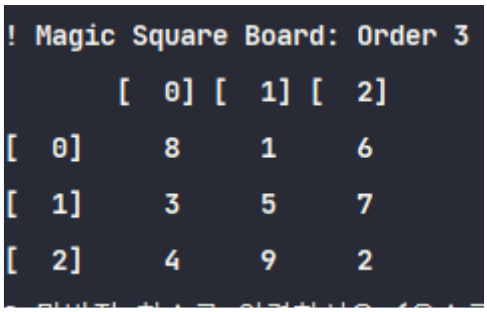
```
<<< 마방진풀이를시작합니다>>>

? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다): 3
! Magic Square Board: Order 3
       [  0] [  1] [  2]
[  0]     8     1     6
[  1]     3     5     7
[  2]     4     9     2
? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다): 5
! Magic Square Board: Order 5
       [  0] [  1] [  2] [  3] [  4]
[  0]    17    24     1     8    15
[  1]    23     5     7    14    16
[  2]     4     6    13    20    22
[  3]    10    12    19    21     3
[  4]    11    18    25     2     9
? 마방진 차수를 입력하시오 (음수를 입력하면 종료합니다):
```

적절한 입력의 경우, 입력한 값을 차수로 하는 마방진을 출력한다.

2. 결과 분석

차수가 3일 때 생성된 마방진은 아래와 같다.



‘프로그램 설명’의 2번‘함수 설명’에서 설명한 알고리즘대로 마방진이 생성되었다는 것을 알 수 있다.

생각해 볼 점에 대한 의견

1. Model - View - Controller 방식의 설계를 하는 이유

'역할의 분리'

MVC 방식을 적용하지 않은 프로그램에서 발생할 수 있는 문제점에 대해 서술해보도록 하겠다.
각각의 클래스들은 가져야 하는 역할이 정해져 있다. 객체지향설계의 5원칙(SOLID)의 맨 처음 S가 뜻하는 바가 단일 책임 원칙(Single Responsibility Principle)인 것처럼, 하나의 클래스는 하나의 책임만 가지는 것이 이상적이다.
그러나 MVC 방식을 적용하지 않고 위 프로그램을 작성한다면 하나의 클래스는 콘솔에 메시지를 출력하며, 사용자로부터 입력값을 받아와서, 내부적으로 마방진을 계산하는 로직을 수행한 후, 다시 그 결과를 직접 사용자에게 보여주어야 한다.
즉 하나의 클래스에 책임이 너무 많이 부여된 것이다.

하나의 클래스가 여러 책임을 갖게 된다면, 책임의 수만큼 변경되는 이유도 늘어나게 되고, 결국 이러한 코드는 유지보수하기 힘들어진다.

예를 들어 결과값을 콘솔에 표시하는 것이 아니라, 텍스트 파일로써 결과값을 반환해야 한다는 요구사항이 새로 생겼다고 생각하자.
해당 요구사항을 만족시키기 위한 새로운 클래스 파일을 생성하고, 해당 파일에 결과를 작성하도록 코드를 수정하여야 할 것이다.
짝수 차수의 마방진을 구할 수 있도록 알고리즘을 수정하려면 원래의 알고리즘을 구현한 부분을 변경하여야 할 것이다.
위와 같이 변경은 여러 가지 이유로 언제든지 발생할 수 있다.
따라서 우리는 유지보수를 편하게끔 프로그램을 작성하여야 한다.

그러나 위의 예시처럼 프로그램을 작성하게 된다면, 화면에 출력하는 방식을 수정하는 것과, 마방진을 구하는 알고리즘을 수정하는 일은 각각 다르게 발생할 가능성이 높으며, 이는 대부분 서로에게 영향을 끼치지 않는다. 즉 변경의 이유가 다른 것이며, 이러한 부분을 하나의 코드로 관리하는 것은 유지보수하기 좋지 않다.

MVC 방식을 사용한다면, 화면에 출력하는 기능을 담당하는 부분과 비즈니스 로직을 처리하는 부분을 분리하여 각각 자신이 맡은 일에만 충실하도록 만든다.
결과적으로 프로그램의 유지보수성을 증가시키는 이점을 얻을 수 있다.

2. Class "AppView"의 모든 함수가 "static"인 이유는?

"AppView"는 굳이 인스턴스화 시키지 않고, 화면 출력과 관련된 기능을 제공할 수 있도록 모든 필드와 함수를 static으로 설정하였다. 즉 “AppView”는 소위 말하는 “유틸리티 클래스”로써 화면에 관련된 기능을 담당하는 부분에 대한 기능을 인스턴스를 생성하지 않고 간편하게 제공하기 위해 설계된 것이다.
인스턴스화될 필요가 없으므로 생성자는 private으로 막아서 인스턴스화를 방지함과 동시에 상속도 불가능하게 만들어 하위클래스를 생성하여 인스턴스화되는 것도 방지하였다.

그러나 AppView 클래스의 모든 필드와 함수를 static으로 설정한 것은 바람직하지 않다고 생각한다.
AppView는 화면에 관련된 기능을 수행한다.
이 프로그램에서 출력하는 화면은 콘솔이며 출력하는 화면이 변하지 않는다는 확신이 있다면 위와 같이 모든 필드와 메서드를 static으로 설정하여 편하게 사용할 수 있도록 만드는 방법도 괜찮은 방법이다.
그러나 위와 같은 경우라면, MagicSquare등도 모든 필드와 메서드를 static으로 설정하여 사용할 수 있다.

요구사항이 콘솔이 아니라 파일로 출력하도록 바뀌거나, 출력하는 방식을 고를 수 있도록 요구사항이 바뀐다면 AppView는 재사용이 거의 불가능하다.
AppView에 구현되어있는 모든 함수를 다시 작성해야 하며, 심지어는 똑같이 View 기능을 담당하는 클래스를 새로 만들어야 할 수도 있다.
또한 AppView를 사용하는 클래스인 AppController도 자신의 역할과는 상관없는 이유인 ‘출력 화면의 변경’이라는 이유로 변경될 수 있다.

AppView의 모든 필드와 함수를 static으로 만든 시점에서, AppView는 객체지향적이지 못한 클래스라고 생각한다.
코드를 바꿀 수 있다면 객체지향의 장점인 다형성을 적극 이용할 수 있도록 AppView를 인터페이스로 선언한 후, 이를 구현하는 ConsoleAppView등의 구현체를 만들어 Console 화면에 출력하는 기능을 담당하도록 설계할 것이다.
이는 이후 출력되는 화면이 바뀌거나 하는 등의 상황에서, 다른 클래스에 영향을 끼치지 않고 자유롭게 변경될 수 있다.
추가적으로 AppController도 마찬가지로 인터페이스로 정의할 수 있으며, 마방진 풀이뿐만 아니라 스도쿠등의 다른 문제에 대한 결과도 얻을 수 있는 프로그램을 생각한다면 MagicBorad 또한 적절한 인터페이스를 생성하고 이를 구현하도록 만들어야 한다.

3. public과 private의 차이는?

객체지향적으로 잘 설계된 객체들은 그 객체가 어떻게 구현되었는지에는 상관없이. 그 객체가 외부에 노출하는 행동만 알고 있다면 쉽게 상호작용할 수 있다.
따라서 객체를 설계할 때에는 외부에 노출할 행동과, 내부에 숨겨지는 구현을 명확하게 분리하여야 한다.
카페에 가서 커피를 사마실 때 바리스타가 커피를 어떻게 만드는 지 알 필요도, 궁금하지도 않다.
단지 바리스타가 나에게 커피를 제공하기만 하면 그 과정은 아무런 상관이 없는 것이다.

객체들도 사람들처럼 원하는 역할을 수행하기만 한다면 어떤 식으로 역할을 수행하는지는 신경쓰지 않아야 한다.
이를 위한 방법이 바로 캡슐화이다.

캡슐화는 두 가지 관점으로 사용되는데, 하나는 '상태와 행위를 하나로 묶는 캡슐화'이며, 다른 하나는 '사적인 구현의 캡슐화'이다.

객체는 스스로 자신의 상태를 관리하며, 자신의 수행할 수 있는 행동을 내부에 보관한다.
이것이 상태와 행위를 하나로 묶는 관점에서의 캡슐화이다.

사람은 자신의 집 비밀번호가 외부에 노출되지 않도록 방어한다. 계정의 비밀번호를 설정하는 것도 마찬가지이다. 사람은 보안에 민감하며, 보안을 지키기 위해 최대한 노력한다.
객체도 마찬가지로 자신의 비밀이 노출되는 것이 민감하다. 다른 객체가 자신의 내부 상태를 변경하거나 관리하지 못하도록, 객체는 자신의 상태를 숨기고, 특정한 경로로만 외부에 노출한다.
만약 객체의 내부 상태를 다른 객체가 마음대로 바꿀 수 있는 상황이 발생한다면, 이는 발견하기 힘든 오류로 이어질 가능성이 있으며, 프로그램를 관리하기 어려워진다.

private는 객체가 자신의 상태를 숨길 때 사용하며, public은 다른 객체와 협력하기 위해 외부에 노출하는 특정한 행위에 적용한다.

이를 통해 객체는 자기 자신의 상태를 스스로 관리할 수 있게 되며, 또한 외부에 노출하는 행위를 변경하지 않는 이상, 자신의 내부 구현을 외부에 영향을 미치지 않고 자유롭게 수정할 수 있다.

이를 통해 설계가 유연해지며 유지보수와 변경에 대응하기 쉬워지는 이점이 있다.

4. 상수 선언에 static과 final이 붙는 이유는?

상수란 '변하지 않는 값'을 의미한다.

staitc은 '정적인'이라는 의미를 가지고 있는 키워드로, 변수, 메서드, 클래스 앞에 위치할 수 있다.
static이 붙으면 객체 생성 시점이 아닌, 클래스 로딩 시점에 메모리가 Heap 영역에 할당된다.
(Java 7 이전에는 Permgen 영역의 Method Area에 할당되었으나 Java8부터 Heap 영역에 할당되도록 바뀌었다.)
따라서 객체를 생성하지 않고도 해당 값을 사용할 수 있게 되는 것이다.

자바에서 final 키워드를 사용한다면, 한번 할당된 값을 바꿀 수 없게 된다. 사실 상수의 의미만 보자면 final만 사용해도 '변하지 않는 값'을 만들 수 있기에 상수로 사용할 수 있다.

그러나 static을 붙이지 않는다면 객체마다 다른 상수값을 사용하는 상황이 발생할수도 있다.
예를 들면 다음과 같다.

```
public class Test {
    private final int finalInt;
    public Test(int finalInt) {
        this.finalInt = finalInt;
    }
}
```

다음과 같은 경우 Test의 인스턴스는 각각 다른 finalInt 값을 가진 체 생성될 수 있다.
이를 방지하기 위해 상수에는 static을 붙여주어 클래스 로딩 시점에 단 한번만 초기화되어 이후 값을 수정할 수 없도록 만들어준다.

또한 클래스의 모든 인스턴스가 같은 값을 사용한다면 인스턴스가 생성될 때 마다 메모리를 할당하는 것 보다는 static을 사용하여 클래스 변수로 설정하여 그 값을 공유하게 함으로써 메모리의 낭비를 방지하는 효과 또한 얻을 수 있다.

정리하면 final만 붙은 변수는 생성자를 통해서 여러 값을 가질 수 있으며, 객체마다 저장되기에 완전한 상수라 볼 수 없다.
static final 변수는 모든 객체에서 같은 값을 참조하며, 선언과 동시에 초기화되기 때문에 완전한 상수라 볼 수 있다.

따라서 위와 같은 이유로 상수 선언에 static과 final이 붙는다.

5. 슬라이드 27쪽의, while문과 for문에서, 변수 “sum”과 "count" 각각의 scope는?

1. While문

```
int sum = 0 ;
{
    int count = 0 ;
    while ( count < size) {
        sum = sum + count ;
        count++
    }
}
```

위 코드에서 sum의 spoce는 int sum = 0;을 포함한 위 사진속 모든 영역에서 유효하다.
즉 sum이 선언된 위치를 포함하여, 선언을 포함하는 코드를 닫아주는 괄호“}”가 나오기 전(혹은 변수가 선언된 블록이 끝나기 전)까지가 sum의 scope이다. (사진 속 코드에는 닫아주는 괄호가 나오지 않았다.)

count는 count가 선언된 블록 내부에서만 사용 가능하며, 따라서 count의 scope는 다음과 같다.

```
int sum = 0 ;
{
    int count = 0 ;
    while ( count < size) {
        sum = sum + count ;
        count++
    }
}
```

2. for문

```
int  sum = 0 ;

for ( int  count = 0 ; count < size ; count++) {
      sum = sum + count ;

}
```

위 코드에서 변수 sum의 spoce는 int sum = 0;을 포함한 위 사진속 모든 영역에서 유효하다.
즉 sum이 선언된 위치를 포함하여, 선언을 포함하는 코드를 닫아주는 괄호"}"가 나오기 전(혹은 변수가 선언된 블록이 끝나기 전)까지가 sum의 scope이다. (사진 속 코드에는 닫아주는 괄호가 나오지 않았다.)

변수 count의 scope은 for문의 괄호"()" 속과 해당 for문의 블록(괄호 "{}" 내부)이다.