그린화 패턴 보고서

소프트웨어공학개론 (SWE3002-41)

Team 6 김선우, 설현원, 이찬구, 이현민, 조재범, 한수민

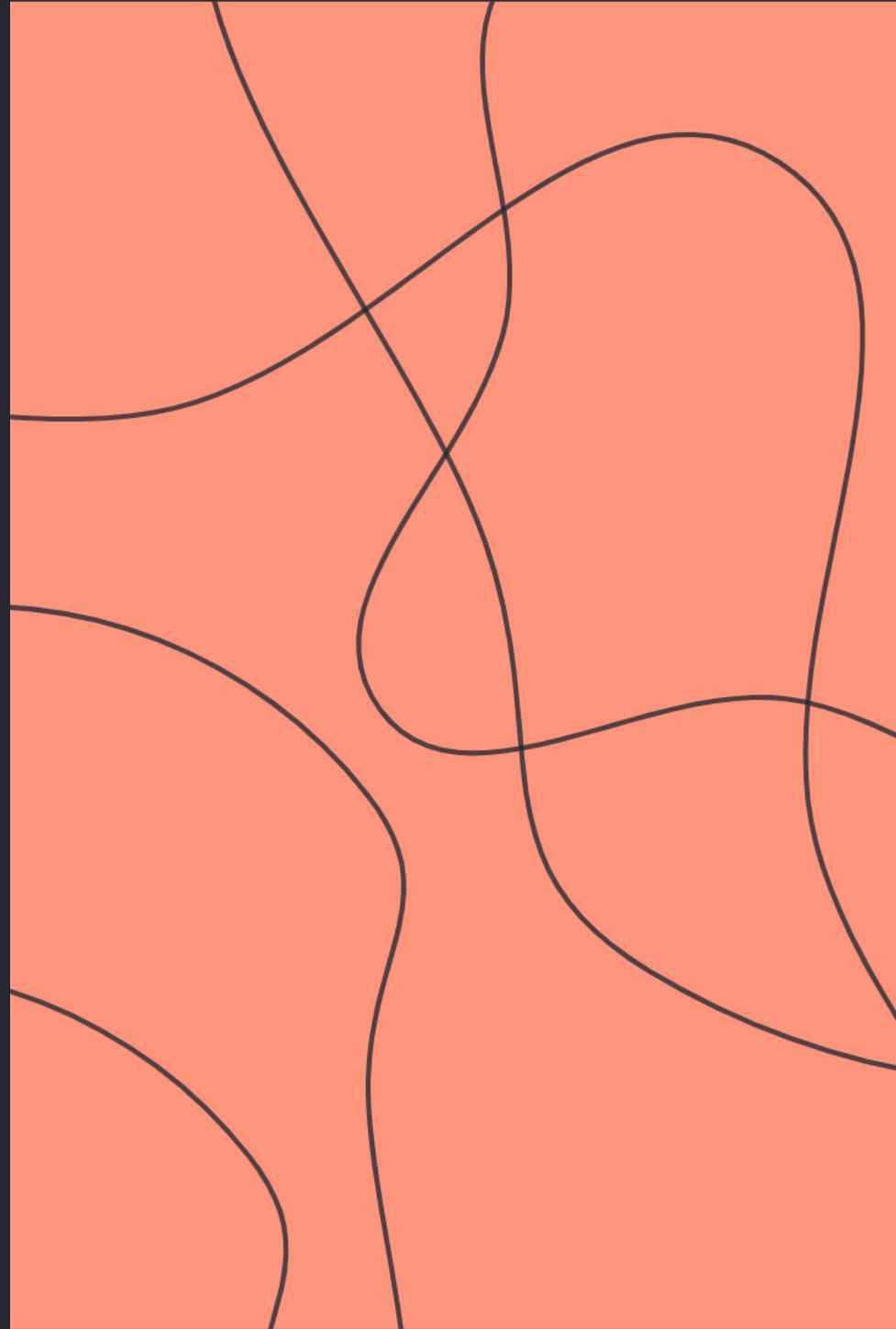
요약: 44개 패턴

Avoid Redundancy

패턴 1. object pooling
패턴 2. priority queue filtering
패턴 3. static variable utilization
패턴 4. final variable utilization
패턴 5. cache utilization
패턴 6. buffer utilization

Drop Unnecessary Ops

패턴 7. do not check array boundary
패턴 8. remark debugging output
패턴 9. short circuiting utilization
패턴 10. static method utilization
패턴 11. initializer compaction



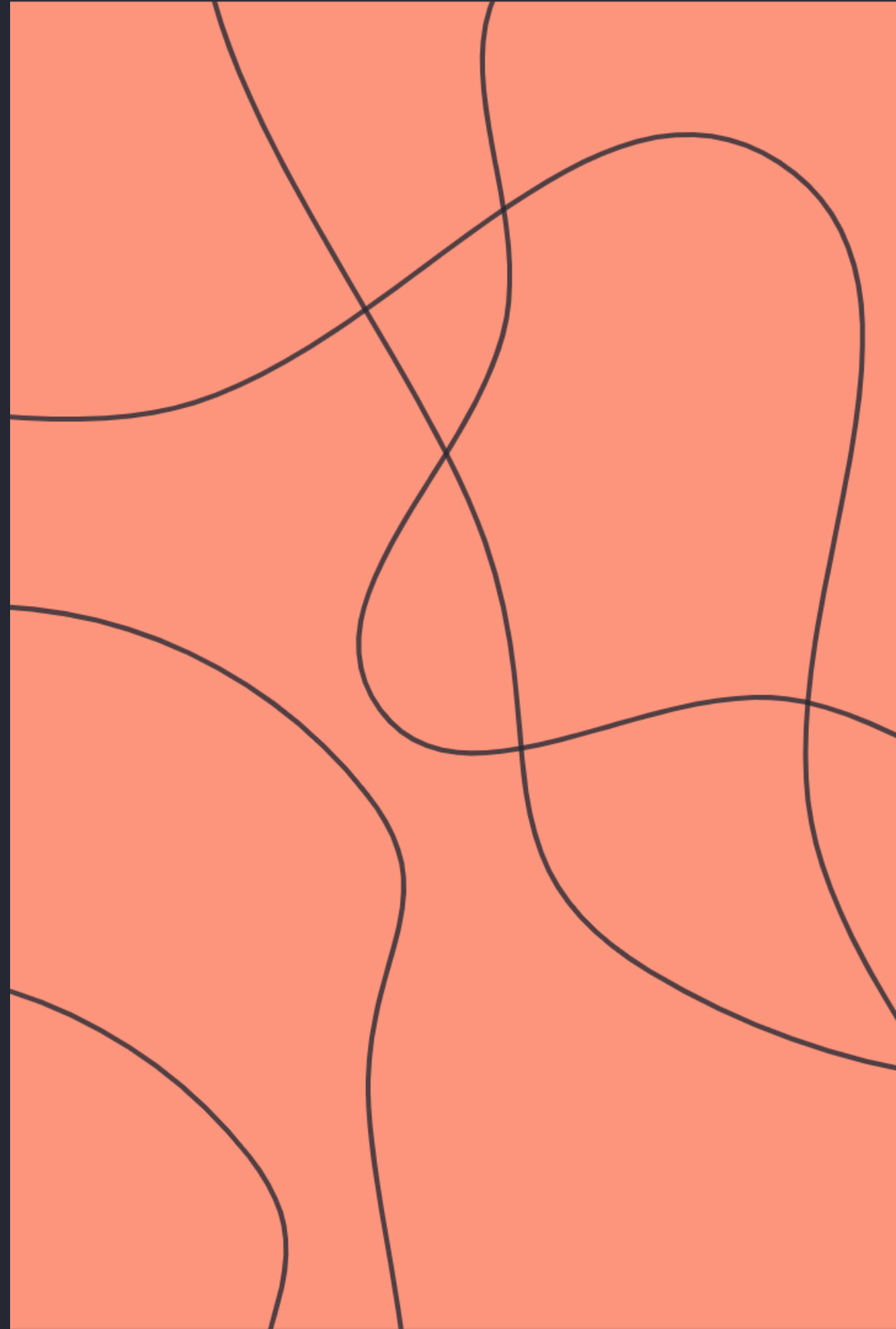
요약: 44개 패턴

Better Algorithm

패턴 12. binary search than linear search
패턴 13. bfs than dfs
패턴 14. use prefix sum
패턴 15. use double pointers
패턴 16. dynamic programming is not always good

Better Coding Styles

패턴 17. for-loop minimization
패턴 18. use enhanced for-loop
패턴 19. switch than if-else
패턴 20. row-major than column-major
패턴 21. shift operator than multiplication
패턴 22. prefix addition than postfix addition
패턴 23. iteration than recursion
패턴 24. thread utilization



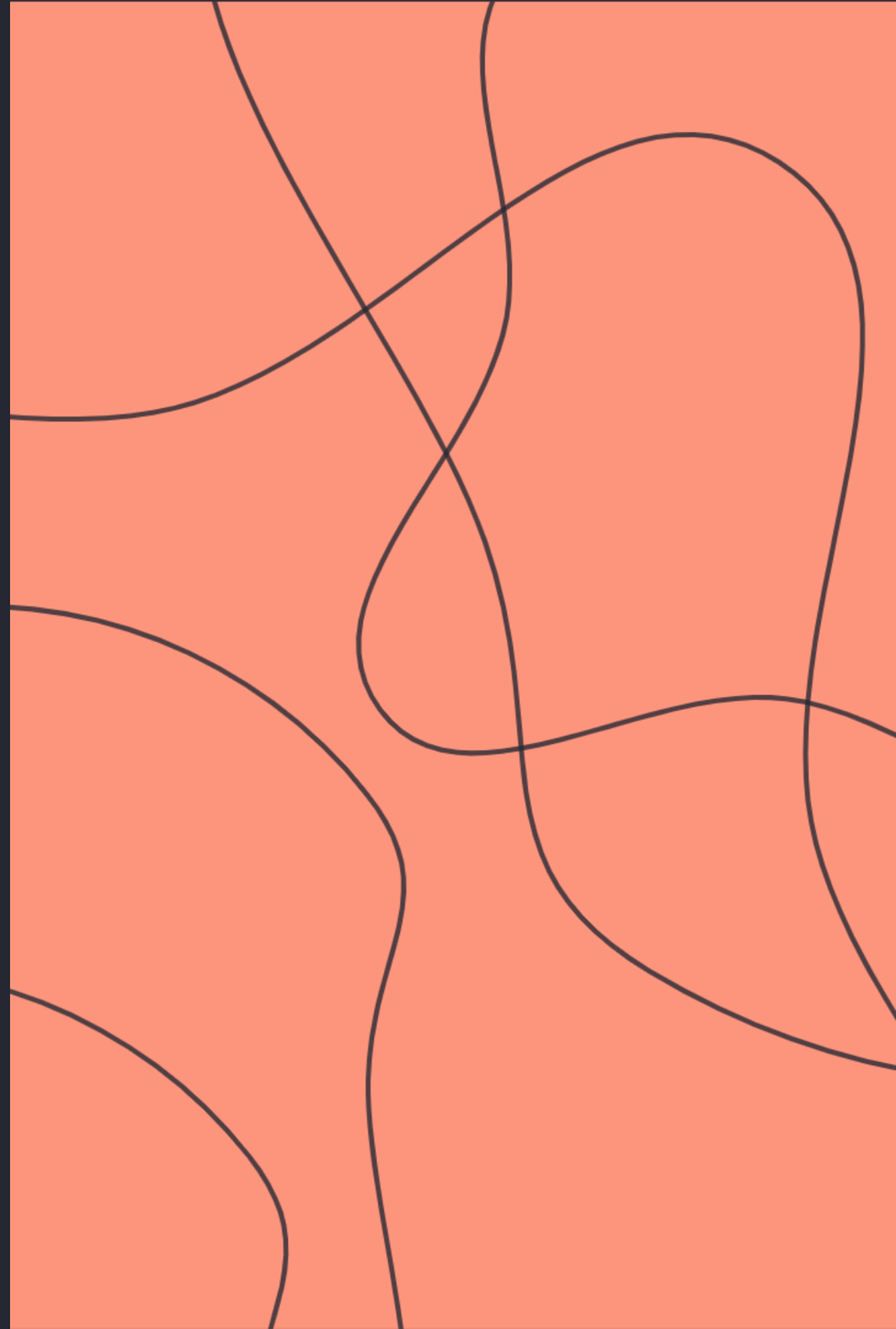
요약: 44개 패턴

Better API

패턴 25. use sublist deletion
패턴 26. use parallel sort
패턴 27. use DecimalFormat
패턴 28. do not use StringTokenizer
패턴 29. charAt than getBytes
패턴 30. stream is not always good 1
패턴 31. stream is not always good 2
패턴 32. regex is not always good

Better Data Structure

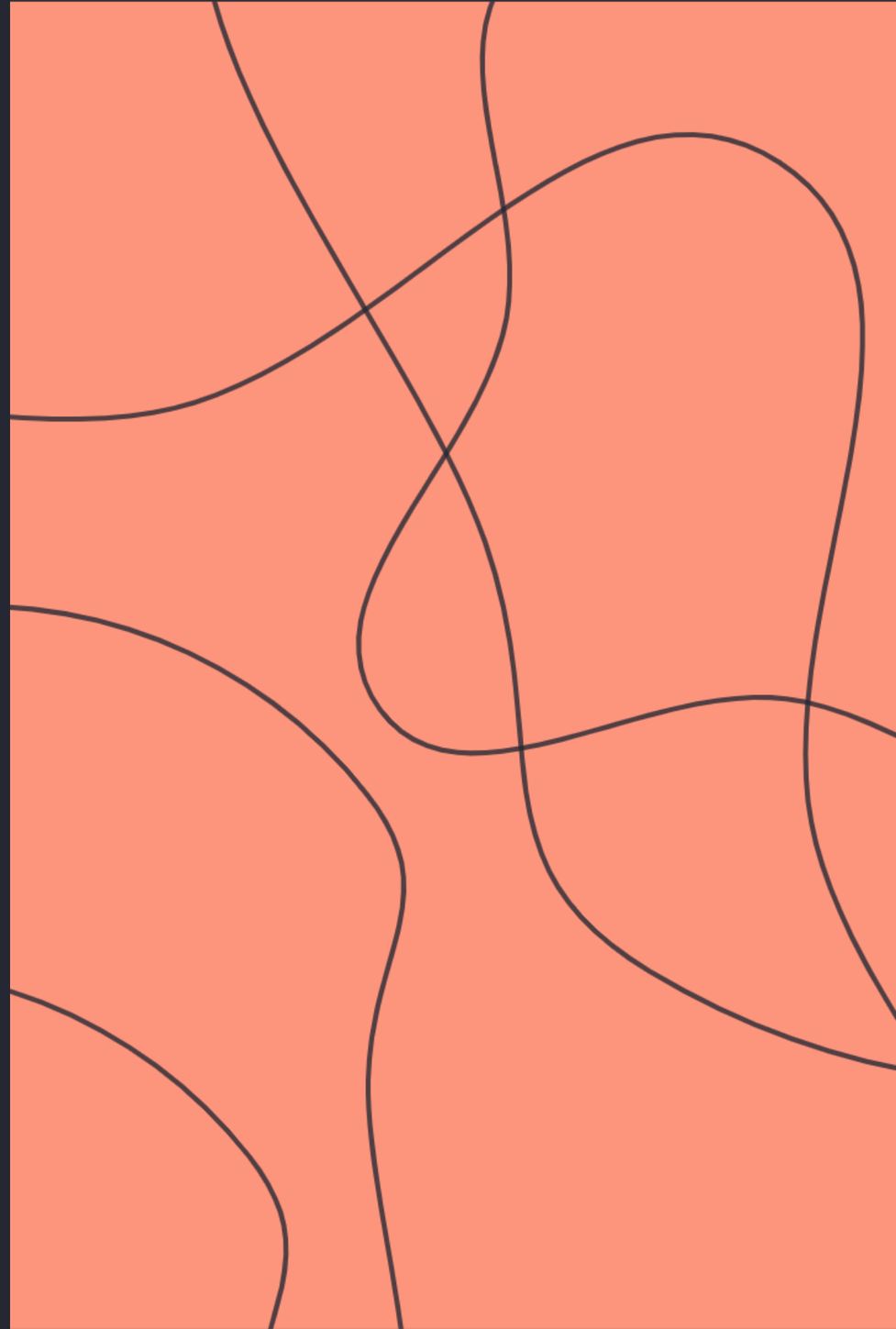
패턴 33. HashMap than TreeMap
패턴 34. linked list than array
패턴 35. literal type than object type
패턴 36. integer than float
패턴 37. use integer flag



요약: 44개 패턴

Save Memory Resources

패턴 38. set initial capacity for collections
패턴 39. free obsolete string
패턴 40. free obsolete list
패턴 41. free obsolete stack item
패턴 42. free obsolete thread
패턴 43. inline functions than fine-grained functions
패턴 44. unnecessary parameters



그린화 패턴

: pattern name

- 탄소배출량 감소 이유 및 패턴 범용성의 근거

구성 보고서

Before: 0.00s

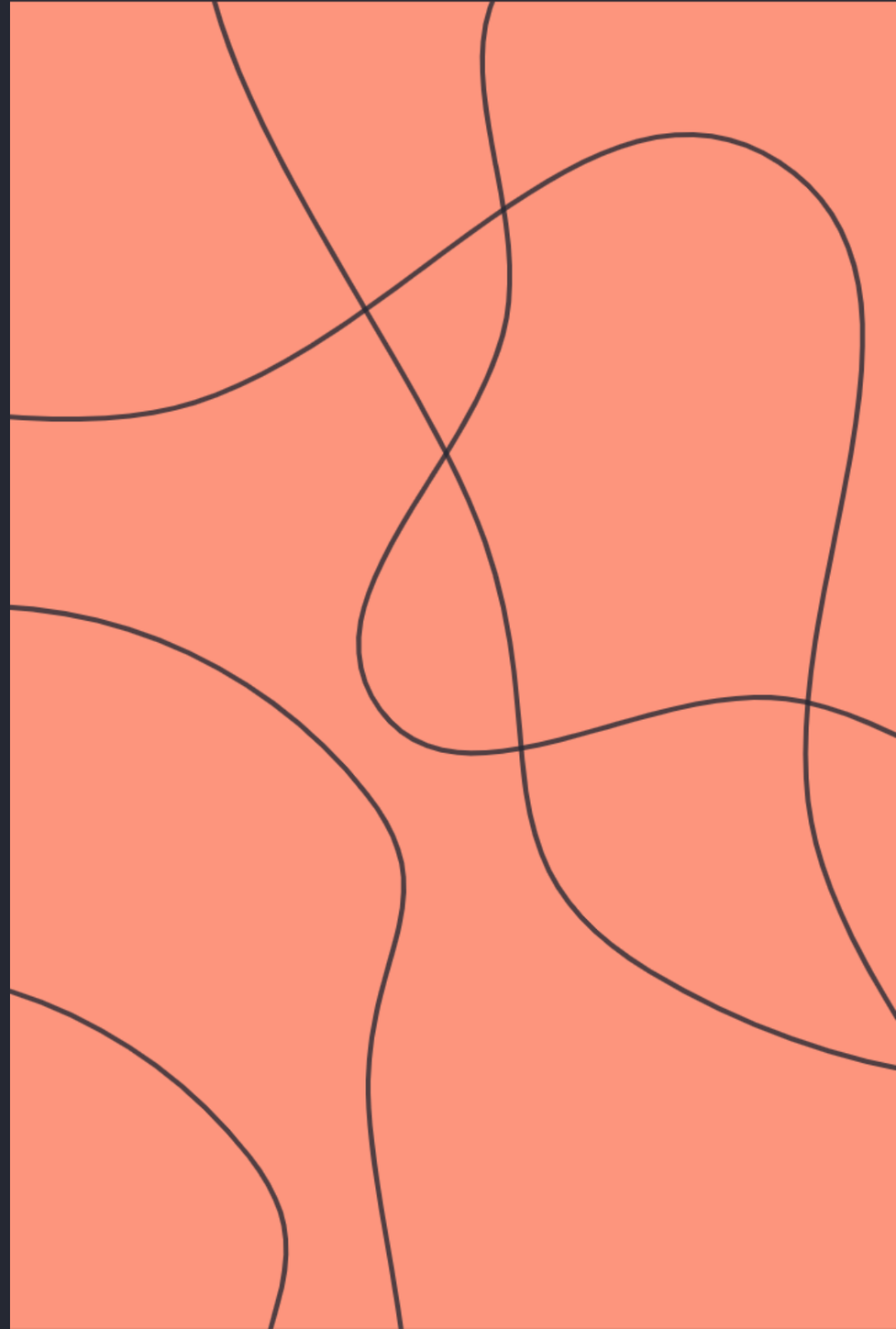
Before code

After: 0.00s

After code

Avoid Redundancy

패턴 1. object pooling
패턴 2. priority queue filtering
패턴 3. static variable utilization
패턴 4. final variable utilization
패턴 5. cache utilization
패턴 6. buffer utilization



그린화 패턴 1

object pooling

- 객체를 여러 번 생성해야 하는 경우, 일반적인 방법으로 생성하는 방법 대신, 오브젝트 풀링을 통해 재사용하여 생성 및 GC 비용을 줄임

Before: 1.3s

```java
public class Before {
    Run | Debug
    public static void main(String[] args){
        for(int i=0; i<100000; i++) {
            new ArrayList<>(i);
        }
    }
}
```

After: 0.01s

```java
public class After {
    public static class ObjectPool {
        private static final int MAX_POOL_SIZE = 3;
        private static Queue<ArrayList<Object>> pool = new LinkedList<>();

        public static Object getObject() {
            if(pool.isEmpty()) {
                return new Object();
            } else {
                return pool.poll();
            }
        }

        public static void returnObject(ArrayList<Object> object) {
            if(pool.size() < MAX_POOL_SIZE) {
                pool.add(object);
            }
        }
    }

    Run | Debug
    public static void main(String[] args){
        for(int i=0; i<100000; i++) {
            ObjectPool.getObject();
        }
    }
}
```

그린화 패턴 2

priority queue filtering

- 작은 수 순서대로 특정 개수만큼 뽑을 때, 최대 힙으로 구현되어 있는 우선순위 큐와 Reverse를 활용하여, 이미 특정 개수에 포함되지 않는 수들을 필터링하여 연산을 줄임

Before: 7s

```java
public class Before {
    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        PriorityQueue<Integer> q = new PriorityQueue<Integer>(Collections.reverseOrder());
        for(int i=0; i<10000000; i++){
            q.offer(i);
        }

        while(q.size() > 0){
            v.add(q.poll());
        }
        Collections.reverse(v);
    }
}
```

After: 0.01s

```java
public class After {
    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        PriorityQueue<Integer> q = new PriorityQueue<Integer>(Collections.reverseOrder());
        for(int i=0; i<10000000; i++){
            if(q.size() < 5){
                q.offer(i);
            }
            else{
                int curr = q.peek();
                if(i < curr){
                    q.poll();
                    q.offer(i);
                }
            }
        }

        while(q.size() > 0){
            v.add(q.poll());
        }
        Collections.reverse(v);
    }
}
```

그린화 패턴 3

: static variable utilization

- 메소드 안에서 자주 초기화되는 변수의 경우, 가능하다면 class의 stati변수로 선언하는 것이 좋음.
- static변수는 메모리 내에서 고정되기 때문에 메소드 호출시마다 중복되는 메모리 연산을 막을 수 있음.

Before: 3.9014s

```java
public class Before {
    private static int func1(){
        int[] a = new int[10000000];
        a[0] = 10;
        return a[0];
    }
    public static void main(String[] args){
        int iterations = 1000;


        for(int i=0; i<iterations; i++){
            func1();
        }
    }
}
```

After: 0.3529s

```java
public class After {
    static int[] a = new int[10000000];
    private static int func1(){
        a[0] = 10;
        return a[0];
    }
    public static void main(String[] args){
        int iterations = 1000;

        for(int i=0; i<iterations; i++){
            func1();
        }
    }
}
```

그린화 패턴 4

: final variable utilization

- for-loop안에서 반복적으로 초기화 되는 변수를 final type으로 변경.
- for-loop밖에서 초기화하는 것보다 가독성을 높이는 한편, 컴파일러로 하여금 최적화도 가능하게 하여 탄소배출량을 감소시킴.

Before: 0.2099s

```java
public class Before {
    public static void main(String[] args) {
        for (int i = 0; i < 1000000; i++) {
            String value = "Hello";
            value = value + "World";
        }
    }
}
```

After: 0.1665s

```java
public class After {
    public static void main(String[] args) {
        for (int i = 0; i < 1000000; i++) {
            final String value = "Hello";
            String result = value + "World";
        }
    }
}
```

그린화 패턴 5

: cache utilization

- Regular Expression이 사용하는 patter처럼, 특정 객체가 동일한 문자열을 반복해서 사용할 경우, 해당 문자열을 caching 하는 것이 유리.
- 아래의 예시에서는 Pattern 패키지의 compile메소드를 사용하여 caching했음.

Before: 1.6503s

```java
public class Before {
    static boolean isRomanNumeralSlow(String s) {
        return s.matches("^(?=.)M*(C[MD]|D?C{0,3})" +
        "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
    }
    public static void main(String[] args){
        int iterations = 1000000;
        for(int i=0; i<iterations; i++){
            isRomanNumeralFast("Lorem ipsum dolor sit amet,
                consectetur adipiscing elit, sed do eiusmod tempor
                incididunt ut labore et dolore magna aliqua.
                Nisl tincidunt eget nullam non.");
        }
    }
}
```

After: 0.5213s

```java
import java.util.regex.Pattern;

public class After {
    private static final Pattern ROMAN = Pattern.compile(
        "^(?=.)M*(C[MD]|D?C{0,3})" +
        "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$"
    );
    static boolean isRomanNumeralFast(String s) {
        return ROMAN.matcher(s).matches();
    }
    public static void main(String[] args){
        int iterations = 1000000;
        for(int i=0; i<iterations; i++){
            isRomanNumeralFast("Lorem ipsum dolor sit amet,
                consectetur adipiscing elit, sed do eiusmod tempor
                incididunt ut labore et dolore magna aliqua.
                Nisl tincidunt eget nullam non.");
        }
    }
}
```

그린화 패턴 6

: buffer utilization

- 파일 입출력시 I/O cost가 가장 크기 때문에, buffer를 사용하여 I/O횟수를 줄이면 CPU utilization을 높이고 runtime을 줄일 수 있음.
- CPU에 쓰이는 power는 증가하지만, 그보다 runtime감소로 인한 탄소배출량 감소가 더 큼.

Before: 0.2453s

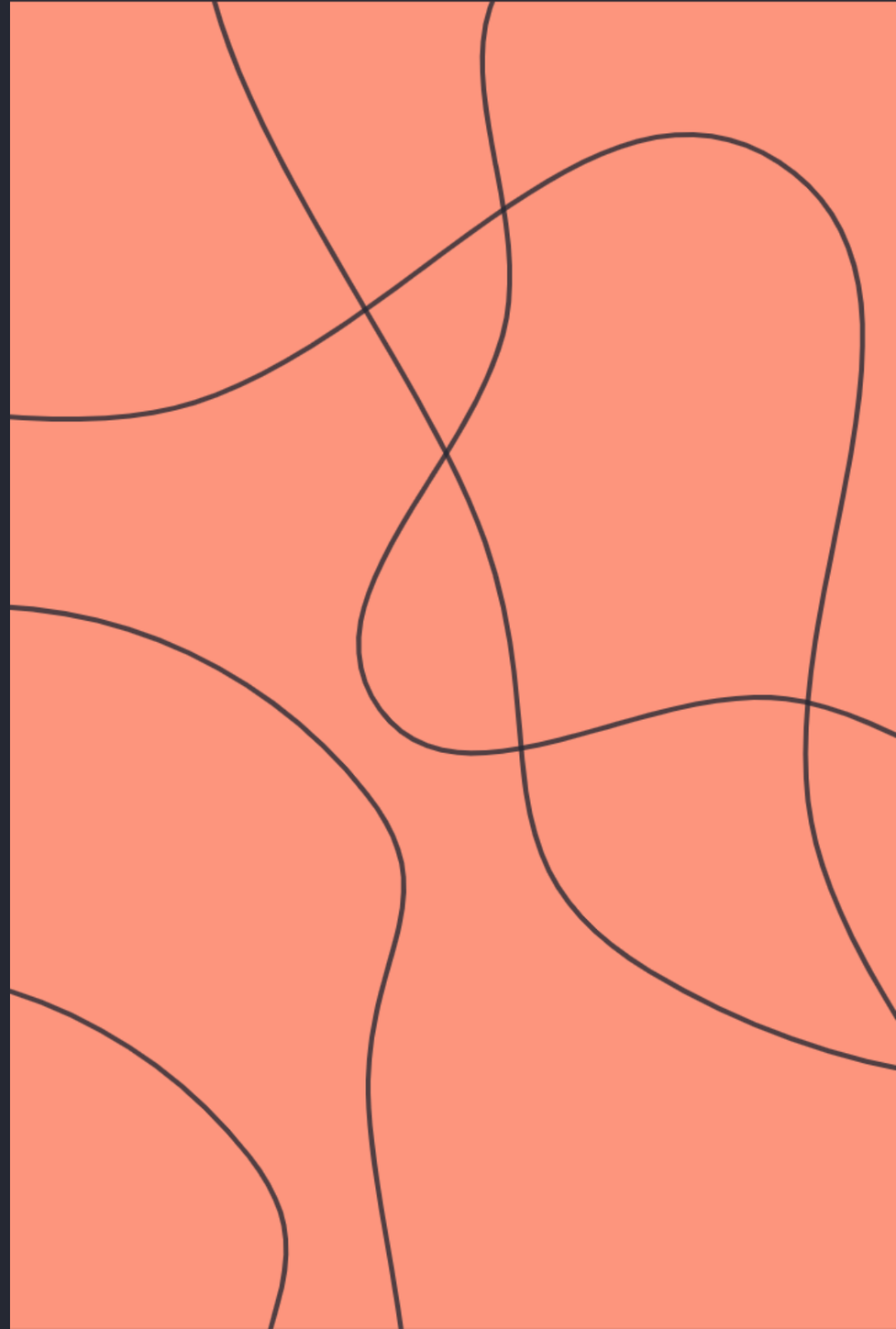
```java
import java.io.*;

public class Before {

    public static void main(String[] args) {
        try {
            FileInputStream in = new FileInputStream("input.txt");
            FileOutputStream out = new FileOutputStream("output.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

            in.close();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

After: 0.1766s

```java
import java.io.*;

public class After {

    public static void main(String[] args) {
        try {
            BufferedInputStream in = new BufferedInputStream(new FileInputStream("input.txt"));
            BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("output.txt"));
            byte[] buffer = new byte[8192];
            int length;

            while ((length = in.read(buffer)) > 0) {
                out.write(buffer, 0, length);
            }

            in.close();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Drop Unnecessary Ops

패턴 7. do not check array boundary
패턴 8. remark debugging output
패턴 9. short circuiting utilization
패턴 10. static method utilization
패턴 11. initializer compaction



그린화 패턴 7

: do not check array boundary

- 기존 코드에서는 안전한 배열 접근을 위해 접근시마다 boundary check를 함.
- try catch로 예외처리를 하면 boundary check없이도 안전한 배열 접근이 가능.

Before: 0.7278s

```java
public class Before {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};

        int iterations = 100000;
        for (int i = 0; i < iterations; i++) {
            if (i < array.length) {
                int res = array[3];
            } else {
                System.out.println("Array index out of bounds");
            }
        }
    }
}
```

After: 0.3114s

```java
public class After {
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};

        int iterations = 100000;
        for (int i = 0; i < iterations; i++) {
            try {
                int res = array[3];
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out of bounds");
            }
        }
    }
}
```

그린화 패턴 8

: remark debugging output

- 입출력 메소드는 많은 자원과 시간을 필요로 하는 작업임.
- 따라서 production 코드에서는 debugging용 I/O를 제거해야 자원의 낭비를 막을 수 있음.

Before: 0.3065s

```java
public class Before {
    public static void main(String[] args) {
        for (int i = 0; i < 10000; i++) {
            System.out.println("Current index: " + i);
            int result = process(i);
        }
    }

    private static int process(int value) {
        return value * 2;
    }
}
```

After: 0.1598s

```java
public class After {
    public static void main(String[] args) {
        for (int i = 0; i < 10000; i++) {
            // System.out.println("Current index: " + i);
            int result = process(i);
        }
    }

    private static int process(int value) {
        return value * 2;
    }
}
```

그린화 패턴 9

: short circuiting utilization

- short circuiting은 AND나 OR연산자에서 첫번째 피연산자가 조건을 만족하면 두번째 피연산자를 검사하지 않는 상황임.
- 따라서 AND연산자의 경우 더 강력한 조건을 첫번째로 배치시키면 두번째 조건은 검사하지 않을 가능성이 증가.

Before: 2.8476s

```java
import java.util.Random;
public class Before {
    public static boolean test(int num, boolean flag){
        System.out.println("This test takes time...");
        System.out.println("This print is for taking time");
        System.out.println("This test takes time...");

        if(num%777==0) return !flag;
        return flag;
    }
    public static void main(String[] args){
        Random random = new Random();
        int randA = random.nextInt(10000);
        int randB = random.nextInt(10000);

        int iterations = 200000;
        for(int i=0; i<iterations; i++){
            if(test(randA, true) && test(randB, false)){
                // System.out.println("randA is odd and randB is 77");
            }
        }
    }
}
```

After: 1.5211s

```java
import java.util.Random;
public class After {
    public static boolean test(int num, boolean flag){
        System.out.println("This test takes time...");
        System.out.println("This print is for taking time");
        System.out.println("This test takes time...");

        if(num%777==0) return !flag;
        return flag;
    }
    public static void main(String[] args){
        Random random = new Random();
        int randA = random.nextInt(10000);
        int randB = random.nextInt(10000);

        int iterations = 200000;
        for(int i=0; i<iterations; i++){
            if(test(randB, false) && test(randA, true)){
                // System.out.println("randA is odd and randB is 77");
            }
        }
    }
}
```

그린화 패턴 10

: static method utilization

- 특정 인스턴스마다 필요한 메서드가 아니라면 static 메서드로 선언하는 것이 좋음.
- 그렇지 않으면 해당 메서드 호출시마다 불필요한 객체 선언을 하게 됨.

Before: 0.2101s

```java
public class Before {
    private int value;
    public Before(int value) {
        this.value = value;
    }
    public int calculateSquare() {
        return value * value;
    }
    public static void main(String[] args) {
        int iterations = 1000000;
        for (int i = 0; i < iterations; i++) {
            Before obj = new Before(i);
            int result = obj.calculateSquare();
        }
    }
}
```

After: 0.1697s

```java
public class After {
    private int value;
    public After(int value) {
        this.value = value;
    }
    public static int calculateSquare(int value) {
        return value * value;
    }
    public static void main(String[] args) {
        int iterations = 1000000;
        for (int i = 0; i < iterations; i++) {
            int result = calculateSquare(i);
        }
    }
}
```

그린화 패턴 11

: initializer compaction

- 객체를 반복적으로 생성하면, 생성자도 반복적으로 호출됨.
- 따라서 초기화할 필요가 없는 멤버변수의 초기화 등 불필요한 연산을 생성자로부터 제거해야 함.

Before: 0.8452s

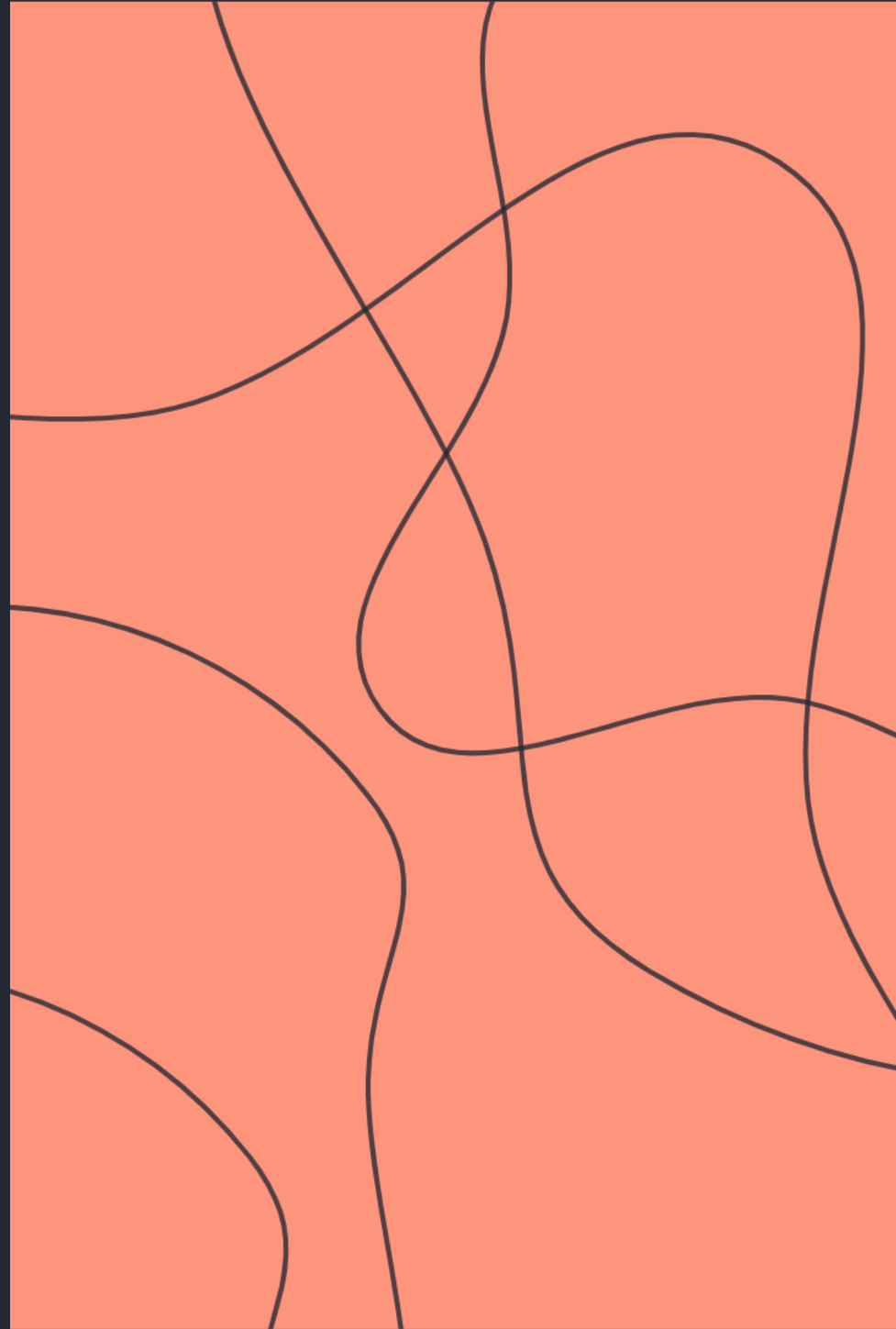
```
class MyObject {
    private int value;
    private String[] str = new String[100];
    public MyObject(int value) {
        this.value = value;
        for(int i=0; i<100; i++){
            str[i] = new String("");
        }
    }
}
public class Before {
    public static void main(String[] args) {
        int iterations = 10000000;
        MyObject[] obj = new MyObject[iterations];
        for (int i = 0; i < iterations; i++) {
            obj[i] = new MyObject(0);
        }
    }
}
```

After: 0.7983s

```
class MyObject {
    private int value;
    private String str;
    public MyObject(int value) {
        this.value = value;
    }
}
public class After {
    public static void main(String[] args) {
        int iterations = 10000000;
        MyObject[] obj = new MyObject[iterations];
        for (int i = 0; i < iterations; i++) {
            obj[i] = new MyObject(0);
        }
    }
}
```

Better Algorithm

패턴 12. binary search than linear search
패턴 13. bfs than dfs
패턴 14. use prefix sum
패턴 15. use double pointers
패턴 16. dynamic programming is not always good



그린화 패턴 12

binary search than linear search

- 선형 자료구조에서 특정 원소를 찾을 때, 리니어 서치 대신 바이너리 서치를 사용하여 N번 연산을 LogN번 연산으로 줄임

Before: 4s

```java
public class Before {
    public static void linearSearch(ArrayList<Integer> nums, int target) {
        for (int i = 0; i < nums.size(); i++) {
            if (nums.get(i) == target) {
                break;
            }
        }
    }

    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        for(int i=0; i<100000; i++) {
            v.add(i);
            linearSearch(v, i);
        }
    }
}
```

After: 0.01s

```java
public class After {
    public static void binarySearch(ArrayList<Integer> nums, int target) {
        int start = 0;
        int end = nums.size() - 1;

        while (start <= end) {
            int mid = start + (end - start) / 2;
            if (nums.get(mid) == target) {
                break;
            } else if (nums.get(mid) < target) {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }
    }

    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        for(int i=0; i<100000; i++) {
            v.add(i);
            binarySearch(v, i);
        }
    }
}
```

그린화 패턴 13

bfs than dfs

- 뎁스가 긴 그래프에서 끝 지점까지의 거리를 빠르게 탐색하고 싶을 때, dfs 대신, bfs로 탐색하여 긴 뎁스에 빠지지 않고 빠르게 끝 지점을 탐색함

Before: 1s

```java
public class Before {
    static int[][] dir = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
    static int row = 4;
    static int col = 6;
    static int[][] vec = new int[row][col];
    static int[][] visited = new int[row][col];

    public static void dfs(int y, int x) {
        if (y == row - 1 && x == col - 1) {
            return;
        }

        for (int i = 0; i < 4; i++) {
            int ny = y + dir[i][0];
            int nx = x + dir[i][1];

            if (ny < 0 || ny >= row || nx < 0 || nx >= col) {
                continue;
            }

            if (visited[ny][nx] >= 1) {
                continue;
            }

            if (vec[ny][nx] == 0) {
                continue;
            }
            visited[ny][nx] = visited[y][x] + 1;
            dfs(ny, nx);
            visited[ny][nx] = 0;
        }
    }

    Run | Debug
    public static void main(String[] args){
        vec = new int[][]{
            {1, 1, 1, 1, 1, 1},
            {1, 0, 1, 1, 0, 0},
            {1, 1, 0, 0, 0, 0},
            {0, 1, 1, 1, 1, 1}
        };

        for(int i=0; i<1000000; i++){
            visited = new int[row][col];
            dfs(y:0, x:0);
        }
    }
}
```

After: 0.7s

```java
public class After {
    static int[][] dir = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
    static int row = 4;
    static int col = 6;
    static int[][] vec = new int[row][col];
    static int[][] visited = new int[row][col];

    public static void bfs(int y, int x) {
        Queue<int[]> q = new LinkedList<>();
        q.add(new int[]{y, x});
        visited[y][x] = 1;

        while (!q.isEmpty()) {
            int[] curr = q.poll();

            if(curr[0] == row-1 && curr[1] == col-1){
                return;
            }

            for (int i = 0; i < 4; i++) {
                int ny = curr[0] + dir[i][0];
                int nx = curr[1] + dir[i][1];

                if (ny < 0 || ny >= row || nx < 0 || nx >= col) {
                    continue;
                }

                if (visited[ny][nx] >= 1) {
                    continue;
                }

                if (vec[ny][nx] == 0) {
                    continue;
                }
                q.add(new int[]{ny, nx});
                visited[ny][nx] = visited[curr[0]][curr[1]] + 1;
            }
        }
    }

    Run | Debug
    public static void main(String[] args){
        long begin = System.currentTimeMillis();

        vec = new int[][]{
            {1, 1, 1, 1, 1, 1},
            {1, 0, 1, 1, 0, 0},
            {1, 1, 0, 0, 0, 0},
            {0, 1, 1, 1, 1, 1}
        };

        for(int i=0; i<1000000; i++){
            visited = new int[row][col];
            bfs(y:0, x:0);
        }

        long dur = System.currentTimeMillis() - begin;
        System.out.println("Execution Time: " + dur/1000.0 + " seconds");
    }
}
```

그린화 패턴 14

use prefix sum

- 선형 자료구조에서 특정 구간 원소들의 합 여러번 구할 때, 매번 구간을 탐색해서 찾는 것 대신, 한번 구간 합 배열을 계산하고 이 배열을 참조해서 탐색 연산을 줄임

Before: 8s

```java
public class Before {
    public static int func(ArrayList<Integer> v, int start, int end){
        int sum = 0;
        for(int i=start; i<end; i++){
            sum += v.get(i);
        }
        return sum;
    }

    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        for (int i = 0; i < 1000000; i++) {
            v.add(i);
        }

        for (int i = 0; i < 100000; i++) {
            func(v, i, i+100000);
        }
    }
}
```

After: 0.07s

```java
public class After {
    public static ArrayList<Integer> psum(ArrayList<Integer> v){
        int len = v.size();
        ArrayList<Integer> ret = new ArrayList<Integer>(len);
        ret.add(index:0, v.get(index:0));
        for(int i=1; i<len; i++){
            ret.add(i, ret.get(i-1) + v.get(i));
        }

        return ret;
    }

    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        for (int i = 0; i < 1000000; i++) {
            v.add(i);
        }

        ArrayList<Integer> vv = psum(v);
        int ret;
        for (int i = 0; i < 100000; i++) {
            ret = vv.get(i+100000) - vv.get(i);
        }
    }
}
```

그린화 패턴 15

use two pointer

- 두 수의 합이 특정 수가 되는 지 확인하고 싶을 경우, 이중 for문으로 조합을 만들어서 두 수의 합을 일일이 확인하는 대신, for문 하나에서 두 개의 포인터를 옮겨가며 확인하여 N^2 연산을 N 연산으로 줄임

Before: 0.6s

```java
public class Before {
    public static boolean func(ArrayList<Integer> v, int target){
        int len = v.size();
        for(int i=0; i<len-1; i++){
            for(int j=i+1; j<len; j++){
                if(v.get(i) + v.get(j) == target){
                    return true;
                }
            }
        }
        return false;
    }

    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        for (int i = 0; i < 10000; i++) {
            v.add(i);
        }

        for (int i = 0; i < 10000; i++) {
            func(v, i);
        }
    }
}
```

After: 0.1s

```java
public class After {
    public static boolean func(ArrayList<Integer> v, int target){
        int len = v.size();
        int left = 0;
        int right = len-1;
        while(left != right){
            int sum = v.get(left) + v.get(right);
            if(sum == target){
                return true;
            }
            else if(sum < target){
                left++;
            }
            else{
                right--;
            }
        }
        return false;
    }

    Run | Debug
    public static void main(String[] args){
        long begin = System.currentTimeMillis();

        ArrayList<Integer> v = new ArrayList<Integer>();
        for (int i = 0; i < 10000; i++) {
            v.add(i);
        }

        for (int i = 0; i < 10000; i++) {
            func(v, i);
        }

        long dur = System.currentTimeMillis() - begin;
        System.out.println("Execution Time: " + dur/1000.0 + " seconds");
    }
}
```

그린화 패턴 16

: dynamic programming is not always good

- 동적계획법은 최적 부분 구조가 가진 알고리즘(ex. Fibonacci)의 연산량을 줄일 수 있음. 그러나, 아래의 예시처럼 재귀함수가 아닌 for-loop로 구현을 하게 되면, 중복 부분 문제가 사라지기 때문에 dynamic programming을 활용하는 것은 overhead가 될 수 있음.

Before: 4.1526s

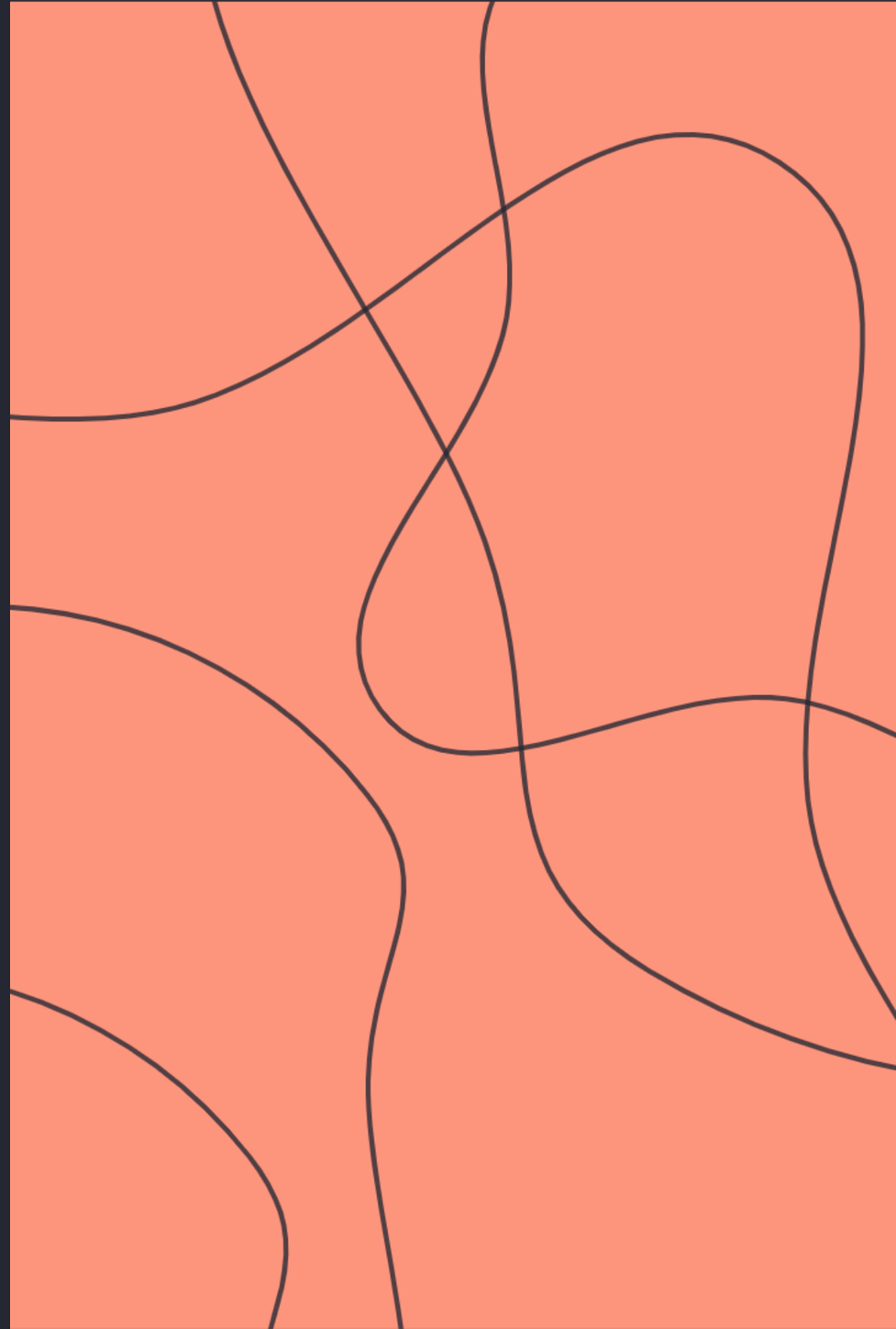
```java
import java.util.HashMap;
import java.util.Map;

public class Before {
    public static void main(String[] args) {
        int iterations = 1000000;
        for(int iter=0; iter<iterations; iter++){
            int n = 500;
            if (n <= 1) {
                System.out.println(n);
            }
            long[] fibArray = new long[n + 1];
            fibArray[0] = 0;
            fibArray[1] = 1;

            for (int i = 2; i <= n; i++) {
                fibArray[i] = fibArray[i - 1] + fibArray[i - 2];
            }
            System.out.println(fibArray[n]);
        }
    }
}
```

After: 3.1237s

```java
import java.util.HashMap;
import java.util.Map;

public class After {
    public static void main(String[] args) {
        int iterations = 1000000;
        for(int iter=0; iter<iterations; iter++){
            int n = 500;
            if (n <= 1) {
                System.out.println(n);
            }
            long prev = 0;
            long current = 1;
            for (int i = 2; i <= n; i++) {
                long next = prev + current;
                prev = current;
                current = next;
            }
            System.out.println(current);
        }
    }
}
```

Better Coding Style

패턴 17. for-loop minimization
패턴 18. use enhanced for-loop
패턴 19. switch than if-else
패턴 20. row-major than column-major
패턴 21. shift operator than multiplication
패턴 22. prefix addition than postfix addition
패턴 23. iteration than recursion
패턴 24. thread utilization



그린화 패턴 17

: for-loop minimization

- for 문의 분기 횟수를 줄여서 한 분기에 10번의 합을 계산하게 되면, 루프 내의 분기 횟수를 줄이고 캐시 효율성을 향상시켜서 실행시간을 단축시킬 수 있음.
- for-loop에 따른 순차계산이 많은 경우에는 cache-utilization이 낮아지기 때문에 한 분기 내에 캐시 효율성을 증가시켜줄 수있는 방향으로 코드를 수정할 수 있음.

Before: 12.018s

```java
import java.util.stream.IntStream;


public class Before {
    Run | Debug
    public static void main(String[] args) {
    long sum = 0;

    for(int k = 0 ; k < 100 ; k++){
        int[] arr = IntStream.range(startInclusive:1,endExclusive:100000000).toArray();
        int len = arr.length;
        int i;

        for (i = 0 ; i < len ; i++) {
            sum += arr[i];
        }
    }
    System.out.println(sum);
        }
}
```

After: 9.881s

```java
import java.util.stream.IntStream;


public class After {
    Run | Debug
    public static void main(String[] args) {
    long sum = 0;

    for(int k = 0 ; k < 100 ; k++){
        int[] arr = IntStream.range(startInclusive:1,endExclusive:100000000).toArray();
    int len = arr.length;
    int i;

    for (i = 0; i < len - 10; i += 10) {
        sum += arr[i] + arr[i + 1] + arr[i + 2] + arr[i + 3] + arr[i+4] + arr[i + 5] + arr[i + 6] + arr[i + 7] + arr[i+8] + arr[i + 9];
        }

    while (i < len) {
        sum += arr[i];
        i++;
        }
    }
    System.out.println(sum);
        }
}
```

그린화 패턴 18

: use enhanced for-loop

- for loop문의 정의를 좀더 간결하게 하여, 연산량을 줄임

Before: 0.04s

```java
import java.util.Arrays;

public class Before {
    Run | Debug
    public static void main(String[] args) {
        int[] array = new int[1000000];


        for (int i = 0; i < array.length; i++) {
            array[i] = array[i] * 2;
        }

    }
}
```

After: 0.01s

```java
import java.util.Arrays;

public class After {
    Run | Debug
    public static void main(String[] args) {
        int[] array = new int[1000000];


        for (int num : array) {
            num = num * 2;
        }

    }
}
```

그린화 패턴 19

: switch than if-else

- switch 구문은 정수나 문자열처럼 특정 유형의 값들을 빠르게 비교할 수 있고 , switch구문 내에서 jump table을 통해서 해당하는 구문의 블록에 빠르게 접근할 수 있기 때문에 실행시간이 줄어듬.
- 값에 대한 비교의 횟수가 많아지는 경우(else if가 다수일 때) 효과를 발휘할 수 있다

Before: 8.039s

```java
public class Before {

    Run | Debug
    public static void main(String[] args) {
    for(int i = 0 ; i < 100000 ; i++){
        int dayOfWeek = 6;

        String day;
        if(dayOfWeek == 1) {
            day = "Sunday";
        }

        else if(dayOfWeek == 2){
            day = "Monday";
        }

        else if(dayOfWeek == 3) {
            day = "Tuesday";
        }
        else if(dayOfWeek == 4){
            day = "Wednsday";
        }
        else if(dayOfWeek == 5){
            day = "Thursday";
        }
        else if(dayOfWeek == 6){
            day = "Friday";
        }
        else if(dayOfWeek == 7){
            day = "Saturday";
        }

        else {
            day = "Unknown";
        }


        System.out.println("Day of the week: " + day);
    }
    }
}
```

After: 7.917s

```java
public class After {

    Run | Debug
    public static void main(String[] args) {
    for(int i = 0 ; i < 100000 ; i++){
        int dayOfWeek = 3;

        String day;
        switch (dayOfWeek) {
            case 1:
                day = "Sunday";
                break;
            case 2:
                day = "Monday";
                break;
            case 3:
                day = "Tuesday";
                break;
            case 4:
                day = "Wednsday";
                break;
            case 5:
                day = "Thursday";
                break;
            case 6:
                day = "Friday";
                break;
            case 7:
                day = "Saturday";
                break;

            default:
                day = "Unknown";
        }

        System.out.println("Day of the week: " + day);
    }
    }
}
```

그린화 패턴 20

: row-major than column-major

- 자바에서 array에 해당하는 값은 메모리에 row-major 형태로 저장한다.
- 따라서 array값에 대해서 순차적으로 접근할 때에는 row-major로 접근하는 것이 column-major로 저장하는 것보다 memory-friendly 하기때문에(메모리 접근시간이 줄어들기 때문에) 런타임이 줄어든다.

Before: 5.837s

```java
import java.util.Random;

public class Before {
    Run | Debug
    public static void main(String[] args) {
    long totalSum = 0;
    for (int p = 0 ; p < 1000 ; p++){
        int rows = 1000;
        int cols = 1000;
        int[][] array = new int[rows][cols];
        Random random = new Random();


        for (int k = 0; k < cols; k++) {
            for (int i = 0; i < rows; i++) {
                array[i][k] = random.nextInt(bound:100);
                }
            }

            for (int k = 0; k < cols; k++) {
                for (int i = 0 ; i < rows ; i++){
                    totalSum += array[i][k];
                }

            }
        }
        System.out.println(totalSum);
    }
}
```

After: 4.274s

```java
import java.util.Random;

public class After {
    Run | Debug
    public static void main(String[] args) {
    long totalSum = 0;
    for (int p = 0 ; p < 1000 ; p++){
        int rows = 1000;
        int cols = 1000;
        int[][] array = new int[rows][cols];
        Random random = new Random();


        for (int i = 0; i < rows; i++) {
            for (int k = 0; k < cols; k++) {
                array[i][k] = random.nextInt(bound:100);
            }
        }

        for (int i = 0; i < array.length; i++) {
            int[] row = array[i];
            for (int k = 0; k < row.length; k++) {
                totalSum += row[k];
            }
        }
    }
        System.out.println(totalSum);
    }
}
```

그린화 패턴 21

: shift operator than multiplication

- 2를 곱하는 경우, 같은 연산을 하지만 더 단순한 shift연산을 하여 연산량을 줄임

Before: 0.13s

```java
public class Before {
    Run | Debug
    public static void main(String[] args) {
        int result = 10;
        for (int i = 0; i < 1000000; i++) {
            result = result * 2;
        }
    }
}
```

After: 0.11s

```java
public class After {
    Run | Debug
    public static void main(String[] args) {
        int result = 10;
        for (int i = 0; i < 1000000; i++) {
            result = result << 2;
        }
    }
}
```

그린화 패턴 22

: postfix addition than prefix addition

- prefix가 postfix보다 연산량이 더 적다는것을 이용하여 사용, 절감

Before: 0.16s

```java
public class Before {
    Run | Debug
    public static void main(String[] args) {
        int[] array = new int[20000000];

        for (int i = 0; i < array.length; i++) {
            // Perform some operation
            array[i] = 10;
        }

    }
}
```

After: 0.12s

```java
public class After {
    Run | Debug
    public static void main(String[] args) {
        int[] array = new int[20000000];

        for (int i = 0; i < array.length; ++i) {
            // Perform some operation
            array[i] = 10;
        }
    }
}
```

그린화 패턴 23

: iteration than recursion

- 재귀방식의 함수호출은 가독성을 높이지만, call stack을 점유함으로써 더 많은 탄소배출량을 야기함.
- 대부분의 재귀함수(꼬리 재귀)에 대해서 이 최적화가 성립함.

Before: 0.6121s

```java
public class Before {
    public static int factorialRecursive(int n) {
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * factorialRecursive(n - 1);
        }
    }
    public static void main(String[] args){
        int iterations = 10000000;
        for(int i=0; i<iterations; i++){
            factorialRecursive(30);
        }
    }
}
```

After: 0.1758s

```java
public class After {
    public static int factorialIterative(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
    public static void main(String[] args){
        int iterations = 10000000;
        for(int i=0; i<iterations; i++){
            factorialIterative(30);
        }
    }
}
```

그린화 패턴 24

: thread utilization

- 필요한 경우, thread를 적절하게 활용하면 runtime을 크게 감소시킬 수 있음.
- 그만큼 CPU Utilization은 커지지만 runtime감소로 인한 탄소배출량 감소가 더 큼.

Before: 4.0554s

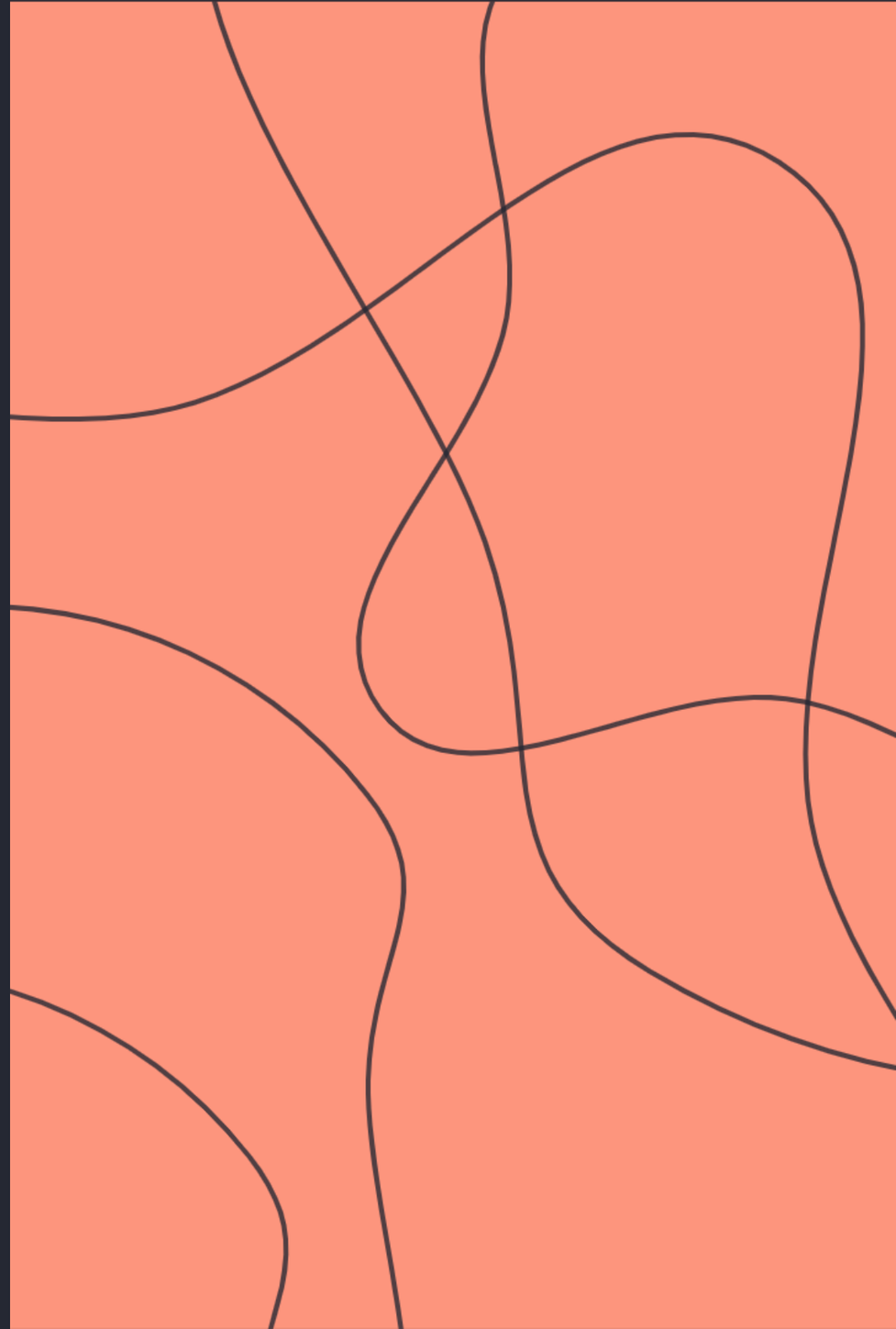
```java
public class Before {
    public static void main(String[] args) {
        int iterations = 1000;
        for (int i=0; i<iterations; i++){
            int result = calculateSum(1, 10000000);
        }
    }

    private static int calculateSum(int start, int end) {
        int sum = 0;
        for (int i = start; i <= end; i++) {
            sum += i;
        }
        return sum;
    }
}
```

After: 0.1904s

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class After {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        ExecutorService executorService = Executors.newFixedThreadPool(5);

        Future<Integer> result1 = executorService.submit(() -> calculateSum(1, 200000));
        Future<Integer> result2 = executorService.submit(() -> calculateSum(200001, 4000000));
        Future<Integer> result3 = executorService.submit(() -> calculateSum(400001, 6000000));
        Future<Integer> result4 = executorService.submit(() -> calculateSum(600001, 8000000));
        Future<Integer> result5 = executorService.submit(() -> calculateSum(800001, 10000000));


        int iterations = 1000;
        try {
            for (int i=0; i<iterations; i++){
                int result1Value = result1.get();
                int result2Value = result2.get();
                int result3Value = result3.get();
                int result4Value = result4.get();
                int result5Value = result5.get();

                int result = result1Value + result2Value + result3Value + result4Value + result5Value;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            executorService.shutdown();
        }
    }
```

Better API

패턴 25. use sublist deletion
패턴 26. use parallel sort
패턴 27. use DecimalFormat
패턴 28. do not use StringTokenizer
패턴 29. charAt than getBytes
패턴 30. stream is not always good 1
패턴 31. stream is not always good 2
패턴 32. regex is not always good



그린화 패턴 25

: use sublist deletion

- ArrayList에서 sublist 메서드를 호출하게 되면 새로운 ArrayList를 생성하는 것이 아니라 해당 List의 뷰를 제공하기 때문에 새로운 객체를 생성하는 비용을 줄여주며, clear() 메서드 또한 내부적으로 최적화되게 구현되어 있기 때문에 ArrayList의 크기가 큰 경우에 반복문보다 효율적으로 구간 삭제를 진행할 수 있음.

Before: 1.785s

```java
import java.util.ArrayList;
import java.util.List;

public class Before {
    private static String generateData() {
        return "Some data";
    }

    Run | Debug
    public static void main(String[] args) {
    List<String> data = new ArrayList<>();
    int i = 0;int j = 0;

    while ( i < 200000) {
        data.add(generateData());
        i++;
    }

    while(j < 100000) {
        data.remove(index:0);
        j++;
    }
    }
}
```

After: 0.005s

```java
import java.util.ArrayList;
import java.util.List;

public class After {
    private static String generateData() {
        return "Some data";
    }

    Run | Debug
    public static void main(String[] args) {
    List<String> data = new ArrayList<>();
    int i = 0;

    while (i < 200000) {
        data.add(generateData());
        i++;
    }

    int startIndex = 0;
    int endIndex = 100000;
    data.subList(startIndex, endIndex + 1).clear();
    }
}
```

그린화 패턴 26

: use parallel sort

- Arrays.sort() 메서드의 경우에는 단순처리를 통해서 array의 값을 정렬하기 때문에 실행시간이 길어지게 됨.
- 그러므로 Arrays.parallelSort() 메서드를 통해서 병렬처리를 이용하게 되면 보다 실행시간을 줄어들게 만들 수 있음.

Before: 65.49s

```java
import java.util.Arrays;
import java.util.Random;

public class Before {

    Run | Debug
    public static void main(String[] args) {
        for (int k = 0 ; k < 1000 ; k++){
            int[] data = new int[1000000];
            Random random = new Random();

        for (int i = 0; i < data.length; i++) {
            data[i] = random.nextInt(bound:1000000);
        }
        Arrays.sort(data);
        }
    }
}
```

After: 11.789s

```java
import java.util.Arrays;
import java.util.Random;

public class After {

    Run | Debug
    public static void main(String[] args) {
        for (int k = 0 ; k < 1000 ; k++){
            int[] data = new int[1000000];
            Random random = new Random();


            for (int i = 0; i < data.length; i++) {
                data[i] = random.nextInt(bound:1000000);
            }
            Arrays.parallelSort(data);
        }
    }
}
```

그린화 패턴 27

: use decimalFormat

- String.format은 general한 사용이 가능하지만, 정수 formatting에 최적화 되어있지 않음.
- 정수를 formatting할 때는 DecimalFomat을 사용하는 것이 유리.

Before: 3.5226s

```java
public class Before {
    public static void main(String[] args){
        int iterations = 1000000;
        for(int i=0 ; i<iterations ; i++){
            String.format("%,6d", i);
        }
    }
}
```

After: 0.7364s

```java
import java.text.DecimalFormat;

public class After {
    public static void main(String[] args){
        DecimalFormat df = new DecimalFormat("#,###");
        int iterations = 1000000;
        for(int i=0 ; i<iterations ; i++){
            df.format(i);
        }
    }
}
```

그린화 패턴 28

: do not use StringTokenizer

- StringTokenizer는 JDK 1.0부터 존재한 패키지이기에 최적화가 되지 않았음.
- substring등 기본 메소드를 쓰는 것이 자원을 아낄 수 있음.

Before: 0.3962s

```java
import java.util.StringTokenizer;

public class Before {
    public static void main(String[] args) {
        String input = "apple,orange,banana,grape";

        for (int i = 0; i < 1000000; i++) {
            StringTokenizer tokenizer = new StringTokenizer(input, ",");

            while (tokenizer.hasMoreTokens()) {
                String token = tokenizer.nextToken();
            }
        }
    }
}
```

After: 0.2908s

```java
public class After {
    public static void main(String[] args) {
        String input = "apple,orange,banana,grape";

        for (int i = 0; i < 1000000; i++) {
            int startIndex = 0;
            int endIndex;

            while ((endIndex = input.indexOf(',', startIndex)) != -1) {
                String token = input.substring(startIndex, endIndex);
                startIndex = endIndex + 1;
            }

            String lastToken = input.substring(startIndex);
        }
    }
}
```

그린화 패턴 29

: charAt than getBytes

- 문자열에서 문자 하나를 가져와야할 때, 문자가 ASCII 코드라면 getBytes()보다 charAt()을 사용해야 연산량이 감소함.
- getBytes는 char 배열을 byte 배열로 바꾸고, 문자 type에 대한 처리도 요구하는 무거운 메소드임.

Before: 0.5077s

```java
public class Before {
    public static void main(String[] args) {
        String text = "Hello, World!";

        int iterations = 1000000;
        for (int i = 0; i < iterations; i++) {
            char c = (char) text.getBytes()[0];
        }
    }
}
```

After: 0.1859s

```java
public class After {
    public static void main(String[] args) {
        String text = "Hello, World!";

        int iterations = 1000000;
        for (int i = 0; i < iterations; i++) {
            char c = text.charAt(0);
        }

    }
}
```

그린화 패턴 30

: stream is not always good 1

- Stream 패키지는 코드의 가독성을 높이기 위해 JAVA에서 지원하는 선언적 프로그래밍 방식의 패키지임.
- 그러나 Stream API는 내부적으로 많은 객체를 생성하기 때문에, 단순히 배열 초기화가 목적이라면 for-loop방식이 권장됨.

Before: 7.1179s

```java
import java.util.Arrays;
import java.util.stream.IntStream;
public class Before {
    public static void main(String args[]) {
        int iterations = 100000;
        int intArray[] = new int[iterations];
        for(int i=1; i<iterations; i++){
            intArray = IntStream.range(0, iterations).toArray();
            // System.out.println(intArray[0]);
        }
    }
}
```

After: 4.1197s

```java
import java.util.Arrays;
public class After {
    public static void main(String[] args){
        int iterations = 100000;
        int intArray[] = new int[iterations];
        for(int i=1; i<iterations; i++){
            for(int index = 0; index < iterations; index++) {
                intArray[index] = index;
            }
            // System.out.println(intArray[0]);
        }
    }
}
```

그린화 패턴 31

: stream is not always good 2

- 데이터의 양이 작거나 계산이 경량화 된 경우에는 병렬처리의 오버헤드가 실제 연산의 비용보다 더 커질 수 있음.
- 그러므로 단순한 계산을 진행할 때에는 반복문을 통한 순차계산이 오히려 더 계산량을 줄여줄 수 있음.

Before: 30.454s

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;



public class Before {

    Run | Debug
    public static void main(String[] args) {
        for (int k = 0 ; k < 1000 ; k++){
            List<Integer> numbers = IntStream.rangeClosed(startInclusive:1,endInclusive:1000000)
                .boxed()
                .collect(Collectors.toList());

        List<Integer> evenSquares = numbers.stream()
                .filter(number -> number % 2 == 0)
                .map(number -> number * number)
                .collect(Collectors.toList());
        evenSquares.stream().limit(maxSize:10).forEach(System.out::println);
        }
    }
}
```

After: 25.643s

```java
import java.util.ArrayList;
import java.util.List;

public class After {

    Run | Debug
    public static void main(String[] args) {
        for (int k = 0 ; k < 1000 ; k++){
            List<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= 1000000; i++) {
            numbers.add(i);
        }

        List<Integer> evenSquares = new ArrayList<>();
        for (int number : numbers) {
            if (number % 2 == 0) {
                evenSquares.add(number * number);
            }
        }

        for (int i = 0; i < 10; i++) {
            System.out.println(evenSquares.get(i));
        }
        }
    }
}
```

그린화 패턴 32

: regex is not always good

- regex는 내부적으로 많은 객체를 생성함.
- 따라서 단순한 문자열을 parsing할 때는 오히려 if condition을 사용한 parsing이 효과적임.

Before: 10.19s

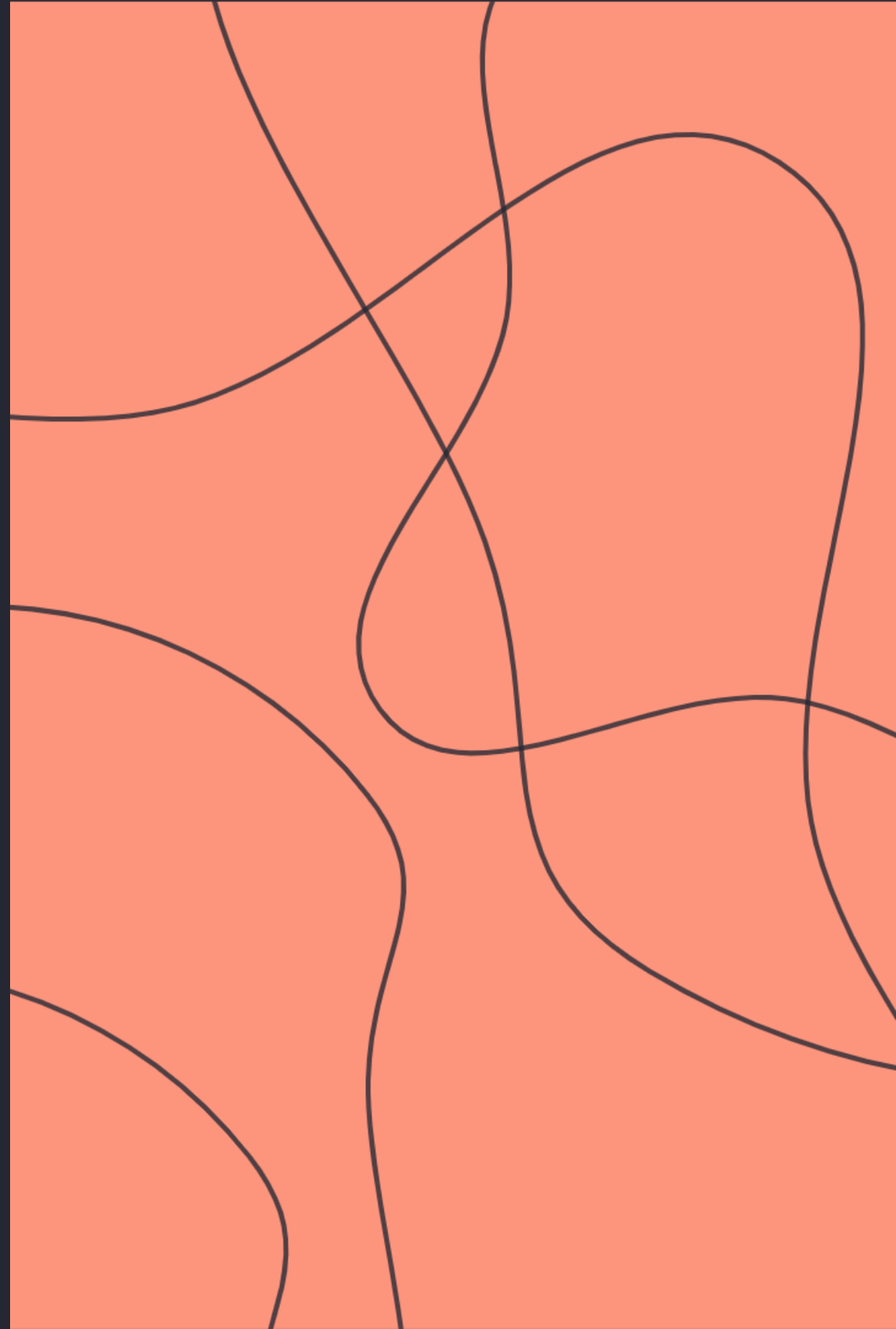
```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Before {
    public static String findDomain(String email) {
        String pattern = "@(.+)";
        Pattern r = Pattern.compile(pattern);
        Matcher m = r.matcher(email);
        if(m.find()){
            return m.group(1);
        } else {
            return "";
        }
    }
    public static void main(String[] args) {
        String email = "example.email@domain.com";

        int iterations = 10000000;
        for(int i=0; i<iterations; i++){
            String domain = findDomain(email);
        }
    }
}
```

After: 9.12s

```java
public class After {
    public static String findDomain(String email) {
        int atPosition = -1;
        for (int i = 0; i < email.length(); i++) {
            if (email.charAt(i) == '@') {
                atPosition = i;
                break;
            }
        }
        if (atPosition != -1) {
            return email.substring(atPosition + 1);
        } else {
            return "";
        }
    }
    public static void main(String[] args){
        String email = "example.email@domain.com";

        int iterations = 10000000;
        for(int i=0; i<iterations; i++){
            String domain = findDomain(email);
        }
    }
}
```

Better Data Structure

패턴 33. HashMap than TreeMap
패턴 34. linked list than array
패턴 35. literal type than object type
패턴 36. integer than float
패턴 37. use integer flag



그린화 패턴 33

: HashMap than TreeMap

- HashMap 데이터구조는 내부에서 해시 테이블을 사용해 데이터를 저장하기 때문에 데이터 구조의 변경이 잦을 때에는 TreeMap 보다 런타임을 줄일 수 있다
- 이는 저장하려는 데이터의 양이 클수록 효과가 크다.

Before: 24.436s

```java
import java.util.Map;
import java.util.TreeMap;

public class Before {
    Run | Debug
    public static void main(String[] args) {

    for (int i = 0 ; i < 100000 ; i++){
                Map<String, Integer> treeMap = new TreeMap<>();
                treeMap.put(key:"Alice", value:25);
                treeMap.put(key:"Bob", value:30);
                treeMap.put(key:"Eve", value:22);

                for (Map.Entry<String, Integer> entry : treeMap.entrySet()) {
                    System.out.println(entry.getKey() + ": " + entry.getValue());
                }
            }
        }
}
```

After: 22.829s

```java
import java.util.HashMap;
import java.util.Map;


public class After {
    Run | Debug
    public static void main(String[] args) {

    for (int i = 0 ; i < 100000 ; i++){
                Map<String, Integer> hashMap = new HashMap<>();
                hashMap.put(key:"Alice", value:25);
                hashMap.put(key:"Bob", value:30);
                hashMap.put(key:"Eve", value:22);


                for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {
                    System.out.println(entry.getKey() + ": " + entry.getValue());
                }
            }
        }
}
```

그린화 패턴 34

linked list than array

- 선형 자료구조에서 삭제가 많은 상황에서는, 배열 대신 연결리스트를 사용하여 삭제 시에 복사 연산 피함

Before: 0.5s

```java
public class Before {
    Run | Debug
    public static void main(String[] args){
        ArrayList<Integer> v = new ArrayList<Integer>();
        for(int i=0; i<100000; i++) {
            v.add(e:1);
        }
        for(int i=0; i<100000; i++) {
            v.remove(index:0);
        }
    }
}
```

After: 0.01s

```java
public class After {
    Run | Debug
    public static void main(String[] args){
        LinkedList<Integer> l = new LinkedList<Integer>();
        for(int i=0; i<100000; i++) {
            l.add(e:1);
        }
        for(int i=0; i<100000; i++) {
            l.removeFirst();
        }
    }
}
```

그린화 패턴 35

: literal type than object type

- 동적 할당이 필요 없는 literal 변수를 객체로 선언하는 것은 불필요함.

Before: 1.5833s

```
public class Before {
    public static void main(String[] args){
        String s1[] = new String[100];
        Boolean b1[] = new Boolean[100];

        int iterations = 1000000;
        for(int i=0; i<iterations; i++){
            for(int j=0; j<100; j++){
                s1[j] = new String("hello");
            }
            for(int j=0; j<100; j++){
                b1[j] = new Boolean(true);
            }
        }
    }
}
```

After: 0.4630s

```
public class After {
    public static void main(String[] args){
        String[] s1 = new String[100];
        Boolean[] b1 = new Boolean[100];

        int iterations = 1000000;
        for(int i=0; i<iterations; i++){
            for(int j=0; j<100; j++){
                s1[j] = "hello";
            }
            for(int j=0; j<100; j++){
                b1[j] = Boolean.valueOf(true);
            }
        }
    }
}
```

그린화 패턴 36

: integer than float

- 부동소수점 연산은 정수 연산보다 더 많은 computational cost를 요구함.
- 받아온 float/double data가 정수형이 돼도 상관없을 때는 정수로 casting하는 것이 좋음.

Before: 0.6317

```java
public class Before {
    public static void main(String[] args){
        int iterations = 100000;

        double test= 100 / 3;
        int output = 0;
        for (int i = 0; i < iterations; i++) {
            for (int j = 0; j < iterations; j++) {
                int res = (int)test + 30;
                int res2 = res + 50;
                if(res2 > 100){
                    res2 += 90;
                }
                output += (int)res2;
            }
        }
    }
}
```

After: 0.3839s

```java
public class After {
    public static void main(String[] args){
        int iterations = 100000;

        double test= 100 / 3;
        int output = 0;
        for (int i = 0; i < iterations; i++) {
            for (int j = 0; j < iterations; j++) {
                double res = test + 30.0;
                double res2 = res + 50.0;
                if(res2 > 100.0){
                    res2 += 90.0;
                }
                output += (int)res2;
            }
        }
    }
}
```

그린화 패턴 37

: use integer flag

- flag를 설정할 때는 flag가 지닌 의미를 explicit하게 문자열로 나타내기 보다는 대응하는 정수로 나타내는 것이 좋음.
- 아래 예시에서는 “gender_female”: 4, “gender_male”: 3으로 설정했음.

Before: 1.4984s

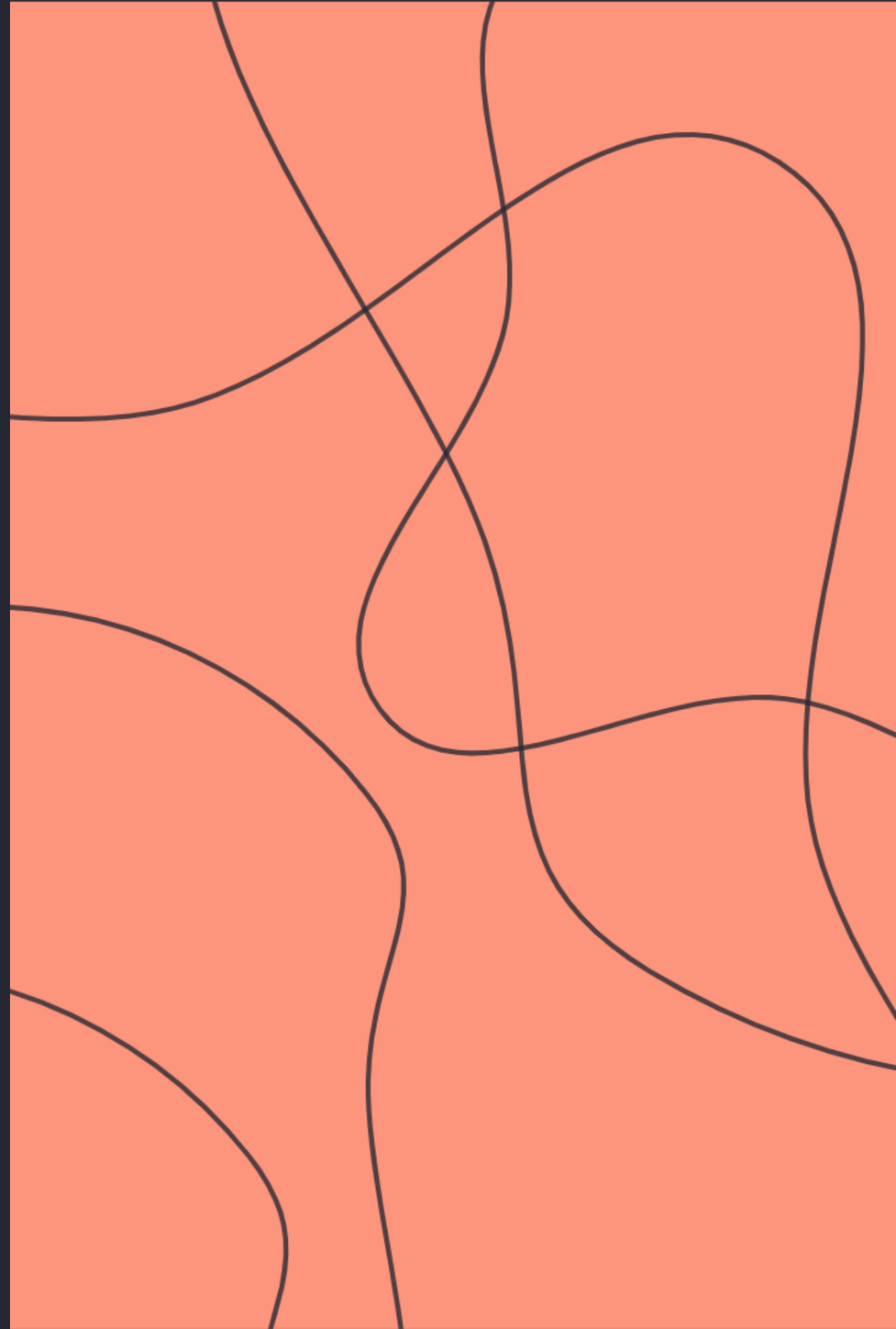
```java
public class Before {
    public static String testFirstNumSSN(int a){
        if(a == 0){
            return "gender_female";           //this means male
        }else{
            return "gender_male";           //this means female
        }
    }
    public static void main(String[] args){
        long iterations = 2000000000;
        int a = 0;
        String result = "";
        result = testFirstNumSSN(a);
        for(long i=0; i<iterations; i++){
            if(result == "gender_female"){
                // System.out.println("this is female");
            }else if(result == "gender_male"){
                // System.out.println("this is male");
            }
        }
    }
}
```

After: 0.8138s

```java
public class After {
    public static int testFirstNumSSN(int a){
        if(a == 0){
            return 4;           //this means female
        }else{
            return 3;           //this means male
        }
    }
    public static void main(String[] args){
        long iterations = 2000000000;
        int a = 0;
        int result = 0;
        result = testFirstNumSSN(a);
        for(long i=0; i<iterations; i++){
            if(result==4){
                // System.out.println("this is female");
            }else if(result==3){
                // System.out.println("this is male");
            }
        }
    }
}
```

Save Memory Resources

패턴 38. setting initial capacity for collections
패턴 39. free obsolete string
패턴 40. free obsolete list
패턴 41. free obsolete stack item
패턴 42. free obsolete thread
패턴 43. inline functions than fine-grained functions
패턴 44. unnecessary parameters



그린화 패턴 38

: set initial capacity for collections

- 리스트 용량을 초기에 설정함으로써 임의의 메모리 할당을 예방하고, 추가적인 메모리 사용을 방지하여 메모리 사용량을 줄임

Before: 0.15s

```java
import java.util.ArrayList;
import java.util.List;

public class Before {
    Run | Debug
    public static void main(String[] args) {
        int n = 1000000;


        List<Integer> list1 = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list1.add(i);
        }

    }
}
```

After: 0.12s

```java
import java.util.ArrayList;
import java.util.List;

public class After {
    Run | Debug
    public static void main(String[] args) {
        int n = 1000000;


        List<Integer> list2 = new ArrayList<>(n);
        for (int i = 0; i < n; i++) {
            list2.add(i);
        }

    }
}
```

그린화 패턴 39

: free obsolete string

- String 객체를 반복적으로 생성하지 않도록 만들어 새로운 객체를 생성하는 비용을 줄임
- 이후 객체의 참조를 제거하여 가비지 컬렉션의 객체가 되게하여 객체를 회수하고 메모리의 성능을 높임.

Before (memory wasted)

```java
public class Before {
    Run | Debug
    public static void main(String[] args) {
    for (int i = 0; i < 100000; i++) {

        String myString = new String("Object " + i);
        System.out.println(myString);
    }
    }
}
```

After (memory saved)

```java
public class After {

    Run | Debug
    public static void main(String[] args) {
    for (int i = 0; i < 100000; i++) {

        String myString2 = "Object " + i;
        System.out.println(myString2);

        myString2 = null;
    }
    }
}
```

그린화 패턴 40

: free obsolete list

- Map과 List에서 할당된 item들은 clear 메서드를 호출하면 GC에 의해 메모리에서 제거됨.
- 그러나, Map과 List자체는 GC에 의해 처리되지 않기 때문에 명시적으로 free해줘야 함.

Before (memory leaked)

```java
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
import java.util.List;

public class Before {
    public static void main(String[] args){
        List<Object> list = new ArrayList<>();
        Map<String, Object> map = new HashMap<>();
        list.clear();
        map.clear();
    }
}
```

After (memory saved)

```java
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
import java.util.List;

public class After {
    public static void main(String[] args){
        List<Object> list = new ArrayList<>();
        Map<String, Object> map = new HashMap<>();
        list.clear();
        map.clear();
        list = null;
        map = null;
    }
}
```

그린화 패턴 41

: free obsolete stack item

- Stack 자료구조를 사용할 때, stack의 크기를 감소시켜서 pop을 구현하면, pop된 item이 메모리에서 제거되지 않음.
- GC에 의해 메모리에서 제거될 수 있도록 명시적으로 free시켜줘야 함.

Before (memory leaked)

```
public class Before {
    public static int size;
    public static Object pop(String[] stack) {
        if (size==0) {
            return -1;
        }
        return stack[--size];
    }
    public static void main(String[] args){
        String a[] = new String[10000001];
        size = a.length;
        for(int index = 0; index < size; index++) {
            a[index] = "hi";
        }

        for(int i=0; i<size; i++){
            pop(a);
        }
    }
}
```

After (memory saved)

```
public class After {
    public static int size;
    public static Object pop(String[] stack) {
        if (size==0) {
            return -1;
        }
        Object result = stack[--size];
        stack[size] = null;
        return result;
    }
    public static void main(String[] args){
        String a[] = new String[10000001];
        size = a.length;
        for(int index = 0; index < size; index++) {
            a[index] = "hi";
        }

        for(int i=0; i<size; i++){
            pop(a);
        }
    }
}
```

그린화 패턴 42

: free obsolete thread

- thread가 작업을 끝내고 main thread로 합류해도 해당 thread는 memory에서 제거되지 않음.
- GC에 의해 메모리에서 제거될 수 있도록 명시적으로 free시켜줘야 함.

Before (memory leaked)

```
public class Before{
    public static void main(String[] args){
        Object resource = new Object();
        Runnable myRunnable = new MyRunnable(resource);
        Thread thread = new Thread(myRunnable);
        thread.start();
    }
}
class MyRunnable implements Runnable {
    private Object someResource;

    public MyRunnable(Object someResource) {
        this.someResource = someResource;
    }
    public void run(){
        System.out.println(someResource);
    }
}
```

After (memory saved)

```
public class After{
    public static void main(String[] args){
        Object resource = new Object();
        Runnable myRunnable = new MyRunnable(resource);
        Thread thread = new Thread(myRunnable);
        thread.start();

        resource = null;
        myRunnable = null;
        thread = null;
    }
}
class MyRunnable implements Runnable {
    private Object someResource;

    public MyRunnable(Object someResource) {
        this.someResource = someResource;
    }
    public void run(){
        System.out.println(someResource);
    }
}
```

그린화 패턴 43

: inline functions than fine-grained functions

- 함수를 너무 많이 쪼개면 가독성을 해칠 뿐만 아니라, call stack을 많이 점유하여 더 많은 탄소배출량을 야기함.

Before (memory wasted)

```java
public class Before {
    public static void main(String[] args) {
        int iterations = 2147483647;
        int[] numbers = {1, 2};
        for (int k = 0; k < iterations; k++ ){
            average(numbers);
        }
    }
    private static int sumAll(int[] array, int len) {
        int sum = 0;
        for (int i = 0; i < len; i++) {
            sum += array[i];
        }
        return sum;
    }
    private static int divideBySize(int sum, int len) {
        return sum/len;
    }
    private static int average(int[] array) {
        int sum = 0;
        int len = array.length;
        sum = sumAll(array, len);
        return divideBySize(sum, len);
    }
}
```

After (memory saved)

```java
public class After {
    public static void main(String[] args) {
        int iterations = 2147483647;
        int[] numbers = {1, 2};
        for (int k = 0; k < iterations; k++){
            average(numbers);
        }
    }

    private static int average(int[] array) {
        int sum = 0;
        int len = array.length;
        for (int i = 0; i < len; i++) {
            sum += array[i];
        }
        return sum/len;
    }
}
```

그린화 패턴 44

: unnecessary parameters

- 함수를 선언할 때, 불필요하게 파라미터를 많이 설정하면 함수 호출 시 메모리가 낭비됨.

Before (memory wasted)

```java
public class Before {
    public static void main(String[] args) {
        int iterations = 1000000;
        int[] numbers = {1, 2, 3, 4, 5};
        for (int k = 0; k < iterations; k++ ){
            calculateSum(numbers, numbers.length);
        }
    }

    private static int calculateSum(int[] array, int length) {
        int sum = 0;
        for (int i = 0; i < length; i++) {
            sum += array[i];
        }
        return sum;
    }
}
```

After (memory saved)

```java
public class After {
    public static void main(String[] args) {
        int iterations = 1000000;
        int[] numbers = {1, 2, 3, 4, 5};
        for (int k = 0; k < iterations; k++ ){
            calculateSum(numbers);
        }
    }

    private static int calculateSum(int[] array) {
        int sum = 0;
        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }
        return sum;
    }
}
```