SpringFramework

1. 스프링 개요

1.1 프로젝트 생성 및 환경설정

- 실습용 프로젝트 : springclass-ch01
- 완성 프로젝트 : springstudy-ch01

SpringSTS(Spring Tool Suite)는 자바 통합개발환경(IDE, Integrated Development Environment)을 제공하는 Eclipse를 Spring 개발에 맞게 수정하여 배포하는 SpringFramework의 통합개발환경으로 “http://spring.io/tools”에 접속하면 다운로드 받을 수 있다. 참고로 이 사이트에 접속하면 Eclipse를 포함해서 또 다른 IDE를 기반으로 하는 SpringTools도 다운로드 받을 수 있다.
SpringSTS 설치 방법은 Eclipse와 마찬가지로 다운로드 받은 파일의 압축을 풀어 설치할 수 있으므로 생략하겠다.
실습 프로젝트를 생성하기 전에 시스템에 설치된 SpringSTS의 기본 환경부터 설정해 보자.
기본 환경설정은 Eclipse와 마찬가지로 SpringSTS의 Window -> Preferences 메뉴를 선택하여 그림 1-1과 같은 Preferences 대화상자에서 설정하게 된다.

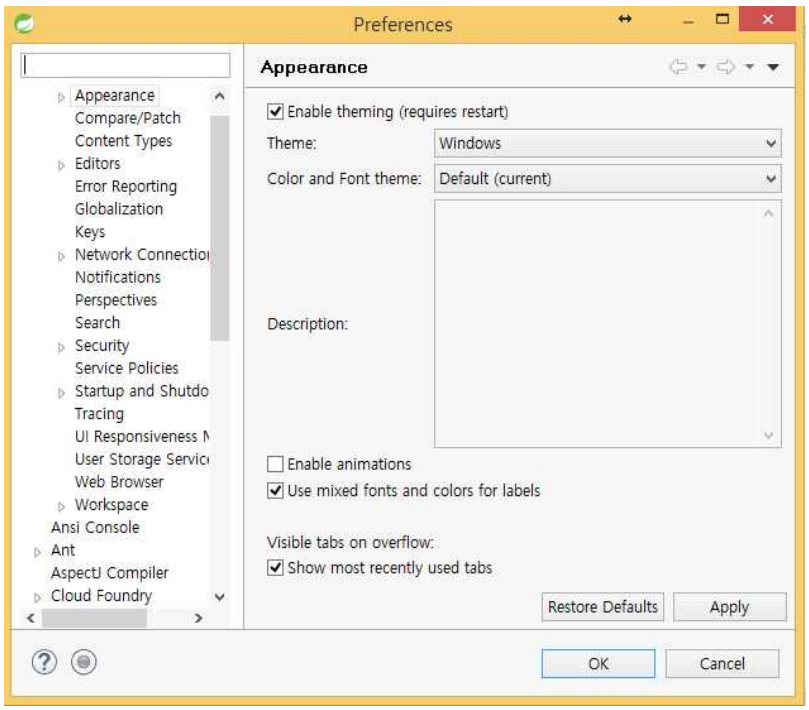

그림 1-1 SpringSTS의 Window->Preferences 메뉴 화면

이 Preferences 화면 좌측 메뉴의 General -> Appearance -> Colors and Fonts를 선택해 본인 취향에 맞는 글꼴과 화면 색상을 선택할 수 있으니 본인이 선호하는 글꼴과 화면 색상 등을 지정해 보자. 그리고 General -> WebBrowser 메뉴를 선택해 Use external web browser를 선택하고 Internet Explorer나 Chrome 브라우저 중에 본인이 선호하는 브라우저를 선택하고 General -> Workspace 메뉴를 선택해 Text file encoding 항목을 other로 선택하고 UTF-8로 파일 인코딩을

설정하자.

국내에서는 SpringFramework를 이용한 개발에서 애플리케이션을 빌드하고 관리할 때 Maven을 사용하는 경우가 많다. 우리도 이 Maven을 사용해 프로젝트에서 의존하는 라이브러리 의존성을 관리할 것이다. 그림 1-2와 같이 Maven 메뉴를 선택하고 우측의 설정 화면에서 그림과 같이 두 가지 항목을 체크해 설정하자.

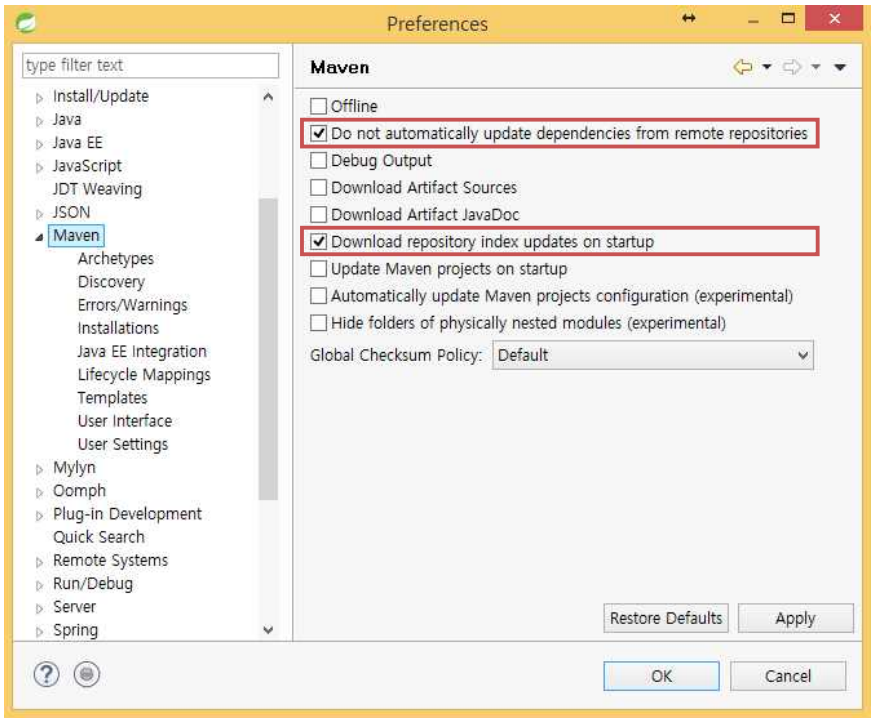

그림 1-2 Maven Repository Index Update 설정

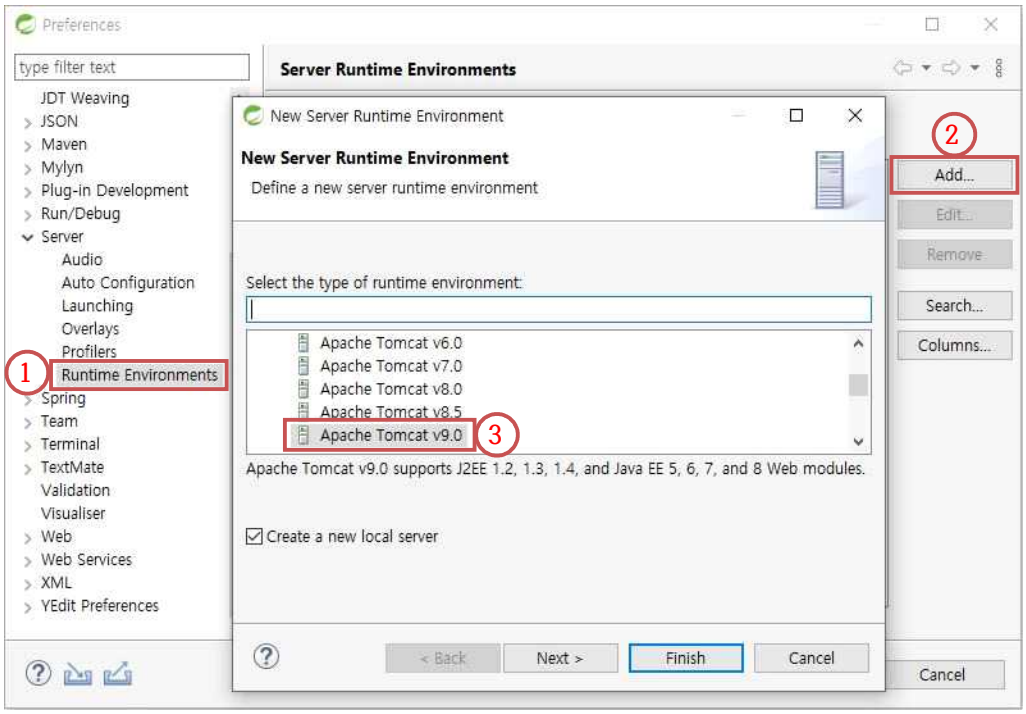

그림 1-3 Server Runtime 설정

SpringWebMVC 프로젝트를 실행할 때 JSP와 마찬가지로 Tomcat 서버를 사용할 것이므로 그림 1-3과 같이 Server -> Runtime Environments 메뉴를 선택하고 Add 메뉴를 통해 Tomcat 9.0 버전을 추가하자. 그리고 그림 1-4와 같이 Web 메뉴를 선택해 Web 관련 파일(CSS, HTML, JSP)의 인코딩을 "ISO 10646/Unicode(UTF-8)"로 설정하자.

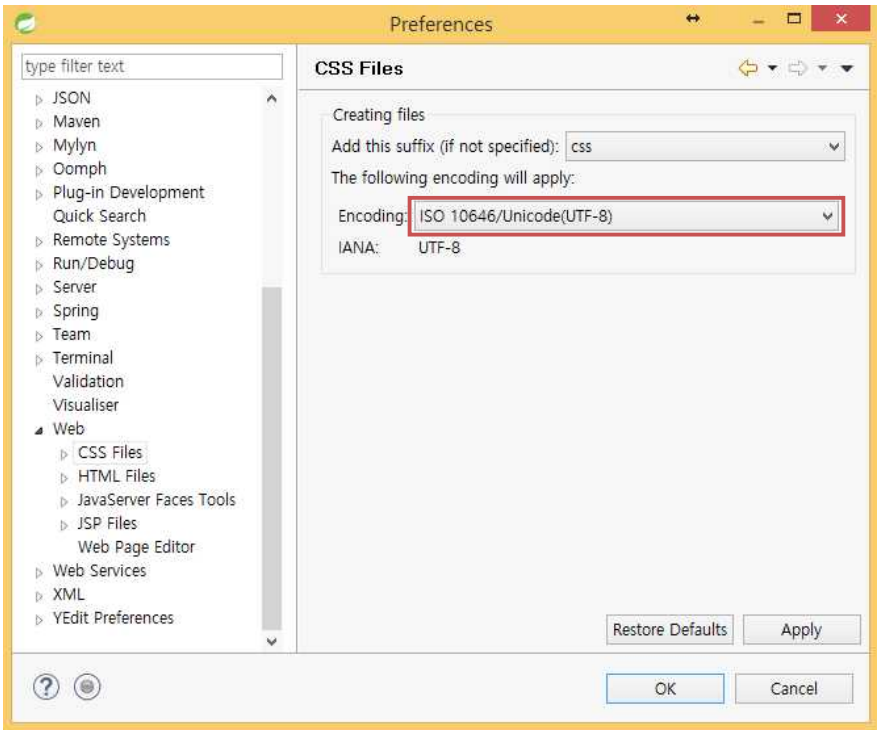

그림 1-4 Web관련 파일 Encoding설정

이제 기본적인 설정은 끝났으니 SpringSTS를 통해 우리의 첫 번째 프로젝트를 생성해 보자.

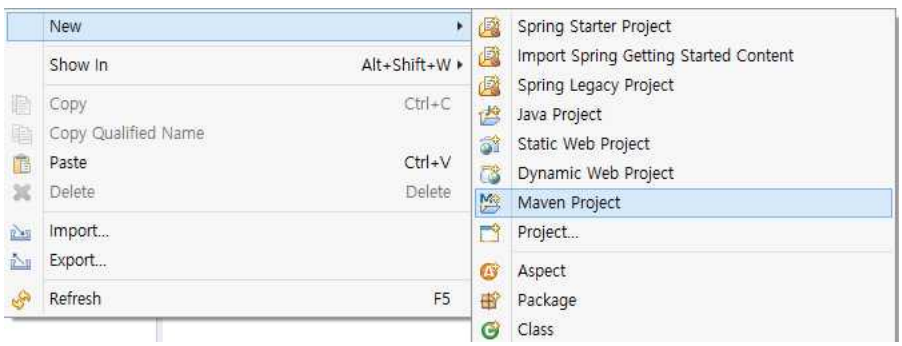

그림 1-5 Maven 프로젝트 생성하기 01

먼저 Package Explorer의 빈 공간에 마우스 우 클릭 하면 그림 1-5와 같은 컨텍스트 메뉴가 나타나는데 이 메뉴에서 Maven Project를 선택하면 그림 1-6과 같이 New Maven Project 대화상자가 화면에 나타난다. 이 대화상자에서 그림과 같이 두 가지 항목을 체크하고 Next 버튼을 클릭한다.

그림 1-7화면에서와 같이 Group Id와 Artifact Id를 입력하고 Packaging을 jar로 선택한다.
이번 예제는 강한 결합과 약한 결합에 대한 개념과 스프링을 사용해 클래스 간의 의존성을 관리하는 가장 기본적인 방법에 대해 학습할 것이므로 Web Project가 아닌 Java Project를 생성할 것이다. 그래서 packaging을 jar로 선택하였다.

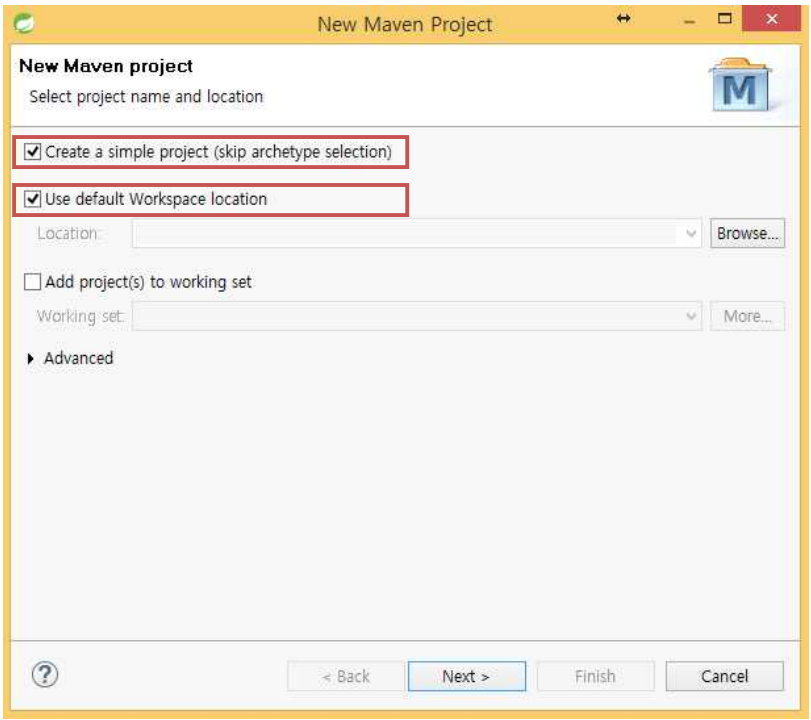

그림 1-6 Maven 프로젝트 생성하기 02

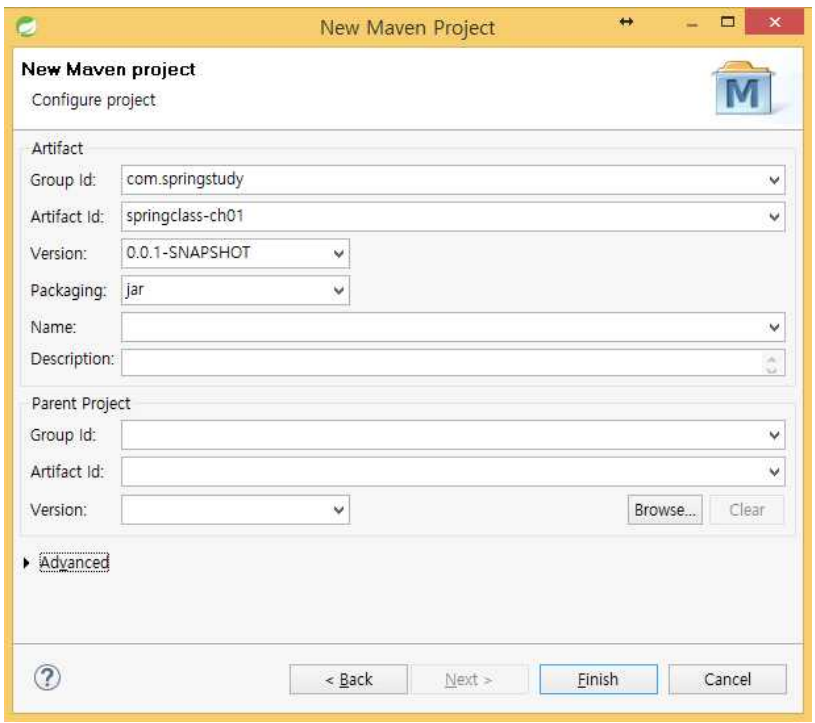

그림 1-7 Maven 프로젝트 생성하기 03

Group Id와 Artifact Id는 메이븐 빌드 툴에서 프로젝트를 유일하게 구분하는 용도로 사용되는 고유한 이름으로 프로젝트를 생성하는 개발자가 임의로 지정하면 되지만 일반적으로 Group Id는 프로젝트를 대표하는 도메인 명을 사용해 지정하기 때문에 그림 1-7과 같은 형식으로 프로젝트 도메인 명을 거꾸로 지정하면 되고 Artifact Id는 프로젝트 산출물의 이름으로 프로젝트 이름이 되기 때문에 그림 1-7과 같이 지정하면 된다. 이렇게 생성되는 Maven Project는 기본적으로 자바 1.5 버전으로 생성되기 때문에 우리가 사용할 자바 1.8 버전으로 수정해야 한다.

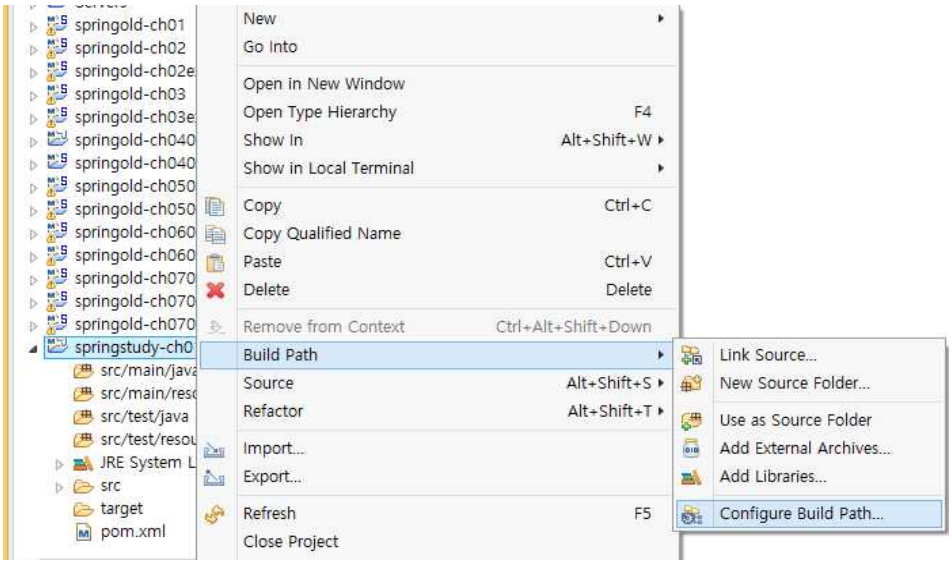

그림 1-8 Maven 프로젝트 생성하기 04

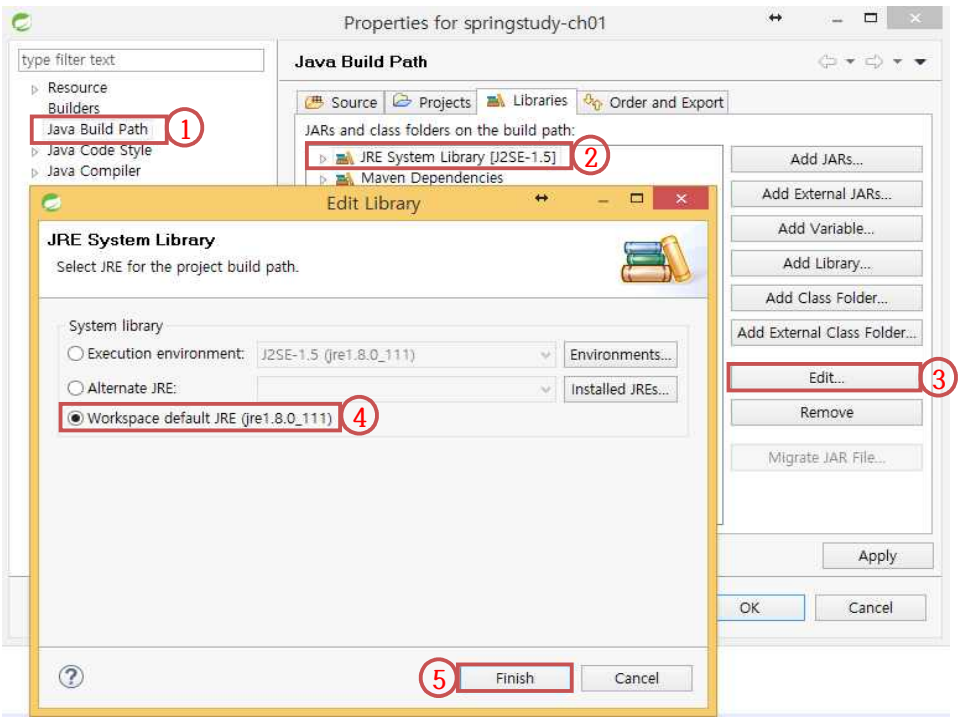

그림 1-9 Maven 프로젝트 생성하기 05

위에서 생성한 프로젝트에 마우스 우 클릭 하여 그림 1-8과 같이 Build Path -> Configure Build Path... 메뉴를 선택하면 그림 1-9와 같은 대화상자가 나타나는데 이 화면에서 그림과 같은 순서로 프로젝트의 Build Path를 설정하면 된다.
Maven 프로젝트를 만들면 Java Compiler도 1.5 버전으로 기본 설정되기 때문에 프로젝트에서 사용할 JRE System의 버전을 1.8로 변경했다면 Java Compiler도 동일한 버전으로 변경해야 한다. 그러므로 아래 그림 1-10의 ②에서 기본으로 설정된 버전 1.5를 1.8로 변경하면 된다.

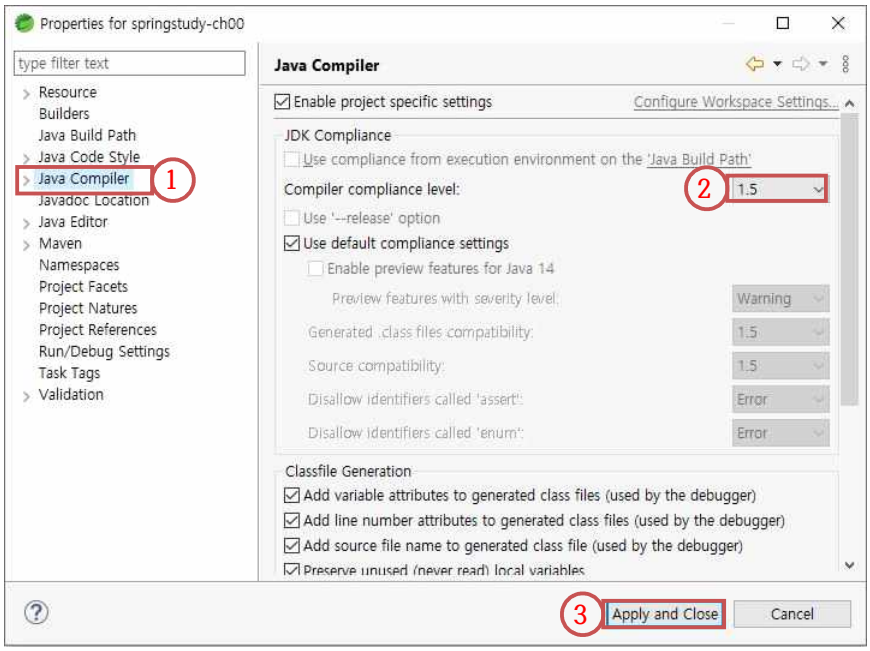

그림 1-10 Maven 프로젝트 생성하기 06

이제 프로젝트의 기본적인 설정은 끝났다. 우리는 SpringFramework를 활용해 애플리케이션을 개발할 것이므로 SpringFramework가 제공하는 기능을 사용하기 위해서는 각각의 기능을 제공하는 SpringFramework의 여러 모듈이 필요하다. 이전에 Maven을 사용하지 않는 프로젝트에서는 필요한 기능을 제공하는 모듈(라이브러리)을 체크하여 일일이 다운로드 받아 프로젝트에서 참조할 수 있도록 Build Path를 설정해야 했지만 우리는 Maven을 사용해 애플리케이션 빌드에 필요한 모듈을 관리할 것이므로 Maven이 애플리케이션 빌드에 필요한 모듈을 관리할 수 있도록 설정해야 한다.
Maven 프로젝트의 설정 파일은 프로젝트 루트에 생성되는 pom.xml 파일이다. 이 pom.xml 파일에서 프로젝트 빌드에 필요한 모든 설정이 이루어진다.
Maven을 사용해 프로젝트 빌드에 필요한 라이브러리 의존성을 설정하는 방법을 알아보기 전에 SpringFramework에서 제공하는 모듈간의 의존관계를 알아볼 것이다. SpringFramework 모듈간의 의존관계는 우리가 스프링을 사용해 애플리케이션을 개발할 때 매우 중요한 개념이다. 왜냐하면 우리가 개발하는 애플리케이션은 스프링 모듈을 사용하기 때문에 스프링 모듈에 의존하게 된다. 그리고 스프링 모듈도 실행하는데 필요한 다른 스프링 모듈이 필요하게 되는데 이렇게 하나의 모듈이 실행하는데 필요한 기능을 제공하는 다른 모듈이 필요할 때 두 모듈 간에는 의존관계가 존재한다.
A라는 모듈이 실행하는데 필요한 기능을 제공하는 B라는 모듈이 있다고 가정하자. 모듈 A가 실행될 때 모듈 B의 기능을 필요하기 때문에 모듈 A는 모듈 B의 기능에 의존하게 된다. 그러므로 두 모듈 간에는 의존관계가 존재하고 모듈 A가 실행될 때 모듈 B가 먼저 메모리에 존재해야 메모리에 있는 모듈 B를 참조해(의존해) 모듈 A가 정상적으로 실행될 수 있다.
그림 1-11은 SpringFramework 모듈 구성을 나타낸 것이다. 이 그림에서 스프링이 태어날 때부터

SpringFramework가 제공하는 가장 핵심적인 기능은 CoreContainer 안에 있다.

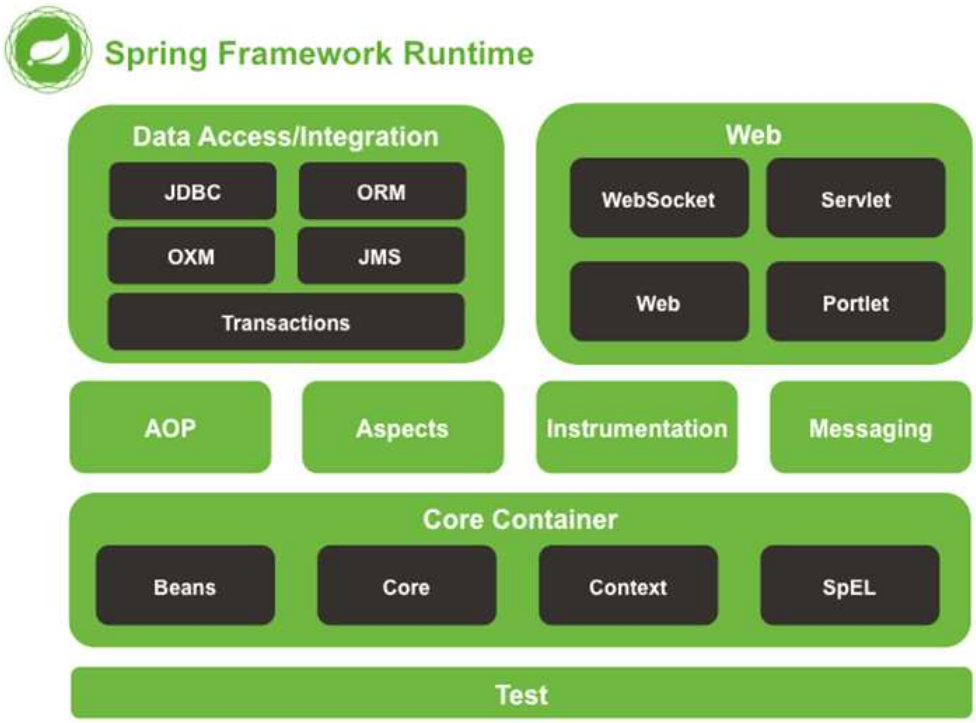

그림 1-11 SpringFramework 모듈

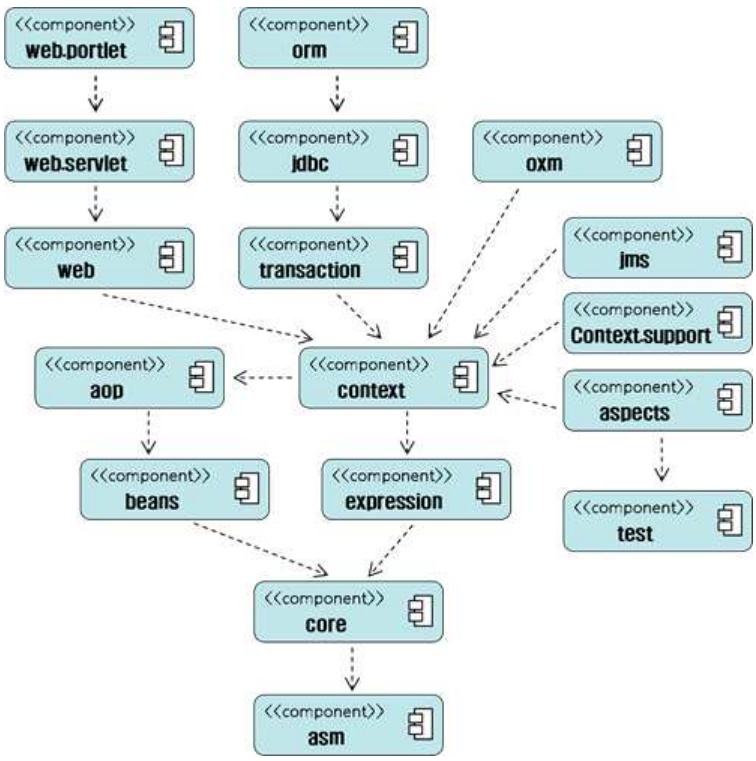

그림 1-12 SpringFramework 모듈의 의존관계

위의 그림 1-12는 SpringFramework 모듈 간의 의존관계를 그림으로 표현한 것이다.

그림 1-12을 살펴보면 화살표를 통해 모듈간의 의존관계를 표현하고 있다. 이 그림의 중심에 contetxt라는 모듈이 보이는데 이 모듈은 그림 1-11에서 Spring Core가 지원하는 기능 외에 추가적인 기능들과 조금 더 쉽고 편리한 개발을 지원하기 위해 스프링 기반에서 구현된 Bean 객체에 대한 접근 방법을 제공하는 모듈이다. 스프링에서 Bean 이라는 용어를 자주사용하게 되는데 이 Bean의 의미는 우리가 일반적으로 사용하는 자바 객체를 의미한다. 조금 더 자세히 말하자면 어떤 프레임워크의 인터페이스도 강제로 상속받지 않은 순순한 자바 객체(Plan Old Java Object)를 일컫는 말이다.
Maven을 통해 스프링 모듈의 대한 의존성을 설정할 때 이 context 모듈을 먼저 설정하게 되면 Maven은 context 모듈이 의존하고 있는 apo 모듈과 expression 모듈을 자동으로 Build Path에 추가해 준다. 그리고 aop, expression 모듈이 의존하고 있는 나머지 모듈도 모두 Maven에 의해 자동으로 Build Path에 추가된다.

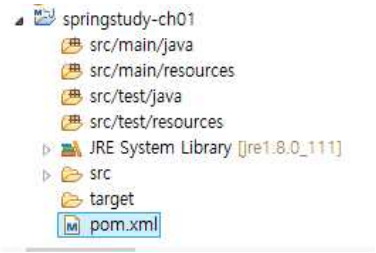

그림 1-13 Maven pom.xml

SpringSTS에서 앞에서 생성한 프로젝트(springclass-ch01 또는 springstudy-ch01)를 그림 1-13과 같이 펼쳐보면 pom.xml 파일이 보일 것이다. 이 파일을 더블 클릭하면 우측(SpringSTS의 소스코드가 표시되는 부분) 화면에 그림 1-14와 같은 설정 화면이 나타나게 된다.

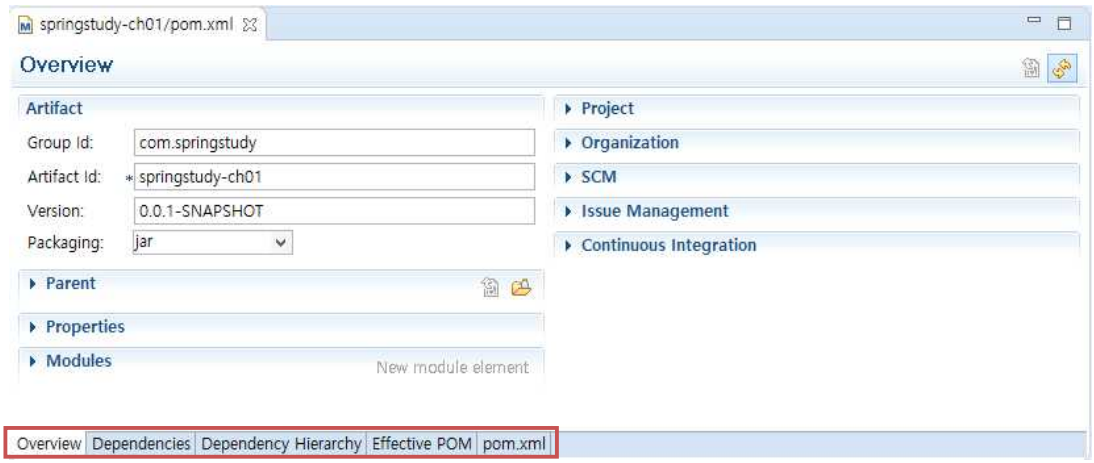

그림 1-14 Maven pom.xml의 설정화면

이 화면의 아래 부분을 살펴보면 각각의 탭 메뉴를 볼 수 있을 것이다. 이 탭 메뉴에서 Overview 메뉴는 프로젝트에 대한 Group Id, Artifact Id 등과 산출물의 패키징 방식, 상위 프로젝트 정보 그리고 pom.xml에서 사용되는 속성(Properties, 프로퍼티 대치 변수) 등을 그림 1-15와 같이 설정하거나 확인할 수 있는 화면이다.

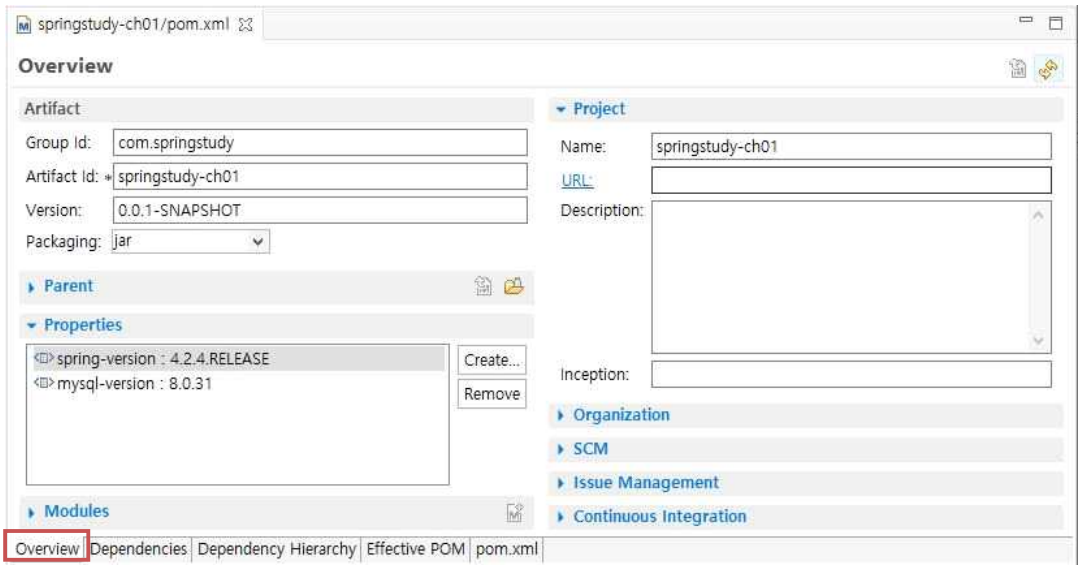

그림 1-15 Maven pom.xml의 Overview 화면

이 화면은 이미 프로젝트를 설정할 때 지정한 정보들이 대부분이고 특별히 수정해야 할 필요가 있을 경우 필요한 부분만 수정하면 된다.

스프링 모듈은 단일 모듈만 사용되는 것이 아니라 서로 의존관계에 있는 여러 개의 스프링 모듈을 사용하기 때문에 모듈의 버전이 모두 동일해야 한다. 우리는 스프링 4.2.4 버전의 모듈을 사용할 것이므로 우리가 사용할 스프링 버전을 Properties에 대치 변수로 설정하고 라이브러리 의존 설정 화면(Dependencies)에서 여기에서 설정한 프로퍼티 대치 변수를 사용해 스프링의 모든 모듈의 버전을 관리할 것이다. Properties에 설정하는 대치 변수는 pom.xml에서 소프트웨어의 버전 정보를 설정하기 위해서 사용하는 일종의 변수라고 생각하면 이해하는데 도움이 될 것이다. 아래 그림 1-16의 순서에 따라서 Properties를 설정해 보자.

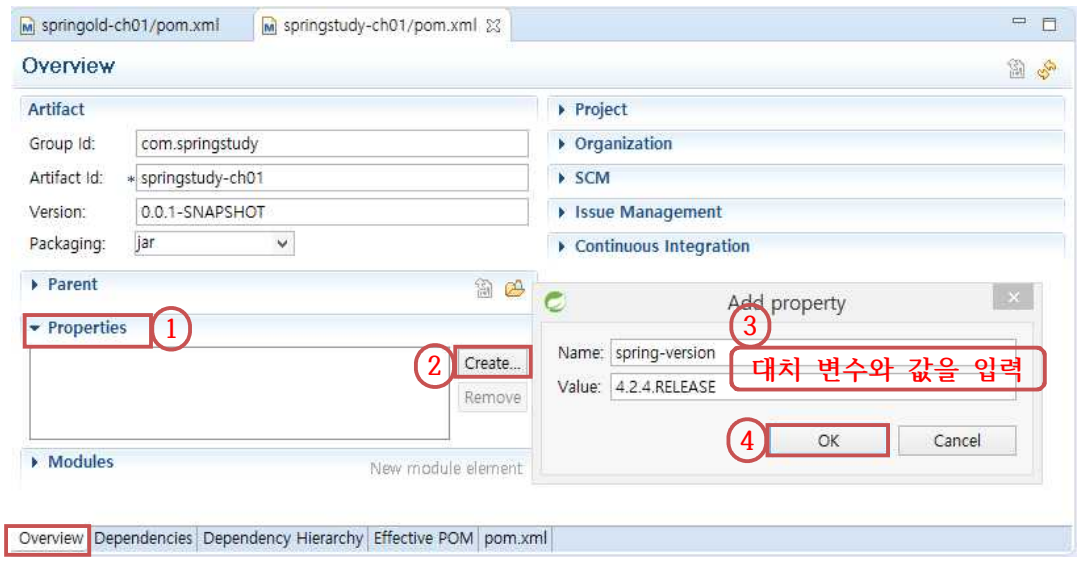

그림 1-16 Maven pom.xml의 Properties 설정

그림 1-17의 Dependencies 메뉴는 이 프로젝트에서 사용하는 라이브러리 의존성을 설정하는 화면이다. 다시 말해 애플리케이션이 실행되면서 참조하는(의존하는) 라이브러리를 이 메뉴에서 설정하면 Maven이 알아서 메이븐 중앙저장소(Maven Central Repository, http://www.maven.org)로부터 다운로드 받아 BuildPath에 추가해 준다.

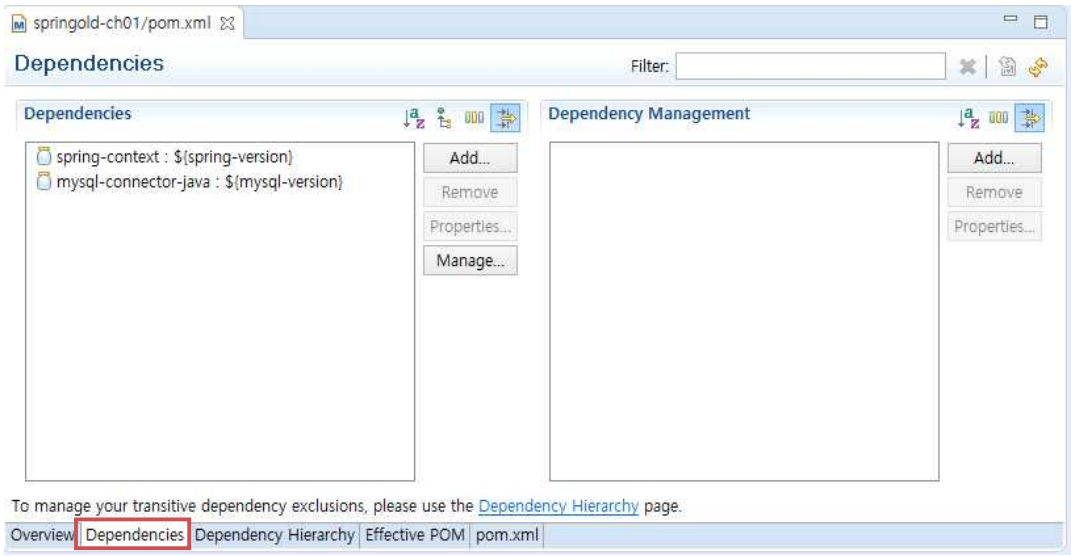

그림 1-17 Maven pom.xml의 Dependencies 화면

그림 1-18과 같은 순서에 따라서 프로젝트에서 의존하는 라이브러리의 의존성을 설정할 수 있다.

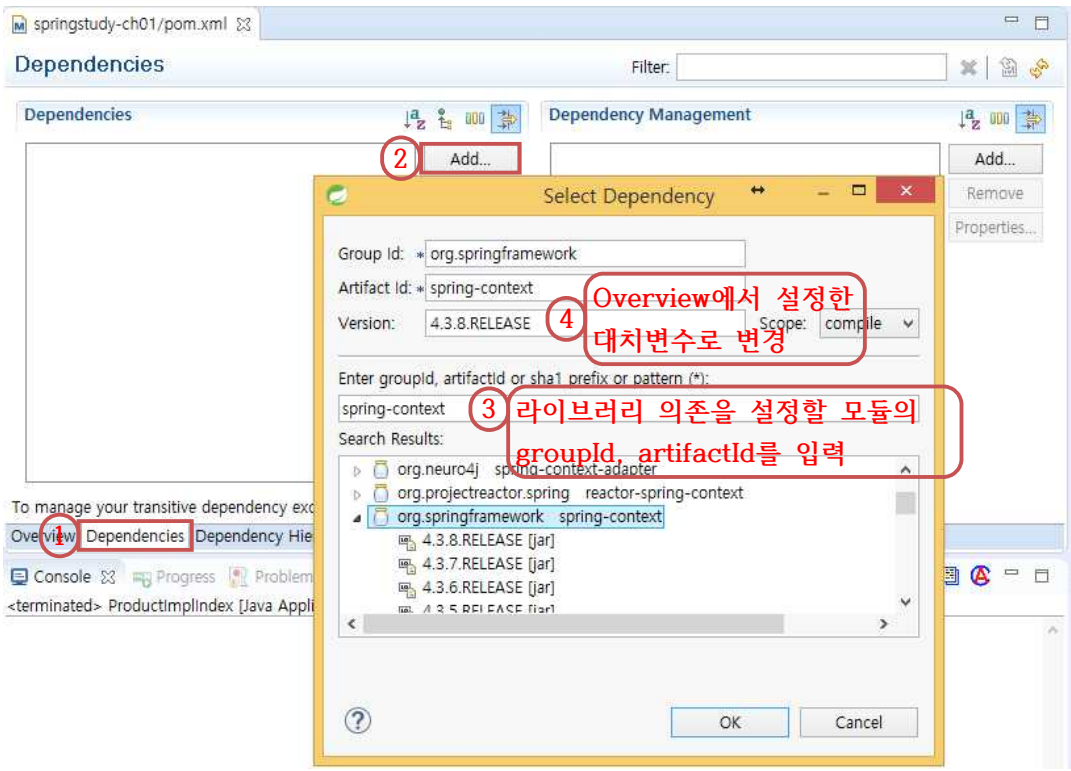

그림 1-18 Maven pom.xml의 Dependencies 설정

그림 1-18의 3번에 이 프로젝트가 의존하는 모듈의 groupId나 artifactId를 입력하면 그 단어가 포함된 라이브러리 리스트가 아래에 나타난다. 이 때 리스트가 나타나지 않는 경우가 종종 있는데 이것은 메이븐 중앙저장소에 저장된 라이브러리의 색인 정보를 읽어오지 못해서 생기는 문제이다.
의존 설정을 하는 라이브러리의 GroupId와 ArtifactId를 모를 경우 http://search.maven.org 또는 http://mvnrepository.com에서 검색할 수 있다.
앞에서 SpringSTS 기본 설정을 할 때 앞에서 살펴보았던 그림 1-2와 같이 Maven Repository Index Update 설정을 하였다. 이 설정은 SpringSTS를 시작할 때 메이븐 중앙저장소에 등록된 라이브러리 색인(Index)을 업데이트하는 설정이다. 만약 라이브러리 리스트가 검색되지 않는다면 그림 1-2(2페이지)를 참고해서 SpringSTS의 Maven 설정이 되어있는지 확인해 보자. 그리고 SpringSTS를 다시 실행하여 라이브러리 색인을 업데이트 한 후 그림 1-18의 순서에 따라서 라이브러리 의존 설정을 하게 되면 제대로 검색 될 것이다.
위의 그림 1-18에서 groupId 또는 artifactId로 검색된 리스트 중에서 하나를 선택하게 되면 그 모듈의 최신 버전이 4번에 자동으로 입력된다. 스프링은 여러 개의 모듈이 존재하기 때문에 하나의 애플리케이션에서 의존하는 스프링 모든 모듈은 같은 버전으로 참조되어야 한다. 그림 1-18의 4번과 같이 특정 버전을 지정할 수도 있지만 만약 애플리케이션에서 참조하는 스프링 버전을 변경해야 한다면 모듈 별로 지정된 버전을 하나하나 변경해야 하는 수고가 따른다. 이럴 경우 그림 1-16에서 Properties에 설정한 대치변수를 사용해 스프링의 버전을 지정하게 되면 다음에 스프링 버전이 변경될 때 Overview 화면의 Properties에 지정된 대치 변수의 값만 변경하면 한 번에 해결할 수 있다. 대치 변수를 지정하는 방법은 Properties에서 지정한 변수를 “${ }”로 감싸서 지정하면 된다. 앞에서 Properties를 설정할 때 name에 spring-version을 value에 4.2.4.RELEASE를 지정했다. 이 프로젝트에서 의존하는 스프링 모듈을 그림 1-16에서 Properties에 설정한 버전으로 지정하려면 그림 1-18에서 4번에 ${spring-version} 를 입력하면 된다.
앞에서 스프링 모듈의 의존관계에 대해서 설명할 때 context 모듈을 기준으로 의존성을 설정하면 이 모듈이 의존하는 모듈을 메이븐이 알아서 Build Path에 추가해 준다고 했다. 스프링 모듈의 의존성을 설정할 때 이 context 모듈이 기준이 되므로 제일 먼저 의존성을 설정해 주고 추가적으로 필요한 스프링 모듈에 대한 의존성을 설정하면 보다 편리하게 의존성을 설정할 수 있을 것이다.
참고로 스프링 모듈 이름에 “spring-”가 붙여진 이름이 해당 모듈의 artifactId가 된다.

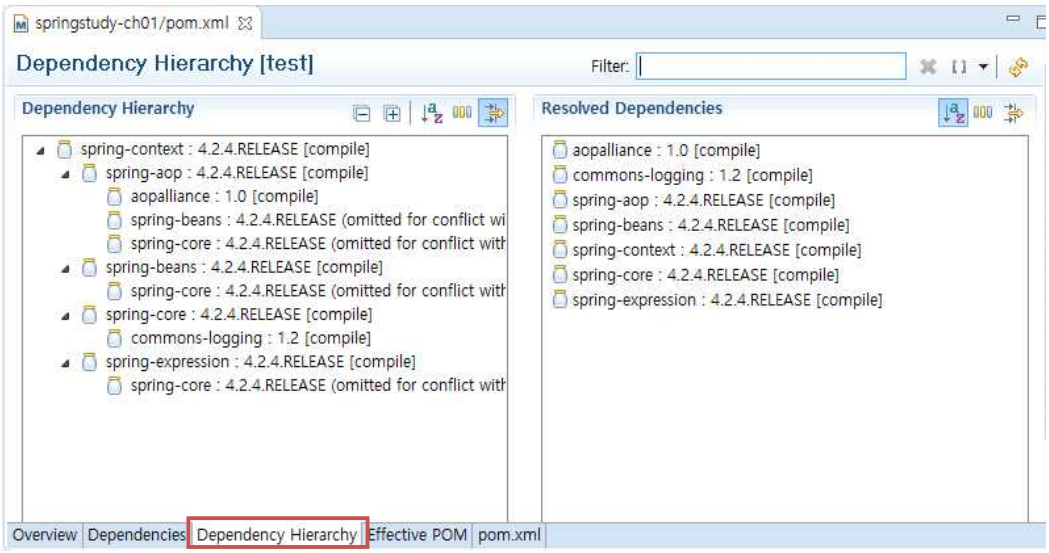

그림 1-19 Maven pom.xml의 Dependency Hierarchy 화면

위의 그림 1-19의 Dependencies Hierarchy 메뉴는 이 프로젝트에서 참조하는 모듈들의 계층 구조를 한 눈에 파악할 수 있는 화면이다.
아래 그림 1-20 Effective POM 메뉴는 모든 pom.xml 파일의 최상위 파일로 메이븐 프로젝트를 생성하게 되면 기본적으로 생성되는 기본 설정 파일이다. 이 설정이 기본적으로 적용되기 때문에 이 프로젝트의 pom.xml에서 별도의 설정 없이 메이븐을 통해 프로젝트 빌드가 가능한 것이다.

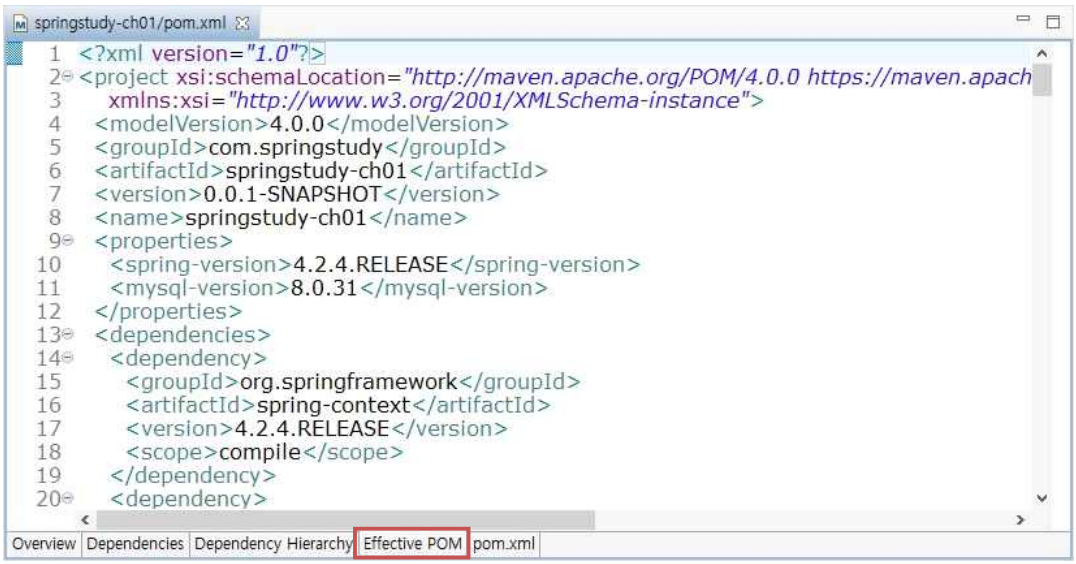

그림 1-20 Maven pom.xml의 Effective POM 화면

아래 1-21의 pom.xml 메뉴는 이 프로젝트의 pom.xml에 설정된 내용을 코드로 확인하고 편집할 수 있는 화면이다.

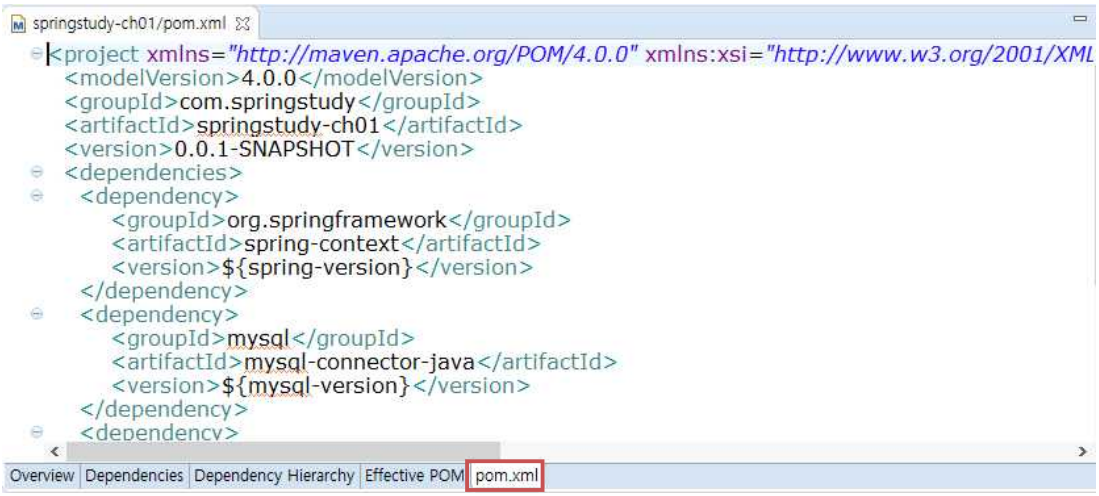

그림 1-21 Maven pom.xml 코드 편집 화면

대부분은 Dependencies 메뉴를 통해서 프로젝트에 필요한 라이브러리의 의존성을 설정할 수 있지만 특별한 경우 직접 코드를 수정해 필요한 라이브러리 의존성을 설정해야 한다. 여기서 특별한 경

우란 라이센스 문제 등으로 라이브러리가 공개되지 않는 경우를 들 수 있는데 Oracle 11g JDBC 드라이버가 대표적인 예 이다. 이런 경우에는 <dependency></dependency>의 groupId와 artifactId 설정만으로 메이븐 중앙저장소를 통해 다운로드 받을 수 없다. 이럴 경우에는 라이브러리를 배포하는 벤더 사이트를 참고해 메이븐 설정을 수동으로 해야 한다.
오라클 접속 드라이버를 메이븐을 사용해 의존성을 설정할 경우 다음의 오라클 사이트를 참고
http://docs.oracle.com/middleware/1213/core/MAVEN/config_maven_repo.htm#MAVEN9010
위의 사이트에서 6.2 Artifacts Provided의 내용을 살펴보면 Oracle Maven Repository는 릴리즈 레벨 artifact만(12.1.2와 12.1.3) 제공한다고 되어있다. 이때 패치가 필요한 경우에는 Oracle Support에서 패치를 구해서 로컬 Oracle Home 설치에 적용하고 Maven Synchronization 플러그인을 사용해 로컬 메이븐 저장소를 업데이트 하라고 되어 있다.
결론부터 말하자면 오라클 데이터베이스를 사용할 경우 별도의 추가 설정 없이 pom.xml에 groupId와 artifactId를 적용해 사용할 수 있는 ojdbc 버전은 12.1 버전으로 SpringSTS의 pom.xml에 아래와 같이 설정하면 이 드라이버를 메이븐이 로컬 저장소로 다운로드 해 준다.
참고로 이 버전의 드라이버는 오라클 10g와 11g에 접속하는데 사용할 수 있다.

```
<dependencies>
  <dependency>
     <groupId>com.oracle</groupId>
     <artifactId>ojdbc6</artifactId>
     <version>12.1.0.2</version>
  </dependency>
</dependencies>

<repositories>
  <repository>
    <id>oracle</id>
    <name>ORACLE JDBC Repository</name>
    <url>http://maven.jahia.org/maven2</url>
  </repository>
</repositories>
```

1.2 강한 결합 vs 약한 결합

- com.springstudy.ch01.strong

```
public class ProductDAO {

    Connection conn;
    PreparedStatement pstmt;
    ResultSet rs;
    String driver = "com.mysql.cj.jdbc.Driver";
    String url = "jdbc:mysql://localhost:3306/spring?useSSL=false&useUnicode=true
&characterEncoding=utf8";
    String user = "root";
    String pass = "12345678";

    public ArrayList<Product> getProductList() {

        String selectSql = "SELECT * FROM product";
        ArrayList<Product> pList = new ArrayList<Product>();

        try{
            Class.forName(driver);
            conn = DriverManager.getConnection(url, user, pass);
            pstmt = conn.prepareStatement(selectSql);
            rs = pstmt.executeQuery();

            while(rs.next()) {
                Product p = new Product();
                p.setCode(rs.getString("code"));
                p.setName(rs.getString("name"));
                p.setPrice(rs.getInt("price"));
                p.setDescription(rs.getString("description"));
                pList.add(p);
            }

        } catch(Exception e) {
            e.printStackTrace();

        } finally {
            try{
                if(rs != null) rs.close();
                if(pstmt != null) pstmt.close();
                if(conn != null) conn.close();
            } catch(SQLException e) { }
        }

        return pList;
```

```
    }
}
```

- com.springstudy.ch01.domain

```
public class Product {

    private String code;
    private String name;
    private int price;
    private String description;

    public Product() { }

    public String getCode() {
        return code;
    }
    public void setCode(String code) {
        this.code = code;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String toString() {
        return code + " : " + name + " : " + price + " : " + description;
    }
}
```

- com.springstudy.ch01.strong

```
public class ProductService {

    public ArrayList<Product> getProductList() {

        ProductDAO dao = new ProductDAO();
        System.out.println("strong-service : ProductService.getProductList()");
        return dao.getProductList();
    }
}
```

- com.springstudy.ch01.strong

```
public class ProductStrongIndex {

    public static void main(String[] args) {

        ProductService service = new ProductService();
        ArrayList<Product> pList = service.getProductList();

        for(Product p : pList) {
            System.out.println(p);
        }
    }
}
```

위의 예제에서 ProductStrongIndex 클래스에서 ProductService 클래스의 인스턴스를 생성하는 모습을 볼 수 있다. 이렇게 A라는 클래스에서 B라는 클래스 이름으로 인스턴스를 직접 생성해 사용하는 경우 클래스 B가 수정되면 클래스 A도 수정이 불가피 하다. 이런 경우 A와 B가 강하게 결합되어 있으며 A가 B 에 강하게 의존하고 있다고 말한다. 이런 부류의 애플리케이션은 확장성과 유연성이 떨어지고 테스트에 어려움이 많기 때문에 유지보수 비용이 많이 들어가게 된다.
위의 예를 살펴보면 ProductStrongIndex 클래스의 main() 메서드를 보면 ProductService 클래스를 new 연산자를 이용해 객체를 직접 생성해 사용하는데 만약 ProductService 클래스가 ProductService01로 수정되어야 한다면 new 연산자를 이용해 객체를 생성하는 부분과 이를 참조하는 참조 타입의 변수 선언 부분도 같이 수정되어야 한다. 이렇게 ProductService 클래스가 수정될 때 ProductStrongIndex 클래스도 수정이 불가피하게 되는데 이런 경우 두 클래스는 강하게 결합되어 있으며 “ProductStrongIndex 클래스가 ProductService 클래스에 강하게 의존한다.”라고 말할 수 있다. 또한 ProductService 클래스도 ProductDAO 클래스를 직접 생성해 사용하기 때문에 ProductDAO 클래스에 강하게 의존하게 된다.
위의 예는 단순히 하나의 클래스를 보고 있지만 이렇게 강하게 의존하는 경우가 애플리케이션 전반에 걸쳐 여러 클래스에 존재한다면 하나의 클래스 수정이 애플리케이션 전반에 걸친 수정으로 이어질 수 있다. 그러므로 이렇게 클래스들 간의 강한 결합을 느슨한 결합으로 만들어 클래스들 간의 의존성을 해결해야 한다.

1.3 인터페이스를 이용한 객체 간 결합 낮추기

- com.springstudy.ch01.interfaces

```
public interface ProductService {
    public ArrayList<Product> getProductList();
}
```

- com.springstudy.ch01.interfaces

```
public class ProductServiceImpl01 implements ProductService {

    public ArrayList<Product> getProductList() {

        ProductDAO dao = new ProductDAOImpl();
        System.out.println("impl-service : ProductServiceImpl01.getProductList()");
        return dao.getProductList();
    }
}
```

- com.springstudy.ch01.interfaces

```
public class ProductServiceImpl02 implements ProductService {

    public ArrayList<Product> getProductList() {

        ProductDAO dao = new ProductDAOImpl();
        System.out.println("impl-service : ProductServiceImpl02.getProductList()");
        return dao.getProductList();
    }
}
```

- com.springstudy.ch01.interfaces

```
public interface ProductDAO {
    public ArrayList<Product> getProductList();
}
```

- com.springstudy.ch01.interfaces

```
public class ProductDAOImpl implements ProductDAO {

    Connection conn;
    PreparedStatement pstmt;
    ResultSet rs;
    String driver = "com.mysql.cj.jdbc.Driver";
```

```
    String url = "jdbc:mysql://localhost:3306/spring?useSSL=false&useUnicode=true
&characterEncoding=utf8";
    String user = "root";
    String pass = "12345678";

    public ArrayList<Product> getProductList() {

        String selectSql = "SELECT * FROM product";
        ArrayList<Product> pList = new ArrayList<Product>();

        try{
            Class.forName(driver);
            conn = DriverManager.getConnection(url, user, pass);
            pstmt = conn.prepareStatement(selectSql);
            rs = pstmt.executeQuery();

            while(rs.next()) {
                Product p = new Product();
                p.setCode(rs.getString("code"));
                p.setName(rs.getString("name"));
                p.setPrice(rs.getInt("price"));
                p.setDescription(rs.getString("description"));
                pList.add(p);
            }

        } catch(Exception e) {
            e.printStackTrace();

        } finally {
            try{
                if(rs != null) rs.close();
                if(pstmt != null) pstmt.close();
                if(conn != null) conn.close();
            } catch(SQLException e) { }
        }
        return pList;
    }
}
```

- com.springstudy.ch01.interfaces

```
public class ProductImplIndex {

    public static void main(String[] args) {
```

```
        ProductService service = new ProductServiceImpl01();
        //ProductService service = new ProductServiceImpl02();
        ArrayList<Product> pList = service.getProductList();

        for(Product p : pList) {
            System.out.println(p.toString());
        }
    }
}
```

위의 예제 ProductImpIndex 클래스에서 ProductService 인터페이스타입으로 이 인터페이스를 구현한 ProductServiceImpl01의 인스턴스를 생성하는 모습을 볼 수 있다. 그리고 그 아래 주석 처리된 ProductService 인터페이스 타입으로 ProductServiceImpl02의 인스턴스를 생성하는 코드도 볼 수 있을 것이다. 이렇게 인터페이스를 사용하면 구현체가 바뀌더라도 상품의 리스트를 받아오는 메서드 호출문은 바뀌지 않는다. 인터페이스를 이용해 상속과 다형성을 구현하고 특정 객체에 대한 의존성을 낮춰 객체 사용의 대한 유연성을 높였다. 하지만 아직도 ProductServiceImpl01나 ProductServiceImpl02라는 클래스 이름으로 객체를 생성하고 있기 때문에 ProductServiceImpl01나 ProductServiceImpl02 클래스가 변경 된다면 ProductImplIndex 클래스의 코드 수정이 불가피하게 되므로 객체간의 의존성 또한 최소화 하였다고 볼 수 없을 것이다. 다음 예제는 객체 간의 의존성을 낮춰서 객체 간의 결합을 느슨하게 만들고 객체 사용의 유연성을 높이기 위해서 현재 클래스에서 필요한 객체를 외부의 다른 주체로부터 전달 받아 사용하는 방식에 대해 알아 볼 것이다.

1.4 인터페이스와 팩토리 클래스를 이용한 객체 간 결합 낮추기

- com.springstudy.ch01.factory

```
public class ProductFactory {

    public static ProductDAO createDAO() {
        // MySQL DB용 DAO
        //return new ProductDAOImpl02();

        // Oracle DB용 DAO
        return new ProductDAOImpl03();
    }

    public static ProductService createService() {
        return new ProductServiceImpl03();
    }
}
```

- com.springstudy.ch01.factory

```
public class ProductServiceImpl03 implements ProductService {

    public ArrayList<Product> getProductList() {

        ProductDAO dao = ProductFactory.createDAO();
        System.out.println("light-service : ProductServiceImpl03.getProductList()");
        return dao.getProductList();
    }
}
```

- com.springstudy.ch01.factory

```
public class ProductServiceImpl04 implements ProductService {

    public ArrayList<Product> getProductList() {

        ProductDAO dao = ProductFactory.createDAO();
        System.out.println("light-service : ProductServiceImpl04.getProductList()");
        return dao.getProductList();
    }
}
```

- com.springstudy.ch01.factory

```
// MySQL DB용 DAO
public class ProductDAOImpl02 implements ProductDAO {
```

```
    Connection conn;
    PreparedStatement pstmt;
    ResultSet rs;
    String driver = "com.mysql.cj.jdbc.Driver";
    String url = "jdbc:mysql://localhost:3306/spring?useSSL=false&useUnicode=true
&characterEncoding=utf8";
    String user = "root";
    String pass = "12345678";

    public ArrayList<Product> getProductList() {

        String selectSql = "SELECT * FROM product";
        ArrayList<Product> pList = new ArrayList<Product>();

        try{
            Class.forName(driver);
            conn = DriverManager.getConnection(url, user, pass);
            pstmt = conn.prepareStatement(selectSql);
            rs = pstmt.executeQuery();

            while(rs.next()) {
                Product p = new Product();
                p.setCode(rs.getString("code"));
                p.setName(rs.getString("name"));
                p.setPrice(rs.getInt("price"));
                p.setDescription(rs.getString("description"));
                pList.add(p);
            }
            System.out.println("MySQL - getProductList()");
        } catch(Exception e) {
            e.printStackTrace();

        } finally {
            try{
                if(rs != null) rs.close();
                if(pstmt != null) pstmt.close();
                if(conn != null) conn.close();
            } catch(SQLException e) { }
        }
        return pList;
    }
}
```

- com.springstudy.ch01.factory

```
// Oracle DB용 DAO
public class ProductDAOImpl03 implements ProductDAO {

    Connection conn;
    PreparedStatement pstmt;
    ResultSet rs;
    String driver = "oracle.jdbc.driver.OracleDriver";
    String url = "jdbc:oracle:thin:@localhost:1521/xe";
    String user = "hr";
    String pass = "hr";

    public ArrayList<Product> getProductList() {

        String selectSql = "SELECT * FROM product";
        ArrayList<Product> pList = new ArrayList<Product>();

        try{
            Class.forName(driver);
            conn = DriverManager.getConnection(url, user, pass);
            pstmt = conn.prepareStatement(selectSql);
            rs = pstmt.executeQuery();

            while(rs.next()) {
                Product p = new Product();
                p.setCode(rs.getString("code"));
                p.setName(rs.getString("name"));
                p.setPrice(rs.getInt("price"));
                p.setDescription(rs.getString("description"));
                pList.add(p);
            }
            System.out.println("Oracle - getProductList()");
        } catch(Exception e) {
            e.printStackTrace();

        } finally {
            try{
                if(rs != null) rs.close();
                if(pstmt != null) pstmt.close();
                if(conn != null) conn.close();
            } catch(SQLException e) { }
        }
        return pList;
    }
}
```

- com.springstudy.ch01.factory

```
public class ProductLooseIndex {

    public static void main(String[] args) {

        ProductService service = ProductFactory.createService();
        ArrayList<Product> pList = service.getProductList();

        for(Product p : pList) {
            System.out.println(p.toString());
        }
    }
}
```

위의 ProductLooseIndex 클래스에서 ProductService 인터페이스 타입으로 이 인터페이스를 구현한 ProductServiceImpl03 클래스의 인스턴스를 클래스 이름을 직접 사용하지 않고 ProductFactory 클래스를 통해 생성하고 있는 모습을 볼 수 있다. 또한 ProductServiceImpl03 클래스와 ProductServiceImpl04 클래스에서 ProductDAO 인터페이스를 구현한 ProductDAOImpl02 클래스의 인스턴스를 클래스 이름을 직접 사용하지 않고 생성하는 모습도 볼 수 있다.
만약 ProductLooseIndex 클래스에서 ProductServiceImpl03 클래스의 인스턴스가 아니라 ProductServiceImpl04 클래스의 인스턴스로 수정해야 할 필요가 있다면 단순히 ProductFactory 에서 객체를 생성하는 부분을 ProductServiceImpl04로 변경하기만 하면 된다.
또한 ProductServiceImpl03 클래스의 인스턴스를 사용하는 ProductLooseIndex와 같은 클래스가 하나가 아니라 수백 개라도 ProductFactory 클래스의 단 한 줄만 수정하면 다른 부분은 수정하지 않아도 애플리케이션은 원하는 기능을 제대로 수행 할 수 있게 된다.
이렇게 의존하는 객체가 변경되더라도 최소한의 수정을 통해 코드의 변경과 확장이 가능하기 때문에 애플리케이션의 유지보수 비용 또한 최소화 할 수 있을 것이다.
애플리케이션이 필요한 객체를 외부의 어떤 주체로부터 주입 받음으로써 의존하는 객체가 변경되더라도 애플리케이션의 수정을 최소화 할 수 있게 되는데 이렇게 외부로부터 필요한 객체를 주입 받는 것을 DI(Dependency Injection, 의존객체 주입)라고 한다.
우리가 학습할 SpringFramework(이하 스프링)는 애플리케이션에서 필요한 객체를 생성하고 주입해 주는 DI 컨테이너를 보유하고 있다. 스프링에서 XML 파일에 Bean(필요한 자바 객체)을 설정하거나 Annotation을 사용해 Bean을 설정 하게 되면 스프링 컨테이너가 자동으로 의존하는 객체를 생성해 주입하고 그 객체를 관리해 준다.
참고로 위의 예제에서 사용된 팩토리 클래스는 이해를 돕기 위해 간단히 작성된 것으로 실제 팩토리 패턴의 구현과는 차이가 있다.

1.5 SpringFramework 참고 사이트

- 스프링 공식 사이트 : http://spring.io
- 스프링 프로젝트 : 스프링 팀이 진행하는 프로젝트들과 및 레퍼런스 문서 참고
  https://spring.io/projects
- 전자정부 프레임워크 포털 : http://www.egovframe.go.kr
- 한국 스프링 사용자 모임 : http://www.ksug.org
- 메이븐 의존성 라이브러리 관리 : 그룹id, artifactId검색
  http://mvnrepository.com/

2. Spring DI(Dependency Injection)

2.1 IoC와 DI

2.1.1 Inversion of Control(IoC)

Inversion of Control이란 단어는 제어역전 또는 제어역행으로 번역되며 어떤 클래스에서 의존하는 객체를 그 클래스 내부에서 생성하지 않고 그림 2-2와 같이 외부의 어떤 주체로 부터 전달 받는 것을 의미한다. 어떤 클래스에서 의존하는 객체를 new 연산자를 사용해 생성하는 것을 제어순행 이라고 한다. new 연산자를 사용해 객체를 생성하게 되면 객체를 생성한 클래스가 그 객체의 소유권을 가지고 객체를 제어할 수 있기 때문에 제어순행 이라는 말을 사용한다. 반면 외부의 주체로부터 객체를 주입받게 되면 객체를 사용할 수는 있지만 객체의 소유권을 가지고 있지 않기 때문에 이를 제어역행이라고 부른다.

IoC는 스프링 프레임워크의 핵심 기능으로 스프링프레임워크가 세상에 나오기 훨씬 이전부터 존재했던 개념이다.

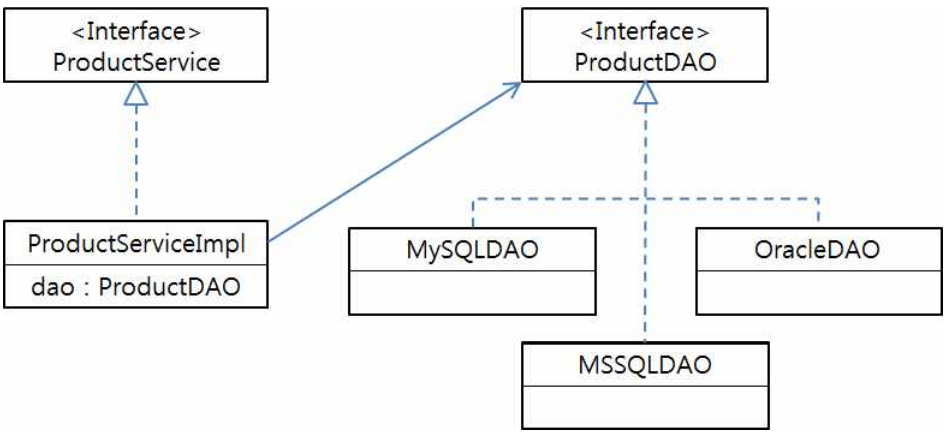

그림 2-1 객체 간의 의존 관계

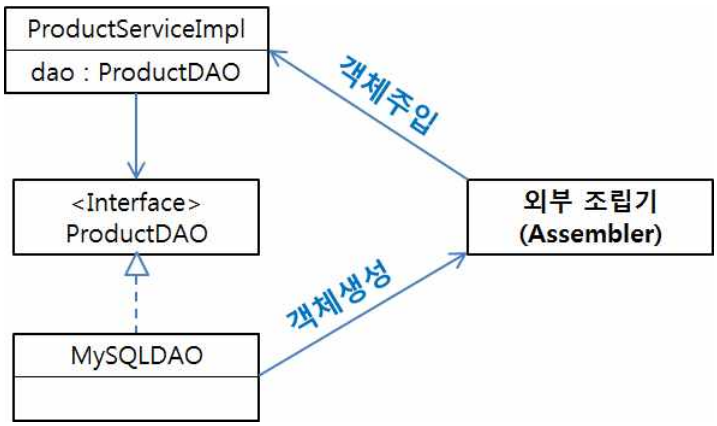

그림 2-2 외부 조립기에 의한 객체 주입

그림 2-1을 보면 ProductServiceImpl 안에서 ProductDAO에 의존한다.
ProductServiceImpl 클래스 안에서 ProductDAO 인터페이스를 구현한 MySQLDAO, OracleDAO, MSSQLDAO 클래스가 사용되므로 ProductServiceImpl 객체는 ProductDAO 인터페이스에 의존하게 된다. 만약 ProductServiceImpl 클래스 안에서 ProductDAO의 구현체를 new 연산자로 직접 생성해 사용하면 객체의 소유권은 ProductServiceImpl에 있으며 객체를 생성 하고 관리하는 권한 또한 소유권을 가진 ProductServiceImpl에게 있기 때문에 제어순행 이라고 한다. 하지만 그림 2-2와 같이 외부 조립기에서 ProductServiceImpl에 필요한 ProductDAO 구현체인 MySQLDAO 객체를 생성해 주입하고 ProductServiceImpl에서 사용되지만 사실 객체의 소유권(객체를 생성하고 관리하는 권한)은 외부 조립기에 있기 때문에 이를 제어역행(Inversion of Control)이라고 말 한다.

2.1.2 Dependency Injection(DI)

2004년 마틴 파울러(Martin Fowler)가 자신의 글에서 제어의 어떤 측면이 역행되는 것인지에 대해 의문을 제기하고 필요한 객체를 외부의 어떤 주체로부터 주입받는 것이기 때문에 객체를 역행적으로 취득하는 것이 아니라는 결론을 내리고 현재 경량 컨테이너(Lightweight Container)들이 주장하는 IoC의 개념은 DI 라는 용어가 더 적합하다는 주장에서 유래 되었다.
DI는 객체 간의 의존 관계를 특정 시스템 안에서 객체의 생성과 생명주기를 관리하는 어떤 주체에 의해 필요한 객체를 주입받기 때문에 의존성(Dependency)이 외부로 부터 주입(Inject) 된다는 의미이다.
어떤 클래스가 다른 클래스가 가진 기능을 사용하기 위해 의존하는 객체를 직접 생성하여 사용하게 되면 그 객체에 강하게 의존한다는 것을 앞에서 예제를 통해서 알아보았다. 또한 외부의 어떤 주체로부터 필요한 객체를 넘겨받아 사용하면 객체 간의 결합이 느슨하게 되는 것도 알아보았다. 이번 예제부터는 어떤 클래스가 의존하는 객체를 스프링 Bean으로 설정하고 스프링으로부터 주입받아 객체에 대한 의존성을 낮추는 방법에 대해 알아볼 것이다.
스프링에서 Bean을 설정하고 객체간의 의존 관계를 설정하는 것을 일반적으로 의존성 주입(Dependency Injection) 또는 의존 관계 주입이라고 한다. 스프링에서 의존성 주입을 설정하는 방법은 크게 세 가지로 나눌 수 있는데 그 첫 번째는 스프링 Bean 설정 파일인 XML 파일을 이용하여 의존성을 설정 하는 방법이며 두 번째는 스프링이 제공하는 Annotation을 사용해 Bean으로 설정하는 방법이 있다. 그리고 세 번째 방법에는 자바 클래스를 스프링 빈 설정 클래스로 정의하여 의존성을 주입하는 방법이 있다. 스프링에서 Bean의 의미는 자바 객체 또는 클래스의 Instance와 같은 의미로 스프링 DI 컨테이너에서 생성되고 관리되는 평범한 자바 객체(Plan Old Java Object)를 말하며 이런 자바 객체를 줄여서 POJO라고 부른다.
예전에는 Java EE와 같은 중량 프레임워크를 많이 사용하였다. 이때의 중량 프레임워크는 그들이 제공하는 인터페이스를 구현하도록 설계되었는데, 이로 인해 그 프레임워크에 강하게 종속되는 객체를 만들게 되었다. 이렇게 프레임워크에 강하게 종속되는 객체를 무거운 객체라고 부르며 이런 무거운 객체를 만들게 된 것에 반해 마틴 파울러와 몇몇의 개발자가 POJO(Plan Old Java Object)라는 말을 사용하기 시작한데서 유래되었다. 다시 말해 POJO란 프레임워크가 제공하는 어떠한 클래스나 인터페이스를 상속받지 않은 순수한 자바 객체를 의미한다. POJO가 아닌 대표적인 클래스로 자바 서블릿 클래스를 예로 들 수 있을 수 있다. 서블릿으로 동작하기 위해서는 HttpServlet 클래스를 상속받아야 하므로 Servlet 기술이 제공하는 클래스의 상속을 강제하고 있기 때문이다.
중량 프레임워크(중량 컨테이너)란 어떤 프레임워크를 사용하기 위해 프레임워크가 제공하는 클래스나, 인터페이스를 강제로 상속받도록 설계되어 있는 프레임워크를 말하며, 스프링 프레임워크는 이와 비교해 스프링이 제공하는 어떠한 클래스나 인터페이스의 구현도 강요하지 않기 때문에 경량 프레임워크(경량 컨테이너)라고 부른다.

스프링프레임워크는 시스템 안에서 어떤 클래스가 의존하는 객체를 생성하고 그 클래스에 필요한 객체를 주입해 주는 역할을 한다. 스프링프레임워크는 객체를 생성하고 그 객체의 생명주기를 관리해 주는 컨테이너를 보유 하고 있어 스프링프레임워크를 IoC 컨테이너 또는 DI 컨테이너라고 부른다. IoC는 DI보다 조금 더 넓은 범위의 개념을 가지고 있는 용어라고 생각하면 좋을 것 같다.

2.1.3 스프링 DI의 종류

그림 2-3은 스프링이 제공하는 DI의 종류를 나타낸 것으로 이 그림에서 IoC는 DL(Dependency Lookup)과 DI(Dependency Injection) 두 가지 분류가 존재함을 알 수 있다.

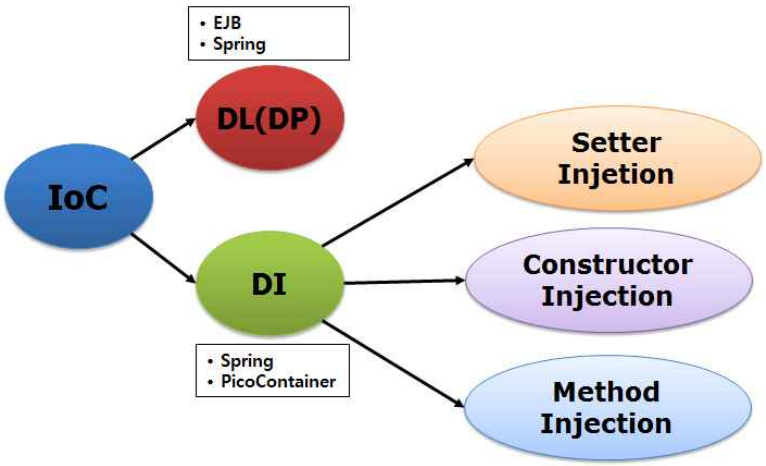

그림 2-3 스프링 DI의 종류

DL은 스프링에게 Bean을 요청해 필요한 객체를 받아오는 것을 의미하며 DI는 스프링으로부터 자동으로 객체가 주입되는 것을 의미 한다. 우리의 애플리케이션에서 필요한 객체를 스프링으로부터 주입(DI) 받기 위해서는 특별한 설정이 필요한데 바로 이 방법이 앞에서 설명한 XML설정과 Annotation 설정 그리고 자바 클래스를 스프링 빈으로 설정하는 3가지 방법을 말한 것이다.

2.1.4 스프링 주요 모듈과 스키마

표 2-1은 스프링프에서 제공하는 주요 모듈과 스키마 그리고 지원 기능을 표로 나타낸 것이다.

모듈 명	URI / 스키마 정의, 모듈(jar) / 설명
spring-beans	http://www.springframework.org/schema/beans
	spring-beans-4.2.xsd, spring-beans-4.2.x.RELEASE.jar
	bean 컴포넌트 설정을 사용해 객체를 생성하는 기본 기능 제공
spring-context	http://www.springframework.org/schema/context

	spring-context-4.2.xsd, spring-context-4.2.x.RELEASE.jar
	Annotation 설정과 Bean 컴포넌트 스캔을 사용한 객체 생성, bean의 라이프 사이클 관리, 스키마 확장 등의 기능을 제공
spring-aop	http://www.springframework.org/schema/aop
	spring-aop-4.2.xsd, spring-aop-4.2.x.RELEASE.jar
	AOP(Aspect Oriented Programming) 기능 제공
spring-jdbc	http://www.springframework.org/schema/jdbc
	spring-jdbc-4.2.xsd, spring-jdbc-4.2.x.RELEASE.jar
	JDBC 프로그래밍을 위한 스프링 JDBC 템플릿 제공
spring-tx	http://www.springframework.org/schema/tx
	spring-tx-4.2.xsd, spring-tx-4.2.x.RELEASE.jar
	스프링 트랜잭션 관리자를 이용한 트랜잭션 처리 기능 제공
spring-orm	URI 없음
	스키마 정의 없음, spring-orm-4.2.x.RELEASE.jar
	MyBatis, Hibernate, JPA 등과 연동할 수 있는 기능 제공
spring-webmvc	http://www.springframework.org/schema/mvc
	spring-mvc-4.2.xsd, spring-webmvc-4.2.x.RELEASE.jar
	스프링 기반 MVC 프레임워크 제공, 웹 애플리케이션에 필요한 컨트롤러, 뷰 구현 기능 제공
spring-web	URI 없음
	스키마 정의 없음, spring-web-4.2.x.RELEASE.jar
	데이터 변환, 서블릿 필터, REST 클라이언트, 파일 업로드 지원 등 웹 애플리케이션 개발에 필요한 추가 기능 제공
spring-websocket	http://www.springframework.org/schema/websocket
	spring-websocket-4.2.xsd, spring-websocket-4.2.x.RELEASE.jar
	스프링 MVC에서 웹 소켓 연동을 처리할 수 있는 기능 제공
spring-oxm	http://www.springframework.org/schema/oxm
	spring-oxm-4.2.xsd, spring-oxm-4.2.x.RELEASE.jar
	XML과 자바 객체 간의 매핑 처리를 위한 기능 제공
spring-jms	http://www.springframework.org/schema/jms
	spring-jms-4.2.xsd, spring-jms-4.2.x.RELEASE.jar
	JMS 서버와 메시지를 쉽게 주고받을 수 있는 템플릿, Annotation 기능 제공
spring-context-support	URI 없음
	스키마 정의 없음, spring-context-support-4.2.x.RELEASE.jar
	스케줄링, 메일 발송, 캐시 연동, 벨로시티 등 부가기능 제공
util	http://www.springframework.org/schema/util
	spring-util-4.2.xsd, spring-core-4.2.x.RELEASE.jar
	설정파일, Properties 파일 읽어오기 등의 유틸리티 기능 제공

jee	http://www.springframework.org/schema/jee
	spring-jee-4.2.xsd, spring-context-4.2.x.RELEASE.jar
	JNDI의 Lookup, EJB의 Lookup 기능 제공
lang	http://www.springframework.org/schema/lang
	spring-lang-4.2.xsd, spring-core-4.2.x.RELEASE.jar
	스크립트 언어를 사용할 경우 추가 기능 제공

표 2-1 스프링프레임워크의 주요 모듈과 스키마

2.1.5 스프링 DI(IoC) 컨테이너

1) 스프링 DI(Dependency Injection) 컨테이너

스프링 애플리케이션에서는 스프링 DI 컨테이너에서 객체(bean)가 생성되고 관리된다.
스프링 DI 컨테이너는 스프링 프레임워크의 핵심 기능 중 하나로 의존 객체 주입(또는 종속객체 주입이라 함)을 통해 객체를 생성하고 관리하며 객체간의 관계를 맺어 준다.
스프링 프레임워크에는 여러 개의 DI 컨테이너 구현체가 존재하며 크게 두 부류로 나눌 수 있다.

2) 빈 팩토리(Bean Factory)

org.springframework.bean.factory.BeanFactory 인터페이스로 DI에 대한 가장 기본적인 기능을 제공하는 스프링프레임워크의 가장 단순한 DI 컨테이너 이다.

3) 애플리케이션 컨텍스트(Application Context)

org.springframework.context.ApplicationContext 인터페이스는 BeanFactory 인터페이스의 자손으로 ApplicationCotext는 bean을 생성하고 관리하는 스프링 DI 컨테이너 이다.
이 ApplicationContext를 상속한 다양한 애플리케이션 컨텍스트 구현체가 존재하며 아래는 많이 사용되는 ApplicationContext 인터페이스의 구현체 이다.

▶ ClassPathXmlApplicationContext :
클래스 패스에 위치한 XML 파일을 빈 설정 정보로 사용하는 스프링 DI 컨테이너 클래스

▶ FileSytemXmlApplicationContext :
지정한 경로에 위치한 XML 파일을 빈 설정 정보로 사용하는 스프링 DI 컨테이너 클래스

▶ GenericXmlApplicationContext :
ClassPathXmlApplicationContext와 FileSystemXmlApplicationContext 두 클래스가 가지고 있는 기능을 모두 지원하는 스프링 DI 컨테이너 클래스로 스프링 3.0 부터 추가됨

▶ AnnotationConfigApplicationContext :
지정한 자바 클래스를 빈 설정 정보로 사용하는 스프링 DI 컨테이너 클래스

▶ XmlWebApplicationContext :

웹 애플리케이션에서 XML 파일을 빈 설정 정보로 사용하는 스프링 DI 컨테이너 클래스

▶ AnnotationConfigWebApplicationContext :

웹 애플리케이션에서 자바 코드를 빈 설정 정보로 사용하는 스프링 DI 컨테이너 클래스

2.2 XML 설정을 이용한 DI

앞에서 스프링을 통해 의존하는 객체를 주입받는 방법에는 크게 3가지가 있다고 했다.
이번에 우리가 학습할 내용은 스프링 Bean 설정 파일인 XML 파일에 의존하는 객체를 스프링 Bean으로 정의하고 이 객체를 필요로 하는 클래스에서 생성자 또는 세터 메서드를 통해 스프링 DI 컨테이너로부터 주입받는 방법에 대해 알아볼 것이다.
먼저 수업시간에 제공한 springclass-ch02.zip과 springstudy-ch02.zip 프로젝트를 import 하고 이 프로젝트의 SQL 폴더에 있는 member.sql을 사용해 MySQL DBMS의 spring 데이터베이스에 테이블을 생성하고 데이터를 추가하자.

실습 프로젝트 : springclass-ch02.zip
완성 프로젝트 : springstudy-ch02.zip

2.2.1 생성자 주입(Constructor Injection)

- com.springstudy.ch02.domain

```
public class Member {
    private String id;
    private String name;
    private String pass;
    private int age;
    private String email;
    private Timestamp regDate;

    public Member() { }

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPass() {
        return pass;
    }
    public void setPass(String pass) {
        this.pass = pass;
    }
    public int getAge() {
```

```
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public Timestamp getRegDate() {
        return regDate;
    }
    public void setRegDate(Timestamp regDate) {
        this.regDate = regDate;
    }

    @Override
    public String toString() {

        Calendar cal = Calendar.getInstance();
        cal.setTimeInMillis(getRegDate().getTime());

        String regDate = cal.get(Calendar.YEAR) + "년 "
                + (cal.get(Calendar.MONTH) + 1) + "월 "
                + cal.get(Calendar.DAY_OF_MONTH);

        return name + "(" + age + ") : " + id + " - " + pass
                + " : " + email + " : " + regDate;
    }
}
```

- com.springstudy.ch02.dao

```
public interface MemberDAO {
    public ArrayList<Member> getMemberList();
}
```

- com.springstudy.ch02.dao

```
public class MemberDAOImpl implements MemberDAO {

    private Connection conn;
    private PreparedStatement pstmt;
```

```
private ResultSet rs;
private DriverManagerDataSource dataSource;

// 스프링 DI 컨테이너로 부터 DriverManagerDataSource 객체를 주입받는 생성자
public MemberDAOImpl(DriverManagerDataSource dataSource) {
    this.dataSource = dataSource;
}

@Override
public ArrayList<Member> getMemberList() {

    String selectAllMember = "SELECT * FROM member;";
    ArrayList<Member> memberList = null;

    try {
        conn = dataSource.getConnection();
        pstmt = conn.prepareStatement(selectAllMember);
        rs = pstmt.executeQuery();

        memberList = new ArrayList<Member>();

        while(rs.next()) {

            Member member = new Member();
            member.setId(rs.getString("id"));
            member.setName(rs.getString("name"));
            member.setPass(rs.getString("pass"));
            member.setAge(rs.getInt("age"));
            member.setEmail(rs.getString("email"));
            member.setRegDate(rs.getTimestamp("reg_date"));

            memberList.add(member);
        }

    } catch(SQLException e) {
        e.printStackTrace();

    } finally {
        try {
            if(rs != null) rs.close();
            if(pstmt != null) pstmt.close();
            if(conn != null) conn.close();
        } catch(SQLException e) { }
    }
    return memberList;
```

```
    }
}
```

- com.springstudy.ch02.service

```
public interface MemberService {
    public ArrayList<Member> getMemberList();
}
```

- com.springstudy.ch02.service

```
public class MemberServiceImplConstructor implements MemberService {

    private MemberDAO memberDAO;

    // 스프링 DI 컨테이너로 부터 MemberDAO 객체를 주입받는 생성자
    public MemberServiceImplConstructor(MemberDAO memberDAO) {
        this.memberDAO = memberDAO;
    }

    @Override
    public ArrayList<Member> getMemberList() {
        return memberDAO.getMemberList();
    }
}
```

- src/main/resources/config/props/datasource.properties

```
## 데이터베이스 설정 Properties ##
db.driverClassName=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/spring?useSSL=false&useUnicode=true
&characterEncoding=utf8
db.username=root
db.password=12345678
```

- src/main/resources/config/MemberBeanConstructorContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
```

```
        http://www.springframework.org/schema/context/spring-context-4.2.xsd">

    <!-- 프로퍼티 대치 변수 설정자(Property Placeholder Configurer)를 설정한다. -->
    <context:property-placeholder
            location="classpath:config/props/datasource.properties" />

    <!--
        MemberDAO 타입의 Bean을 선언하고 DriverManagerDataSource 객체를
        와이어링(wiring) 한다. 와이어링이란 객체간의 관계를 형성하는 것을 말한다.
        스프링에서 객체간의 관계를 형성하는 것은 의존성 주입을 의미하며 여기서는
        DriverManagerDataSource 객체를 MemberDaoImpl 클래스의
        생성자를 이용해  스프링 DI 컨테이너가 주입해 준다.
    -->
    <bean id="memberDAO" class="com.springstudy.ch02.dao.MemberDAOImpl">
        <constructor-arg ref="dataSource" />
    </bean>

    <!--
        MemberService 타입의 Bean을 선언하고 MemberDAO 타입의 객체를
        MemberServiceImplConstructor 클래스의 생성자를 통해 주입한다.
        c 네임스페이스를 이용해 MemberDAO 객체를 생성자 주입받고 있다.
    -->
    <bean id="memberService"
        class="com.springstudy.ch02.service.MemberServiceImplConstructor"
        c:memberDAO-ref="memberDAO"/>

    <!--
        스프링 JDBC의 DriverManagerDataSource 타입의 Bean을 선언하고
        위에서 프로퍼티 대치 변수 설정자로 지정한 properties 파일로 부터 읽어온
        데이터를 대치 변수를 사용해 dataSource의 각 프로퍼티에 지정하고 있다.
     -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
        <property name="driverClassName" value="${db.driverClassName}" />
        <property name="url" value="${db.url}" />
        <property name="username" value="${db.username}" />
        <property name="password" value="${db.password}" />
    </bean>
</beans>
```

- com.springstudy.ch02.main

```
// 생성자 주입(Constructor Injection)
public class MemberBeanConstructorIndex {
```

```
    public static void main(String[] args) {

        // 스프링 설정 파일에 정의한 Bean을 생성해 빈 컨테이너에 담는다.
        ApplicationContext ctx = new GenericXmlApplicationContext(
                "classpath:config/MemberBeanConstructorContext.xml");

        // 빈 컨테이너에서 "memberService"란 id 또는 name을 가진 빈 객체를 얻어온다.
        MemberService service = (MemberService) ctx.getBean("memberService");

        // MemberService 타입의 객체를 통해 회원 리스트를 가져와 출력한다.
        ArrayList<Member> memberList = service.getMemberList();
        System.out.println("## 회원 리스트 - 생성자 주입 ##");
        for(Member m : memberList) {
            System.out.println(m);
        }
    }
}
```

2.2.2 세터 주입(Setter Injection)

- com.springstudy.ch02.dao

앞의 생성자 주입에서 작성한 MemberDAOImpl 클래스에 Setter 주입(또는 Property 주입 이라고 함)에 필요한 기본 생성자와 DriverManagerDataSource를 주입할 수 있는 setter 메서드를 추가하자.

```
    // 기본 생성자 - Property 주입할 때 필요함
    public MemberDAOImpl() {}

    // 스프링 DI 컨테이너로 부터 DriverManagerDataSource 객체를 주입받는 setter 메서드
    public void setDataSource(DriverManagerDataSource dataSource) {
        this.dataSource = dataSource;
    }
```

- com.springstudy.ch02.service

```
public class MemberServiceImplProperty implements MemberService {

    private MemberDAO memberDAO;

    // 스프링 DI 컨테이너로 부터 MemberDAO 객체를 주입받는 setter 메서드
    public void setMemberDAO(MemberDAO memberDAO) {
        this.memberDAO = memberDAO;
    }
```

```
    @Override
    public ArrayList<Member> getMemberList() {
        return memberDAO.getMemberList();
    }
}
```

- src/main/resources/config/MemberBeanPropertyContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd">

    <!-- 프로퍼티 대치 변수 설정자(Property Placeholder Configurer)를 설정한다. -->
    <context:property-placeholder
            location="classpath:config/props/datasource.properties" />

    <!--
        DriverManagerDataSource 객체를 MemberDaoImpl 클래스의
        setDataSource()를 이용해  setter 주입한다.
     -->
    <bean id="memberDAO" class="com.springstudy.ch02.dao.MemberDAOImpl">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!--
        MemberService 타입의 Bean을 선언하고 MemberDAO 타입의 객체를
        MemberServiceImplProperty 클래스의 setMemberDAO()를 이용해 setter 주입한다.
        property 요소를 사용하지 않고 p 네임스페이스를 이용해 memberDAO
        속성에 MemberDAOImpl 객체를 주입하고 있다.
    -->
    <bean id="memberService"
        class="com.springstudy.ch02.service.MemberServiceImplProperty"
        p:memberDAO-ref="memberDAO" />

    <!--
        스프링 JDBC의 DriverManagerDataSource 타입의 Bean을 선언하고
        위에서 프로퍼티 대치 변수 설정자로 지정한 properties 파일로 부터 읽어온
        데이터를 대치 변수를 사용해 각 프로퍼티에 지정하고 있다.
        p 네임스페이스를 이용해 dataSource의 각 속성에 데이터를 지정하고 있다.
```

```
    -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        p:driverClassName="${db.driverClassName}"
        p:url="${db.url}"
        p:username="${db.username}"
        p:password="${db.password}" />
</beans>
```

- com.springstudy.ch02.main

```
// 세터 주입(Setter Injection)
public class MemberBeanPropertyIndex {

    public static void main(String[] args) {

        // 스프링 설정 파일에 정의한 Bean을 생성해 빈 컨테이너에 담는다.
        ApplicationContext ctx = new GenericXmlApplicationContext(
                "classpath:config/MemberBeanPropertyContext.xml");

        // 빈 컨테이너에서 "memberService"란 id 또는 name을 가진 Bean 객체를 얻어온다.
        MemberService service = (MemberService) ctx.getBean("memberService");

        // MemberService 타입 객체를 통해 회원 리스트를 가져와 출력한다.
        ArrayList<Member> memberList = service.getMemberList();
        System.out.println("## 회원 리스트 - setter 주입 ##");
        for(Member m : memberList) {
            System.out.println(m);
        }
    }
}
```

[연습문제 2-1] 메이븐 프로젝트를 생성하고 아래와 같은 애플리케이션을 구현해 보자.

1. springstudy-ch02exam 메이븐 프로젝트를 생성하고 수업시간에 제공한 springstudy-ch02 프로젝트의 SQL 폴더에 있는 product.sql을 사용해 spring 데이터베이스에 테이블을 생성하고 데이터를 추가 하시오.
2. 테이블 한 행의 데이터를 저장하고 관리하는 Product 도메인 클래스를 작성하시오.
3. ProductDAO 인터페이스를 생성하고 이 인터페이스에 테이블에서 모든 상품 정보를 읽어와 반환하는 getProductList() 추상 메서드를 정의 하시오.
4. ProductService 인터페이스를 생성하고 이 인터페이스에 ProductDAO의 getProductList()를 사용해 비즈니스 로직을 처리하는 getProductList() 추상 메서드를 정의하고 있다.
5. ProductDAO 인터페이스와 ProductService 인터페이스를 구현하는 클래스를 작성하고 스프링프레임워크를 이용해 이 두 인터페이스의 구현체를 스프링 Bean으로 등록하여 클래스

간에 관계(와이어링 또는 의존 객체 주입)를 설정하시오.

6. 의존객체 주입은 생성자 주입 방식과 세터 주입 방식으로 나눠서 구현하시오.
   - ProductDAO 구현체에서 필요한 DriverManagerDataSource 타입의 객체 주입
   - ProductService 구현체에서 필요한 ProductDAO 타입의 객체 주입

2.3 Annotation을 이용한 DI

우리는 앞에서 스프링 Bean 설정 파일인 XML 파일에 여러 클래스를 스프링 Bean으로 정의하고 이 객체를 필요로 하는 클래스에서 생성자 또는 세터 메서드를 통해 스프링 DI 컨테이너로부터 주입받는 방법에 대해 알아보았다. 이번에는 조금 더 간편한 방법인 Annotation을 사용해 스프링 Bean으로 등록하고 필요한 객체를 스프링으로부터 주입받는 방법에 대해 알아보자.

Annotation 방식을 사용한다고 해서 스프링 Bean 설정 파일이 전혀 필요 없는 것은 아니다.

Annotation 방식을 사용하기 위해서는 스프링 Bean 설정 파일을 통해 Annotation 방식을 사용한다고 알려줘야 한다. 또한 우리가 정의하지 못하는 클래스 즉 외부에서 정의한 클래스는 그 소스에 Annotation을 적용할 수 없기 때문에 스프링 Bean 설정 파일에 Bean으로 등록을 해야 스프링 DI 컨테이너로부터 필요한 객체를 주입받을 수 있다.

Annotation을 사용한 방법도 생성자 주입과 셋터 메서드 주입방식으로 객체를 주입받을 수 있기 때문에 XML 설정에서 사용했던 클래스와 동일한 클래스를 사용할 것이다.

먼저 springclass-ch02 프로젝트에 com.springstudy.ch02.annotation 패키지를 만들고 앞에서 사용했던 com.springstudy.ch02.dao 패키지의 MemberDAOImpl 클래스를 이 패키지로 복사하여 가져온다. 또한 com.springstudy.ch02.service 패키지의 MemberServiceImplConstructor클래스와 MemberServiceImplProperty 클래스를 복사해 이 패키지로 가져온다. 그리고 아래 교안을 참고해 복사해 온 클래스에 Annotation을 적용해 스프링 Bean으로 설정하고 클래스 간의 관계를 정의하자.

2.3.1 Annoataion을 이용한 생성자 주입

- com.springstudy.ch02.annotation

```
// 이 클래스가 스프링 컴포넌트임을 선언하고 "memberDAOAnno"라는 빈 이름을 지정한다.
//@Component("memberDAOAnno")

/* @Component 대신에 이를 확장한 Annotation인 @Controller, @Service,
 * @Repository 등을 사용해 클래스의 용도에 따라 스프링 빈 임을 정의할 수 있다.
 *
 * 아래는 @Repository Annotation을 이용해 데이터 저장소 계층의 스프링
 * 빈임을 선언하고 있다. 빈의 이름은 별도로 부여하지 않았으므로 스프링이 알아서
 * 클래스 이름을 카멜 케이싱한 "memberDAOImpl"의 이름을 부여한다.
 **/
@Repository
public class MemberDAOImpl implements MemberDAO {

    private Connection conn;
    private PreparedStatement pstmt;
```

```
private ResultSet rs;
private DriverManagerDataSource dataSource;

/* 스프링 DI 컨테이너로 부터 DriverManagerDataSource 객체를 주입받는 생성자
 * 아래와 같이 @Autowired를 생성자에 붙이면 이 생성자를 통해 객체를 주입 받을 수 있다.
 **/
@Autowired
public MemberDAOImpl(DriverManagerDataSource dataSource) {
    this.dataSource = dataSource;
}

@Override
public ArrayList<Member> getMemberList() {

    String selectAllMember = "SELECT * FROM member;";
    ArrayList<Member> memberList = null;

    try {
        conn = dataSource.getConnection();
        pstmt = conn.prepareStatement(selectAllMember);
        rs = pstmt.executeQuery();

        memberList = new ArrayList<Member>();

        while(rs.next()) {

            Member m = new Member();
            m.setId(rs.getString("id"));
            m.setName(rs.getString("name"));
            m.setPass(rs.getString("pass"));
            m.setAge(rs.getInt("age"));
            m.setEmail(rs.getString("email"));
            m.setRegDate(rs.getTimestamp("reg_date"));

            memberList.add(m);
        }

    } catch(SQLException e) {
        e.printStackTrace();

    } finally {
        try {
            if(rs != null) rs.close();
            if(pstmt != null) pstmt.close();
            if(conn != null) conn.close();
```

```
            } catch(SQLException e) { }
        }
        return memberList;
    }
}
```

- com.springstudy.ch02.annotation

```
//@Component Annotation을 이용해 이 클래스가 스프링 컴포넌트임을 선언한다.
//@Component("memberServiceImpl")

/* @Component 대신에 이를 확장한 Annotation인 @Controller, @Service,
 * @Repository 등을 사용해 스프링 빈 임을 정의할 수 있다.
 *
 * 아래는 @Service Annotation을 이용해 이 클래스가 서비스 계층의 스프링 빈임을
 * 선언하고 있다. 빈의 이름은 별도로 부여하지 않았으므로 스프링이 알아서 클래스 이름을
 * 카멜 케이싱해 "memberServiceImplConstructor"이라는 빈 이름을 부여한다.
 **/
@Service
public class MemberServiceImplConstructor implements MemberService {

    MemberDAO memberDAO;

    /* 스프링 설정 파일에 <context:component-scan>의 base-package에
     * 지정한 패키지를 기준으로 스캔하여 @Component, @Controller, @Service,
     * @Repository 애노테이션이 붙은 클래스의 객체를 생성하고 MemberDAO 타입의
     * 객체를 아래와 같이 @Autowired 애노테이션이 붙은 생성자에 주입해 준다.
     **/
    @Autowired
    public MemberServiceImplConstructor(MemberDAO memberDAO) {
        this.memberDAO = memberDAO;
    }

    @Override
    public ArrayList<Member> getMemberList() {
        return memberDAO.getMemberList();
    }
}
```

- src/main/resources/config/annotation/AnnotationConstructorContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
```

```
xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd">

<!-- 프로퍼티 대치 변수 설정자(Property Placeholder Configurer)를 설정한다. -->
<context:property-placeholder
        location="classpath:config/props/datasource.properties"/>

<!--
    스프링에서 annotation 기반 bean wiring을 사용하겠다는 설정으로
    프로퍼티, 메서드 또는 생성자에 자동으로 와이어링 됨을 의미한다.
    이 설정은 스프링  빈 설정 파일에서 <property>, <constructor-arg> 설정은
    생략할 수 있지만 <bean> 태그를 사용해 명시적으로 bean을 정의해야
    스프링이 annotaion 기반으로 와이어링 할 수 있다는 의미이다.
-->
<context:annotation-config />

<!--
    스프링 설정 파일에 지정한 base-package를 기준으로 컴포넌트(클래스)를 스캔하여
    자동으로 bean을 생성해 주는 설정으로 <context:annotation-config />가
    수행하는 모든 것을 수행하며 여기에 더해 스프링이 자동으로 bean을 스캔하여
    선언하고 스프링 빈 설정 파일에서 명시적으로 <bean> 태그를 사용하지 않아도
    정상적으로 annotation 기반으로 와이어링 할 수 있다는 의미이다.

    <context:component-scan />은 <context:annotation-config />가
    가지는 모든 기능을 포함하고 있기 때문에 <context:component-scan />을
    사용하면 <context:annotation-config />는 설정할 필요가 없다.
-->
<context:component-scan
    base-package="com.springstudy.ch02.annotation" />

<!--
    스프링 JDBC의 DriverManagerDataSource 타입의 Bean을 선언하고
    위에서 프로퍼티 대치 변수 설정자로 지정한 properties 파일로 부터 읽어온
    데이터를 대치 변수를 사용해 각 프로퍼티에 지정하고 있다.
    p 네임스페이스를 이용해 dataSource의 각 속성에 데이터를 지정하고 있다.
    @Component로 설정된 클래스에 @Autowired로 지정된 DriverManagerDataSource
    타입의 필드나 setter 메서드 또는 생성자의 인수에 주입된다.
 -->
<bean id="dataSource01"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    p:driverClassName="${db.driverClassName}"
    p:url="${db.url}"
```

```
        p:username="${db.username}"
        p:password="${db.password}" />
</beans>
```

우리의 실습 프로젝트에 아래 패키지를 만들고 아래 교안을 참고해 스프링 빈 컨테이너를 생성하여 프로젝트를 실행해 보자.

- com.springstudy.ch02.main

```
// Annotaion 생성자 주입 main
public class MemberAnnotationConstructorIndex {

    public static void main(String[] args) {


        /* 지금은 자동으로 DI 컨테이너가 생성되지 않기 때문에 스프링이 제공하는
         * DI 컨테이너 역할을 하는 클래스의 인스턴스를 우리가 수동으로 생성하고
         * 빈 설정 파일을 인수로 지정해야 한다. 그래야 스프링 DI 컨테이너가 생성되면서
         * 빈 설정 파일을 읽어 그 설정 파일에 빈으로 정의된 클래스를 객체로 생성한다.
         *
         * Bean 정의 파일에 component-scan의 base-package에 지정한 패키지를
         * 기준으로 스캐닝 하여 @Component, @Controller, @Service, @Repository
         * Annotation이 붙은 클래스의 객체를 생성하고 @Autowired, @Resource 등의
         * Annotation이 붙은 생성자나 세터 메서드를 이용해 필요한 객체를 주입하게 된다.
         **/
        ApplicationContext ctx = new GenericXmlApplicationContext(
                "classpath:config/annotation/AnnotationConstructorContext.xml");

        /* 빈 컨테이너에서 MemberService 타입(MemberService를 구현한 구현체)의
         * "memberServiceImplConstructor"라는 Bean의 이름을 가진 객체를 구한다.
         **/
        MemberService service = ctx.getBean(
                "memberServiceImplConstructor", MemberService.class);

        // MemberService 타입의 객체를 통해 회원 리스트를 가져와 출력한다.
        ArrayList<Member> memberList = service.getMemberList();
        System.out.println("## 회원 리스트 - Annotation 생성자 주입 ##");
        for(Member m : memberList) {
            System.out.println(m);
        }
    }
}
```

2.3.2 Annotation을 이용한 세터 주입

Annotation을 이용한 생성자 주입에서 MemberDAOImpl 클래스는 DriverManagerDataSource 객체를 생성자를 통해 받기 위해서 DriverManagerDataSource 타입의 파라미터를 가지는 생성자를 정의하고 이 생성자 위에 @Autowired 애노테이션을 붙여 DriverManagerDataSource 객체를 주입 받아 사용하였다. 이번 예제는 setter 메서드를 이용해서 DriverManagerDataSource 객체를 주입 받는 예제로 앞에서 사용한 생성자는 필요하지 않기 때문에 이 생성자에 붙였던 @Autowired 애노테이션은 주석 처리한다. 그리고 기본 생성자를 추가하고 DriverManagerDataSource 타입의 파라미터를 가지는 setter 메서드를 추가해서 이 setter 메서드 위에 @Autowired 애노테이션을 붙여 DriverManagerDataSource 객체가 setter 메서드로 주입될 수 있도록 설정하면 된다.

- com.springstudy.ch02.annotation

```
// 이 클래스가 스프링 컴포넌트임을 선언하고 "memberDAOAnno"라는 빈 이름을 지정한다.
//@Component("memberDAOAnno")

/* @Component 대신에 이를 확장한 Annotation인 @Controller, @Service,
 * @Repository 등을 사용해 클래스의 용도에 따라 스프링 빈 임을 정의할 수 있다.
 *
 * 아래는 @Repository Annotation을 이용해 데이터 저장소 계층의 스프링
 * 빈임을 선언하고 있다. 빈의 이름은 별도로 부여하지 않았으므로 스프링이 알아서
 * 클래스 이름을 카멜 케이싱한 "memberDAOImpl"의 이름을 부여한다.
 **/
@Repository
public class MemberDAOImpl implements MemberDAO {

    private Connection conn;
    private PreparedStatement pstmt;
    private ResultSet rs;
    private DriverManagerDataSource dataSource;

    /* 스프링 DI 컨테이너로 부터 DriverManagerDataSource 객체를 주입받는 생성자
     * 아래와 같이 @Autowired를 생성자에 붙이면 이 생성자를 통해 객체를 주입 받을 수 있다.
     **/
    // setter 주입일 때는 아래의 애노테이션을 주석처리한다.
    // @Autowired
    public MemberDAOImpl(DriverManagerDataSource dataSource) {
        this.dataSource = dataSource;
    }

    // 기본 생성자를 새로 추가한다.
    public MemberDAOImpl() {}

    /* 스프링 DI 컨테이너로 부터 DriverManagerDataSource 객체를 주입받는 setter 메서드
     * DriverManagerDataSource 객체를 주입 받을 수 있는 setter 메서드를 새로 추가한다.
     *
```

```
 * 이 클래스의 기본 생성자를 통해 객체를 생성하고 setter 메서드로 주입되므로
 * 이 클래스에 반드시 기본 생성자가 존재해야 한다.
 **/
@Autowired
public void setDataSource(DriverManagerDataSource dataSource) {
    this.dataSource = dataSource;
}

@Override
public ArrayList<Member> getMemberList() {

    String selectAllMember = "SELECT * FROM member;";
    ArrayList<Member> memberList = null;

    try {
        conn = dataSource.getConnection();
        pstmt = conn.prepareStatement(selectAllMember);
        rs = pstmt.executeQuery();

        memberList = new ArrayList<Member>();

        while(rs.next()) {

            Member m = new Member();
            m.setId(rs.getString("id"));
            m.setName(rs.getString("name"));
            m.setPass(rs.getString("pass"));
            m.setAge(rs.getInt("age"));
            m.setEmail(rs.getString("email"));
            m.setRegDate(rs.getTimestamp("reg_date"));

            memberList.add(m);
        }

    } catch(SQLException e) {
        e.printStackTrace();

    } finally {
        try {
            if(rs != null) rs.close();
            if(pstmt != null) pstmt.close();
            if(conn != null) conn.close();
        } catch(SQLException e) { }
    }
    return memberList;
```

```
    }
}
```

- com.springstudy.ch02.annotation

```
// 이 클래스가 스프링 컴포넌트임을 선언하고 "memberService"라는 빈 이름을 지정한다.
// @Component("memberService")

/* @Component 대신에 이를 확장한 Annotation인 @Controller, @Service,
 * @Repository 등을 사용해 스프링 빈 임을 정의할 수 있다.
 *
 * 아래는 @Service Annotation을 이용해 이 클래스가 서비스 계층의 스프링 빈임을
 * 선언하고 "memberService"라는 빈의 이름을 지정하고 있다.
 **/
@Service("memberService")
public class MemberServiceImplProperty implements MemberService {

    private MemberDAO memberDAO;

    /* 스프링 설정 파일에 <context:component-scan>의 base-package에
     * 지정한 패키지를 기준으로 스캔하여 @Component, @Controller, @Service,
     * @Repository 애노테이션이 붙은 클래스의 객체를 생성하고 MemberDAO 타입의
     * 객체를 아래와 같이 @Autowired 애노테이션이 붙은 setter 메서드에 주입해 준다.
     *
     * 이 동작은 클래스의 기본 생성자를 통해 객체를 생성하고 setter 메서드로 주입한다.
     * 그러므로 이 클래스에 기본 생성자가 반드시 존재해야 하지만 다른 생성자가
     * 정의되어 있지 않으므로 컴파일러에 의해 기본 생성자가 자동으로 만들어 진다.
     **/
    @Autowired
    public void setMemberDAO(MemberDAO memberDAO) {
        this.memberDAO = memberDAO;
    }

    @Override
    public ArrayList<Member> getMemberList() {
        return memberDAO.getMemberList();
    }
}
```

- src/main/resources/config/annotation/AnnotationPropertyContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
```

```
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd">

    <!-- 프로퍼티 대치 변수 설정자(Property Placeholder Configurer)를 설정한다. -->
    <context:property-placeholder
            location="classpath:config/props/datasource.properties" />

    <!--
        스프링 설정 파일에 지정한 base-package를 기준으로 컴포넌트(클래스)를 스캔하여
        @Componet가 붙은 클래스의 bean을 생성하고 @Autowired가 붙은 필드나
        메서드를 찾아 그 필드나 메서드의 매개변수 타입과 일치하는 bean을 생성하여 주입해
        주는 설정으로 스프링이 자동으로 bean을 발견하여 선언하고 <bean> 태그를
        사용하지 않아도 정상적으로 annotation 기반으로 와이어링 할 수 있다는 의미이다.

        <context:component-scan />는 <context:annotation-config />가
        가지는 모든 기능을 포함하고 있기 때문에 <context:component-scan />을
        사용하면 <context:annotation-config />는 설정할 필요가 없다.
    -->
    <context:component-scan base-package="com.springstudy.ch02.annotation" />

    <!--
        스프링 JDBC의 DriverManagerDataSource 타입의 Bean을 선언하고
        위에서 프로퍼티 대치 변수 설정자로 지정한 properties 파일로 부터 읽어온
        데이터를 대치 변수를 사용해 각 프로퍼티에 지정하고 있다.
        p 네임스페이스를 이용해 dataSource의 각 속성에 데이터를 지정하고 있다.
        @Component로 설정된 컴포넌트에 DriverManagerDataSource 타입의
        필드나 setter 메서드 또는 생성자의 인수에 주입된다.
     -->
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        p:driverClassName="${db.driverClassName}"
        p:url="${db.url}"
        p:username="${db.username}"
        p:password="${db.password}" />
</beans>
```

- com.springstudy.ch02.main

```
// Annotation 세터 주입 main
public class MemberAnnotationPropertyIndex {

    public static void main(String[] args) {
```

```
        /* 지금은 자동으로 DI 컨테이너가 생성되지 않기 때문에 스프링이 제공하는
         * DI 컨테이너 역할을 하는 클래스의 인스턴스를 우리가 수동으로 생성하고
         * 빈 설정 파일을 인수로 지정해야 한다. 그래야 스프링 DI 컨테이너가 생성되면서
         * 빈 설정 파일을 읽어 그 설정 파일에 빈으로 정의된 클래스를 객체로 생성한다.
         *
         * Bean 정의 파일에 component-scan의 base-package에 지정한 패키지를
         * 기준으로 스캐닝 하여 @Component, @Controller, @Service, @Repository
         * Annotation이 붙은 클래스의 객체를 생성하고 @Autowired, @Resource 등의
         * Annotation이 붙은 생성자나 세터 메서드를 이용해 필요한 객체를 주입하게 된다.
         **/
        ApplicationContext ctx = new GenericXmlApplicationContext(
                "classpath:config/annotation/AnnotationPropertyContext.xml");

        /* 빈 컨테이너에서 MemberService 타입(MemberService를 구현한 구현체)의
         * "memberService"라는 Bean의 이름을 가진 객체를 구한다.
         **/
        MemberService service = ctx.getBean("memberService", MemberService.class);

        // MemberService 타입의 객체를 통해 회원 리스트를 가져와 출력한다.
        ArrayList<Member> memberList = service.getMemberList();
        System.out.println("## 회원 리스트 - Annotation setter 주입 ##");
        for(Member m : memberList) {
            System.out.println(m);
        }
    }
}
```

[연습문제 2-2] 다음 요구사항에 부합되는 애플리케이션을 구현해 보자.

1. 앞에서 springstudy-ch02exam 프로젝트를 통해서 XML 파일을 이용해 Bean을 정의하고 클래스 간의 관계(의존객체 주입)를 설정하는 방식을 실습하였으므로 이번에는 Annotation을 활용해 Bean을 정의하고 의존객체를 주입하는 애플리케이션을 구현해 봅시다.
2. Annotation 방식의 의존객체 주입도 생성자 주입과 세터 주입 방식으로 나눠서 구현하시오.
   - ProductDAO 구현체에서 필요한 DriverManagerDataSource 타입의 객체 주입
   - ProductService 구현체에서 필요한 ProductDAO 타입의 객체 주입
3. 새로운 패키지를 추가해 작성하고 패키지 명은 적절한 이름을 부여해 구현하시오.

2.4 Bean의 생명주기(Life Cycle)

스프링프레임워크의 DI 컨테이너는 애플리케이션이 시작될 때 bean이 설정된 XML 파일이나 Annotation이 지정된 클래스 또는 Bean이 정의된 클래스를 스캔해서 객체를 생성하고 의존하는 객체를 주입해 준다. 그리고 애플리케이션이 종료될 때 DI 컨테이너에서 관리되는 객체를 모두 소멸시킨다. 이 과정에서 스프링은 그림 2-4와 같이 bean 객체를 생성하고 의존 관계 설정에 따라서 의존 객체를 주입하는 것 외에도 bean 객체의 초기화 작업과 소멸화 작업에 필요한 여러 가지 작업을 하게 된다. 이 때 스프링이 관리하는 bean 객체의 초기화나 소멸화 작업에 사용되는 메서드를 알려줘야 스프링이 bean을 생성하고 bean의 라이프 사이클을 관리하게 된다. 초기화 메서드와 소멸화 메서드를 설정하는 방법에는 XML 설정과 Annotation을 이용한 방법 그리고 스프링이 제공하는 초기화 및 소멸화 인터페이스를 구현하는 방법이 있다. 인터페이스를 구현하는 방식은 예전에 사용했던 방식으로 요즘은 잘 사용하지 않기 때문에 XML 설정과 Annotation을 이용한 Bean의 라이프 사이클에 대해서 알아 볼 것이다.

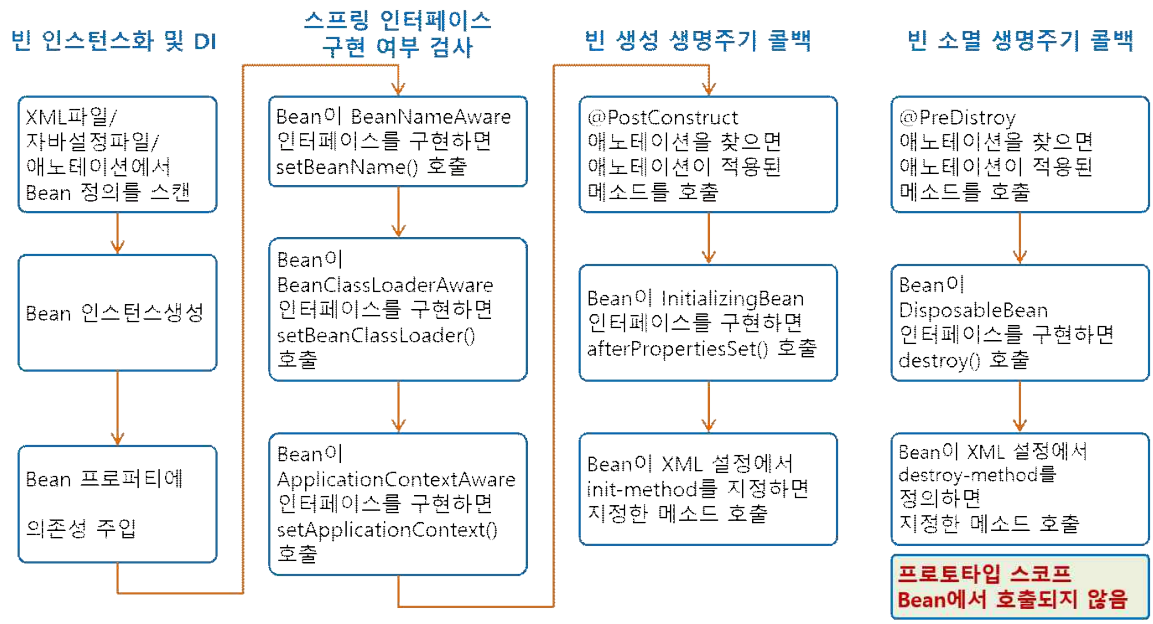

그림 2-4 스프링 Bean의 라이프 사이클

지금까지 우리는 System.out.println() 메서드를 사용해 프로그램 운영 상황에 대한 로깅을 처리하였다. 그러나 이 방법은 간편한 반면 여러 가지 문제점을 가지고 있어 별로 권장되지 않는다. 그래서 이번 예제에 로깅 프레임워크인 log4j을 이용한 로깅을 처리하는 방법에 대해 같이 알아 볼 것이다. 먼저 간단히 log4j에 대해 알아보자. log4j의 가장 큰 특징은 계층(Category) 구조를 가지는 로거라는 것인데 로거(logger)를 사용할 때 카테고리 라는 이름을 붙여 사용한다. 이렇게 함으로써 여러 소스에 지정한 로거들 중에서 필요한 특정 부분만 구분하여 로깅을 처리 할 수 있다.

◈ log4j의 구조

log4j의 구조는 크게 appender, logger, layout 3가지 요소로 구성되고 각각의 요소의 역할과 의미는 다음과 같다.

▶ appender :

logger로부터 전달받은 로깅 메시지를 파일에 기록할지, 콘솔에 출력만 할지 아니면 DB등에 저장할 것인지를 지정하는 매개체 역할을 한다.

▶ logger :

로깅 메시지를 apperder에게 전달하고, 로그 레벨에 따른 로그 출력을 결정하는 역할을 한다.

▶ layout :

apperder로 어디에 출력할 것인지 결정했다면 어떤 형식으로 로그를 출력할 것이지 출력에 대한 layout을 결정하는 역할을 한다.

◈ log4j의 로깅 레벨

어떤 레벨에서 로깅을 할지 결정하는 logger에 설정하는 로깅 레벨은 아래와 같이 6단계의 레벨로 나뉜다. logger에 로깅 레벨을 지정하면 지정한 레벨을 포함한 상위에 존재하는 로깅 레벨 상황이 발생하게 되면 로깅처리 된다. 만약 logger에 로깅 레벨을 WARN으로 지정했다면 WARN 레벨을 포함하여 그 상위 레벨인 ERROR, FATAL 상황에서 로깅처리가 된다.

FATAL : 가장 심각한(치명적인) 에러가 발생했을 때 로깅처리 - 최상위 레벨
ERROR : 일반적인 Exception이나 에러가 발생했을 때 로깅처리
WARN : 에러는 아니지만 주의가 필요한 상황이 발생했을 때 로깅처리
INFO : 일반적인 정보를 출력할 때 로깅처리
DEBUG : 일반적인 정보를 보다 상세히 출력할 때 로깅처리
TRACE : 최하위 레벨의 로깅처리

◈ 로그 파일명 날짜 패턴 포맷(Date Pattern Format)

로그를 기록하는 파일명을 설정하는 것으로 로그 파일명에 날짜 패턴을 지정하면 날짜, 시간 또는 분 단위까지 로그 파일을 분할하여 로그를 기록할 수 있도록 아래와 같이 지정할 수 있다.

날짜 패턴 형식	설 명
'.'yyyy-MM	매달 첫 번째 일에 로그 파일명을 변경하여 기록한다.
'.'yyyy-ww	매주의 시작 일에 로그 파일명을 변경하여 기록한다.
'.'yyyy-MM-dd	매일 자정을 기준으로 로그 파일명을 변경하여 기록한다.
'.'yyyy-MM-dd-a	매일 자정과 정오에 로그 파일명을 변경하여 기록한다.
'.'yyyy-MM-dd-HH	매시간 마다 로그 파일명을 변경하여 기록한다.
'.'yyyy-MM-dd-HH-mm	매분마다 로그 파일명을 변경하여 기록한다.

◈ Appender의 종류

log4j는 아래와 같이 여러 가지 Appender를 지원하고 있지만 ConsoleAppender, DailyRolling Appender, RollingFileAppender가 주로 많이 사용되는 Appender 이다.

Appender	설 명
org.apache.log4j.AsyncAppender	비동기 출력으로 네트워크 전송 등 특수한 용도로 사용되고 다른 Appender와 결합해 사용된다.
org.apache.log4j.ConsoleAppender	stdout, stderr으로 출력
org.apache.log4j.DailyRollingFileAppender	지정한 시간 단위로 파일 출력
org.apache.log4j.varia. ExternallyRolledFileAppender	외부 Roller로 출력
org.apache.log4j.FileAppender	파일로 출력
org.apache.log4j.jdbc.JDBCAppender	데이터베이스로 출력
org.apache.log4j.net.JMSAppender	JMS로 출력
org.apache.log4j.lf5.LF5Appender	LogFactor5라는 스윙 로그 뷰어로 출력
org.apache.log4j.nt.NTEventLogAppender	Windows 이벤트 로그로 출력
org.apache.log4j.varia.NullAppender	아무것도 안함
org.apache.log4j.RollingFileAppender	파일 크기 단위로 파일 출력
org.apache.log4j.net.SMTPAppender	메일로 출력
org.apache.log4j.net.SocketAppender	외부 서버에 Socket으로 출력
org.apache.log4j.net.SocketHubAppende	SocketServer로서 출력
org.apache.log4j.net.SyslogAppender	Unix Syslog로 출력
org.apache.log4j.net.TelnetAppender	telnet으로 출력

◈ 패턴 레이아웃 포맷(Pattern Layout Format)

로그를 어떤 포맷으로 출력할 것인지를 지정하는 출력 포맷으로 layout에는 PatternLayout, HTMLLayout, XMLLayout, SimpleLayout 등이 있으며 PatternLayout이 가장 일반적으로 사용되는 레이아웃으로 아래와 같은 패턴 형식을 지정하여 로그를 출력한다.

형식	설 명
%c	카테고리를 출력 한다. 예) 카테고리가 com.springstudy.ch02.TestClass로 구성 되었다면 %c{3}은 springstudy.ch02.TestClass가 출력 된다.
%C	클래스 명을 출력 한다. 예) 클래스구조가 com.springstudy.ch02.TestClass 처럼 되어있다면 %C{2}는 ch02.TestClass 가 출력 된다.
%d	로깅 이벤트가 발생한 시간을 출력한다. %d{HH:mm:ss, SSS}, %d{yyyy MMM dd HH:mm:ss, SSS}와 같은 형태로 포맷을 지정하며 SimpleDateFormat을 기준으로 포맷팅 하여 사용하면 된다.
%p	debug, info, warn, error, fatal 등의 priority를 출력한다.

%m	지정한 로그 메시지를 출력한다.
%l	로깅이 발생한 caller의 정보와 소스 코드에서 몇 번째 라인인지를 출력한다. 로그가 발생한 위치의 LocationInfo 정보를 출력한다. 아래 %F, %L, %M 정보를 합친 것이라고 보면 되는데, JVM statck을 참조하므로, 특정 JVM에서 정상 작동하지 않을 수도 있고, overhead가 심한 편이라서 잘 사용하지 않는다.
%F	로깅이 발생한 프로그램 소스 파일명을 출력한다.
%L	로깅이 발생한 caller의 소스 코드에서 몇 번째 라인인지를 출력한다.
%M	로깅이 발생한 method 이름을 출력한다.
%%	% 표시를 출력한다.
%n	플랫폼 종속적인 개행 문자가 출력된다. \r\n 또는 \n 으로 사용한다.
%r	애플리케이션 시작부터 로깅이 발생한 시점의 시간(milliseconds)을 출력한다.
%t	로그이벤트가 발생된 쓰레드의 이름을 출력 한다.
%x	로깅이 발생한 thread와 관련된 NDC(nested diagnostic context)를 출력한다.
%X	로깅이 발생한 thread와 관련된 MDC(mapped diagnostic context)를 출력한다.

◈ log4j의 로깅 환경 설정 방법

log4j의 로깅 환경 설정 방법은 소스코드 내부에 하드코딩 하는 방법과 설정파일(.properties 또는 .xml)을 사용 하는 방법이 있으며 여기서는 properties와 xml 파일에 설정하는 방법을 사용하여 로깅 하는 방법에 대해 알아 볼 것이다. properties나 xml 파일에 로깅 환경을 설정 하는 방법은 웹 애플리케이션인 경우 WEB-INF 바로 아래에 log4j.xml이나 log4j.properties 파일을 작성해야 하고, 자바(Spring) 애플리케이션인 경우 클래스 패스 바로 아래에 log4j.xml 이나 log4j.properties 파일을 작성하면 된다. 우리는 log4j.xml이나 log4j.properties 파일을 리소스로 취급할 것이므로 java/main/resources 폴더 바로 아래에 작성하면 된다.

만약 클래스 패스에 log4.j.xml과 log4.properties 두 가지 설정 파일이 존재하면 log4j.xml이 우선적으로 적용된다.

log4j를 활용한 로깅 처리에서 properties 설정 방식 보다 xml 설정 방식이 더 많이 사용되므로 properties 설정 방식은 아래 예를 참고하고 우리는 xml 설정 방식을 이용해 로깅을 처리하는 방식을 사용할 것이다.

▶ log4j.properties 작성예

```
## 로그를 콘솔로 출력하는 Appender 지정 ##
log4j.appender.stdout=org.apache.log4j.ConsoleAppender

## Target은 옵션이기 때문에 생략이 가능하다 ##
## log4j.appender.stdout.Target=System.out

## 로그를 constructor.log 파일로 출력하는 FileAppender 지정 ##
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=src/main/resources/constructor.log

## 로그 출력 레이아웃을 지정하고 로그 출력 패턴을 지정 ##
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
```

```
log4j.appender.stdout.layout.ConversionPattern=%p - %C{1}.%M{%L} - %m%n

## 시스템(또는 톰캣서버)이 재시작 되도 파일이 리셋 되지 않게 설정 ##
log4j.appender.file.Append=true
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%d]%p - %C{1}.%M{%L} - %m%n

## 루트 카테고리 설정 ##
log4j.rootLogger=WARN, file

## 로그 카테고리 지정 ##
log4j.logger.org.apache=WARN, stdout
log4j.logger.org.springframework=WARN, stdout
log4j.logger.com.springstudy.ch02=DEBUG, stdout
```

▶ log4j.xml 작성예

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

    <!-- 로그를 콘솔로 출력하는 Appender를 지정한다. -->
    <appender name="stdout" class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="%p - %C{1}.%M{%L} - %m%n" />
        </layout>
    </appender>

    <!--로그를 property.log 파일로 출력하는 Appender를 지정한다. -->
    <appender name="file" class="org.apache.log4j.FileAppender">
        <!-- param 태그가 layout 태그 보다 먼저 기술되어야 한다. -->
        <param name="File" value="src/main/resources/property.log" />
        <param name="Append" value="true" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern"
                value="[%d]%p - %C{1}.%M{%L} - %m%n" />
        </layout>
    </appender>

    <!-- 로그 카테고리를 지정한다. -->
    <logger name="org.apache">
        <level value="WARN" />
    </logger>
    <logger name="org.springframework">
```

```
        <level value="WARN" />
    </logger>

    <!-- additivity=false는 로그의 중복 출력을 막는 역할을 한다. -->
    <logger name="com.springstudy.ch02" additivity="false">
        <level value="DEBUG" />
        <appender-ref ref="stdout" />
    </logger>

    <!-- 루트 카테고리를 설정한다. -->
    <root>
        <level value="INFO" />
        <appender-ref ref="stdout"/>
        <appender-ref ref="file" />
    </root>
</log4j:configuration>
```

◈ 로깅처리를 위한 클래스 멤버설정과 메서드

▶ 클래스 멤버 설정하기

```
// log4j가 제공하는 Logger 클래스를 이용한 로깅 처리
protected static Logger logger = Logger.getLogger("test.jsp");
protected static Logger logger = Logger.getLogger(test.class);

// commons-logging이 제공하는 Log 클래스를 이용한 로깅처리
protected static Log logger = LogFactory.getLog("test.jsp");
protected static Log logger = LogFactory.getLog(test.class);
```

▶ 로깅 메서드

각 레벨 마다 아래와 같이 두 개의 메서드로 오버로딩 되어 있다.

```
debug(Object message);
debug(Object message, Throwable t);
```

▶ 로깅 레벨을 체크하여 로깅하기

다음은 Debug 레벨에서 로그를 출력하는 예이다.

```
if(logger.isDebugEnabled()) {
    logger.debug("main() 시작");
}

~ ~  애플리케이션 코드 ~ ~

if(logger.isDebugEnabled()) {
    logger.debug("main() 종료");
}
```

1) Bean 초기화 메서드와 소멸화 메서드(XML 설정)

- com.springstudy.ch02.lifecycle

```
public class BeanLifeCycle01 {

    /* 로깅 작업을 위해 log4j 라이브러리가 제공하는 Logger 클래스를
     * 통해 Logger 객체를 얻는다. getLogger()의 인수로 로깅 작업을
     * 할 클래스 타입을 지정하거나 클래스 명을 String으로 지정할 수 있다.
     **/
    protected static final Logger logger =
            Logger.getLogger(BeanLifeCycle01.class);

    /* 아래와 같이 commons-logging 라이브러리를 log4j 설정에 맞게 로깅
     * 처리를 할 수도 있다. org.apache.commons.logging.LogFactory
     * 클래스를 통해 Log 객체를 얻는다. getLog()의 인수로 로깅 작업을
     * 할 클래스 타입을 지정하거나 클래스 명을 String으로 지정할 수 있다.
     **/
    /*
    protected static final Log logger =
            LogFactory.getLog(BeanLifeCycle01.class);
    */

    private String name;
    private int age;

    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }

    /* 초기화 콜백 메서드 - bean 설정에서 이 메서드를 초기화 콜백 메서드로 지정했다.
     * 스프링이 이 클래스의 기본 생성자로 bean을 생성하고 프로퍼티에 의존성을
     * 주입한 직후에 init-method에 지정된 이 메서드가 호출되는 순서로 진행된다.
     **/
    public void beanInit() {

        // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
        if(logger.isDebugEnabled()) {
            logger.debug("BeanLifeCycle01 - initMethod 시작");
            logger.debug("name : " + name + ", age : " + age);
        }
```

```
        System.out.println("애플리케이션 초기화에 필요한 작업을 수행");

        // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
        if(logger.isDebugEnabled()) {
            logger.debug("BeanLifeCycle01 - initMethod 종료");
        }
    }

    /* 소멸화 콜백 메서드 - bean 설정에서 이 메서드를 소멸화 콜백 메서드로 지정했다.
     * 스프링이 이 클래스의 싱글턴 인스턴스를 소멸시키기 직전에 <bean> 태그에
     * destroy-method 속성에 지정된 이 메서드를 호출해 준다.
     * 소멸화 콜백 메서드의 이름이 정해져 있는 것이 아니라 소멸화 메서드로 사용할
     * 메서드 이름을 <bean> 태그의 destroy-method에 지정해 주면 된다.
     * 소멸 메서드에는 주로 사용한 자원을 반납하거나 닫는 코드를 기술한다.
     **/
    public void beanDestroy() {
        if(logger.isDebugEnabled()) {
            logger.debug("BeanLifeCycle01 - Destroy 시작");
        }

        System.out.println("애플리케이션 소멸화에 필요한 작업을 수행");

        if(logger.isDebugEnabled()) {
            logger.debug("BeanLifeCycle01 - Destroy 종료");
        }
    }

    public static void main(String[] args) {

        // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
        if(logger.isDebugEnabled()) {
            logger.debug("main() 시작");
        }

        // GenericXmlApplicationContext는 ApplicationContext의 구현체 이다.
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext(
                "classpath:config/lifecycle/BeanLifeCycle01.xml");

        BeanLifeCycle01 bean1 = (BeanLifeCycle01) ctx.getBean(
                        "beanInit01", BeanLifeCycle01.class);

        /* close() 메서드는 현재 실행중인 DI 컨테이너를 종료하는 메서드로
         * 스프링은 빈 컨테이너가 종료되기 직전에 모든 싱글턴 bean을 소멸 시킨다.
         * 이 때 bean이 가지고 있는 소멸화 메서드를 호출한다.
```

```
         **/
        ctx.close();

        // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
        if(logger.isDebugEnabled()) {
            logger.debug("main() 종료");
        }
    }
}
```

- src/main/resources/config/lifecycle/BeanLifeCycle01.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd" >

    <!--
        <bean> 태그의 init-method 속성은 초기화 콜백 메서드를 지정하는 속성으로
        스프링이 bean 설정을 마친 후 바로 init-method에 지정된 초기화 콜백 메서드를
        호출해야 한다고 스프링에게 알려주는 역할을 한다. 또한 destroy-method 속성은
        스프링이 bean을 소멸시키기 직전에 destroy-method에 지정된 소멸화 메서드를
        호출해야 한다고 스프링에게 알려주는 역할을 한다.
        init-method나 destroy-method에 지정하는 메서드는 정해져 있는 것이 아니라
        초기화 또는 소멸화 메서드로 사용할 메서드를 작성하고 그 이름을 지정하면 된다.
    -->
    <bean id="beanInit01"
        class="com.springstudy.ch02.lifecycle.BeanLifeCycle01"
        init-method="beanInit" destroy-method="beanDestroy">
        <property name="name" value="이순신" />
        <property name="age" value="35" />
    </bean>
</beans>
```

2) Bean 초기화 메서드와 소멸화 메서드(Annotation)

- com.springstudy.ch02.lifecycle

```
@Component
public class AnnotationLifeCycle01 {

    /* 로깅 작업을 위해 log4j 라이브러리가 제공하는 Logger 클래스를
     * 통해 Logger 객체를 얻는다. getLogger()의 인수로 로깅 작업을
     * 할 클래스 타입을 지정하거나 클래스 명을 String으로 지정할 수 있다.
```

```
 **/
protected static final Logger logger =
        Logger.getLogger(BeanLifeCycle01.class);

/* 아래와 같이 commons-logging 라이브러리를 log4j 설정에 맞게 로깅
 * 처리를 할 수도 있다. org.apache.commons.logging.LogFactory
 * 클래스를 통해 Log 객체를 얻는다. getLog()의 인수로 로깅 작업을
 * 할 클래스 타입을 지정하거나 클래스 명을 String으로 지정할 수 있다.
 **/
/*
protected static final Log logger =
        LogFactory.getLog(BeanLifeCycle01.class);
*/

private String name;
private int age;

@Autowired
public void setName(String name) {
    this.name = name;
}

@Autowired
public void setAge(int age) {
    this.age = age;
}

/* 초기화 콜백 메서드 - JSR250(Java Specification Request) - Annotation사용
 * @PostConstruct는 초기화 콜백 메서드를 지정하는 annotation으로
 * 스프링이 bean 설정을 마친후 바로 @PostConstruct가 붙은 초기화 콜백
 * 메서드를 호출해야 한다고 스프링에게 알려주는 역할을 한다.
 * 초기화 콜백 메서드의 이름이 정해져 있는 것이 아니라 초기화 메서드로 사용할
 * 메서드에 @PostConstruct annotation을 지정하면 된다.
 * 스프링이 이 클래스의 기본 생성자로 bean을 생성하고 프로퍼티에 의존성을
 * 주입한 직후에 @PostConstruct가 붙은 이 메서드가 호출되는 순서로 진행된다.
 **/
@PostConstruct
public void beanInit() {

    // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
    if(logger.isDebugEnabled()) {
        logger.debug("BeanLifeCycle01 - initMethod 시작");
        logger.debug("name : " + name + ", age : " + age);
    }
```

```
        System.out.println("애플리케이션 초기화에 필요한 작업을 수행");

        // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
        if(logger.isDebugEnabled()) {
            logger.debug("BeanLifeCycle01 - initMethod 종료");
        }
    }

    /* 소멸화 콜백 메서드 - JSR250(Java Specification Request) - Annotation사용
     * @PreDestroy는 소멸화 콜백 메서드를 지정하는 annotation으로 스프링이
     * 이 클래스의 bean을 소멸시키기 직전에 @PreDestroy가 붙은 소멸화 콜백
     * 메서드를 호출해야 한다고 스프링에게 알려주는 역할을 한다.
     * 소멸화 콜백 메서드의 이름이 정해져 있는 것이 아니라 소멸화 메서드로 사용할
     * 메서드에  @PreDestroy annotation을 지정하면 된다.
     * 소멸 메서드에는 주로 사용한 자원을 반납하거나 닫는 코드를 기술한다.
     **/
    @PreDestroy
    public void beanDestroy() {
        if(logger.isDebugEnabled()) {
            logger.debug("BeanLifeCycle01 - Destroy 시작");
        }

        System.out.println("애플리케이션 소멸화에 필요한 작업을 수행");

        if(logger.isDebugEnabled()) {
            logger.debug("BeanLifeCycle01 - Destroy 종료");
        }
    }

    public static void main(String[] args) {

        // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
        if(logger.isDebugEnabled()) {
            logger.debug("main() 시작");
        }

        // GenericXmlApplicationContext는 ApplicationContext의 구현체 이다.
        GenericXmlApplicationContext ctx = new GenericXmlApplicationContext(
                "classpath:config/lifecycle/AnnotationLifeCycle01.xml");

        AnnotationLifeCycle01 bean1 =  ctx.getBean(AnnotationLifeCycle01.class);

        /* close() 메서드는 현재 실행중인 DI 컨테이너를 종료하는 메서드로
         * 스프링은 빈 컨테이너가 종료되기 직전에 모든 싱글턴 bean을 소멸 시킨다.
         * 이 때 bean이 가지고 있는 소멸화 메서드를 호출한다.
```

```
         **/
        ctx.close();

        // 로그 레벨이 DEBUG 레벨 이상이면 로그를 출력한다.
        if(logger.isDebugEnabled()) {
            logger.debug("main() 종료");
        }
    }
}
```

- src/main/resources/config/lifecycle/AnnotationLifeCycle01.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd">

    <!--
        스프링이 지정한 base-package를 기준으로 컴포넌트(클래스)를 스캔하여
        자동으로 bean을 생성해 주는 설정으로 <context:annotation-config />가
        수행하는 모든 것을 수행하며 여기에 더해 스프링이 자동으로 bean을 스캔하여
        선언하고 스프링 빈 설정 파일에서 명시적으로 <bean> 태그를 사용하지 않아도
        정상적으로 annotation 기반으로 와이어링 할 수 있다는 의미이다.
    -->
    <context:component-scan
            base-package="com.springstudy.ch02.lifecycle"  />

    <!-- Annotation으로 bean에 주입될 기본형 데이터와 String 데이터 -->
    <bean id="age" class="java.lang.Integer" c:_0="37" />
    <bean id="name" class="java.lang.String" c:_0="강감찬" />
</beans>
```

3. Spring AOP

3.1 스프링 AOP란?

앞에서 살펴본 DI외에 스프링 프레임워크의 또 다른 핵심 기능은 관점지향 프로그래밍(AOP, Aspect Oriented Programming)을 지원하는 프레임워크라는 것이다.
관점지향 프로그래밍이란 애플리케이션 전반에 걸쳐 산재되어 있는 공통 기능을 분리해서 별도의 모듈로 묶어 관리하고 이런 공통 기능을 애플리케이션이 실행될 때 필요한 여러 지점에 적용하는 것을 의미 한다.

▶ 횡단 관심사(Cross-Cutting Concerns)

애플리케이션에서 로깅처리, 보안처리, 트랜잭션 처리 등은 애플리케이션 전반에 걸쳐 여러 곳에 적용되는 공통기능 이다. 이렇게 **핵심 기능을 제외한 공통 기능이 애플리케이션 전반에 걸쳐서 비슷한 코드가 반복되므로 중복적인 코드가 많이 만들어지게 되는데 이렇게 공통 기능이 애플리케이션 전반에 걸쳐 여러 곳에서 반복적으로 나타나는 것을 횡단 관심사 또는 공통 관심사**라고 부른다.
횡단 관심사는 애플리케이션을 개발하고 유지하는데 꼭 필요한 기능이지만 핵심 관심사와 뒤섞여 애플리케이션 전반에 걸쳐 반복되어 나타나는 중복적인 코드이므로 코드의 변경이 어렵고 가독성이 떨어지며 단위 테스트를 어렵게 만든다. 또한 개발의 분업이 어려워지고 유지보수에도 많은 비용이 소요된다.

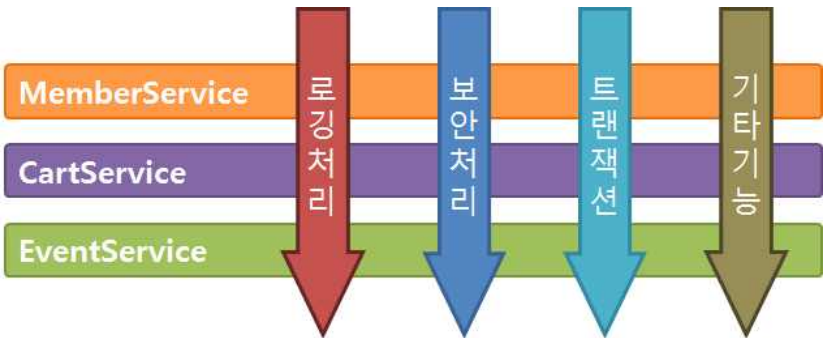

그림 3-1 횡단 관심사

▶ AOP(Aspect Oriented Programming)

AOP는 **공통 기능(횡단 관심사, Closs Cutting Concern)을 핵심 기능(핵심 관심사, Core Concern)과 분리하여 애스펙트(Aspect, 횡단 관심사를 모아놓은 것)라는 클래스로 모듈화 하고 이렇게 분리된 횡단 관심사를 애플리케이션이 실행될 때 필요한 지점에 다시 적용하는 프로그래밍 기법이다.** 객체지향 기법(상속 등)을 이용해 애플리케이션에 횡단 관심사를 효과적으로 적용하는 것은 한계가 있기 때문에 이를 보완하기 위하여 객체지향 프로그래밍에 AOP 기법이 도입 되었다. 그러므로 AOP 기법은 객체지향 프로그래밍과 경쟁관계가 아닌 객체지향을 보완하는 관계가 된다. 다시 말해 **AOP는 객체지향 프로그래밍에서 핵심 관심 모듈에 애플리케이션 개발 및 유지보수에 필요한 횡단관심사(로깅, 트랜잭션 처리 등)를 직접 구현(로깅이 필요한 부분에 직접 로깅을 처리하는 코드를 기술)하는 대신 핵심 관심사와 분리하여 독립된 모듈(로깅을 처리하는 별도의 클래스)로 구현하고 분리된 관심사를 애플리케이션이 실행할 때 핵심 관심 모듈의 사이사이에 결합하여 동작하도록 구현하는 프로그래밍 기법이라 할 수 있다.** 또한 AOP를 이용하면 기존의 OOP로 작성된 코드를 단 한 줄

도 수정하지 않고 필요한 공통 관심사를 핵심 관심사와 효과적으로 연동시킬 수 있다.

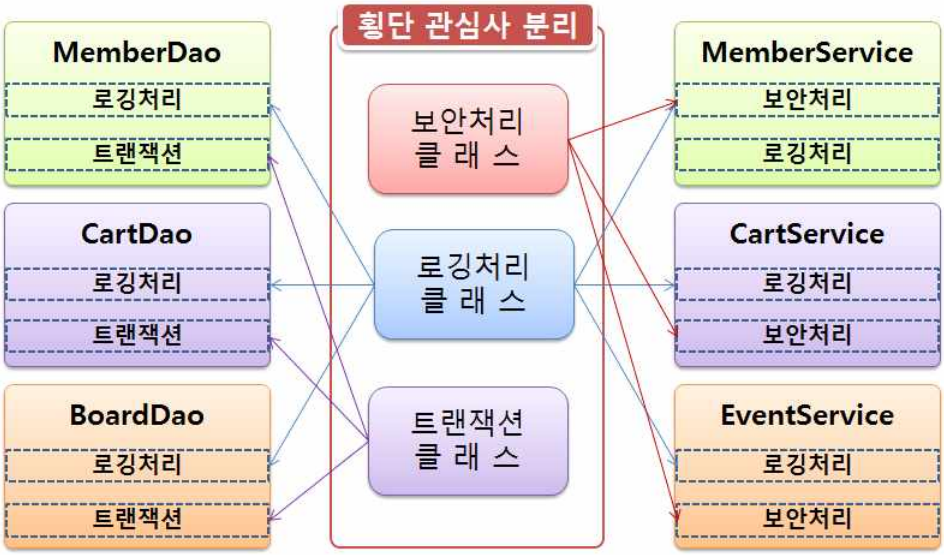

그림 3-2 횡단 관심사의 분리

3.2 스프링 AOP 용어

AOP를 설명하는 용어들은 스프링에만 특화된 것은 아니며 안타깝게도 스프링 입문자들에게 직관적으로 와 닿지 못한다. 하지만 자신의 프로그램에서 AOP를 사용하기 위해서는 반드시 이해하고 있어야 하는 용어들이므로 꼭 이해하길 바란다.

3.2.1 어드바이스(Advice)

어드바이스는 어떤 공통 기능(로깅처리, 보안처리 등)을 언제 수행할지를 정의한 것이다.
다시 말해 횡단 관심사로 분리되어 모듈화(클래스) 된 애스펙트(Aspect, 횡단 관심사)가 애플리케이션 실행시점에 핵심 관심 모듈의 특정 지점에서 수행해야 할 작업(로깅처리, 보안처리 등)과 이 작업을 언제 수행할지를 정의한 것이다. 한마디로 **특정 지점에서 실행되는 코드(공통기능을 의미, Advice 클래스 또는 애스펙트 클래스의 메서드로 정의한다.)를 어드바이스라 한다.**

◈ 어드바이스의 종류

▶ Before Advice
어드바이스 대상 메서드가 호출되기 전에 공통 기능을 수행한다.

▶ After Returning Advice
어드바이스 대상 메서드가 정상 종료된 후에 공통 기능을 수행한다.

▶ After Throwing Advice(== catch)
어드바이스 대상 메서드 실행 중 예외가 발생하면 공통 기능을 수행한다.

▶ After Advice(== finally)
어드바이스 대상 메서드가 종료(정상, 예외 모두)된 후 공통 기능을 수행한다.

▶ Around Advice
어드바이스 대상 메서드를 감싸서 대상 메서드의 실행 전과 종료(정상, 예외 모두)된 후 공통 기능을 수행한다.
다양한 시점에 필요한 기능을 적용할 수 있어 가장 강력하고 널리 사용되는 어드바이스 이다.

▶ Introduction
어드바이스 대상 객체를 수정하지 않고 새로운 메서드나 멤버 변수를 추가하는 기능이다.

3.2.2 조인포인트(JoinPoint)

조인포인트는 애플리케이션에서 어드바이스를 적용할 수 있는 곳(어드바이스 대상이 되는 지점)을 의미하며 핵심 관심사(Core Concern)에서 분리된 횡단 관심사(Closs Cutting Concern)를 적용할 수 있는 지점(Point)을 의미 한다. 이 지점(조인포인트)은 애플리케이션 실행 흐름에서 메서드 호출 지점, 로깅처리 지점, 예외처리 지점 등 무수히 많이 존재할 수 있다. Spring AOP에서는 AOP 대

상이 되는 객체의 메소드 호출 시점을 의미한다.

3.2.3 포인트 컷(Pointcut)

한 마디로 말해 **어드바이스를 어느 지점(조인포인트)에 적용할지를 정의한 것**이다. 즉 애플리케이션 실행 흐름 에서 무수히 **많이 존재하는 조인포인트(각 메서드 실행 시점) 중에서 어드바이스가 실제 적용될 지점(조인포인트, 공통 기능이 실행되어야 할 메서드를 선별)을 선별하기 위해 어드바이스의 적용 조건을 정의한 것**으로 연산자를 이용해 하나 또는 복수의 조인포인트를 묶어서 지정할 수 있다. 어드바이스가 어떤 공통기능(로깅처리, 보안처리 등)을 언제(메서드 호출 전, 메서드 호출 후 등) 수행할지를 정의한 것이라면 포인트 컷은 이 작업을 어디(어느 메서드가 실행될 때)에 적용할 지를 정의한 것이다.
어드바이스 대상이 되는 객체의 모든 메서드에 어드바이스를 적용할지 또는 특정 이름으로 시작하는 메서드(message로 시작하는 메서드)에만 어드바이스를 적용할지를 선별하는 필터 역할을 수행 한다고 보면 되겠다.
하나의 포인트컷에 여러 개의 어드바이스를 연결할 수도 있고 하나의 어드바이스에 여러 개의 포인트 컷을 연결할 수도 있다.

3.2.4 애스펙트(Aspect)

애스펙트는 한 개 또는 다수의 어드바이스와 포인트 컷의 조합이며 핵심 관심 모듈에 적용할 공통 기능(로깅처리, 보안처리 등)과 적용할 위치(어떤 클래스의 메서드 호출 시점을 기준으로) 그리고 적용할 시기(그 메서드의 호출 시점 이전 또는 이후 등등)를 정의한 것으로 주로 클래스로 구현한다.
애스펙트에는 어떤 횡단 관심사(공통 기능)를 어디에(어떤 클래스의 메서드가 실행되는 시점에), 언제(그 메서드가 실행되기 이전 또는 실행된 이후) 적용할지에 대한 모든 정보가 정의되어 있다.

(횡단 관심사) (어디에 - 포인트 컷) (언제)
예) 로깅처리를 대상객체(Target)의 특정 메서드가 실행되기 전에 적용되는 동작을 정의한다.
예) 로깅처리를 MessageBean객체의 messageHello()가 실행 전/후에 적용되는 동작을 정의한다.

스프링 AOP에서만 사용하는 어드바이저(Advisor)란 용어가 있다. 이는 한 개의 어드바이스와 한 개의 포인트 컷을 모듈화 한 애스펙트를 가리키는 말로 어드바이스와 포인트 컷을 하나로 묶어 관리하기 위한 스프링 AOP의 기본이 되는 객체이다. 또한 스프링 AOP에서 애스펙트는 Advisor 인터페이스를 구현한 클래스의 인스턴스로 표현되기도 한다.

3.2.5 위빙(Weaving)

핵심 관심사(Core Concern)에서 분리된 공통 관심사(Aspect)를 필요한 시점에 다시 그 핵심 관심사에 적용 시키는 것을 위빙(Weaving)이라 한다. 핵심 관심사[대상 객체(Target)]에서 분리되어 모듈화된 관심사(Aspect)를 다시 핵심 관심사에 적용하기 위해 새로운 프록시 객체를 생성하는 과정을 말한다. 애스펙트는 포인트컷에 정의한 대상 객체의 조인포인트로 위빙 된다. 위빙 되는 시점은 대

상 객체가 컴파일 될 때(Compile Time), 대상 객체의 클래스가 메모리에 로드 될 때(Classload Time), 대상객체가 실행중일 때(Runtime – 이 방식은 스프링에서 애스펙트를 위빙 하는 방식 이다.)와 같이 3가지 방식이 있다.

지금까지 살펴본 스프링 AOP 용어를 다시 정리하자면 어드바이스는 애플리케이션의 여러 객체 에 적용해야 할 공통적인 기능(로깅 처리, 보안처리 등)을 언제(특정 메서드 실행 전/후, 예외가 발생하면, 메서드 정상 종료 후 등) 실행할지 정의한 것이고 조인포인트는 애플리케이션 실행 흐름에서 어드바이스를 적용할 수 있는 모든 지점(메서드 호출 지점, 필드 값 변경 지점, 예외 발생 지점 등 – 스프링 프레임워크는 메서드 호출 지점만 지원 한다.)을 의미하며 포인트 컷은 어드바이스를 어디(실제 특정 어드바이스를 적용 할 위치 – 한 개 또는 그 이상의 조인포인트)에 적용해야 하는지를 정의한 것이라 할 수 있다.

3.2.6 대상 객체(Target)

어드바이스를 적용할 객체를 대상 객체 또는 어드바이스 적용 객체라 부른다.

아래 그림 3-3은 지금까지 설명한 스프링 AOP 용어와 AOP가 적용되는 개념을 그림으로 표현한 것이다.

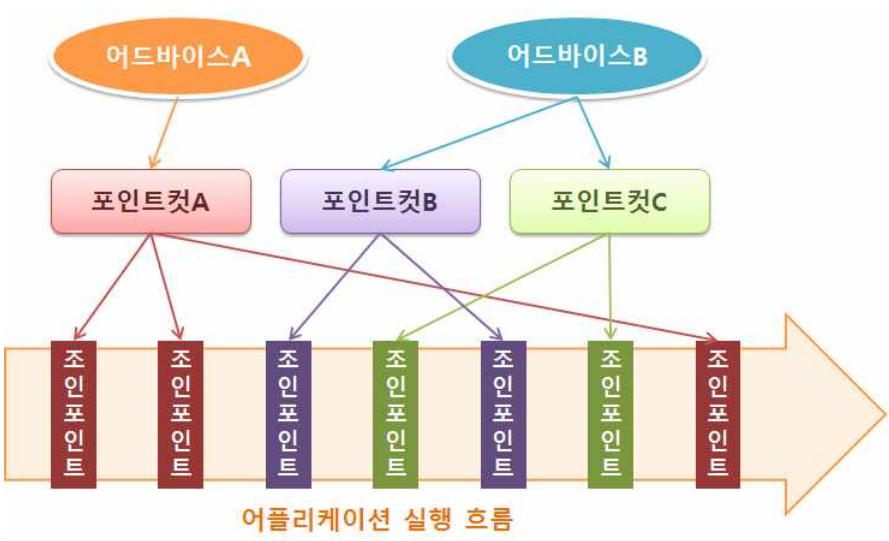

그림 3-3 스프링 AOP

3.3 AOP 프레임워크와 스프링 AOP 지원

스프링에서 지원하는 AOP는 AspectJ에서 가져온 개념도 많으며 스프링과 AspectJ를 결합하면 보다 정밀한 AOP를 구현할 수 있다.

▶ 많이 사용되는 AOP 프레임워크

- Aspect(http://eclipse.org/aspect)
- Jboss AOP(http://www.jboss.org/jbossaop)
- Spring AOP(http://www.springframework.org)

▶ 스프링에서 지원하는 AOP

- 클래식 스프링 프록시 기반 AOP
- AspectJ 애너테이션 기반 AOP
- POJO 기반 AOP
- AspectJ aspect 설정 AOP(스프링 모든 버전에서 지원)

3.4 스프링 AOP

AOP는 위빙이 일어나는 지점과 적용되는 방식에 따라 정적 AOP와 동적 AOP로 나눠진다. 정적 AOP는 컴파일 시점에 위빙이 일어나고 동적 AOP는 런타임 시점에 위빙이 일어난다.
대상 객체가 하나 이상의 인터페이스를 구현했다면 JDK 동적 프록시 기법을 사용해 대상 객체가 구현한 모든 인터페이스와 대상 객체가 프록시 되고 대상 객체가 어떤 인터페이스도 구현하지 않았다면 CGLIB를 사용해 대상 객체의 프록시를 생성한다.

그림 3-4 스프링 AOP 프록시

스프링 AOP는 Proxy를 기반으로 하며 런타임 시점에 빈 컨테이너가 관리하는 빈에 정의된 횡단 관심사(공통기능)를 분석한 후 필요한 프록시 빈을 동적으로 생성하여 타깃 빈(대상 객체)을 감싼다.
타깃 빈을 직접 호출하는 호출자에 프록시 빈을 주입하여 타깃에 대한 호출을 모두 프록시 빈이 받아 실행조건(조인포인트, 포인트 컷, 어드바이스 등)을 분석하고 조건에 맞는 어드바이스가 위빙(Weaving) 되도록 한다.
프록시란 말이 낯설어 어렵게 느껴질 것이다. 여기서 프록시란 AOP 대상이 되는 객체의 대용 객체를 말하는데 일반적으로 프록시는 다른 무엇인가와 연결되는 역할을 하는 클래스를 의미한다.
프록시는 AOP 대상이 되는 객체가 구현한 인터페이스를 구현하거나 AOP 대상이 되는 객체를 상속해 구현하게 되는데 글로 보다는 직접 프록시 패턴이 어떤 구조로 구현되는지 코드를 살펴보자.

3.4.1 Proxy 패턴

- com.springstudy.ch03.proxypattern

```
public interface ReadFile {
    public void fileDisplay();
}
```

- com.springstudy.ch03.proxypattern

```
public class ReadFileImpl implements ReadFile{

    private String filePath;

    public ReadFileImpl(String filePath) {
```

```
        this.filePath = filePath;
        loadFileFromDisk();
    }

    public void loadFileFromDisk() {
        System.out.println(filePath + " 파일 읽는 중...");
    }

    public void fileDisplay() {
        System.out.println(filePath + " 파일 표시 중...");
    }
}
```

- com.springstudy.ch03.proxypattern

```
public class ReadFileProxy implements ReadFile {

    // 프록시 객체는 대상 객체의 프로퍼티와 동일한 타입의 프로퍼티를 갖는다.
    private String filePath;

    /* 프록시 객체 내부에 대상 객체와 동일한 타입의 프로퍼티를 갖는다.
     * 대상 객체는 프록시 객체 안에서 직접 생성할 수 있거나 외부로 부터
     * 공급 받을 수 있도록 구현해야 대상 객체의 메소드에 접근 할 수 있다.
     **/
    private ReadFile readFile;

    private static final Log logger = LogFactory.getLog(ReadFileProxy.class);

    public ReadFileProxy(String filePath) {
        this.filePath = filePath;
    }

    // 대상 객체를 외부로 부터 공급 받을 수 있는 생성자
    public ReadFileProxy(String filePath, ReadFile readFile) {
        this.filePath = filePath;
        this.readFile = readFile;
    }

    public void fileDisplay() {

        if(logger.isDebugEnabled()) {
            logger.debug("Proxy fileDisplay() 시작");
        }

        if(readFile == null) {
```

```
            // 대상 객체가 생성되지 않았다면 직접 생성한다.
            readFile = new ReadFileImpl(filePath);
        }
        readFile.fileDisplay();

        if(logger.isDebugEnabled()) {
            logger.debug("Proxy fileDisplay() 종료");
        }
    }
}
```

- com.springstudy.ch03.proxypattern

```
public class ReadFileProxyTest {

    private static final Log logger = LogFactory.getLog(ReadFileProxyTest.class);

    public static void main(String[] args) {

        if(logger.isInfoEnabled()) {
            logger.debug("main() 시작");
        }

        // 원본 객체를 생성하고 있다.
        ReadFile readFile1 = new ReadFileImpl("d:\\ReadFileImpl.txt");

        // 원본 객체의 프록시 객체를 생성하고 있다.
        ReadFile readFile2 = new ReadFileProxy("d:\\ReadFileProxy.txt");
        ReadFile readFile3 = new ReadFileProxy("d:\\ReadFileProxy.txt", readFile1);

        // 원본 객체의 메소드를 호출하고 있다.
        readFile1.fileDisplay();

        /* 프록시 객체의 메소드를 호출하고 있다.
         * 프록시 객체의 메소드에서 대상 객체의 메소드를 호출하게 된다.
         **/
        readFile2.fileDisplay();
        readFile3.fileDisplay();

        if(logger.isInfoEnabled()) {
            logger.debug("main() 종료");
        }
    }
}
```

3.4.2 선언적 AOP

예전에는 스프링에서 AOP를 구현할 때 스프링이 제공하는 인터페이스를 구현하는 방식이 사용되었다. 이 방식은 핵심 관심사에 횡단관심사(공통기능)를 적용하기 위해 대상 객체의 프록시를 생성 하고 어드바이스를 적용하는 모든 과정을 코드로 구현하는 방식으로 스프링 초기에 사용되던 AOP 구현 방법으로 요즘은 거의 사용되지 않는 방식이다. 사실 ProxyFactory를 통해 대상 객체의 프록시를 생성하고 어느 조인 포인트에 어떤 어드바이스를 적용할 지를 일일이 코드로 작성하는 방식의 AOP 구현은 지나치게 복잡하고 성가신 작업이 아닐 수 없다. 하지만 스프링 프레임워크는 이 복잡한 AOP 구현 방식 외에 보다 단순하고 쉽게 대상 객체의 프록시를 생성하여 어드바이스를 적용할 수 있는 선언적인 AOP 방식을 제공하고 있다. 선언적 AOP 방식은 Bean 설정 파일이나 Annotation을 사용해 AOP를 설정하고 스프링이 내부적으로 AOP를 처리해 주는 방식이다. 스프링이 제공하는 선언적인 AOP 방식에는 다음 3가지 형태로 구분할 수 있다.

▶ ProxyFactoryBean을 사용한 선언적 AOP 구현

ProxyFactoryBean은 스프링 빈 설정 파일에 빈을 정의하는 방식으로 어드바이스를 적용할 대 상 객체의 프록시를 생성하는 기능을 제공한다. FactoryBean의 구현체인 ProxyFactoryBean 클래스는 대상 객체(Target)를 지정할 수 있는 target 속성과 대상 객체의 프록시에 적용할 Advice 또는 Advisor를 지정할 수 있는 interceptorNames 속성을 지원한다.

이 방식은 AOP를 선언적으로 정의하는 방식이기는 하지만 Advice 클래스를 구현할 때 AOP 연합에서 제공하는 인터페이스를 직접 구현해야 하는 부담이 따른다. 그렇기 때문에 이 방식도 요즘은 많이 사용되지 않는 방식이다. 아래에 설명하고 있는 aop 네임스페이스를 사용하는 방식과 AspectJ 애노테이션을 사용한 선언적 AOP 방식이 주로 사용되므로 ProxyFactoryBean을 사용한 선언적 AOP 구현 방식이 있다는 것만 알아두고 아래 두 가지 방식에 더 집중해 주기 바란다. 요즘에는 편리성 때문에라도 아래 두 번째에서 소개하고 있는 AspectJ 애노테이션을 사용한 선언적 AOP를 많이 사용하므로 우리도 이 방법에 대해서만 알아볼 것이다.

▶ aop 네임스페이스를 사용한 선언적 AOP 구현(XML 스키마 기반)

스프링 2.0부터 도입된 방식으로 스프링 빈 설정 파일에서 aop 네임스페이스를 사용해 AOP를 선언적으로 정의할 수 있도록 지원하고 있다. 이 방식은 여러 개의 어드바이스를 메서드로 정의 한 클래스를 스프링 설정 파일에 빈으로 선언 하고 <aop:config></aop:config> 요소 사이에 다수의 포인트 컷과 다수의 애스펙트(Aspect)를 정의할 수 있도록 지원하고 있다.

aop 네임스페이스를 사용하는 방식도 내부적으로 ProxyFactoryBean을 사용하여 대상 객체의 프록시를 생성하고 어드바이스를 적용해 준다.

▶ AspectJ 애노테이션을 사용한 선언적 AOP 구현

커스텀 클래스에 @Aspect 애노테이션을 지정해 어떤 클래스든지 애스펙트(Aspect)로 선언하여 AOP를 선언적으로 정의할 수 있는 방식으로 AspectJ 애노테이션을 사용해 스프링 설정 파일에서 aop 네임스페이스를 사용할 때와 동일한 AOP를 구현할 수 있다.

클래스 레벨에 @Aspect 애노테이션을 지정하고 메서드에 @Pointcut 애노테이션을 사용해 포인트 컷을 정의할 수 있고 @Before, @After @AfterThrowing, @AfterReturning, @Around 등의 애노테이션을 사용해 여러 개의 어드바이스를 정의 할 수도 있다. 또한 위의 어드바이스 지정 애노테이

션과 포인트컷 지정자를 사용해 어드바이저(Advisor)를 정의할 수도 있다.

@Aspect(AspectJ) 애노테이션을 이용한 AOP

▶ Advice가 위빙(Weaving)될 대상 객체(Target) 1

- com.springstudy.ch03.declaration

```
/* Advice가 위빙(Weaving)될 대상 객체(Target)
 * @Component annotation을 사용해 이 클래스가 스프링
 * 컴포넌트임을 선언하고 messageBean 이라는 이름을 지정하고 있다.
 **/
@Component("messageBean")
public class MessageBeanAspectJAnnotation {

    private String name;

    @Autowired
    public void setName(String name) {
        this.name = name;
    }
    public void messageDisplay() {
        System.out.println(
                "messageBean의 messageDisplay() : 안녕하세요 " + name + "님!");
    }
    public void messageDisplay(String name) {
        System.out.println(
                "messageBean의 messageDisplay(name) : 안녕하세요 " + name + "님!");
    }
    public void messageHello() {
        System.out.println(
                "messageBean의 messageHello() : 안녕하세요 " + name + "님!");
    }
    public void messagePrint(String name) {
        System.out.println(
                "messageBean의 messageHello(name) : 안녕하세요 " + name + "님!");
    }
}
```

▶ Advice가 위빙(Weaving)될 대상 객체 2

- com.springstudy.ch03.declaration

```
/* Advice가 위빙(Weaving)될 대상 객체
 * @Component annotation을 사용해 이 클래스가 스프링
 * 컴포넌트임을 선언하고 messageAnnotation 이라는 이름을 지정하고 있다.
 **/
@Component("messageAnnotation")
```

```
public class MessageBeanAnnotation {

    private String name;

    @Autowired
    public void setName(String name) {
        this.name = name;
    }
    public void messageDisplay() {
        System.out.println(
                "messageAnnotation의 messageDisplay() : 안녕하세요 " + name + "님!");
    }
    public void messageDisplay(String name) {
        System.out.println(
                "messageAnnotation의 messageDisplay(name) : 안녕하세요 " + name + "님!");
    }
    public void messageHello() {
        System.out.println(
                "messageAnnotation의 messageHello() : 안녕하세요 " + name + "님!");
    }
    public void messagePrint(String name) {
        System.out.println(
                "messageAnnotation의 messageHello(name) : 안녕하세요 " + name + "님!");
    }
}
```

▶ AspectJ Annotation을 사용해 Aspect를 정의한 클래스

- com.springstudy.ch03.declaration

```
/* @Component annotation을 사용해 이 클래스가 스프링 컴포넌트임을
 * 선언하고 @Aspect annotation을 사용해 이 클래스를 애스펙트로 지정하고 있다.
 *
 * 이 클래스는 AspectJ 기반의 annotation을 사용해 aop 네임스페이스를
 * 사용할 때와 동일한 애스펙트를 구현하고 있지만 스프링은 어드바이스 대상 메소드에
 * 어드바이스를 적용할 때 AspectJ 위빙 메커니즘을 사용하는 것이 아니라 스프링의
 * 자체 프록시 메커니즘을 사용해 대상 객체의 프록시를 생성한다는 것을 기억해야 한다.
 *
 * 이 클래스에서 사용하는 JoinPoint, ProceedingJoinPoint 클래스는
 * org.aspectj.lang 패키지의 클래스이고 @Aspect, @Pointcut, @Around,
 * @Before, @After, @AfterReturning, @AfterThrowing 등의 애노테이션은
 * org.aspectj.lang.annotation 패키지에 커스텀 애노테이션을 정의한 인터페이스이므로
 * AspectJ 런타임 라이브러리와 위빙 라이브러리인 aspectjrt-1.8.7.jar와
 * aspectjweaver-1.8.7.jar를 Maven을 통해 라이브러리 의존성을 설정 하였다.
 *
 * 스프링 AOP에서 애스펙트는 다른 애스펙트의 어드바이스 타겟이 될 수 없다.
```

* 클래스에 @Aspect 애노테이션을 적용하게 되면 이 클래스가 애스펙트라는
* 것을 스프링에게 알려주므로 스프링은 이 클래스를 오토프로싱에서 제외시킨다.
*
* ## AspectJ execution() 지정자를 사용한 포인트컷 설정하기 ##
* AspectJ는 포인트컷을 지정할 수 있는 다양한 지정자(Pointcut Designator)를
* 제공하는데 스프링은 메소드 호출과 관련된 지정자만 지원하고 있어 AspectJ의
* 포인트컷 지정자 중 메소드 호출과 관련된 지정자만 사용할 수 있다.
* 이 포인트컷 표현식은 AspectJ 포인트컷 표현식이 사용되는 ProxyFactoryBean을
* 이용한 AOP 구현, aop 네임스페이스를 이용한 AOP구현, AspectJ 애노테이션을
* 이용한 AOP 구현 등에서 포인트컷을 설정하는데 사용할 수 있다.
* execution() 포인트컷 지정자는 Advice를 적용할 메소드를 매치시키는데 사용되며
* 이 지정자를 사용해 포인트컷을 설정하는 방법은 다음과 같다.
*
* execution(메소드접근지정자v메소드리턴타입v패키지.클래스명.메소드이름(파라미터))v예외타입
*
* - v : 공백을 의미한다.
* - 메소드 접근지정자, 예외타입, 패키지.클래스명은 생략할 수 있다.
* - 메소드 리턴 타입, 패키지, 클래스명, 메소드 이름에는 와일드카드(*)를 사용할 수 있다.
* - 와일드카드(*)는 패키지 구분자(.)와 일치하지 않으므로 여러 패키지와 일치 시키려면
* .. 을 사용해 현재 패키지와 하부 패키지를 지정할 수 있다.
* - 메소드 인수에 (..)를 지정하면 파라미터가 0개 이상인 메소드와 일치된다.
* - 메소드 인수에(*)를 지정하면 파라미터가 1개인 모든 메소드와 일치된다.
* 메소드 인수에 *를 지정하면 모든 타입의 파라미터와 일치된다.
*
* - execution(protected String *Print())
* 리턴 타입이 String 이고 이름이 Print로 끝나며 파라미터가 없는 메소드가 일치된다.
*
* - execution(public void *Display(..))
* 리턴 타입이 void이고 이름이 Display로 끝나며 0개 이상의 파라미터를 가진 메소드가
* 일치된다.
*
* - execution(* com.springstudy.ch03.*.set*(*))
* 리턴 타입이 어떤 것이든 상관없고 com.springstudy.ch03 패키지의 모든 클래스에
* 정의된 이름이 set으로 시작하며 1개의 파라미터를 가진 메소드가 일치된다.
* 파라미터 타입은 무엇이든 상관없다.
*
* - execution(* com.springstudy.ch03.MessageBean.*(java.lang.String, *))
* 리턴 타입이 어떤 것이든 상관없고 com.springstudy.ch03.MessageBean 클래스에
* 정의된 두 개의 파라미터를 가진 메소드 중 첫 번째 파라미터가 String 타입인 메소드가
* 일치된다. 두 번째 파라미터 타입은 무엇이든 상관없다.
*
* - execution(* com.springstudy.ch03.*.*Print(*, *))
* 리턴 타입이 어떤 것이든 상관없고 com.springstudy.ch03 패키지의 모든 클래스에
* 정의된 이름이 Print로 끝나며 2개의 파라미터를 가진 메소드가 일치된다.
* 파라미터 타입은 무엇이든 상관없다.

```
 *
 * - execution(* com.springstudy.ch03.*.*(Integer, ..))
 *  리턴 타입이 어떤 것이든 상관없고 com.springstudy.ch03 패키지의 모든 클래스에
 *  정의된 메소드 중 첫 번째 파라미터 타입이 Integer이며 1개 이상의 파라미터를 가진
 *  메소드가 일치된다.
 *
 * ## AspectJ within() 지정자를 사용한 포인트컷 설정하기 ##
 * within() 포인트컷 지정자는 단순히 메소드 이름으로 매치하는 것이 아닌
 * 특정 타입에 정의한 메소드로 포인트컷을 제한하는데 사용하는 포인트컷
 * 지정자로 이 지정자를 사용해 포인트컷을 설정하는 방법은 다음과 같다.
 *
 * - within(com.springstudy.ch02.member.MemberService)
 *   MemberService 인터페이스의 모든 메소드가 매치된다.
 *
 * - within(com.springstudy.ch02.member.*)
 *   com.springstudy.ch02.member 패키지에 있는 모든 메소드가 매치된다.
 *
 * - within(com.springstudy.ch02.member..*)
 *  com.springstudy.ch02.member 패키지와 그 하위 패키지에 있는
 *  모든 메소드가 매치된다.
 *
 * ## 스프링 bean() 지정자를 사용한 포인트컷 설정하기 ##
 * bean() 포인트컷 지정자는 스프링 2.5 부터 스프링에서 추가적으로
 * 지원하는 포인트컷 지정자로 스프링 빈 이름(id나 이름)을 사용해 포인트컷을 설정하는데
 * 사용하는 포인트컷 지정자 이다. bean() 포인트컷 지정자에 지정한 이름을 가진 빈으로
 * 포인트컷을 제한하는데 사용되며 이 지정자를 사용해 포인트컷을 설정하는 방법은 다음과 같다.
 *
 * - bean(memberService)
 *   빈의 이름이 memberService인 빈의 모든 메소드가 매치된다.
 *
 * - bean(*Service)
 *  빈의 이름이 Service로 끝나는 빈의 모든 메소드가 매치된다.
 *
 * 이외에도 this(), args(), target(), @args(), @target(), @within(), @annotation()등의
 * 포인트컷 지정자를 사용해 포인트컷을 설정할 수 있으며 논리 연산자[and(&&), or(||),
 * not(!)]를 사용해 여러 개의 포인트컷 지정자를 조합하여 포인트컷을 설정할 수 있다.
 **/
@Component
@Aspect
public class MessageBeanAspectJAnnotationAspect {

    private final static Log logger =
            LogFactory.getLog(MessageBeanAspectJAnnotationAspect.class);

    /* AspectJ 기반의 애노테이션을 사용해 argument 체크 포인트컷을 정의하고 있다.
```

```
 * 아래의 포인트컷 표현식을 보면 aop 네임스페이스에서 사용한 and 연산자를 사용한
 * 것이 아니라 && 연산자를 사용해 AND 연산을 한다는 것에 주의를 기울여야 한다.
 *
 * com.springstudy.ch03.declaration 패키지 하위에 정의된 클래스
 * (하위 패키지 포함)의 String 타입의 매개 변수가 하나인 Display로 끝나는
 * 모든 메소드에 어드바이스를 적용하는 포인트컷을 정의하고 있다.
 * 또한 && 연산자와 args() 포인트컷 지정자를 사용해 어드바이스에 name이라는
 * 이름의 인수를 넘겨주도록 지정하고 있다.
 * args 표현식에서 타입 이름 즉 args(MessageBean) 대신 파라미터
 * 이름을 사용하면 어드바이스를 호출할 때 args 표현식에 지정한 인수명과
 * 이름이 같은 어드바이스의 매개변수로 값이 전달된다.
 * args 표현식에 지정한 인수의 이름과 동일한 파라미터의 이름이 어드바이스에
 * 반드시 존재해야 하며 그렇지 않으면 IllegalArgumentException이 발생한다.
 *
 * 애스펙트 클래스에서 하나의 포인트컷은 아래와 같이 하나의 메소드로 정의할 수
 * 있으며 포인트컷의 이름은 포인트컷으로 정의한 메소드 이름을 사용하면 된다.
 **/
@Pointcut("execution("
        + "* com.springstudy.ch03.declaration..*Display(String)) && args(name)")
public void argPointcut(String name) { }

/* AspectJ 기반의 애노테이션을 사용해 포인트컷을 정의하고 있다.
 * bean() 포인트컷 지정자를 사용해 messageBean으로 시작하는
 * id를 가진 빈에게 만 어드바이스를 적용하는 포인트컷을 정의하고 있다.
 **/
@Pointcut("bean(messageBean*)")
public void beanScopePointcut() { }

/* default 비포 어드바이스를 정의하고 포인트컷 연결
 * AspectJ 기반의 애노테이션을 사용해 Before Advice를 정의하고 있다.
 * com.springstudy.ch03.declaration 패키지 하위에 정의된 클래스
 *  (하위 패키지 포함)의 String 타입의 매개 변수가 하나인 Display로 끝나는
 * 모든 메소드에 어드바이스를 적용하는 포인트컷을 연결하고 있다.
 *
 * @Around, @Before, @After, @AfterReturning, @AfterThrowing 등의
 * 애노테이션 인수에 별도로 선언된 포인트컷을 연결할 수도 있고 아래와 같이
 * AspectJ 포인트컷 지정자를 사용해 바로 포인트컷을 연결할 수도 있다.
 **/
@Before("execution(* com.springstudy.ch03.declaration..*Display(String))")
public void messageBeforeAdvice(JoinPoint joinPoint) throws Throwable{

    if(logger.isDebugEnabled()) {
        logger.debug("Before - default : "
                + joinPoint.getSignature().getDeclaringTypeName()
                + "." + joinPoint.getSignature().getName());
```

```
    }
}

/* argument 체크 비포 어드바이스를 정의하고 포인트컷 연결
 * AspectJ 기반의 애노테이션을 사용해 Before Advice를 정의하고 있다.
 * 위에서 정의한 포인트컷을 사용해 어드바이스에 포인트컷을 연결하고 있다.
 **/
@Before("argPointcut(name) && beanScopePointcut()")
public void messageBeforeAdvice(
        JoinPoint joinPoint, String name)  throws Throwable{

    if(name.equals("강감찬")) {

        if(logger.isDebugEnabled()) {
            logger.debug("Before - name : "
                + joinPoint.getSignature().getDeclaringTypeName()
                + "." + joinPoint.getSignature().getName() + " - arg : " + name);
        }
    }
}

/* 어라운드 어드바이스를 정의하고 포인트컷 연결
 * AspectJ 기반의 애노테이션을 사용해 Around Advice를 정의하고 있다.
 * 대상 객체에서 String 타입의 매개 변수가 하나인 message로 시작하는
 * 모든 메소드에 어드바이스를 적용하는 포인트컷을 연결하고 있다.
 * 그리고 && 연산자와 args() 포인트컷 지정자를 사용해 어드바이스에
 * name이라는 이름의 인수를 넘겨주도록 지정하고 있으며 위에서 포인트컷으로
 * 선언한 beanScopePointcut()을 지정해 messageBean으로 시작하는
 * id를 가진 빈에게 만 어드바이스가 적용되도록 지정하고 있다.
 * args 표현식에서 타입 이름 즉 args(MessageBean) 대신 파라미터
 * 이름을 사용하면 어드바이스를 호출할 때 args 표현식에 지정한 인수명과
 * 이름이 같은 어드바이스의 매개변수로 값이 전달된다.
 * args 표현식에 지정한 인수의 이름과 동일한 파라미터의 이름이 어드바이스에
 * 반드시 존재해야 하며 그렇지 않으면 IllegalArgumentException이 발생한다.
 **/
@Around("execution(* message*(String)) "
        + "&& args(name) && beanScopePointcut()")
public Object messageAroundAdvice(
        ProceedingJoinPoint joinPoint, String name)  throws Throwable{

    Object obj = null;
    if(logger.isDebugEnabled()) {
        logger.debug("Around - Before: "
                + joinPoint.getSignature().getDeclaringTypeName()
                + "." + joinPoint.getSignature().getName());
```

```
        }

        /* ProceedingJoinPoint 객체를 통해 호출되는 메소드, 대상 조인포인트,
         * AOP 프록시, 메소드의 인수에 대한 정보를 얻을 수 있고 이 객체를 사용하면
         * 메소드 호출이 실행되는 시점도 제어할 수 있다.
         * 매개변수로 받은 ProceedingJoinPoint 객체의 proceed()를 호출해야
         * 조인 포인트의 메소드가 호출되고 그 메소드의 반환 값을 얻을 수 있다.
         * 조인 포인트의 메소드 반환 타입이 void면 null이 반환 된다.
         **/
        obj = joinPoint.proceed();

        if(logger.isDebugEnabled()) {
            logger.debug("Around - After : "
                    + joinPoint.getSignature().getDeclaringTypeName()
                    + "." + joinPoint.getSignature().getName());
        }

        // Around Advice는 proceed() 메서드의 반환 값을 반환해야 한다.
        return obj;
    }
}
```

▶ AspectJ Annotation 방식의 AOP를 적용하기 위한 Bean 정의파일

- src/main/resources/config/MessageAspectJAnnotationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">

    <!--
        ## @Aspect(AspectJ) 애노테이션을 이용한 AOP 구현 과정 ##
        1. spring-aop 라이브러리 의존성을 설정한다.
           spring-context 라이브러리 의존성을 설정하면 spring-context 가
           spring-aop에 의존하기 때문에 자동으로 spring-aop 의존성이 설정된다.

        2. 횡단관심사(로깅처리와 같은 공통기능)를 정의한 Advice와 어느 지점에 Advice를
```

적용할지를 정의한 포인트컷을 모듈화한 애스펙트 클래스를 구현한다.
애스펙트 클래스에서 하나의 포인트컷은 하나의 메소드로 정의하며
포인트컷의 이름은 포인트컷으로 정의한 메소드 이름을 사용하면 된다.

AspectJ 애노테이션을 사용해 애스펙트 클래스에 @Aspect 애노테이션을
적용하고 포인트컷을 정의한 메소드에 @Pointcut 애노테이션을 적용한다.
Advice로 사용할 메소드에 @Before(), @After(), @AfterReturning(),
@AfterThrowing(), @Around() 등의 애노테이션을 적용한다.

Advice와 포인트컷 연결은 Advice를 의미하는 AspectJ 애노테이션인
@Before(), @Around()에서 () 안에 AspectJ 포인트컷 지정식을
사용해 포인트컷을 연결할 수 있고 이미 포인트컷으로 정의한 메소드 이름을
참조해 포인트컷을 Advice에 연결할 수도 있다.

3. <beans> 요소에 aop 네임스페이스를 추가한다.

4. XML 설정 파일에 <aop:asptecj-autoproxy />를 설정한다.
AspectJ에서 사용하는 애노테이션을 스캔하기 위해 사용한다.

5. XML 설정 파일에 <context:component-scan basepackage="" />를 설정한다.
애플리케이션에 필요한 클래스를 스프링이 스캔하여 자동으로 빈을 생성하고
DI 컨테이너에 담을 수 있도록 하기 위해 이 설정이 필요하다. 애플리케이션에서
필요한 클래스에 @Component나 스프링이 빈으로 인식하기 위해 필요한
애노테이션을 지정해야 스프링이 지정한 패키지의 클래스를 빈으로 생성할 수 있다.
이 설정을 사용하지 않는다면 어플리케이션에서 필요한 모든 클래스를
<bean class="" /> 요소를 사용해 스프링 빈으로 설정해야 한다.
-->

<!--
@AspectJ 기반의 애노테이션 지원을 활성화 하도록 지정하고 있다.
@AspectJ 기반의 애노테이션 지원이 활성화되면 ApplicationContext에서
@AspectJ 기반의 애노테이션인 @Aspect가 지정된 애스펙트 클래스를
스프링이 자동으로 탐지하고 AOP를 설정하는데 사용한다.
아래의 요소에는 proxy-target-class라는 속성이 있는데 기본 값은 false로
스프링은 JDK 동적 프록시를 사용해 인터페이스 기반 프록시를 생성하고 이 속성을
true로 지정하면 스프링은 CGLIB를 사용해 클래스 기반의 프록시를 생성한다.
-->
<aop:aspectj-autoproxy />

<!--
어플리케이션에서 사용하는 모든 클래스를 스캔하고 빈을 생성하여 DI 컨테이너에
담는 데는 <aop:aspectj-autoproxy />만 가지고는 부족하다.
스프링에게 어플리케이션에서 필요한 대상객체나 애스펙트 클래스가 스프링
컴포넌트라는 것을 알려주기 위해 아래와 같이 추가로 기술해야 한다.
<context:component-scan base-package=""/>를 설정하지 않으면

```
        어플리케이션에서 필요한 클래스를 <bean class="" /> 요소를 사용해 빈으로
        설정하고 @Aspect 애노테이션을 지정한 애스펙트 클래스도 빈으로 설정해야 한다.
     -->
    <context:component-scan
            base-package="com.springstudy.ch03.declaration" />

    <!-- Annotation으로 bean에 주입될 String 데이터 -->
    <bean id="name" class="java.lang.String" c:_0="이순신" />
</beans>
```

▶ 스프링 DI 컨테이너를 생성해 AspectJ Annotation 방식의 AOP를 실행시키는 메인 클래스

- com.springstudy.ch03.declaration

```
public class MessageBeanAspectJAnnotationExample {

    public static void main(String[] args) {

        /* 스프링 설정 파일에 정의한 bean을 생성해 빈 컨테이너에 담는다.
         * 이 예제는 @AspectJ annotation 기반 bean 설정을 사용하므로 @Autowired와
         * @Componrnt에 관련된 bean을 생성해 빈 컨테이너에 담는다.
         **/
        ApplicationContext ctx = new ClassPathXmlApplicationContext(
                "config/MessageAspectJAnnotationContext.xml");

        /* 빈 컨테이너로 부터 클래스에 @Component 애노테이션이 설정된
         * messageBean이라는 id 또는 name을 가진 Advice 대상 객체(Target)의
         * 프록시 객체를 얻어와 메시지를 출력한다.
         **/
        MessageBeanAspectJAnnotation bean1 =
                ctx.getBean("messageBean", MessageBeanAspectJAnnotation.class);

        System.out.println("## messageBean ##");
        bean1.messageDisplay();
        System.out.println("");
        bean1.messageDisplay("홍길동");
        System.out.println("");
        bean1.messageHello();
        System.out.println("");
        bean1.messagePrint("강감찬");
        System.out.println();

        /* 빈 컨테이너로 부터 클래스에 @Component 애노테이션이 설정된
         * messageAnnotation이라는 id 또는 name을 가진 Advice 대상 객체(Target)의
         * 프록시 객체를 얻어와 메시지를 출력한다.
         **/
```

```
        MessageBeanAnnotation bean2 =
                ctx.getBean("messageAnnotation", MessageBeanAnnotation.class);

        System.out.println("## messageAnnotation ##");
        bean2.messageDisplay();
        System.out.println("");
        bean2.messageDisplay("홍길동");
        System.out.println("");
        bean2.messageHello();
        System.out.println("");
        bean2.messagePrint("강감찬");
    }
}
```

4. SpringWebMVC

JSP에서 MVC 패턴을 구현할 때 우리가 만든 서블릿 클래스가 특정 패턴(*.do 등)에 대한 모든 요청을 받을 수 있도록 서블릿 매핑을 한 기억이 있을 것이다. 이렇게 하나의 서블릿 클래스가 특정 패턴에 대한 모든 요청을 처리하도록 구현하는 방식을 프런트 컨트롤러 패턴이라고 부른다.
스프링 MVC에서도 프런트 컨트롤러 패턴이 사용되며 스프링 MVC가 제공하는 DispatcherServlet 클래스가 프런트 컨트롤러가 된다. 그러므로 스프링 MVC를 이용해 웹 애플리케이션을 구현하기 위해서는 먼저 DispatcherServlet을 배포서술자(web.xml)에 서블릿으로 등록하고 특정 패턴에 대한 모든 요청을 이 서블릿 클래스가 받을 수 있도록 서블릿 매핑을 설정해야 한다.
그림 4-1은 스프링 MVC를 이용해 웹 애플리케이션을 구현하는 핵심 구성요소를 그림으로 나타낸 것이다. 이 그림을 살펴보면 스프링 MVC의 가장 핵심인 DispatcherServlet 클래스가 중앙에 위치한 것을 볼 수 있을 것이다. DispatcherServlet은 클라이언트로부터 요청이 들어오면 그 요청을 처리하기 위한 컨트롤러를 HandlerMapping을 통해서 어떤 Controller(개발자가 구현한 Controller)가 요청을 처리할 지를 결정하고 결정된 해당 Controller를 호출해 요청을 처리하게 된다. 이때 호출된 Controller는 Service 계층의 클래스를 사용해 요청을 처리하게 되며 Service 계층의 클래스는 DataAccess 계층의 클래스를 사용해 DB 작업을 하게 된다. 그러므로 그림 4-1에서 개발자가 구현해야 하는 것은 Controller와 요청을 처리한 결과를 보여줄 View 부분이다. 또한 아래 그램에서는 표현되지 않았지만 Controller에 의해 호출되는 Service 계층의 클래스와 Service 클래스에 의해 호출되는 DataAccess 계층의 클래스를 구현해야 한다. 나머지는 스프링 MVC에서 제공하는 클래스들이기 때문에 빈 설정 파일을 통해 이 클래스 들이 빈으로 등록되어 있어야 한다.

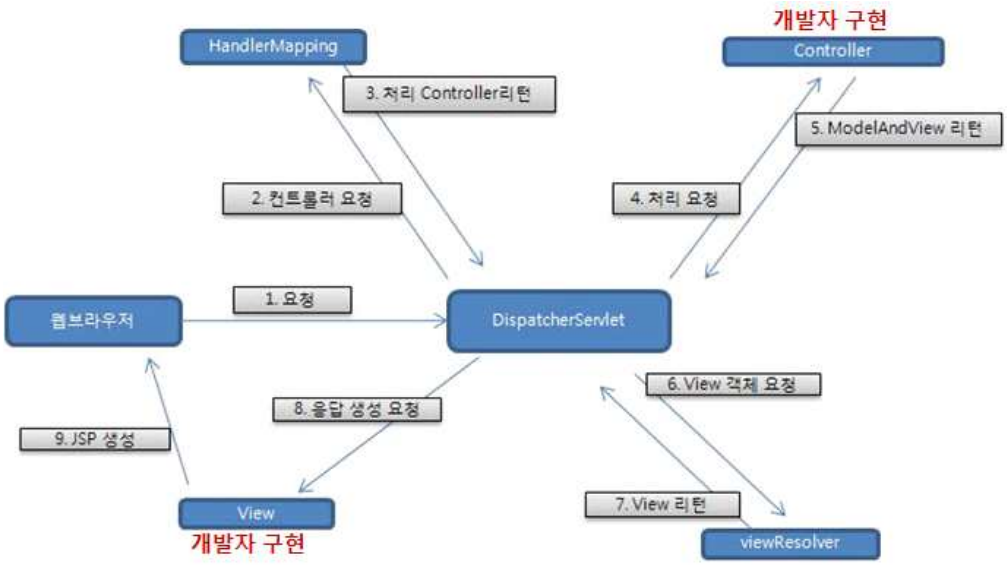

그림 4-1 스프링 MVC 핵심 구성요소

다시 위의 그림 4-1을 살펴보면 DispatcherServlet으로부터 호출된 Controller(개발자가 구현한 Controller)가 요청을 처리한 결과를 ModelAndView 객체에 담아 DispatcherServlet에게 전달하는 것을 볼 수 있는데, 이것은 요청을 동일한 방식으로 처리하기 위한 것으로 그림 4-1에는 없지만 DispatcherServlet과 Controller 사이에 HandlerAdapter라는 객체가 존재한다.
예전에는 스프링 MVC가 제공하는 Controller 인터페이스를 상속받아 Controller를 구현하였는데 이때 구현하는 메서드의 반환타입이 모델과 뷰의 정보를 담고 있는 ModelAndView 객체였다. 하지만 현재는 스프링 MVC가 제공하는 Controller 인터페이스를 직접 상속받는 방식은 사용되지 않으

며 Annotation 방식이 주로 사용되고 있다. Annotation 방식이 널리 사용되면서 Controller의 메서드는 ModelAndView 객체를 반환하지 않고 뷰의 정보만 String으로 반환되도록 구현되고 있다.
DispatcherServlet은 예전의 인터페이스를 구현한 방식과 최근에 많이 사용되는 Annotation 방식을 모두 지원하도록 설계되어 있기 때문에 요청 처리를 동일한 방식으로 처리해야 할 필요가 있다. 이를 위해 DispatcherServlet과 Controller 사이에 HandlerAdapter를 두어 DispatcherServlet이 직접 Controller를 호출하지 않고 HandlerAdapter를 통해 호출하게 되면 Controller가 요청을 처리한 결과를 HandlerAdapter가 받아서 ModelAndView 객체로 변환한 후 DispatcherServlet에게 전달하도록 설계되어 있다.
Annotation 방식도 Controller의 메서드에서 예전처럼 ModelAndView 객체를 반환할 수도 있지만 우리가 구현하는 Controller 안에서 ModelAndView 객체를 생성해 필요한 데이터를 이 객체에 저장하여 반환하는 작업은 오히려 번거롭기만 하다.

4.1. SpringMVC 첫 번째 예제

우리의 첫 번째 SpringMVC 예제로 SpringSTS에서 SpringLegacyProject를 생성하는 방법에 대해 알아보고 웹 애플리케이션에 스프링프레임워크를 적용하는 방법에 대해 알아볼 것이다. 이 예제는 Spring MVC가 지원하는 애노테이션을 사용해 Controller, Service, DAO를 구현하고 모델에 데이터를 저장해 뷰로 전달하는 간단한 Spring MVC 예제 이다.
이번 예제에서 DAO는 DB에 접속해 데이터를 가져오거나 저장하는 기능을 제공하지 않는다. 다만 단순히 파라미터를 받아서 간단히 처리하고 그 결과를 String으로 반환하는 기능을 제공하며 서비스 계층의 객체 또한 단순히 DAO 객체를 스프링으로부터 주입받아 DAO가 제공하는 getMessage() 메서드만 호출하는 아주 간단한 예제이다.

▶ 웹 애플리케이션 배포 서술자(Deployment Descriptor)

- src/main/webapp/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_1.xsd">
 <!--
  SpringSTS에서 SpringLegacyProject로 SpringMVCProject 만들기

  이 예제는 SpringSTS에서 Spring Legacy Project로 Spring MVC Project를
  생성하고 설정하는 방법에 대해 설명하기 위해 작성된 예제이다.

  Spring Legacy Project 메뉴를 통해 Spring MVC Project를 생성하게 되면
  JSP의 Dynamic Web Project와는 전혀 다른 프로젝트 구조를 가진다.
  Spring MVC Project가 이런 프로젝트 구조를 가질 수 있는 것은 web.xml에서
  리스너와 애플리케이션 초기화 파라미터, 그리고 서블릿 초기화 파라미터 설정을 통해
  스프링이 필요로 하는 bean 설정 파일의 위치를 지정해 사용하기 때문이다.

  1. 먼저 File -> New 메뉴에서 Spring Legacy Project를 선택하거나 Package Explorer의
     빈 공간에 마우스 오른쪽을 클릭해 나타나는 컨텍스트 메뉴의 New 메뉴에서 Spring
```

Legacy Project를 선택하여 나타나는 New Spring Legacy Project 대화상자에서 Spring MVC Project를 선택하고 프로젝트 이름을 springstudy-ch0401로 지정한 후 "Next" 버튼을 클릭한다. 이어서 나타나는 대화상자에서 top-level package를 com.springstudy.ch04로 입력하고 "Finish" 버튼을 클릭해 프로젝트를 생성한다.

2. 스프링프레임워크 버전과 의존 라이브러리 버전 그리고 Artifact Id 등을 변경한다.
Spring MVC Project를 생성하면 스프링프레임워크 버전 3.1.1이 기본 설정되어 있다. 우리는 스프링프레임워크 버전 4.2.4를 사용할 것이므로 Maven을 통해 버전을 수정해야 한다. 또한 테스트, 로깅. AOP, 데이터베이스 작업 등에 필요한 라이브러리 의존성도 추가로 설정하고 사용할 버전에 맞게 수정해야 한다.

pom.xml을 더블클릭해서 선택하고 Overview 탭에서 Properties 부분을 아래와 같이 변경해 준다. 만약 스프링프레임워크 버전을 변경한다면 org.springframework-version 부분을 더블 클릭해 Value의 값을 "4.2.4.RELEASE"로 지정한 후 OK 버튼을 클릭하면 된다. 나머지 의존 라이브러리도 아래를 참고해 property를 설정하면 된다.

```
<properties>
    <java-version>1.8</java-version>
    <org.springframework-version>4.2.4.RELEASE</org.springframework-version>
    <log4j-version>1.2.17</log4j-version>
    <jstl-version>1.2</jstl-version>
    <junit-version>4.7</junit-version>
    <mysql-version>8.0.31</mysql-version>
</properties>
```

그리고 이 Overview 탭에서 Artifact Id를 springstudy-ch0401로 수정하자.

3. 메이븐을 통해 스프링 MVC 관련 라이브러리 의존성을 설정한다.
pom.xml의 Dependencies 탭을 선택하고 "Add" 버튼을 클릭해 아래 모듈을 의존 설정하면 메이븐이 의존 관계에 있는 모듈을 자동으로 등록해서 스프링 mvc를 위한 가장 기본적인 라이브러리 의존 설정을 할 수 있다.

- spring-context 모듈
- spring-webmvc 모듈
- spring-jdbc 모듈
- commons-dbcp2 모듈
- mysql-connector-java 모듈
- 기타 필요한 라이브러리(jstl, log4j, slf4j, junit, aspectjrt 등등)

4. Java Build Path와 Compiler 등을 변경한다.
Spring MVC Project를 생성하면 자바 버전 1.6, Dynamic Web Module 2.5가 기본 설정되어 있다. 우리는 자바 1.8 이상과 Dynamic Web Module 3.1을 사용할 것이므로 Configure Build Path를 통해 Java Build Path의 자바 버전과 Java Compiler 버전을 1.8로 설정하고 Server Runtime 설정을 변경해야 한다. 그리고 프로젝트를 생성할 때 top-level package의 3번째 단계에 지정한 ch04가

자동으로 ContextRoot와 Artifact Id로 설정되기 때문에 ch04가 아닌 프로젝트 이름을 ContextRoot와 Artifact Id로 사용하려면 추가적인 설정이 필요하다.

* Configure Build Path에서 자바 버전과 Dynamic Web Module 변경하기
새로 생성한 프로젝트 springstudy-ch0401에 마우스 오른쪽 버튼을 클릭해 나타나는 컨텍스트 메뉴에서 "Build Path" -> "Configure Build Path"를 선택하여 나타나는 Properties for springstudy-ch0401 대화상자에서 다음과 같이 설정한다.

- Java Build Path 설정
Java Build Path 선택 -> JRE System Library 선택 -> Edit 버튼을 클릭
Edit Libray 대화상자에서 Workspace default JRE를 선택하고 Finish 클릭 한 후 Apply 버튼을 클릭한다.

- Java Compiler 설정
Java Compiler 선택 -> Enalbe project specific settings 선택 해제
우측의 Confiqure Workspace Settings 선택하여 나타나는 대화상자에서 Java Build 설정에 지정한 버전을 선택하고 Apply 버튼을 클릭하면 다시 이전 대화상자로 돌아오는데 여기서도 Apply 버튼 클릭

- Dynamic Web Module 설정
Project Facets 선택 -> 우측의 Dynamic Web Module 3.1 선택 -> Java도 Java Build Path 설정에서 지정한 버전과 동일한 버전을 선택한다.
그리고 우측의 Runtimes 탭을 선택하여 Apache Tomcat v8.5를 선택하고 Apply 버튼을 클릭한다.

- Dynamic Web Module을 3.1으로 변경 했다면 web.xml의 <web-app> 태그에서 version과 XML 스키마 정의 파일(xsd)의 버전도 아래와 같이 3.1로 변경하자.

version="3.1"
http://java.sun.com/xml/ns/javaee/web-app_3_1.xsd

- Context root 변경
Web Project Settings 선택 -> 우측의 Context root에 springstudy-ch0401를 입력하고 Apply 버튼을 클릭한다.

5. web.xml에 스프링이 제공하는 프런트 컨트롤러인 DispatcherServlet을 서블릿으로 등록하고 요청 처리를 위한 서블릿 매핑을 설정한다. 기본적으로 Spring MVC Project를 생성하면 Spring MVC에 필요한 bean 설정 파일이 서블릿 초기화 파라미터로 설정되어 있다. 또한 Spring MVC 외에 필요한 bean 설정은 별도의 파일로 분리되어 설정할 수 있도록 웹 애플리케이션 초기화 파라미터와 리스너가 설정되어 있다. 이외에도 추가로 스프링이 제공하는 Character Encoding Filter 등을 설정할 수 있다.

6. Spring Web MVC 설정 파일을 작성한다.
Spring MVC Project를 생성하게 되면 기본적으로 두 개의 스프링 Bean 설정 파일이

생성된다. 하나는 DispatcherServlet이 읽어서 DI 컨테이너를 생성하고 SpringWebMVC에 필요한 Bean을 설정하는데 사용하는 SpringWebMVC 설정용 XML 파일이고 또 다른 하나는 ContextLoaderListener가 읽어서 DI 컨테이너를 생성하고 추가적으로 필요한 Bean을 설정하는데 사용하는 Bean 설정용 XML 파일이다.

대규모 애플리케이션 개발에서는 스프링 설정을 각각의 성격에 맞게 여러 개를 작성하여 관리하는 경우가 대부분이다. 보통은 SpringWebMVC 관련 설정과 이외의 설정을 각각의 용도에 맞게 분할하여 작성한다. 분할한 설정 파일을 스프링이 인식할 수 있도록 지정하는 방법은 아래의 Listener와 웹 애플리케이션 초기화 파라미터, 서블릿 초기화 파라미터를 참고하기 바란다.

- SpringWebMVC 관련 설정(annotation 적용, viewResolver 등등)
- 추가적인 Bean 설정(DBCP, Mybatis, 기타 애플리케이션에서 필요한 Bean 등등)

7. 실제 요청을 처리할 Controller 클래스를 구현한다.
   이때 Controller 클래스와 연동해서 사용되는 Service 계층 클래스, DataAccess 계층 클래스는 Interface 규칙을 적용해 구현하고 View 페이지를 같이 구현한다.
-->

<!-- Listener와 웹 애플리케이션 초기화 파라미터 설정 -->
<!--
ContextLoaderListener는 ServletContextListener를 구현하고
ContextLoader 클래스를 상속한 클래스로 특정 이벤트에 의해서 실행된다.
ServletContextListener 구현체는 ServletContext 인스턴스가 생성될 때
즉 톰캣 서버가 웹 애플리케이션을 로드 할 때 이벤트가 발생하여 실행된다.

ContextLoaderListener는 SpringWebMVC 설정 외에 추가적으로 필요한
Bean 설정 정보를 DispatcherServlet이 실행되기 전에 애플리케이션 초기화
파라미터에서 읽어 Spring DI 컨테이너를 생성하고 필요한 Bean을 생성하는
기능을 제공한다. DispatcherServlet이 실행되면서 SpringWebMVC 설정을
읽어서 Spring DI 컨테이너를 생성하고 SpringWebMVC에 필요한 Bean을
생성할 때 필요한 객체를 미리 생성해야 할 필요가 있거나 스프링 설정 파일을
분리하여 관리하기 위해서 사용하며 ContextLoaderListener도 서블릿과
마찬가지로 web.xml에 등록해야 해당 이벤트가 발생할 때 실행될 수 있다.
Listener는 특정 이벤트에 의해서 실행되기 때문에 리스너 매핑이 필요 없다.

ContextLoaderListener가 실행될 때 읽어야 할 스프링 Bean 설정 파일을
웹 애플리케이션 초기화 파라미터를 통해 지정할 수 있다.
만약 ContextLoaderListener가 읽어야 할 스프링 Bean 설정 파일이 하나가
아니라 여러 개인 경우 아래와 같이 콤마(","), 공백(" "), 탭("\t"), 줄 바꿈("\n"),
세미콜론(";") 등을 사용해 각각의 설정 파일을 구분하여 지정할 수 있으며 이때
웹 애플리케이션 초기화 파라미터 이름은 반드시 contextConfigLocation으로
지정해야 한다.

```
  <context-param>
       <param-name>contextConfigLocation</param-name>
   <param-value>
       /WEB-INF/applicationConfig-context.xml
       classpath:resources/resources-context.xml
       classpath:resources/dataSouce.xml
   </param-value>
  </context-param>

  ContextLoaderListener는 contextConfigLocation이라는 웹 애플리케이션
  초기화 파라미터가 없을 경우 WEB-INF/applicationContext.xml 파일을 찾는다.

  아래는 SpringToolSuite에서  SpringMVCProject를 만들면 기본적으로
  생성해 주는 애플리케이션 초기화 파라미터 설정과 ContextLoaderListener를
  등록하는 설정을 그대로 사용한 것이다.

  이렇게 설정하면 ContextLoaderListener는 톰캣 서버가 웹 애플리케이션을 로드 할
  때 실행되어 웹 애플리케이션 초기화 파라미터의 contextConfigLocation에 지정한
  /WEB-INF/spring/root-context.xml 파일을 읽어서 스프링 DI 컨테이너인
  WebApplicationContext를 생성 한다.
-->
  <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>
  <listener>
      <listener-class>
          org.springframework.web.context.ContextLoaderListener
      </listener-class>
  </listener>

<!--
  DispatcherServlet은 스프링 MVC의 핵심으로 스프링 MVC의 프론트 컨트롤러
  역할을 담당한다. 다른 서블릿과 마찬가지로 DispatcherServlet도 서블릿으로
  동작하기 위해서는 웹 애플리케이션의 배포서술자인 web.xml에 등록해야 한다.
  스프링 MVC에서는 클라이언트로부터 들어오는 모든 요청을 DispatcherServlet이
  받아 각 요청에 대응하는 각각의 처리는 개발자가 구현한 컨트롤러를 통해 처리한다.

  DispatcherServlet이 처음 실행될 때 SpringWebMVC에 필요한 여러 가지
  Bean을 생성할 수 있도록 SpringWebMVC 설정 파일을 자신의 서블릿 초기화
  파라미터에서 읽어와 Spring DI 컨테이너를 생성하는 기능도 제공한다.
  DispatcherServlet이 실행될 때 읽어야 할 SpringWebMVC 설정 파일을
  서블릿 초기화 파라미터를 통해 지정할 수 있다.
  만약 DispatcherServlet이 읽어야할 SpringWebMVC 설정 파일이 하나가
  아니라 여러 개인 경우 아래와 같이 콤마(","), 공백(" "), 탭("\t"), 줄 바꿈("\n"),
```

```
  세미콜론(";") 등을 사용해 각각의 설정 파일을 구분하여 지정할 수 있으며 이때
  서블릿 초기화 파라미터 이름은 반드시 contextConfigLocation으로 지정해야
  한다.

  <servlet>
      <servlet-name>appServlet</servlet-name>
      <servlet-class>
          org.springframework.web.servlet.DispatcherServlet
      </servlet-class>
      <init-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>
              /WEB-INF/mainServlet/spring-context.xml
              classpath:resources/spring-context02.xml
              file:d:\spring\spring-context03.xml
          </param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
  </servlet>

  DispatcherServlet은 contextConfigLocation이라는 서블릿 초기화
  파라미터가 없을 경우 자신의 서블릿 이름인 appServlet에 -servlet.xml을
  추가한 파일을 WEB-INF 디렉터리에서 찾는다. 다시 말해 별도의 설정이 없는
  경우 DispatcherServlet은 WEB-INF/appServlet-servlet.xml 파일을
  읽어서 Spring DI 컨테이너를 생성하게 된다.

  아래는 SpringProject에서  SpringMVCProject로 프로젝트를 생성하면
  Spring STS가 기본적으로 생성해 주는 DispatcherServlet의 서블릿 설정이다.

  아래는 SpringToolSuite에서  SpringMVCProject를 만들면 기본적으로
  생성해 주는 DispatcherServlet의 서블릿 설정을 그대로 사용한 것이다.

  이렇게 설정하면 DispatcherServlet은 자신이 톰캣 서버에 의해 실행 될 때
  자신의 서블릿 초기화 파라미터의 contextConfigLocation에 지정한
  /WEB-INF/spring/appServlet/servlet-context.xml 파일을 읽어서
  스프링 DI 컨테이너인 WebApplicationContext를 생성 한다.
-->
<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>
      org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
          /WEB-INF/spring/appServlet/servlet-context.xml
```

```
        </param-value>
    </init-param>
<!--
    서블릿 클래는 최초 요청이 들어올 때 톰갯 서버에 의해 서블릿 클래스가 로딩되고
    인스턴스화 된다. 그리고 서블릿 초기화 작업이 이루어지고 서블릿 컨테이너에 담겨
    서블릿으로 동작하게 된다. 이렇게 서블릿 클래스는 최초 요청이 들어올 때
    클래스 로딩 -> 인스턴스화 -> 서블릿 초기화  작업을 거치므로 맨 처음 실행될 때
    보통의 실행보다 많은 시간이 걸리게 되는데 이런 문제를 해결하기 위해
    <load-on-startup>을 설정하여 톰캣이 실행되면서 서블릿을 초기화 하도록
    설정할 수 있다. <load-on-startup> 에 지정하는 값이 0 보다 크면 톰갯이
    실행되면서  서블릿을 초기화 하게 되는데 이 값이 0보다 크고 가장 작은 정수 값을
    가진 서블릿이 우선 순위가 제일 높다. 다시 말해 <load-on-startup>에 지정된
    값이 1, 2, 3의 값을 가진 서블릿이 있다면 가장 작은 1의 값을 지정한 서블릿이
    제일 먼저 초기화 된다. 같은 값을 가진 서블릿이 존재 한다면 먼저 정의된 서블릿이
    먼저 초기화 된다.
-->
    <load-on-startup>1</load-on-startup>
</servlet>

<!--
    아래는 ContextRoot 들어오는 .jsp 요청을 제외한 모든 요청을
    DispatcherServlet이 받을 수 있도록 url-pattern을 지정한 것이다.
-->
<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- 스프링 MVC가 제공하는 인코딩용 필터 정의 -->
<!--
    CharacterEncodingFilter 클래스를 사용하면 폼 입력으로
    넘어오는 요청 파라미터 데이터를 지정한 문자 셋으로 처리해 준다.

    get 방식의 요청은 톰캣 서버의 servlet.xml에 지정한 문자 셋으로 처리되고
    post 방식의 요청은 별도로 문자 셋 처리 코드를 작성하지 않아도 된다.
-->
 <filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
```

```
    <!--
        기존에 문자 셋이 설정되어 있다 하더라도 request, response에
        강제로 위에서 지정한 문자 셋으로 인코딩을 설정하라고 지정하는 셋팅
        즉 getCharacterEncoding()을 호출해 null이 아니라 하더라도
        request와 response에 utf-8 문자 셋을 강제로 설정한다.
    -->
    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
  </filter>
  <!--
    Servlet과 마찬가지로 Filter도 <filter-mappign> 태그를 사용해
    필터를 매핑하며 <url-pattern> 태그에 지정한 패턴에 따라서 실행된다.
  -->
  <filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

▶ Spring MVC 설정

- src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
    <!--
        Annotation 기법을 이용한 SpringMVC 설정 파일

        이 설정 파일은 Spring MVC 관련 설정 파일로 MVC 위주의 설정이 많기 때문에
        xmlns를 지정할 때 mvc에는 접두어를 사용하지 않았다. 그래서 mvc 네임스페이스는
        접두어를 사용하지 않은 태그를 사용해야 한다. 나머지 네임스페이스는 접두어를
        사용해 xmlns지정했기 때문에 접두어를 붙여 태그를 사용해야 한다.

        <annotation-driven /> 태그는 Spring MVC에서 필요한 Annotation 기반
        모든 기능을 사용할 수 있는 설정으로 이 태그는 JSR-303 검증 지원, 메시지 변환,
```

```
    필드 포매팅 지원 등을 포함한 강력하고 다양한 기능을 제공한다.

    <context:component-scan>에 의해 컨트롤러가 스프링 DI 컨테이너에
    등록되고 @Controller Annotation이 지정된 클래스의 객체가 컨트롤러라는
    것을 판단하고 Spring MVC의 다양한 Annotation을 사용하기 위한 설정으로
    Spring MVC의 Annotation을 사용할 때는 필수라고 생각하면 되겠다.

    <annotation-driven /> 태그는 아래 두 클래스를 스프링 빈으로 등록하여
    @Controller 애노테이션이 지정된 클래스를 컨트롤러로 사용할 수 있도록 해준다.

    * org.springframework.servlet.mvc.method.annotation 패키지의
        - RequestMappingHandlerMapping
        - RequestMappingHandlerAdapter

    또한 이 태그는 JSON이나 XML에 대한 요청과 응답 처리를 위한 모듈이나 파라미터
    데이터 바이딩 처리를 위한 ConversionService 등을 스프링 빈으로 등록 해준다.
-->
<annotation-driven />

<!--
    DispatcherServlet을 경유해 정적 리소스에 접근하기 위한 설정으로
    정적 리소스란 Html, 이미지, CSS, JavaScript 파일 등을 의미한다.

    아래 설정은 ContextRoot/resources/ 로 들어오는 정적 리소스 요청에
    대해 ContextRoot/resources/ 디렉터리를 매핑한 것이다. 이 설정은
    DispatcherServlet의 url-pattern을 "/"로 지정하여 DispatcherServlet이
    정적 컨텐츠를 포함한 모든 요청을 처리할 수 있도록 설정해야 제대로 동작한다.
-->
<resources mapping="/resources/**" location="/resources/" />

<!--
    <context:component-scan /> 태그는 base-package에 지정한 패키지를
    기준으로 스프링이 자동으로 bean을 스캔하여 선언하고 생성해 주는 설정이다.

    다시 말해 Spring bean 설정 파일에 <bean> 태그를 사용해 클래스를 등록하지
    않아도 스프링프레임워크가 아래 Annotation이 적용된 클래스를 찾아 annotation
    기반으로 스프링 DI 컨테이너에 등록하고 클래스 간의 관계를 맺어 주는 설정이다.

    컨트롤러 클래스(@Controller), DAO 클래스(@Repository),
    Service(@Service) 클래스가  스프링에 의해 스프링 DI 컨테이너에 등록된다.
-->
<context:component-scan base-package="com.springstudy.ch04" />

<!--
    ## 1. HandlerMapping 정의하기 ##
```

```
    HandlerMapping은 클라이언트가 보낸 웹 요청 URL과 그 요청을 처리할
    컨트롤러를 매핑해 주는 클래스로 최초로 요청을 받은 DispatcherServlet이
    그 요청을 어느 컨트롤러로 보내야 할지 HandlerMapping에게 의뢰해 요청을
    처리할 컨트롤러를 선택하게 된다. HandlerMapping은 요청 URL을 참고해
    요청을 처리할 컨트롤러를 결정하여 DispatcherServlet에게 전달하게 된다.
    HandlerMapping을 지정하지 않으면 BeanNameUrlHandlerMappging과
    DefaultAnnotationHandlerMapping이 기본 적용된다.

    하지만 스프링 설정 파일에 <annotation-driven /> 태그를 설정하게 되면
    @Controller 애노테이션이 지정된 클래스를 컨트롤러로 사용할 수 있도록 해주는
    RequestMappingHandlerMapping과 RequestMappingHandlerAdapter를
    스프링 빈으로 등록하게 되므로 이 예제는 이 두 클래스를 우선 사용되게 된다.
-->
<!--
    ## 2. Controller 정의하기 ##
    <annotation-driven/>과 <context:component-scan />을 적용했기
    때문에 Spring MVC Annotation 기반의 Bean을 선언하고 검색하게 되므로
    우리가 구현하는 Controller 클래스에 @Controller 애노테이션을 적용하면
    Spring DI 컨테이너가 Bean을 스캔하여 Controller로 등록해 준다.
-->
<!--
    ## 3. ViewResolver 정의하기 ##
    컨트롤러로 부터 요청에 대한 처리 결과를 전달 받은 DispatcherServlet은
    뷰 리졸버에게 논리적인 뷰 이름을 실제 뷰 구현체(JSP등)에 매핑하도록 요청하고
    뷰 리졸버는 요청 결과를 표시할 실제 뷰 객체를 DispatcherServlet에게
    전달한다. DispatcherServlet은 뷰 리졸버로 부터 전달 받은 뷰 객체에
    컨트롤러로 부터 받은 모델 정보를 렌더링 하고 그 결과를 클라이언트에 응답한다.

    뷰 리졸버는 컨트롤러로 부터 전달 받은 논리적인 뷰 이름을 가지고 실제 응답
    결과를 생성할 뷰 객체를 구할 때 사용되며 실제로 데이터가 출력될 뷰를 선택하는
    역할을 한다. 만약 컨트롤러로부터 반환된 논리적인 뷰의 이름이 main이라면
    뷰 리졸버는 prefix와 suffix를 조합해 다음과 같은 물리적인 뷰의 이름을 만든다.
    뷰가 결정되면 DispatcherServlet에서 이 뷰로 포워딩 된다.

    /WEB-INF/index.jsp?body=views/main.jsp

    뷰 리졸버 구현체는 몇 가지가 있으며 뷰 리졸버를 별도로 지정하지 않으면
    InternalResourceViewResolver가 기본 적용된다.
    뷰 리졸버의 viewClass 속성에 뷰 리졸버가 생성할 뷰 객체를 지정할 수 있는데
    이를 생략하면 InternalResourceView가 기본 적용된다.
    InternalResourceView는 JSP, HTML과 같이 웹 애플리케이션의 내부 자원을
    사용해 응답 결과를 만드는 객체로 속성에 모델을 지정하고 RequestDispatcher를
    이용해 지정된 뷰로 포워딩 된다.
-->
<beans:bean class=
```

```
        "org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/index.jsp?body=views/" />
        <beans:property name="suffix" value=".jsp" />
    </beans:bean>

    <!--
        클라이언트의 요청이 특별한 처리 없이 단순히 뷰만 보여줘야 할 경우에
        아래와 같이 뷰 전용 컨트롤러를 설정하여 뷰 페이지를 지정할 수 있다.

        <view-controller /> 태그는 뷰 전용 컨트롤러를 설정하는 태그로
        어떤 요청이 특별한 비즈니스 로직 처리 없이 단순히 뷰만 보여줘야 할 필요가
        있을 때 유용하게 사용할 수 있는 설정이다. 아래는 /default, /main으로
        들어오는 요청에 대한 뷰를 main으로 지정한 예로 이 설정 파일 맨 아래에
        bean으로 등록한 ViewResolver의 prefix와 suffix가 적용된 뷰를 보여주는
        설정이다. 다시 말해 view-name 속성에 지정하는 뷰의 이름은 ViewResolver에서
        prefix, suffix에 지정한 정보를 제외한 나머지를 지정하면 된다.

        여러 개의 요청 URI에 같은 뷰를 적용하려면 아래와 같이 여러 번 지정하면 된다.
    -->
    <view-controller path="/first/default" view-name="main" />
    <view-controller path="/first/main" view-name="main" />
</beans:beans>
```

▶ Spring Bean 설정

- src/main/webapp/WEB-INF/spring/root-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!--
        Annotation 기법을 이용한 Controller 클래스가 의존하는
        Service 계층의 클래스와 DataAccess 계층의 클래스를 스프링 빈으로 설정

        이외에도 스프링 Bean 설정 파일을 분할하여 작성하였다면 import를 통해
        아래와 같이 이 설정 파일에 포함 시킬 수 있다.

        <import resource="classpath:datasource/datasource.xml" />
    -->
    <!--
        ## Service 정의하기 ##
        ## Dao 정의하기 ##
        servlet-context.xml 파일에 <context:component-scan />을 적용했기
        때문에 Spring MVC Annotation 기반의 Bean을 선언하고 검색하게 되므로
```

```
        별도로 Service 클래스와 Dao 클래스를 이 Bean 설정 파일에 정의하지 않고
        서비스 클래스에는 @Service 애노테이션과 Dao 클래스에는 @Repository
        애노테이션을 적용시키면 Spring DI 컨테이너가 Bean을 스캔해 등록한다.
    -->
</beans>
```

▶ 데이터 액세스 계층(DAO) 클래스

- com.springstudy.ch04.dao

```
public interface FirstMvcDao {
    public abstract String getMessage(int no, String id);
}
```

- com.springstudy.ch04.dao

```
// 이 클래스가 데이터 액세스(데이터 저장소) 계층의 컴포넌트(Bean) 임을 선언한다.
@Repository
public class FirstMvcDaoImpl implements FirstMvcDao {

    @Override
    public String getMessage(int no, String id) {

        String msg = null;

        if(id.equals("spring")) {
            msg = "안녕 스프링 - " + no;
        } else {
            msg = "Hello Spring - " + no;
        }

        return msg;
    }
}
```

▶ Service 계층 클래스

- com.springstudy.ch04.service

```
public interface FirstMvcService {
    public abstract String getMessage(int no, String id);
}
```

- com.springstudy.ch04.service

```
//이 클래스가 서비스 계층(비즈니스 로직)의 컴포넌트(Bean) 임을 선언하고 있다.
@Service
public class FirstMvcServiceImpl implements FirstMvcService {
```

```
    /*  인스턴스 필드에 @Autowired annotation을 사용하면 접근지정자가
     * private이고 setter 메서드가 없다 하더라도 문제없이 주입 된다.
     * 하지만 우리는 항상 setter 메서드를 준비하는 습관을 들일 수 있도록 하자.
     *
     * setter 주입 방식은 스프링이 기본 생성자를 통해 이 클래스의 인스턴스를
     * 생성한 후 setter 주입 방식으로 BoardDao 타입의 객체를 주입하기 때문에
     * 기본 생성자가 존재해야 하지만 이 클래스에 다른 생성자가 존재하지 않으므로
     * 컴파일러에 의해 기본 생성자가 만들어 진다.
     **/
    @Autowired
    private FirstMvcDao firstMvcDao;

    public void setFirstMvcDao(FirstMvcDao firstMvcDao) {
        this.firstMvcDao = firstMvcDao;
    }

    @Override
    public String getMessage(int no, String id) {
        return firstMvcDao.getMessage(no, id);
    }
}
```

▶ Controller 클래스

- com.springstudy.ch04.controller

```
//스프링 MVC의 컨트롤러임을 선언하고 있다.
@Controller

/* @RequestMapping을 이용해 처리할 요청 경로를 지정한다.
 * @RequestMapping은 클래스 레벨과 메서드 레벨에 지정할 수 있다.
 * 아래는 "ContextRoot/first"로 들어오는 요청을 이 컨트롤러가 처리할 수 있도록
 * 설정한 것이다. 이 컨트롤러의 모든 메서드는 "ContextRoot/first/*"로
 * 들어오는 요청만을 처리할 수 있다.
 **/
@RequestMapping("/first")
public class FirstMvcController {

    /*  인스턴스 필드에 @Autowired annotation을 사용하면 접근지정자가
     * private이고 setter 메서드가 없다 하더라도 문제없이 주입 된다.
     * 하지만 우리는 항상 setter 메서드를 준비하는 습관을 들일 수 있도록 하자.
     *
     * setter 주입 방식은 스프링이 기본 생성자를 통해 이 클래스의 인스턴스를
     * 생성한 후 setter 주입 방식으로 BoardService 타입의 객체를 주입하기 때문에
     * 기본 생성자가 존재해야 하지만 이 클래스에 다른 생성자가 존재하지 않으므로
     * 컴파일러에 의해 기본 생성자가 만들어 진다.
```

```
 **/
@Autowired
private FirstMvcService service;

public void setFirstMvcService(FirstMvcService service) {
    this.service = service;
}

/* @RequestMapping 애노테이션이 적용된 메서드의 파라미터와 반환 타입
 *
 * 1. 메서드의 파라미터 타입으로 지정할 수 있는 객체와 애노테이션
 * @RequestMapping 애노테이션이 적용된 컨트롤러 메서드의 파라미터에
 * 아래와 같은 객체와 애노테이션을 사용할 수 있도록 지원하고 있다.
 *
 * - HttpServletRequest, HttpServletResponse
 *   요청/응답을 처리하기 위한 서블릿 API
 *
 * - HttpSession
 *   HTTP 세션을 위한 서블릿 API
 *
 * - org.springframework.ui.Model, ModelMap, java.util.Map
 *   뷰에 모델 데이터를 전달하기 위한 모델 객체
 *
 * - 커맨드 객체(VO, DTO)
 *   요청 데이터를 저장할 객체
 *
 * - Errors, BindingResult
 *   검증 결과를 저장할 객체로 커맨드 객체 바로 뒤에 위치 시켜야 한다.
 *
 * - @RequestParam
 *   HTTP 요청 파라미터의 값을 메서드의 파라미터로 매핑하기 위한 애노테이션
 *
 * - @RequestHeader
 *   HTTP 요청 헤더의 값을 파라미터로 받기 위한 애노테이션
 *
 * - @RequestCookie
 *   Cookie 데이터를 파라미터로 받기 위한 애노테이션
 *
 * - @RequestVariable
 *   RESTful API 방식의 파라미터를 받기 위한 경로 변수 설정 애노테이션
 *
 * - @RequestBody
 *   요청 몸체의 데이터를 자바 객체로 변환하기 위한 애노테이션
 *   String이나 JSON으로 넘어오는 요청 몸체의 데이터를 자바 객체로
 *   변환하기 위한 사용하는 애노테이션 이다.
```

```
 *
 * - Writer, OutputStream
 *   응답 데이터를 직접 작성할 때 메서드의 파라미터로 지정해 사용한다.
 *
 * 2. 메서드의 반환 타입으로 지정할 수 있는 객체와 애노테이션
 * - String
 *   뷰 이름을 반환할 때 메서드의 반환 타입으로 지정
 *
 * - void
 *   컨트롤러의 메서드에서 직접 응답 데이터를 작성할 경우 지정
 *
 * - ModelAndView
 *   모델과 뷰 정보를 함께 반환해야 할 경우 지정
 *   이전의 컨트롤는 스프링이 지원한는 Controller 인터페이스를
 *   구현해야 했는데 이때 많이 사용하던 반환 타입이다.
 *
 * - 자바 객체
 *   메서드에 @ResponseBody가 적용된 경우나 메서드에서 반환되는
 *   객체를 JSON 또는 XML과 같은 양식으로 응답을 변환 할 경우에 사용한다.
 **/

/* @RequestMapping은 클래스 레벨과 메서드 레벨에 지정할 수 있다.
 * @RequestMapping의 ()에 처리할 요청 URI만 지정할 때는 value 속성을
 * 생략하고 처리할 요청 URI를 String 또는 String 배열을 지정할 수 있지만
 * 다른 속성을 같이 지정할 경우 value 속성에 처리할 요청 URI를 지정해야 한다.
 * 또한 method 속성을 지정해 컨트롤러가 처리할 HTTP 요청 방식을 지정할 수
 * 있으며 method 속성을 생략하면 기본 값은 RequestMethod.GET 이다.
 *
 * 아래는 "ContextRoot/first/", "ContextRoot/first/intro",
 * "ContextRoot/first/index"로 들어오는 GET 방식 요청을 이 메서드가 처리할 수
 * 있도록 설정한 것이다. 아래와 같이 비즈니스 로직 처리 없이 단순히 뷰만
 * 보여줘야 할 경우 Spring Web MVC 설정에서<view-controller /> 태그를
 * 사용해 뷰 페이지를 지정하면 편리하게 뷰 전용 컨트롤러를 설정할 수 있다.
 **/
@RequestMapping(value={"/", "/intro", "/index"}, method=RequestMethod.GET)
public String index() {

    /* servlet-context.xml에 설정한 ViewResolver에서 prefix와 suffix에
     * 지정한 정보를 제외한 뷰 이름을 문자열로 반환하면 된다.
     *
     * 아래와 같이 뷰 이름을 반환하면 포워드 되어 제어가 뷰 페이지로 이동한다.
     **/
    return "/main";
}
```

```
/* 아래는 "/detail"로 들어오는 GET 방식 요청을 처리하는 메서드를 지정한 것이다.
 * method 속성을 생략했기 때문에 RequestMethod.GET이 적용된다.
 *
 * 스프링은 클라이언트로 부터 넘어 오는 요청 파라미터를 받을 수 있는 여러 가지
 * 방법을 제공하고 있다. 아래와 같이 Controller 메서드에 요청 파라미터 이름과
 * 동일한 이름의 메서드 파라미터를 지정하면 스프링으로부터 요청 파라미터를 넘겨
 * 받을 수 있다. 만약 요청 파라미터와 메서드의 파라미터 이름이 다른 경우에는
 * 메서드의 파라미터 앞에 @RequestParam("요청 파라미터 이름")을 사용해
 * 요청 파라미터의 이름을 지정하면 된다.
 *
 * @RequestMapping 애노테이션이 적용된 Controller 메서드의 파라미터에
 * @RequestParam 애노테이션을 사용해 요청 파라미터 이름을 지정하면
 * 이 애노테이션이 앞에 붙은 매개변수에 요청 파라미터 값을 바인딩 시켜준다.
 *
 * @RequestParam 애노테이션에 사용할 수 있는 속성은 아래와 같다.
 * value : HTTP 요청 파라미터의 이름을 지정한다.
 * required : 요청 파라미터가 필수인지 설정하는 속성으로 기본 값은 true 이다.
 *           이 값이 true인 상태에서 요청 파라미터의 값이 존재하지 않으면
 *           스프링은 Exception을 발생시킨다.
 * defaultValue : 요청 파라미터가 없을 경우 사용할 기본 값을 문자열로 지정한다.
 *
 * @RequestParam(value="no" required=false defaultValue="1")
 *
 * @RequestParam 애노테이션은 요청 파라미터 값을 읽어와 Controller 메서드의
 * 파라미터 타입에 맞게 변환해 준다. 만약 요청 파라미터를 Controller 메서드의
 * 파라미터 타입으로 변환할 수 없는 경우 스프링은 400 에러를 발생시킨다.
 *
 * @RequestMapping 애노테이션이 적용된 Controller 메서드의 파라미터
 * 이름과 요청 파라미터의 이름이 같은 경우 @RequestParam 애노테이션을
 * 지정하지 않아도 스프링으로부터 요청 파라미터를 받을 수 있다.
 *
 * 요청을 처리한 결과를 뷰에 전달하기 위해 사용하는 것이 모델이다.
 * 컨트롤러는 요청을 처리한 결과 데이터를 모델에 담아 뷰로 전달하고 뷰는
 * 모델로 부터 데이터를 읽어와 클라이언트로 보낼 결과 페이지를 만들게 된다.
 *
 * 스프링은 컨트롤러에서 모델에 데이터를 담을 수 있는 다양한 방법을 제공하는데
 * 아래와 같이 파라미터에 Model을 지정하는 방식이 많이 사용된다.
 * @RequestMapping 애노테이션이 적용된 메서드의 파라미터에 Model
 * 을 지정하면 스프링이 이 메서드를 호출하면서 Model 타입의 객체를 넘겨준다.
 * 우리는 Model을 받아 이 객체에 결과 데이터를 담기만 하면 뷰로 전달된다.
 **/
@RequestMapping("/detail")
public String detail(Model model, HttpServletRequest request,
        @RequestParam(value="num", defaultValue="1") int no, String id) {
```

```
        // HttpServletRequest 객체로 이용해 요청 파라미터를 직접 받을 수도 있다.
        // String id = request.getParameter("id");

        // FirstMvcService 객체를 이용해 메시지를 가져와 Model에 담는다.
        String msg = service.getMessage(no, id);
        model.addAttribute("msg", msg);

        // 뷰로 보낼 필요한 데이터가 있으면 추가로 Model에 담는다.
        model.addAttribute("title", "명준이의 디테일 당부");
        model.addAttribute("comment", "정말 루~ 스프링 공부 열심히 해야 해여~ %_%");
        return "detail";
    }
}
```

▶ 웹 템플릿 View

*** 메인 템플릿 페이지**

- WebContent/WEB-INF/index.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>스프링 MVC 첫 번째 예제</title>
<!--
    FirstMvcController 클래스 위에 @RequestMapping("/first")를 설정하고
    index() 메서드에 @RequestMapping({"/", "/intro", "/index"})으로 설정했다.
    그리고 WEB-INF/spring/appServlet/servlet-context.xml에서 정적 리소스와
    관련된 url 맵핑을 아래와 같이 설정했기 때문에
    <mvc:resources mapping="/resources/**" location="/resources/" />
    css의 위치를 "../resources/css/index.css"와 같이 지정해야 한다.

    브라우저 주소 표시줄에 http://localhost:8080/springstudy-ch0401/first/,
    또는 http://localhost:8080/springstudy-ch0401/first/intro,
    또는 http://localhost:8080/springstudy-ch0401/first/index 등으로
    표시되므로 css 디렉터리는 ContextRoot/resources/css에 위치하기 때문에 현재
    위치를 기준으로 상대 참조 방식으로 "../resources/css/index.css"를 지정해야 한다.
-->
<link rel="stylesheet" type="text/css"
    href="../resources/css/index.css" />
</head>
<body>
    <%@ include file="template/header.jsp" %>
    <jsp:include page="${ param.body }" />
```

```
    <%@ include file="template/footer.jsp" %>
</body>
</html>
```

*** 템플릿 헤더 페이지**

- WebContent/WEB-INF/template/header.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<header>
    <div id="logo">
        <a href="${ pageContext.servletContext.contextPath }/first/main">
        <img src="https://via.placeholder.com/176x67" alt="Books2U" width="176"
            height="67" /></a></div>
    <div id="header_link">
        <ul>
            <li>
                <a href='#'>로그인</a>
            </li>
            <li>
                <a href="#">게시글 리스트</a>
            </li>
            <li>
                <c:if test="${ not sessionScope.isLogin }" >
                    <a href="#">회원가입</a>
                </c:if>
                <c:if test="${ sessionScope.isLogin }" >
                    <a href="#">정보수정</a>
                </c:if>
            </li>
            <li><a href="#">주문/배송조회</a></li>
            <li class="no_line"><a href="#">고객센터</a></li>
        </ul>
    </div>
</header>
```

*** 템플릿 푸터 페이지**

- WebContent/WEB-INF/template/footer.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<footer>
    <div id="footer_info">
        고객상담 전화주문:1234-5678 사업자등록번호 :111-11-123456
```

```
        대표이사: 홍길동  통신판매업 서울 제 000000호<br/>
        개인정보관리책임자:임꺽정  분쟁조정기관표시  :  소비자보호원,  전자거래분쟁중재위원회
<br/>
        Copyright (c) 2016 Spring2U Corp. All right Reserved.
    </div>
</footer>
```

▶ Spring MVC 첫 번째 View

- WebContent/WEB-INF/views/main.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<article>
    <h1>명준이의 당부</h1>
    <!--
        FirstMvcController 클래스 위에 @RequestMapping("/first")를 설정하고
        index() 메서드에 @RequestMapping({"/", "/intro", "/index"})으로 설정했다.
        그리고 WEB-INF/spring/appServlet/servlet-context.xml에서 정적 리소스와
        관련된 url 맵핑을 아래와 같이 설정했기 때문에
        <mvc:resources mapping="/resources/**" location="/resources/" />
        이미지의 위치를 "../resources/images/myungjun01.jpg"와 같이 지정해야 한다.

        브라우저 주소 표시줄에 http://localhost:8080/springstudy-ch0401/first/,
        또는 http://localhost:8080/springstudy-ch0401/first/intro,
        또는 http://localhost:8080/springstudy-ch0401/first/index 등으로 표시되므로
        images 디렉터리는 ContextRoot/resources/images에 위치하기 때문에 현재 위치를
        기준으로 상대 참조 방식으로 "../resources/images/myungjun01.jpg"를 지정해야 한다.
    -->
    <div id="photo">
        <img src="../resources/images/myungjun01.jpg" id="img"/>
    </div>
    <div id="comment">
        <p>
            삼촌~ 이모들~ 스프링 공부 열심히 하삼~~<br/>
            <a href="detail?id=spring&no=20">디테일 당부보기</a>
        </p>
    </div>
</article>
```

- WebContent/WEB-INF/views/detail.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<article>
    <!--
```

```
        컨트롤러에서 Model에 담은 데이터는 뷰 페이지에서 EL로 바로 접근이 가능하다.
        우리가 컨트롤러에서 Model에 담은 데이터는 스프링에 의해서 request 객체에
        저장되기 때문에 JSP에서와 같이 뷰 페이지에서 EL로 바로 접근이 가능하다.
    -->
    <h1 id="bigH1">${title}</h1>
    <div id="bigPhoto">
        <img src="../resources/images/myungjun01.jpg" id="bigImg"/>
    </div>
    <div id="bigComment">
        <p>${comment}</p><br/>
        <h3>모델로 받은 - 메시지</h3>
        msg : ${ msg }
    </div>
</article>
```

▶ 스타일(CSS)

- WebContent/WEB-INF/resources/css/index.css

```
@CHARSET "UTF-8";
* {
    margin: 0;
    padding: 0;
}
body {
    width: 980px;
    margin: 0px auto;
    font-size: 12px;
}

.clear {
    clear: both;
}

/* 링크 설정 */
a:link {
    text-decoration: none;
    color: #333;
}
 a:visited {
    text-decoration: none;
    color: #333;
}
a:hover {
    text-decoration: underline;
    color: red;
```

```
}
a:active {
    text-decoration: none;
    color: blue;
}

/* header */
header {
    margin-top: 20px;
    border-bottom: 1px dashed blue;
    height: 80px;
}
header img { border: none; }
header #logo {
    margin-left: 10px;
    float: left;
}
header #header_link {
    float: right;
    margin-right: 10px;
    width: 500px;
}
header #header_link span {
    display: inline-block;
    padding: 0 8px;
    border-right: 1px solid #999;
}
header #header_link ul { float: right; list-style: none; }
header #header_link li {
    float: left;
    padding: 0 8px;
    border-right: 1px solid #999;
}
header #header_link ul li.no_line { border-right: none; }

/* footer */
footer {
    border-top: 1px dotted blue;
}
footer #footer_info {
    margin: 10px auto;
    line-height: 1.5em;
    text-align: center;
    width: 600px;
}
```

```
/* content */
article{
    margin: 30px auto 0px;
    height: 750px;
}

/* 명준 intro 페이지 */
#photo, #comment, h1 {
    width: 400px;
    margin: 50px auto;
}
#img {
    width: 400px;
    border-radius: 20px 20px;
    box-shadow: 5px 5px 5px #ACACAC;
}

/* 명준 detail 페이지 */
#bigPhoto, #bigComment, #bigH1 {
    width: 600px;
    margin: 50px auto;
}
#bigImg {
    width: 600px;
    border-radius: 30px 30px;
    box-shadow: 7px 7px 7px #ACACAC;
}
```

[연습문제 4-1] 다음 요구사항에 알맞은 애플리케이션을 구현하시오.

1. 앞에서 실습한 springstudy-ch02 프로젝트의 MemberDAO 인터페이스와 MemberDAOImpl 클래스를 springstudy-ch0401 프로젝트의 dao 패키지에 복사하여 가져온 후 MemberDAOImpl 클래스를 Annotation 방식으로 스프링 Bean으로 등록하고 이 MemberDAOImpl 클래스에서 사용하는 DriverManagerDataSource 객체를 Annotation 방식으로 setter 주입 받을 수 있도록 설정하시오.
2. FirstMvcServiceImpl에서 MemberDAO 타입의 객체를 Annotation 방식으로 setter 주입 받을 수 있도록 설정하고 FirstMvcController 클래스에 회원 리스트 요청을 처리하는 memberList() 메서드를 새롭게 추가하여 ContextRoot/first/mList와 ContextRoot/first/memberList로 들어오는 두 가지 요청을 처리할 수 있도록 요청 매핑(RequestMapping)을 설정 하시오.
3. 요청을 처리한 결과를 WEB-INF/views/memberList.jsp를 만들어 출력하시오.