

Java 짹 잡아!

- JVM부터 GC, 스레드 동기화까지!

[1-1] Java의 정의와 동작 방식에 대해 살펴봅니다.

김재녕 (2023-05-27)

[0] OT

OT

강의 정보

- 이메일
 - jaenyeong.dev@gmail.com
- 깃허브
 - https://github.com/jaenyeong/Teach_Wanted-PreOnBoarding-Backend-Challenge

OT

다루는 내용

- JVM과 메모리 구조
- 클래스 로딩 등 Java의 동작 방식
- Java 바이트코드와 JIT(AOT) 컴파일러
- Java가 지원하는 GC 알고리즘
- 스레드 동기화

OT

다루지 않는 내용

- 안드로이드 (Dalvik)
- Java 일반적인 문법과 기능, 기본 개념, 작성 컨벤션
- Gradle, Maven 등과 같은 빌드 툴
- Spring, Quarkus 같은 프레임워크
- 현재 일반적으로 많이 사용되지 않는 라이브러리나 기능
 - AWT, Swing, JavaFX 같은 GUI 라이브러리
 - Java Applet (동적 웹 페이지 생성) 같은 앱
- 기타

OT

참고

- 여러 JDK(JVM) 벤더 중 Oracle JDK, OpenJDK 문서를 기준으로 설명
- JDK LTS 버전인 17 버전을 기준으로 설명
 - 하지만 공식 자료를 찾을 수 없는 경우 다른 버전을 인용
- 실습 환경
 - MacOS Ventura (13.4), (CPU - M1 Pro, Memory - 32GB)
 - IntelliJ IDEA Ultimate (2023.1.2)
- 질문은 언제든 편하게

OT

학습 목표

- 단순히 무지성으로 Java의 문법만 사용하지 말고 한 번 더 생각하기
 - 구현하지 못하고 지식만 떠드는 것은 그저 이쁜 쓰레기에 불과하다
 - 하지만 정확한 지식 없이 구현하는 것은 시한폭탄을 만드는 것과 같다
- 지식을 쌓으면서 동시에 숙련도 높이기
 - 이론과 구현을 별개로 나누지 말 것
- 해당 강의를 통해 프로그래밍 언어에 대한 이해도를 높이기 (Polyglot)
 - 물론 현실적으로 폴리글랏은 쉽지 않고 시간이 많이 듈다!
- 나아가 언어 뿐 아니라 어떠한 기술이든 깊게 살펴보는 습관 만들기
 - 다른 것을 살펴볼 때 아는 만큼 보게 된다!

[1] Java의 정의와 동작 방식

Java의 정의와 동작 방식

Java란?

- 1995년 썬 마이크로시스템즈에서 나온 프로그래밍 언어
- WORA 사상을 기반으로 설계된 범용 프로그래밍 언어
 - Write once, run anywhere
- High-Level Language
- Class based
- JVM을 통해 작동
- 객체 지향 언어 (Object-Oriented Language)
 - 하지만 Pure한 Object-Oriented Language가 아니다?



[https://en.wikipedia.org/wiki/Java_\(programming_language\)](https://en.wikipedia.org/wiki/Java_(programming_language))

Java의 정의와 동작 방식

Java란?

- 순수 객체 지향 언어의 특징?
 - 추상화
 - 다형성
 - 캡슐화
 - 상속
 - 모든 사전 정의 데이터 타입과 사용자 정의 타입은 객체여야 한다
 - 객체에 대한 모든 작업은 객체 스스로 정해야 한다

Java의 정의와 동작 방식

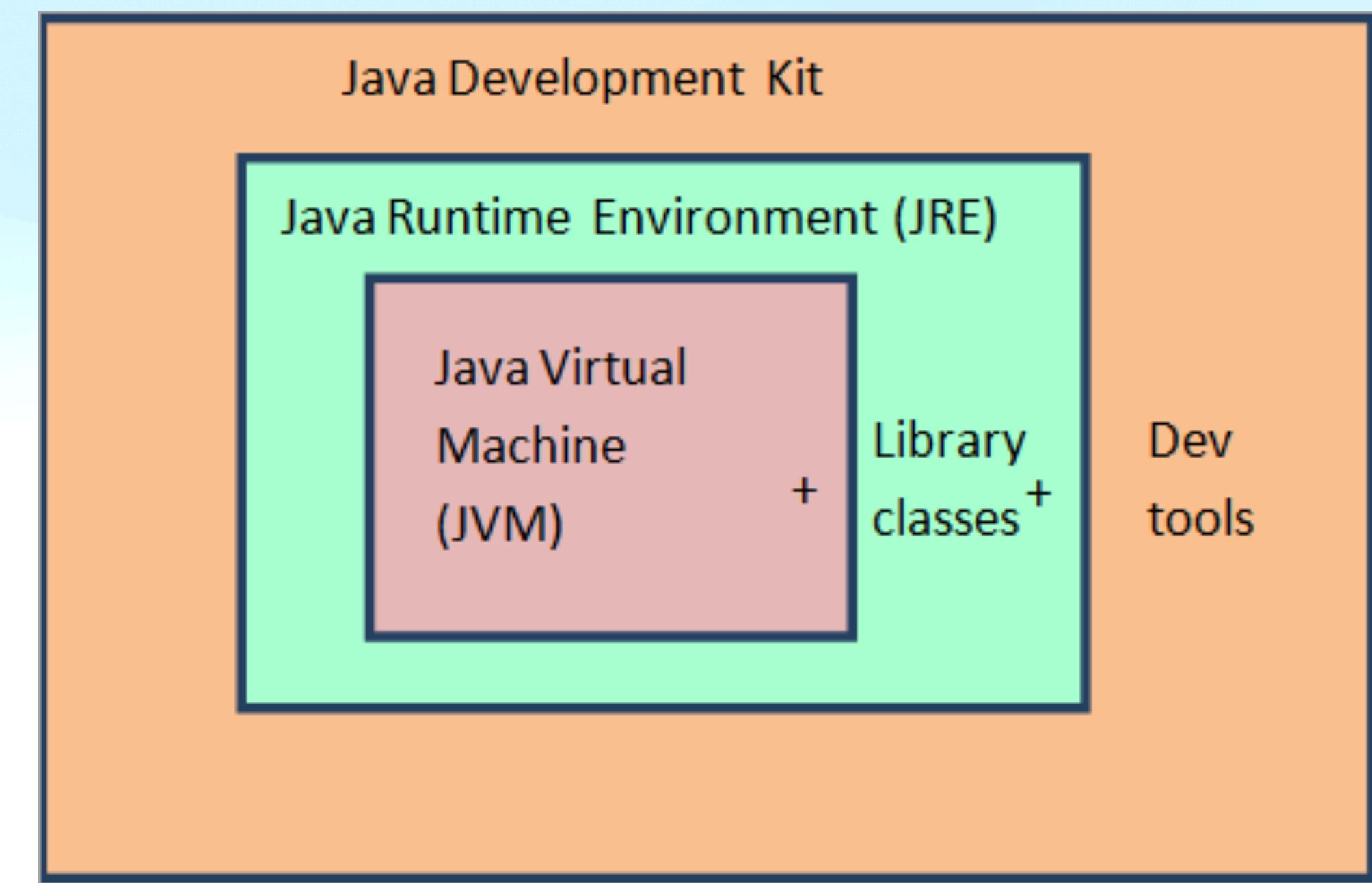
Java란?

- Java가 지키지 못한 순수 객체 지향 언어의 특징?
 - 원시 타입 (Primitive Type)
 - 일반 변수는 공유 가능
 - 정적 메서드 (Static Method)
 - 인스턴스의 생성 없이 호출 가능
 - 래퍼 클래스(Wrapper Class) 또한 Auto Boxing/Unboxing을 통해 원시 타입 변수 사용
 - 여기서 래퍼 클래스는 Integer, Double 등 원시 타입을 래핑한 클래스
 - Java는 OOP를 위해 설계된 객체 지향 언어
 - 하지만 일부 절차적인 요소가 있는 언어
 - 순수 객체 지향 언어는?
 - Ruby, Scala, Smalltalk 등

Java의 정의와 동작 방식

Java 아키텍처

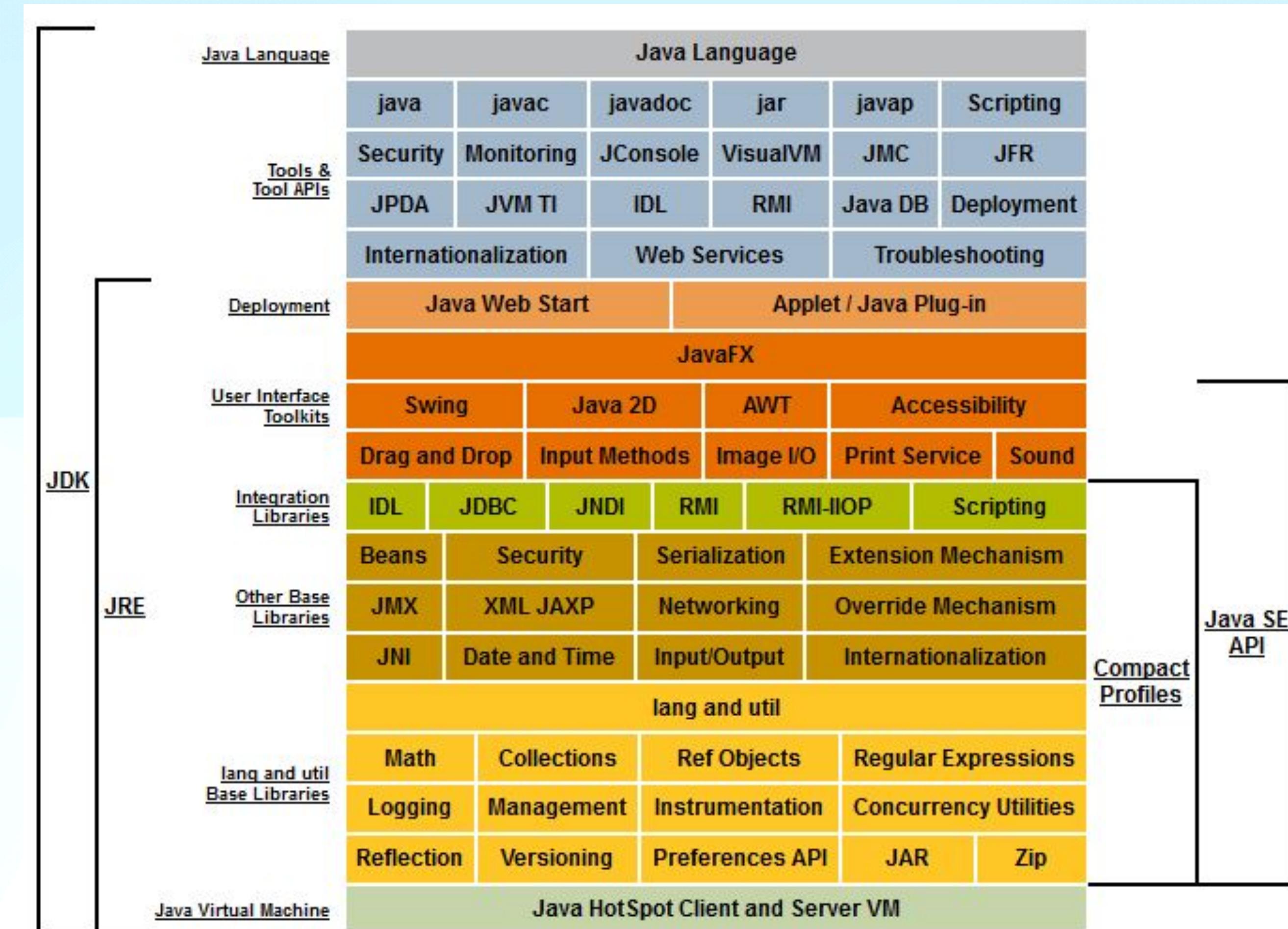
- Java 플랫폼
 - SE, EE, ME 등 JDK를 구현한 제품
 - 일반적으로 Java 개발 및 실행 환경을 의미
 - 흔히 JDK보다 더 넓은 의미로 사용
- JDK(Java Development Kit)
 - 자바 개발 키트 (JRE + Development Tools)
- JRE(Java Runtime Environment)
 - 자바 실행 환경 (JVM + Library)
- JVM(Java Virtual Machine)
 - 자바 가상 머신 (프로그램 작동)
 - Java 바이트코드를 기계어로 변환하고 실행 (+@)



<https://www.softwaretestinghelp.com/java-components-java-platform-jdk/>

Java의 정의와 동작 방식

JDK 구성 요소

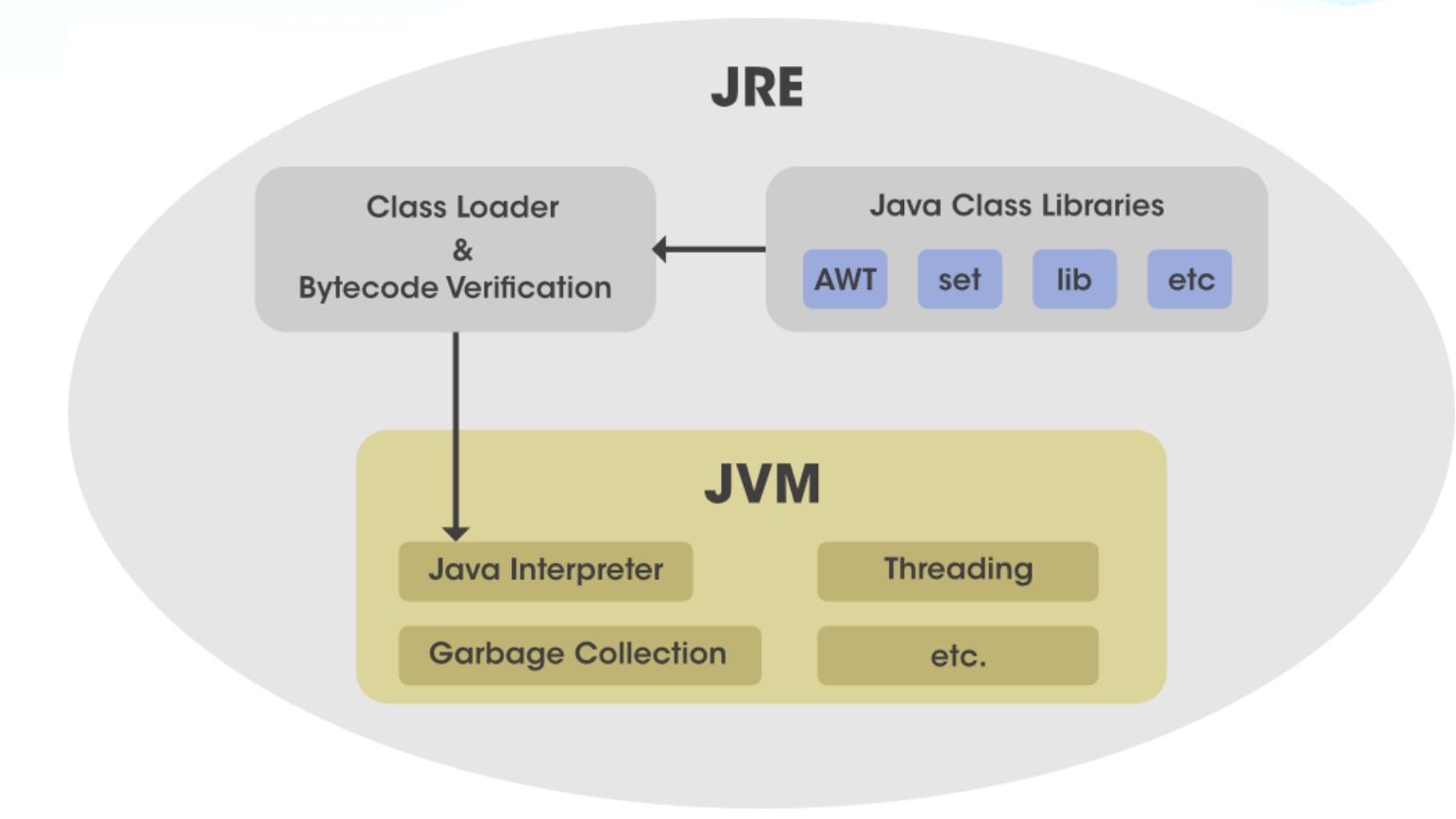


<https://www.oracle.com/java/technologies/platform-glance.html>

Java의 정의와 동작 방식

JDK(+JRE)

- Java 프로그램 실행 및 개발 환경(툴) 제공
- JDK 11 이후 JRE를 포함
 - OpenJDK 제공 및 일관된 환경 제공을 위해 단순화
 - 기존 JRE는 JVM, Class Loader, Java Class Libraries, Resource 등 포함
- JDK 9 버전에 도입된 JPMS로 내부 기능을 모듈화
 - JPMS - Java Platform Module System
 - 필요한 모듈만 모아 커스텀 JRE 생성해 메모리, 용량 등 절약
- 참고
 - 모듈은 코드, 데이터를 그룹화하여 재사용이 가능한 정적인 단위
 - 컴포넌트는 독립적으로 실행될 수 있는 소프트웨어 단위

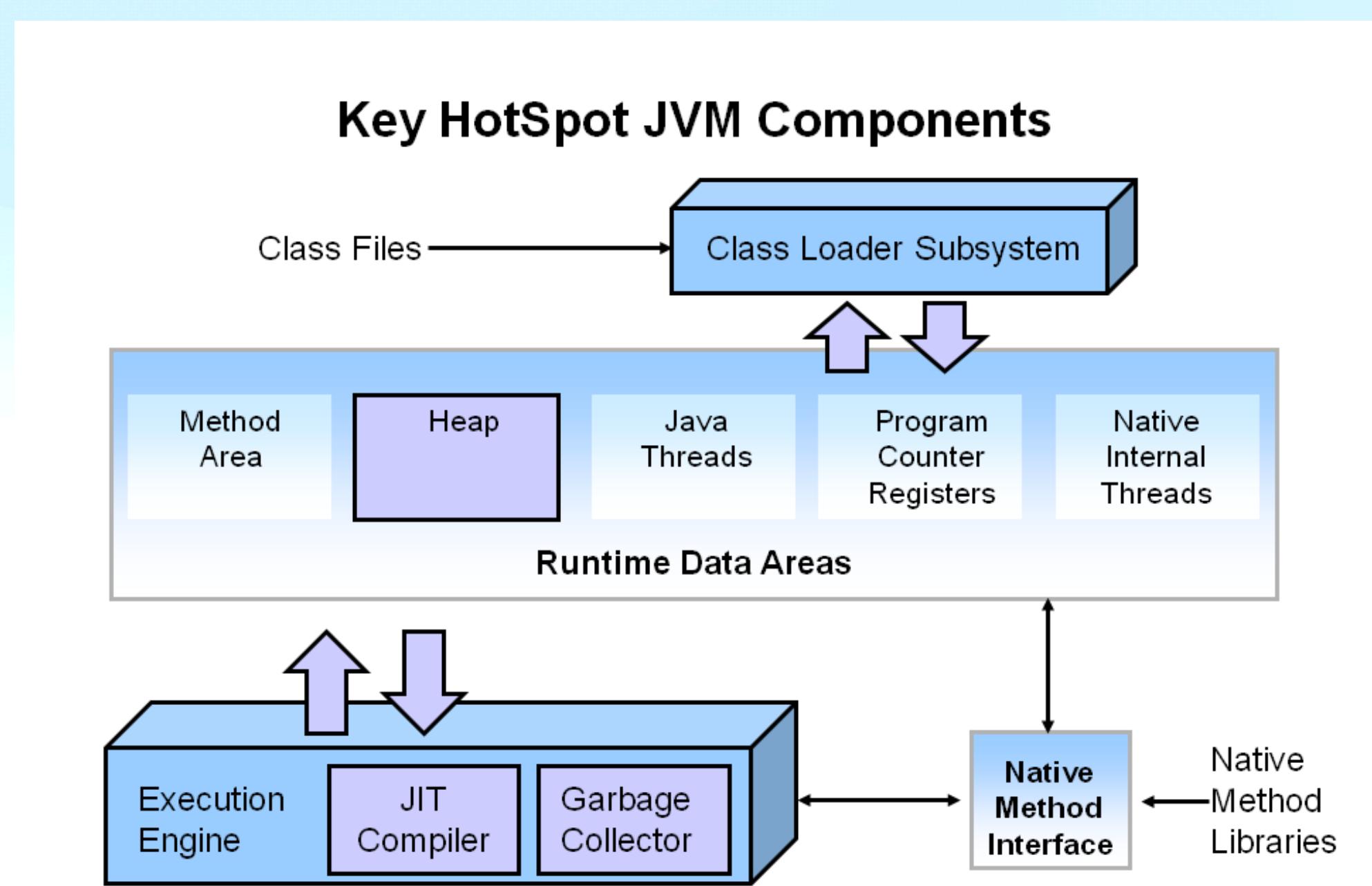


<https://www.geeksforgeeks.org/ire-full-form/>

Java의 정의와 동작 방식

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

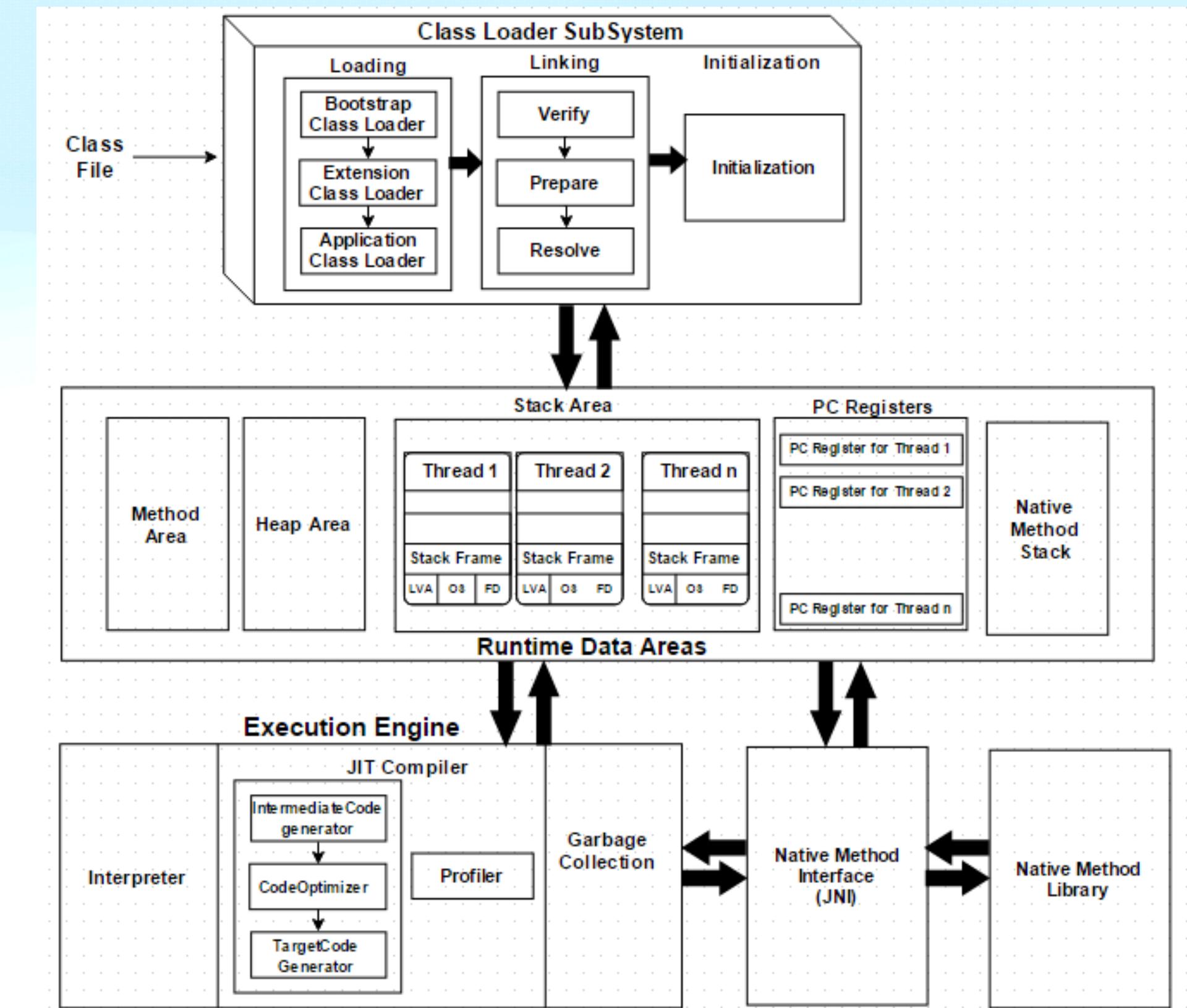


<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

Java의 정의와 동작 방식

JVM Architecture

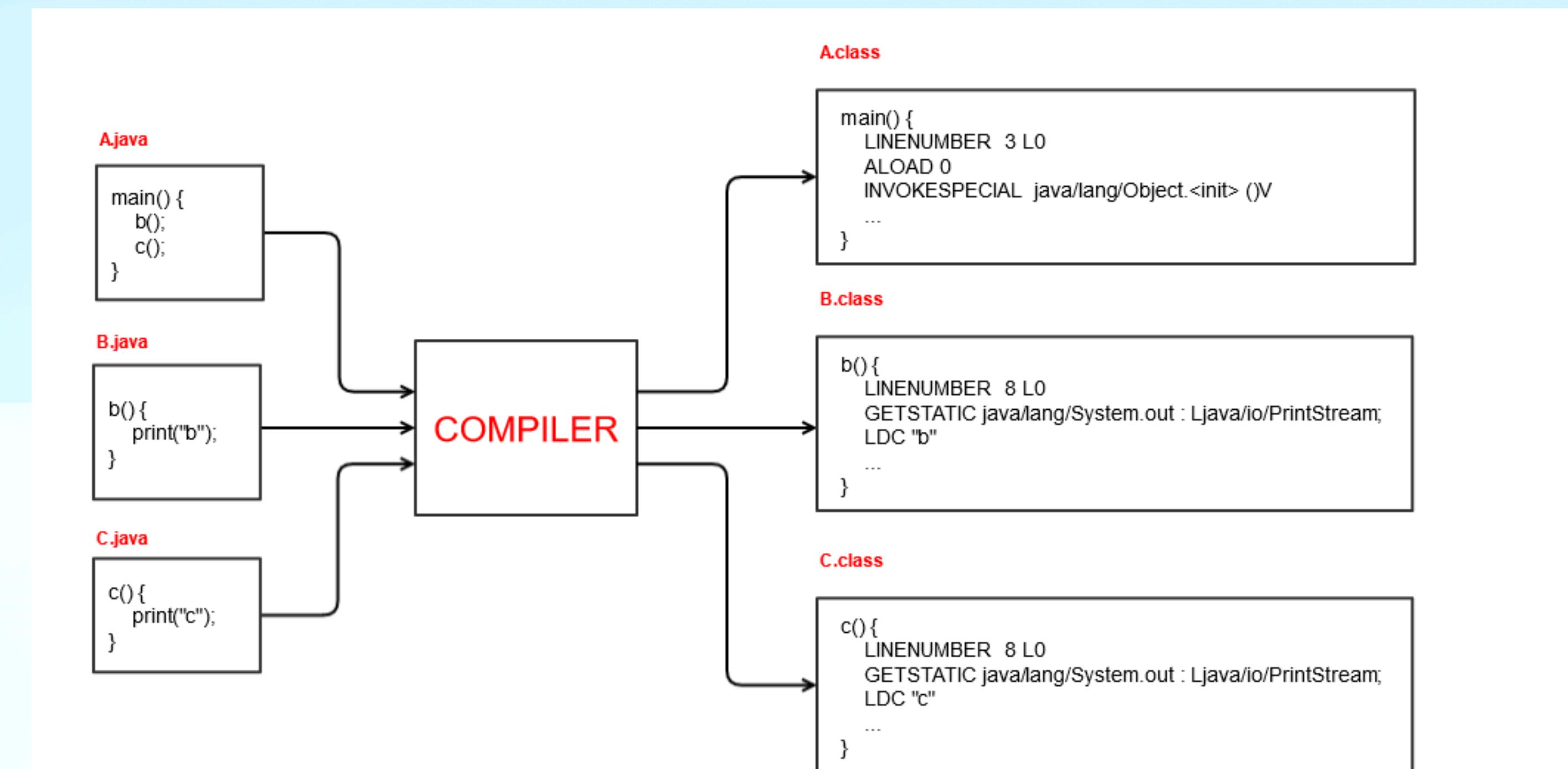
- Class Loaders
 - 바이트코드 로딩, 검증, 링킹 등 수행
- Runtime Data Areas
 - 앱 실행을 위해 사용되는 JVM 메모리 영역
- Execution Engine
 - 메모리 영역에 있는 데이터를 가져와 해당하는 작업 수행
- JNI (Java Native Interface)
 - JVM과 네이티브 라이브러리 간 이진 호환성을 위한 인터페이스
 - 네이티브 메서드(네이티브 언어 C/C++ 등으로 작성) 호출, 데이터 전달과 메모리 관리 등 수행
- Native Libraries
 - 네이티브 메서드의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



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

Java의 정의와 동작 방식

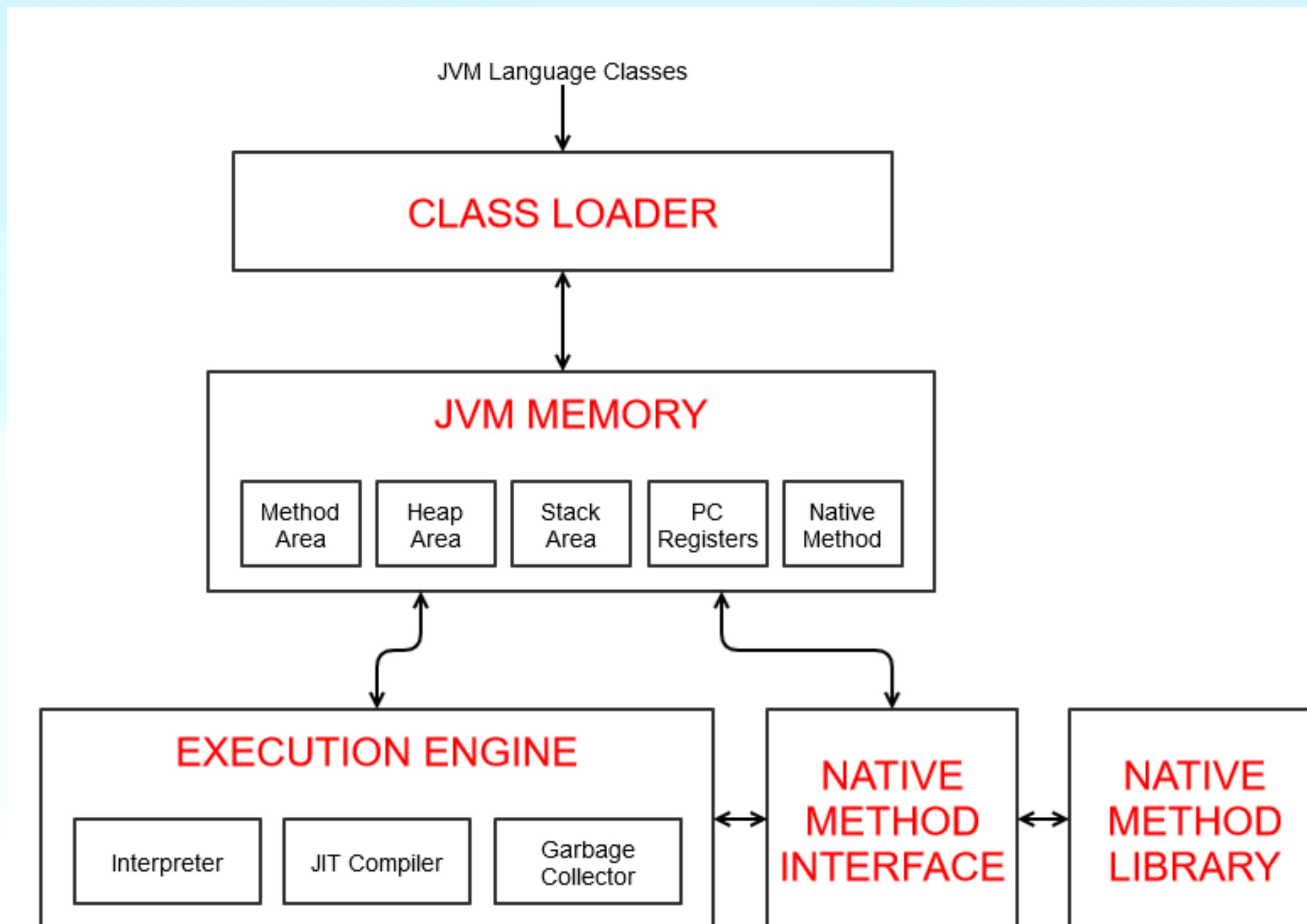
Java 동작 방식



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

Java의 정의와 동작 방식

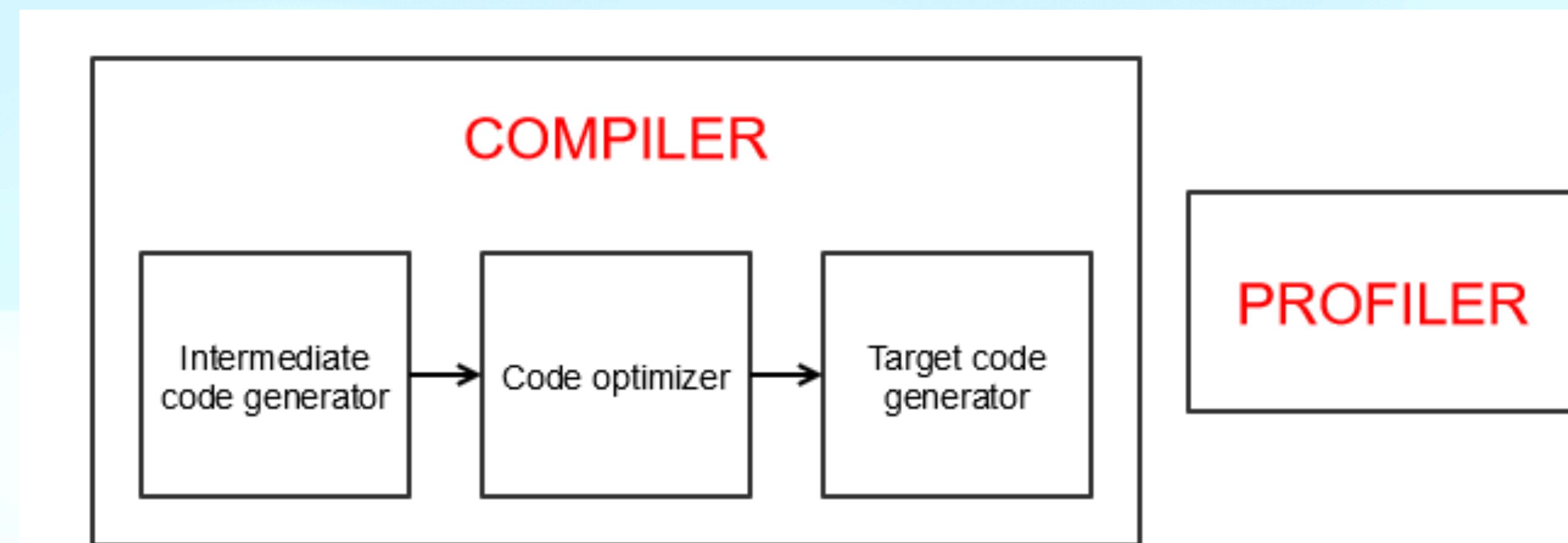
Java 동작 방식



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

Java의 정의와 동작 방식

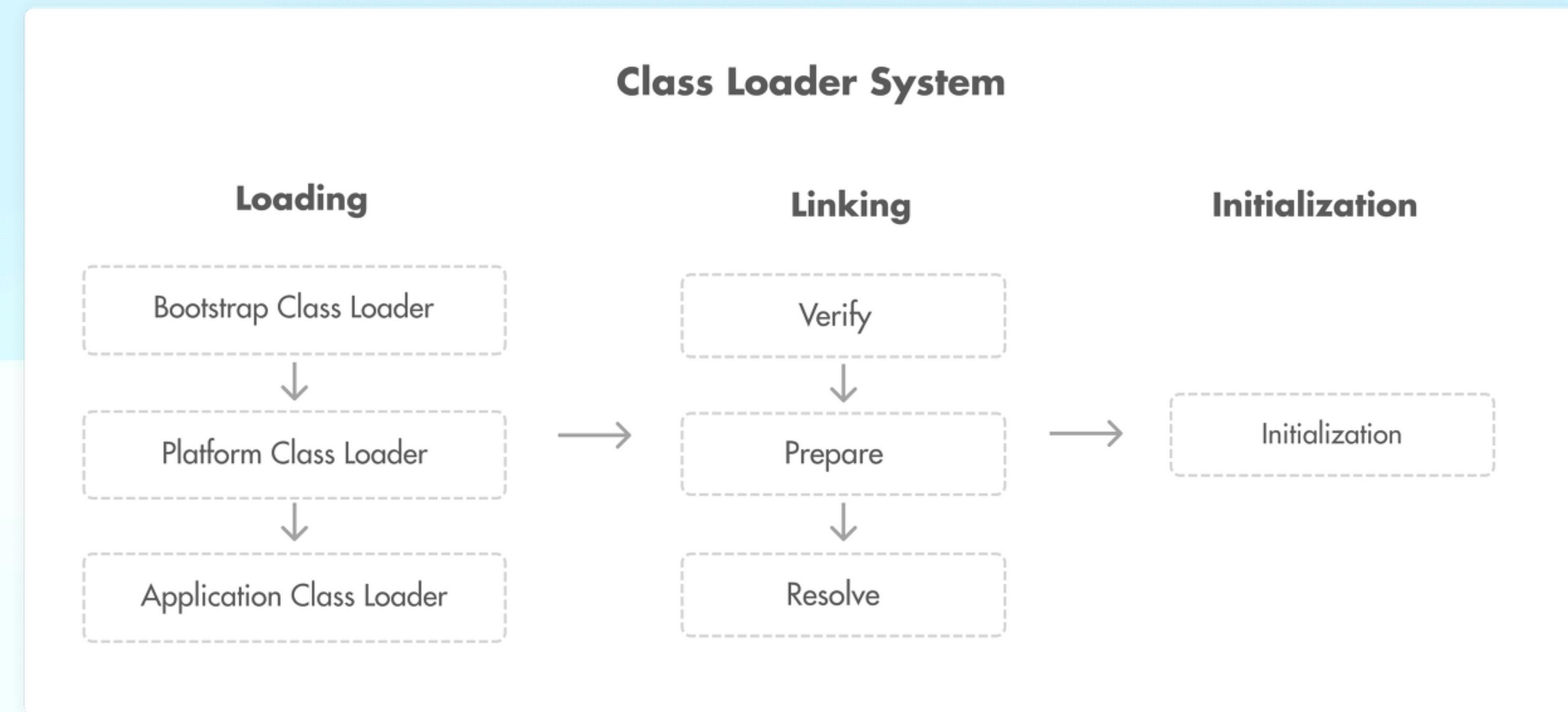
Java 동작 방식



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

Java의 정의와 동작 방식

Class Loaders



<https://ngsn.tistory.com/252>

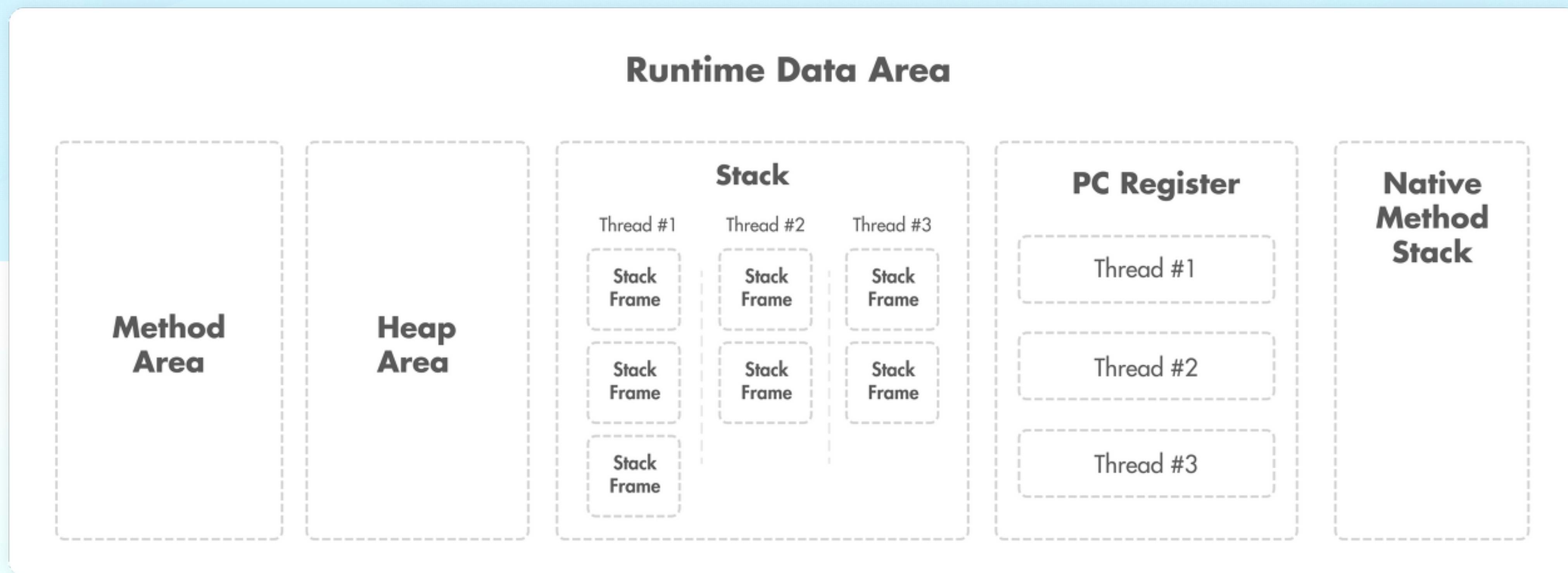
Java의 정의와 동작 방식

Class Loaders

- 클래스 로더는 런타임에 Java 클래스/인터페이스의 바이트코드를 동적으로 메모리에 로딩
 - 한 번에 모든 클래스가 메모리에 로드되지 않고 아닌 필요할 때마다 로드
- 로딩 작업은 크게 3가지로 분리됨
 - Loading : JVM이 필요한 클래스 파일 로드
 - Linking : 로드된 클래스의 verify, prepare, resolve 작업 수행
 - Initializing : 클래스/정적 변수 등 초기화

Java의 정의와 동작 방식

JVM Run-Time Data Areas



<https://gngsn.tistory.com/252>

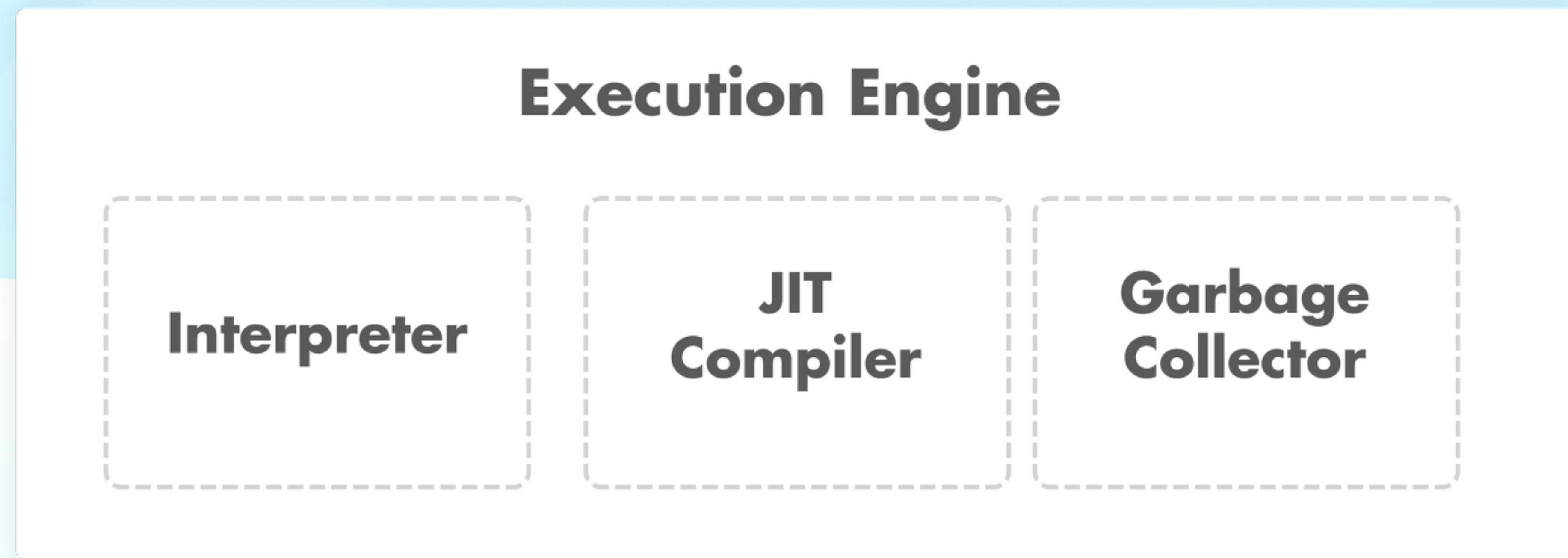
Java의 정의와 동작 방식

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

Java의 정의와 동작 방식

Execution Engine



<https://ngsn.tistory.com/252>

Java의 정의와 동작 방식

Execution Engine

- JVM 메모리 영역에 있는 바이트코드를 읽어 네이티브코드로 변환하고 실행
 - Interpreter
 - 메모리에 로드된 바이트코드를 한줄 씩 해석/실행
 - JIT(Just-In-Time) Compiler
 - 자주 호출되는 메서드(hot method)의 바이트코드를 네이티브 코드로 컴파일
 - JVM이 실행 메서드를 모니터, JIT 컴파일러의 프로파일러가 수집한 프로파일 정보를 기반으로 처리
 - 중간 코드 생성 > 코드 최적화 > 네이티브 코드 생성
 - GC (Garbage Collector)
 - 메모리에서 사용하지 않는 객체를 식별해 삭제하는 프로세스 (대표적으로 Heap 영역)
 - 데몬 스레드로 동작 (명시적으로 호출해도 즉시 실행되지 않음)
 - 필요한 경우 JNI를 통해 네이티브 메서드 라이브러리를 호출

Java의 정의와 동작 방식

JNI (Java Native Interface)

- 네이티브 라이브러리 사용을 위한 인터페이스이자 동시에 해당 역할을 수행 (일종의 프레임워크)
- JNI를 통해 JVM내 Java 코드는 네이티브 언어/라이브러리와 상호 운용될 수 있음
 - JNI는 Java VM에 의존적이지 않아 다른 부분에 영향을 주지 않고 JNI에 대해서 추가 가능

Java의 정의와 동작 방식

Native Method Libraries

- 네이티브 언어/어셈블리와 같은 언어로 작성된 네이티브 메서드를 포함한 라이브러리
- JVM에서 호출할 때 JNI를 통해 로딩

Java의 정의와 동작 방식

컴파일러와 인터프리터

- 컴파일러 (Compile 방식)
 - 프로그래밍 언어로 작성된 코드를 타겟 언어로 변환(번역)하는 프로그램
 - 주로 High-Level 언어를 Low-Level 언어(assembly, object code, machine code 등)로 변환
 - 또한 전처리와 어휘/구문/의미 분석, 코드 최적화(optimization), 기계어 생성 등 역할도 수행
- 인터프리터 (Interpret 방식)
 - 읽은 코드 및 해당 명령을 직접 분석/실행하는 프로그램
 - 인터프리터 전략
 1. 코드 구문을 분석, 동작을 직접 수행
 2. 코드를 object code (중간 코드)로 변환, 즉시 실행
 3. 컴파일러에 의해 생성된 바이트코드를 명시적으로 실행
 - Java는 2, 3 번의 혼합

Java의 정의와 동작 방식

Java 코드 실행 방식

- Java는 Interpret? Compile? 무슨 방식일까?
 - Java는 두 가지 방식을 혼합하여 사용하는 하이브리드 모델
 - javac로 소스 코드를 바이트코드(object code)로 변환
 - 변환된 바이트코드를 JVM 인터프리터가 분석, 실행
- 일반적으로 Java를 컴파일 언어로 분류하는 이유?
 - 실제로 javac를 통해 .java 파일을 .class 파일로 컴파일하기 때문에

Java의 정의와 동작 방식

Java 동작 방식 정리

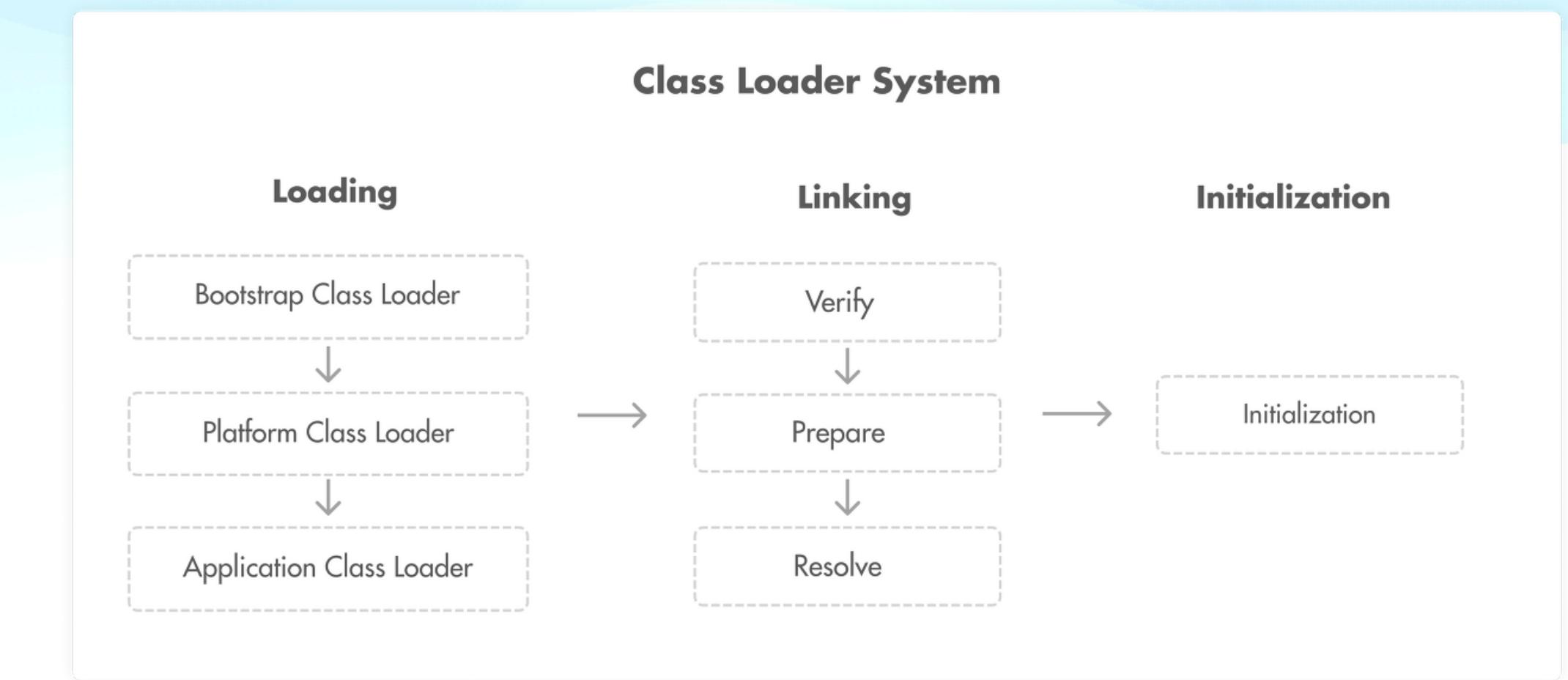
1. 작성한 소스 파일(.java)을 Java 컴파일러(javac)가 바이트코드(.class)로 변환
2. JVM이 실행되면 바이트코드 실행에 필요한 것들을 클래스 로더가 로딩
 - 메모리에 로딩된 클래스의 구조, 메서드 등 일련 정보들을 JVM 내부에 저장
3. 로딩된 클래스의 바이트코드를 JVM의 실행 엔진이 해석, 실행
 - 인터프리터를 통해 해석과 동시에 실행
 - 추후에 자주 사용되는 메서드 등을 JIT 컴파일러를 통해 네이티브 코드로 변환
4. 실행 준비가 모두 완료 되면 JVM은 메인 메서드(Entry point)를 호출
5. 호출된 메인 메서드를 실행할 메인 스레드가 생성되며 메인 스레드의 JVM stacks이 생성됨
6. 그 후 생성된 메인 스레드 JVM stacks에 메인 메서드 스택 프레임이 생성됨
7. 이후 앱이 실행되며 필요한 시점마다 필요한 처리를 수행하며 메모리 확보 및 데이터 저장
 - 클래스, 메서드 정보를 Method Area에 저장
 - Heap에 해당 인스턴스 할당
 - Frame 생성
 - JNI 스택 생성
8. JVM은 실행되는 Java 앱에 대해서 메모리, 스레드 등 관리

[2] 클래스 로더와 클래스 로딩

클래스 로더와 클래스 로딩

Class Loaders

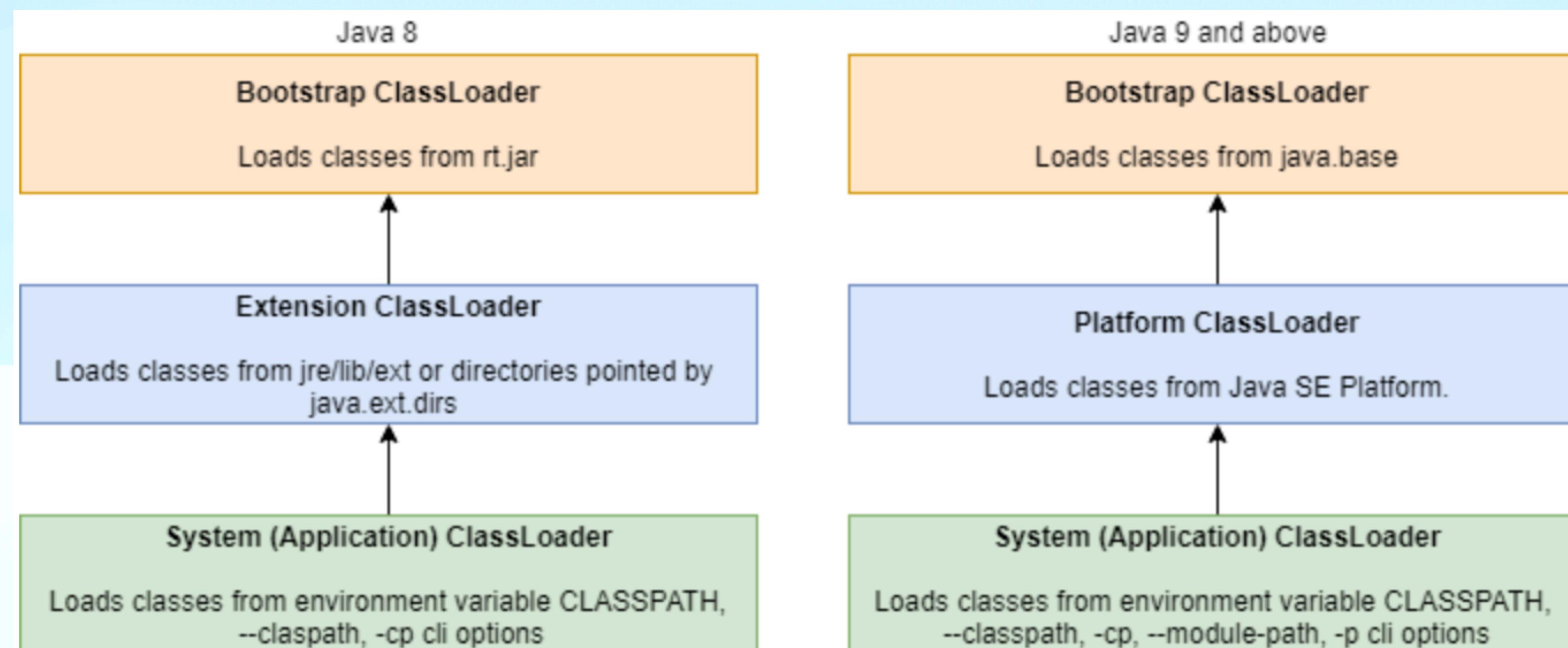
- 런타임에 Java Bytecode를 JVM 메모리로 동적 로딩
 - Loading (Creating)
 - Linking (Verification, Preparation, Resolution)
 - Initialization
- 필요할 때마다 필요한 클래스/리소스만 로딩
 - 한 번에 모두 로딩하지 않음
 - 클래스 로더는 계층 구조



<https://gngsn.tistory.com/252>

클래스 로더와 클래스 로딩

Class Loaders



<https://www.baeldung.com/java-find-all-classes-in-package>

클래스 로더와 클래스 로딩

Class Loaders - JDK 17 문서

Run-time Built-in Class Loaders

The Java run-time has the following built-in class loaders:

- **Bootstrap class loader.** It is the virtual machine's built-in class loader, typically represented as `null`, and does not have a parent.
- **Platform class loader.** The platform class loader is responsible for loading the *platform classes*. Platform classes include Java SE platform APIs, their implementation classes and JDK-specific run-time classes that are defined by the platform class loader or its ancestors. The platform class loader can be used as the parent of a `ClassLoader` instance.

To allow for upgrading/overriding of modules defined to the platform class loader, and where upgraded modules read modules defined to class loaders other than the platform class loader and its ancestors, then the platform class loader may have to delegate to other class loaders, the application class loader for example. In other words, classes in named modules defined to class loaders other than the platform class loader and its ancestors may be visible to the platform class loader.

- **System class loader.** It is also known as *application class loader* and is distinct from the platform class loader. The system class loader is typically used to define classes on the application class path, module path, and JDK-specific tools. The platform class loader is the parent or an ancestor of the system class loader, so the system class loader can load platform classes by delegating to its parent.

Normally, the Java virtual machine loads classes from the local file system in a platform-dependent manner. However, some classes may not originate from a file; they may originate from other sources, such as the network, or they could be constructed by an application. The method `defineClass` converts an array of bytes into an instance of class `Class`. Instances of this newly defined class can be created using `Class.newInstance`.

The methods and constructors of objects created by a class loader may reference other classes. To determine the class(es) referred to, the Java virtual machine invokes the `loadClass` method of the class loader that originally created the class.

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ClassLoader.html>

클래스 로더와 클래스 로딩

Class Loaders

- 부트스트랩 클래스 로더 (Bootstrap Class Loader)
 - 최상위 클래스 로더이자 네이티브 코드로 작성된 클래스 로더 (JVM이 로딩)로 base 모듈 로딩
 - 따라서 JVM에 따라 부트스트랩 클래스 로더의 구현이 다를 수 있음
 - 다른 Java 클래스의 로딩을 담당하는 `java.lang.ClassLoader`를 로드하는 클래스 로더
 - JDK 9 버전부터 `jre/lib/rt.jar`가 없어짐에 따라 내부 클래스들은 lib 디렉터리에 저장
- 플랫폼 클래스 로더 (Platform Class Loader)
 - Java SE platform의 모든 API/클래스 로딩
 - 플랫폼 클래스 로더 또는 상위 클래스로 정의된 JDK 런타임 클래스
 - JDK 9 버전부터 `jre/lib/ext`, `java.ext.dirs` 미지원 (일부 메커니즘 변경)
 - 그 전 버전까지 확장 메커니즘이 존재
 - 클래스패스에 별다른 네이밍 없이도 확장 클래스를 로딩, 확장 디렉터리에서 코드를 로딩
- 시스템 클래스 로더 (System Class Loader)
 - Java 앱 레벨의 클래스 로딩
 - 클래스패스, 모듈패스에 있는 클래스 로딩 (-classpath, -cp 등 환경 변수 옵션 포함)

클래스 로더와 클래스 로딩

Class Loaders 원칙 (Principle)

- 위임 (Delegation Model)
 - 클래스/리소스 등을 자신이 찾기 전에 먼저 상위 클래스 로더에게 위임
 - 그 전에 이미 로딩되어 있다면 위임하지 않고 반환
 - 상위 클래스 로더가 해당 클래스/리소스를 찾지 못하면 요청한 클래스 로더가 찾아서 로딩 (반복)
- 유일성 (Unique Classes)
 - 상위 클래스 로더가 로딩한 클래스를 하위 클래스 로더가 다시 로딩하는 것을 방지
 - Delegation 원칙으로 인해 클래스의 유일성을 확보하기 쉬움
- 가시성 (Visibility)
 - 하위 클래스 로더는 상위 클래스 로더가 로딩한 클래스를 볼 수 있음
 - 하지만 반대로 상위 클래스 로더는 하위 클래스 로더가 로딩한 클래스를 볼 수 없음

클래스 로더와 클래스 로딩

* 좋은 클래스 로더의 속성

- 클래스명이 같다면 클래스 로더는 항상 같은 클래스 객체를 반환
- 하위 클래스 로더가 상위 클래스로더에게 특정 클래스 로딩을 위임한 경우 두 클래스 로더는 해당 클래스에 대해 동일한 객체를 반환해야 함 (관련된 슈퍼클래스, 필드 타입, 생성자 파라미터, 메서드 반환 타입 등 포함)
- 사용자 정의 클래스 로더가 클래스 바이너리를 미리 가져올 때(Prefetching) 로딩 에러가 발생할 가능성이 있더라도 미리 가져올 때가 아닌 '적시'에 발생해야 하며 이런 경우 실제 사용 여부 판단이 선행되어야 함
- Prefetching 참고
 - 다음과 같은 경우 Prefetching은 JVM 구현 방식에 따라 가능
 - 사용 빈도가 높다고 예상되거나 복잡한 종속성이 있다고 판단되는 경우
 - 해당 클래스와 관련된 클래스 그룹을 같이 로딩하는 경우 (그룹 로딩)

클래스 로더와 클래스 로딩

Class Loaders 예시

```
public static void main(String[] args) { ↗Jaenyeong Complexity is 4 Everything is cool!
    System.out.println("ClassLoader of ArrayList: " + ArrayList.class.getClassLoader());
    System.out.println("ClassLoader of DriverManager: " + DriverManager.class.getClassLoader());
    System.out.println("ClassLoader of this class: " + Example03.class.getClassLoader());
}
```

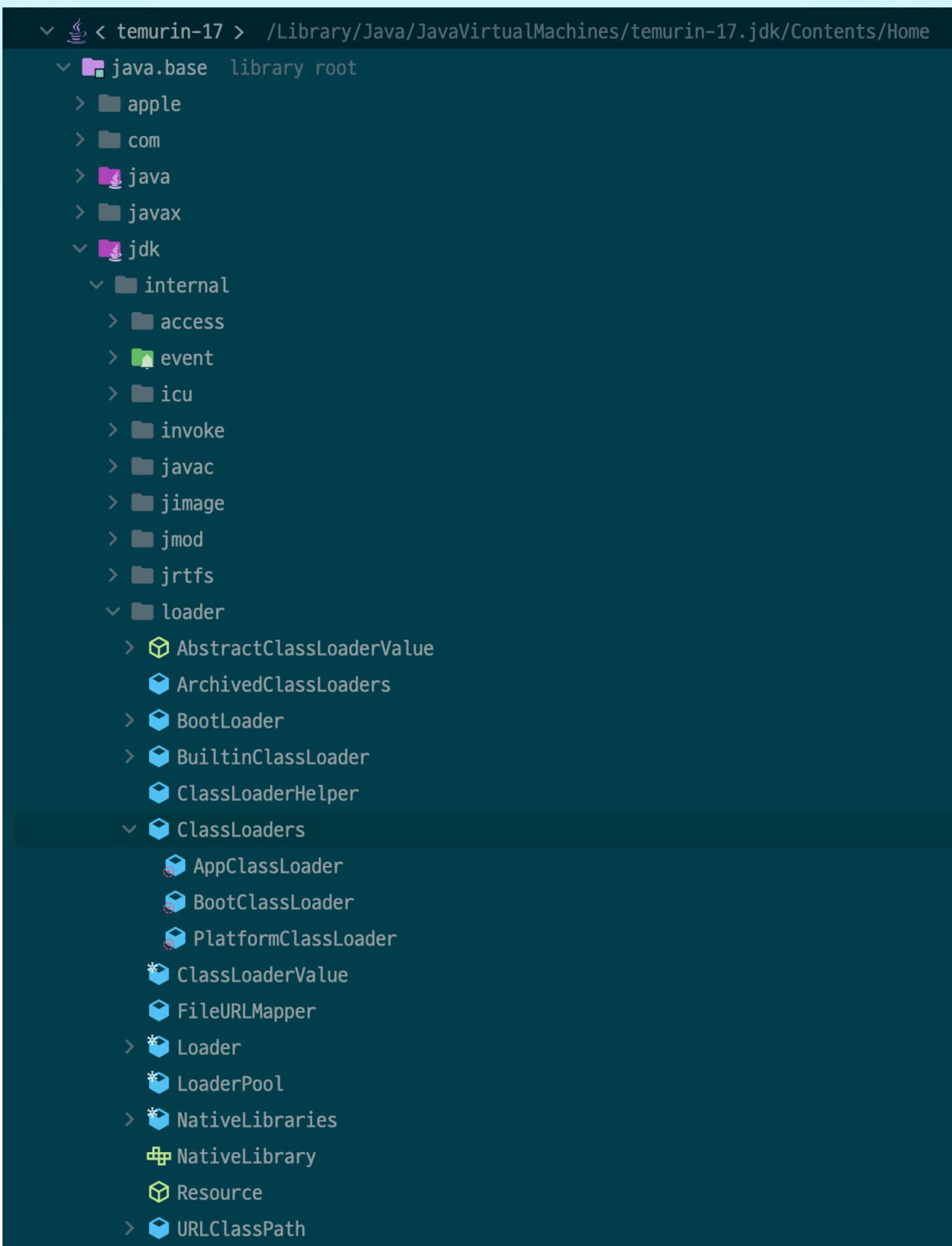


```
ClassLoader of ArrayList: null
ClassLoader of DriverManager: jdk.internal.loader.ClassLoaders$PlatformClassLoader@1996cd68
ClassLoader of this class: jdk.internal.loader.ClassLoaders$AppClassLoader@42110406
```

<https://www.baeldung.com/java-classloaders>

클래스 로더와 클래스 로딩

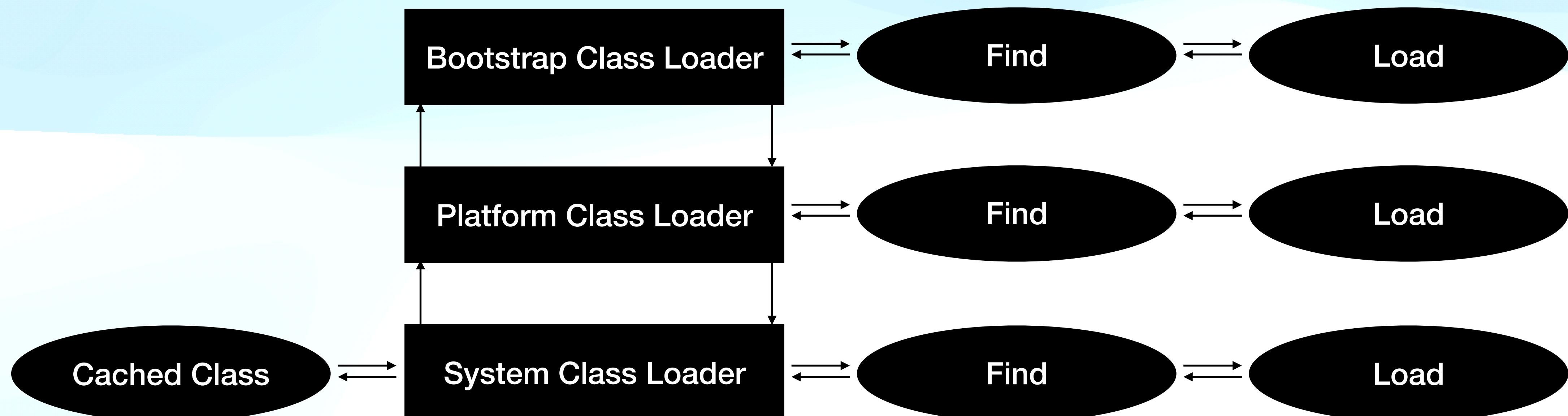
Class Loaders temurin JDK 17



클래스 로더와 클래스 로딩

Loading

- 특정 이름(FQCN)을 가진 클래스(인터페이스 등)의 바이트코드를 찾아 클래스를 만드는 프로세스



클래스 로더와 클래스 로딩

Loading - 동작 방식

1. 최하위 클래스 로더부터 클래스를 찾음

- `java.lang.ClassLoader`의 생성자, 하위 클래스를 사용해서 새 클래스 로더의 상위 지정 가능
 - 명시적으로 상위 클래스 로더를 지정하지 않으면 JVM의 System 클래스 로더로 지정됨
- FQCN(Fully Qualified Class Name)을 기반으로 로딩된 클래스인지 찾음
 - 상위 클래스 로더에서 로딩하는 클래스와 이름이 같아도 FQCN이 다르기 때문에 다른 클래스로 인식
 - 하지만 만약 상위 클래스와 FQCN까지 같다면 충돌 발생

2. `java.lang.ClassLoader`의 loadClass 메서드를 통해 클래스 로딩 수행 (배열 클래스는 JVM이 생성)

- 이미 로딩된 클래스가 있다면 이를 반환
- 해당 클래스가 로딩되지 않았다면 이를 바로 로딩하지 않고 상위 클래스 로더에게 위임 (최상위 클래스 로더까지 반복)
- 상위 클래스 로더가 로드하지 못했다면 findClass 메서드를 호출해 클래스 로딩
 - 상위 클래스 로더가 null 또는 ClassNotFoundException 예외를 발생한 경우
- 클래스가 로딩될 때 Run-Time constant Pool도 같이 생성
- 클래스 로더에 의해 생성된 객체가 참조하는 클래스를 로딩할 때도 같은 메커니즘

3. 상위 클래스 또한 해당 클래스를 찾지 못한다면 하위 클래스가 찾아서 로딩

- 최하위 클래스 로더까지 로드에 실패하면 NoClassDefFoundError 또는 ClassNotFoundException 발생

클래스 로더와 클래스 로딩

Loading - FQCN 충돌

The screenshot shows a Java project named "Teach_Wanted-PreOnBoarding-Challenge" in an IDE. The project structure is as follows:

- Project**: Teach_Wanted-PreOnBoarding-Challenge
 - .github
 - .gradle
 - .idea
 - Chapter1
 - build
 - src
 - main
 - java
 - com
 - java
 - lang
 - resources
 - test
 - java
 - resources
 - Chapter2
 - Chapter3
 - Chapter4
 - gradle
 - howtosubmit
 - .gitignore

The code editor displays two files:

- String.java**: package java.lang;
public class String {
}
- FqcnTest.java**: package java.lang;

The status bar at the bottom indicates the file is 1:1, LF, UTF-8, 4 spaces, and 619 of 2048M.

A compilation error is shown in the bottom right corner:

```
'Teach_Wanted-PreOnBoarding-Challenge/Chapter1/src/main/java/java/lang/String.java:1: error:  
  package exists in another module: java.base  
package java.lang;  
^'
```

클래스 로더와 클래스 로딩

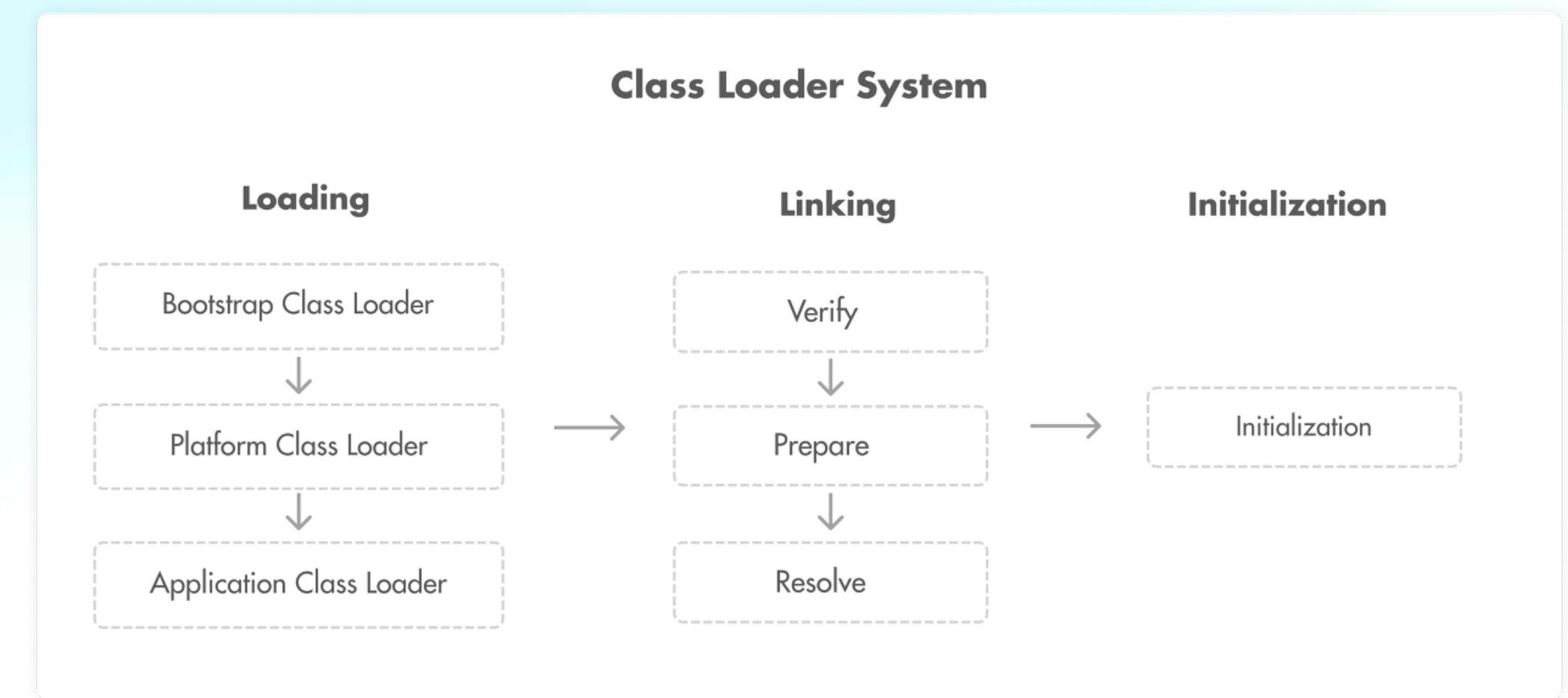
* **ClassNotFoundException & NoClassDefFoundError**

- **ClassNotFoundException**
 - FQCN을 통해 클래스 로딩 시 클래스패스에서 해당 클래스를 찾을 수 없는 경우 발생
 - 일반적으로 클래스명 등을 통해 리플렉션할 때 발생
 - `Class.forName`, `ClassLoader.loadClass`, `ClassLoader.findSystemClass` 메서드 등
- **NoClassDefFoundError**
 - new 키워드 또는 팩터리 메서드 등을 통한 객체를 로드할 때 발생
 - 일반적으로 static 블록 실행이나 static 변수 초기화 시 예외가 발생한 상황에서 에러 발생
- 위 문제를 막기 위해 신경써야 할 주의점
 - 해당 파일의 존재 유무
 - 올바른 클래스패스 설정 여부
 - 해당 클래스를 로딩하는 클래스 로더
 - 기타

클래스 로더와 클래스 로딩

Linking

- 로딩된 클래스를 클래스(인터페이스 등)을 실행하기 위해서 결합(연결)하는 프로세스
 - 검증 (Verification), 정적 필드 준비(Preparation), 심볼릭 레퍼런스 처리 (Resolution)



<https://gngsn.tistory.com/252>

클래스 로더와 클래스 로딩

Linking

- 해당 바이트코드의 결합해 실행할 수 있도록 하는 프로세스
- 로딩된 바이트코드의 유효성을 검증하고 (Verification)
선언된 스태틱 필드를 초기화, 필요한 메모리 할당을 하며 (Preparation)
심볼릭 레퍼런스를 실제 참조, 프로세스 등으로 변환(Resolution)하는 작업
- 심볼릭 레퍼런스의 검증은 링킹보다 나중에 처리될 수 있음
- 링킹은 새로운 자료구조의 할당을 포함하기 때문에 OutOfMemoryError가 발생할 수 있음

클래스 로더와 클래스 로딩

Linking 활성화 조건

- 클래스/인터페이스는 '링킹' 되기 전에 완전히 '로딩' 되어야 함
- 클래스/인터페이스는 '초기화' 되기 전에 완전히 '검증' 되어야 함
- 링킹 중 감지된 오류는 관련 클래스/인터페이스의 링킹이 직간접적으로 필요한 지점에서 발생해야 함
- 심볼릭 레퍼런스는 실제 참조될 때까지 확인되지 않음 (일종의 Lazy Loading)
 - 동적으로 계산되는 상수의 심볼릭 레퍼런스도 부트스트랩 메서드가 호출할 때까지 확인되지 않음

클래스 로더와 클래스 로딩

* 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 char[] 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

클래스 로더와 클래스 로딩

Linking - Verification

- Binary Representation(Java Bytecode)의 유효성을 검사하는 프로세스
 - 이때 관련 클래스(인터페이스)가 로딩될 수 있지만 이 클래스가 꼭 검증, 준비될 필요는 없음
 - 이는 JVM 구현 방식에 따라 다름
 - 클래스/인터페이스 바이트코드가 유효하지 않은 경우 이 시점에서 VerifyError 에러가 발생해야 함
 - 검증 중에 LinkageError 에러가 발생하면 이후 검증 시에는 항상 동일한 에러로 실패 해야 함

클래스 로더와 클래스 로딩

Linking - Preparation

- 클래스/인터페이스의 스태틱 필드를 생성, 필요한 메모리를 할당하며 기본값으로 초기화하는 프로세스
 - 예를 들어 스태틱 int 타입은 0, 객체 참조는 null로 초기화
 - 스태틱 필드의 값 초기화(바인딩)는 다음 작업인 초기화(Initialization) 과정의 일부
 - 인스턴스 필드는 객체의 인스턴스화 과정에서 초기화, 메모리 할당
 - 이때 런타임 상수 풀(Run-Time Constant Pool)도 메서드 영역(Method Area)에 할당됨

클래스 로더와 클래스 로딩

* 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 스탠틱 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

클래스 로더와 클래스 로딩

Linking - Resolution

- 심볼릭 레퍼런스가 구체적인 값을 가리키도록 동적으로 결정하는 프로세스
 - JVM의 많은 명령들이 런타임 상수 풀의 심볼릭 레퍼런스에 의존
 - 처음부터 많은 심볼릭 레퍼런스가 확인/검증되지는 않음
- 심볼릭 레퍼런스에 대한 동적 계산(실제 주소값으로 변경)은 아래 규칙을 준수하며 수행
 - 해당 작업을 수행하는 동안 오류가 발생하지 않음
 - 후속 시도는 초기 시도와 동일한 결과를 만듦
- 이 과정에서 아래와 같은 에러가 발생한다면 심볼릭 레퍼런스를 직간접적으로 사용하는 지점에서 발생해야 함
 - IncompatibleClassChangeError (클래스 정의가 컴파일과 런타임에 따라 다를 때 발생)
 - 부트스트랩 메서드로 인한 에러
 - 클래스 로더 제약 조건 위반으로 발생한 LinkageError

클래스 로더와 클래스 로딩

Initialization

- 클래스/인터페이스의 초기화 메서드 등을 실행하며 초기화
스페셜 메서드(Special Method) 중 `Class Initialization Methods`와 정적 필드에 대한 초기화
 - 이전에 Verification, Preparation, Resolution 작업이 모두 완료되어야 수행 가능
- 클래스/인터페이스의 초기화 조건 (해당 작업 직전에 초기화)
 - 인스턴스 생성
 - 스탠다드 메서드가 호출되거나 참조
 - 상수가 아닌 스탠다드 필드 사용(호출) 시
 - (컴파일 타임)상수는 Java 컴파일러 최적화에 의해 인라인 될 가능성 존재
인라인 되면 해당 스탠다드 필드를 참조하는 형태가 아니기 때문에 클래스 로딩이 이뤄지지 않을 수 있음
 - 리플렉션을 통한 메서드 호출
 - 클래스일 때 서브 클래스의 초기화
 - 추상메서드와 스탠다드메서드를 선언하지 않은 인터페이스일 때 서브클래스의 초기화
 - JVM 로딩 시 초기화되는 클래스/인터페이스로 지정

클래스 로더와 클래스 로딩

Initialization - Synchronization

- 멀티스레드 환경이기 때문에 초기화 시 동기화에 신경써야 함
 - 서로 다른 스레드에서 같은 클래스/인터페이스를 동시에 초기화를 시도하는 상황
 - 초기화 작업으로 재귀적인 초기화가 발생하는 상황
- 각 클래스/인터페이스는 초기화 잠금을 가지고 있고
세부 사항은 JVM 구현여부에 따라 다르나 보통 고유락(intrinsic lock) 또는 모니터(monitor)를 의미함
- 초기화 전제 조건
 - 검증/준비 등이 완료된 초기화 전 상태인 클래스
 - 다른 스레드에 의해 초기화 중인 클래스
 - 완전히 초기화가 끝난 클래스
 - 초기화가 실패로 끝난 클래스

클래스 로더와 클래스 로딩

Initialization - Procedure

1. 클래스/인터페이스의 초기화 잠금 획득 시도 (대기)
2. 다른 스레드가 해당 클래스/인터페이스를 초기화하고 있다면 초기화 잠금을 해제하고 초기화가 완료될 때까지 해당 스레드 차단을 반복 (스레드 인터럽트 상태 무관)
3. 현재 스레드에서 초기화가 진행되고 있다면 초기화에 대한 재귀 요청이어야 함
이 경우 초기화 잠금을 해제하고 정상적으로 완료해야 함
4. 이미 초기화가 완료된 상태라면 그대로 초기화 잠금 해제 (추가 작업 필요 없음)
5. 클래스/인터페이스가 잘못되었다면 초기화가 불가능하며 `NoClassDefFoundError`가 발생하고 잘못되지 않았다면 해당 스레드에서 초기화 진행중이라고 기록 후 초기화 잠금 해제
 - 문제가 없다면 선언된 필드 순서대로 상수 초기화 특성대로 `static final` 필드 초기화
6. 초기화 대상이 클래스라면 수퍼클래스와 수퍼인터페이스(non-abstract, non-static 중 최소 하나가 선언된) 초기화
수퍼인터페이스는 역순으로 초기화 (반복)
 - 이때 예외가 발생하면 예외가 발생한 클래스/인터페이스에 '에러' 레이블을 태깅
대기 중인 모든 스레드에게 전파하고 초기화 잠금 동기화를 해제, 최초에 발생했던 예외와 동일한 예외를 발생
7. 정의된 클래스 로더에게 쿼리하여 해당 클래스/인터페이스의 어설션(Assertions) 활성화 여부 확인
8. 위 작업들을 마치고 클래스/인터페이스의 초기화 메서드 실행
 - 정상적으로 완료되면 초기화 잠금을 동기화하여 '완료' 레이블 태깅, 대기 중인 모든 스레드에게 전파
 - 예외가 발생했다면 `ExceptionInInitializerError` 인스턴스를 생성해 전달
만약 `OutOfMemoryError`로 인해 `ExceptionInInitializerError` 인스턴스를 생성할 수 없다면 이를 다음 단계에 `OutOfMemoryError` 객체로 대신 사용
초기화 잠금을 획득하고 해당 클래스/인터페이스에 대해 '에러' 레이블을 태깅 후 대기 중인 모든 스레드에게 전파한 후에 초기화 잠금 해제 후 초기화 작업 종료
 - JVM 구현에 따라 초기화 잠금 획득을 생략해서 초기화 절차를 최적화할 수 있음

클래스 로더와 클래스 로딩

* 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

클래스 로더와 클래스 로딩

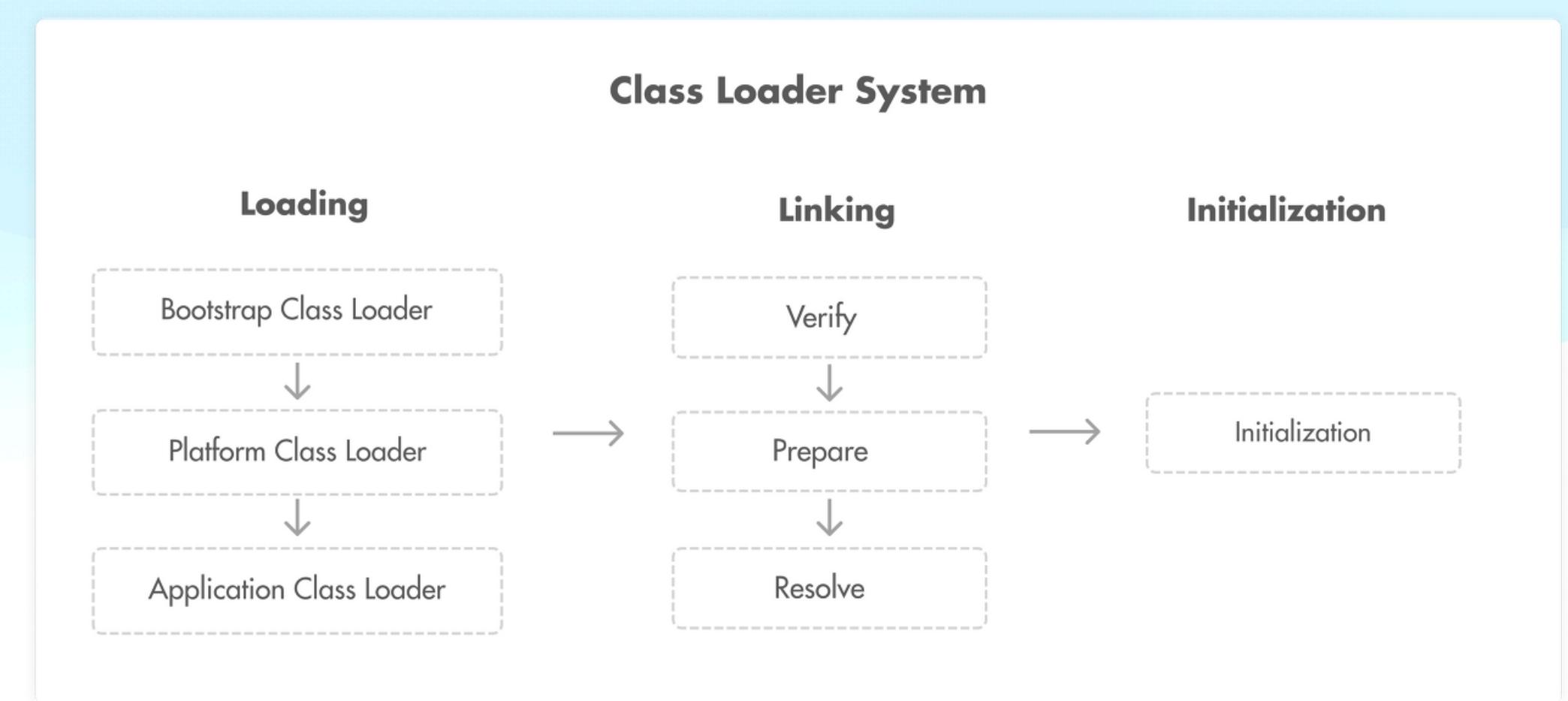
* Monitor

- 동기화를 관리하는 주체이며 일종의 스레드 세이프를 위한 클래스(객체, 모듈)
여러 스레드가 변수/메서드에 동시에 접근하는 것을 안정하게 수행할 수 있도록 지원
- 일반적으로 아래와 같은 동작을 수행함
 - 상호 배제 (Mutual Exclusion)
 - 특정 조건이 `false`가 될 때 까지 차단
 - 상태 변경 시 다른 스레드들에게 알림
- Java의 모니터는 객체(인스턴스) 별로 1개씩 소유하며 힙에 위치함
- 보통 뮤텍스(Mutex) 객체와 조건 변수로 구성

클래스 로더와 클래스 로딩

Class Loaders - 클래스 로딩 정리

1. 각 클래스 로더는 상위 로더로 위임하며 클래스 로딩
 - 로딩(캐시)된 클래스가 있다면 종료
2. 로딩한 클래스에 문제가 있는지 유효성 검사
3. 정적 필드 메모리 영역 할당, 기본값으로 초기화
4. 심볼릭 레퍼런스 등을 실제 값으로 변경
5. 초기화 메서드를 실행하여 초기화



<https://gngsn.tistory.com/252>

[3] Java 바이트코드와 코드 캐시

Java 바이트코드와 코드 캐시

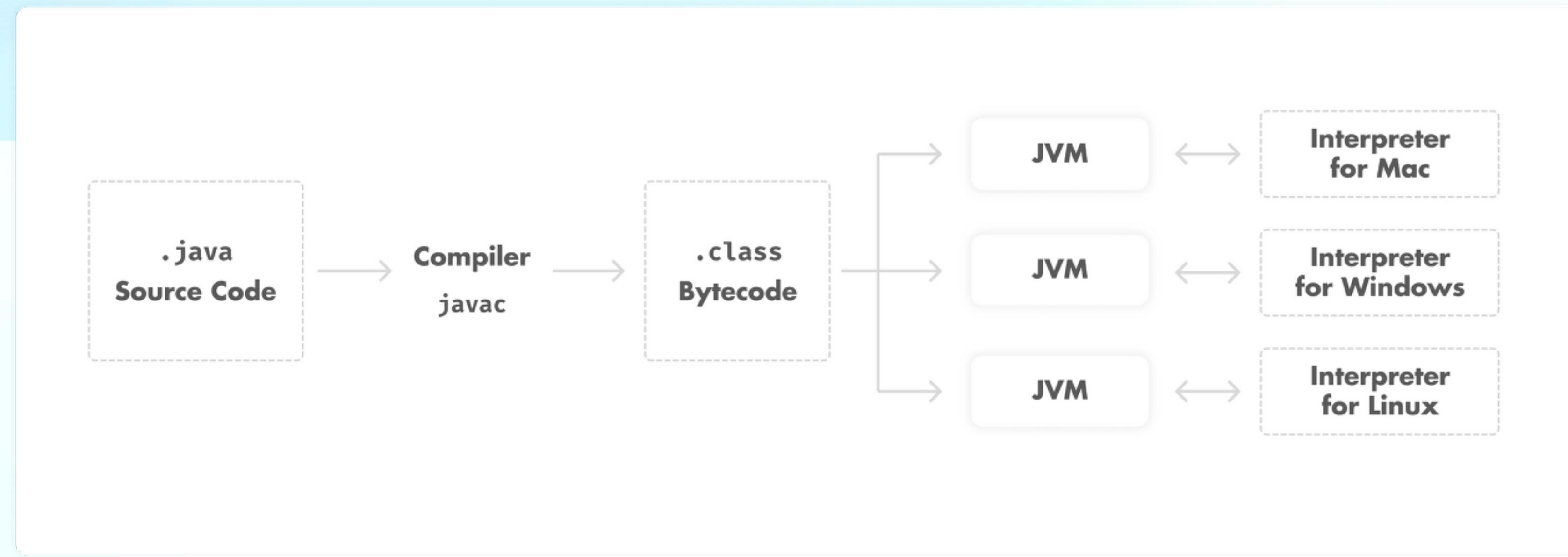
Java 바이트코드 - 들어가기 전에

- Machine code
 - CPU를 제어하는 기계어로 된 명령 코드
 - 각 명령 코드는 CPU 레지스터 또는 메모리에 있는 데이터에 대한 로딩, 저장, 연산 등 수행
 - 일반 프로그래머가 액세스할 수 없는 특정 CPU 내부 코드
- Binary code
 - 두 개의 기호(일반적으로 0, 1)를 사용해 텍스트 또는 프로세스 명령을 나타내는 코드
- Object code
 - 일반적으로 컴파일러에 의해 생성된 중간 언어로 작성된 명령 코드

Java 바이트코드와 코드 캐시

Java 바이트코드

- Java 컴파일러가 소스 코드를 통해 생성한 JVM이 인식할 수 있는 명령어 집합
 - 참고 : Java 컴파일러(javac)는 JVM의 요소가 아닌 JDK의 일부



Java 바이트코드와 코드 캐시

Java 바이트코드 Instruction Set

```
outer:  
for (int i = 2; i < 1000; i++) {  
    for (int j = 2; j < i; j++) {  
        if (i % j == 0)  
            continue outer;  
    }  
    System.out.println (i);  
}
```

A Java compiler might translate the Java code above into bytecode as follows, assuming the above was put in a method:

```
0:  iconst_2  
1:  istore_1  
2:  iload_1  
3:  sipush   1000  
6:  if_icmpge     44  
9:  iconst_2  
10:  istore_2  
11:  iload_2  
12:  iload_1  
13:  if_icmpge     31  
16:  iload_1  
17:  iload_2  
18:  irem  
19:  ifne      25  
22:  goto      38  
25:  iinc      2, 1  
28:  goto      11  
31:  getstatic #84; // Field java/lang/System.out:Ljava/io/PrintStream;  
34:  iload_1  
35:  invokevirtual #85; // Method java/io/PrintStream.println:(I)V  
38:  iinc      1, 1  
41:  goto      2  
44:  return
```

Java 바이트코드와 코드 캐시

Java 바이트코드 Instruction Set

- 명령 집합 아키텍처
 - Java는 레지스터 머신과 스택 머신 둘다 (Operand Stack, Local variables)
- 종류와 인스트럭션 예시
 - 로딩, 저장 (Load & store)
 - `aload_0`, `istore` 등
 - 연산, 논리 (Arithmetic and logic)
 - `ladd`, `fcmpl` 등
 - 타입 변환 (Type conversion)
 - `i2b`, `d2i` 등
 - 객체 생성, 조작 (Object creation and manipulation)
 - `new`, `putfield` 등
 - 피연산자 스택 관리 (Operand stack management)
 - `swap`, `dup2` 등
 - 제어, 전송 (Control transfer)
 - `ifeq`, `goto` 등
 - 메서드 호출, 반환 (Method invocation and return)
 - `invokespecial`, `areturn` 등
- 바이트코드 명령 참조
 - https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions

Java 바이트코드와 코드 캐시

Java 바이트코드 - 메서드 호출, 반환

- `invokevirtual`
 - 가상 메서드를 호출할 때 사용 (다이나믹 디스패치)
 - Java에서 가상 메서드는 기본적으로 `non-static`, `non-final` 메서드
- `invokespecial`
 - 특정 메서드(생성자, 프라이빗 메서드, super 클래스 메서드 등)를 호출할 때 사용
- `invokeinterface`
 - 인터페이스를 통해 메서드가 호출되는 경우 사용
- `invokestatic`
 - 정적 메서드 호출할 때 사용 (컴파일 타임에 결정됨)
- `invokedynamic`
 - 런타임에 동적으로 호출할 메서드를 결정하기 위해 사용 (런타임에 `Call Site` 변경 가능)
 - lambda, Record 클래스, String concat 기능 등에 활용됨
 - JRuby 같은 동적 언어를 지원하기 위해서 추가됨
 - JVM 부트스트랩 메서드를 호출, 반환되는 MethodHandle을 통해 동적으로 메서드를 호출하는 명령(바이트코드)

Java 바이트코드와 코드 캐시

Java 바이트코드 - invokedynamic

invokedynamic

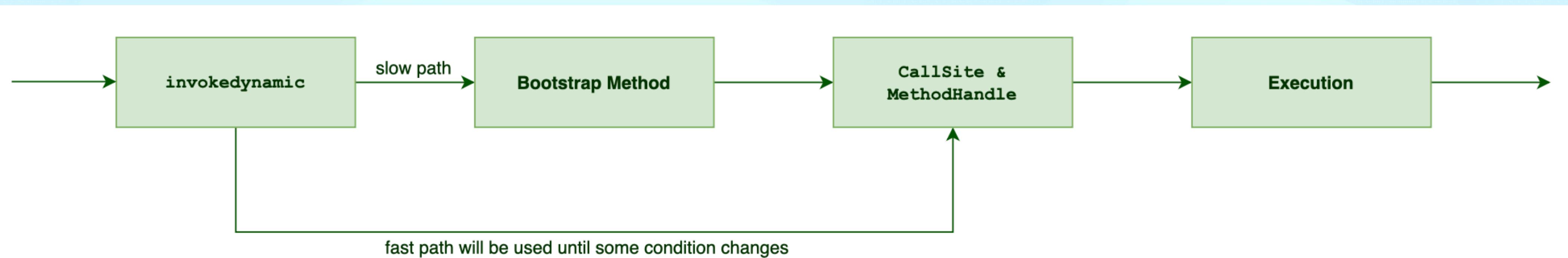
- The name is misleading
- It's not about *invocation*
- ... but about runtime *linkage*

```
graph TD; invoker[invokedynamic] -- Invokes --> bootstrap[Bootstrap Method]; bootstrap -- Returns --> callSite[Call Site]; bootstrap -- MethodHandle --> target[Target]; target -- Compile-time --> bootstrap
```

The diagram illustrates the runtime linkage process for an invokedynamic call. It starts with an **invokedynamic** instruction (represented by a grey box). This instruction **Invokes** a **Bootstrap Method** (also in a grey box). The **Bootstrap Method** **Returns** to the **Call Site** (another grey box). Simultaneously, the **Bootstrap Method** **MethodHandle**s to the **Target** (a grey box). Finally, there is a **Compile-time** dependency from the **Target** back to the **Bootstrap Method**.

Java 바이트코드와 코드 캐시

Java 바이트코드 - invokedynamic



<https://www.baeldung.com/java-invoke-dynamic>

Java 바이트코드와 코드 캐시

Java 바이트코드 - invokedynamic

- 부트스트랩 메서드 (Bootstrap Method)
 - `invokedynamic` 명령 호출 단계에서 JVM이 호출하는 메서드이며 MethodHandle 객체의 생성과 초기화를 담당
 - 동적으로 CallSite를 링킹하는데 사용되는 메서드 (메서드 호출을 유연하게 해줌)
 - Java 컴파일러에 의해 생성되며 JVM에 의해 단 한번만 호출되고 CallSite 객체를 생성
- 메타 팩터리 메서드 (LambdaMetafactory.metafactory)
 - 모든 람다의 부트스트랩 메서드이며 CallSite 객체를 생성
- 콜 사이트 (Call Site)
 - Bootstrap 메서드 호출 결과로 반환되는 객체(CallSite)이며 Method Handle을 담아두는 훌더
 - `invokedynamic` 명령이 호출되는 지점을 의미하며 JVM에 의해 관리됨
- 메서드 핸들 (Method Handle)
 - 메서드를 동적으로 찾거나, 적용, 호출하기 위한 저수준(JVM) 메커니즘이며 해당 연산의 참조를 타입화 한 것
 - 직접 실행 가능한 기본 메서드/생성자/필드 등
 - 로우레벨 명령
 - 반환값/인자의 옵셔널 변환 등
 - Java 컴파일러가 추가한 스태틱 메서드(Method Area에 위치하는 람다로 전달되는 메서드)를 참조
 - 메서드 핸들 생성 순서
 - Lookup 생성 > MethodType 생성 > MethodHandle 찾기 > MethodHandle 호출(invocation)
- 메서드 디스크립터 (Method Descriptor)
 - 메서드의 파라미터(타입, 순서)와 반환 타입을 문자열로 표현한 것

Java 바이트코드와 코드 캐시

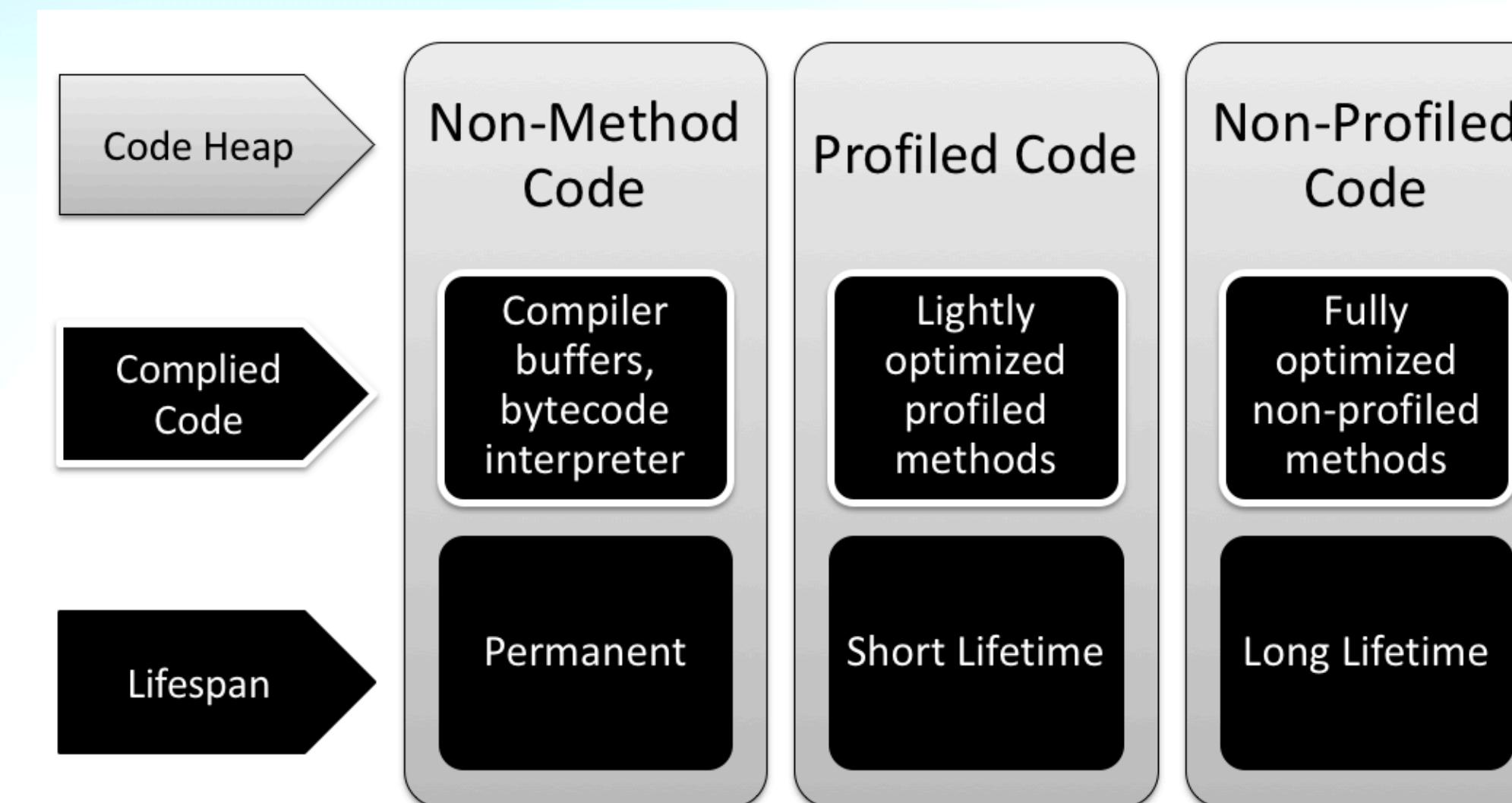
Java 코드 캐시

- JVM이 Java 바이트코드를 컴파일한 네이티브 코드를 저장하는 메모리 영역
 - 실행 가능한 네이티브 코드의 블록을 `method`이라고 함
 - 미할당된 영역에 할당되며 힙(자료구조) 형태로
- JIT 컴파일러가 코드 캐시 영역을 가장 많이 사용하게 됨
- 고정된 크기로 확장 불가하며 가득차면 JIT 컴파일러가 꺼져 추가 코드를 컴파일 하지 않음 (성능 부하)
- 사이즈 옵션
 - InitialCodeCacheSize
 - 캐시 크기 초기값 (기본값 160 KB)
 - ReservedCodeCacheSize
 - 최대 코드 캐시 크기 (기본값 48 MB)
 - CodeCacheExpansionSize
 - 코드 캐시 확장 크기 (기본값 64KB)

Java 바이트코드와 코드 캐시

Java 코드 캐시의 세그먼트

- JDK 9 버전부터 코드 캐시를 3개의 세그먼트로 분리
- non-profiled: 세그먼트 수명이 긴 완전히 최적화된 코드가 포함됨
- profiled-code: 세그먼트에는 수명이 짧은 조금 최적화된 코드가 포함됨
- non-method: 세그먼트에는 바이트코드 인터프리터와 같은 JVM 내부 관련 코드가 포함됨



Java 바이트코드와 코드 캐시

Java 코드 캐시 옵션 변경 예시

```
~/Projects/Teach/Teach_Wanted-PreOnBoarding-Challenge/Chapter1/build/classes/java/main ➤ main ➤ java -XX:+PrintCodeCache com.wanted.Main1
Chapter 1

CodeHeap 'non-profiled nmethods': size=120032Kb used=11Kb max_used=11Kb free=120020Kb
bounds [0x0000000115df8000, 0x0000000116068000, 0x000000011d330000]

CodeHeap 'profiled nmethods': size=120016Kb used=80Kb max_used=80Kb free=119935Kb
bounds [0x000000010e330000, 0x000000010e5a0000, 0x0000000115864000]

CodeHeap 'non-nmethods': size=5712Kb used=1015Kb max_used=1019Kb free=4696Kb
bounds [0x0000000115864000, 0x0000000115ad4000, 0x0000000115df8000]

total_blobs=338 nmethods=60 adapters=195
compilation: enabled
    stopped_count=0, restarted_count=0
full_count=0

~/Projects/Teach/Teach_Wanted-PreOnBoarding-Challenge/Chapter1/build/classes/java/main ➤ main ➤ java -XX:+PrintCodeCache -XX:InitialCodeCacheSize=240m -XX:ReservedCodeCacheSize=1g com.wanted.Main1
Chapter 1

CodeHeap 'non-profiled nmethods': size=521440Kb used=12Kb max_used=12Kb free=521427Kb
bounds [0x00000001412c8000, 0x00000001502c8000, 0x0000000161000000]

CodeHeap 'profiled nmethods': size=521424Kb used=84Kb max_used=84Kb free=521339Kb
bounds [0x0000000121000000, 0x0000000130000000, 0x0000000140d34000]

CodeHeap 'non-nmethods': size=5712Kb used=1015Kb max_used=1020Kb free=4696Kb
bounds [0x0000000140d34000, 0x00000001412c8000, 0x00000001412c8000]

total_blobs=338 nmethods=60 adapters=195
compilation: enabled
    stopped_count=0, restarted_count=0
full_count=0
```

[4] 바이트코드를 컴파일하는 AOT, JIT 컴파일러

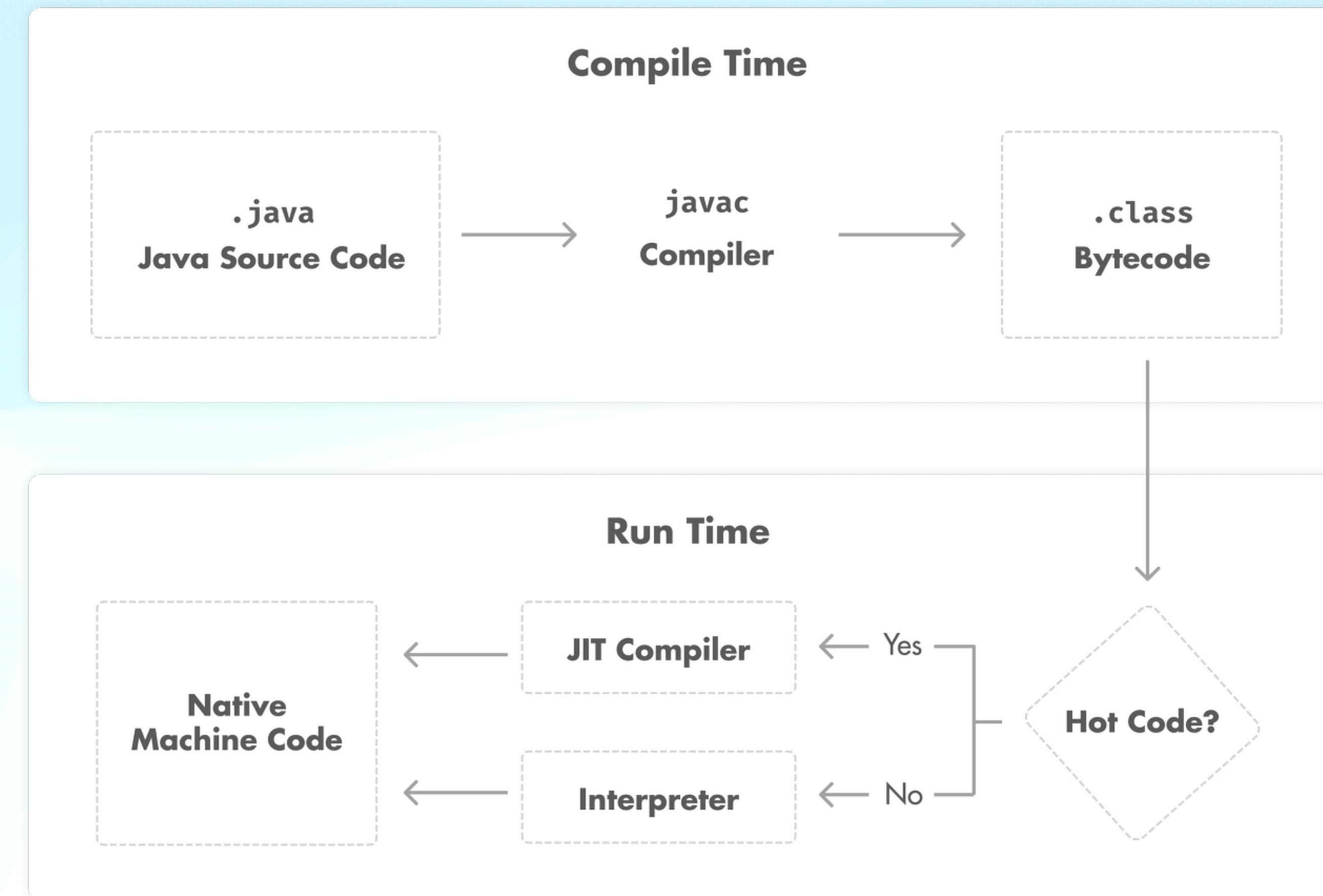
Java 바이트코드를 컴파일하는 AOT, JIT 컴파일러

AOT(Ahead of Time) 컴파일러

- AOT 컴파일 방식은 Java 바이트코드를 앱 '실행 전' 모두 한 번에 컴파일 해두는 방식
 - 워밍업을 단축하는 것을 목표로 함
- Java 바이트코드(.class 파일)를 AOT 컴파일러(jaotc) 통해 컴파일 하는 방식
- 해당 기능(컴파일러)을 수행하는 jaotc(Graal 컴파일러)가 JDK 17에서 제거됨
 - 하지만 해당 프로젝트는 계속 진행중이기 때문에 Graal VM을 별도로 추가하면 사용할 수 있음

Java 바이트코드를 컴파일하는 AOT, JIT 컴파일러

JIT(Just-In-Time) 컴파일



Java 바이트코드를 컴파일하는 AOT, JIT 컴파일러

JIT(Just-In-Time) 컴파일

- JIT 컴파일 방식은 'Hot Method'를 추적하여 컴파일 (Profile-Guided Optimization)
- JIT 컴파일러는 C1과 C2 컴파일러 모드로 구성
 - Java 7 버전 이후에는 두 모드를 혼합하여 C1 모드로 웨업 후 C2로 전환 (7 이전 버전에선 택일)
- C1 (Client Compiler)
 - 런타임에 바이트코드를 기계어로 컴파일
 - 빠른 시작과 견고한 최적화가 필요한 앱에서 사용됨 (GUI 앱 등)
- C2 (Server Compiler)
 - 변환된 기계어를 분석하여 C1보다 오래 걸리지만 더욱 최적화된 네이티브 코드로 컴파일
 - 오랫동안 실행을 하는 서버 앱 용도

Java 바이트코드를 컴파일하는 AOT, JIT 컴파일러

AOT vs JIT

- AOT 장점
 - 부팅 시간(JVM Warm up) 최소화
 - 메모리 사용량 최소화 (런타임 오버헤드)
 - 사전 컴파일 효과로 런타임 특성 중 일부는 예측 가능 (런타임 시간, 리소스 사용량 등)
 - 일반적인 현대 백엔드 API 서버의 아키텍처(클라우드/컨테이너/MSA)와 어울림
 - 빠른 시작으로 부팅시간 단축
 - 최적화를 통한 컨테이너 이미지 크기의 최소화 (리소스 최적화)
- JIT 장점
 - 플랫폼 확장성 (Write Once Run Anywhere)
 - 런타임 정보를 포함한 최적화

실습

아하! 모먼트

- [내가] 수행 업무와 성과를 따로 기록하는 이유는?

[내가] 수행 업무와 성과를 따로 기록하는 이유는?

수행 업무와 성과를 따로 기록하기 전에 고민해보자

1. 수행 업무와 성과를 기록한다는 것은 어떤 의미일까?
2. 왜 기록해야 하고 무엇을 기록해야 할까?
3. 기록을 했던 경험을 통해 기록의 장단점 되짚어 보자!
4. 나의 수행 업무 중 무엇을 기록하면 좋을까?
5. 성과의 기준은 무엇일까?

[내가] 수행 업무와 성과를 따로 기록하는 이유는?

중요 포인트

- 해당 업무에 대한 정의는 뭘까?
 - 기록하며 정의를 정리해보자
- 해당 업무를 해야하는 이유는 무엇일까?
 - 기업/팀/개인 등 다양한 측면에서 생각하기
 - 다른 업무들과 우선순위 비교하기
- 업무의 수행 기간과 마일스톤?
 - 해당 기간 안에 완료할 수 있을까? (누군가에 도움이 필요한가? 어떤 도구가 필요한가?)
 - 기간을 줄일 수 있을까?
 - 무엇을 미리 준비해야 할까?
- 누구에게 어떤 피드백을 어떻게 받아볼 수 있을까?

[내가] 수행 업무와 성과를 따로 기록하는 이유는?

실제로 기록한 포맷 (예시)

- [Jira 티켓 번호/링크] 수행 업무 공식 이름
 - [1234] `공통 모듈`에서 카프카 의존성 제거
- 업무 기간
 - 2023/06/05 ~ 2023/06/14 (또는 ing)
- 수행 업무 설명
 - 현재 공통 모듈에서 JPA의 엔티티 리스너를 통해 카프카 이벤트를 발생시키는 형태이나 해당 공통 모듈을 의존하는 다른 하위 모듈들은 필요 없는 카프카 의존성이 추가됨
 - 따라서 해당 카프카 의존성을 직접 사용하는 하위 모듈로 내리는 작업이 필요
- 실제 수행한 방식 (구현한 코드, 설정 등 포함)
 1. 공통 모듈의 카프카 이벤트 발생 로직을 인터페이스로 추상화 및 `build.gradle`에서 카프카 의존성 제거
 2. 하위 모듈의 `build.grade` 파일에 카프카 의존성 추가 및 이벤트 발생 로직 컴포넌트 추가
 3. ...
- 주의 사항
 - 실제 프로덕션 서버에 적용하기 전에 스테이지 서버에서 테스트 필요
- 지적 사항 및 피드백
 - 해당 업무로 다른 업무 일정이 너무 지연됨
- 배운 점
 - 엔티티 리스너만으로 모든 엔티티의 CRUD 이벤트를 감지할 수 없음
- 성과
 - 공통 모듈의 카프카 의존성을 제거함으로써 공통 모듈의 의미가 퇴색되는 것을 방지
 - 의존하는 다른 하위 모듈들의 카프카 의존성을 제거하여 공통 모듈의 카프카 의존성으로 인해 하위 모듈이 영향을 최소화
- 기타

[내가] 수행 업무와 성과를 따로 기록하는 이유는?

기록의 장점

- 업무 중
 - 은연 중 머리에서만 맴돌던 것들을 밖으로 꺼냄으로써 생각 정리가 깔끔해진다.
 - 사전에 업무를 미리 정의하고 이유 등을 확실히 해두면 집중할 포커스, 마일스톤 등이 확실해진다.
 - 지원 인력, 일정, 다른 업무와의 우선순위 조정 등 많은 것들을 미리 준비할 수 있다.
 - 사전에 미리 준비할 수 있는 것들이 존재한다.
 - 잘못한 부분을 기록하며 같은 잘못을 하지 않게 된다.
- 업무 후
 - 수행 업무를 한 번 더 훑어 보며 복기할 수 있다.
 - 비슷한 업무 수행 시 참고할 수 있다.
 - 오래 전 수행한 업무를 기억할 때 도움이 된다.
 - 연봉 협상, 아직 준비 등 다양한 측면에서도 활용될 수 있다.

[내가] 수행 업무와 성과를 따로 기록하는 이유는?

기록 시 주의 사항

- 업무 내용은 전체공개를 하면 안된다!
 - 나만이 볼 수 있는 비공개 공간에 기록하자!
 - 비공개면서 나는 쉽게 참조할 수 있는 공간은 어딜까?
- 기록을 업무 시간에 하면 안된다!
 - 업무에 포함되지 않았으니 업무 시간 외에 별도의 시간을 확보하자!
- 기록하는 것에 너무 많은 시간을 투자해선 안된다!
 - 핵심 내용들만 최대한 간결하게 축약해서 기록하자!

[내가] 수행 업무와 성과를 따로 기록하는 이유는? 그 외 어떤 것들을 기록하면 좋을까?

- 취직/이직 정보
 - 회사명과 지원하는 팀/포지션
 - 지원 일자
 - 요구하는 기술/스킬 등 자격요건
 - 코딩테스트, 과제, 기술면접 내용 등
- 커피챗
 - 내가 한 질문과 받았던 질문
- 기타

참고 및 출처

- <https://www.oracle.com/java/technologies/javase/jdk-faqs.html>
- https://www.java.com/en/download/help/whatis_java.html
- https://en.wikipedia.org/wiki/Object-oriented_programming#OOP_languages
- https://en.wikipedia.org/wiki/Java_%28programming_language%29
- https://simple.wikipedia.org/wiki/Object-oriented_programming
- https://en.wikipedia.org/wiki/Object-oriented_programming#Shared_with_non-OOP_languages
- <https://www.baeldung.com/java-static-dynamic-binding>
- <https://docs.oracle.com/javase/8/docs/>
- <https://www.oracle.com/java/technologies/javase/jdk11-archive-downloads.html>
- <https://www.oracle.com/java/technologies/javase/jre8-readme.html>
- <https://www.baeldung.com/jvm-vs-jre-vs-jdk>
- <https://docs.oracle.com/javase/8/docs/technotes/tools/>

- <https://docs.oracle.com/en/java/javase/20/migrate/removed-tools-and-components.html>
- <https://openjdk.org/jeps/240>
- <https://www.oracle.com/java/technologies/javase/12-relnote-issues.html#Removed>
- <https://openjdk.org/jeps/313>
- <https://openjdk.org/jeps/322>
- <https://docs.oracle.com/en/java/javase/17/docs/specs/man/index.html>
- <https://docs.oracle.com/en/java/javase/18/docs/specs/man/index.html>
- <https://docs.oracle.com/en/java/javase/19/docs/specs/man/index.html>
- <https://docs.oracle.com/en/java/javase/20/docs/specs/man/index.html>
- <https://openjdk.org/jeps/363>
- <https://openjdk.org/jeps/407>
- <https://www.baeldung.com/oracle-jdk-vs-openjdk>

- <https://docs.oracle.com/javase/8/docs/technotes/guides/jni/>
- <https://docs.oracle.com/javase/specs/jvms/se17/jvms17.pdf>
- <https://www.geeksforgeeks.org/jre-full-form/>
- <https://en.wikipedia.org/wiki/OpenJDK>
- <https://openjdk.org/jeps/410>
- <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html>
- <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html>
- <https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html#jvms-2.5>
- <https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html#jvms-2.6>
- <https://www.baeldung.com/jvm-init-clinit-methods>
- <https://www.baeldung.com/java-polymorphism>
- <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ClassLoader.html>

- <https://docs.oracle.com/javase/tutorial/ext/basics/load.html>
- <https://www.baeldung.com/java-classloaders>
- <https://www.geeksforgeeks.org/classloader-in-java/>
- <https://www.baeldung.com/java-list-classes-class-loader>
- <https://docs.oracle.com/javase/specs/jls/se17/html/jls-6.html#jls-6.7>
- <https://www.baeldung.com/java-classes-same-name>
- <https://www.baeldung.com/java-method-signature-return-type>
- <https://docs.oracle.com/en/java/javase/17/vm/support-non-java-languages.html>
- <https://www.baeldung.com/java-compile-time-constants>
- <https://docs.oracle.com/javase/specs/jls/se17/html/jls-15.html#jls-15.29>
- <https://docs.oracle.com/javase/specs/jls/se17/html/jls-4.html#jls-4.12.4>
- <https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html>

- [https://en.wikipedia.org/wiki/Monitor_\(synchronization\)](https://en.wikipedia.org/wiki/Monitor_(synchronization))
- https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions
- <https://docs.oracle.com/en/java/javase/17/vm/java-virtual-machine-technology-overview.html>
- <https://www.baeldung.com/java-method-handles>
- <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/invoke/CallSite.html>
- <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/invoke/MethodHandle.html>
- <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/invoke/LambdaMetafactory.html>
- <https://www.baeldung.com/java-invoke-dynamic>
- <https://wiki.openjdk.org/display/HotSpot/Method+handles+and+invokedynamic>
- https://en.wikipedia.org/wiki/Symbol_table
- <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>
- [https://en.wikipedia.org/wiki/Interpreter_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing))

- <https://en.wikipedia.org/wiki/Compiler>
- <https://www.baeldung.com/java-compiled-interpreted>
- <https://docs.oracle.com/en/java/javase/17/docs/specs/jni/index.html>
- https://en.wikipedia.org/wiki/Machine_code
- https://en.wikipedia.org/wiki/Binary_code
- https://en.wikipedia.org/wiki/Object_code
- https://en.wikipedia.org/wiki/Java_bytecode
- <https://www.baeldung.com/jvm-code-cache>
- <https://www.baeldung.com/cs/cache-memory>
- <https://www.oreilly.com/library/view/mastering-java-11/9781789137613/f5b7efc5-4e35-46ea-ba2f-95b99651b1b7.xhtml>
- <https://www.oracle.com/technical-resources/articles/java/architect-evans-pt1.html>
- <https://www.oreilly.com/library/view/java-performance-2nd/9781492056102/ch04.html>

- <https://www.baeldung.com/ahead-of-time-compilation>
- <https://www.baeldung.com/graal-java-jit-compiler>
- <https://www.graalvm.org/latest/docs/getting-started/>
- <https://docs.oracle.com/en/java/javase/17/docs/specs/man/javap.html>
- <https://bytebuddy.net/#/>
- <https://www.baeldung.com/byte-buddy>