

chool

*Spring AOP*



段維瀚 老師





# *Session*

- Spring AOP（方面導向）
- Cross-cutting concern
- AOP 觀念與術語
- PointCut EL
- AOP 四大攔截器
- AOP Introduction





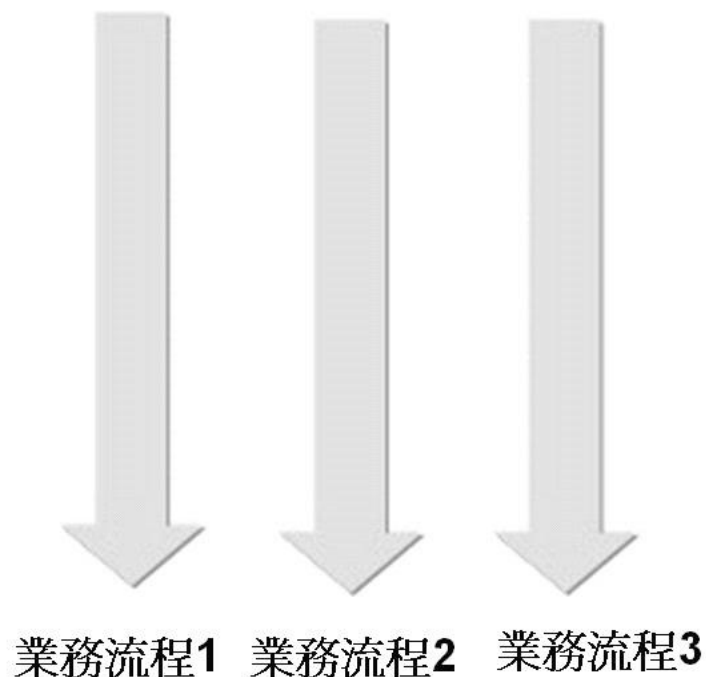
# *Spring AOP(方面導向)*





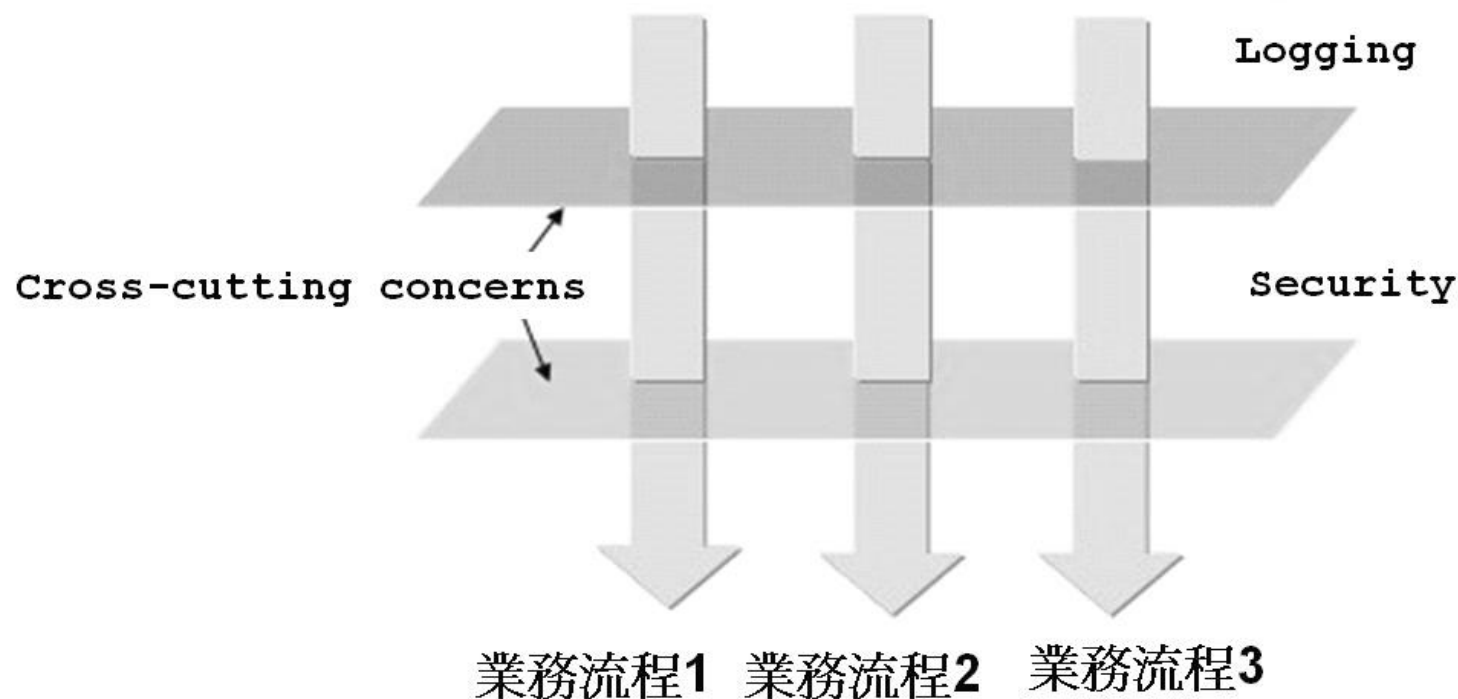
# 物件導向系統流程

- 物件導向-垂直架構



# Cross-cutting concern

- 橫切關注點（AOP-水平架構）





# Cross-cutting concern

- 橫切關注點

- 若直接將公用邏輯（例如：日誌）撰寫在商務流程中，會使得維護程式的成本增高
  - 若今天要將物件中的日誌功能修改或是移除，則必須修改所有有日誌服務的商務程式碼，然後重新編譯，另一方面，這種日誌功能的撰寫設計，  
「**混雜**」於商務邏輯之中，使得商務物件本身的邏輯或程式的撰寫更為複雜。





# 切面（方面）導向設計 - AOP

- 《Clean Code》書中說：「將所有關注的事（Concerns）分離開來，是軟體技巧中，最古老、最重要的設計技巧之一。」
  - 有些關注與程式主流一致，易識別與分離為獨立的程式庫或框架，
  - 有些關注則是對主要流程橫切的關注（Cross-cutting concerns），易破碎地出現在各主要流程中
  - 面對這些，設計成可獨立、重用的切面（Aspect）模組為目標，就是切面導向程式設計（Aspect-oriented programming, AOP）。



# 攔截機制 *Filter & AOP*

- **Filter**

- 針對路徑過濾

- **AOP**

- 針對業務方法進行橫切攔截

- Spring AOP 僅支援方法代理（不支援建構子）







# AOP / AspectJ

- AOP (Aspect-Oriented Programming)
  - AOP 是一種程式開發方式
- AspectJ
  - 是一種將 AOP 的開發方式引入 Java 程式語言的專案
    - 目前由 PARC 組織建立與維護，並得到廣大的使用。





# Spring 中 AOP 的支援形式有

- Classic Spring proxy-based AOP
  - 早期 Spring 以代理人機制設計模式為基礎的開發。
- Pure-POJO aspects
  - 透過 XML 設定檔，將單存的 POJO 物件轉換為 aspect 物件
- @AspectJ annotation-driven aspects
  - 藉由 AspectJ 來啟用以 annotation 為基礎的 AOP 程式
- Injected AspectJ aspects
  - 將切入點的規則注入/套用到目標物件。





# AOP 觀念與術語

- Aspect（方面）
  - 將散落於各個商務物件之中的Cross-cutting concerns收集起來，設計成各個獨立可重用的物件，這些物件稱之為Aspect。
- Advice（通知）
  - Aspect的具體實作稱之為Advice。
    - 其中包括了Cross-cutting concerns的行為或所要提供的服務。
    - 前置通知、後置通知、例外通知、環繞通知等...





# AOP 觀念與術語

- Joinpoint（連接點）
  - Aspect在應用程式執行時加入商務流程的點或時機稱之為Joinpoint，具體來說，就是Advice在應用程式中被呼叫執行的時機，這個時機可能是某個方法被呼叫之前或之後（或兩者都有），或是某個例外發生的時候。





# AOP 觀念與術語

- **Pointcut**（切入點）
  - Pointcut是一個定義，藉由這個定義您可以指定某個Aspect在哪些Joinpoint時被應用至應用程式之上。
  - 具體的說，您可以在某個定義檔中撰寫Pointcut，當中說明了哪些Aspect要應用至應用程式中的哪些Joinpoint。



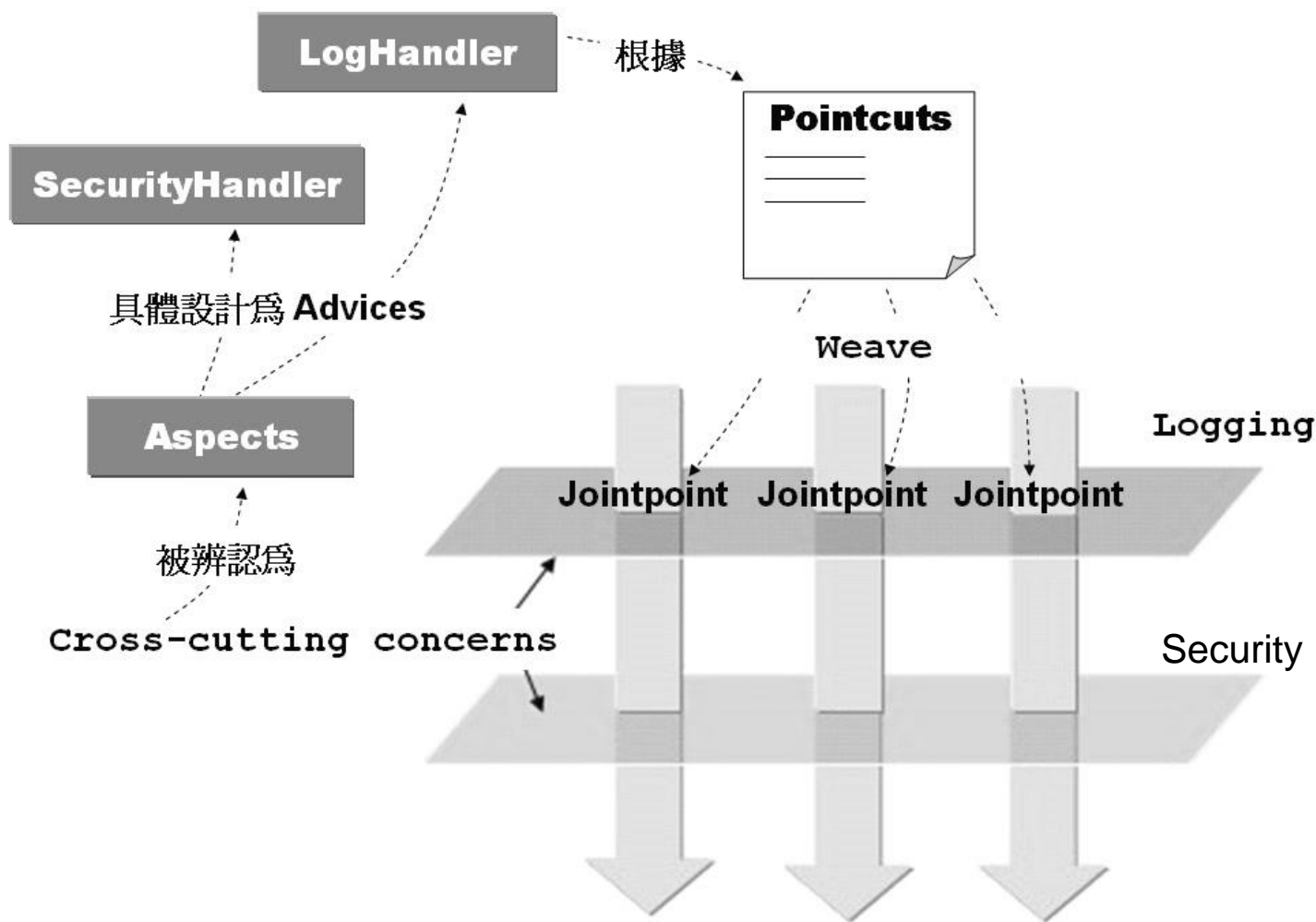


# AOP 觀念與術語

- Target（目標對象）
  - 一個Advice被應用的對象或目標物件
- Proxy（代理）
  - Spring的AOP主要是透過動態代理來完成。
- Weave（縫合）
  - Advice被應用至物件之上的過程稱之為縫合（Weave）。



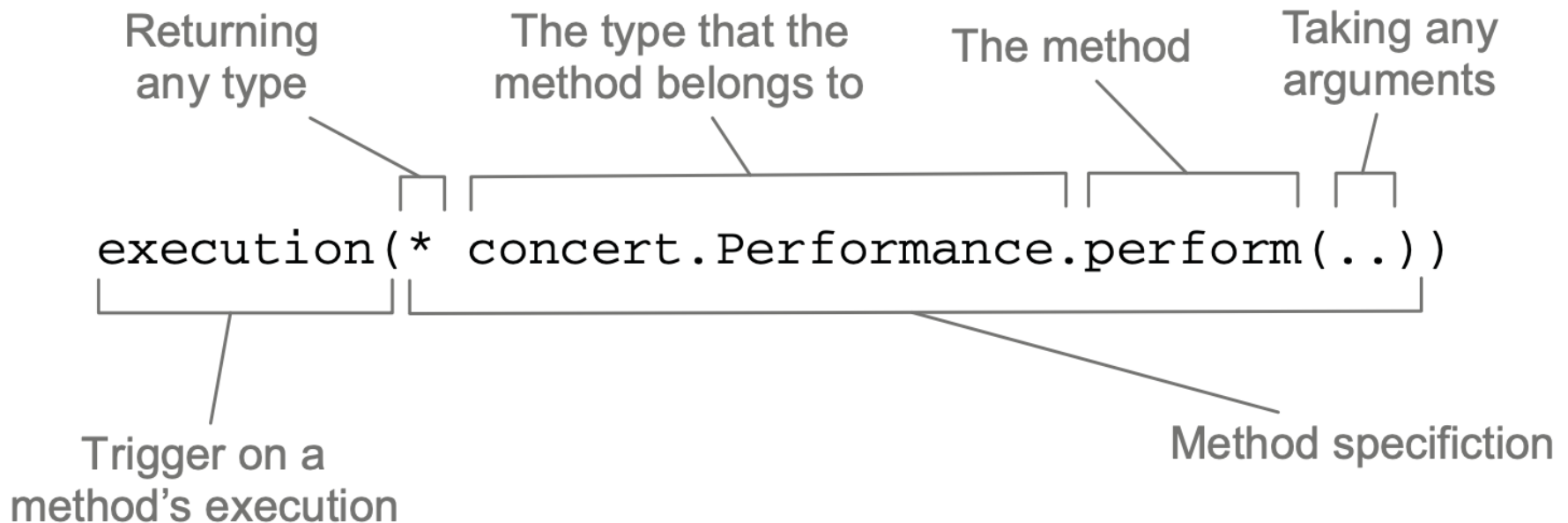
# AOP 觀念與術語



# PointCut EL

- EL 語法設定樣式

designator (modifiers return-type declaring-type method-name (params) exception)





# PointCut EL

- EL 語法設定樣式

designator (modifiers return-type declaring-type method-name (params) exception)

The execution of the Instrument.play() method

```
execution(* concert.Performance.perform(..))  
  && within(concert.*))
```

Combination (and)  
operator

When the method is called from within any  
class in the concert package

designator (modifiers return-type declaring-type method-name (params) exception)

- **designator(必要)**
  - **execution()**
- **modifiers(非必要)**
  - 存取修飾字
- **return-type(必要)**
  - 回傳型態，可以使用「\*」來表示任意。
- **declaring-type(非必要)**
  - 套件與類別名稱
- **method-name(必要)**
  - 方法名稱，可以使用「\*」來表示任意。
- **params(必要)**
  - 方法參數。() 無參數、(..)任意參數、(\*, String)任意+String參數
- **exception(非必要)**
  - 拋出例外的型態





# Spring-AOP 四大攔截器

- AOP 四大攔截器

- Before Advice

- 在方法執行前被呼叫

- After Advice

- 在方法執行後被呼叫

- Around Advice

- 要在方法呼叫前後加入Advices的服務邏輯

- Throws Advice

- 想要在例外發生時通知某些服務物件作某些事





# *Spring-AOP*

- AOP

- Before Advice 前置通知

- it is executed before the actual method call.





# *Spring-AOP*

- AOP

- After Advice 後置通知

- it is executed after the actual method call. If method returns a value, it is executed after returning value.





# Spring-AOP

- AOP

- Around Advice 環繞通知








- it is executed before and after the actual method call.

- import  
org.aopalliance.intercept.MethodInterceptor;



# AOP Lab

## • 範例程式：

 AOPConfig.java	Java 配置檔
 Audience.java	Aspect (觀眾)
 BackSinger.java	幕後歌者
 Dancer.java	前台舞者
 Introducter.java	經理人
 Performance.java	表演介面
 Singer.java	歌者介面



# AOP Lab Test

```
// Java 配置使用 AnnotationConfigApplicationContext
ApplicationContext ctx = new AnnotationConfigApplicationContext(AOPConfig.class);
Performance dancer = ctx.getBean("dancer", Performance.class);

try {
    dancer.perform(); // 執行表演
} catch (Exception e) {
    System.out.println(e); // 例外發生與處理

    // 轉換跑道 (AOP Introduction 機制)
    // dancer 透過 introducer(經紀人) 轉換跑道
    // 例如：改行去唱歌
    System.out.println("舞者轉歌者");
    Singer singer = (Singer)dancer; // 舞者轉歌者
    singer.sing();
}
```





# AOP 觀念與術語

- Introduction

- 對於一個現存的類別，Introduction可以為其增加行為，而不用修改該類別的程式，具體的說，您可以為某個已撰寫、編譯完成的類別，在執行時期動態加入一些方法或行為，而不用修改或新增任何一行程式碼。





# Introduction

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class Introducter { // 介紹人/經紀人

    // 將 Performance 及其子類別轉換為 Singer (預設實作: BackSinger)
    @DeclareParents(value = "com.spring.core.session05.aop_lab.Performance+",
                    defaultImpl = BackSinger.class)
    public Singer singer;

}
```