JVM Class Loading

JVM은 클래스 파일을 어떻게 알아서 찾을 수 있을까요?

20210463 박연종

park@duck.com

SSL SEMINAR

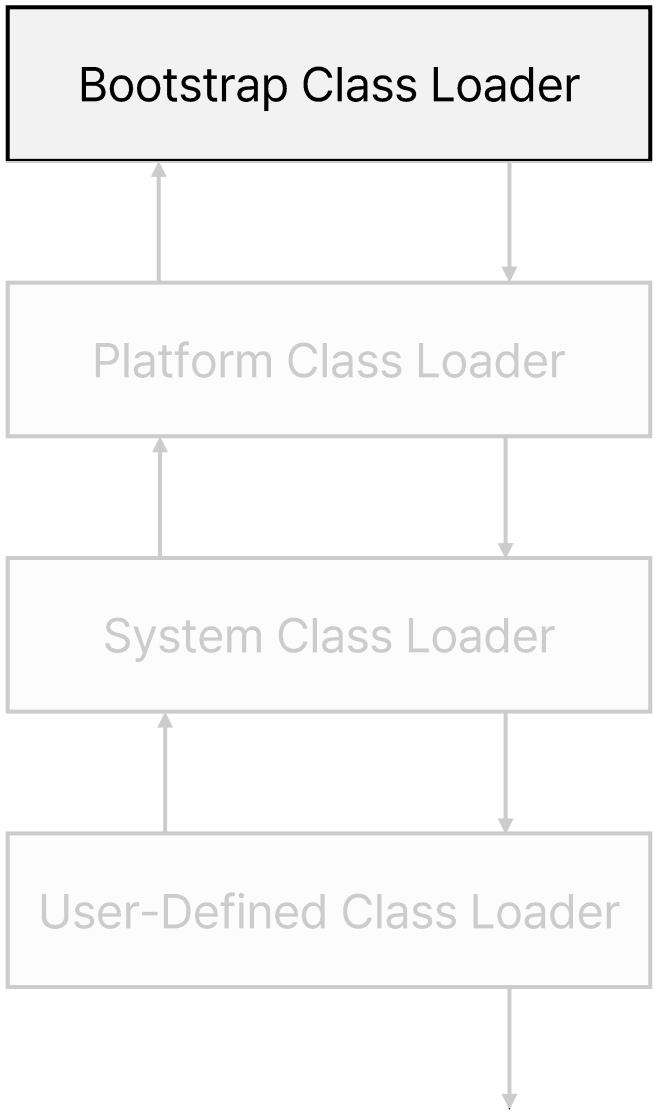
Photo by Pawel Czerwinski

목차

01.
클래스 로더

Class Loader

Bootstrap Class Loader

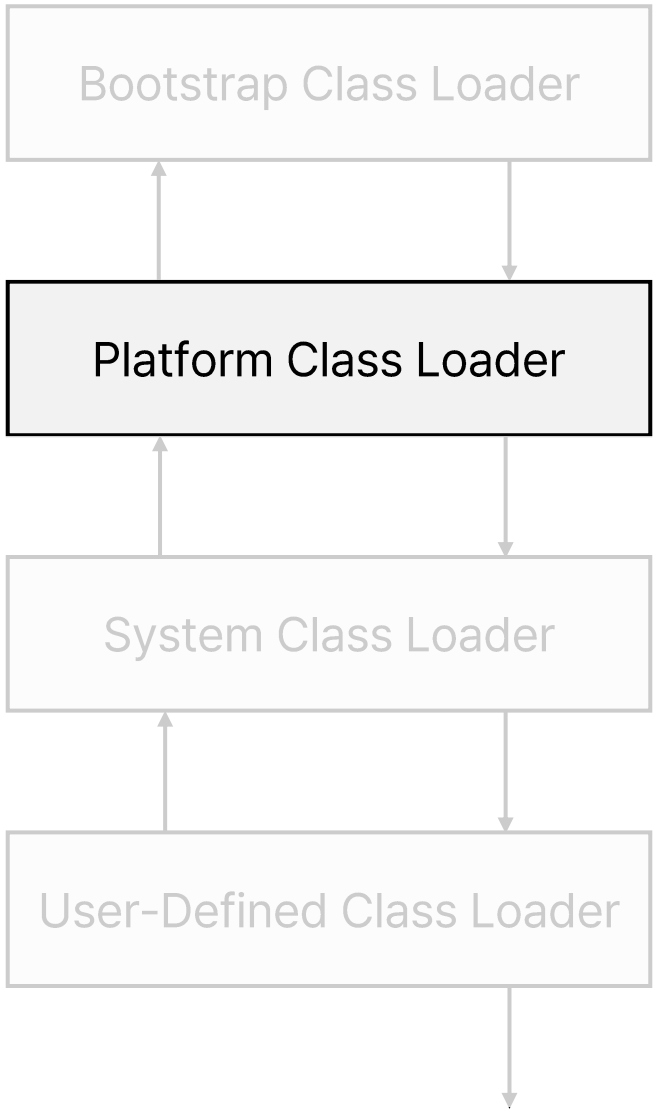


- 자바 가상 머신 시작 시 가장 먼저 실행되는 클래스 로더
- 네이티브 언어(C, C++, assembly 등)로 구현
- JVM이 동작하는 플랫폼 환경에 따라 다르게 구현
- 자바 클래스 로더와 필수 자바 클래스만 로드

- Java 8 `$JAVA_HOME/jre/lib/rt.jar`
- Java 9 `$JAVA_HOME/lib/modules`
  (Jigsaw 프로젝트*의 일환)

Jigsaw 프로젝트 Java 언어에 모듈 시스템을 도입하는 프로젝트

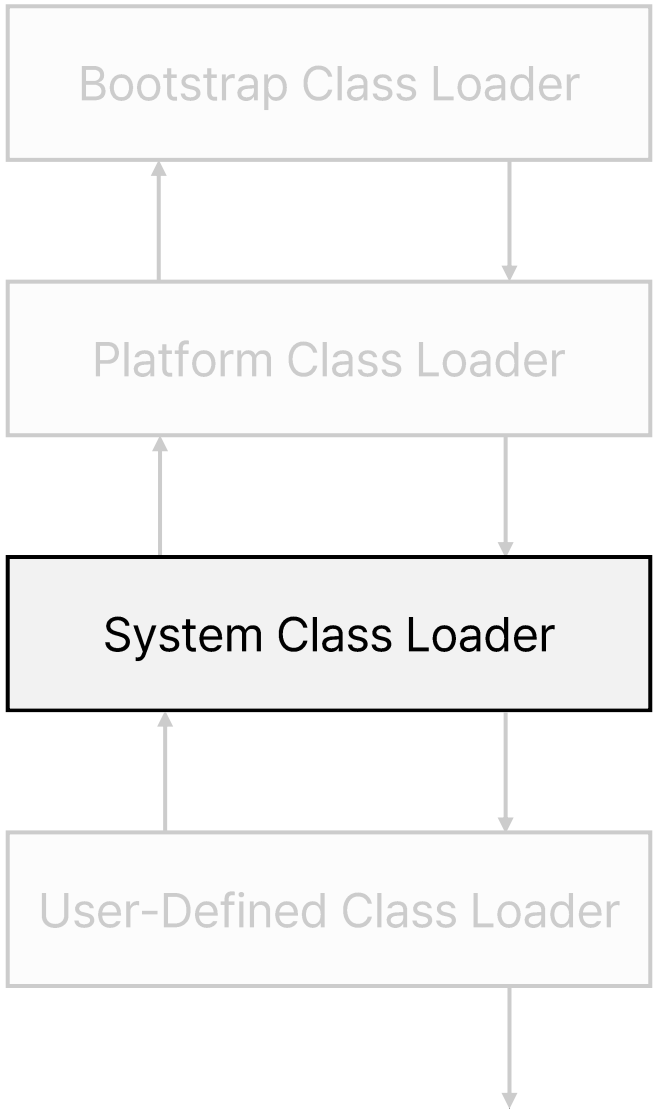
Platform Class Loader *(Extension Class Loader)*



- java.base 모듈을 제외한 표준 자바 클래스를 로드하는 자바로 작성된 클래스 로더

- Java 8 `$JAVA_HOME/jre/lib/ext/*`
  `java.ext.dirs`
- Java 9 `$JAVA_HOME/lib/modules`

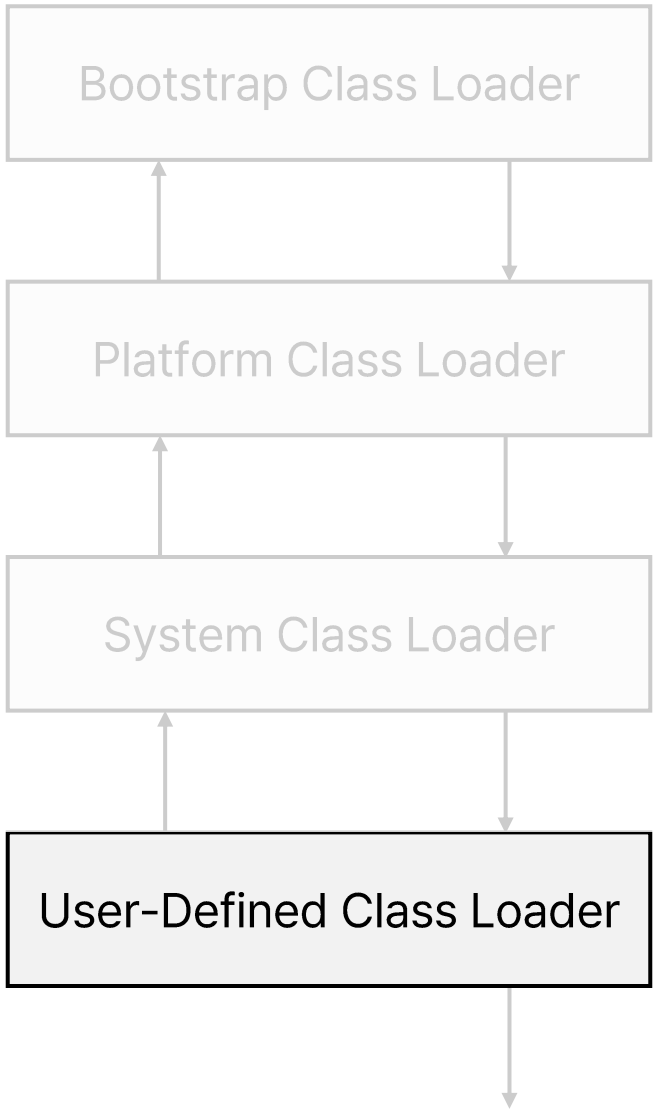
System Class Loader *(Application Class Loader)*



- 자바 프로그램 실행 시 지정한 클래스 패스에 있는 클래스 파일 혹은 jar에 속한 클래스 파일을 로드하는 자바로 작성된 클래스 로더

- 지정한 클래스 패스

User-Defined Class Loader



- 애플리케이션 개발자가 추상 클래스 ClassLoader를 상속 받아 `findClass()` 메서드를 재정의한 자바로 작성된 클래스 로더

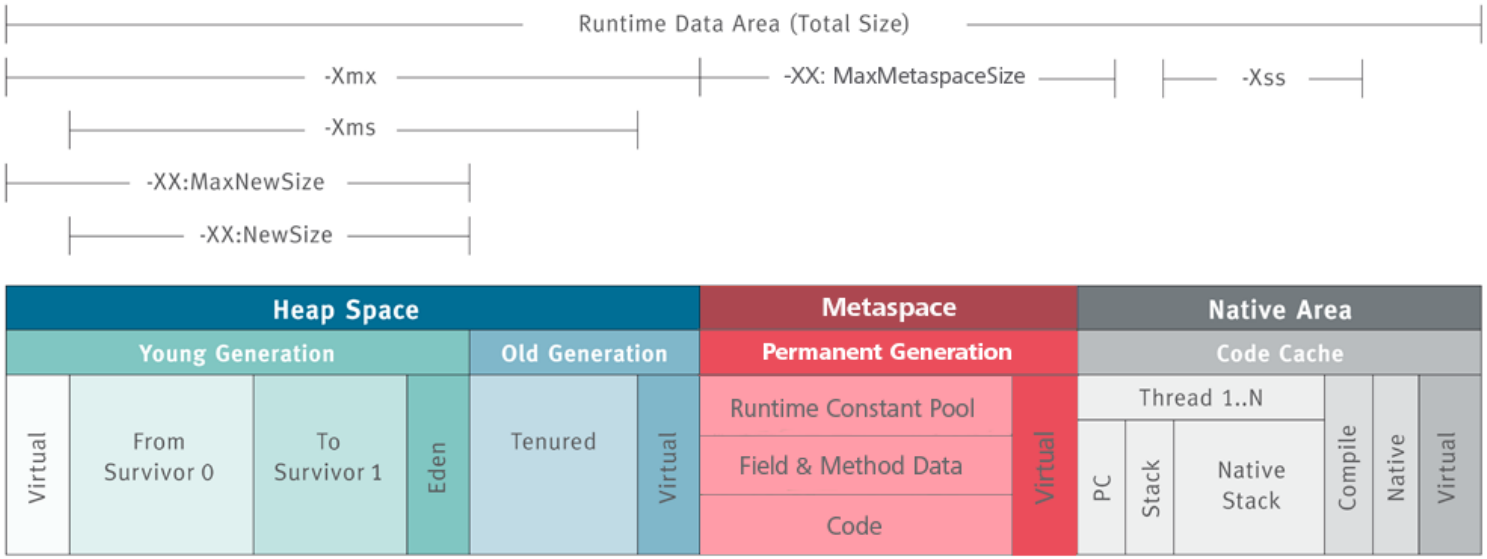
- 해당 ClassLoader에 지정한 탐색 경로

02.
로딩

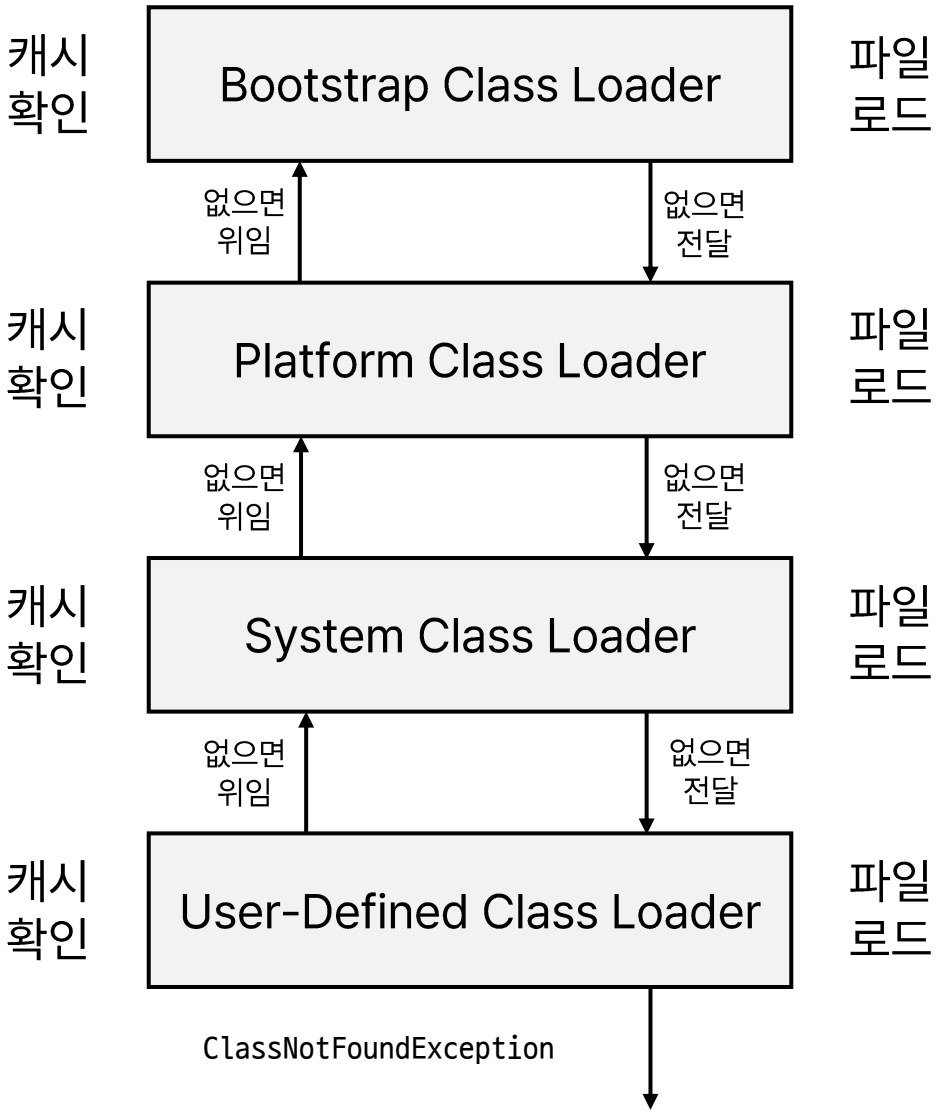
Loading

로딩 *(Loading)*

- .class 파일을 Runtime Data Area의 Metaspace*(Method)* 영역에 적재하는 과정
- 필드 이름, 타입, 접근 지정자와 메서드 이름, 반환형, 매개변수, 접근 지정자 등이 Field & Method Data 영역에, 메서드의 바이트 코드가 Code 영역에 적재

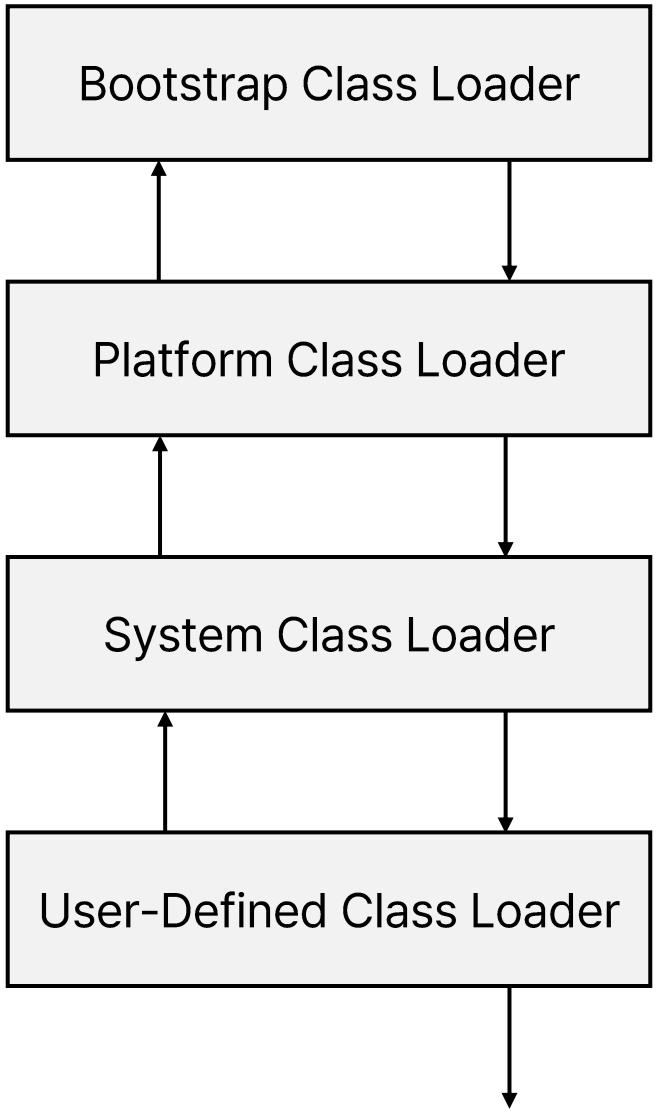


위임 계층 원칙 *(Delegation Hierarchy Principle)*



- 클래스 로드 요청을 받으면 클래스 로더 캐시, 상위 클래스 로더, 하위 클래스 로더 순으로 해당 클래스가 이전에 불러온 클래스인지 확인
- 최상단 클래스 로더까지 확인해도 불러온 적이 없으면 파일 시스템에서 해당 클래스를 탐색

가시성 원칙 *(Visibility Principle)*

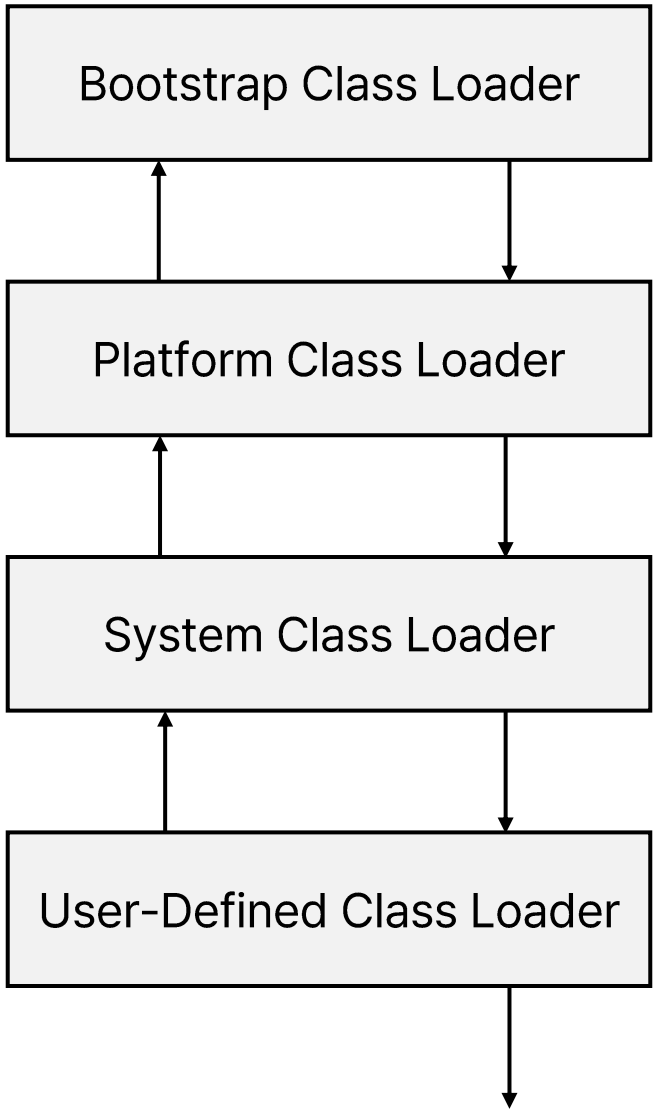


- 하위 클래스 로더는 상위 클래스 로더가 로드한 클래스를 확인 가능
- 상위 클래스 로더는 하위 클래스 로더가 로드한 클래스를 확인 불가

- 위임 모델에 따라 클래스 로드 요청이 발생 시 하위 클래스 로더가
  상위 클래스 로더에 해당 클래스를 로드 요청하기 때문에
  자연스럽게 하위 클래스 로더는 상위 클래스 로더의 로드 클래스를 확인 가능

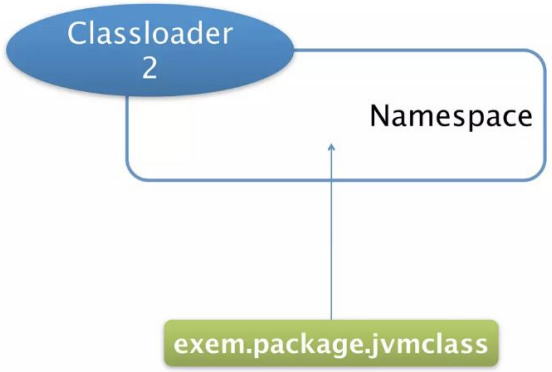
- 클래스 로더의 탐색 경로는 지정돼 있으나 상위 클래스 로더가 하위 클래스 로더의 로드 클래스를 확인할 수 있다면 탐색 경로를 나누어 로드 효율을 높인다는 이점이 없음

유일성 원칙 *(Uniqueness Principle)*

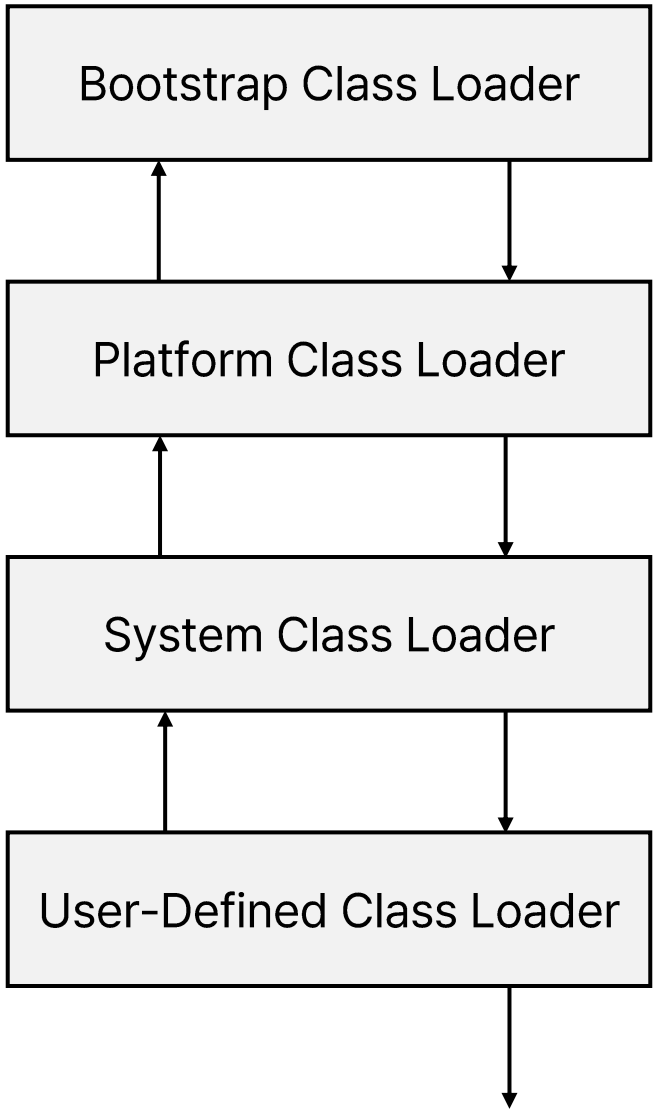


- 상위 클래스 로더가 로드한 클래스를 하위 클래스 로더가 로드하지 않음
- 클래스를 중복 로드하지 않아 클래스의 유일성을 보장함

- 유일성을 보장하기 위해 각 클래스 로더마다 클래스 이름을 보관하는 공간인 '네임 스페이스'(캐시)에 FQCN*(Fully Quallified Class Name)*을 저장
- 로드해야 하는 클래스가 네임 스페이스의 FQCN에 있다면 해당 클래스를 사용, 없다면 위임 모델에 따라 클래스를 로드



언로드 금지 원칙 *(No Unloading Principle)*



- 클래스 로더는 클래스 로드만 가능하고 언로드할 수 없음

- 언로드는 가비지 컬렉션에 의해 수행됨
- 클래스 로더에 의해 언로드 시 클래스 로더 및 클래스의 모든 참조가 제거되지 않는 등 문제가 발생할 수 있음

03.
링킹

Linking

링킹 *(Linking)*

- 로드한 .class 파일을 검증하고 사용할 수 있도록 준비하는 과정
- 세 단계(검증 → 준비 → 해석)로 나누어 살펴볼 수 있음

검증 *(Verification)*

- .class 파일이 자바 언어 명세를 준수해 작성됐는지, JVM 규격에 따라 검증된 컴파일러에서 바이트 코드가 생성됐는지 등을 확인하는 과정
- 검증 과정을 통해 안전하지 않은 코드가 JVM을 손상시키는 것을 방지할 수 있음
- 바이트 코드 검증기가 검증 과정을 담당
  - 클래스 로드 과정 중 가장 복잡한 단계이자 가장 오랜 시간이 소요
  - 클래스 로드 과정이 지연되나 바이트 코드 실행 시 검사를 여러 번 할 필요가 없어 전체적으로 효율적이며 효과적
  - 바이트 코드 검증기에서 검증 실패 시 `VerifyError` 예외 발생

준비 *(Preparation)*

- 클래스 또는 인터페이스에 필요한 메모리 할당하고, 정적 필드를 생성 및 기본 값*으로 초기화
- Java 8 이상에서는 Method 영역에 생성되던 정적 필드가 Heap 영역에 생성됨

- `static int SSL = 449;`
  - `boolean : false`
  - `int, long, byte, short : 0`
  - `double, float : +0.0`
  - `char : '\u0000'`
  - `Reference : null`

해석 *(Resolution)*

- Runtime Data Area의 Runtime Constant Pool에 존재하는 심볼릭 참조를 직접 참조로 대체
- 클래스 이름, 필드 이름, 메서드 이름의 심볼릭 참조를 실제 메모리 주소인 직접 참조로 변환

04.
초기화

Initializing

초기화 *(Initializing)*

- 적재된 자바 바이트 코드를 읽어 명시한 값을 정적 변수에 할당*하고 정적 초기화 블록을 실행
- 상수(static final)를 Runtime Constant Pool에 생성 및 초기화

- `static int SSL = 449;`

05.
클래스 로더 동작 확인

CustomClassLoader


```java
@Override
public Class loadClass(String name) throws ClassNotFoundException {
    System.out.println("로드 시작 ("+name+")");
    if (name.startsWith("com.patulus.classloader")) {
        System.out.println(name + " 클래스는 커스텀 클래스 로더에 의해 로드되었습니다.");
        return getClass(name);
    }
    System.out.println("부모 클래스 로더("+((this.getParent().getName().equals("app")) ? "System Class Loader" : this.getParent().getName())+")로 위임");
    return super.loadClass(name);
}
```

인스턴스 생성 시


```java
package com.patulus.classloader;

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

public final class Main {
    static {
        System.out.println("* Main 로드  완료 *");
    }

    public static void main(String[] args)
    {
        System.out.println("=====");

        Map<String, String> map = new HashMap<>();
        map.put("name", "kitebell");
        System.out.println(map.get("name"));

        System.out.println("=====");

        ClassA a = new ClassA();
        a.createB();

        System.out.println("=====");

        ClassA a2 = new ClassA();
        a2.createB();
    }
}
```

```
PS D:\calva\Programming\Java\helloJava\out\production\helloJava> java -D"java.system.class.loader"="com.patulus.classloader.CustomClassLoader" com.patulus.classloader.Main
[0.009s][warning][cds] Archived non-system classes are disabled because the java.system.class.loader property is specified (value = "com.patulus.classloader.CustomClassLoader"). To use archived non-system classes, this property must not be set
로드 시작 (com.patulus.classloader.Main)
com.patulus.classloader.Main 클래스는 커스텀 클래스 로더에 의해 로드되었습니다.
로드 시작 (java.lang.Object)
부모 클래스 로더(System Class Loader)로 위임
로드 시작 (java.util.Map)
부모 클래스 로더(System Class Loader)로 위임
로드 시작 (java.lang.String)
부모 클래스 로더(System Class Loader)로 위임
로드 시작 (java.lang.System)
부모 클래스 로더(System Class Loader)로 위임
로드 시작 (java.io.PrintStream)
부모 클래스 로더(System Class Loader)로 위임
* Main 로드  완료 *
===
로드 시작 (java.util.HashMap)
부모 클래스 로더(System Class Loader)로 위임
kitebell
===
로드 시작 (com.patulus.classloader.ClassA)
com.patulus.classloader.ClassA 클래스는 커스텀 클래스 로더에 의해 로드되었습니다.
* ClassA 로드 완료 *
ClassA 인스턴스 생성
createB() 실행됨
로드 시작 (com.patulus.classloader.ClassB)
com.patulus.classloader.ClassB 클래스는 커스텀 클래스 로더에 의해 로드되었습니다.
* ClassB 로드 완료 *
ClassB 인스턴스 생성
===
ClassA 인스턴스 생성
createB() 실행됨
ClassB 인스턴스 생성
```

실제 로드된 클래스의 클래스 로더 확인

클래스 이름.class.getClassLoader()

```
Main 클래스는 com.patulus.classloader.CustomClassLoader@5674cd4d에 의해 로드되었습니다.
Object 클래스는 null에 의해 로드되었습니다.
Map 클래스는 null에 의해 로드되었습니다.
String 클래스는 null에 의해 로드되었습니다.
System 클래스는 null에 의해 로드되었습니다.
PrintStream 클래스는 null에 의해 로드되었습니다.
HashMap 클래스는 null에 의해 로드되었습니다.
ClassA 클래스는 com.patulus.classloader.CustomClassLoader@5674cd4d에 의해 로드되었습니다.
ClassB 클래스는 com.patulus.classloader.CustomClassLoader@5674cd4d에 의해 로드되었습니다.
Class 클래스는 null에 의해 로드되었습니다.
```

로드되지 않은 클래스의 정적 필드/상수 사용 시


```java
public static void main(String[] args)
{
    System.out.println("정적 필드 호출 시");
    System.out.println(TestA.SSL);

    System.out.println("상수 호출 시");
    System.out.println(TestB.SSL);
}
```

```java
public static int SSL = 4491;
```

```java
public static final int SSL = 4492;
```

```
정적 필드 호출 시
로드 시작 (com.patulus.classloader.test.TestA)
com.patulus.classloader.test.TestA 클래스는 커스텀 클래스 로더에 의해 로드되었습니다.
로드 시작 (com.patulus.classloader.test.Parent)
com.patulus.classloader.test.Parent 클래스는 커스텀 클래스 로더에 의해 로드되었습니다.
* Parent 로드 완료 *
* TestA 로드 완료 *
4491
상수 호출 시
4492
```

실제 로드된 클래스의 클래스 로더 확인

클래스 이름.class.getClassLoader()

```
TestA 클래스는 com.patulus.classloader.CustomClassLoader@5674cd4d에 의해 로드되었습니다.
TestB 클래스는 로드하지 않았습니다.
```

감사합니다

Email / park@duck.com
Insta / @yeonjong.park
GitHub / patulus

Photo by Pawel Czerwinski