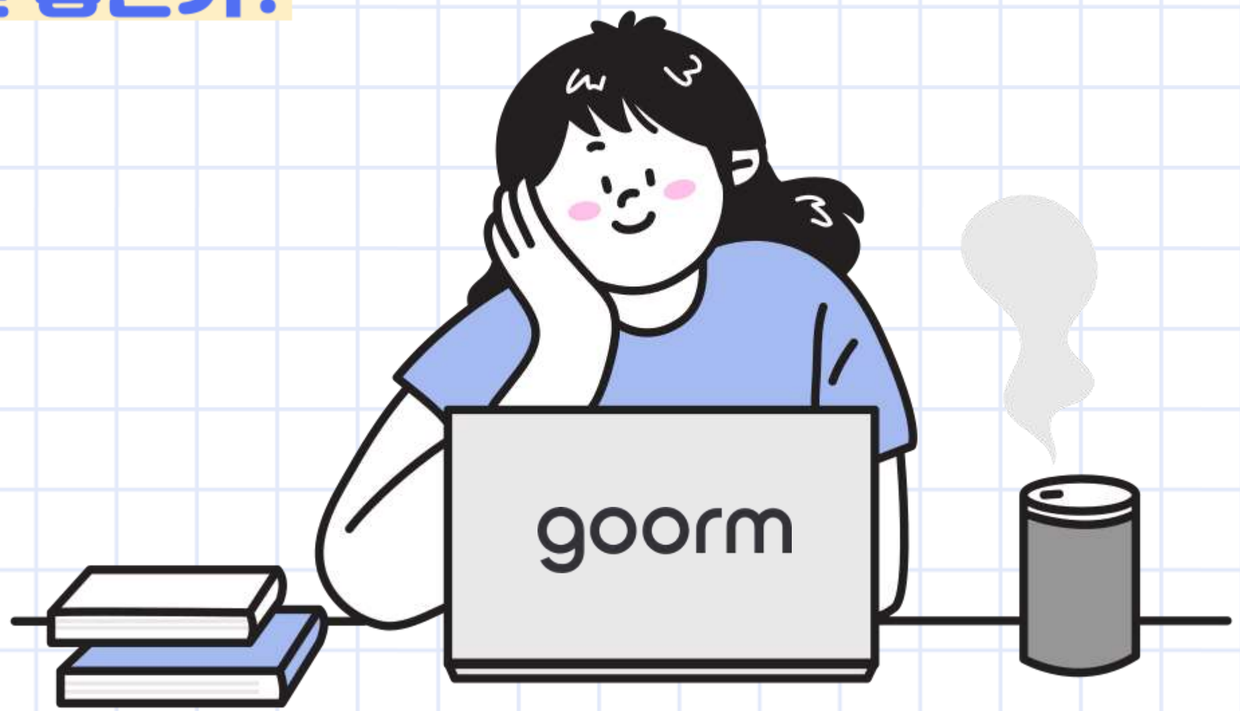
Effective Java : item 6

불필요한 객체 생성을 피하라

goorm

CONTENTS

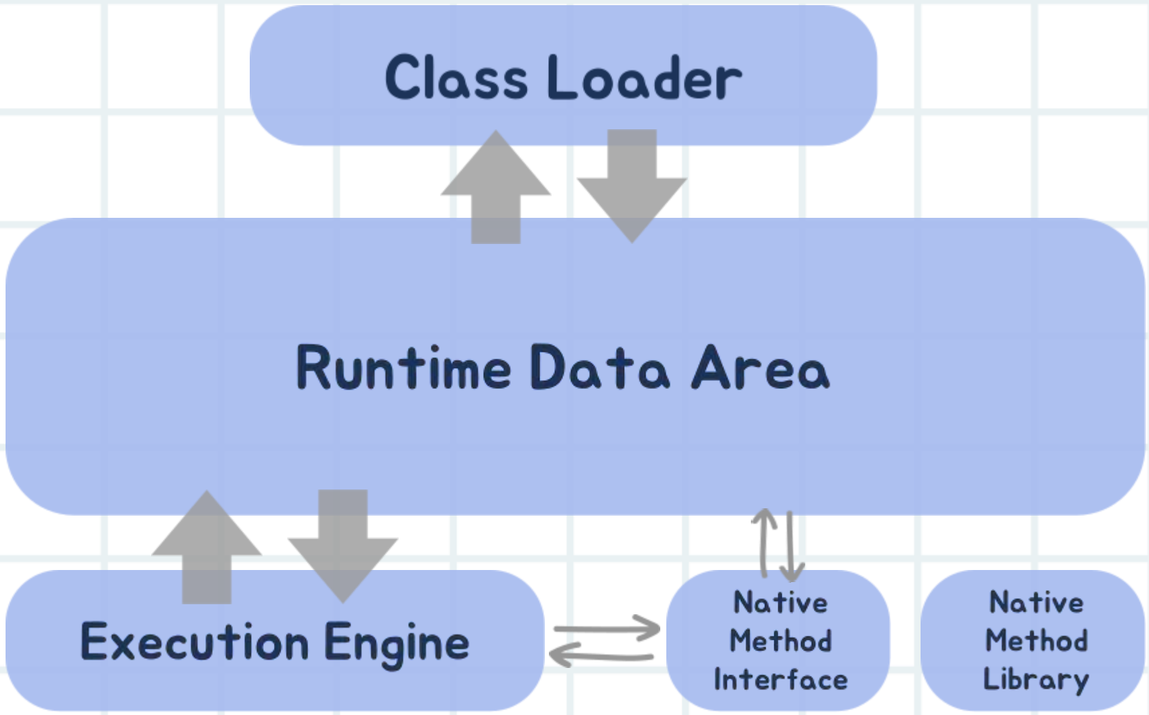
- 객체와 메모리
- 불필요한 객체 생성은 왜 안 좋은가?
- 예시
  - Case 1
  - Case 2
  - Case 3
  - Case 4



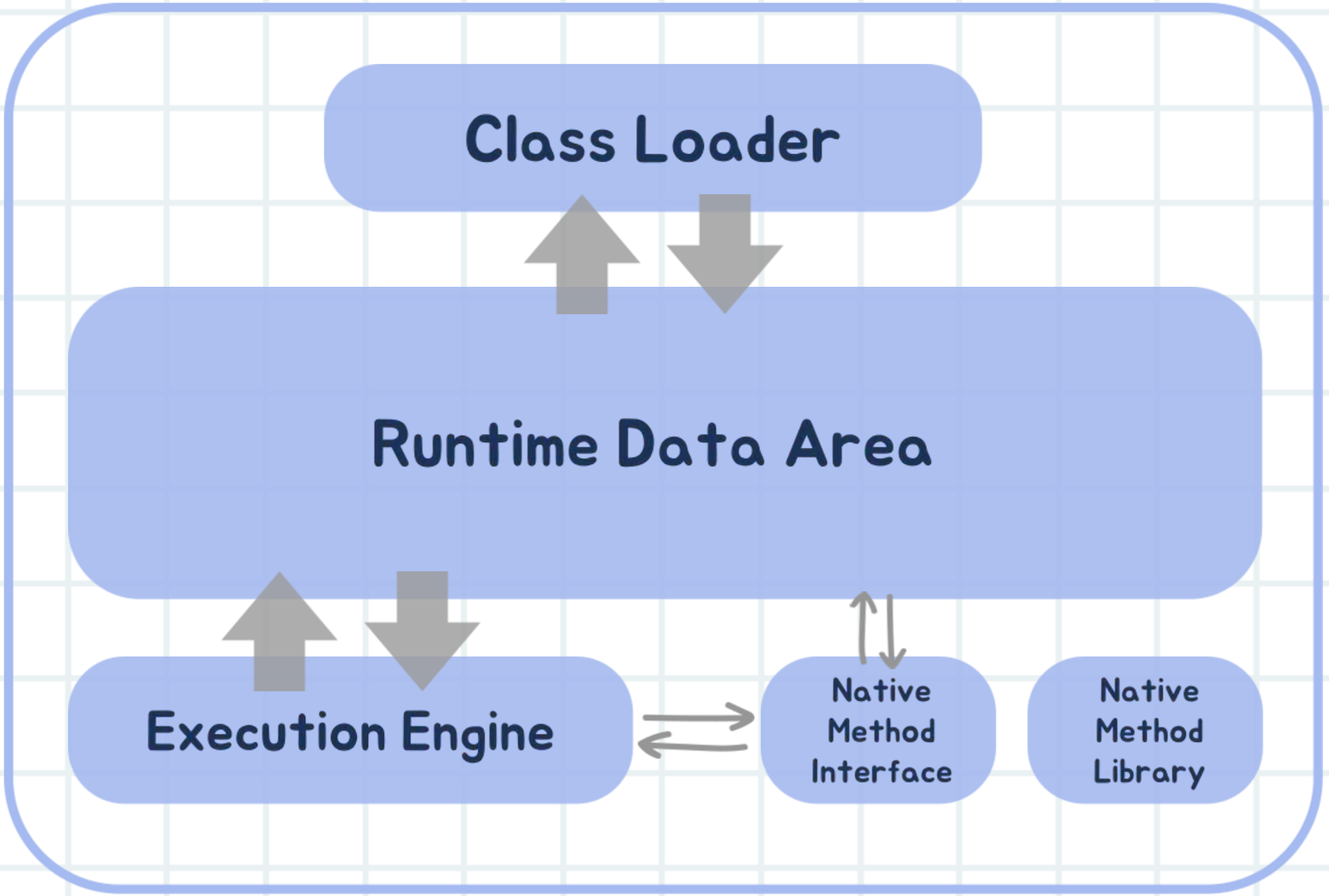
JVM이란?

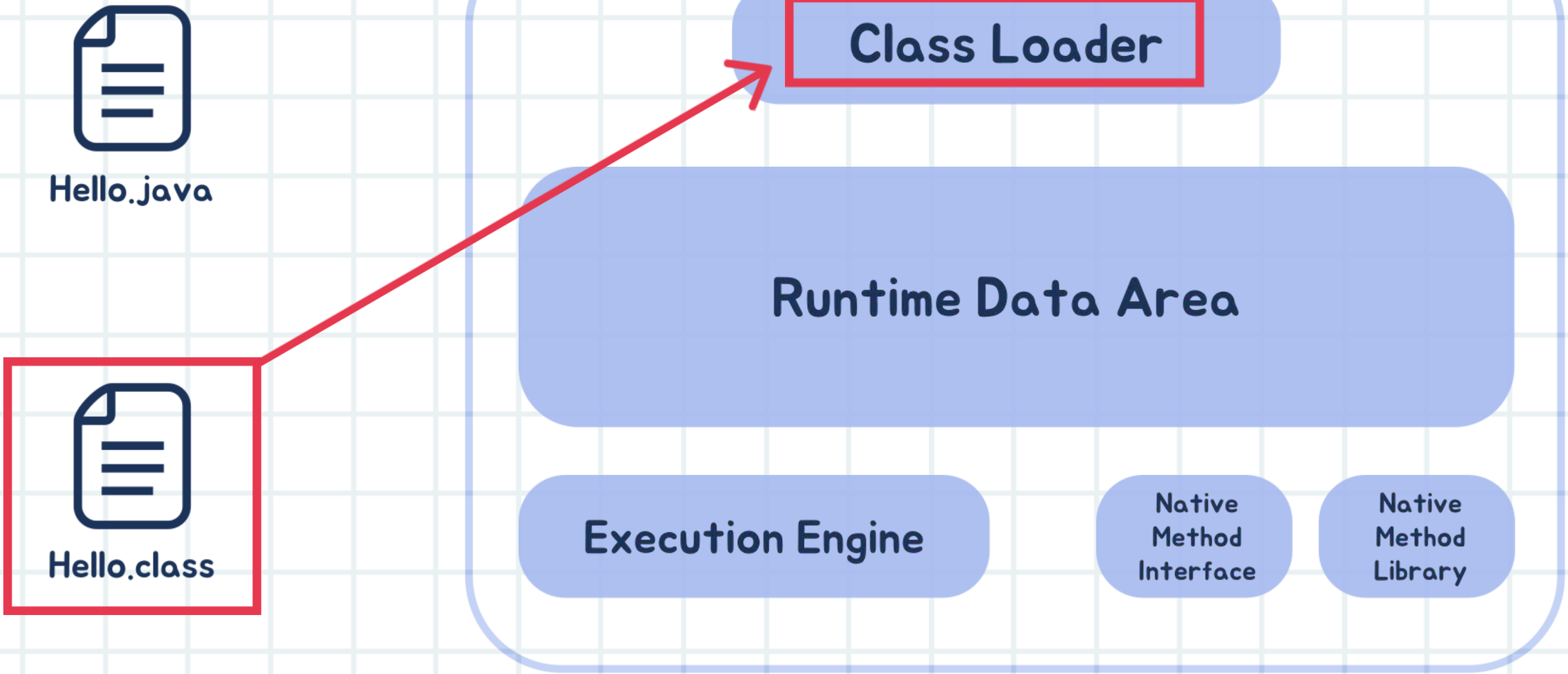
- Java Virtual Machine
- 자바 어플리케이션을 어느 CPU나 OS에서도 잘 작동하도록 도와주는 역할 수행

구조



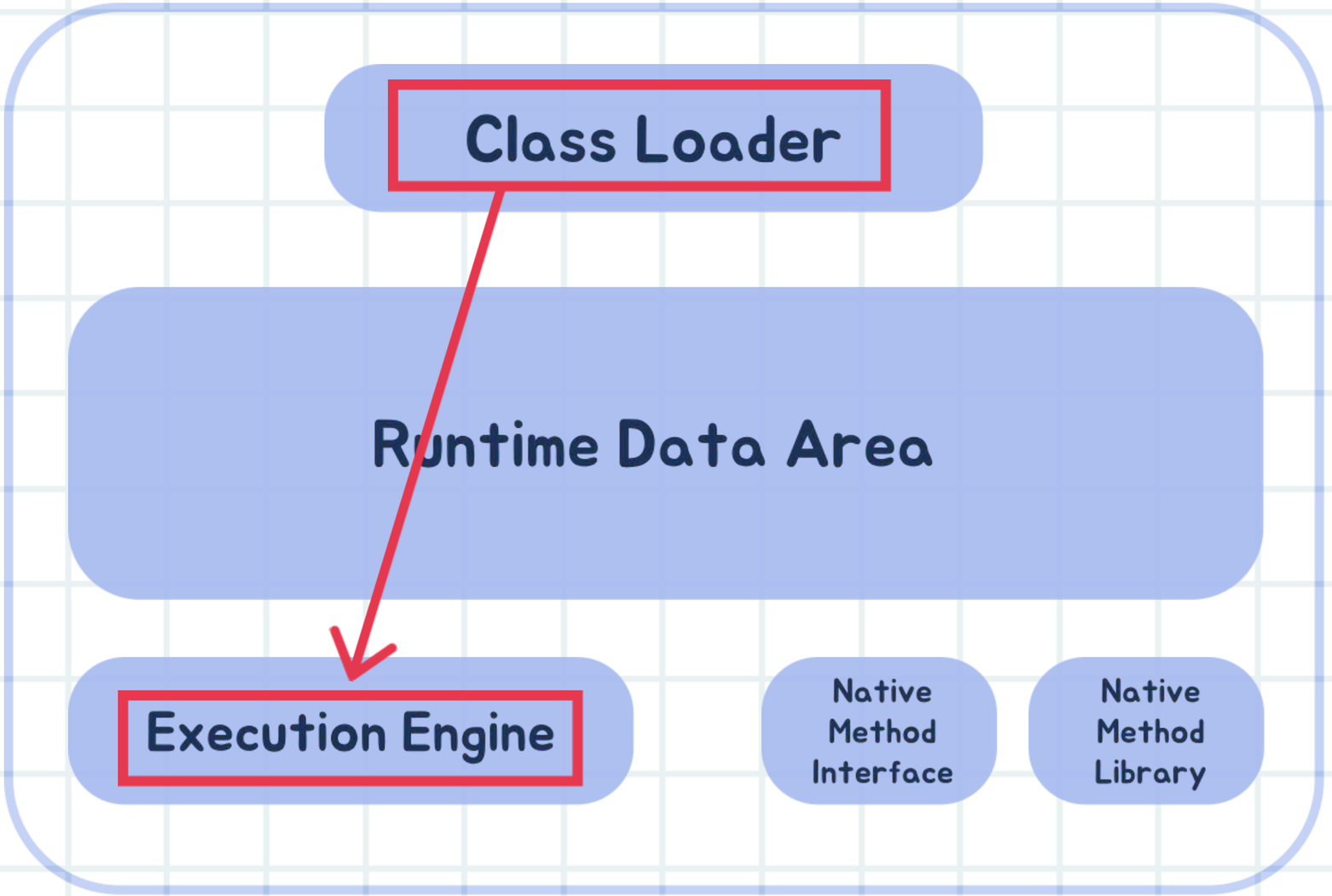

Hello.java
compile
Hello.class


Class Loader
Runtime Data Area
Execution Engine
Native Method Interface
Native Method Library


Hello.java
Hello.class
Class Loader
Runtime Data Area
Execution Engine
Native Method Interface
Native Method Library

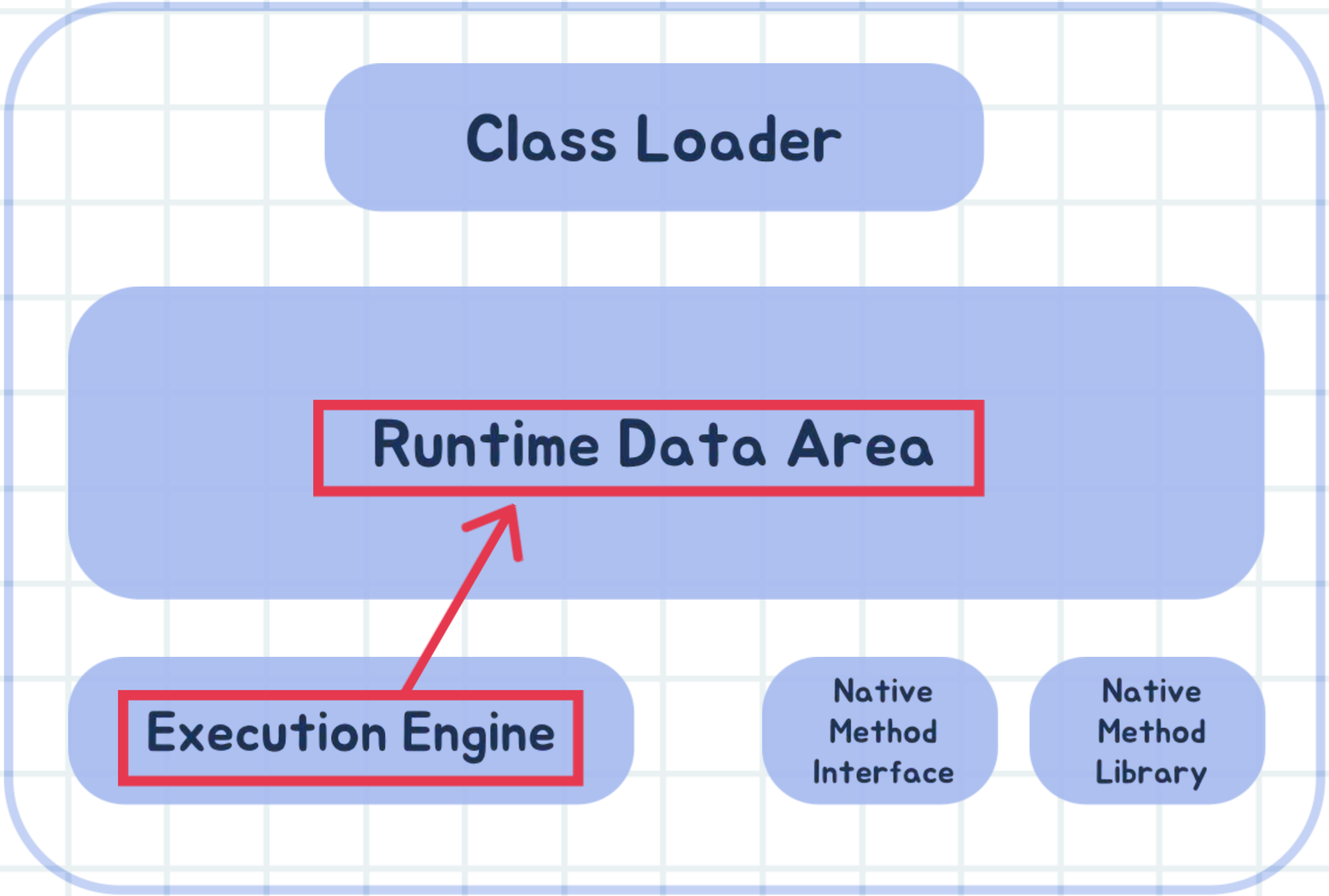
Hello.java

Hello.class



Hello.java

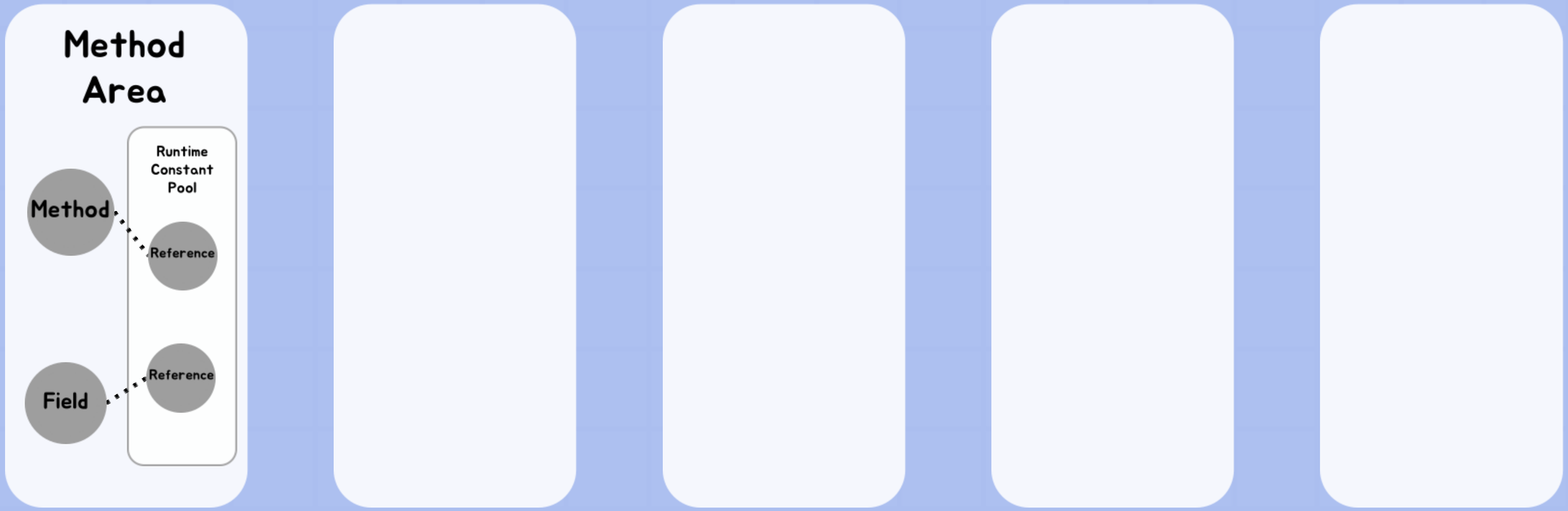
Hello.class



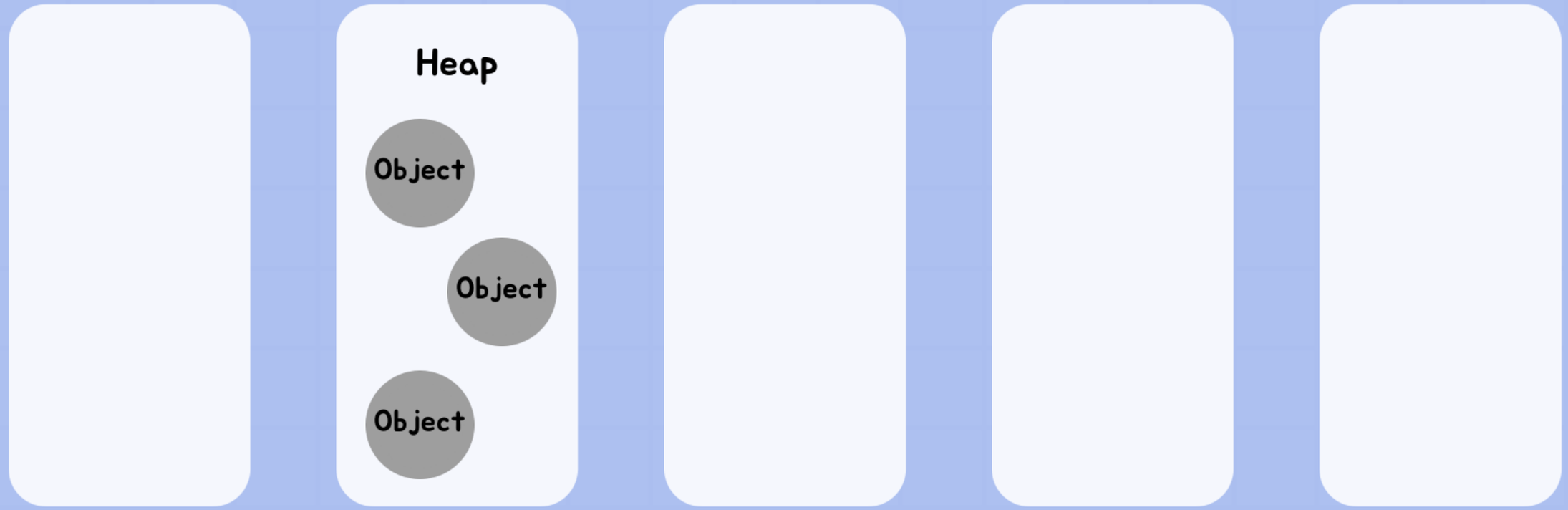
Runtime Data Area

- Method Area
- Heap
- JVM Stack
- PC Registers
- Native Method Stack

Runtime Data Area



Runtime Data Area



객체 생성은

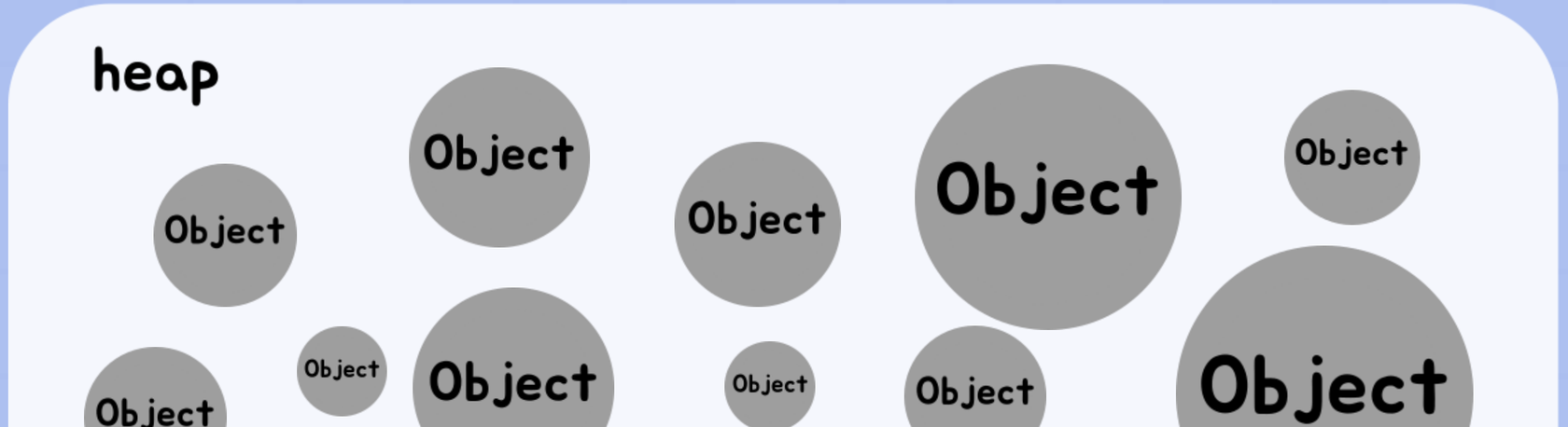
메모리를 요구한다.

불필요한 객체 생성은 왜 안 좋은가?

1. 메모리 사용량 증가

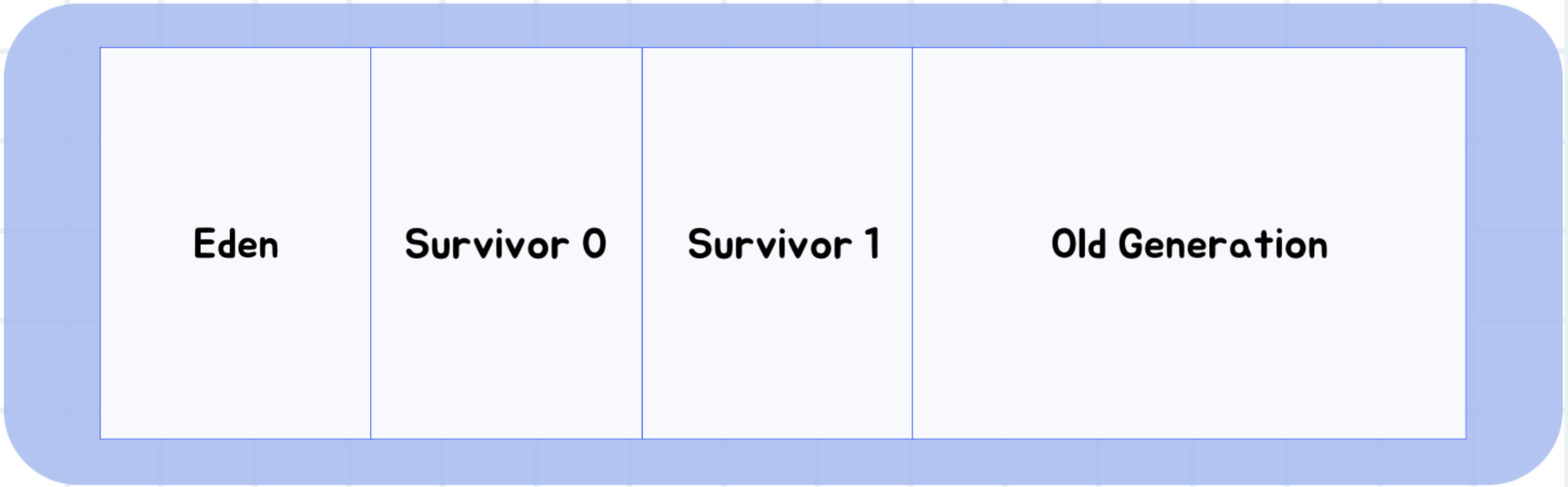
- 객체 생성시, Runtime Data Area의 Heap 영역 객체 저장.
- 불필요한 객체 생성 → 불필요한 heap 영역 사용 → 불필요한 메모리 사용
- OutofMemoryError

Runtime Data Area

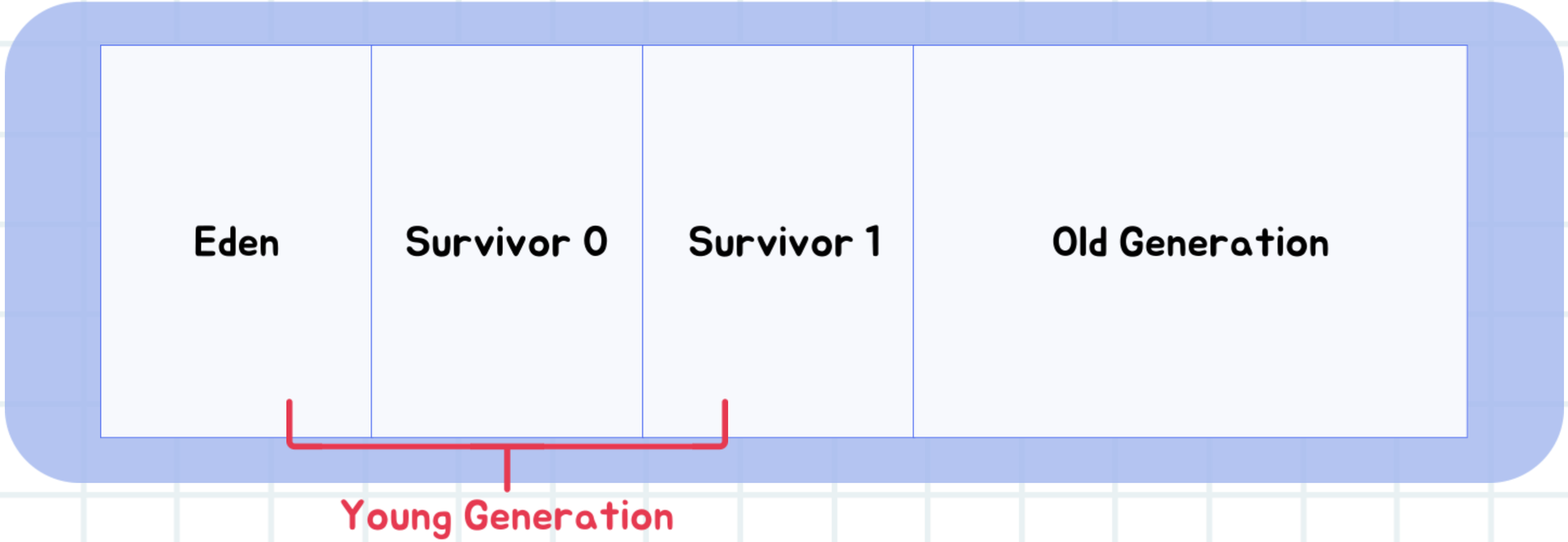


2. Stop-the-world로 인한 지연

- Garbage Collection(GC) 이란?
  - Heap 영역에서 사용되지 않는 객체들을 주기적으로 삭제하는 프로세스
  - JVM 속 Execution Engine 내부에 존재
- Heap 영역 구조

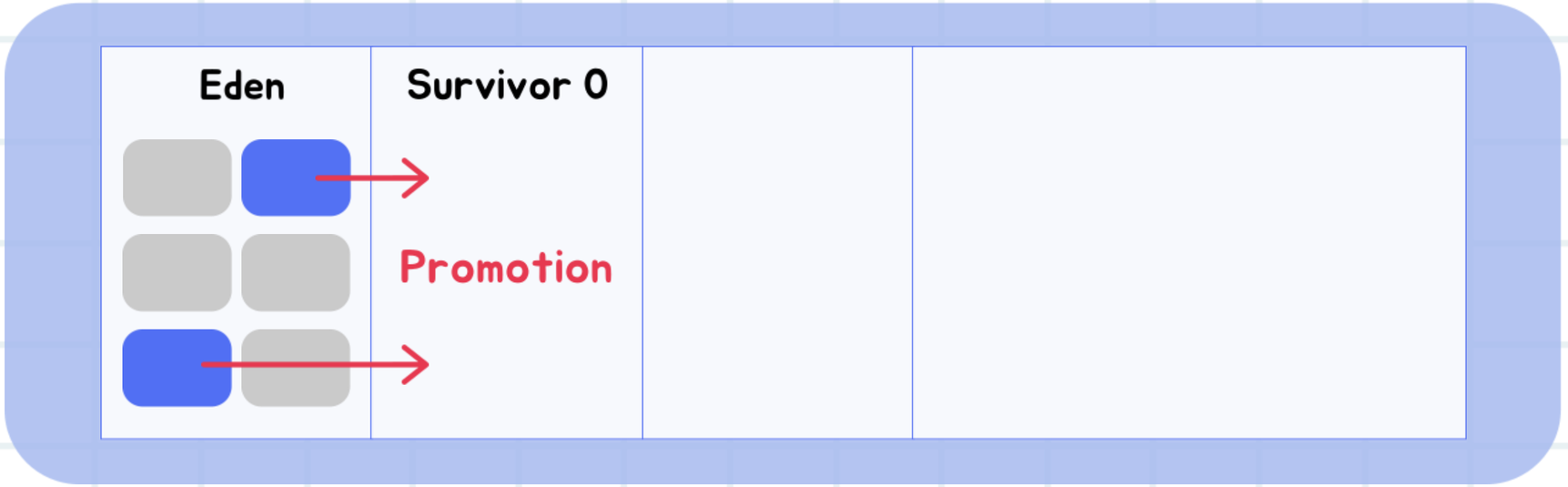


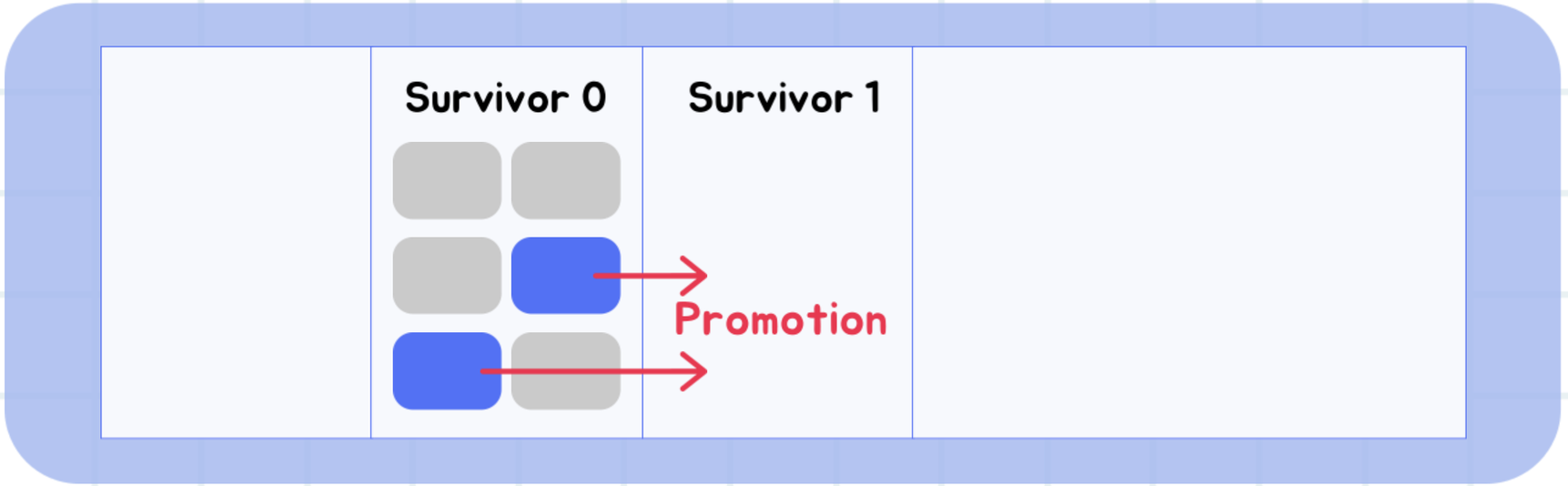
불필요한 객체 생성은 왜 안 좋은가?



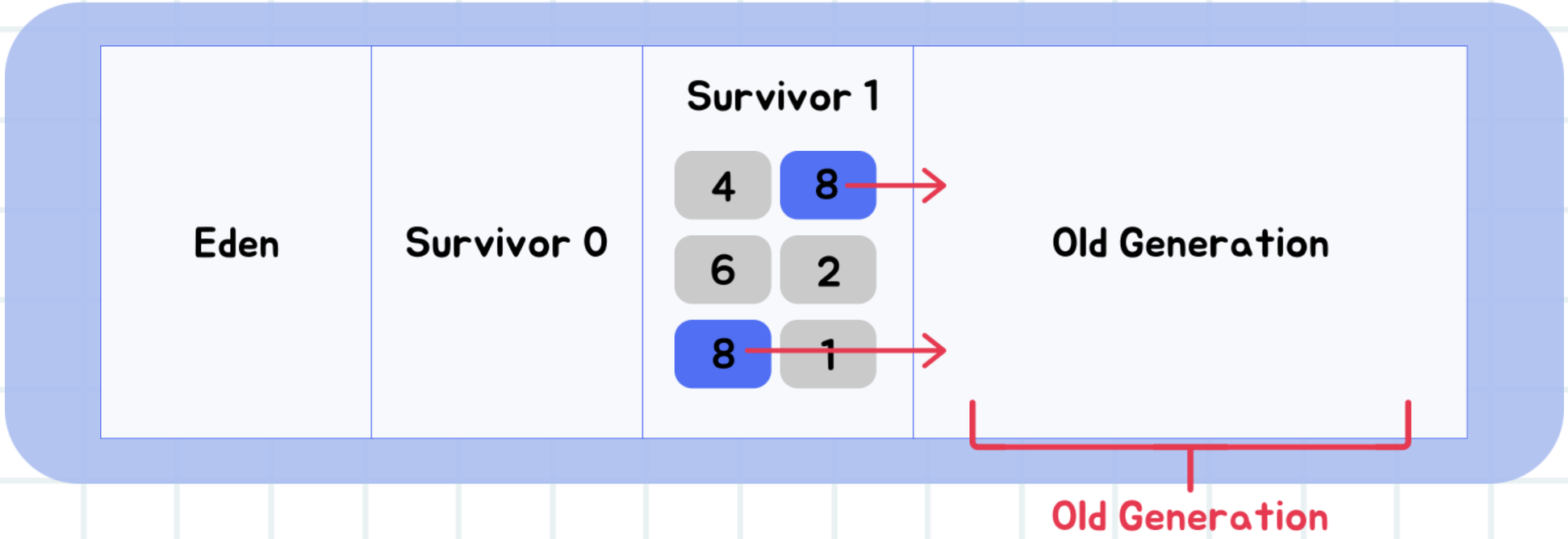
- Minor GC
  - Young Generation 영역에서 발생하는 GC
  - 참조되지 않거나 사용되지 않는 객체 삭제
- Eden
  - 새로 생성된 객체가 저장되는 영역
- Survivor 0
  - Eden영역에서의 GC에서 살아남은 객체가 저정되는 영역
- Survivor 1
  - Survivor 0 영역에서의 GC에서 살아남은 객체가 저정되는 영역

불필요한 객체 생성은 왜 안 좋은가?





불필요한 객체 생성은 왜 안 좋은가?



- Old Generation
  - Minor GC가 수행될 때마다 살아남은 객체에 age 부여
  - Survivor 1에서 특정 age 임계치를 넘어간 객체들의 저장되는 곳
- Major GC
  - Old Generation 영역이 꽉찼을때 발생하는 GC

불필요한 객체 생성은 왜 안 좋은가?

- Major GC
  - Old Generation영역은 Young Generation 영역보다 큰 영역 보유
  - Major GC가 수행되는데 시간이 오래걸리기 때문에 GC를 수행하는 스레드 이외의 스레드는 모두 멈춤
- Stop the world
  - 지연 발생

불필요한 객체 생성 → Promotion 빈도 증가 → Old Generation 꽉차는 횟수 증가 → Major GC 발생 빈도 증가

불필요한 객체 생성은 지속적인 프로그램 지연 초래

3. 비싼 객체

- 비싼 객체란?
  - 생성하는데에 많은 양의 메모리, 디스크 등을 필요로 하는 객체
  - Ex. Connection 객체, Pattern 객체
  - 1번과 같은 이유로 불필요한 비싼 객체 생성은 전체적인 프로그램의 성능 저하 유발

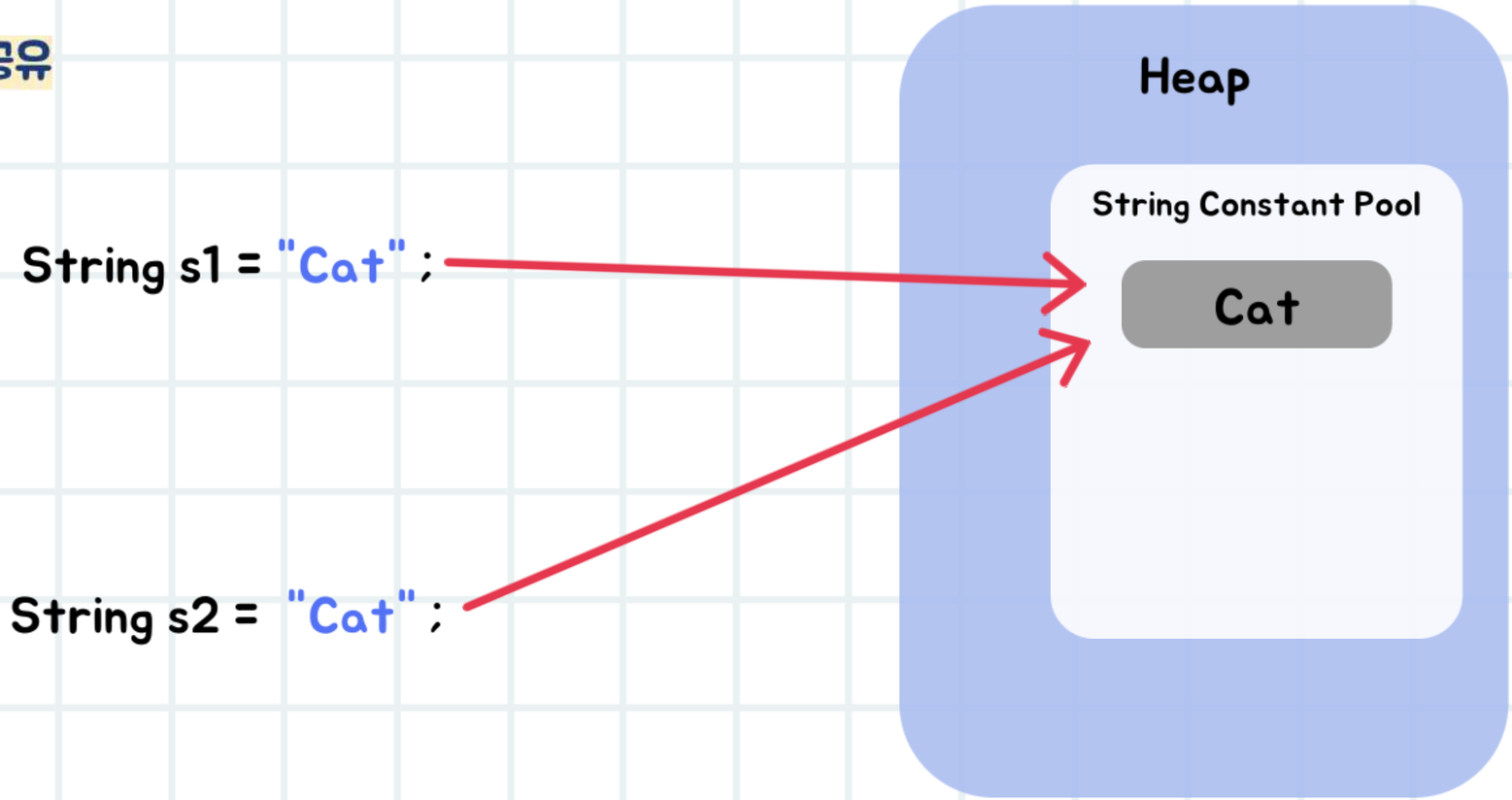
Case 1

- String 객체를 생성하는 방법 1
  - new 연산자
  - 참조값이 다른 객체 생성

```
String s1 = new String ("Cat");
String s2 = new String ("Cat");
```

불필요한 객체 생성의 예

- String 객체를 생성하는 방법 2
  - " "를 이용한 객체 생성
  - String Constant pool에 객체 저장
  - 문자가 같다면 같은 객체 공유

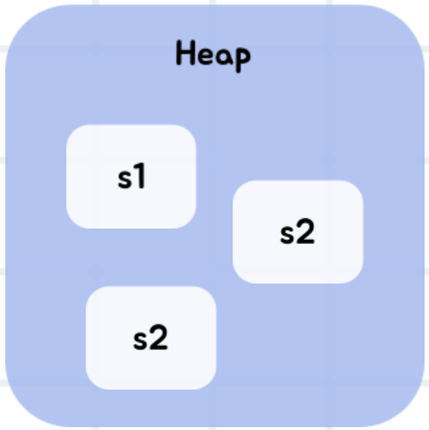


String Constant Pool과 Runtime Constant Pool은 서로 다른 것이다.

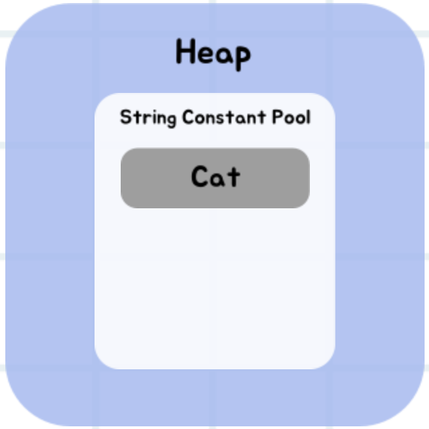
불필요한 객체 생성의 예

if. 문자 "Cat"을 가진 변수 3개를 선언하고 싶다면?

```
String s1 = new String ("Cat");
String s2 = new String ("Cat");
String s3 = new String ("Cat");
```




```
String s1 = "Cat" ;
String s2 = "Cat" ;
String s3 = "Cat" ;
```



Case 2

String 타입을 Boolean 타입으로 변환하고 싶을때

```
String s = "True";

// 방법 1
Boolean result1 = Boolean(s);

// 방법 2
Boolean result2 = Boolean.valueOf(s);
```

불필요한 객체 생성의 예

1. Boolean result1 = Boolean(s) ;

```
public Boolean(String s) { this(parseBoolean(s)); }
```

```
 * Example: {@code Boolean.parseBoolean("True")} returns {@code true}.<br>
 * Example: {@code Boolean.parseBoolean("yes")} returns {@code false}.
 *
 * @param      s   the {@code String} containing the boolean
 *                 representation to be parsed
 * @return the boolean represented by the string argument
 * @since 1.5
 */
@Contract(value = "null->false", pure = true)
public static boolean parseBoolean(String s) { return "true".equalsIgnoreCase(s); }
```

- 생성자를 이용해 parseBoolean() 메서드 수행
- 생성자를 이용하기 때문에 객체가 생성된다.

2. Boolean result1 = Boolean.valueOf(s) ;

```
@NotNull
public static Boolean valueOf(String s) { return parseBoolean(s) ? TRUE : FALSE; }
```

Returns a String object representing the specified boolean. If the specified boolean is true, then t
string "true" will be returned, otherwise the string "false" will be returned.

Params: b – the boolean to be converted

Returns: the string representation of the specified boolean

Since: 1.4

- valueOf() 메서드를 이용해서 parseBoolean() 메서드 수행
- 정적 메서드를 바로 호출하는 것이기 때문에 객체 생성이 발생하지 않는다.

객체를 생성하지 않는 valueOf() 방식이 메모리 측면에서 유리

Case 3

```
static boolean isRomanNumeral(String s){
    return s.matches("^(?=.)M*(C[MD]|D?C{0,3}")
}
```

- 정규 표현식을 통해 로마 숫자를 판별하는 메서드
- 정규 표현식 이란?
  - 문자열 속 '특정한 형태나 규칙' , 패턴을 정의하는 식
  - EX. goorm1234@naver.com

    [a-zA-Z0-9._+-]+@[a-zA-Z0-9]+₩.[a-zA-Z0-9]

불필요한 객체 생성의 예

```
 * @since 1.4
 */
@Contract(pure = true)
public boolean matches( @NonNls  @NotNull String regex) { return Pattern.matches(regex,  input: this); }
```

```
public static boolean matches( @NotNull  @NonNls String regex, CharSequence input) {
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(input);
    return m.matches();
}
```

Pattern.matches

Pattern.compile

Matcher.matches

- Pattern.compile()
  - 정규표현식을 해석하고 패턴을 생성과정이 계산적으로 매우 복잡
  - Pattern 객체 자체를 생성하는데에 많은 메모리를 요구함 (비싼 객체)
  - Pattern.matches()를 반복 수행시, 비싼 객체가 계속해서 새롭게 생성

불필요한 객체 생성의 예

```
public class RomanNumerals{
    private static final Pattern ROMAN = Pattern.compile("^(?=.)M*(C[MD]|D?C{0,3}")

    static boolean isRomanNumeral(String s) {
      return ROMAN.matcher(s).matches();
    }
}
```

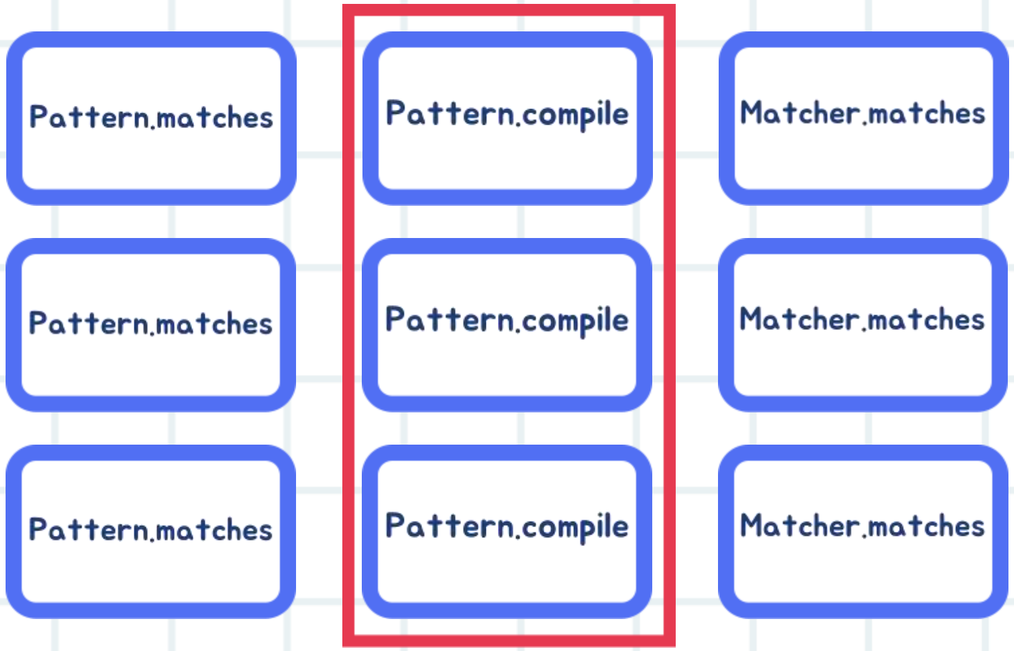
- ROMAN에 Pattern.compile 결과를 캐싱
- isRomanNumeral()를 실행할때마다,

  캐싱해 놓은 ROMAN(Pattern객체)를 재사용하여 Matcher.matches 수행

불필요한 객체 생성의 예

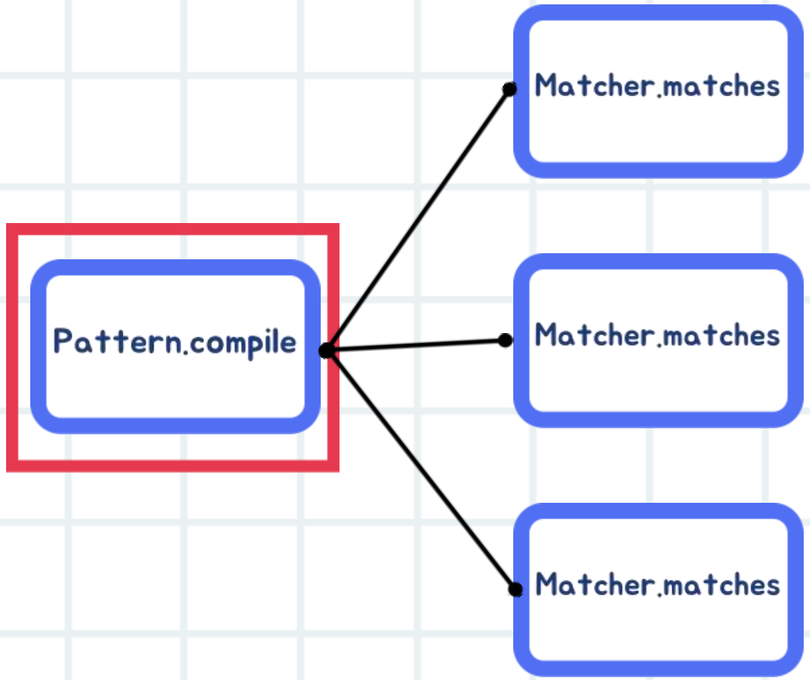
if. 로마 숫자 판별 메서드를 3번 반복 실행한다면?

[첫번째 코드]



3개의 Pattern 객체

[두번째 코드]



1개의 Pattern 객체

Case 4

```
private static long sum(){
    Long sum = 0L;
    for (long i = 0;  i <= Integer.MAX_VALUE;  i++){
        sum + = i ;
    }
    return sum
}
```

- long : 원시타입 (primitive type)
- Long : 래퍼 클래스 (wrapper class)

불필요한 객체 생성의 예

- 래퍼 클래스(wrapper class)란?
  - 원시 타입을 참조 타입으로 다룰 수 있도록 객체를 생성해주는 것
  - 원시 타입보다 더 넓은 범위를 커버할 수 있다.
- 래퍼 클래스는 언제 쓰는가?
  - NULL을 사용하고 싶을때
  - Generic을 사용하고 싶을때

- 오토 박싱(Auto Boxing)란?
  - 원시타입과 래퍼 클래스가 만났을때, 원시타입을 래퍼 클래스로 자동 변환해주는 것

- Generic이란?
  - 데이터 타입이 명시되지 않은 상태
  - 즉, 데이터 타입을 미리 정의하지 않고, 객체를 생성하여 개발자의 필요에 의해 설정할 수 있게 해주는 것
- Generic의 장점
  - 데이터 타입을 유연하게 처리 가능
  - 잘못된 타입오류로 인해 발생할 수 있는 런타임 에러를 컴파일 과정에서 검출 가능

불필요한 객체 생성의 예

```
private static long sum(){
    Long sum = 0L;
    for (long i = 0;  i <= Integer.MAX_VALUE;  i++){
        sum + = i ;
    }
    return sum
}
```

Auto Boxing 발생

- 원시타입인 i 와 래퍼클래스인 sum이 만나 AutoBoxing 발생
- for 반복마다 i 가 래퍼클래스로 변환 / 즉, 반복해서 객체 생성

Long sum 을 long sum 으로 수정하면 속도를 높이면서 메모리 낭비를 막을 수 있다.

마무리

기존 객체를 재사용해야 한다면 새로운 객체를 만들지 말자