
# Spring Framework

*Spring Framework 소개*

# 객체 지향 프로그래밍
# (Object Oriented Programming)

- 객체 지향 프로그래밍이란?
  - 객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이다.
  - 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있다.
  - 객체 지향 프로그래밍은 프로그램을 유연하고 변경이 용이하게 만들기 때문에 대규모 소프트웨어 개발에 많이 사용된다.
  - 또한 프로그래밍을 더 배우기 쉽게 하고 소프트웨어 개발과 보수를 간편하게 하며, 보다 직관적인 코드 분석을 가능하게 하는 장점을 갖고 있다.
  - 그러나 지나친 프로그램의 객체화 경향은 실제 세계의 모습을 그대로 반영하지 못한다는 비판을 받기도 한다.

# 객체 지향 프로그래밍 언어의 특징

- 자료 추상화
  - 자료 추상화는 불필요한 정보는 숨기고 중요한 정보만을 표현함으로써 프로그램을 간단히 만드는 것이다.
  - 자료 추상화를 통해 정의된 자료형을 추상 자료형이라고 한다.
  - 추상 자료형은 자료형의 자료 표현과 자료형의 연산을 캡슐화한 것으로 접근 제어를 통해서 자료형의 정보를 은닉할 수 있다.
  - 객체 지향 프로그래밍에서 일반적으로 추상 자료형을 클래스, 추상 자료형의 인스턴스를 객체, 추상 자료형에서 정의된 연산을 메소드(함수)라고 한다.

# 객체 지향 프로그래밍 언어의 특징

- 상속
  - 상속은 새로운 클래스가 기존의 클래스의 자료와 연산을 이용할 수 있게 하는 기능이다.
  - 상속을 받는 새로운 클래스를 부클래스, 파생 클래스, 하위 클래스, 자식 클래스라고 하며 새로운 클래스가 상속하는 기존의 클래스를 기반 클래스, 상위 클래스, 부모 클래스라고 한다.
  - 상속을 통해서 기존의 클래스를 상속받은 하위 클래스를 이용해 프로그램의 요구에 맞추어 클래스를 수정할 수 있고 클래스 간의 종속 관계를 형성함으로써 객체를 조직화 할 수 있다.

# 객체 지향 프로그래밍 언어의 특징

- 다중 상속
  - 다중 상속은 클래스가 2개 이상의 클래스로부터 상속받을 수 있게 하는 기능이다.
  - 클래스들의 기능이 동시에 필요할 때 용이하나 클래스의 상속 관계에 혼란을 줄 수 있고(예: 다이아몬드 상속) 프로그래밍 언어에 따라 사용 가능 유무가 다르므로 주의해서 사용해야 한다.
  - JAVA는 일반 객체의 상속에선 지원하지 않는다.

# 객체 지향 프로그래밍 언어의 특징

- 다형성 개념
  - 다형성 개념이란 어떤 한 요소에 여러 개념을 넣어 놓는 것으로 일반적으로 오버라이딩(같은 이름의 메소드가 여러 클래스에서 다른 기능을 하는 것)이나 오버로딩(같은 이름의 메소드가 인자의 갯수나 자료형에 따라서 다른 기능을 하는 것)을 의미한다.
  - 다형 개념을 통해서 프로그램 안의 객체 간의 관계를 조직적으로 나타낼 수 있다.

# 객체 지향 프로그래밍 언어의 특징

- 동적 바인딩
  - 동적 바인딩은 실행 시간 중에 일어나거나 실행 과정에서 변경될 수 있는 바인딩
  - 컴파일 시간에 완료되어 변화하지 않는 정적 바인딩과 대비되는 개념이다.
  - 동적 바인딩은 프로그램의 한 개체나 기호를 실행 과정에 여러 속성이나 연산에 바인딩함으로써 다형 개념을 실현한다.
  - 런타임까지 타입에 대한 결정을 끌고 갈 수 있기 때문에 유연성이 탁월하다.
  - 하지만 실행 도중에 변수에 예상치 못한 타입이 들어오는 경우가 생겨 안정성이 떨어지는 경우가 있다.
  - 또한 유연한 변화 때문에 타입에 대한 체크로 인한 수행속도 저하와 메모리의 공간 낭비, 메모리 점유의 잦은 변화를 위해 일어나는 높은 오버헤드가 단점으로 꼽힐 수 있다.

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 객체지향 프로그래밍은 장점도 있지만 단점도 존재하는 프로그래밍 기술
- 보다 나은 객체지향 프로그래밍을 위해 5개의 대 원칙을 제시함
  - 단일 책임 원칙 (Single Responsibility Principle)
  - 개방 폐쇄 원칙 (Open-Closed Principle)
  - 리스코프 치환 원칙(Liskov Substitution Principle)
  - 인터페이스 분리 원칙(Interface Segregation Principle)
  - 의존성 역전 원칙(Dependency Inversion Principle)

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 단일 책임 원칙 (Single Responsibility Principle)
  - 작성된 클래스는 하나의 기능만 가지며 클래스가 제공하는 모든 서비스는 그 하나의 책임(변화의 축: axis of change)을 수행하는 데 집중되어 있어야 한다는 원칙.
  - 어떤 변화에 의해 클래스를 변경해야 하는 이유는 오직 하나뿐이어야 함.
  - SRP원리를 적용하면 무엇보다도 책임 영역이 확실해지기 때문에 한 책임의 변경에서 다른 책임의 변경으로의 연쇄작용에서 자유로울 수 있다.
  - 뿐만 아니라 책임을 적절히 분배함으로써 코드의 가독성 향상, 유지보수 용이라는 이점까지 누릴 수 있으며 객체지향 원리의 대전제 격인 OCP 원리뿐 아니라 다른 원리들을 적용하는 기초가 된다.

# 객체지향 프로그래밍 5대 원칙 (SOLID)

## 단일 책임 원칙 (Single Responsibility Principle)

```java
public class Employee{
  private String empId;
  private String name;
  private string address;

  public boolean isPromotionDueThisYear(){
    //promotion logic
  }

  //Getters & Setters
}
```

```java
public class Promotions{
  public boolean isPromotionDueThisYear(Employee emp){
    //promotion logic
  }
}

public class Employee{
  private String empId;
  private String name;
  private string address;

  //Getters & Setters
}
```

종업원 클래스가 직원의 핵심 속성을 유지하는 단일 책임을 져야 하기 때문에 단일 책임 원칙을 따르지 않는다.
그 직원의 승진 기한을 결정하는 것은 직원 클래스의 책임이 아니다.

---

더 나은 해결책:
우리는 승진 계산 논리를 별도로 포함하는 새로운 클래스를 만들 수 있다. 현재 계급은 직원 및 승진 모두가
단일 책임을 가지기 때문에 이 접근법은 단일 책임 원칙을 따를 것이다.

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 개방 폐쇄 원칙 (Open-Closed Principle)
  - 소프트웨어의 구성요소(컴포넌트, 클래스, 모듈, 함수)는 확장에는 열려있고, 변경에는 닫혀 있어야 한다는 원리
  - 변경을 위한 비용은 가능한 줄이고 확장을 위한 비용은 가능한 극대화 해야 한다는 의미
  - 요구사항의 변경이나 추가사항이 발생하더라도, 기존 구성요소는 수정이 일어나지 말아야 하며, 기존 구성요소를 쉽게 확장해서 재사용할 수 있어야 한다.
  - OCP를 가능케 하는 중요 메커니즘은 추상화와 다형성

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 개방 폐쇄 원칙 (Open-Closed Principle)

```java
public class Rectangle{
 public double length;
 public double width;
}
```

```java
public class Circle{
 public double radius;
}
```

```java
public class AreaCalculator{
  public double calculateRectangleArea(Rectangle rectangle){
    return rectangle.length *rectangle.width;
  }
  public double calculateCircleArea(Circle circle){
    return (22/7)*circle.radius*circle.radius;
  }
}
```

```java
public interface Shape{
  public double calculateArea();
}

public class Rectangle implements Shape{
  double length;
  double width;
  public double calculateArea(){
    return length * width;
  }
}

public class Circle implements Shape{
  public double radius;
  public double calculateArea(){
    return (22/7)*radius*radius;
  }
}
```

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 리스코프 치환 원칙(Liskov Substitution Principle)
  - 자식 클래스는 언제나 자신의 부모 클래스를 **대체**할 수 있다는 원칙이다. 즉 부모 클래스가 들어갈 자리에 자식 클래스를 넣어도 계획대로 잘 작동해야 한다는 것.
  - 상속의 본질인데, 이를 지키지 않으면 부모 클래스 본래의 의미가 변해서 is a 관계가 망가져 다형성을 지킬 수 없게 된다.
  - 상속은 구현상속(extends 관계)이든 인터페이스 상속(implements 관계)이든 궁극적으로는 다형성을 통한 확장성 획득을 목표로 함.
  - LSP원리도 역시 서브 클래스가 확장에 대한 인터페이스를 준수해야 함을 의미한다.

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 리스코프 치환 원칙(Liskov Substitution Principle)

```java
public class Animal {
    public void makeNoise() {
        System.out.println("I am making noise");
    }
}
// (O)
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("bow wow");
    }
}
// (O)
public class Cat extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("meow meow");
    }
}
// (X)
class DumbDog extends Animal {
    @Override
    public void makeNoise() {
        throw new RuntimeException("I can't make noise");
    }
}
```

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 인터페이스 분리 원칙(Interface Segregation Principle)
  - 한 클래스는 자신이 사용하지 않는 인터페이스는 구현하지 말아야 한다는 원리
  - 하나의 일반적인 인터페이스보다는 구체적인 여러 개의 인터페이스가 낫다
  - SRP가 클래스의 단일책임을 강조한다면 ISP는 인터페이스의 단일책임을 강조한다.
  - 하지만 ISP는 어떤 클래스 혹은 인터페이스가 여러 책임 혹은 역할을 갖는 것을 인정한다.
  - 이러한 경우 ISP가 사용되는데 SRP가 클래스 분리를 통해 변화에의 적응성을 획득하는 반면, ISP에서는 인터페이스 분리를 통해 같은 목표에 도달 할 수 있다.

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 인터페이스 분리 원칙(Interface Segregation Principle)

```
/*
    인터페이스를 사용하려고 하지만 보고서를 PDF 형식으로만 사용하고 Excel은 사용하지 않으려는 경우를 고려해보자.
    위의 설계로는 두 가지 방법을 모두 실행하도록 강요하고 있기 때문에 달성할 수 없다.
    그러므로 이 설계는 인터페이스 분리 원칙을 따르지 않는다.
*/
public interface GenerateReport{
    public void generateExcel();
    public void generatePDF();
}
/* 해결책 */
public interface GenerateReportByExcel
    public void generateExcel();
}

public interface GenerateReportByPDF
    public void generatePDF();
}
```

# 객체지향 프로그래밍 5대 원칙 (SOLID)

- 의존성 역전 원칙(Dependency Inversion Principle)
  - 추상성이 높고 안정적인 고수준의 클래스는 구체적이고 불안정한 저수준의 클래스에 의존해서는 안된다는 원칙.
  - 일반적으로 객체지향의 인터페이스를 통해서 이 원칙을 준수할 수 있게 된다.
  - (상대적으로 고수준인) 클라이언트는 저수준의 클래스에서 추상화한 인터페이스만을 바라보기 때문에, 이 인터페이스를 구현한 클래스는 클라이언트에 어떤 변경도 없이 얼마든지 나중에 교체될 수 있다. (전략 패턴을 떠올리면 된다)

# 객체지향 프로그래밍 5대 원칙 (SOLID)

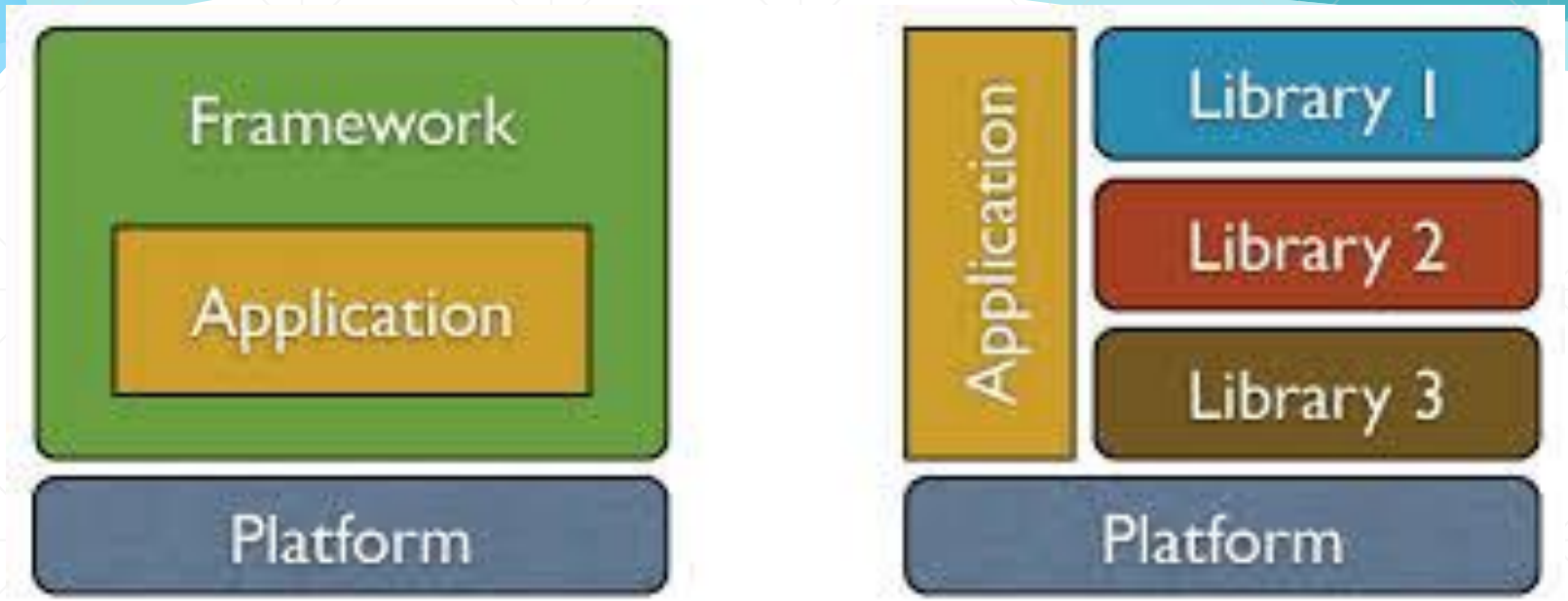
- 의존성 역전 원칙(Dependency Inversion Principle)

```
class Worker {
    public void work() {
        // ....working
    }
}

class Manager {
    Worker worker;
    public void setWorker(Worker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}

class SuperWorker {
    public void work() {
        //.... working much more
    }
}
```

```
interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}

class SuperWorker  implements IWorker{
    public void work() {
        //.... working much more
    }
}

class Manager {
    IWorker worker;

    public void setWorker(IWorker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```

# Framework

Framework란?




# Framework

Framework == Library?

# Framework

- Java Framework

EJB

Enterprise Java Bean

Struts 2

# Framework

- EJB란?
  - 애플리케이션의 업무 로직을 가지고 있는 서버 애플리케이션
  - EJB 사양은 Java EE의 자바 API 중 하나
  - EJB는 '구조가 복잡한 대규모 분산 객체 환경'을 쉽게 구현하기 위해서 등장한 기술
  - 따라서 EJB를 사용하기 전에, 프로젝트가 분산 객체 환경을 필요로 하는지에 대한 고려가 반드시 필요하며 분산 객체 환경이 필요 없는 경우에는 EJB를 사용해야할 필요가 없다고 해도 과언이 아니다.
  - 하지만 EJB를 사용하면 시스템의 무조건적인 향상 있을 것이라는 잘못된 지식 때문에 EJB가 등장한 초기에는 EJB 프로젝트를 무조건 사용하는 경우도 많았다.

# Framework

- EJB 장점
  - 동시접속자수가 많은 가운데 안정적인 트랜잭션이 필요한 사이트 구축 시 용이
- EJB 단점
  - 복잡한 프로그래밍 모델
  - 특정 환경에 쉽게 종속적인 코드
  - 필요없이 특정 기술에 종속적인 코드
  - 컨테이너에 안에서만 동작할 수 있는 객체구조
  - 자동화된 테스트가 매우 어렵거나 불가능
  - 객체지향적이지 않음
  - 형편없는 개발생산성
  - 한심한 이동성(portablity)

# Framework

- Struts
  - Java EE 웹 어플리케이션을 개발하기 위한 오픈소스 프레임워크
  - MVC 아키텍처를 적용하는 개발자들을 지원하기 위하여 자바 서블릿 API 를 사용하고 확장.
  - 크레이그 맥클라나한(Craig McClanahan) 에 의해 최초로 만들어졌으며 아파치 재단에 2000년 5월 이관되었다.
  - 페이지 디자이너, 컴포넌트 개발자, 프로젝트 일부를 담당하는 다른 개발자 등 성격이 다른 그룹들에 의해 다루어지는 대형 웹 어플리케이션의 설계와 구현을 가능케 함
  - I18N (국제화), 강력한 커스텀 태그 라이브러리, 타일형 디스플레이, 폼 유효성 검사등의 특징을 가진다.
  - 또한 다양한 프레젠테이션 레이어들을 지원하여, JSP, XML/XSLT, JSF, 벨로시티 등을 포함하고 있으며 또한 JavaBeans와 EJB 등 다양한 모델 레이어를 포함하고 있다.

# Framework

- Struts 장점
  - 가볍다
  - POJO기반의 액션(Action)
  - 많은 설정 생략 가능, 어노테이션 사용으로 설정 파일에서 설정 생략(과거형)
  - 변경된 환경 설정 파일의 내용을 웹 컨테이너 재시작 없이 리로드

# Framework

- Struts 단점
  - 그럼에도 불구하고 너무나 많은 설정
  - web applicatio에만 사용할 수 있는 국한성
  - JSP에 종속적임
  - 큰 어플리케이션 개발 시 소스의 난잡화

# Spring

# Spring

- Spring?
  - 자바(JAVA) 플랫폼을 위한 오픈소스(Open Source) 어플리케이션 프레임워크
  - 자바 개발을 편하게 해주는 경량급 애플리케이션 프레임워크
- Spring을 쓰게 된 배경
  - 복잡함에 반기가 들어서 만들어진 프레임워크
  - 프로젝트의 전체 구조를 설계할 때 유용
  - 다른 프레임워크의 포용

# Spring

- Spring 특징
  - 경량 컨테이너로서 자바 객체를 직접 관리
  - POJO(Plain Old Java Object)
  - IoC(Inversion of Control)
  - DI(Dependency Injection)
  - AOP(Aspect Oriented Programming)
  - PSA(Portable Service Abstraction)
  - Application Object의 생명 주기와 설정을 관리

# Spring

# Spring

- POJO (Plain Old Java Object)
  - 오래된 방식의 간단한 자바 오브젝트
  - Java EE 등의 중량 프레임워크들을 사용하게 되면서 해당 프레임워크에 종속된 "무거운" 객체를 만들게 된 것에 반발해서 사용되게 된 용어

# Spring

- IoC (Inversion of Control) - 제어의 역전
  - 제어 반전, 제어의 반전, 역제어는 프로그래머가 작성한 프로그램이 재사용 라이브러리의 흐름 제어를 받게 되는 소프트웨어 디자인 패턴을 말한다.
  - 전통적인 프로그래밍에서 흐름은 프로그래머가 작성한 프로그램이 외부 라이브러리의 코드를 호출해 이용한다. 하지만 제어 반전이 적용된 구조에서는 외부 라이브러리의 코드가 프로그래머가 작성한 코드를 호출한다.
  - 설계 목적상 제어 반전의 목적
    - 작업을 구현하는 방식과 작업 수행 자체를 분리한다.
    - 모듈을 제작할 때, 모듈과 외부 프로그램의 결합에 대해 고민할 필요 없이 모듈의 목적에 집중할 수 있다.
    - 다른 시스템이 어떻게 동작할지에 대해 고민할 필요 없이, 미리 정해진 협약대로만 동작하게 하면 된다.
    - 모듈을 바꾸어도 다른 시스템에 부작용을 일으키지 않는다.

# Spring

- Bean 객체
  - IoC 컨테이너에서 직접 관리하고 있는 객체
  - Bean 객체는 싱글턴 형태로 IoC 의 빈 풀(Bean Pool)에서 관리가 되어진다.
  - 해당 Bean 객체는 등록하는 방식이 몇가지가 있다.
    - component-scan을 통해 등록하는 방식
    - ApplicaionContext를 통해 직접 등록하는 방식

# Spring

- DI(Dependency Injection) – 의존성 주입
  - 의존성 주입(Dependency Injection, DI)은 프로그래밍에서 구성 요소 간의 의존 관계가 소스코드 내부가 아닌 외부의 설정파일 등을 통해 정의되게 하는 디자인 패턴 중의 하나이다.
  - 의존성 주입의 이점
    - 의존 관계 설정이 컴파일시가 아닌 실행시에 이루어져 모듈들 간의 결합도를 낮출 수 있다.
    - 코드 재사용을 높여서 작성된 모듈을 여러 곳에서 소스코드의 수정 없이 사용할 수 있다.
    - 모의 객체 등을 이용한 단위 테스트의 편의성을 높여준다.

# Spring

- AOP(aspect-oriented programming) – 관점 지향 프로그래밍
  - 횡단 관심사(cross-cutting concern)의 분리를 허용함으로써 모듈성을 증가시키는 것이 목적인 프로그래밍 패러다임
  - 코드 그 자체를 수정하지 않는 대신 기존의 코드에 추가 동작(어드바이스)을 추가함으로써 수행
  - "함수의 이름이 'set'으로 시작하면 모든 함수 호출을 기록한다"와 같이 어느 코드가 포인트 컷(pointcut) 사양을 통해 수정되는지를 따로 지정한다.
  - 이를 통해 기능의 코드 핵심부를 어수선하게 채우지 않고도 비즈니스 로직에 핵심적이지 않은 동작들을 프로그램에 추가할 수 있게 한다.
  - 관점 지향 프로그래밍은 관점 지향 소프트웨어 개발의 토대를 형성한다.

# Spring

- PSA(Portable Service Abstraction) – 이식 가능한 서비스 추상화
  - 환경과 세부 기술의 변화에 관계없이 일관된 방식으로 기술에 접근할 수 있게 함
  - POJO로 개발된 코드는 특정 환경이나 구현방식에 종속적이지 않아야 한다
  - 다시 말해, Spring은 POJO 원칙으로 만들었기 때문에, Spring 패키지 외의 것들을 POJO화 시키기 위해 껍데기를 씌우겠다는 것
  - 각 벤더 들이 여러가지 인터페이스로 제공을 하더라도, Spring 에서 Adapter pattern 을 적용하여 제공하므로, 사용하는 클라이언트에서는 공통된 메소드를 호출하는 형태로 구현하면 됨.
  - Junit, MyBatis, JDBC, Data, Social, etc

# 개인적으로 생각하는 특징

- Java파일에서 Java 코드를 줄일 수 있다.
- 반복되는 작업을 줄일 수 있어 기능 개발에 집중할 수 있다.
- 프로젝트 관리가 용이하다.
- 다수의 개발자와 동시에 프로젝트 하기가 용이하다.
- 처음 프로젝트 셋팅이 다소 복잡하다.
- 개념을 제대로 숙지하지 못하면 코드 분석 조차 하기 힘들다.

# 강좌의 범위

- 본 강좌는 Spring Framework 버전 5를 중심으로 수업을 진행한다.
- 모든 예제는 Spring Framework 버전 4에서 테스트가 완료된 상태이다.
- 버전 4, 버전 5간의 차이가 있는 부분은 비교 예제를 통해 살펴본다.

# 학습정리

- Spring Framework는 Java 애플리케이션을 보다 쉽고 빠르게 개발할 수 있는 Framework이다.
- J2EE 기반 웹 애플리케이션 개발에 널리 쓰여지고 있다.