<JSTORM>

JVM의 기본원리 및 클래스로더의 기초

Revision <1.0>

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

1)

Document title:	JVM의 기본 원리 및 클래스로더의 기초
Document file name:	jvm_jinho.pdf
Revision number:	<1.0>
Issued by:	<최진호>(csecau@orgio.net)
Issued Date:	<2000/07/25>
Status:	Draft

Content Information

Audience:	중급, 고급
Abstract:	JVM의 기본 작동원리와 클래스로더의 기초 및 간단한 클래스로더 구현을 다루었다.
Reference:	1) http://www.artima.com/jvm/index.html 2) 자바가상머신(곽용재역, 인포북,2000)
Benchmark information:	

1)

Document Approvals

박지훈		
고문	Signature	date
지위	Signature	date

Revision History

Revision	Date	Author	Description of change

Table of Contents

1. 개념이해

'CPU에 대한 추상화를 수행하고, 각각의 컴퓨터에서 이 추상화된 CPU를 구현해주면 어떨가?'라는 것이 자바언어의 개발자들이 처음 자바를 개발할 때의 아이디어였다. 만약 어떤 컴퓨터에 이러한 가상 컴퓨터가 구현되어있다면, 가상 컴퓨터에서 수행될 수 있도록 작성된 프로그램들은 그 시스템에서 동작할 것이고, 따라서 프로그래머가 한 번만 프로그램을 작성하면 이 프로그램이 모든 시스템에서 실행될 수 있다. 이러한 가상 컴퓨터를 자바 가상 머신(Java Virtual Machine, JVM)이라고 부른다. JVM이 어떤 특정한 CPU에 기반을 두고 있지 않기 때문에, 각 CPU에서 제공해주는 비트나 바이트 수준의 메모리 접근보다는, 보다 일반적으로 메모리 접근할 수 있는 고 수준의 추상화가 제공되어야만 했다. 그래서 JVM에서는 메모리를 객체들의 집합이라는 개념으로 다루어준다.

컴파일: 자바 프로그램은 자바 언어로 쓰여진, 클래스 정의의 집합이라고 할 수 있다. 자바 컴파일러는 자바 프로그램을 JVM이 이해할 수 있는 형식으로 바꾸어준다. 이 변환 작업을 컴파일이라고 한다.
컴파일 과정을 거쳐 변환된 자바 프로그램은 바이트들의 집합으로 나타내어지는 클래스 파일 형식으로 바뀌어진다. 이 바이트들은 파일, 메모리, 웹서버 또는 데이터베이스등, 바이트를 저장할 수 있는 곳이라면 어디에라도 저장할 수 있다.

2. JVM의 정의

자바 가상머신이 어떻게 구성되어야 하는지를 나타내는 공식적인 문서는, 팀 린드홀름과 프랭크 엘린이 쓴 "자바 가상 머신 명세서(Java Virtual Machine Specification)"이다. 이 명세서는 다음의 세 가지가 정의되어 있다.
- 명령어들의 집합과 각 명령어의 의미의 정의: 이 명령어들은 바이트 코드(Bytecode)라고 불린다.
- 클래스 파일 형식이라고 불리는 바이너리 형식: 이 형식은 바이트 코드를 전송하는데 사용되며, 클래스가 플랫폼에 독립적으로 동작할 수 있도록 해 준다.
- 프로그램에 문제가 없는지를 확인하는 알고리즘: 이 알고리즘을 검증이라고 한다.

3. JVM의 구성

- Class Area: 프로그램 코드와 상수가 저장된다.
- Java Stack: 프로그램의 수행 도중 어떤 메소드가 호출되었는지를 기록하고, 각 메소드의 호출과 관련된 데이터들의 정보를 기록한다.
- Heap: 객체가 저장된다.
- Native method stacks: 네이티브 메소드를 지원하기 위해 존재한다.

1) Class Area

시스템에 로드된 클래스들이 저장된다. 구현된 메소드는 메소드 영역이라는 공간에 저장되며, 상수들은 상수풀이라는 곳에 저장된다. 클래스 정의는 객체를 생성하기 위한 템플리트로 사용되며, 생성된 객체들은 힙에 저장된다.
클래스는 다양한 구성요소를 가지고 있다.
* 수퍼클래스
* 인터페이스들의 리스트
* 필드들의 리스트

* 메소드들의 리스트와 구현된 메소드
* 상수들의 리스트

2) Java Stack

메소드가 호출될 때마다, 스택 프레임이라고 불리는 새로운 데이터 영역이 생성된다. 이러한 스택 프레임이 모여서 Java Stack을 생성한다. 자바 스택에서 스택의 최상단에 존재하는 스택 프레임은 활성화된 스택프레임이라고 불린다. 프로그램이 수행될 때는 활성화된 프레임의 피연산자 스택과 지역변수 배열만이 사용된다. 만약 메소드가 호출된다면, 새로운 자바 스택 프레임이 생성되고, 새로 생성된 프레임이 자바 스택의 최상단으로 오게된다.

각각의 스택 프레임은 피연산자 스택(operand stack), 지역 변수들을 저장하는 배열, 그리고 현재 실행중인 명령어를 나타내는 포인터를 갖는다.

3) Heap

객체들은 힙에 저장된다. 각 객체는 Class Area에 있는 클래스와 연관되어있다. 각 객체에는 필드들을 저장하는 빈 슬롯들이 존재한다. 클래스의 Non-static 필드 각각에 대해 하나씩의 슬롯이 할당되고, 수퍼클래스의 Non-Static 필드 각각에 대해 하나씩의 슬롯이 할당된다.

4) Native Method Stack

Native Method Stack은 다른 JVM 메소드와 같은 방식으로 사용된다. 그러나 이 스택을 사용하는 메소드들은 JVM의 명령어들이 아닌, 다른 언어를 사용해서 구현된다. 이것은 프로그래머가 자바만을 사용해서는 해결할 수 없는 일, 예를 들면, 플랫폼에 독립적인 인터페이스를 제작하는 일들을 다른 프로그래밍 언어를 사용하여 처리할 수 있도록 해 준다.

5) 예

다음의 코드는 하나의 필드로부터 객체를 얻어내고, 얻어낸 객체의 메소드를 호출하며, 메소드의 매개변수로는 문자열을 넘겨주게 된다.

```
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "Hello, World"
invokevirtual java/io/PrintStream/println (Ljava/lang/String;)V
```

첫 번째 명령어는 java/lang/System 클래스의 out필드의 값을 검색한다. 이 필드의 값은 java/io/PrintStream 클래스, 혹은 서브클래스의 객체일 것이다.

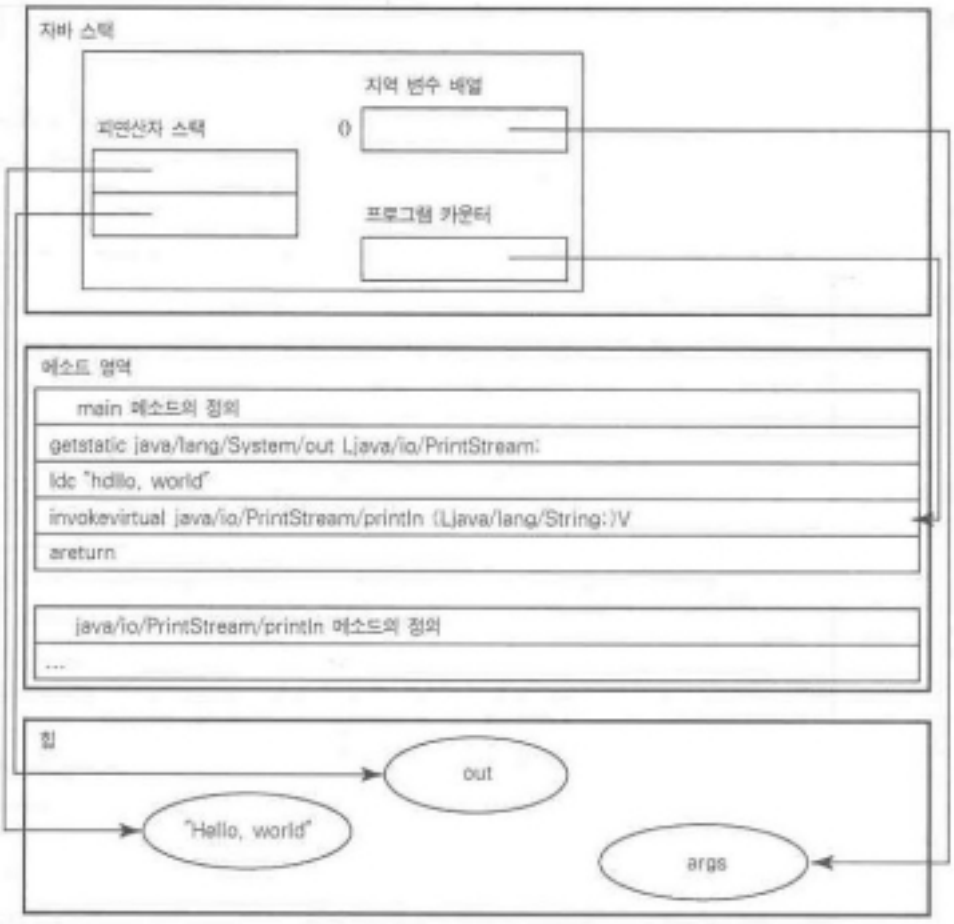
두 번째 명령어는 'Hello, World'라는 문자열 상수를 스택에 푸시한다. 이 문자열은 java/lang/String 클래스의 객체이다. 여기까지 실행하게 되면 스택은 다음과 같은 모습이 될 것이다.

마지막 명령어는 메소드를 호출한다. 메소드의 이름은 println이며, java/io/PrintStream클래스에 이 메소드가 정의되어 있을 것이다. 이 명령어는 스택에 java.lang.String 타입의 인수가 있을 것이고, 메소드는 아무것도 반환하지 않는다는 것을 나타내고 있다. 또한 이 명령어는 스택에서, java.lang.String 인수의 아래쪽에 java/io/PrintStream클래스의 객체가 있을 것이라고 나타내고 있으며, 이것은 실제로 메소드가 호출되는 목적지를 나타낸다. 결과적으로 이 명령어는 메소드를 호출할 것이고, 'Hello, World'라는 문자열을 출력하게 될 것이다.

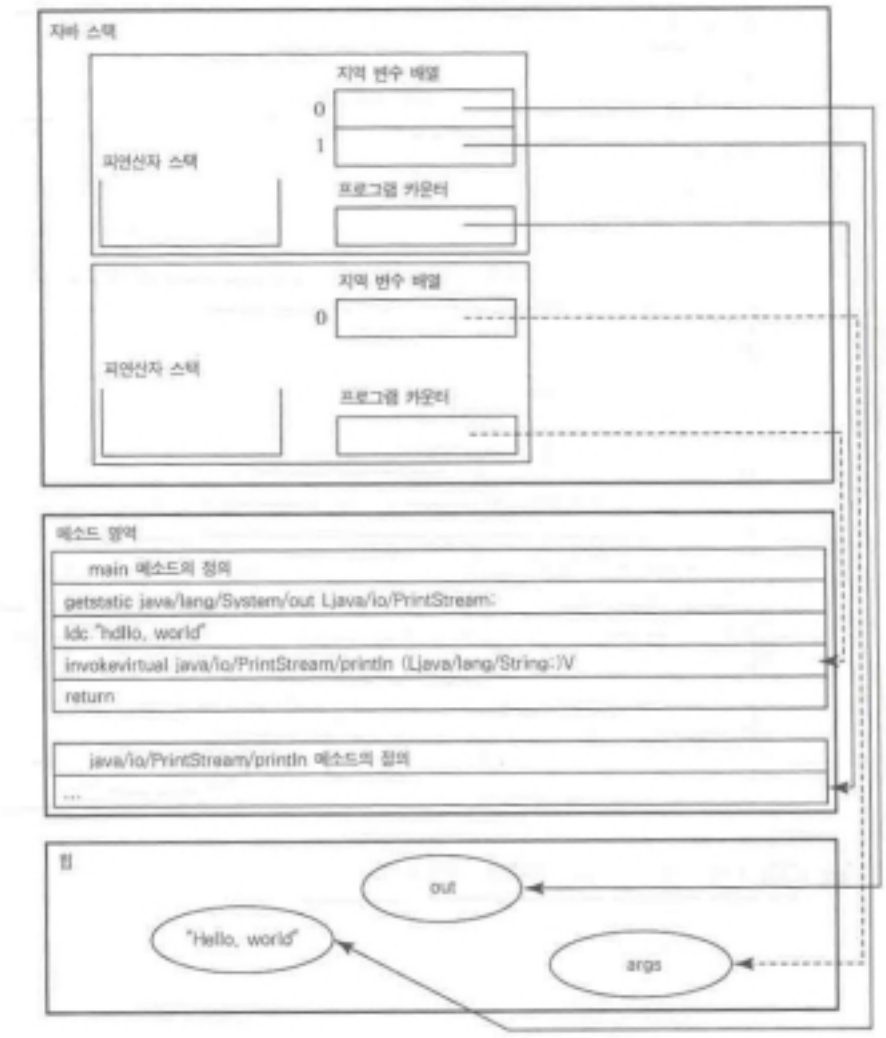
이 메소드가 호출되고 나면, 메소드의 매개변수로 넘겨진 인수와 메소드가 호출되는 목적지가 스택에서 모두 제거되고, 스택은 다시 텅 빈 상태가 될 것이다.

다음 그림은 getstatic과 ldc를 실행한 직후의 시스템의 상태를 보여준다. 자바 스택에는 하나의 프레임이 있다. 이 프레임 내의 피연산자 스택의 최상단은 힙에 있는 Hello,World 문자열을 가리키고 있으며, 그 아래쪽 슬롯은 System.out 객체를 가리키고 있다. 프레임에는 하나의 지역 변수가 있으며, 이것은 명령어 라인 인수를 저장하고 있는 area배열을 가리키고 있다. 마지막으로 스택 프레임 내의 프로그램 카

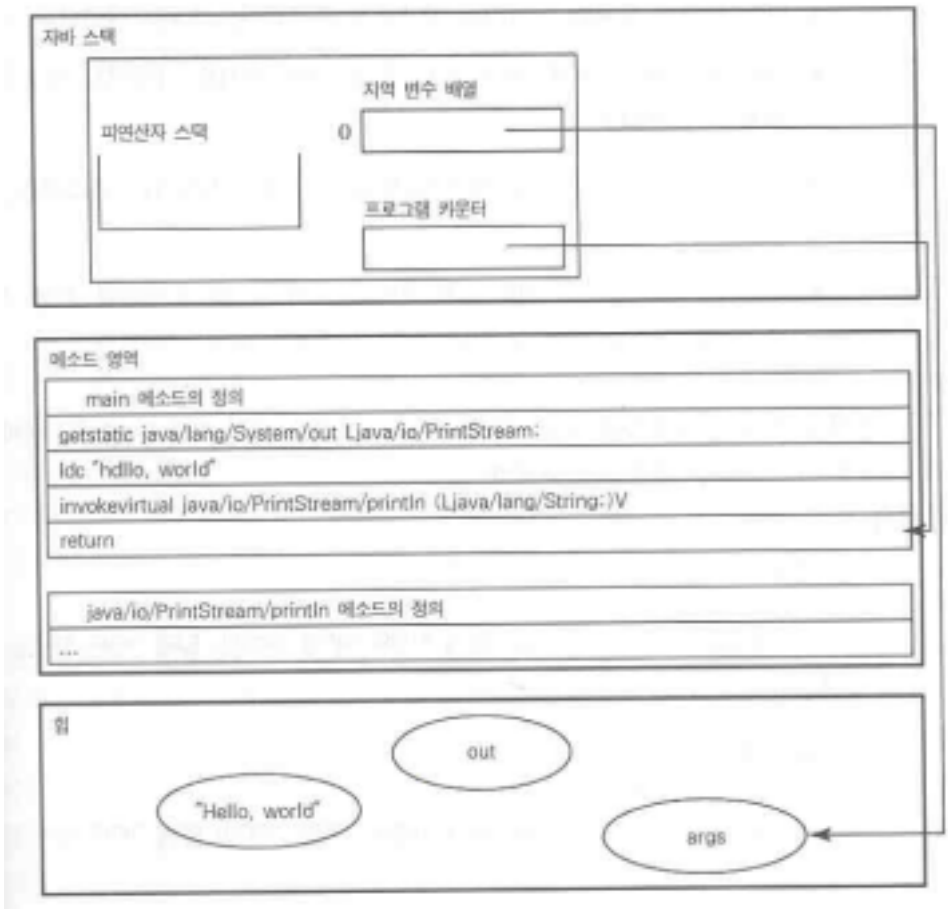
운터는 invokevirtual명령어를 가리키고 있다.



또한 다음 그림은 invokevirtual 명령어가 실행된 뒤의 상태를 보여준다. 명령어의 인수들은 피연산자 스택으로부터 꺼내어지고, 새로운 스택 프레임이 생성된다. 이전의 스택 프레임은 비 활성화되고 스택 프레임이 다시 활성화될 때까지는 바뀌지않는다. 새로운 메소드가 호출되었을 때 벌어지는 일들은 이것이 전부이다.



이제 JVM은 println 메소드가 종료할 때까지, println 메소드 내의 명령어들을 수행한다. 다음 그림은 println 메소드가 종료한 뒤의 상태를 보여준다.



새로 생성되었던 스택 프레임은 제거되어있다. 메소드의 매개변수로 주어졌던 데이터들로 피연산자 스택으로부터 제거되었기 때문에, 피연산자 스택은 텅 비어 있다. 프로그램 카운터는 메소드를 호출한 명령어의 다음 명령어를 가리키고 있으며, JVM은 이 명령어부터 계속 진행하게 될 것이다. 주의해서 볼 부분은 힙에 있는 2개의 객체를 가리키는 선이 더 이상 존재하지 않는다는 점이다. JVM은 이 객체들이 더 이상 사용되지 않는 것을 알게되고, 새로운 객체들이 이 공간을 사용할 수 있도록 객체들을 제거한다. 이것을 가비지 콜렉션이라고 부른다.

6) 캐스팅, 필드, 메소드, 그리고 자바

이제부터 설명할 예제는 필드와 메소드간의 관계를 명확하게 나타내어준다. 자바에서 다음과 같은 2개의 클래스의 경우를 생각해보자.

```
class Greeting
{
    String intro = "Hello";
```

```
    String target(){
        return "world";
    }
}
class FrenchGreeting extends Greeting
{
    String intro = "Bonjour";
    String target()
    {
        return "le monde";
    }
}
```

이 경우 다음과 같은 프로그램을 실행시키면 어떻게 될까?

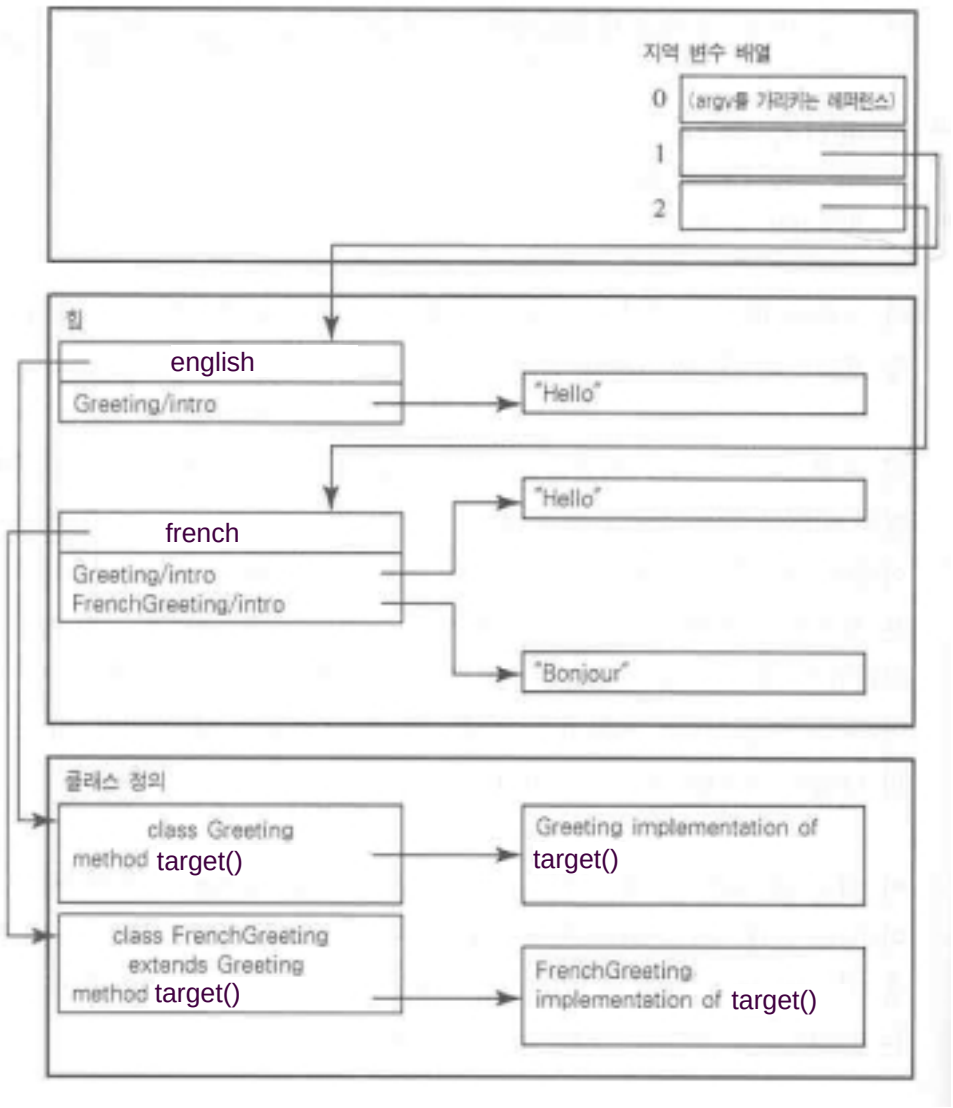
```
public static void main(String argv[]){
    Greeting english = new Greeting();
    Greeting french = new FrenchGreeting();

    System.out.println(english.intro + ", "+ english.target());
    System.out.println(french.intro + ", " + french.target());
    System.out.println(((FrenchGreeting) french).intro + ", "
                                        + ((FrenchGreeting) french).target());
```

이 프로그램의 결과는 다음과 같다.

```
Hello. world
Hello, le monde
Bonjour, le monde
```

다음 그림은 메인 함수가 실행되는 동안 메모리 공간의 상태를 보여준다.



변수 1번은 english이고, 변수 2번은 french이다. 목적필드는 클래스의 이름과 필드의 이름을 함께 사용하여 나타내어진다. 자바 컴파일러는 여기서 french 객체의 필드에 접근하기 위해서 클래스 이름을 Greeting으로 사용한다. 만약 자바 컴파일러에게 클래스 이름을 FrenchGreeting으로 사용하게 하기 위해서는 다음과 같이 해주어야 한다.

((FrenchGreeting) french).intro

이 경우 자바 컴파일러는 french의 변수의 캐스팅된 타입을 체크하고 French.intro필드를 얻어낸다. 이는 두 번째 줄과 세 번째 줄의 앞부분이 왜 서로 다른 결과를 나타내었는지를 설명해준다. 그러나 이것은 뒷부분이 왜 서로 같은 결과를 나타내었는지는 설명해주지 않는다. 뒷부분이 같은 결과를 보여주는 것은 가상 디스패치 알고리즘 때문이다.

두 번째 경우, 이것은 동일한 메소드 이름과 클래스 이름을 사용하지만, 호출되는 메소드는 전혀 다르다. 이것은 가상 디스패치 알고리즘이 호출할 메소드를 결정하기 위해서 인자로 주어진 타입에 의존하지 않고 자신이 실제 타입을 알아내기 때문이다. 위 그림을 보면 지역 변수 2번이 FrenchGreeting 클래

스의 객체를 가리키고 있는 것을 볼 수 있을 것이다. 이 객체에서는 FrenchGreeting 클래스에서 구현된 target 메소드를 가리키고 있으며, 이 메소드는 "world" 문자열이 아니라 "le monde"문자열을 반환한다.

세 번째 경우, 객체는 두 번째 경우와 동일하다. 이것은 객체의 클래스가 객체의 클래스가 동일하다는 이야기이다. 그러므로 target 메소드를 호출하면, 역시 "le monde"문자열을 얻을 수 있을 것이다.

4. 클래스 로더

유닉스 시스템에는 .so파일이 있고, 윈도우 시스템에는 DLL파일이 있다. 이는 모두 동적으로 링크되는 파일인데, JVM의 경우에는 모든 프로그램이 이에 해당한다. 다시 말해, 모든 클래스들은 한 번에 하나의 클래스만을 로드하고서 필요할 때마다 새로운 클래스를 로드해 나간다. 즉, 프로그래머는 모든 프로그램이 동적으로 로드되므로, 별다른 구별없이 프로그램 내의 모든 부분들에 대해서 동일하게 접근할 수 있다.

JVM이 class파일을 동적으로 로드할 수 있는 비법은 바로 java.lang.ClassLoader라는 클래스에 있다. 이번에는 자바 프로그래밍 언어를 이용하여 새로운 클래스 로더를 간단하게 작성하겠다.

1) 클래스의 로딩과정

① 로딩: 클래스의 이름을 사용하여 클래스 파일 형태의 바이트 덩어리를 찾고, 이를 JVM에게 클래스의 구현(implementation)이라고 알리는 작업을 수행한다. 클래스 로더는 수퍼클래스를 로드하고, 마찬가지로 수퍼클래스를 로드하기 위해 수퍼클래스의 수퍼클래스도 로드된다. 로딩 단계에 의해, 가상머신은 클래스의 이름과 클래스 계층도 상에서의 클래스의 우치 및 그 클래스의 필드와 메소드의 종류에 대해 알게된다.

② 링킹 or 레졸루션(resolution): 클래스가 기본적인 형태를 제대로 갖추고 있는지, 가상 머신의 보안과 관련된 제약조건을 거스르지않음을 보장할 수 있도록 클래스를 검증하는 단계. 그리고나서, static 초기화를 수행하는 <clinit>이 호출된다. 이 검증과정의 부가적 영향으로 여러 클래스들이 함께 로드되며, 이 작업이 끝나면 클래스를 사용할 모든 준비는 완료된 것이다.

2) 로딩

모든 클래스 로더는 클래스 java.lang.ClassLoader의 서브 클래스이다. loadClass라고 부르는 메소드에 의해 로드 단계가 시작되는데, 이 메소드는 추상 메소드로 구현되지 않았으며, ClassLoader내부에 존재한다. 그러므로 ClassLoader의 서브클래스는 이 메소드를 구현하여야 한다. 다음은 loadClass의 설명이다.

```
Class loadClass(String name, boolean resolve);
```

여기서 name은 loadClass가 로드할 클래스의 이름을 의미하며, 이름은 패키지이름까지 완전하게 명시되어야한다. resolve는 loadClass가 링킹 단계까지 수행할 것인가를 결정한다.

클래스로더는 name을 사용하여 하나의 클래스로 로드하고자 하는 바이트 덩어리를 찾는다. 이 바이트 덩어리는 디스크나 메모리 내의 배열에도 존재할 수 있으며, 데이터베이스 내부 또는 네트워크를 건너서도 존재할 수 있다. 또 암호화된 채로 존재할 수도 있으며, 표준이 아닌 클래스 파일 포맷으로도 존재할 수 있다. 이러한 형태들은 loadClass에 의하여 가상 머신이 이해할 수 있는 형태로 바뀌진다.

다음으로 클래스 로더는 가상머신에게 이 특정한 바이트들의 집합이 바로 가상 머신이 찾고자 하는 클래스라는 것을 알려주어야 하는데, 이러한 작업은 ClassLoader안의 defineClass라는 메소드에 의해 수행된다. defineClass는 protected로 선언된 메소드이므로, ClassLoader의 서브클래스만이 새로운 클래스를 시스템으로 불러올 수 있다.

메소드 defineClass는 바이트의 배열을 입력받아서 class파일이 적절한 형태를 갖추었는지를 확인하는 약간의 표면적인 검사를 수행한 후에 이 클래스의 수퍼클래스를 로드하기 위하여(링크는 수행하지 않

음) loadClass를 호출한다. loadClass는 수퍼클래스를 로드하기 위해 defineClass를 사용하여 클래스를 정의하며, 이러한 과정은 계속 반복된다.

3) 직접 만드는 클래스 로더

ClassLoader클래스는 추상 클래스이며, loadClass메소드를 구현하지 않았으므로, 인스턴스를 만들 수 없다. 그러므로 ClassLoader의 서브클래스를 만들고 loadClass를 구현함으로써 새로운 클래스 로더를 만들 수 있다. 다음은 자바 1.0 플랫폼에서 클래스로더를 작성하는 템플리트이다.

```
class TemplateClassLoader extends ClassLoader{
        // 클래스들을 저장할 장소를 생성한다.
        Hashtable cache=new Hashtable();

        // 특정한 곳에서 클래스를 로드하기 위해 loadClass를 오버라이드한다.
        protected Class loadClass(String name, boolean resolve)
                                throws ClassNotFoundException{
                // 우선, 캐시에서 이름을 찾는다.
                Class c=(Class) cache.get(name);
                if(c==null){
                        //둘째, 특정한 곳에서 이름을 찾는다.
                        //이름을 찾으면 이를 클래스로 정의하고
                        //그렇지않으면 찾는 이름을 시스템 클래스에서 탐색한다.
                        byte bytes[] = findClass(name);
                        if(bytes == null)
                                c=findSystemClass(name);
                        else    c=defineClass(bytes,0,bytes.length);
                }
                if(resolve) resolveClass(c);
                return c;
        }
        // 주어진 이름의 클래스를 찾는다. 이 부분의 구현은 전적으로 사용자 몫이다.
        // 적절한 클래스를 찾지 못하면 null을 반환한다.
        protected byte[] findClass(String name){
                return null;
        }
}
```

그러면 위의 클래스를 이용하여 자신만의 클래스로더를 작성하는 예제를 들어보겠다.
많은 웹 브라우저들은 자바 애플릿을 로드하기 위한 클래스로더를 내장하고 있다. TemplateClassLoader를 이용하여 작성한 애플릿 로더를 보자.

```
import java.net.URL;
class AppletLoader extends TemplateClassLoader{
```

```
        // 코드를 가져오는 곳
        private URL codebase;
        AppletLoader(URL codebase){
                this.codebase = codebase;
        }
        public byte[] findClass(String name){
                try{
                        URL code_url=new URL(codebase,name+ ".class");
                        URLConnection connection=code_url.openConnection();
                        connection.connect();
                        int length=connection.getContentLength();
                        byte[] bytes=new byte[length];
                        connection.getInputStream().read(bytes);
                        return bytes;
                }
                catch(IOException e){
                        // 뭔가 잘못되었다.
                        return null;
                }
        }
}
```

findClass메소드는 코드베이스에 근거하여 애플릿에 대한 URL을 생성하기 위하여 java.net.URL 클래스를 사용한다. 가령 코드 베이스가 http://appletsource.com/applets/page.html이라면, 메소드는 package/MyTestApplet 애플릿을 다음에서 찾을 수 있을 것이다.
http://appletsource.com/applets/package/MyTestApplet.class
findClass메소드는 애플릿의 전체 텍스트를 읽어 들인 후 바이트의 배열을 반환한다. 그리고나서 MyTestApplet이 package/GreenButton클래스를 가리키는 경우에 시스템은 이 클래스를 로드하기 위하여 같은 클래스 로더를 사용한다.

이것은 사소한 예일 뿐이다. 좀 더 복잡한 브라우저는 자바 압축 포맷(JAR)으로 파일을 로딩함으로써 관련된 클래스들을 한꺼번에 다운로드 할 수도 있다. JAR파일은 많은 클래스들을 포함할 것이므로 클래스 로더는 클래스를 찾기 위해 JAR파일까지 탐색해야 한다.

End of Document