Java 꽉 잡아!
- JVM부터 GC, 스레드 동기화까지!

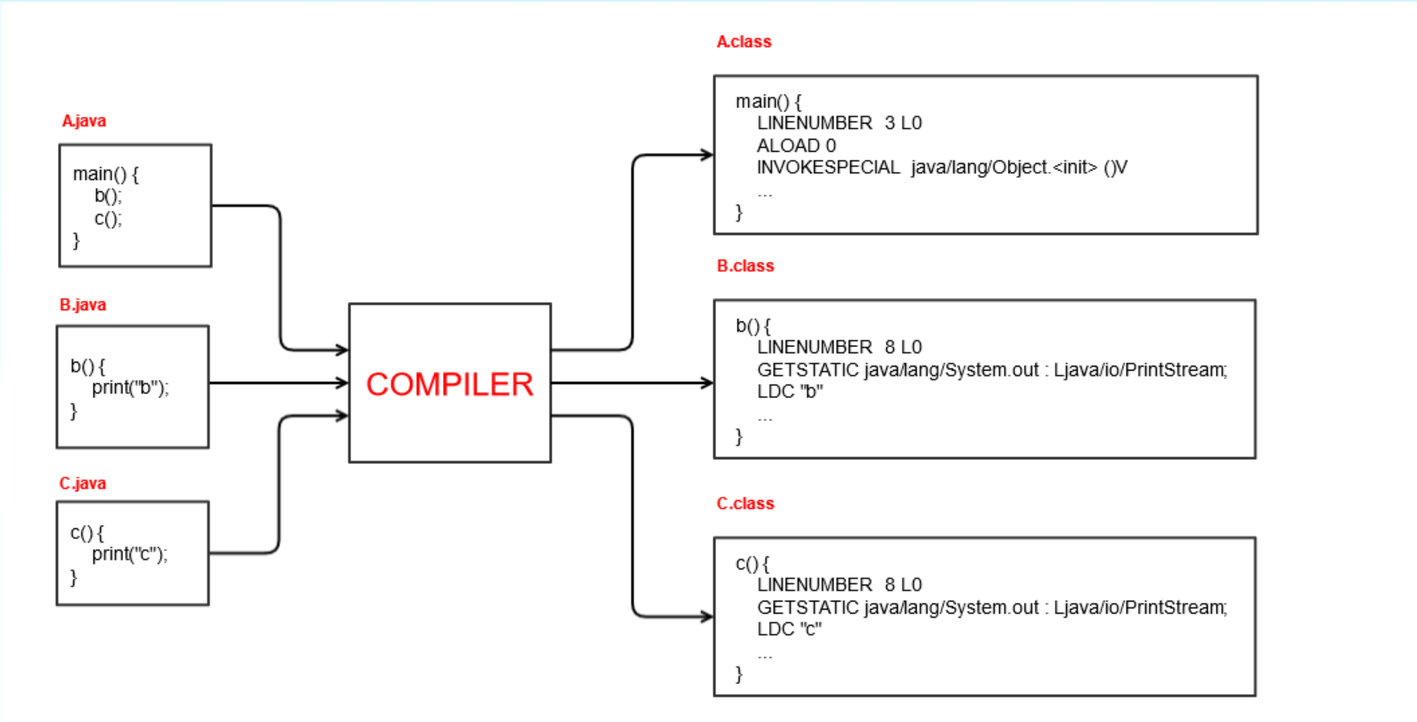
[1-2] JVM의 정의와 구조, 메모리에 대해 살펴봅니다.

김재녕 (2023-06-05)

[1] JVM과 메모리 구조

JVM과 메모리 구조

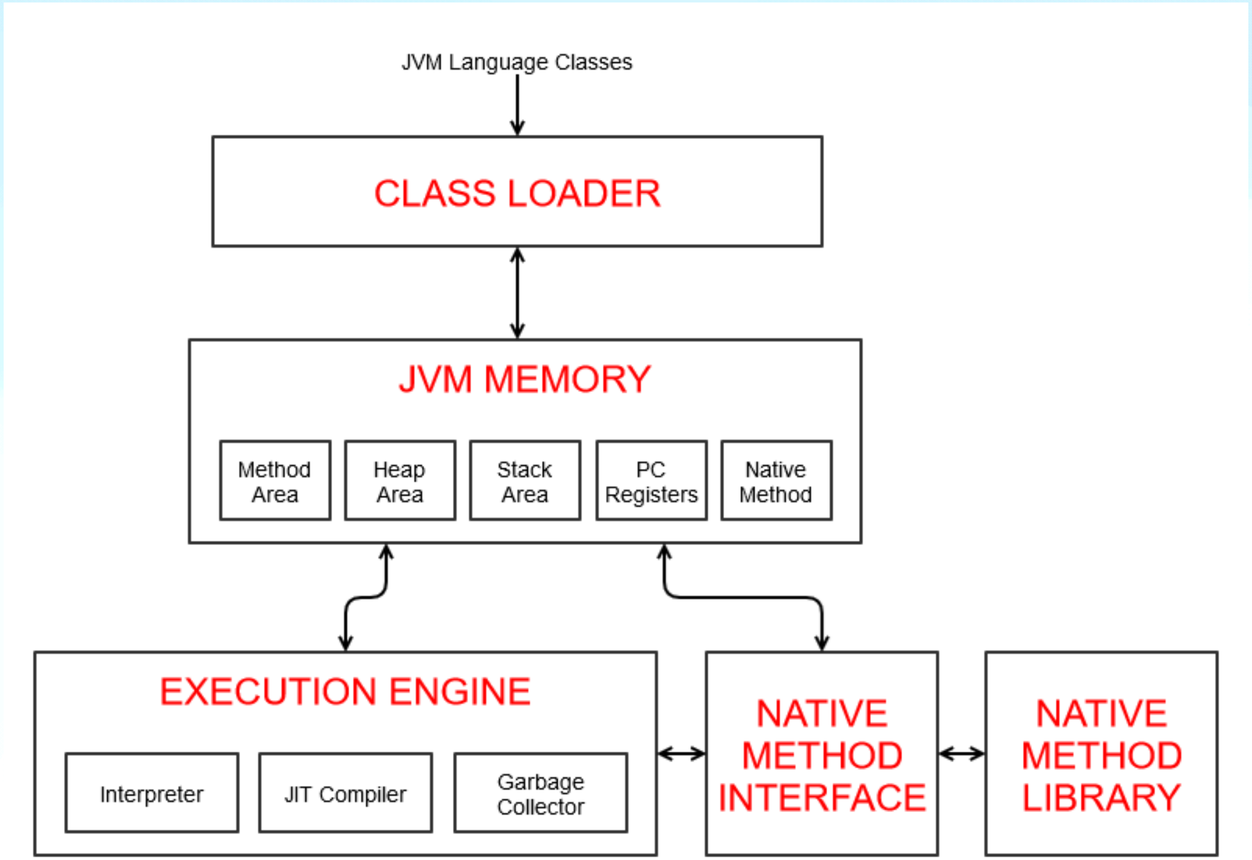
Java 동작 방식



https://www.baeldung.com/wp-content/uploads/2021/01/java_compilation-3.png

JVM과 메모리 구조

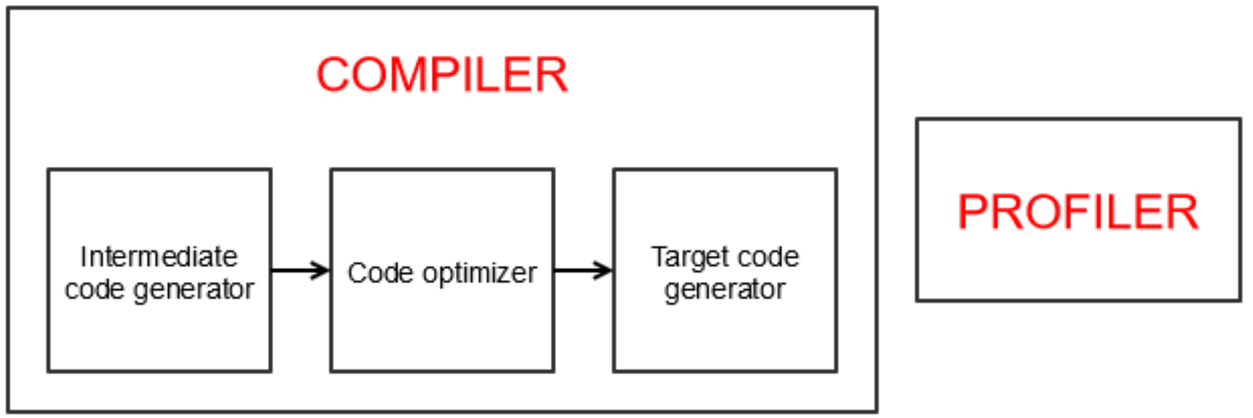
Java 동작 방식



https://www.baeldung.com/java-compiled-interpreted

JVM과 메모리 구조

Java 동작 방식

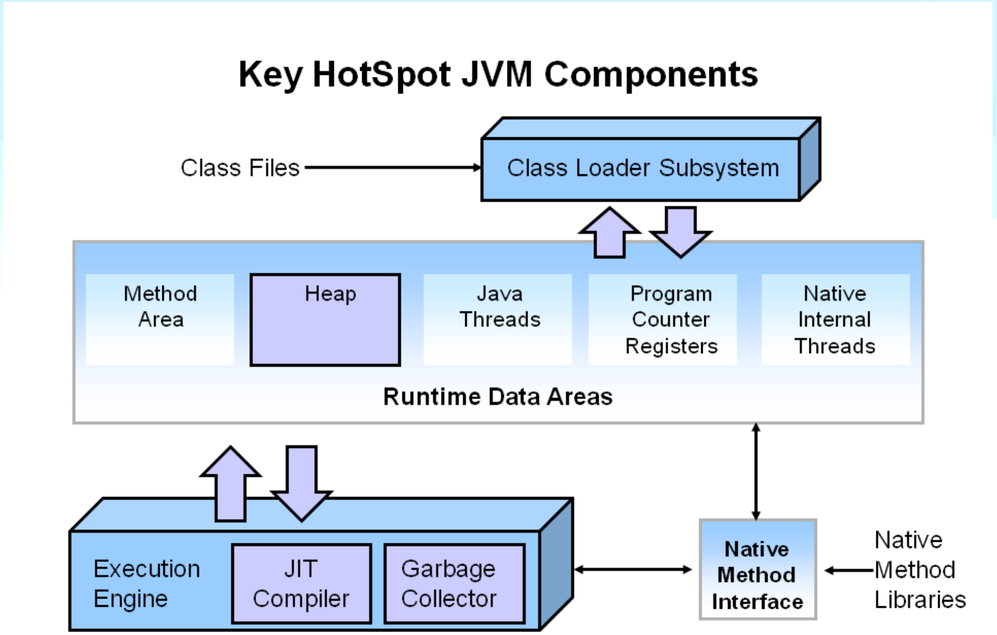


https://www.baeldung.com/java-compiled-interpreted

JVM과 메모리 구조

JVM

- JVM(Java Virtual Machine), 자바 가상 머신
- 논리적인 개념, 여러 모듈의 결합체
- Java 앱을 실행하는 주체
- JVM 때문에 다양한 플랫폼 위에서 동작 가능
- 대표적인 역할, 기능
  - 클래스 로딩
  - GC 등 메모리 관리
  - 스레드 관리
  - 예외 처리

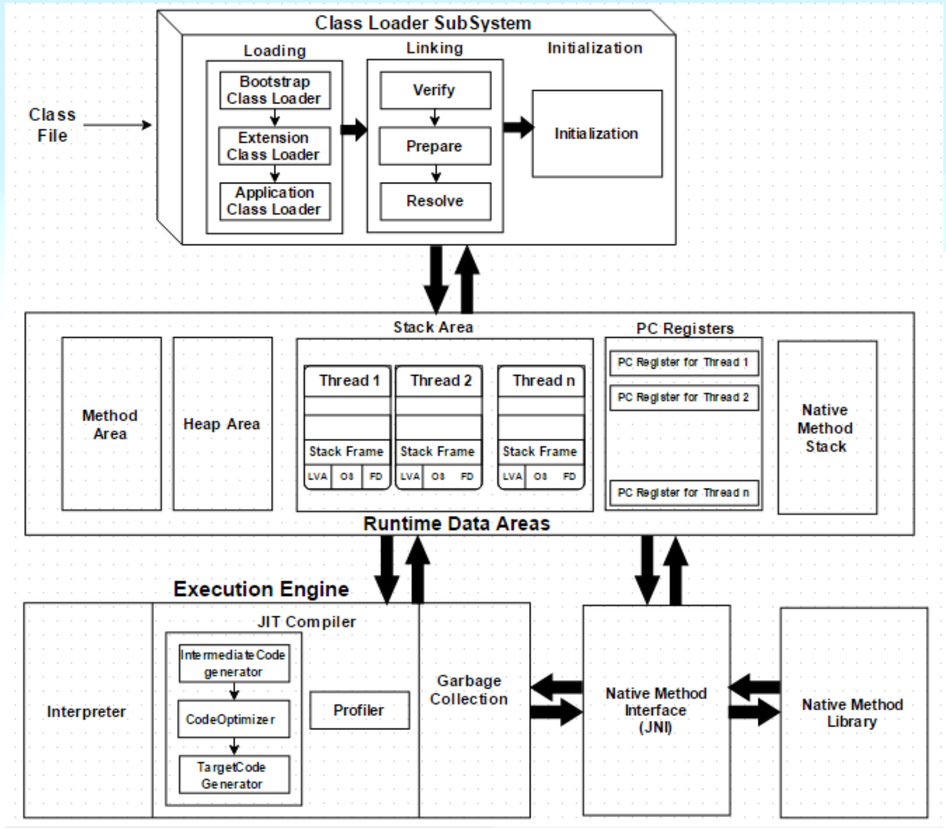


https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

JVM과 메모리 구조

JVM Architecture

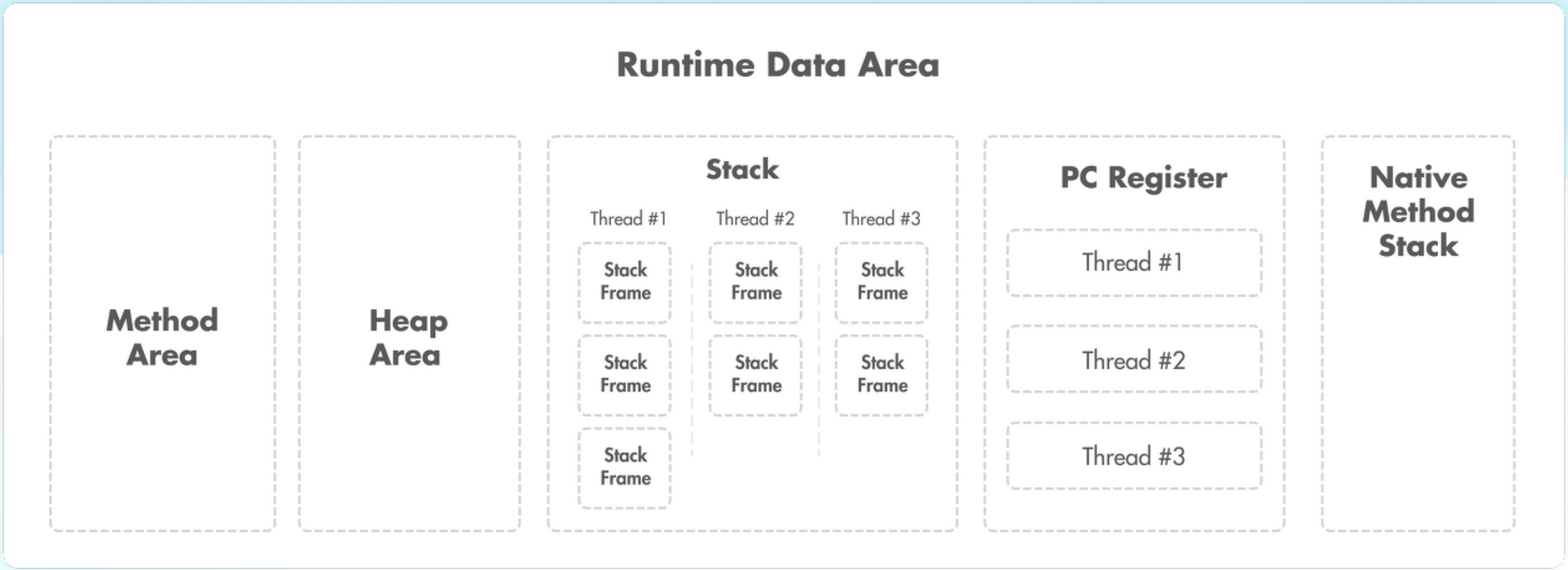
- Class Loaders
  - 바이트코드 로딩, 검증, 링킹 등 수행
- Runtime Data Areas
  - 앱 실행을 위해 사용되는 JVM 메모리 영역
- Execution Engine
  - 메모리 영역에 있는 데이터를 가져와 해당하는 작업 수행
- JNI (Java Native Interface)
  - JVM과 네이티브 라이브러리 간 이진 호환성을 위한 인터페이스
  - 네이티브 메서드(네이티브 언어 C/C++ 등으로 작성) 호출, 데이터 전달과 메모리 관리 등 수행
- Native Libraries
  - 네이티브 메서드의 구현체를 포함한 플랫폼별 라이브러리



https://dzone.com/articles/jvm-architecture-explained

JVM과 메모리 구조

JVM Run-Time Data Areas



https://gngsn.tistory.com/252

JVM과 메모리 구조

JVM Run-Time Data Areas

- The pc Register
  - 스레드 별로 생성되며 실행 중인 명령(오프셋)을 저장하는 영역
- Java Virtual Machine Stacks (Stack Area, Java Stack)
  - 스레드 별로 생성되며 메서드 실행 관련 정보를 저장하는 영역 (프레임 저장)
- Heap
  - JVM 실행 시 생성되며 모든 객체 인스턴스/배열에 대한 메모리가 할당되는 영역
- Method Area
  - JVM 실행 시 생성되며 클래스의 구조나 정보를 저장하는 영역
- Native Method Stacks
  - 스레드 별로 생성되며 네이티브 코드 실행에 관련 정보를 저장하는 영역

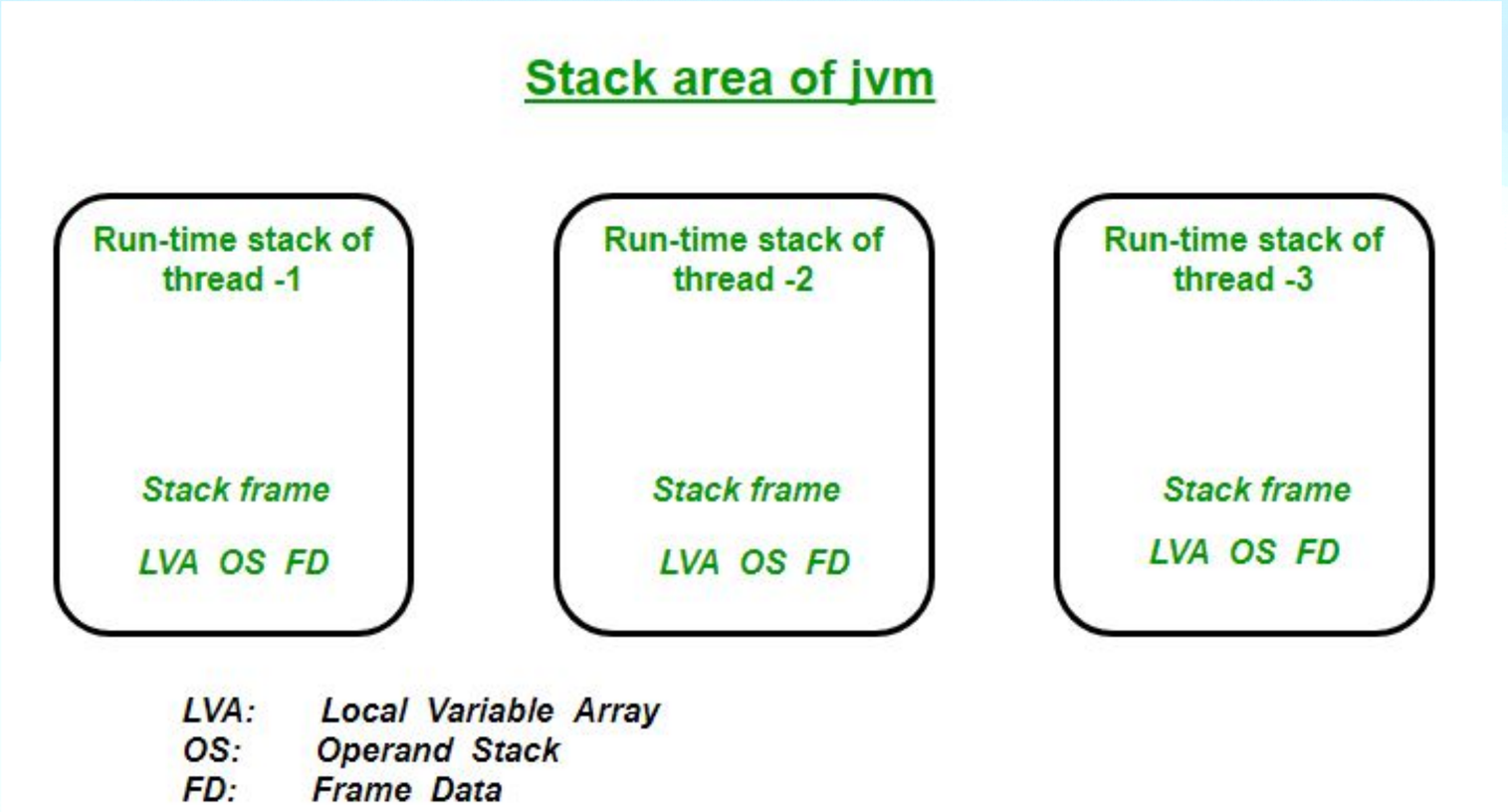
JVM과 메모리 구조

JVM Run-Time Data Areas - The pc Register

- 스레드 생성 시 생성/할당되며 현재 실행 중인 명령의 주소를 저장하는 영역 (명령어 포인터)
  - 레지스터는 프로세서 내에서 자료를 보관하는 빠른 기억 장치
  - 저장되는 명령의 주소는 Java 바이트 스트림(바이트코드) 안에 오프셋을 의미
    - 바이트코드 명령 자체(opcode)는 메서드 영역에 저장
- 각 스레드가 메서드를 실행할 때 실행 메서드에 따라 저장 여부가 결정됨
  - Java 메서드 -> 실행 명령의 주소를 저장
    네이티브 메서드-> 저장되지 않음

JVM과 메모리 구조

JVM Run-Time Data Areas - Java Virtual Machine Stacks



https://www.geeksforgeeks.org/java-virtual-machine-jvm-stack-area/

JVM과 메모리 구조

JVM Run-Time Data Areas - Java Virtual Machine Stacks

- 스레드 생성 시 생성/할당되며 프레임(Frame)이 저장되는 영역 (Stack Area, Java Stack)
  - JVM의 구현 방식에 따라 크기와 프레임 관리 방법 등이 다를 수 있음
- 로컬 변수 저장과 메서드 호출/반환 등과 같은 작업 시 사용
- 스택은 push/pop을 제외하고 직접 조작되지 않기 때문에 프레임은 힙에 할당될 수 있음
  - 메모리는 순차적일 필요가 없음
- 허용된 것 보다 더 큰 스택이 필요한 경우 StackOverFlowError 발생
- 스택을 생성하거나 동적으로 크기를 확장할 때 메모리가 부족하면 OutOfMemoryError 발생

JVM과 메모리 구조

* Frame

- JVM stack에 생성(push)되는 메서드 관련 정보 저장 단위 (Stack Frame, Activation Record)
  메서드가 호출될 때 생성(push)되며 종료되면 소멸(pop)됨
- 메서드의 데이터(매개 변수와 지역 변수 등)와 부분 결과 등을 저장함
  동적 연결, 메서드 값 반환, 예외 전달 등에 사용됨
- 각 프레임에는 런타임에 필요한 메서드 참조를 위해 런타임 상수 풀 참조를 포함
- 구성
  - 지역변수 배열 (Local Variables)
    매개 변수와 지역 변수 저장
  - 오퍼랜드 스택 (Operand Stacks)
    실행 중간 연산 결과 등을 임시로 저장
  - 프레임 데이터 (Frame Data)
    반환 주소 등 기타 데이터 저장

JVM과 메모리 구조

* Frame - Local Variables

```java
public void bike(int i, long l, float f, double d, Object o, byte b) {  no usages
    // TODO
}
```

Bike()

Index	Type	Parameter
0	Int	int i
1	Long	long l
3	Float	Float f
4	Double	Double d
6	Reference	Object O
7	int	Byte b

https://www.geeksforgeeks.org/java-virtual-machine-jvm-stack-area/

JVM과 메모리 구조

* Frame - Local Variables

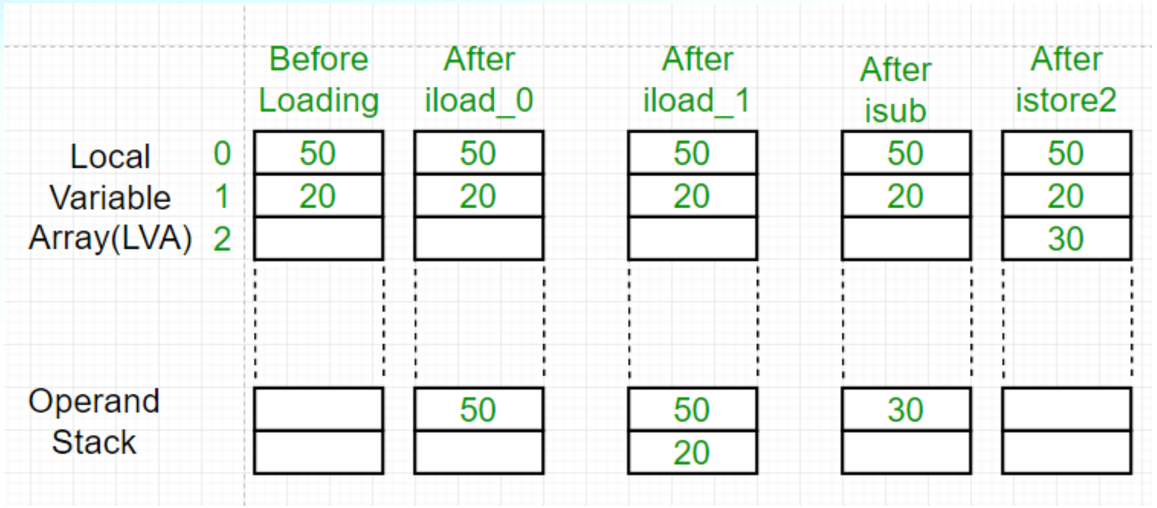
- 이 지역 변수 배열의 길이는 컴파일 타임에 결정되며
  관련 메서드, 클래스 등의 정보와 함께 바이트코드로 제공됨
- 지역 변수는 원시 타입과 reference, returnAddress이 한 곳(slot)에 저장되며
  특히 8바이트인 long, double 타입은 연속된 두 곳(slot)에 저장됨
- 지역 변수 배열의 첫번째 인덱스는 0이며 0부터 `array.length-1` 까지 유효한 인덱스 범위로 간주
- JVM에 의해 메서드 호출 시 매개 변수(또는 Args)는 지역 변수 배열에 담겨 전달
  - 인스턴스 메서드 호출 시에 지역 변수 0은 인스턴스 메서드 객체 참조인 `this`를 전달하는데 사용
    그 이후 1부터 모든 매개변수들이 표현됨

JVM과 메모리 구조

* Frame - Operand Stacks

```
iload_o     // push the int in operand stack
iload_1     // push the int in the operand stack
isub        // pop two int, subtract them & push result in operand stack
istore_2    // pop result, store into local variable at index 2
```

https://www.geeksforgeeks.org/java-virtual-machine-jvm-stack-area/



https://www.geeksforgeeks.org/java-virtual-machine-jvm-stack-area/

JVM과 메모리 구조

* Frame - Operand Stacks

- 일반적인 LIFO 방식의 스택으로 최대 길이는 컴파일 타임에 결정되며
  관련된 메서드 코드와 함께 제공되며 최초 생성 시 오퍼랜드 스택은 비어있음
  - 문맥에 따라 다를 수 있지만 일반적으로 오퍼랜드 스택은 프레임의 오퍼랜드 스택 지칭
- 지역 변수 배열과 다르게 인덱스가 아닌 push/pop을 할 수 있는 명령어에 의해 액세스 가능
- 저장되는 오퍼랜드의 타입은 JVM에서 제공하는 모든 타입의 값
- JVM은 지역 변수, 필드의 값을 오퍼랜드 스택으로 로딩하는 명령 제공하며
  다른 명령들을 통해 오퍼랜드 스택에서 값을 가져와 연산, 결과를 다시 저장함
  - 오퍼랜드 스택은 호출할 메서드에 전달할 매개 변수 전달과 결과를 수신할 때도 사용됨
- 소수의 JVM 명령은 타입에 관계 없이 원시 타입의 값으로 런타임 데이터 영역에서 작동함

JVM과 메모리 구조

* Frame - Frame Data

- 연관된 메서드들의 심볼릭 레퍼런스와 메서드 반환에 필요한 데이터 저장
  - 예외가 발생한 경우 catch 블록 정보를 제공하는 Exception 테이블 참조 포함

JVM과 메모리 구조

JVM Run-Time Data Areas - Method Area

- JVM 실행 시 생성되어 모든 스레드에게 공유되며 클래스 별 구조와 정보(메타 데이터)를 저장하는 영역
  - 클래스/인터페이스/인스턴스 초기화에 사용되는 스페셜 메서드
  - 런타임 상수 풀, 필드/메서드 데이터
  - 생성자/메서드 코드
- Hotspot VM 기준으로 Metaspace(PermGen) 영역에 관리됨 (JVM 구현마다 다름)
- 컴파일된 코드를 저장하는 영역 또는 OS 프로세스의 text 세그먼트와 유사
- 메서드 영역은 논리적으로 힙의 일부, 간단한 구현의 경우 GC/압축/컴팩트를 선택하지 않을 수 있음
  - 메서드 영역의 위치, 컴파일된 코드 관리에 대한 정책을 요구하지 않음
- 메서드 영역의 크기는 상황에 따라 고정/확장/축소될 수 있음
- 메모리 할당 요청을 충족할 수 없으면 OutOfMemoryError 발생

JVM과 메모리 구조

JVM Run-Time Data Areas - Heap

- 모든 객체 인스턴스, 배열에 대한 메모리가 할당되는 데이터 영역
- JVM 실행 시 생성되며 모든 JVM 스레드에게 공유되는 영역
- GC가 처리되는 영역
- 특정 스토리지 시스템에 종속적이지 않은 구조
- 힙의 크기는 상황에 따라 고정/확장/축소될 수 있음
  - 힙 메모리는 순차적일 필요가 없음
- 계산된 것보다 더 많은 힙 메모리가 필요한 경우 OutOfMemoryError 발생

JVM과 메모리 구조

* Run-Time Constant Pool

- 클래스/인터페이스가 로딩될 때 메서드 영역(Method Area)에 할당되는 자료구조
  - 컴파일 시 `.class` 파일에 생성되는 일반 상수 풀의 런타임 표현
- 일반 상수 풀의 데이터를 기반으로 생성되며 스태틱 상수와 심볼릭 레퍼런스(또는 실제 참조) 등을 포함
  - 상수 뿐 아니라 메서드, 필드 참조까지 여러 종류의 상수가 포함됨
- 일반 프로그래밍 언어의 심볼 테이블과 유사하지만 그보다 더 넓은 범위에 데이터를 포함함
  - 심볼 테이블은 컴파일러/인터프리터가 프로그램을 분석/처리 시 사용하는 자료구조이며
    코드의 식별자/상수/프로시저/함수 등과 관련된 정보를 저장함
- Method Area 영역에서 허용 가능한 메모리를 초과하면 OutOfMemoryError 발생

JVM과 메모리 구조

* Constant Pool

- 상수 풀 (Constant Pool)
  - Java 바이트코드에 포함되어 있는 모든 상수 값을 저장하는 심볼(룩업) 테이블
    .java 파일이 Java 컴파일러에 의해 컴파일 되어 Java 바이트코드로 변환될 때 생성
  - 클래스명, 필드명, String/Primitive type 리터럴, 심볼릭 레퍼런스 등이 저장됨
  - 컴파일 타임 시점에 알 수 있는 정보들이 저장됨
- 상수 풀에서 런타임 상수 풀에 새로 생성(이동)되는 데이터
  - 심볼릭 레퍼런스나 String.intern 등 런타임에 달라질 수 있는 데이터
  - 컴파일 타임에 확정되어 런타임에 변경되지 않는 데이터(리터럴 값 등)들을 제외한 데이터

JVM과 메모리 구조

* Symbolic Reference

- Java 바이트코드에서 클래스/인터페이스/필드 등 참조하는 다른 요소를 표현 방식
- JVM 구현에 따라 심볼릭 레퍼런스가 Lazy가 아닌 Eager로 확인/연결될 수 있음
    - 이때 발생하는 오류는 직간접적으로 참조(사용)하는 지점에서 발생해야 함
- 클래스가 로딩 후 링킹(Resolution)되는 시점에서 심볼릭 레퍼런스가 실제 주소값으로 대체 됨
- 일반적인 예시 (간접적인 표현)
    - `java.lang.String` 클래스의 `private final byte[] value` 필드
        - `Ljava/lang/String;.value:[C`
            - `L` -> 참조 타입(클래스/인터페이스)
            - `;` -> 클래스와 필드의 구분자 (일반적으로 생략함)
            - `.value` -> 필드명
            - `:` -> 참조
            - `[` -> 배열 표현 (2차원이라면 `[[`)
            - `C` -> `char` 타입
    - `java.lang.String` 클래스의 `public char charAt(int index)` 메서드
        - `Ljava/lang/String;.charAt:(I)C`
            - `(I)` -> 메서드의 파라미터 타입
            - `C` -> 메서드의 반환 타입

JVM과 메모리 구조

JVM Run-Time Data Areas - Native Method Stacks

- 스레드 생성 시 생성/할당되며 네이티브 메서드 데이터가 저장되는 영역
  - 네이티브 메서드는 Java 외에 언어로 작성된 메서드 (일반적으로 C/C++)
  - 네이티브 로딩 상태나 사용 여부에 따라 JVM이 해당 영역을 제공하지 않을 수 있음
  - C와 같은 언어로 구성된 JVM 명령셋을 위한 인터프리터 구현에 사용될 수 있음
- 네이티브 메서드 스택의 크기는 상황에 따라 고정되거나 확장, 축소될 수 있음
  - 네이티브 메서드 스택 크기가 고정되어 있다면 생성될 때 크기를 독립적으로 선택할 수 있음
- 허용된 것 보다 더 큰 스택이 필요한 경우 StackOverFlowError 발생
- 동적 확장 시도 중에 사용 가능한 메모리가 부족하면 OutOfMemoryError 발생

JVM과 메모리 구조

JVM Run-Time Data Areas - 생성 시점 정리

- JVM 실행 시
  - Heap
  - Method Area
- 스레드 실행 시
  - pc register
  - Java Virtual Machine Stacks
  - Native Method Stacks (필요한 경우)
- 클래스/인터페이스 생성 시
  - Run-Time constant Pool (Method Area에 저장)
- 메서드 호출 시
  - Frame (Java Virtual Machine Stacks에 저장)

JVM과 메모리 구조

* Special Methods

- Instance Initialization Methods (인스턴스 초기화 메서드)
  - Constructor, Instance Initializer Blocks (바이트 수준에선 동일)
  - 조건
    - 클래스 안에 정의
    - <init> 이라는 이름을 가진 스페셜 메서드
    - 반환 타입은 `void`
- Class Initialization Methods (클래스 초기화 메서드)
  - static Initializer Blocks
  - 조건
    - <clinit> 이라는 이름을 가진 스페셜 메서드
    - 반환 타입은 `void`
    - 클래스 파일의 버전이 51.0 이상인 경우 해당 인자를 사용하지 않고 ACC_STATIC 플래그가 설정된 메서드
- Signature Polymorphic Methods
  - 런타임에 메서드 시그니처가 결정되는 메서드
  - 대표적인 예시로 JDK 7부터 도입된 `java.lang.invoke.MethodHandle` 클래스의 `invoke`, `invokeExact` 메서드
  - 조건
    - `java.lang.invoke.MethodHandle` 또는 `java.lang.invoke.VarHandle` 클래스에서 선언
    - Object[] 타입의 단일 공식 파라미터
    - `ACC_VARARGS`, `ACC_NATIVE` 플래그 설정

JVM과 메모리 구조

* Special Methods - Signature Polymorphic Methods

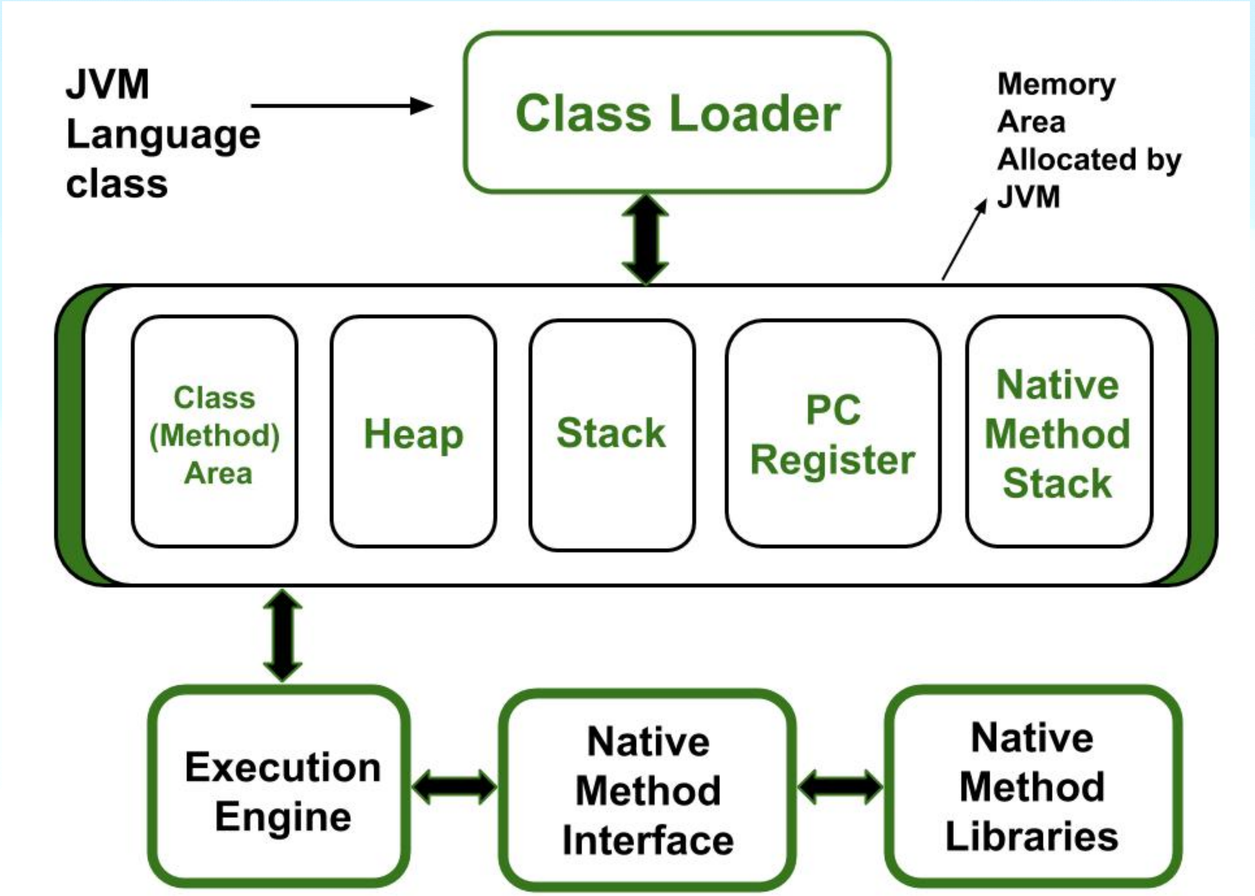
```
public abstract class MethodHandle implements Constable { Complexity is 25 You must be kidding

    Internal marker interface which distinguishes (to the Java compiler) those methods which are
    signature polymorphic.
@java.lang.annotation.Target({java.lang.annotation.ElementType.METHOD})
@java.lang.annotation.Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
@interface PolymorphicSignature { }
```

```
/**
 * Invokes the method handle, allowing any caller type descriptor, but requiring an exact type match.
 * The symbolic type descriptor at the call site of {@code invokeExact} must
 * exactly match this method handle's {@link #type() type}.
 * No conversions are allowed on arguments or return values.
 * <p>
 * When this method is observed via the Core Reflection API,
 * it will appear as a single native method, taking an object array and returning an object.
 * If this native method is invoked directly via
 * {@link java.lang.reflect.Method#invoke java.lang.reflect.Method.invoke}, via JNI,
 * or indirectly via {@link java.lang.invoke.MethodHandles.Lookup#unreflect Lookup.unreflect},
 * it will throw an {@code UnsupportedOperationException}.
 * @param args the signature-polymorphic parameter list, statically represented using varargs
 * @return the signature-polymorphic result, statically represented using {@code Object}
 * @throws WrongMethodTypeException if the target's type is not identical with the caller's symbolic type descriptor
 * @throws Throwable anything thrown by the underlying method propagates unchanged through the method handle call
 */
@IntrinsicCandidate
public final native @PolymorphicSignature Object invokeExact(Object... args) throws Throwable;
```

JVM과 메모리 구조

JVM Run-Time Data Areas



https://www.geeksforgeeks.org/how-many-types-of-memory-areas-are-allocated-by-jvm/

JVM과 메모리 구조

* Memory Type

- Heap memory
  - JVM 힙 영역, 모든 객체를 저장
    메모리 설정 가능(`java -Xms4096M -Xmx6144M com.example.Class`)
    부족한 경우 OutOfMemoryError 발생
- Stack memory
  - JVM stacks(지역 변수나 메서드 정보 등) 데이터 저장 (GC 영역 아님)
    JVM이 관리하는 영역이며 JVM에 의해 고정된 크기를 가짐
    부족한 경우 StackOverflowError 발생
- Native memory
  - 힙 외부에 할당되는 영역으로 오프힙(off-heap) 메모리라고도 표현함 (GC 영역 아님)
    외부 영역에 존재하기 때문에 데이터 입출력 시 직렬화 수행이 필요하고
    성능은 버퍼, 직렬화 프로세스, 디스크 공간 등 환경에 따라 달라짐
    JVM stacks, 내부 자료 구조, 메모리 매핑 파일 등 저장
- Direct memory (Direct Buffer Memory)
  - 힙 외부에 할당되지만 JVM 프로세스에 의해 사용되는 네이티브 메모리 영역 (GC 영역 아님)
    대표적으로 Java NIO에서 사용되는 영역(JNI 등)
    `-XX:MaxDirectMemorySize=1024M` 처럼 메모리 설정 가능

JVM과 메모리 구조

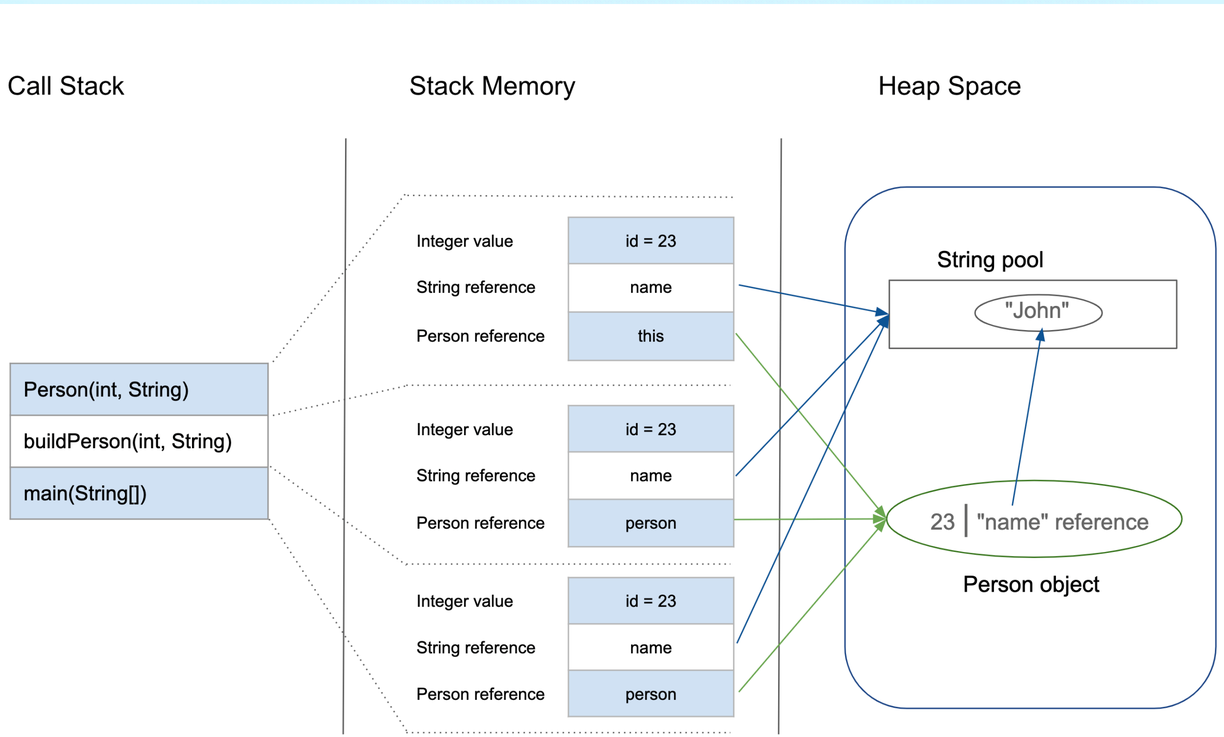
* Stack Memory & Heap Memory

```java
class Person {
    int id;
    String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

public class PersonBuilder {
    private static Person buildPerson(int id, String name) {
        return new Person(id, name);
    }

    public static void main(String[] args) {
        int id = 23;
        String name = "John";
        Person person = null;
        person = buildPerson(id, name);
    }
}
```



https://www.baeldung.com/java-stack-heap

JVM과 메모리 구조

* Stack Memory & Heap Memory

1. main 메서드 호출 시 이를 위한 스택 메모리가 할당되며 프레임(main)이 생성
   - id 변수의 int 값은 직접 저장됨
   - name 참조 변수는 힙의 스트링 풀을 참조
   - person 변수 생성 후 null 참조
2. main 메서드가 buildPerson 메서드를 호출하면서 프레임(buildPerson)이 생성됨
   - 파라미터로 넘어온 id의 값과 name의 참조가 저장됨
3. 연이어 Person 클래스의 인스턴스 생성을 위해 생성자가 호출되며 프레임(Person)이 생성됨
   - 파라미터로 넘어온 id의 값과 name의 참조 그리고 자신을 가리키는 `this`가 저장됨
   - 생성되는 Person 객체는 Heap에 저장
4. Person 인스턴스가 생성된 후 Person 프레임이 소멸되며 Person 참조 주소를 반환
5. 마찬가지로 buildPerson 프레임이 소멸되며 Person 참조 주소를 반환
6. main 메서드의 person 변수에 Person 참조 주소가 바인딩되고 main 프레임이 소멸되며 메서드 종료

JVM과 메모리 구조

* Stack Memory & Heap Memory

- Stack Memory (스레드 실행과 `static memory` 할당을 위해 사용)
  - 메서드 실행/완료에 따라 메모리 크기가 변경됨
  - 스택 내부 지역 변수는 메서드가 실행 동안에만 존재
  - 이 영역이 가득 차면 StackOverFlowError 발생
  - 힙 메모리 보다 상대적으로 빠름
  - 스레드 별로 할당 받기 때문에 스레드 세이프함
- Heap (객체 생성과 `dynamic memory` 할당을 위해 사용)
  - Young Generation, Old Generation 같은 메모리 엑세스 기술이 활용됨
  - 이 영역이 가득차면 java.lang.OutOfMemoryError 발생
  - 스택 메모리보다 상대적으로 느림
  - 스택 메모리와 다르게 해당 영역은 GC에 의해 메모리가 비워짐
  - JVM 앱 전체에서 공유되는 영역이기 때문에 스레드 세이프하지 않음 (별도의 동기화 처리 필요)

JVM과 메모리 구조

* Stack Memory & Heap Memory

- Applicaiton
  - 스택은 스레드에 의해 한 번에 하나씩 부분적으로 사용됨
  - 힙은 앱 전체에서 런타임 동안 사용됨
- Size
  - 스택 크기는 OS에 의존적이며 보통 힙보다 작음
  - 힙 크기는 제한 없음
- Storage
  - 스택은 힙에서 생성된 객체의 변수와 참조만 저장
  - 힙은 새로 생성된 모든 객체를 저장
- Order
  - 스택은 LIFO(Last-in First-out) 기준으로 사용/액세스
  - 힙은 Old/Young Generation 등 다소 복잡한 관리 방법을 통해 사용/액세스
- Life
  - 스택은 현재 메서드가 실행되는 동안만 존재
  - 힙은 앱이 실행되는 동안 존재
- Efficiency
  - 스택은 힙에 비해서 할당 속도가 매우 빠름
  - 힙은 스택보다 할당 속도가 느림
- Allocation/Deallocation
  - 스택은 메서드 호출 시 자동으로 할당되며 반환될 때 해제됨
  - 힙은 새 객체가 생성되면 할당되며 더이상 참조(사용)되지 않을 때 GC에 의해 해제됨

JVM과 메모리 구조

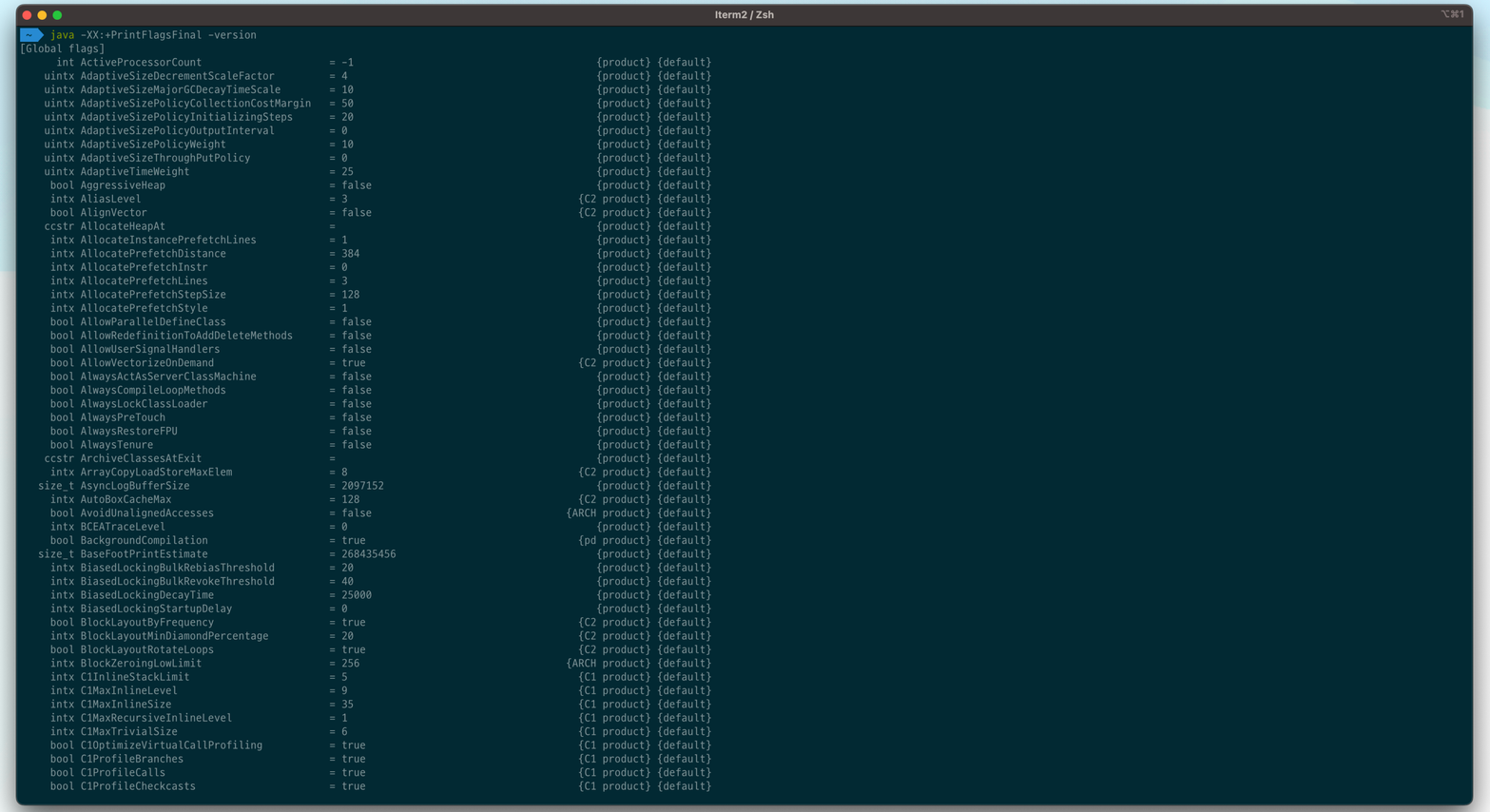
* Native Memory

- 네이티브 메모리는 OS 레벨에서 직접 관리되는 메모리 영역
  JVM 메모리 외에 추가 할당이 가능하며 이 경우 JVM의 최대 메모리 설정보다 더 많은 메모리를 사용하게 됨
- 네이티브 메모리 영역
  - Metaspace(Permanent Generation)
    로딩된 클래스의 메타데이터를 보관하는 영역 (힙 외부의 별도 영역)
    따라서 힙이 아닌 메타스페이스 메모리 설정 필요(`-XX:MetaspaceSize`, `-XX:MaxMetaspaceSize`)
  - Threads
    스레드가 사용하는 JVM stack, 일반적으로 약 1MB(`-Xss` 튜닝 플래그로 설정 가능)
    다른 영역과 달리 스레드 수 제한이 없다면 실질적으로 스택에 할당된 메모리는 제한이 없음
    작업을 수행하는 스레드 외에 GC처럼 별도의 내부작업을 위한 스레드도 필요
  - Code Cache
    JIT 컴파일러에 의해 생성된 네이티브 코드를 저장하는 영역(non-heap 영역)
    메모리 설정 가능(`-XX:InitialCodeCacheSize `, `-XX:ReservedCodeCacheSize`)
  - GC (Garbage Collection)
    GC 알고리즘과 함께 제공되며, 이 작업을 위해 일부 오프-힙 데이터 구조가 필요 (더 많은 네이티브 메모리를 필요로 함)
  - Symbols
    대표적으로 String 상수 풀(String 인터닝을 통해 비효율적인 메모리 사용을 줄임)이 있고 이를 네이티브 고정된 크기의 해시테이블에 저장
    메모리 설정 가능(`-XX:StringTableSize`), 런타임 상수 풀 또한 마찬가지로 여기에 해당함
  - Native Byte Buffers
    개발자가 직접 접근할 수 있는 네이티브 메모리 영역
    JNI 및 NIO 등에서 ByteBuffers를 통한 malloc 호출로 사용 가능

JVM과 메모리 구조

* Native Memory Tracking

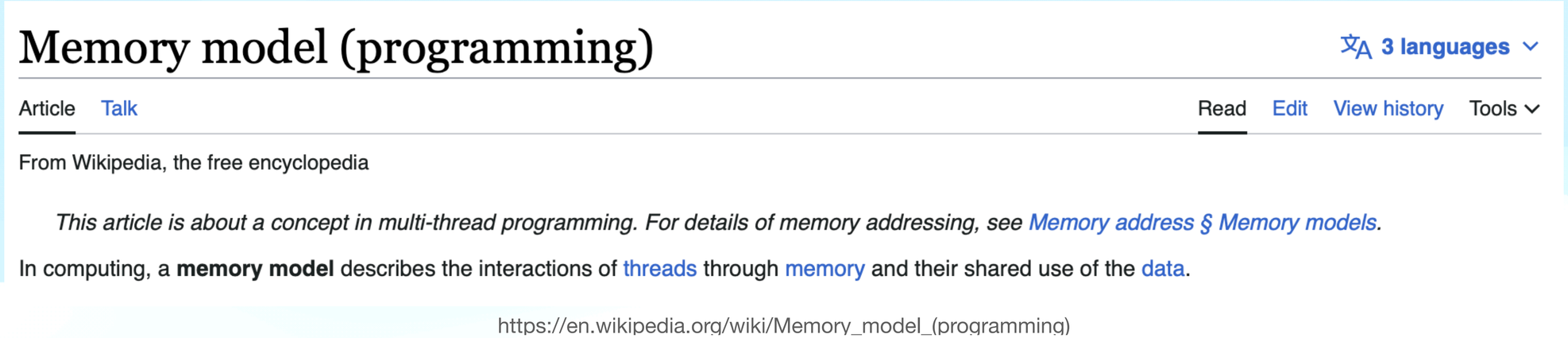
- 튜닝 플래그 출력하기 (`java -XX:+PrintFlagsFinal -version | grep <concept>`)



[2] Java 메모리 모델과 메모리 누수

Java 메모리 모델과 메모리 누수

Memory Model



Memory model (programming)

3 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article is about a concept in multi-thread programming. For details of memory addressing, see Memory address § Memory models.

In computing, a **memory model** describes the interactions of threads through memory and their shared use of the data.

https://en.wikipedia.org/wiki/Memory_model_(programming)

- 컴퓨팅에서 메모리 모델은
  메모리를 통한 스레드의 상호 작용과 데이터를 공유하며 사용하는 것을 설명하는 것
- 즉, 멀티스레드 환경에서 데이터를 공유하기 위해 메모리를 사용하는 방법에 대한 명세
- 이를 통해 더 많은 최적화(Loop fusion 등)가 가능하여
  잠재적으로 공유될 수 있는 데이터에 대한 명령문을 이동하기도 함

Java 메모리 모델과 메모리 누수

Java Memory Model

Java memory model

1 language

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

The **Java memory model** describes how threads in the Java programming language interact through memory. Together with the description of single-threaded execution of code, the memory model provides the semantics of the Java programming language.

The original Java memory model developed in 1995, was widely perceived as broken,[1] preventing many runtime optimizations and not providing strong enough guarantees for code safety. It was updated through the Java Community Process, as Java Specification Request 133 (JSR-133), which took effect back in 2004, for Tiger (Java 5.0).[2][3]

https://en.wikipedia.org/wiki/Java_memory_model

- Java 멀티스레드 환경에서 메모리 할당과 동작 방식을 정의하는 모델
  여러 스레드가 공유하는 데이터에 대한 가시성/접근성 등에 대한 명세
- 최신 플랫폼에서는 최적화 등을 이유로 명령이 순서대로 실행되지 않음(보장되지 않음)
  멀티 프로세서는 메인 메모리와 별개인 캐시 메모리를 소유할 수 있음
- 여러 스레드가 완벽히 동기화된 상태를 유지하는 것은 성능적으로 비용이 많이 듦

Java 메모리 모델과 메모리 누수

Java Memory Model - Specification

- Shared Variables
  여러 스레드에서 접근 가능한 공유될 수 있는 변수
- Actions
  읽기/쓰기, 잠금/해제 그리고 스레드 생성/종료를 포함한 프로그램에 의해 수행되는 동작
- Programs and Program Order
  각 스레드에서 수행되는 동작의 순서
- Synchronization Order
  동기화 작업들 간 순서
- Happens-before Order
  두 개의 동작이 순서가 있다면 첫번째 동작은 항상 두번째 동작에 영향을 미치며 순서가 변경되지 않음
- Executions
  모든 동작들의 순서를 포함한 프로그램의 실행을 의미하며, Synchronization Order와 Happens-before Order 따라 결정됨
- well-formed Executions
  JMM의 모든 규칙을 충족하는 실행을 의미
- Executions and Causality Requirements
  실행과 인과 관계에 대한 요구사항으로, 시간에 따라 동작이 어떻게 되는지 정의함
- Observable Behavior and Nonterminating Executions
  실행 중 볼 수 있는 동작과 종료되지 않고 계속 실행되는 동작에 대한 정의

Java 메모리 모델과 메모리 누수

Java Memory Model - Specification

- Shared Variables
  - 힙에 저장되는 모든 변수는 공유됨 (인스턴스 필드, 배열 요소 등이 저장되는 힙은 스레드 간 공유되는 영역)
  - 지역 변수와 매개 변수, 예외 핸들러 매개 변수 등은 스레드 간 공유되지 않음 (메모리 모델과 상관 없음)
  - 여러 스레드가 같이 엑세스하는 변수에 대해 한 스레드가 쓰기 작업을 하는 경우 엑세스가 충돌함
- Actions
  - 다른 스레드에서 감지되거나 직접 영향을 받을 수 있는 한 스레드의 작업
    - 읽기/쓰기 (non-volatile)
    - 동기화 작업 (Synchronization)
읽기/쓰기 (volatile)
잠금/해제 (monitor)
처음 또는 마지막 스레드 작업
스레드의 시작/종료를 감지하는 작업
    - 실행 외부에서 관찰할 수 있는 외부 작업
    - 스레드 분기 작업 (무한 루프 내에 스레드에 의해서만 가능)

Java 메모리 모델과 메모리 누수
Java Memory Model - Specification

- Programs and Program Order
  - 스레드 작업에서 수행되는 전체 순서 (모든 작업이 프로그램 오더와 일치하는 순서로 처리됨)
    - 읽기 작업 후 쓰기 작업 실행 (쓰기 작업 전 필요한 쓰기 작업 같은 건 없음)
  - 순차 일관성 (Sequential consistency)
    - 프로그램 오더와 일치하는 모든 작업은 전체적인 실행 순서가 있으며
      각 작업은 원자적이며 모든 스레드에서 즉시 볼 수 있음 (가시성과 순서에 대한 보증)
    - 데이터에 대한 레이스 컨디션이 없다면 프로그램은 내 작업은 모두 일관적이고 순차적으로 실행됨
    - 따라서 원자적 작업으로 처리되어야 하지만 그렇지 않은 작업의 오류도 허용
- Synchronization Order
  - 실행되는 모든 작업에 대한 전체 동기화 순서
    - 모니터 락 해제 시 다른 스레드들의 뒤이은 잠금 작업이 순서대로 처리됨
    - volatile 변수에 대한 쓰기는 해당 변수를 참조하는 모든 스레드들이 읽을 때 동기화 됨
    - 스레드의 생성과 시작은 첫 스레드의 작업 전에 완료되어야 함
    - 모든 스레드의 첫 작업 전에 각 변수가 초기값으로 초기화되어야 함
    - 해당 스레드의 작업이 모두 완료된 후에 다른 스레드에게 종료 시점을 알려야 함 (isAlive, join 등)
    - 스레드 간 인터럽트가 발생했을 때 인터럽트 작업이 완료된 후에 다른 스레드에서 발생시점을 알 수 있어야 함

Java 메모리 모델과 메모리 누수

Java Memory Model - Specification

- Happens-before Order
  - 모든 작업의 작업 간 순서 (데이터에 대한 레이스 컨디션 시점을 정의)
    - 여러 스레드 중 먼저 온 스레드가 선행 작업
    - 객체의 생성자가 소멸자보다 선행 작업
    - 1번 작업이 2번 작업과 Happens-before Order 관계가 된다면
      1번 작업이 2번 작업보다 먼저 발생해야 함
    - 작업 순서가 `스레드1 -> 스레드2` 이고 `스레드2 -> 스레드3` 이면
      스레드1이 스레드3 보다 선행 작업
  - 작업 간 순서가 있지만 항상 순서대로 발생하지는 않음
    상관없는 작업이라면 순서대로 일어나지 않은 것처럼 보여도 무관함
  - 작업 간 순서 지정
    - 모니터 락 해제는 그 이후 발생하는 모든 락 시도 전에 발생
    - volatile 필드에 쓰기 작업은 그 후에 발생하는 모든 읽기 작업보다 먼저 발생
    - 스레드 start 메서드 호출은 시작된 스레드의 어떤 작업보다 먼저 발생
    - 스레드의 모든 작업은 다른 스레드에 의해 join 메서드가 성공을 반환하기 전에 발생
    - 모든 객체의 기본 초기화는 앱의 그 어떤 작업보다 먼저 발생
  - 배열 길이를 읽거나 가상 메서드 호출 등과 같은 작업은 데이터에 대한 레이스 컨디션의 영향을 받지 않음

Java 메모리 모델과 메모리 누수

Java Memory Model - Specification

- Executions
  - 실행은 아래 요소들의 집합
    - 프로그램
    - 작업의 집합
    - 프로그램 순서 (각 스레드가 수행하는 모든 작업에 대한 순서)
    - 동기화 순서 (모든 동기화 작업에 대한 순서)
    - `Write` 함수 (각 읽기 작업에서 어떤 쓰기 작업을 보았는 지 반환하는 함수)
    - `Value` 함수 (각 쓰기 작업에서 쓰여진 값을 반환하는 함수)
    - 동기화 작업 순서 (동기화 작업 간 부분 순서)
    - 작업 순서 (작업 간 부분 순서)
- well-formed Executions
  - 정형화가 잘된 실행의 조건
    - 어떤 변수에 대한 각 읽기 작업은 동일한 쓰기 작업 결괏값을 볼 수 있어야 함
    - volatile 변수에 대한 모든 R/W 작업은 volatile 작업이어야 함
    - 모든 작업 간엔 발생 순서가 있어야 함
    - 각 스레드의 작업 결과는 어떠한 간섭도 받지 않았을 때의 수행 결과와 동일 해야 함
    - 작업 순서에 대한 일관성이 있어야 함
    - 동기화 순서가 일관성이 있어야 함 (volatile 변수의 Read 작업 조건을 만족해야 함)

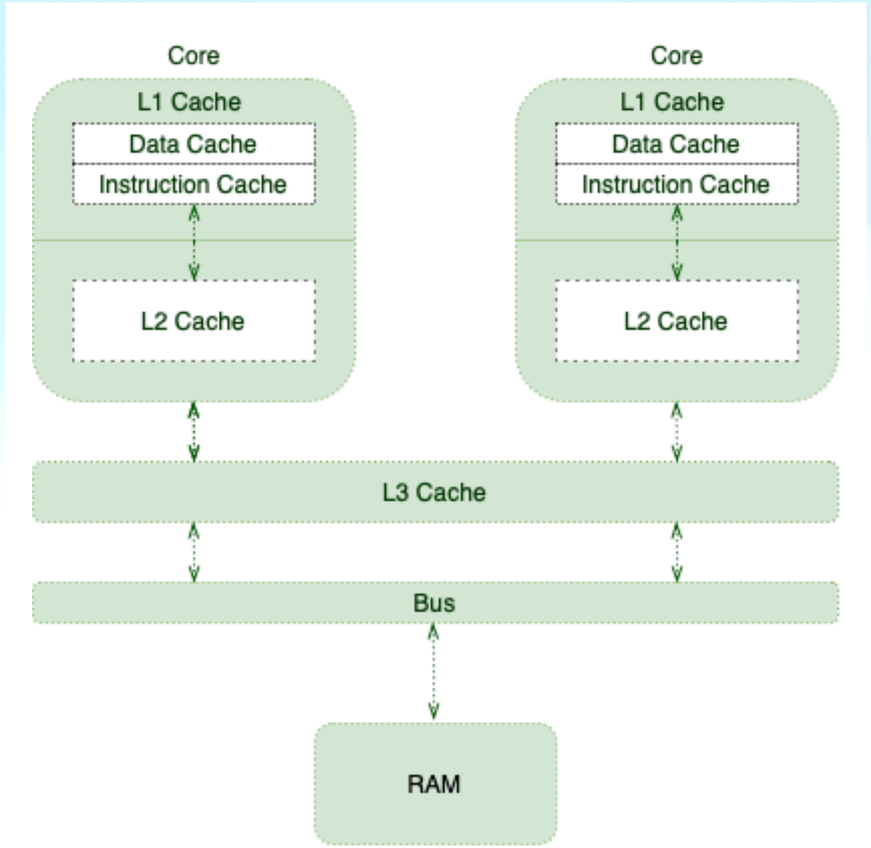
Java 메모리 모델과 메모리 누수

Java Memory Model - Specification

- Executions and Causality Requirements (실행과 인과 관계 요구 사항)
- Observable Behavior and Nonterminating Executions
  - 유한 외부 작업 집합은 관찰 가능한 동작
    외부 동작은 프로그램 출력 등 외부 환경에 영향을 미치는 동작
  - 중단 동작(hang action)은 외부 동작 등 무한 실행하는 동작을 표현
    모든 스레드 차단 또는 무한 실행은 프로그램이 종료될 수 있음
  - 잠금 획득 또는 외부 작업 시 스레드가 차단되는 경우 스레드가 무기한 차단되며
    실행이 종료되지 않을 수 있음

Java 메모리 모델과 메모리 누수

* Shared Multiprocessor Architecture



https://www.baeldung.com/java-volatile

Java 메모리 모델과 메모리 누수

Memory Consistency & Memory Visibility

- 메모리 일관성 (Memory Consistency)
  - 서로 다른 스레드가 하나의 데이터에 접근할 때 일관적인 상태를 보장하는 속성
    Happens-Before를 통해 공유 메모리에 대한 일관성(연산, 순서 등) 보장
    - 메커니즘을 파헤치기보다는 피하기 방법을 익히는 것이 중요
  - 데이터를 읽을 때 마지막 데이터를 읽지만 아무도 데이터의 일관성을 보장해주지 않음
  - 메모리 일관성 에러
    - 여러 스레드가 동일 데이터를 읽을 때 일관적이지 않은 값을 읽는 경우 발생
    - 대표적인 예시로 int 타입의 변수에 대한 증감 연산이 있음
- 메모리 가시성 (Memory Visibility)
  - 멀티스레드 환경에서 한 스레드에서 변경한 값을 다른 스레드에서 언제 보게 될지를 정의한 것
  - 어떤 스레드에 의해 값이 변경되었을 때 다른 스레드가 가장 최신 값을 읽을 수 있도록 하는 속성

Java 메모리 모델과 메모리 누수

final field

- final 필드는 별도의 동기화 처리가 필요 없기 때문에 세이프한 불변 객체 구현을 지원하며
  JIT 컴파일러는 레지스터에 최종 캐시 값을 유지할 수 있음 (다시 로딩할 필요가 없음)
- final 필드는 스레드 간 레이스 컨디션 상황에서도 모든 스레드에서 불변으로 간주됨
- 생성자 호출이 완료되면 객체의 초기화(인스턴스화) 되었다고 간주
  초기화가 완료된 객체의 참조만 볼 수 있는 스레드는 올바르게 초기화된 final 필드를 볼 수 있음
- 생성자가 다른 생성자를 연이어 호출한 상황에서 호출된 생성자가 final 필드를 초기화하면
  필드에 초기화는 호출된 생성자가 끝나면서 확정됨

Java 메모리 모델과 메모리 누수

volatile

- volatile 키워드는 모든 스레드에서 레지스터가 아닌 메인 메모리로부터 값을 읽고 쓰도록 강제함
  공유되는 변수의 가시성을 보장해줌
- 현대의 멀티 프로세서 아키텍처는 쓰기 작업은 바로 갱신하지 않고
  버퍼에 잠시 대기 시킨 후 추후 일괄적으로 메인 메모리에 적용함
- 하지만 멀티스레드 환경에서 원자성을 보장해주지는 않음
  작업 간 재정렬(reordering) 문제를 해결해주지 않기 때문에 고유락(Intrinsic Lock)을 사용해야 함

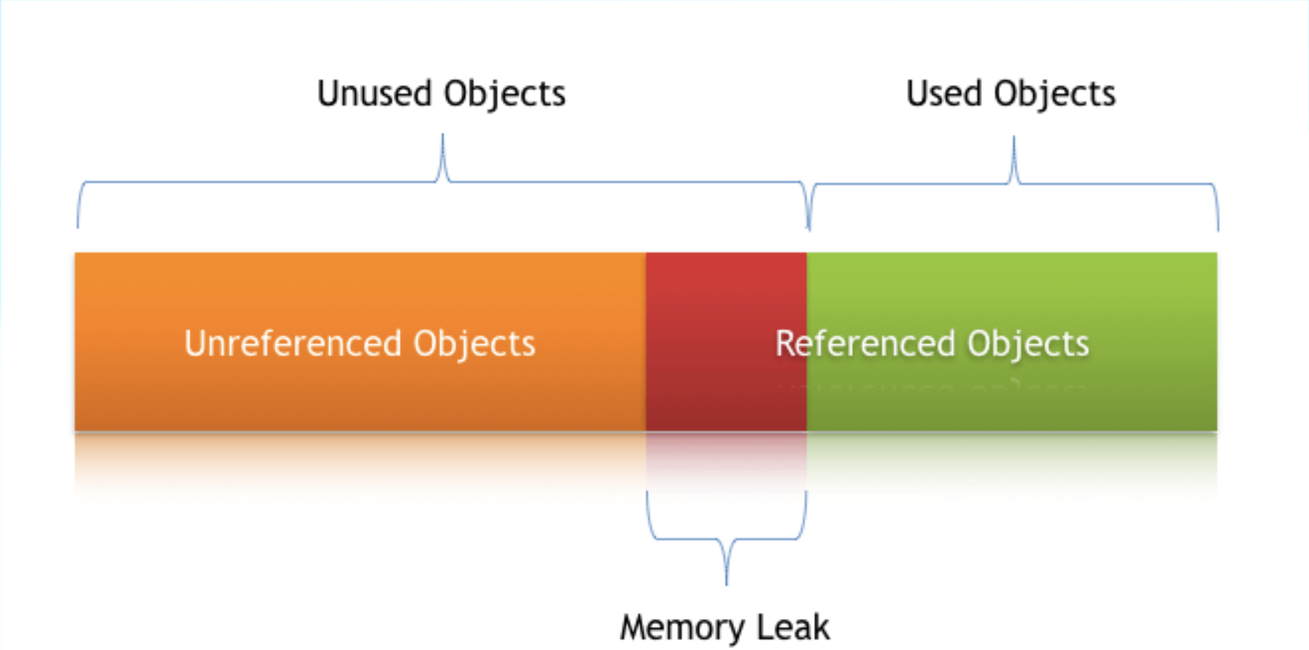
Java 메모리 모델과 메모리 누수

Memory leak

- 필요하지 않은 메모리를 계속 참조(점유)하여 해제를 하지 못해서 발생하는 누수
  즉, 할당된 메모리를 잘못 관리하여 생기는 리소스 낭비
- 메모리에 할당되어 있지만 실행 코드에서 접근할 수 없는 경우에도 발생
- 메모리 누수는 결과적으로 시스템 성능을 저하시키고 치명적인 에러를 발생시킬 확률이 높음
- 메모리 누수는 메모리 노화의 기여하는 요인이 되기도 함

Java 메모리 모델과 메모리 누수

Java Memory leak



https://www.baeldung.com/java-memory-leaks

Java 메모리 모델과 메모리 누수

Java Memory leak

- JVM은 GC를 통해 자동으로 메모리가 관리되기 때문에 웬만한 누수는 방지할 수 있음
- 그러나 완벽하지는 않기 때문에 종종 메모리 누수가 발생함
- 힙에 더 이상 사용되지 않는 객체를 GC가 제거할 수 없을 때 발생
- 메모리 누수가 오래 지속되면 실행중인 앱이 OutOfMemoryError로 종료될 수 있음
- 힙 영역에는 참조, 비참조 유형의 객체가 존재하고 이에 따라 GC의 제거 여부가 나뉨
  즉, 참조중인 객체는 제거하지 않아 메모리 누수가 발생할 수 있음
- 증상
  - 앱이 지속적으로 실행되는 경우에 심각한 성능 저하
  - 앱의 힙에서 OutOfMemoryError 발생
  - 앱 내에 이상한 충돌 현상이 발생
  - 간헐적으로 연결 객체가 부족한 상황 발생

Java 메모리 모델과 메모리 누수

Java Memory leak types

- <스태틱 필드 객체>로 인해 발생
  사용하지 않아도 참조되고 있기 때문에 누수가 발생할 수 있음
  - 최대한 스태틱 필드 객체의 사용을 지양할 것
- <사용한 리소스를 해제(반환 등)하지 않아서> 발생
  새로 생성한 리소스에 할당된 메모리를 해제하지 않거나 해제 전에 예외가 발생하면 누수가 발생할 수 있음
  - finally 블록을 통해 리소스 해제(try-with-resources), 또한 해제하는 코드 블록에도 예외가 발생하지 않게 처리
- <equals, hashCode 메서드를 올바르게 구현하지 않아서> 발생
  이로 인해 hashCode를 활용하는 HashSet, HashMap 등에서 불필요한 참조가 발생해 누수가 발생할 수 있음
  또한 위 메서드들에 의존하는 로직으로 인해 불필요한 인스턴스를 생성할 수 있음
  - 가급적 항상 equals, hashCode 메서드를 재정의할 것
- <아우터 클래스를 참조하는 이너 클래스(non-static)>로 인해 발생
  이 경우 이너 클래스를 사용하기 위해 항상 아우터 클래스를 참조하기 때문에 아우터 클래스에 대한 누수가 발생할 수 있음
  - 아우터 클래스를 참조하지 않는 경우 이너 클래스를 static 클래스로 구현할 것

Java 메모리 모델과 메모리 누수

Java Memory leak types

- <finalize 메서드>를 통해 발생
  해당 메서드가 오버라이딩 되어도 GC에 의해 바로 제거되는 것이 아닌 대기열에 삽입되는데
  이때 작성된 코드가 올바르지 않다면 누수가 발생할 수 있음
  - 소멸자가 필요한 설계/구현을 지양할 것
- <String의 intern>으로 인해 발생 (JDK 6 이하일 때 발생)
  크기가 큰 문자열에 대해 intern 메서드를 호출하면 PermGen 메모리 영역의 스트링 풀로 이동되는데
  이곳은 GC가 처리되지 않아 누수가 발생할 수 있음
  - 최신 버전에 JDK를 사용하거나 예전 버전이라면 `-XX:MaxPermSize=512m` 옵션처럼 크기를 조정할 것
- <ThreadLocal 클래스>로 인해 발생
  ThreadLocal 클래스를 사용해 데이터를 스레드별로 사용할 때 명시적으로 값을 제거하지 않으면 참조가 유지되어 누수가 발생할 수 있음
  현대 서버 앱에서는 스레드를 재활용하는 스레드풀을 활용하기 때문에
  - remove 메서드를 호출하여 명시적으로 참조를 제거할 것
    예외 상황에서도 finally 블록 사용해서 제거할 것
  - 값을 비우는데 `ThreadLocal.set(null)`을 사용하지 말 것 (실제로 값을 지우는 것이 아님)

Java 메모리 모델과 메모리 누수

Java Memory leak prevention

- VisualVM, YourKit 등과 같은 프로파일러를 활용한 모니터링
  - 또는 `-verbose:gc` 옵션 사용으로 GC 모니터링
- 참조 객체를 통해 메모리 누수 방지
  - `java.lang.ref` 패키지에 참조 객체 활용
- 벤치마크를 활용해 Java 코드의 성능 모니터링
- 코드 리뷰

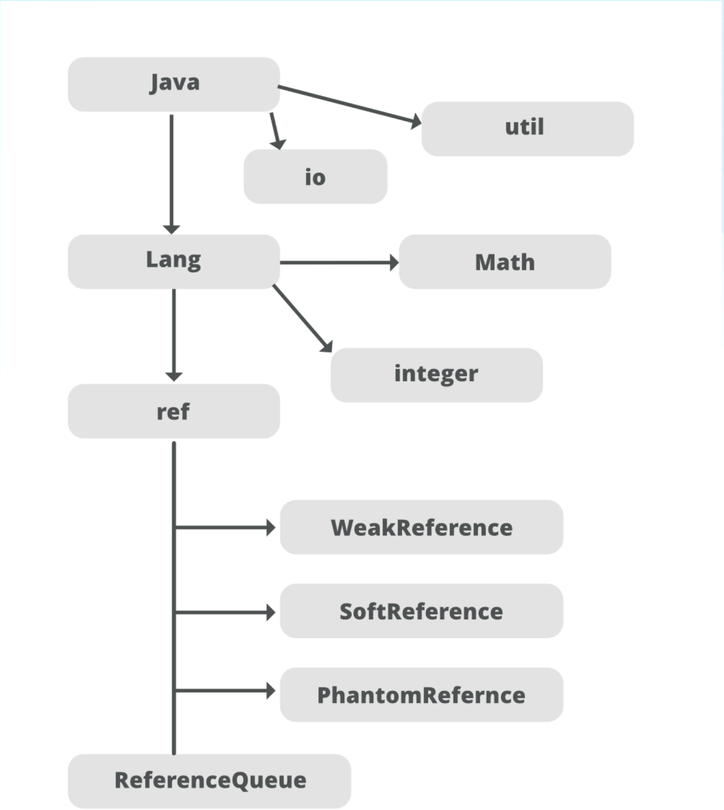
Java 메모리 모델과 메모리 누수

* Memory Optimization Tip

- 컬렉션 사용 시 다음과 같은 사항들을 고민해볼 것
  - 가능하다면 초기 사이즈를 지정해 초기화
  - 컬렉션 별 CRUD에 대한 시간 복잡도를 확인해서 의도와 맞게 적절한 컬렉션을 사용할 것
  - HashMap 사용시 리사이징이 일어나면 해시값을 다시 조정하기 때문에 이를 고려할 것
- 가능하다면 특정 객체의 재사용 (메모리 사용량, GC 빈도 등)
  - 불변 객체나 캐싱 형태의 구현을 활용해 볼 것
- 박싱/언박싱을 가급적 피할 것
- Stream API 사용시 메모리를 더 사용하거나 성능이 더 느릴 수 있어 주의할 것

Java 메모리 모델과 메모리 누수

* Reference Object



https://www.geeksforgeeks.org/java-lang-ref-phantomreference-class-in-java/

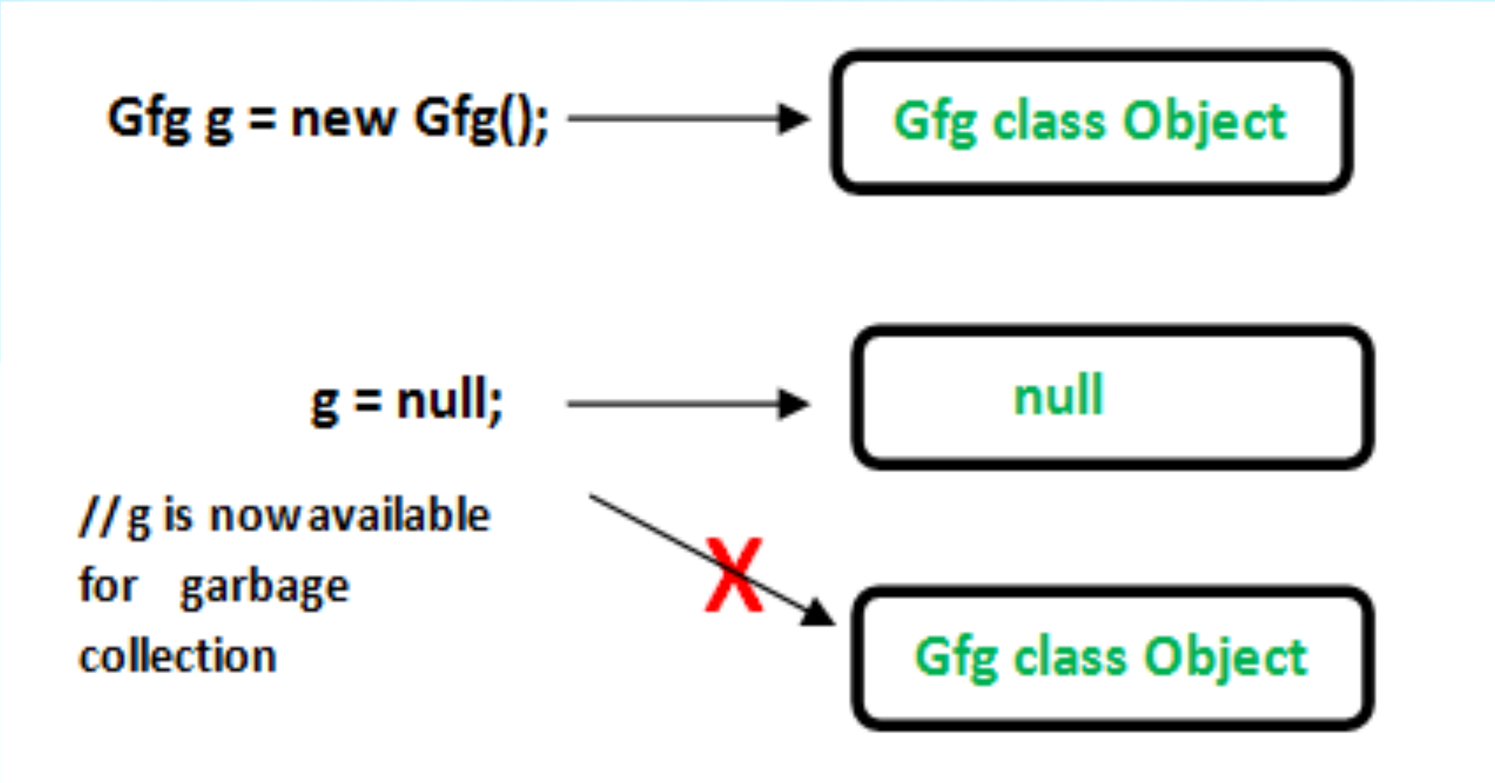
Java 메모리 모델과 메모리 누수

* Reference Object

- Java에서 지원하는 참조 객체(java.lang.ref.Reference 서브 클래스)로 다양한 객체 참조의 대한 타입
- GC 수집 방식의 기준이 되는 레퍼런스 타입으로 아래와 같이 4가지가 존재 (모두 Heap 영역에 위치)
  - Strong(Hard) Reference
    개발자가 흔히 사용되는 형태로 이 경우 GC에 의해 수집이 되지 않음
  - Soft Reference
    메모리가 부족한 경우에만 GC에 의해 수집이 되는 참조 형태
  - Weak Reference
    항상 GC에 의해 수집 대상이 되는 참조 형태
  - Phantom Reference
    직접 참조할 수 없고 별도의 참조 큐가 필요한 참조 형태
- ReferenceQueue
  위 참조를 통한 참조 객체들이 GC에 의해 수집되는 타이밍을 트래킹하기 위해 사용되는 큐
  이를 통해 객체의 생명주기를 추적하거나 캐싱,롤링,폴링 등 다양한 구현하는데 활용됨

Java 메모리 모델과 메모리 누수

* Reference Object - Strong(Hard) Reference



https://www.geeksforgeeks.org/types-references-java/

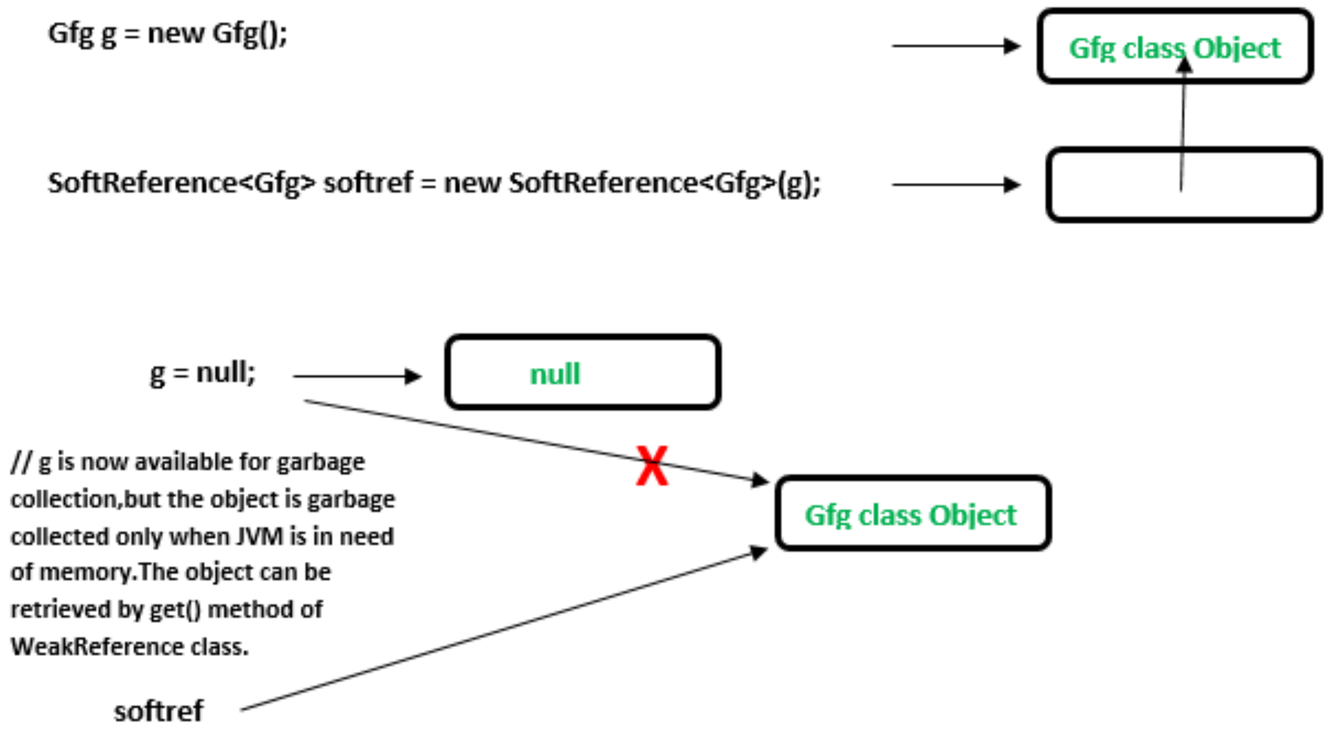
Java 메모리 모델과 메모리 누수

* Reference Object - Strong(Hard) Reference

- 일반적으로 사용하는 객체 참조 타입
  `List<String> list = new ArrayList<>();`
- 이 타입으로 참조되고 있는 객체는 GC 수집 대상이 되지 않음

Java 메모리 모델과 메모리 누수

* Reference Object - Soft Reference



https://www.geeksforgeeks.org/types-references-java/

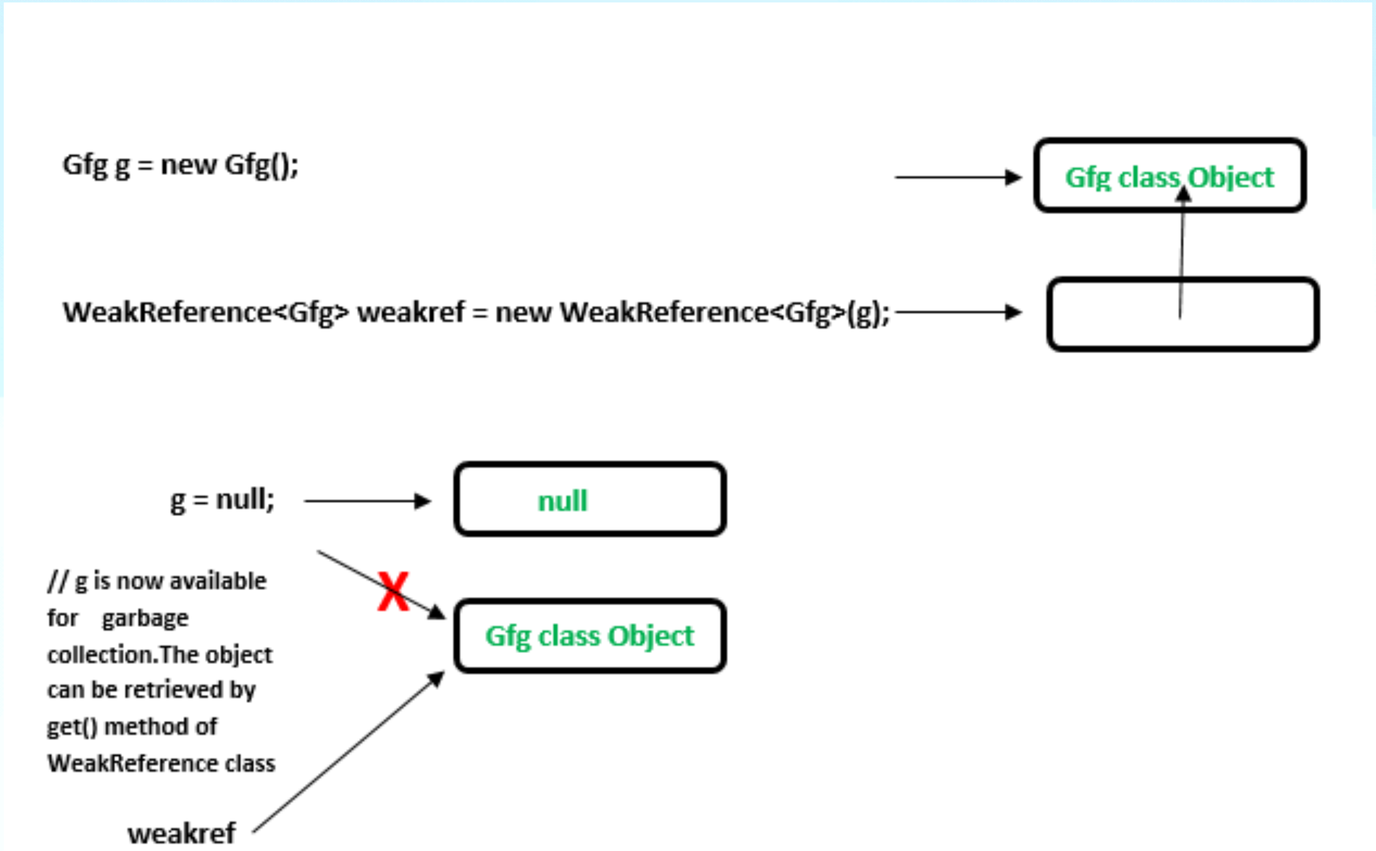
Java 메모리 모델과 메모리 누수

* Reference Object - Soft Reference

- 메모리가 필요한 양을 기준으로 GC 수집이 결정되는 참조 형태
  대표적인 사용 예시로는 캐시 구현 등이 있음
  아래 설명은 GC가 해당 객체에 softly reachable 하다고 가정
- GC에 의해 제거되는 타이밍이 정해짐
- 일반적으로 OutOfMemoryError 발생 전 모두 지워지며
  메모리 상태가 부족하지 않다면 지워질 타이밍은 미정
  - 제거될 때 연결된 모든 soft 참조를 함께 제거할 지 선택 가능
- 제거된 Soft 참조는 ReferenceQueue에 삽입됨
- 일반적으로 JVM은 최근에 생성(참조)된 객체를 먼저 지우도록 설계됨
- 해당 클래스를 통해 간단한 캐시 구현이 가능하며
  해당 클래스(서브 클래스 포함)는 상황에 따라 더 큰 데이터 구조에서 사용될 수 있음
- 실제로 꺼내서 사용중인 참조는 지워지지 않음 (꺼낸 시점에서 strong 참조)
  - 그래서 정교한 캐시를 생성할 수 있음 (가장 최근 참조가 삭제되는 것을 방지시킬 수 있음)

Java 메모리 모델과 메모리 누수

* Reference Object - Weak Reference



https://www.geeksforgeeks.org/types-references-java/

Java 메모리 모델과 메모리 누수

* Reference Object - Weak Reference

- Weak 레퍼런스는 자신이 종료 가능한 상태가 되거나 종료(회수)되는 것을 막지 않는 타입
  대표적인 사용 예시로는 정규화 매핑 구현, 수명이 짧은 캐시 구현 등이 있음
  아래 설명은 GC가 해당 객체에 weakly reachable 하다고 가정
- GC는 strong, soft 참조로 닿을 수 있는 weak 참조를 모두 지우며
  동시에 이전에 모든 weak 참조 객체들을 종료 가능한 상태로 지정
- 제거된 weak 참조는 ReferenceQueue에 삽입됨

Java 메모리 모델과 메모리 누수

* Reference Object - Phantom Reference

- GC가 참조 객체를 회수가 가능하다고 판단해 대기열에 추가할 수 있는 참조 타입
  대표적인 사용 예시로는 사후 정리 작업 예약 시 활용
  아래 설명은 GC가 해당 객체에 phantom reachable 하다고 가정
- 모든 phantom 참조 객체와 해당 객체에서 phantom 참조 객체로 연결된 모든 참조를 원자적으로 제거
- 제거된 phantom 참조는 ReferenceQueue에 삽입됨
- phantom 참조의 특징은 참조 대상을 가져올 수 없는 상태여야 GC에 의한 회수가 유지된다는 것
  즉 이 참조의 get 메서드는 항상 null을 반환함
- 결론적으로 참조 대상이 메모리에서 완전히 제거 되기 전 작업 수행과 알림 제공하는 등
  고급 메모리 관리 작업 등에 활용됨

[3] 스레드덤프를 통한 스레드의 상태 정보 확인

스레드덤프를 통한 스레드의 상태 정보 확인

Thread dump

- Java 프로세스의 모든 스레드 상태(활동)에 대한 스냅샷
  스레드의 문제를 진단할 때 유용하게 사용됨
- 각 스레드의 상태는 스레드 스택의 추적과 함께 표시
- 스레드덤프는 일반 텍스트로 작성되어 내용을 파일에 저장, 텍스트 편집기로 확인 가능
- 스레드덤프 방법
  - jstack
    가장 빠르고 쉬운 방법이지만 최신 JDK에선 더 좋은 방법들이 있음
  - JMC (Java Mission Control)
    프로파일링에 따라 붙는 성능 오버헤드를 최소화한 프로파일링 도구
  - jvisualvm
    GUI를 포함한 경량 오픈소스 프로파일링 도구
  - jcmd
    jstack(스레드 덤프), jmap(힙 덤프), jinfo(시스템 속성) 등을 제공하는 도구
  - jconsole
    스레드 스택 트레이스 정보를 검사할 수 있음

[4] 힙덤프를 통한 힙 메모리 확인

힙덤프를 통한 힙 메모리 확인

Heap dump

- 특정 순간에 JVM 메모리 상에 있는 모든 객체의 스냅샷
- 힙 영역에 대한 스냅샷을 통해 앱 실행 시 힙 메모리 사용량 분석 가능
  메모리 누수, GC 이슈 등을 해결하는 데 활용
- 힙덤프 방법
  - jmap
  - VisualVM
  - OutOfMemoryError 발생 시 자동 생성 (옵션이 활성화가 된 경우)
    `java -XX:+HeapDumpOnOutOfMemoryError <ClassName>`

Java 메모리 모델과 메모리 누수

* JVM heap dump parameters

- -XX:+HeapDumpOnOutOfMemoryError
  - OutOfMemoryError가 발생하면 물리 파일에 힙 덤프
- -XX:HeapDumpPath=./java_pid<pid>.hprof
  - 파일이 기록될 경로 지정, 모든 파일명 지정 기능
  - 하지만 JVM이 파일명에서 <pid> 태그를 발견하면
    OutOfMemoryError를 발생시키는 현재 프로세스의 pid가 파일명에 추가되며 `.hprof` 포맷이 됨
- -XX:OnOutOfMemoryError="< cmd args >;< cmd args >"
  - OutOfMemoryError 발생 시 긴급 실행할 명령
- -XX:+UseGCOverheadLimit
  - OutOfMemoryError 발생 전에 GC에서 소비되는 VM 시간 비율을 제한

Java 메모리 모델과 메모리 누수

* JVM important parameters

- Heap 크기 지정
  - `-Xms<size>[unit]`, `-Xmx<size>[unit]`
- Metaspace 크기 지정 (JDK 8부터 지정되지 않음, 자동으로 조절되나 지정 가능)
  - `-XX:MaxMetaspaceSize=<size>[unit]`
- Young Generation 크기 지정 (최소 크기 1310MB, 최대는 무제한)
  - `-XX:NewSize=<young size>[unit]`, `-XX:MaxNewSize=<young size>[unit]`
- GC 알고리즘 선택
  - `-XX:+UseSerialGC`, `-XX:+UseParallelGC`, `-XX:+USeParNewGC`, `-XX:+UseG1GC`
- GC 로깅 (활동 기록)
  모니터링을 위해 데몬이 추가되면서 성능 오버헤드가 발생해 프로덕션에선 사용을 권장하지 않음
  - `-XX:+UseGCLogFileRotation` (log4j 등으로 로그 파일의 롤링 정책)
  - `-XX:NumberOfGCLogFiles=< number of log files >` (단일 앱 주기 동안 기록 가능한 최대 로그 파일 개수)
  - `-XX:GCLogFileSize=< file size >[ unit ]` (파일 최대 크기)
  - `-Xloggc:/path/to/gc.log` (해당 위치)

Java 메모리 모델과 메모리 누수

* JVM Misc parameters

- -server
  - 서버 Hotspot VM 활성화 (64비트 JVM에서 기본적으로 적용됨)
- -XX:+UseStringDeduplication
  - Java 8u20의 불필요한 문자 인스턴스 생성을 줄여 힙 메모리 최적화
    중복 문자열 값을 char[] 배열을 통해 처리
- -XX:+UseLWPSynchronization
  - 스레드 기반 동기화 대신 LWP(Light Weight Process) 기반 동기화 정책 설정
- -XX:LargePageSizeInBytes
  - 힙에 사용되는 대형 페이지 크기 설정
- -XX:MaxHeapFreeRatio
  - 힙의 축소를 방지하기 위해 GC 후 사용 가능한 최대 힙 비율 설정
- -XX:MinHeapFreeRatio
  - 힙의 확장을 방지하기 위해 GC 후 사용 가능한 최소 힙 비율 설정

Java 메모리 모델과 메모리 누수

* JVM Misc parameters

- -XX:SurviorRatio
  - eden/survivor 공간 크기 비율
- -XX:+UseLargePages
  - 대용량 페이지 메모리 사용 (시스템 지원이 되는 경우에만)
- -XX:+UseStringCache
  - 일반적으로 스트링 풀에서 사용할 수 있는 할당된 문자열의 캐싱 활성화
- -XX:+UseCompressedStrings
  - 문자열 압축을 위해 ASCII 포맷으로 표현 가능한 스트링 객체에 byte[] 유형 사용
- -XX:OptimizeStringConcat
  - 문자열 연결 작업을 최적화 (가능한 경우에만)

Java 메모리 모델과 메모리 누수

* APM & Profiling

- APM(Application Performance Management)
  - 앱의 성능, 가용성 등을 모니터링 또는 관리하는 것
- Profiling
  - 메모리 사용량, 시간 복잡도, 특정 명령의 호출 빈도 등을 측정하는 분석하는 것
    즉 런타임 동안에 앱에서 어떤일이 발생하는 지에 대한 정보를 제공
  - 일반적으로 프로파일링에 측정된 정보는 최적화에 활용 (특히 성능 최적화 측면)
  - 프로파일러는 이벤트, 통계, 시뮬레이션 등 다양한 기술을 지원함
- 차이
  - Profiling은 개발자가 성능 문제(코드 레벨)를 최적화하는데 유용
  - APM은 한단계 높은 수준(앱 레벨)에서 성능을 모니터링, 분석 (사용자 경험 추적 등 포함)

Java 메모리 모델과 메모리 누수

* Profiling Tools

- VisualVM
- JProfiler
- YourKit
- JFR/JMC
- Redhat thermostat
- Newrelic

실습

아하! 모먼트

- [내가] 채용 공고를 볼 때 중요하게 보는 항목은?

[내가] 채용 공고를 볼 때 중요하게 보는 항목은?

채용 공고 항목 정리

- 회사 소개, 주력 서비스나 상품 (+ 투자 현황, 향후 목표 등)
- 자본금, 투자 단계
- 채용하는 포지션, 맡게 될 업무
- 자격 요건과 우대 사항
- 채용 프로세스
- 업무환경이나 문화
- 혜택/복지
- 사무실 위치
- 기타
  - 기재된 채용 공고의 분량
  - 사용한 단어,어휘,표현 등

[내가] 채용 공고를 볼 때 중요하게 보는 항목은?
나에게 필요한 게 뭘까?

1. 기술 스택
    - 이를 통해 대략적인 지원하는 팀의 아키텍처, 프로세스, 팀 문화 등을 유추해볼 수 있음
    - 내가 자신 있는 기술인가? 아니면 하고 싶은 기술인가? (향후에도 내가 가져가고 싶은 기술이라면 더욱 좋음)
    - 포지션이 애매하지는 않은가? (풀스택, 또는 기획, 영업 등)
2. 업무 환경
    - 적어도 사람 관계에서 스트레스를 받지 않는 환경인가?
    - 업무 프로세스가 효율적일 것인가?
    - 변화나 발전을 추구하는 방향과 속도가 나와 맞는가?
3. 연봉과 복지 (별도로 기재된 연봉 테이블 및 복지)
    - 자체적인 연봉테이블이 있고 또한 이 테이블이 다른 회사들과 비슷한가?
    - 복지는 의외로 사내 문화를 대변하는 척도
    - 복지를 통해 지출을 절약해야 함 (도서/교육비 등 자기개발비 지원)
4. 회사의 재정 상황 (재무제표)
    - 회사의 기본 자본금, 투자 단계, 영업 실적(흑자 여부) 등으로 회사의 런웨이 기간을 예상해 볼 것 (정보가 없다면 별도로 찾아볼 것)
    - 스타트업을 지원하는 경우 임금 체불이 더이상 남 얘기가 아님 (돈보다도 시간과 정신적 스트레스가 더 큰 문제)
5. 자격 요건과 채용 프로세스
    - 자격 요건에서 내가 잘하는 기술을 기준으로 이력서 수정과 면접 주도 시뮬레이션 연습
    - 반대로 내가 못하거나 모르는 기술은 최대한 찾아보면서 준비
    - 이력서에서 뺄 기술과 경험이 있는 지 검토
    - 현재 내가 이 채용 프로세스를 모두 수용할 능력, 여유, 시간 등이 되는가?
6. 도메인과 주력 서비스(상품)
    - 기술과 함께 업무 속도(생산성)에 영향을 크게 미치는 요소
    - 특정 도메인들은 해당 도메인에 대한 경력이 중요한 경우도 많음
7. 사무실 위치
    - 출퇴근 시간과 환승 빈도 > 출퇴근 시간을 활용할 수 있는가의 문제
    - 또한 출근 전, 퇴근 후에 시간을 개발이나 영어 공부, 운동 등 자기개발에 투자할 수 있는가가 달려있음
8. 외근, 출장 빈도

참고 및 출처

- https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html
- https://www.oracle.com/java/technologies/javase/jdk-faqs.html
- https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
- https://www.geeksforgeeks.org/jvm-works-jvm-architecture/
- https://dzone.com/articles/jvm-architecture-explained
- https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html#jvms-2.6.1
- https://blogs.oracle.com/javamagazine/post/java-class-file-constant-pool
- https://www.baeldung.com/java-stack-heap
- https://www.geeksforgeeks.org/run-time-stack-mechanism-java/
- https://www.baeldung.com/native-memory-tracking-in-jvm
- https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html
- https://en.wikipedia.org/wiki/Memory_model_(programming)

- https://en.wikipedia.org/wiki/Loop_fission_and_fusion
- https://en.wikipedia.org/wiki/Java_memory_model
- https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html
- https://www.baeldung.com/native-memory-tracking-in-jvm
- https://docs.oracle.com/en/java/javase/17/vm/native-memory-tracking.html
- https://docs.oracle.com/en/java/javase/17/docs/specs/man/java.html
- https://www.oracle.com/java/technologies/javase/vmoptions-jsp.html
- https://docs.oracle.com/javase/specs/jls/se17/html/jls-17.html
- https://www.baeldung.com/java-final-performance
- https://www.baeldung.com/java-volatile
- https://en.wikipedia.org/wiki/Memory_leak
- https://www.baeldung.com/java-threadlocal

- https://www.baeldung.com/java-profilers
- https://en.wikipedia.org/wiki/Application_performance_management
- https://en.wikipedia.org/wiki/Profiling_(computer_programming)
- https://www.baeldung.com/java-microbenchmark-harness
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ref/Reference.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ref/SoftReference.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ref/WeakReference.html
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ref/PhantomReference.html
- https://www.baeldung.com/java-reference-types
- https://www.baeldung.com/java-soft-references
- https://www.baeldung.com/java-weak-reference
- https://www.baeldung.com/java-phantom-reference

- https://www.geeksforgeeks.org/types-references-java/
- https://www.geeksforgeeks.org/java-lang-ref-softreference-class-in-java/
- https://www.geeksforgeeks.org/java-lang-ref-weakreference-class-in-java/
- https://www.geeksforgeeks.org/java-lang-ref-phantomreference-class-in-java/
- https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/using_threaddumps.html
- https://www.baeldung.com/java-thread-dump
- https://www.baeldung.com/java-heap-dump-capture
- https://www.baeldung.com/jvm-parameters
- https://docs.oracle.com/en/java/javase/17/gctuning/factors-affecting-garbage-collection-performance.html#GUID-5508674B-F32D-4B02-9002-D0D8C7CDDC75
- https://visualvm.github.io/documentation.html