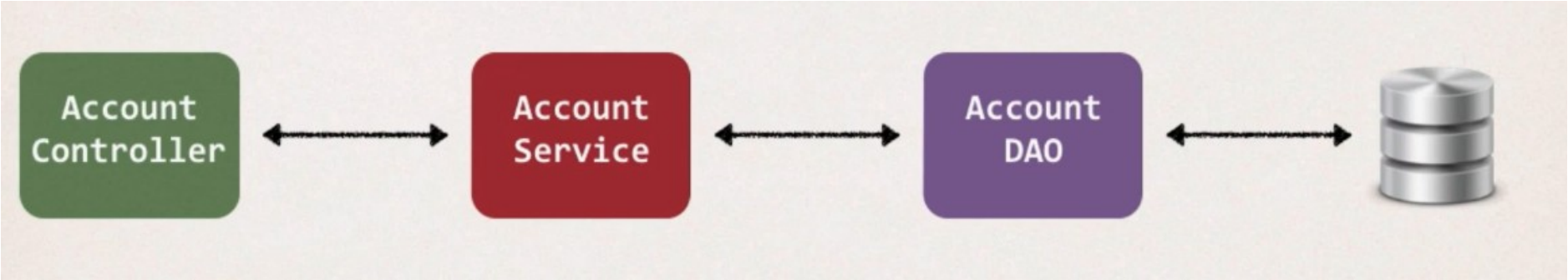
Spring AOP

Prof. Siddharth Shah
Department of Computer Engineering
Dharmsinh Desai University

Outline

- High Level View of the task
- Application Flow
- Edit configuration files (web.xml & dispatcher-servlet.xml)
- Create FormController
- Create Views

Application Architecture



New Requirement - Logging

- Need to add logging to our DAO methods
  - Add some logging statements before the start of the method
- Possibly more places … but get started on that ASAP!

DAO – Add Logging Code

```
public void addAccount(Account theAccount, String userId) {

    // add code for logging


    Session currentSession = sessionFactory.getCurrentSession();

    currentSession.save(theAccount);


}
```

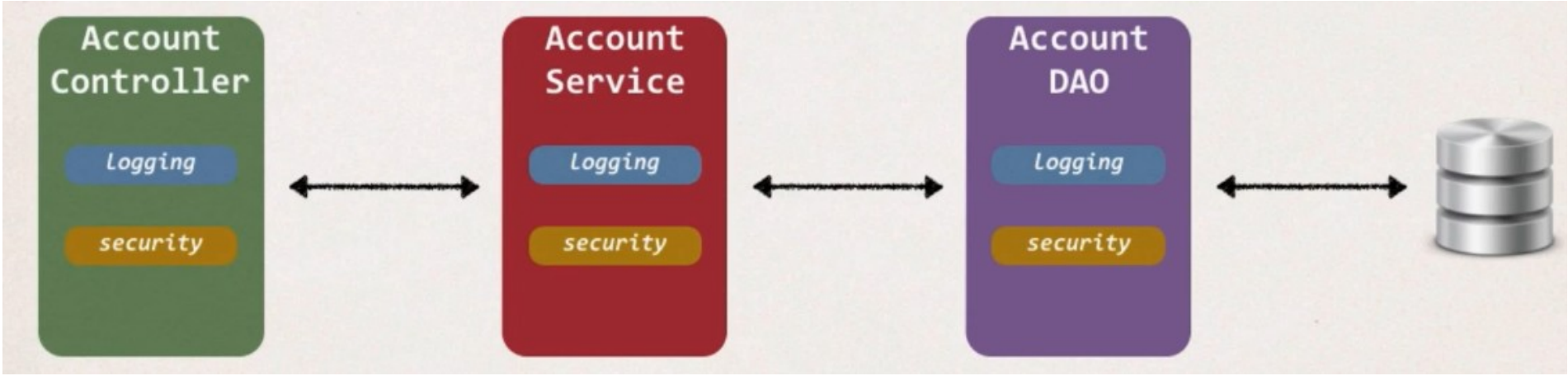
New Requirement - Security

- Need to add security code to our DAO
  - Make sure user is authorized before running DAO method

Add Security Code

```
public void addAccount(Account theAccount, String userId) {

    // add code for logging

    // add code for security check

    Session currentSession = sessionFactory.getCurrentSession();

    currentSession.save(theAccount);

}
```

Let’s add it to all layers....



Two Main Problems

- **Code Tangling**
  - For a given method: addAccount(…)
  - We have logging and security code tangled in

- **Code Scattering**
  - If we need to change logging or security code
  - We have to update ALL classes

Other Possible Solutions

- **Inheritance?**
  - Every class would need to inherit from a base class
  - Can all classes extends from your base class? … plus no multiple inheritance
- **Delegation?**
  - Classes would delegate logging, security calls
  - Still would need to update classes if we wanted to
    - add/remove logging or security
    - add new feature like auditing, API management, instrumentation

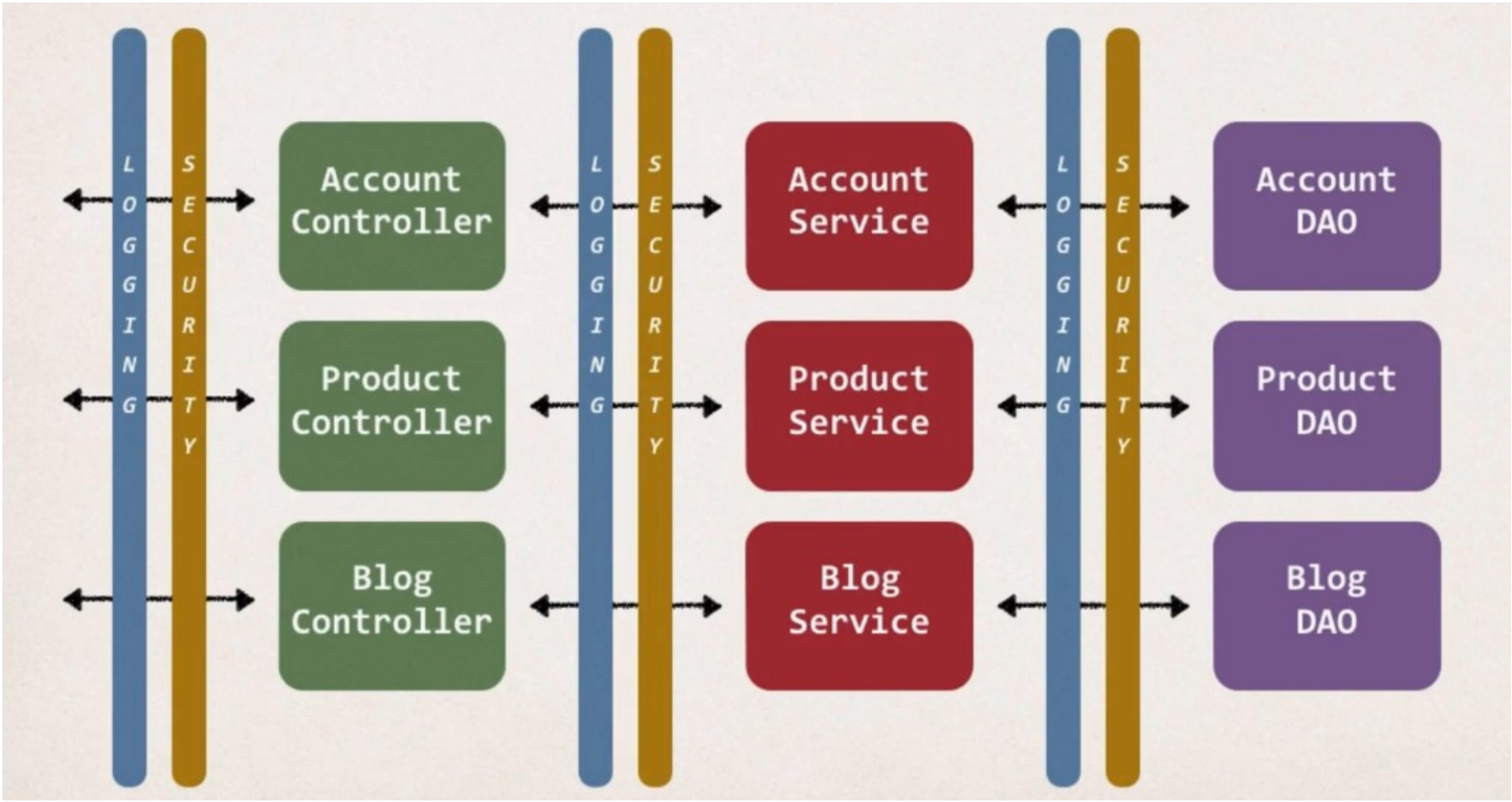
Aspect Oriented Programming

- Programming technique based on concept of an Aspect
- Aspect encapsulates cross-cutting logic

Cross-Cutting Concerns

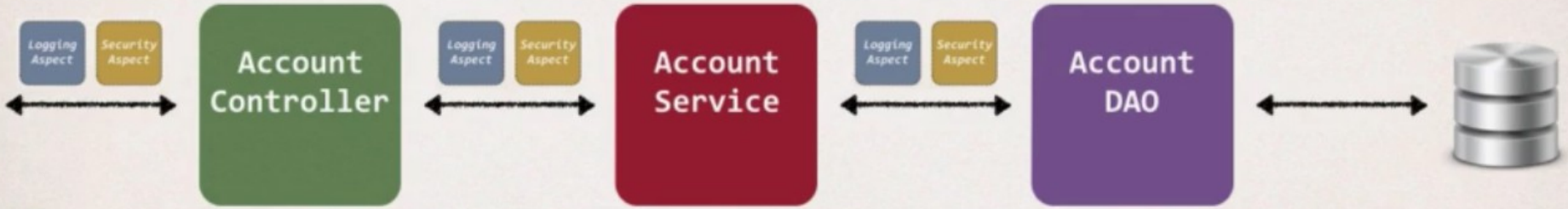
- "Concern" means logic / functionality

Cross-Cutting Concerns



Aspects

- Aspect can be reused at multiple locations
- Same aspect/class … applied based on configuration



AOP Solution

- Apply the Proxy design pattern


```
MainApp

// call target object
targetObj.doSomeStuff();
```

```
TargetObject

public void doSomeStuff() {
 …
}
```

Benefits of AOP

- **Code for Aspect is defined in a single class**
  - Much better than being scattered everywhere
  - Promotes code reuse and easier to change
- **Business code in your application is cleaner**
  - Only applies to business functionality: addAccount
  - Reduces code complexity
- **Configurable**
  - Based on configuration, apply Aspects selectively to different parts of app
  - No need to make changes to main application code … very important!

AOP Use Cases

- **Most common**
  - logging, security, transactions
- **Audit logging**
  - who, what, when, where
- **Exception handling**
  - log exception and notify DevOps team via SMS/email
- **API Management**
  - how many times has a method been called user
  - analytics: what are peak times? what is average load? who is top user?

AOP: Advantages & Disadvantages

Advantages

- Reusable modules
- Resolve code tangling
- Resolve code scatter
- Applied selectively based on configuration

Disadvantages

- Too many aspects and app flow is hard to follow
- Minor performance cost for aspect execution (run-time weaving)

AOP Terminology

- **Aspect**: module of code for a cross-cutting concern (logging, security, …)
- **Advice**: What action is taken and when it should be applied
- **Join Point**: When to apply code during program execution
- **Pointcut**: A predicate expression for where advice should be applied

Advice Types

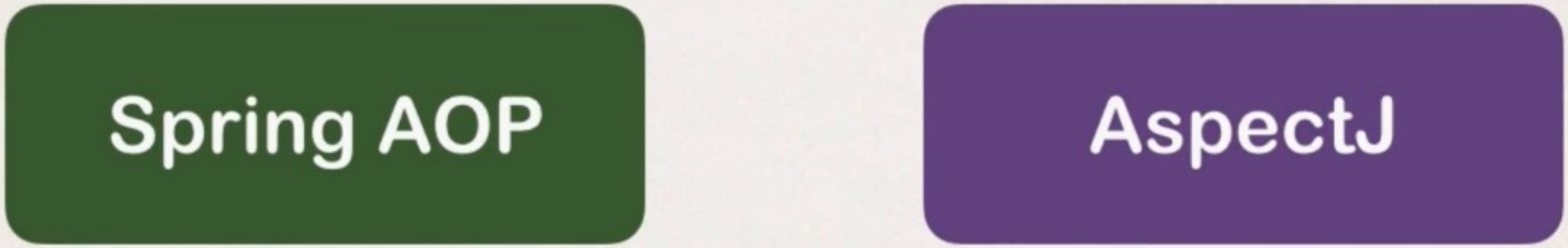
- **Before advice**: run before the method
- **After finally advice**: run after the method (finally)
- **After returning advice**: run after the method (success execution)
- **After throwing advice**: run after method (if exception thrown)
- **Around advice**: run before and after method

Weaving

- Connecting aspects to target objects to create an advised object
- Different types of weaving
  - Compile-time, load-time or run-time
- Regarding performance: run-time weaving is the slowest

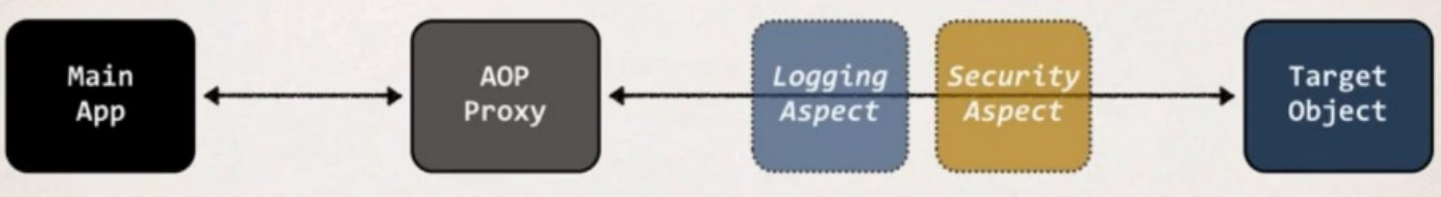
AOP Frameworks

- Two leading AOP Frameworks for Java



Spring AOP

- Spring provides AOP support
- Key component of Spring
  - Security, transactions, caching etc
- Uses run-time weaving of aspects



AspectJ

- Original AOP framework, released in 2001
- Provides complete support for AOP
- Rich support for
  - join points: method-level, constructor, field
  - code weaving: compile-time, post compile-time and load-time

Spring AOP Comparison

Advantages

- Simpler to use than AspectJ
- Uses Proxy pattern
- Can migrate to AspectJ when using @Aspect annotation

Disadvantages

- Only supports method-level join points
- Can only apply aspects to beans created by Spring app context
- Minor performance cost for aspect execution (run-time weaving)

AspectJ Comparison

Advantages

- Support all join points
- Works with any POJO, not just beans from app context
- Faster performance compared to Spring AOP
- Complete AOP support

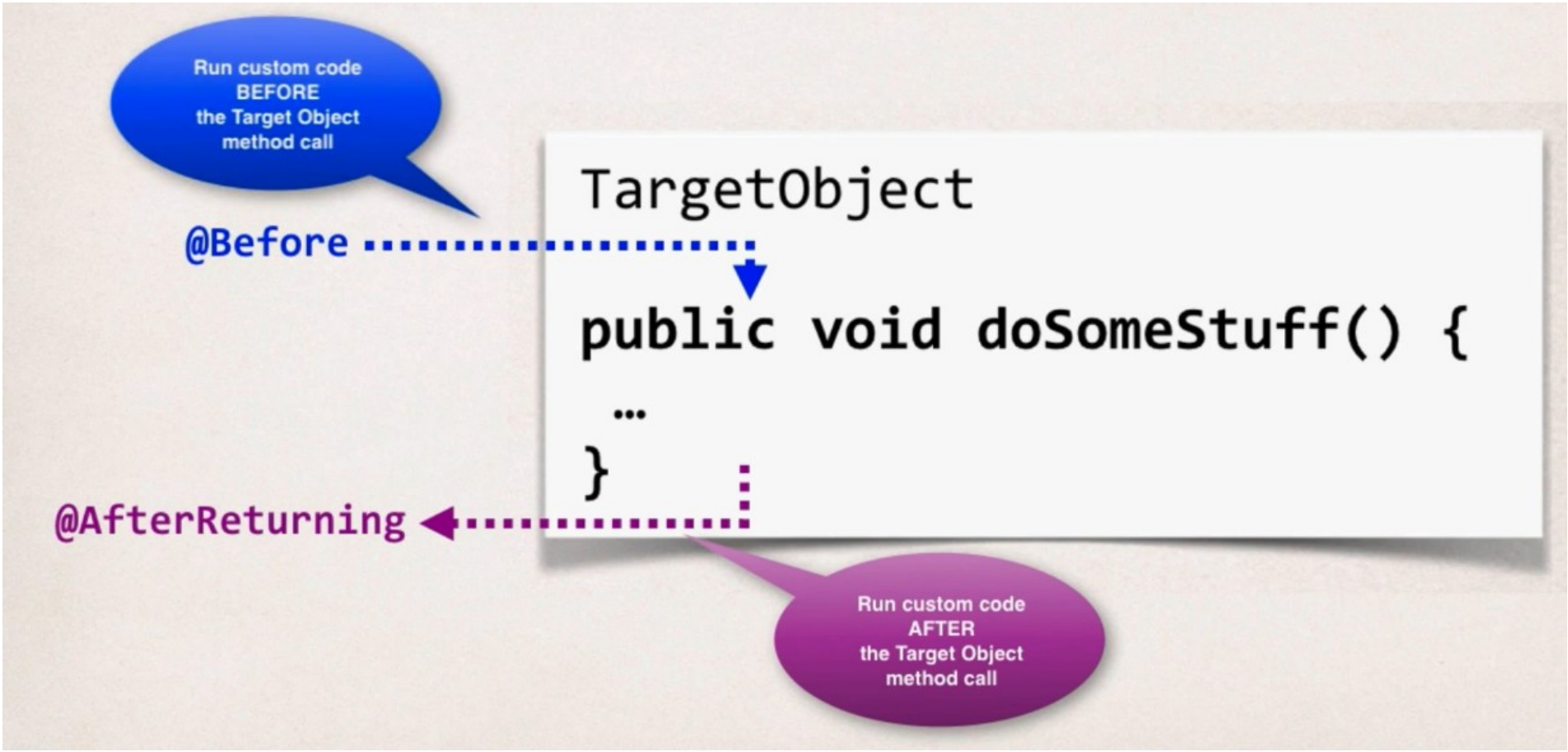
Disadvantages

- Compile-time weaving requires extra compilation step
- AspectJ pointcut syntax can become complex

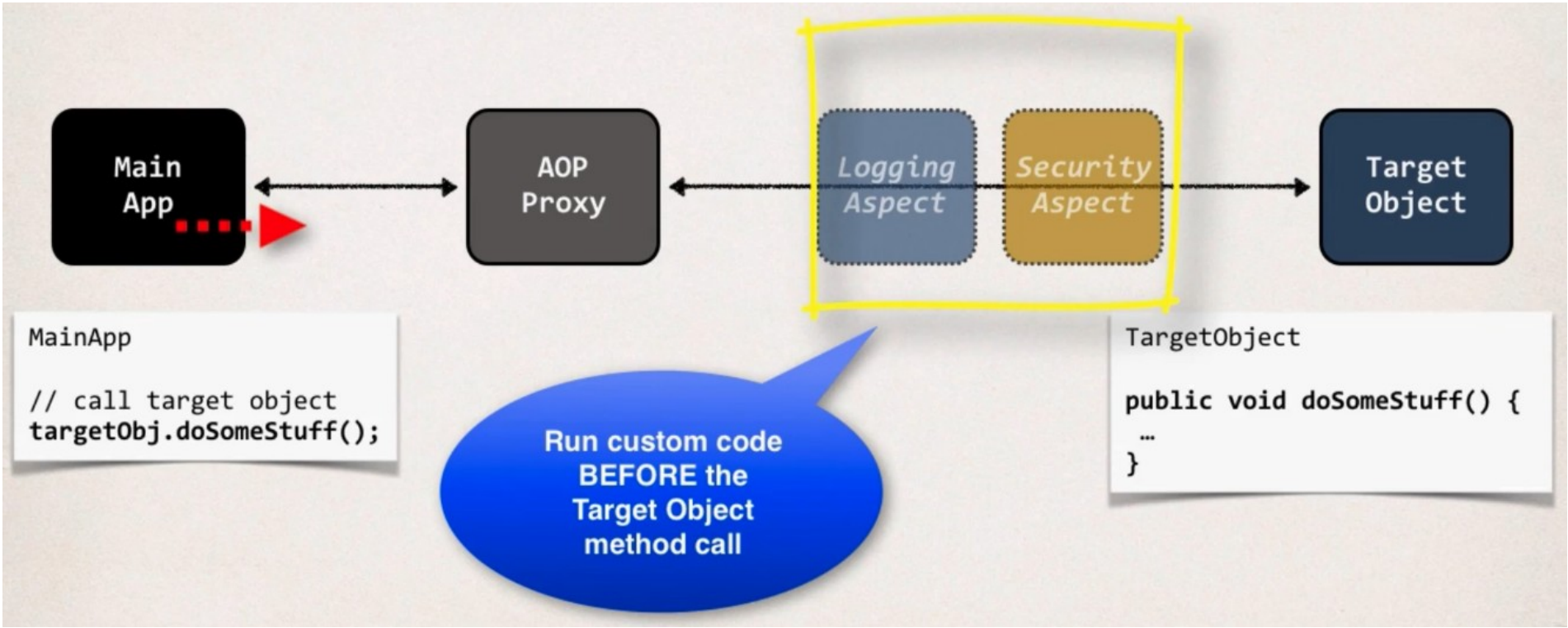
Comparing Spring AOP & AspectJ

- Spring AOP only supports
  - Method-level join points
  - Run-time code weaving (slower than AspectJ)
- AspectJ supports
  - join points: method-level, constructor, field
  - weaving: compile-time, post compile-time and load-time

Advice – Interaction



@Before Advice - Interaction



Development Process - @Before

1. Create target object: AccountDAO
2. Create Spring Java Config class
3. Create main app
4. Create an Aspect with @Before advice