더 자바, “코드를 조작하는 다양한 방법”

이번 강좌는 자바 개발자라면 한 번쯤은 사용해보거나 들어봤을 스프링, 스프링 데이터 JPA, 하이버네이트, 롬복 등의 기반이 되는 자바 기술에 대해 학습합니다.

스프링은 어떻게 @Autowired라는 애노테이션을 사용한 필드 또는 매개변수 타입의 객체를 가져와 주입해 주는 것일까? 롬복은 어떻게 @Data라는 애노테이션을 붙였더니 게터, 세터, hashCode, equals 등의 메소드를 만들어 준 걸까? 궁금하신 적이 있으신가요?

이 강좌는 자바가 제공하는 기술 중에 소스 코드, 바이트 코드 그리고 객체를 조작하는 기술에 대해 학습합니다. 그러려면 우선 JVM의 기본적인 구조와 클래스로더의 동작 방식에 대해 이해하는 것이 좋습니다. 따라서 이번 강좌는 "JVM", "바이트코드 조작", "리플렉션", "다이나믹 프록시 기법" 그리고 "애노테이션 프로세서"에 대해서 학습합니다. 따라서, 자바 기초 학습 이후에 어떤 것을 학습하면 좋을지 고민이었던 분들께 추천합니다.

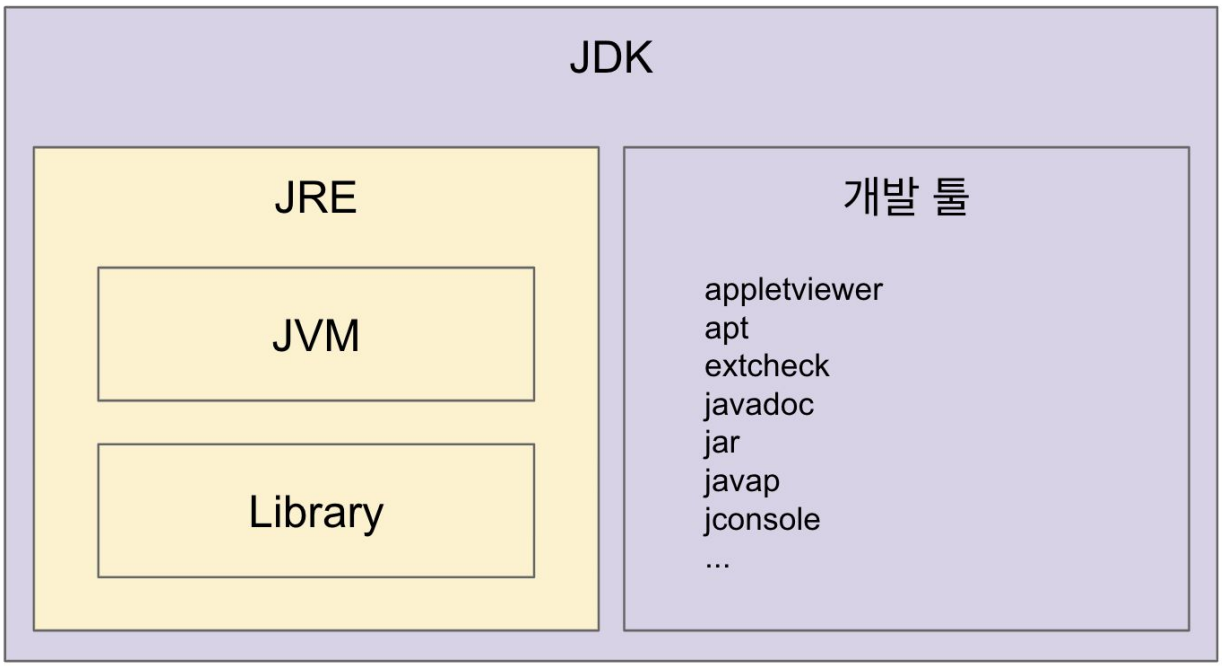
이번 강좌를 학습하고 나면 여러분은 한층 더 자바에 대해 깊이 있는 지식을 습득할 수 있으며 자바를 둘러싼 여러 기술을 학습할 때에도 더 쉽게 이해할 수 있을 것으로 기대합니다. 또한 이 강좌에서 학습한 기술에서 파생해서 GC, 서비스 프로바이더, 프로파일러 등 보다 다양한 자바 기술에도 관심을 두는 계기가 되길 바랍니다.

감사합니다.

1부. JVM 이해하기

1. 자바, JVM, JDK, JRE

목표: 이것들이 뭔지, 뭐가 다른지 이해한다.



JVM (Java Virtual Machine)

- 자바 가상 머신으로 자바 바이트 코드(.class 파일)를 OS에 특화된 코드로 변환(인터프리터와 JIT 컴파일러)하여 실행한다.
- 바이트 코드를 실행하는 표준(JVM 자체는 표준)이자 구현체(특정 밴더가 구현한 JVM)다.
- JVM 스팩: https://docs.oracle.com/javase/specs/jvms/se11/html/
- JVM 밴더: 오라클, 아마존, Azul, ...
- 특정 플랫폼에 종속적.

JRE (Java Runtime Environment): JVM + 라이브러리

- 자바 애플리케이션을 실행할 수 있도록 구성된 배포판.
- JVM과 핵심 라이브러리 및 자바 런타임 환경에서 사용하는 프로퍼티 세팅이나 리소스 파일을 가지고 있다.
- 개발 관련 도구는 포함하지 않는다. (그건 JDK에서 제공)

JDK (Java Development Kit): JRE + 개발 툴

- JRE + 개발에 필요할 툴

- 소스 코드를 작성할 때 사용하는 자바 언어는 플랫폼에 독립적.
- 오라클은 자바 11부터는 JDK만 제공하며 JRE를 따로 제공하지 않는다.
- Write Once Run Anywhere

자바

- 프로그래밍 언어
- JDK에 들어있는 자바 컴파일러(javac)를 사용하여 바이트코드(.class 파일)로 컴파일 할 수 있다.
- 자바 유료화? 오라클에서 만든 Oracle JDK 11 버전부터 상용으로 사용할 때 유료.
    - https://medium.com/@javachampions/java-is-still-free-c02aef8c9e04

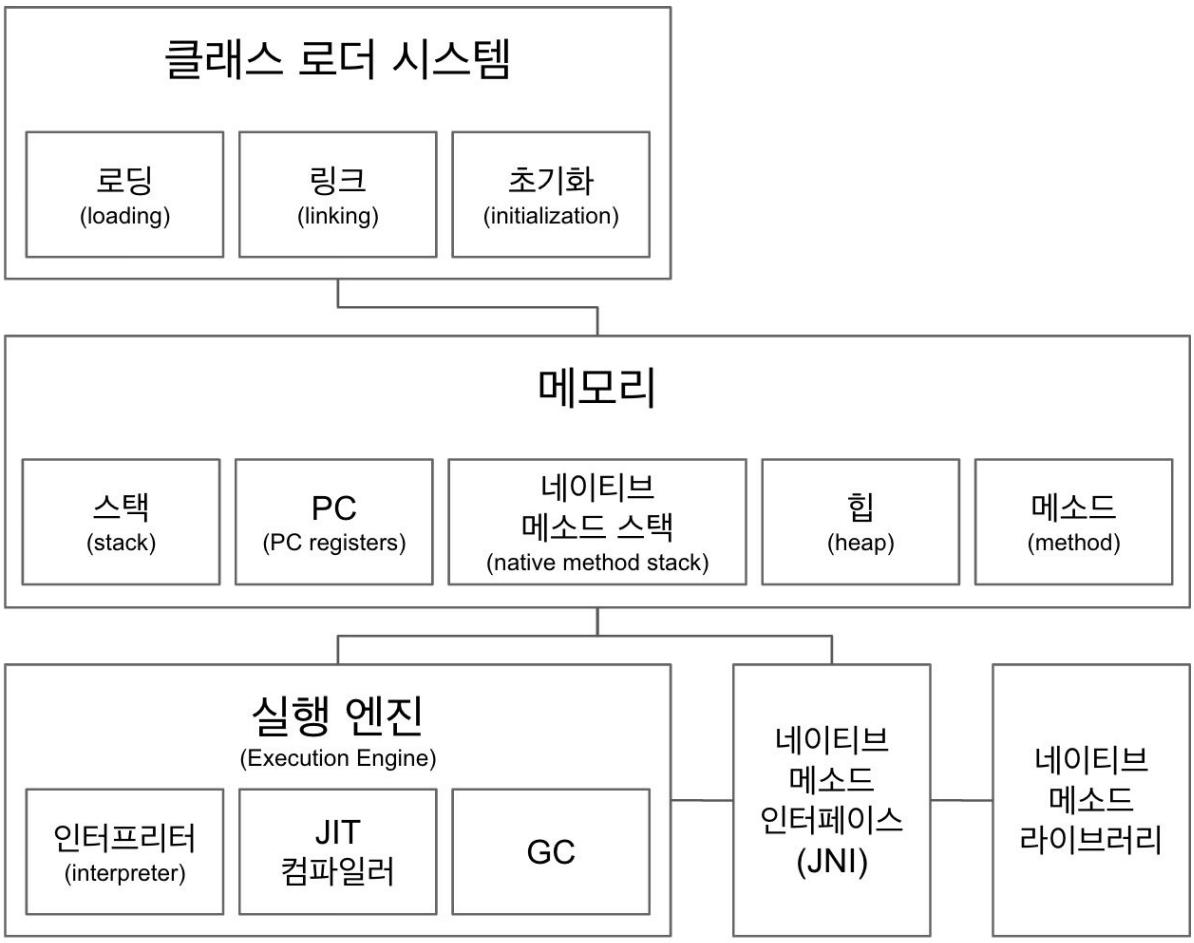
JVM 언어

- JVM 기반으로 동작하는 프로그래밍 언어
- 클로저, 그루비, JRuby, Jython, Kotlin, Scala, ...

참고

- JIT 컴파일러: https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/
- JDK, JRE 그리고 JVM: https://howtodoinjava.com/java/basics/jdk-jre-jvm/
- https://en.wikipedia.org/wiki/List_of_JVM_languages

2. JVM 구조



클래스 로더 시스템

- .class 에서 바이트코드를 읽고 메모리에 저장
- 로딩: 클래스 읽어오는 과정
- 링크: 레퍼런스를 연결하는 과정
- 초기화: static 값들 초기화 및 변수에 할당

메모리

- 메모스 영역에는 클래스 수준의 정보 (클래스 이름, 부모 클래스 이름, 메소드, 변수) 저장. 공유 자원이다.
- 힙 영역에는 객체를 저장. 공유 자원이다.
- 스택 영역에는 쓰레드 마다 런타임 스택을 만들고, 그 안에 메소드 호출을 스택 프레임이라 부르는 블럭으로 쌓는다. 쓰레드 종료하면 런타임 스택도 사라진다.
- PC(Program Counter) 레지스터: 쓰레드 마다 쓰레드 내 현재 실행할 스택 프레임을 가리키는 포인터가 생성된다.
- 네이티브 메소드 스택

- https://javapapers.com/core-java/java-jvm-run-time-data-areas/#Program_Counter_PC_Register

실행 엔진

- 인터프리터: 바이크 코드를 한줄 씩 실행.
- JIT 컴파일러: 인터프리터 효율을 높이기 위해, 인터프리터가 반복되는 코드를 발견하면 JIT 컴파일러로 반복되는 코드를 모두 네이티브 코드로 바꿔둔다. 그 다음부터 인터프리터는 네이티브 코드로 컴파일된 코드를 바로 사용한다.
- GC(Garbage Collector): 더이상 참조되지 않는 객체를 모아서 정리한다.

JNI(Java Native Interface)

- 자바 애플리케이션에서 C, C++, 어셈블리로 작성된 함수를 사용할 수 있는 방법 제공
- Native 키워드를 사용한 메소드 호출
- https://medium.com/@bschlining/a-simple-java-native-interface-jni-example-in-java-and-scala-68fdafe76f5f

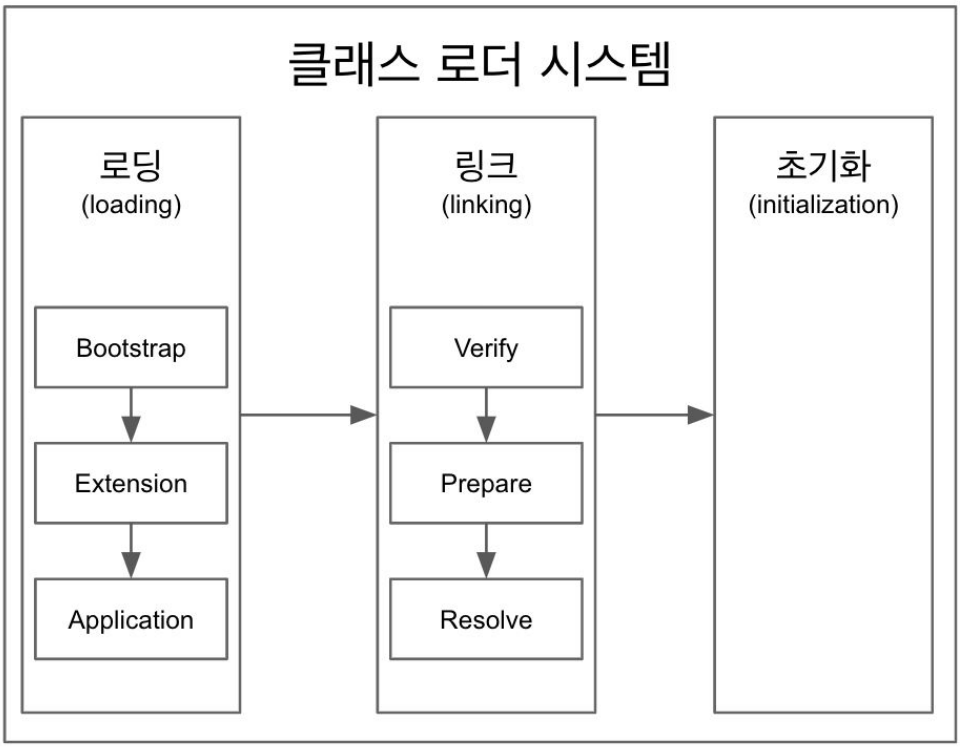
네이티브 메소드 라이브러리

- C, C++로 작성 된 라이브러리

참고

- https://www.geeksforgeeks.org/jvm-works-jvm-architecture/
- https://dzone.com/articles/jvm-architecture-explained
- http://blog.jamesdbloom.com/JVMInternals.html

3. 클래스 로더



클래스 로더

- 로딩, 링크, 초기화 순으로 진행된다.
- 로딩
    - 클래스 로더가 .class 파일을 읽고 그 내용에 따라 적절한 바이너리 데이터를 만들고 “메소드” 영역에 저장.
    - 이때 메소드 영역에 저장하는 데이터
        - FQCN
        - 클래스 | 인터페이스 | 이늄
        - 메소드와 변수
    - 로딩이 끝나면 해당 클래스 타입의 Class 객체를 생성하여 “힙" 영역에 저장.
- 링크
    - Verify, Prepare, Reolve(optional) 세 단계로 나눠져 있다.
    - 검증: .class 파일 형식이 유효한지 체크한다.
    - Preparation: 클래스 변수(static 변수)와 기본값에 필요한 메모리
    - Resolve: 심볼릭 메모리 레퍼런스를 메소드 영역에 있는 실제 레퍼런스로 교체한다.
- 초기화
    - Static 변수의 값을 할당한다. (static 블럭이 있다면 이때 실행된다.)
- 클래스 로더는 계층 구조로 이뤄져 있으면 기본적으로 세가지 클래스 로더가 제공된다.

- 부트 스트랩 클래스 로더 - JAVA_HOME\lib에 있는 코어 자바 API를 제공한다. 최상위 우선순위를 가진 클래스 로더
- 플랫폼 클래스로더 - JAVA_HOME\lib\ext 폴더 또는 java.ext.dirs 시스템 변수에 해당하는 위치에 있는 클래스를 읽는다.
- 애플리케이션 클래스로더 - 애플리케이션 클래스패스(애플리케이션 실행할 때 주는 -classpath 옵션 또는 java.class.path 환경 변수의 값에 해당하는 위치)에서 클래스를 읽는다.

2부. 바이트코드 조작

4. 코드 커버리지는 어떻게 측정할까?

코드 커버리지? 테스트 코드가 확인한 소스 코드를 %

- JaCoCo를 써보자.
- https://www.eclemma.org/jacoco/trunk/doc/index.html
- http://www.semdesigns.com/Company/Publications/TestCoverage.pdf

pom.xml에 플러그인 추가

```
<plugin>
   <groupId>org.jacoco</groupId>
   <artifactId>jacoco-maven-plugin</artifactId>
   <version>0.8.4</version>
   <executions>
      <execution>
         <goals>
            <goal>prepare-agent</goal>
         </goals>
      </execution>
      <execution>
         <id>report</id>
         <phase>prepare-package</phase>
         <goals>
            <goal>report</goal>
         </goals>
      </execution>
   </executions>
</plugin>
```

메이븐 빌드

```
mvn clean verify
```

커버리지 만족 못할시 빌드 실패하도록 설정

```
      <execution>
         <id>jacoco-check</id>
         <goals>
            <goal>check</goal>
         </goals>
         <configuration>
            <rules>
```

```
            <rule>
               <element>PACKAGE</element>
               <limits>
                  <limit>
                     <counter>LINE</counter>
                     <value>COVEREDRATIO</value>
                     <minimum>0.50</minimum>
                  </limit>
               </limits>
            </rule>
         </rules>
      </configuration>
   </execution>
```

5. 모자에서 토끼를 꺼내는 마술

아무것도 없는 Moja에서 “Rabbit”을 꺼내는 마술

Moja.java

```
public class Moja {

    public String pullOut() {
        return "";
    }
}
```

Masulsa.java

```
public class Masulsa {

    public static void main(String[] args) {
        System.out.println(new Moja().pullOut());
    }
}
```

바이트코드 조작 라이브러리

- ASM: https://asm.ow2.io/
- Javassist: https://www.javassist.org/
- ByteBuddy: https://bytebuddy.net/#/

6. javaagent 실습

Javaagent JAR 파일 만들기

- https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html
- 붙이는 방식은 시작시 붙이는 방식 premain과 런타임 중에 동적으로 붙이는 방식 agentmain이 있다.
- Instrumentation을 사용한다.

Javaagent 붙여서 사용하기

- 클래스로더가 클래스를 읽어올 때 javaagent를 거쳐서 변경된 바이트코드를 읽어들여 사용한다.

MasulsaJavaAgent.java

```
public class MasulsaAgent {

    public static void premain(String agentArgs, Instrumentation inst) {
        new AgentBuilder.Default()
                .type(ElementMatchers.any())
                .transform((builder, typeDescription, classLoader, javaModule) ->
builder.method(named("pullOut")).intercept(FixedValue.value("Rabbit!"))).installOn(inst);
    }

}
```

pom.xml

```
  <dependencies>
    <dependency>
      <groupId>net.bytebuddy</groupId>
      <artifactId>byte-buddy</artifactId>
      <version>1.10.1</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.1.2</version>
        <configuration>
          <archive>
```

```
        <index>true</index>
        <manifest>
          <addClasspath>true</addClasspath>
        </manifest>
        <manifestEntries>
          <mode>development</mode>
          <url>${project.url}</url>
          <key>value</key>
          <Premain-Class>me.whiteship.MasulsaAgent</Premain-Class>
          <Can-Redefine-Classes>true</Can-Redefine-Classes>
          <Can-Retransform-Classes>true</Can-Retransform-Classes>
        </manifestEntries>
      </archive>
    </configuration>
  </plugin>
 </plugins>
</build>
```

Javaagent 적용 (경로는 각자 환경에 맞춰 변경하세요.)

```
-javaagent:/Users/keesun/workspace/MasulsaJavaAgent/target/MasulsaAgent-1.0-SNAPSHOT.jar
```

7. 바이트코드 조작 정리

프로그램 분석
- 코드에서 버그 찾는 툴
- 코드 복잡도 계산

클래스 파일 생성
- 프록시
- 특정 API 호출 접근 제한
- 스칼라 같은 언어의 컴파일러

그밖에도 자바 소스 코드 건리지 않고 코드 변경이 필요한 여러 경우에 사용할 수 있다.
- 프로파일러 (newrelic)
- 최적화
- 로깅
- ...

스프링이 컴포넌트 스캔을 하는 방법 (asm)
- 컴포넌트 스캔으로 빈으로 등록할 후보 클래스 정보를 찾는데 사용
- ClassPathScanningCandidateComponentProvider -> SimpleMetadataReader
- ClassReader와 Visitor 사용해서 클래스에 있는 메타 정보를 읽어온다.

참고
- https://www.youtube.com/watch?v=39kdr1mNZ_s
- ASM, Javassist, ByteBuddy, CGlib

4부. 리플렉션

8. 스프링의 Depedency Injection은 어떻게 동작할까?

BookService.java

```
@Service
public class BookService {

   @Autowired
   BookRepository bookRepository;

}
```

- bookRepository 인스턴스는 어떻게 null이 아닌걸까?
- 스프링은 어떻게 BookService 인스턴스에 BookRepository 인스턴스를 넣어준 것일까?

9. 리플렉션 API 1부: 클래스 정보 조회

리플렉션의 시작은 Class<T>

- https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html

Class<T>에 접근하는 방법

- 모든 클래스를 로딩 한 다음 Class<T>의 인스턴스가 생긴다. “타입.class”로 접근할 수 있다.
- 모든 인스턴스는 getClass() 메소드를 가지고 있다. “인스턴스.getClass()”로 접근할 수 있다.
- 클래스를 문자열로 읽어오는 방법
    - Class.forName(“FQCN”)
    - 클래스패스에 해당 클래스가 없다면 ClassNotFoundException이 발생한다.

Class<T>를 통해 할 수 있는 것

- 필드 (목록) 가져오기
- 메소드 (목록) 가져오기
- 상위 클래스 가져오기
- 인터페이스 (목록) 가져오기
- 애노테이션 가져오기
- 생성자 가져오기
- ...

10. 애노테이션과 리플렉션

중요 애노테이션

- @Retention: 해당 애노테이션을 언제까지 유지할 것인가? 소스, 클래스, 런타임
- @Inherit: 해당 애노테이션을 하위 클래스까지 전달할 것인가?
- @Target: 어디에 사용할 수 있는가?

리플렉션

- getAnnotations(): 상속받은 (@Inherit) 애노테이션까지 조회
- getDeclaredAnnotations(): 자기 자신에만 붙어있는 애노테이션 조회

11. 리플렉션 API 1부: 클래스 정보 수정 또는 실행

Class 인스턴스 만들기

- Class.newInstance()는 deprecated 됐으며 이제부터는
- 생성자를 통해서 만들어야 한다.

생성자로 인스턴스 만들기

- Constructor.newInstance(params)

필드 값 접근하기/설정하기

- 특정 인스턴스가 가지고 있는 값을 가져오는 것이기 때문에 인스턴스가 필요하다.
- Field.get(object)
- Filed.set(object, value)
- Static 필드를 가져올 때는 object가 없어도 되니까 null을 넘기면 된다.

메소드 실행하기

- Object Method.invoke(object, params)

12. 나만의 DI 프레임워크 만들기

@Inject 라는 애노테이션 만들어서 필드 주입 해주는 컨테이너 서비스 만들기

```
public class BookService {

    @Inject
    BookRepository bookRepository;

}
```

ContainerService.java

```
public static <T> T getObject(T classType)
```

- classType에 해당하는 타입의 객체를 만들어 준다.
- 단, 해당 객체의 필드 중에 @Inject가 있다면 해당 필드도 같이 만들어 제공한다.

13. 리플렉션 정리

리플렉션 사용시 주의할 것

- 지나친 사용은 성능 이슈를 야기할 수 있다. 반드시 필요한 경우에만 사용할 것
- 컴파일 타임에 확인되지 않고 런타임 시에만 발생하는 문제를 만들 가능성이 있다.
- 접근 지시자를 무시할 수 있다.

스프링

- 의존성 주입
- MVC 뷰에서 넘어온 데이터를 객체에 바인딩 할 때

하이버네이트

- @Entity 클래스에 Setter가 없다면 리플렉션을 사용한다.

JUnit

- https://junit.org/junit5/docs/5.0.3/api/org/junit/platform/commons/util/ReflectionUtils.html

참고

- https://docs.oracle.com/javase/tutorial/reflect/index.html

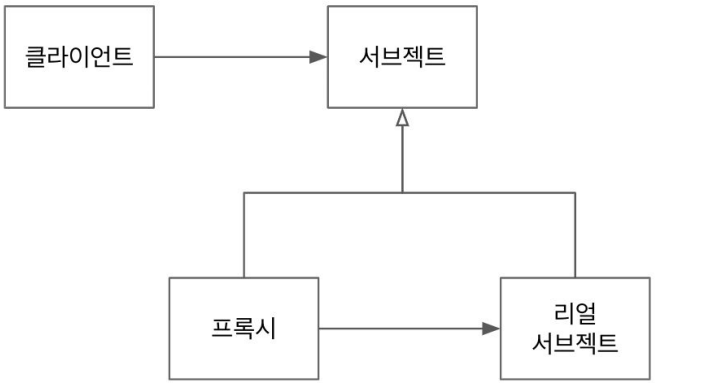
4부. 다이나믹 프록시

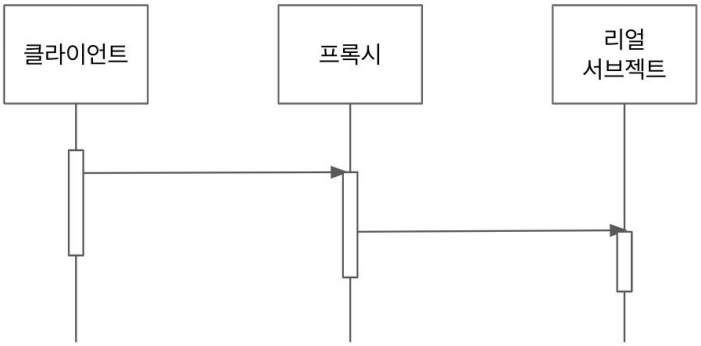
14. 스프링 데이터 JPA는 어떻게 동작하나?

스프링 데이터 JPA에서 인터페이스 타입의 인스턴스는 누가 만들어 주는것인가?

- Spring AOP를 기반으로 동작하며 RepositoryFactorySupport에서 프록시를 생성한다.

15. 프록시 패턴





- 프록시와 리얼 서브젝트가 공유하는 인터페이스가 있고, 클라이언트는 해당 인터페이스 타입으로 프록시를 사용한다.
- 클라이언트는 프록시를 거쳐서 리얼 서브젝트를 사용하기 때문에 프록시는 리얼 서브젝트에 대한 접근을 관리거나 부가기능을 제공하거나, 리턴값을 변경할 수도 있다.
- 리얼 서브젠트는 자신이 해야 할 일만 하면서(SRP) 프록시를 사용해서 부가적인 기능(접근 제한, 로깅, 트랜잭션, 등)을 제공할 때 이런 패턴을 주로 사용한다.

참고

- https://www.oodesign.com/proxy-pattern.html
- https://en.wikipedia.org/wiki/Proxy_pattern
- https://en.wikipedia.org/wiki/Single_responsibility_principle

16. 다이나믹 프록시 실습

런타임에 특정 인터페이스들을 구현하는 클래스 또는 인스턴스를 만드는 기술

“an application can use a dynamic proxy class to create an object that implements multiple arbitrary event listener interfaces”

- https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html

프록시 인스턴스 만들기

- Object Proxy.newProxyInstance(ClassLoader, Interfaces, InvocationHandler)

```
BookService bookService = (BookService) Proxy.newProxyInstance(BookService.class.getClassLoader(), new
Class[]{BookService.class},
        new InvocationHandler() {
            BookService bookService = new DefaultBookService();
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                if (method.getName().equals("rent")) {
                    System.out.println("aaaa");
                    Object invoke = method.invoke(bookService, args);
                    System.out.println("bbbb");
                    return invoke;
                }

                return method.invoke(bookService, args);
            }
        });
```

- 유연한 구조가 아니다. 그래서 스프링 AOP 등장!
- 스프링 AOP에 대한 더 자세한 토비의 스프링 3.1, 6장 AOP를 참고하세요.

참고

- https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html#newProxyInstance-java.lang.ClassLoader-java.lang.Class:A-java.lang.reflect.InvocationHandler-

17. 클래스의 프록시가 필요하다면?

서브 클래스를 만들 수 있는 라이브러리를 사용하여 프록시를 만들 수 있다.

CGlib

- https://github.com/cglib/cglib/wiki
- 스프링, 하이버네이트가 사용하는 라이브러리
- 버전 호환성이 좋치 않아서 서로 다른 라이브러리 내부에 내장된 형태로 제공되기도 한다.

```
    <dependency>
      <groupId>cglib</groupId>
      <artifactId>cglib</artifactId>
      <version>3.3.0</version>
    </dependency>
```

```
MethodInterceptor handler = new MethodInterceptor() {
   BookService bookService = new BookService();
   @Override
   public Object intercept(Object o, Method method, Object[] objects, MethodProxy
methodProxy) throws Throwable {
       return method.invoke(bookService, objects);
   }
};

BookService bookService = (BookService) Enhancer.create(BookService.class, handler);
```

ByteBuddy

- https://bytebuddy.net/#/
- 바이트 코드 조작 뿐 아니라 런타임(다이나믹) 프록시를 만들 때도 사용할 수 있다.

```

```

서브 클래스를 만드는 방법의 단점

- 상속을 사용하지 못하는 경우 프록시를 만들 수 없다.
    - Private 생성자만 있는 경우
    - Final 클래스인 경우
- 인터페이스가 있을 때는 인터페이스의 프록시를 만들어 사용할 것.

18. 다이나믹 프록시 정리

다이나믹 프록시

- 런타임에 인터페이스 또는 클래스의 프록시 인스턴스 또는 클래스를 만들어 사용하는 프로그래밍 기법

다이나믹 프록시 사용처

- 스프링 데이터 JPA
- 스프링 AOP
- Mockito
- 하이버네이트 lazy initialzation
- ...

참고

- http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html

5부. 애노테이션 프로세서

19. Lombok(롬복)은 어떻게 동작하는 걸까?

Lombok

- @Getter, @Setter, @Builder 등의 애노테이션과 애노테이션 프로세서를 제공하여 표준적으로 작성해야 할 코드를 개발자 대신 생성해주는 라이브러리.

롬복 사용하기

- 의존성 추가

```
<dependency>
   <groupId>org.projectlombok</groupId>
   <artifactId>lombok</artifactId>
   <version>1.18.8</version>
   <scope>provided</scope>
</dependency>
```

- IntelliJ lombok 플러그인 설치
- IntelliJ Annotation Processing 옵션 활성화

롬복 동작 원리

- 컴파일 시점에 애노테이션 프로세서를 사용하여 소스코드의 AST(abstract syntax tree)를 조작한다.

논란 거리

- 공개된 API가 아닌 컴파일러 내부 클래스를 사용하여 기존 소스 코드를 조작한다.
- 특히 이클립스의 경우엔 java agent를 사용하여 컴파일러 클래스까지 조작하여 사용한다. 해당 클래스들 역시 공개된 API가 아니다보니 버전 호환성에 문제가 생길 수 있고 언제라도 그런 문제가 발생해도 이상하지 않다.
- 그럼에도 불구하고 엄청난 편리함 때문에 널리 쓰이고 있으며 대안이 몇가지 있지만 롬복의 모든 기능과 편의성을 대체하진 못하는 현실이다.
    - AutoValue
        - https://github.com/google/auto/blob/master/value/userguide/index.md
    - Immutables
        - https://immutables.github.io

참고

- https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/Processor.html
- https://projectlombok.org/contributing/lombok-execution-path
- https://stackoverflow.com/questions/36563807/can-i-add-a-method-to-a-class-from-a-compile-time-annotation

- http://jnb.ociweb.com/jnb/jnbJan2010.html#controversy
- https://www.oracle.com/technetwork/articles/grid/java-5-features-083037.html

20. 애노테이션 프로세서 1부

Processor 인터페이스

- 여러 라운드(rounds)에 거쳐 소스 및 컴파일 된 코드를 처리 할 수 있다.

유틸리티

- AutoService: 서비스 프로바이더 레지스트리 생성기

```
<dependency>
  <groupId>com.google.auto.service</groupId>
  <artifactId>auto-service</artifactId>
  <version>1.0-rc6</version>
</dependency>
```

```
@AutoService(Processor.class)
public class MagicMojaProcessor extends AbstractProcessor {
...
}
```

- 컴파일 시점에 애노테이션 프로세서를 사용하여 META-INF/services/javax.annotation.processor.Processor 파일 자동으로 생성해 줌.

ServiceProvider

- https://itnext.io/java-service-provider-interface-understanding-it-via-code-30e1dd45a091

참고

- http://hannesdorfmann.com/annotation-processing/annotationprocessing101
- http://notatube.blogspot.com/2010/12/project-lombok-creating-custom.html
- https://medium.com/@jintin/annotation-processing-in-java-3621cb05343a
- https://medium.com/@iammert/annotation-processing-dont-repeat-yourself-generate-your-code-8425e60c6657
- https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html#processing

21. 애노테이션 프로세서 2부

Filer 인터페이스

- 소스 코드, 클래스 코드 및 리소스를 생성할 수 있는 인터페이스

```
@Override
public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
   Set<? extends Element> elements = roundEnv.getElementsAnnotatedWith(Magic.class);
   for (Element element : elements) {
      Name elementName = element.getSimpleName();
      if (element.getKind() != ElementKind.INTERFACE) {
         processingEnv.getMessager().printMessage(Diagnostic.Kind.ERROR, "Magic
annotation can not be used on " + elementName);
      } else {
         processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE, "Processing " +
elementName);
      }

      TypeElement typeElement = (TypeElement)element;
      ClassName className = ClassName.get(typeElement);

      MethodSpec pullOut = MethodSpec.methodBuilder("pullOut")
            .addModifiers(Modifier.PUBLIC)
            .returns(String.class)
            .addStatement("return $S", "Rabbit!")
            .build();

      TypeSpec magicMoja = TypeSpec.classBuilder("MagicMoja")
            .addModifiers(Modifier.PUBLIC)
            .addSuperinterface(className)
            .addMethod(pullOut)
            .build();

      Filer filer = processingEnv.getFiler();
      try {
         JavaFile.builder(className.packageName(), magicMoja)
               .build()
               .writeTo(filer);
      } catch (IOException e) {
         processingEnv.getMessager().printMessage(Diagnostic.Kind.ERROR, "FATAL
ERROR: " + e);
      }
   }
   return true;
```

```
}
```

유틸리티

- Javapoet: 소스 코드 생성 유틸리티

22. 애노테이션 프로세서 정리

애노테이션 프로세서 사용 예

- 롬복
- AutoService: java.util.ServiceLoader용 파일 생성 유틸리티
- @Override
    - https://stackoverflow.com/questions/18189980/how-do-annotations-like-override-work-internally-in-java/18202623
- Dagger 2: 컴파일 타임 DI 제공
- 안드로이드 라이브러리
    - ButterKinfe: @BindView (뷰 아이디와 애노테이션 붙인 필드 바인딩)
    - DeepLinkDispatch: 특정 URI 링크를 Activity로 연결할 때 사용

애노테이션 프로세서 장점

- 런타임 비용이 제로

애노테이션 프로세서 단점

- 기존 클래스 코드를 변경할 때는 약간의 hack이 필요하다.

23. 마무리

이번 강의에서 다룬 내용

- JVM 구조
- 바이트 코드 조작 - ASM 또는 Javassist, **ByteBuddy**
- 리플렉션 API - 클래스 정보 참조 (메소드, 필드, 생성자, ...)
- 다이나믹 프록시 기법 - Proxy, CGlib, **ByteBuddy**
- 애노테이션 프로세서 - Abstract**Processor**, **Filer, ..., AutoService, Javapoet**

이제는 여러분의 상상력에 달렸습니다.

감사합니다.