SWE 7조 그린화 패턴 보고서

차승일 김민수 유규환 장영우 이시혁 임소현 정단호

그린화 패턴 #1

Before

```
public static void main (String[] args) throws java.lang.Exception
{

    List<Integer> numbers = new ArrayList<>();
    for (int i = 1; i <= 10000000; i++) {
        numbers.add(i);
    }
}
```

After

```
public static void main (String[] args) throws java.lang.Exception
{

    List<Integer> numbers = new ArrayList<>(10000000);
    for (int i = 1; i <= 10000000; i++) {
        numbers.add(i);
    }
}
```

사용할 양 만큼 동적할당을 통해 최적화

그린화 패턴 #2

Before

```
for (int i = 0; i < 10; i++) {
    Wait_one_second.waiting(i);
}
```

After

```
ExecutorService executor = Executors.newFixedThreadPool(10);
for (int i = 0; i < 10; i++) {
    final int taskId = i;
    executor.submit(() -> Wait_one_second.waiting(taskId));
}
executor.shutdown();
```

Wait_one_second는 1초를 기다리는 class을 이용

그린화 패턴 #3

Before

```
class EagerLoader {
    private String data;
    public EagerLoader() {
        this.data = load();
    }
    private String load() {
        return "Eagerly loaded";
    }
    public String getData() {
        return data;
    }
}

class Ideone
{
    public static void main (String[] args) throws java.lang.Exception
    {
        EagerLoader res = new EagerLoader();
        String data = res.getData();
        System.out.println(data);
    }
}
```

After

```
class Lazyloader {
    private String data;

    public String getData() {
        if (data == null) {
            data = load();
        }
        return data;
    }
    private String load() {
        return "Lazy loaded";
    }
}

/* Name of the class has to be "Main" only if the class is public. */
class Ideone
{
    public static void main (String[] args) throws java.lang.Exception
    {
        Lazyloader resource = new Lazyloader();
        String data = resource.getData();
        System.out.println("Data: " + data);
    }
}
```

Lazy loading을 사용하면 느린 지연시간을 가지고 초기 로딩 시간이 감소, 메모리 소비량이 감소한다

그린화 패턴 #4

Before

```java
public class before {
    public static void main(String[] args) {
        for (int i = 0; i < 10000; i++) {
            double result = Math.pow(i, 2);
        }
    }
}
```

After

```java
public class after {
    public static void main(String[] args) {
        for (int i = 0; i < 10000; i++) {
            int result = i * i;
        }
    }
}
```

상대적으로 CPU 사용량이 큰 제곱 연산 대신 곱셈 연산 사용

그린화 패턴 #5

Before

```java
public class before {
    public static void main(String[] args) {
        Double sum = 0.0;
        for (int i = 0; i < 10000; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```

After

```java
public class after {
    public static void main(String[] args) {
        int sum = 0;
        for (int i = 0; i < 10000; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```

불필요한 데이터 타입 변경. 불필요하게 사용되는 메모리 사용량을 줄인다.

그린화 패턴 #6

Before

```java
public class before {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < 10000; i++) {
            list.add(i);
        }
        for (int i = 0; i < 10000; i++) {
            list.contains(i);
        }
    }
}
```

After

```java
public class after {
    public static void main(String[] args) {
        Set<Integer> set = new HashSet<>();
        for (int i = 0; i < 10000; i++) {
            set.add(i);
        }
        for (int i = 0; i < 10000; i++) {
            set.contains(i);
        }
    }
}
```

데이터 구조에 따른 비효율적인 메소드 수정.

ArrayList의 contains 메소드의 시간 복잡도 = O(n)
HashSet의 contains 메소드의 시간 복잡도 = O(1)

그린화 패턴 #7

Before

```java
public class pattern1b {
    public enum Score {APLUS, AMINUS, BPLUS, BMINUS, CPLUS, CMINUS, DPLUS, DMINUS,F};

    Run | Debug
    public static void main(String[] args) {
        Score cur_score = Score.DMINUS;

        if (cur_score == Score.APLUS) {
            System.out.println(x:"Your score is A+.");
        } else if (cur_score == Score.AMINUS) {
            System.out.println(x:"Your score is A-.");
        } else if (cur_score == Score.BPLUS) {
            System.out.println(x:"Your score is B+.");
        } else if (cur_score == Score.BMINUS) {
            System.out.println(x:"Your score is B-.");
        } else if (cur_score == Score.CPLUS) {
            System.out.println(x:"Your score is C+.");
        } else if (cur_score == Score.CMINUS) {
            System.out.println(x:"Your score is C-.");
        } else if (cur_score == Score.DPLUS) {
            System.out.println(x:"Your score is D+.");
        } else if (cur_score == Score.DMINUS) {
            System.out.println(x:"Your score is D-.");
        } else if (cur_score == Score.F) {
            System.out.println(x:"Your score is F.");
        } else {
            System.out.println(x:"No score.");
        }
    }
}
```

After

```java
public class pattern1a {
    public enum Score {APLUS, AMINUS, BPLUS, BMINUS, CPLUS, CMINUS, DPLUS, DMINUS,F};
    Run | Debug
    public static void main(String[] args) {
        Score cur_score = Score.DMINUS;

        switch (cur_score) {
            case APLUS:
                System.out.println(x:"Your score is A+.");
                break;
            case AMINUS:
                System.out.println(x:"Your score is A-.");
                break;
            case BPLUS:
                System.out.println(x:"Your score is B+.");
                break;
            case BMINUS:
                System.out.println(x:"Your score is B-.");
                break;
            case CPLUS:
                System.out.println(x:"Your score is C+.");
                break;
            case CMINUS:
                System.out.println(x:"Your score is C-.");
                break;
            case DPLUS:
                System.out.println(x:"Your score is D+.");
                break;
             case DMINUS:
                System.out.println(x:"Your score is D-.");
                break;
            case F:
                System.out.println(x:"Your score is F.");
                break;
            default:
                System.out.println(x:"No score.");
        }
    }
}
```

정수 혹은 열거형(enum) 처리시, switch 문 적용. switch 문에서는 jump table을 이용하여 즉각적으로 분기가 가능하기에 runtime 감소

그린화 패턴 #8

Before

```java
import java.util.Scanner;
public class pattern2a {
    /* function for inputting user's name */
    public static String inputName() {
        Scanner input = new Scanner(System.in);
        System.out.printf(format:"Your name: ");
        String userName = input.nextLine();
        return userName;
    }
    /* function for inputting user's initial balance */
    public static float inputBalance() {
        Scanner input = new Scanner(System.in);
        System.out.printf(format:"Initial Balance: ");
        float userBalance = input.nextFloat();
        return userBalance;
    }
    /* function for inputting percentage, meaning interest */
    public static float inputPercentage() {
        Scanner input = new Scanner(System.in);
        System.out.printf(format:"Percentage: ");
        float userPercentage = input.nextFloat();
        return userPercentage;
    }
    /* function for inputting number of years */
    public static int inputYears() {
        Scanner input = new Scanner(System.in);
        System.out.printf(format:"Number of years: ");
        int userYears = input.nextInt();
        return userYears;
    }

    Run | Debug
    public static void main(String[] args) {
        String userName = inputName(); //for storing the bank user's name
        float userBalance = inputBalance(); //for storing the bank user's balance
        float userPercentage = inputPercentage(); //for storing the percentage
        int userYears = inputYears(); // for storing the years
        System.out.format(format:"Name : %s, Balance : %.1f, Percentage : %.1f, Year : %d\n", userName, userBalance, userPercentage, userYears);
    }
}
```

After

```java
import java.util.Scanner;
public class pattern2b {
    /* function for inputting user's name */
    private static Scanner input = new Scanner(System.in);

    public static String inputName() {
        System.out.printf(format:"Your name: ");
        String userName = input.nextLine();
        return userName;
    }
    /* function for inputting user's initial balance */
    public static float inputBalance() {
        System.out.printf(format:"Initial Balance: ");
        float userBalance = input.nextFloat();
        return userBalance;
    }
    /* function for inputting percentage, meaning interest */
    public static float inputPercentage() {
        System.out.printf(format:"Percentage: ");
        float userPercentage = input.nextFloat();
        return userPercentage;
    }
    /* function for inputting number of years */
    public static int inputYears() {
        System.out.printf(format:"Number of years: ");
        int userYears = input.nextInt();
        return userYears;
    }

    Run | Debug
    public static void main(String[] args) {
        String userName = inputName(); //for storing the bank user's name
        float userBalance = inputBalance(); //for storing the bank user's balance
        float userPercentage = inputPercentage(); //for storing the percentage
        int userYears = inputYears(); // for storing the years
        System.out.format(format:"Name : %s, Balance : %.1f, Percentage : %.1f, Year : %d\n", userName, userBalance, userPercentage, userYears);
    }
}
```

한 class 내에서의 여러 함수에서의 동일한 객체 생성. 한 class의 멤버 함수들은 하나의 객체를 생성 후 공유 가능하므로 불필요한 메모리 낭비를 줄인다.

그린화 패턴 #9

Before

```java
import java.util.Random;
import java.util.Scanner;
public class pattern3a {
    Run | Debug
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner input = new Scanner(System.in);

        System.out.println(x:"Enter number of dices: ");
        int dice_num = input.nextInt();

        System.out.println(x:"Enter number of rolls: ");
        int dice_roll = input.nextInt();

        int [] arraySum = new int[6*dice_num];

        for (int i = 0; i < dice_roll; i++) {
            Random randomGenerator = new Random();
            int sum = 0;
            for (int j = 0; j < dice_num; j++) {
                int randomInt = randomGenerator.nextInt(bound:7);
                sum += randomInt;
            }
            arraySum[sum-1] += 1;
        }
        for (int k = 0; k < arraySum.length;k++) {
            float percentages =(arraySum[k]*100)/dice_roll;
            System.out.printf(format:"Sum: %d Frequency: %d Percentages : %.2f\n", k+1, arraySum[k], percentages);
        }
    }
}
```

After

```java
import java.util.Random;
import java.util.Scanner;
public class pattern3b {
    Run | Debug
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner input = new Scanner(System.in);

        System.out.println(x:"Enter number of dices: ");
        int dice_num = input.nextInt();

        System.out.println(x:"Enter number of rolls: ");
        int dice_roll = input.nextInt();

        int [] arraySum = new int[6*dice_num];
        Random randomGenerator = new Random();
        int sum, randomInt, percentages;
        for (int i = 0; i < dice_roll; i++) {
            sum = 0;
            for (int j = 0; j < dice_num; j++) {
                randomInt = randomGenerator.nextInt(bound:7);
                sum += randomInt;
            }
            arraySum[sum-1] += 1;
        }
        for (int k = 0; k < arraySum.length;k++) {
            percentages =(arraySum[k]*100)/dice_roll;
            System.out.printf(format:"Sum: %d Frequency: %d Percentages : %.2f\n", k+1, arraySum[k], percentages);
        }
    }
}
```

설명

반복문 내의 변수 선언

반복문 내의 변수 선언은 반복 횟수만큼 동일한 이름의 변수 선언이 반복되므로 메모리 낭비 발생 -> 반복문 밖의 변수 선언으로 변경

그린화 패턴 #10

Before

```java
public class After {
    no usages
    public static void main(String[] args) throws Exception{
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

        long startTime = System.currentTimeMillis();
        for(int i=0; i<1000; i++){
            for(int j=0; j<1000; j++){
                bw.write( str: "i is " + i + " j is "+ j+ "\n");
            }
        }
        bw.write( str: "finish with buffer!!\n");
        bw.flush();

        long timeConsumed = System.currentTimeMillis()-startTime;
        bw.write( str: "Time(milisecond) " + timeConsumed);
        bw.flush();
        bw.close();
    }
}
```

After

```java
public class Before {
    no usages
    public static void main(String[] args) throws Exception {

        long startTime = System.currentTimeMillis();
        for(int i=0; i<1000; i++){
            for(int j=0; j<1000; j++){
                System.out.println("i is " + i + " j is " + j);
            }
        }
        System.out.println("finish without buffer!!");
        long timeConsumed = System.currentTimeMillis()-startTime;
        System.out.println("Time(milisecond) " + timeConsumed);
    }
}
```

설명

Buffer를 이용하여 대량의 출력을 효율적으로 실행.

그린화 패턴 #11

Before

```java
public class Main {

    public static void main(String[] args) {
        ArrayList<Integer> arrayList = new ArrayList<>();
        LinkedList<Integer> linkedList = new LinkedList<>();
        for (int i = 0; i < 1000000; i++) {
            arrayList.add(i);
            linkedList.add(i);
        }

        for (Integer integer : arrayList) {
            System.out.println("integer = " + integer);
        }

        for (int i = 0; i < linkedList.size(); i++) {
            System.out.println("linkedList = " + linkedList.get(i));
        }
    }
}
```

After

```java
public class Main {

    public static void main(String[] args) {
        ArrayList<Integer> arrayList = new ArrayList<>();
        LinkedList<Integer> linkedList = new LinkedList<>();
        for (int i = 0; i < 1000000; i++) {
            arrayList.add(i);
            linkedList.add(i);
        }

        for (int i = 0; i < arrayList.size(); i++) {
            System.out.println("arrayList.get(i) = " + arrayList.get(i));
        }

        for (Integer integer : linkedList) {
            System.out.println("integer = " + integer);
        }
    }
}
```

설명

자료구조별 적합한 반복 연산 기법을 도입하였다. ArrayList의 경우, 내부적으로 배열을 사용하여 데이터를 저장한다. 인덱스를 통해 배열에 직접 접근하는 것은 매우 빠른 작업이다. 반면, for-each 루프는 Iterator 객체를 사용하여 요소를 반복한다. Iterator는 요소에 접근하기 위해 추가적인 메소드 호출(next() 및 hasNext())이 필요하며, 이는 추가적인 오버헤드를 발생시킬 수 있다. ArrayList의 경우에는 index 기반 for문이 효율적이다.

그린화 패턴 #12

Before

```java
public class Main {

    public static void main(String[] args) {
        LocalDateTime start = LocalDateTime.now();
        for (int i = 0; i < 1000000; i++) {
            LocalDateTime now = LocalDateTime.now();
            long millis = ChronoUnit.MILLIS.between(start, now);
            System.out.println("millis = " + millis);
        }
    }
}
```

After

```java
public class Main {

    public static void main(String[] args) {
        Instant start = Instant.now();
        for (int i = 0; i < 1000000; i++) {
            Instant now = Instant.now();
            long millis = Duration.between(start, now).toMillis();
            System.out.println("millis = " + millis);
        }
    }
}
```

설명

자료구조별 적합한 반복 연산 기법을 도입하였다. ArrayList의 경우, 내부적으로 배열을 사용하여 데이터를 저장한다. 인덱스를 통해 배열에 직접 접근하는 것은 매우 빠른 작업이다. 반면, for-each 루프는 Iterator 객체를 사용하여 요소를 반복한다. Iterator는 요소에 접근하기 위해 추가적인 메소드 호출(next() 및 hasNext())이 필요하며, 이는 추가적인 오버헤드를 발생시킬 수 있다. LinkedList의 경우에는 iterator 기반 for문이 효율적이다.

그린화 패턴 #13

Before

```java
public class Main {

    public static void main(String[] args) {
        Integer sum = 0;
        for (Integer i = 0; i < 1000000; i++) {
            sum += i;
        }
        System.out.println("sum = " + sum);
    }
}
```

After

```java
public class Main {

    public static void main(String[] args) {
        int sum = 0;
        for (int i = 0; i < 1000000; i++) {
            sum += i;
        }
        System.out.println("sum = " + sum);
    }
}
```

설명

시간 연산이 많은 경우에는 인간 친화적인 LocalDateTime이나 DateTimeZone을 사용하는 것보다, Duration과 Instant를 사용하여 시간 변환 오버헤드를 줄이는 것이 효율적이다.

그린화 패턴 #14

Before

```java
public class Main {

    public static void main(String[] args) {
        int sum = 0;
        for (int i = 0; i < 1000000; i++) {
            Integer num = new Integer( value: 100);
            sum += num;
        }
        System.out.println("sum = " + sum);
    }
}
```

After

```java
public class Main {

    public static void main(String[] args) {
        int sum = 0;
        for (int i = 0; i < 1000000; i++) {
            Integer num = 100;
            sum += num;
        }
        System.out.println("sum = " + sum);
    }
}
```

설명

Wrapper Class는 연산시 boxing과 unboxing 연산이 추가로 수행되기 때문에, 연산이 잦다고 판단되거나, null값이 필요가 없거나, Collection에 담아야 할 필요가 없는 경우에는 primitive type을 사용하는 것이 효율적이다. 또한 primitive type은 스택에, Wrapper Class는 Heap영역에 해당되므로 추가적인 GC 오버헤드도 줄일 수 있다. Wrapper 클래스 중 Boolean, Byte, Character, Integer는 일정 범위의 값을 사전에 생성해 pool에 저장하는데, 사용하고자 하는 값이 이 범위 안에 해당할 경우 Wrapper 클래스 생성시 추가적인 Heap영역이 필요하지 않고, 생성 및 GC 과정이 생략되어 효율적이다.