1. Advice Parameters and Generics

Spring AOP는 제네릭을 포함한 메서드와 클래스의 파라미터를 처리할 수 있습니다. 예를 들어, 다음과 같은 제네릭 인터페이스가 있다고 가정해보겠습니다:

```
public interface Sample<T> {
    void sampleGenericMethod(T param);
    void sampleGenericCollectionMethod(Collection<T> param);
}
```

이 인터페이스를 구현하는 클래스에서 메서드에 대한 AOP 어드바이스를 설정할 수 있습니다. 다음은 `sampleGenericMethod` 메서드에 대해 advice를 설정하는 예제입니다:

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
public void beforeSampleMethod(MyType param) {
    // Advice 구현
}
```

[^포인트컷분석] 이 포인트컷 표현식은 `Sample` 인터페이스 및 이 인터페이스를 구현하는 모든 클래스에 적용됩니다. `Sample+`는 `Sample` 인터페이스와 그 서브타입을 모두 포함합니다.

하지만, `sampleGenericCollectionMethod` 메서드와 같은 제네릭 컬렉션 타입의 경우, 아래와 같은 포인트컷 표현식은 사용할 수 없습니다:

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<MyType> param) {
    // Advice 구현
}
```

이는 Spring AOP가 제네릭 컬렉션의 구체적인 타입을 처리하는 데 제약이 있기 때문입니다. 대신, `Collection<?>`으로 지정하고, 파라미터의 타입을 수동으로 확인해야 합니다:

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<?> param) {
    if (!param.isEmpty()) {
        Object firstElement = param.iterator().next();
        // Type check and processing here
    }
}
```

2. Determining Argument Names

Spring AOP에서는 advice 호출 시 파라미터 이름을 매칭하는 방식을 사용합니다. 이 매칭은 포인트컷 표현식에서 지정한 이름과 advice 메서드 서명에 선언된 파라미터 이름을 기준으로 합니다.

다음은 파라미터 이름을 결정하기 위한 `ParameterNameDiscoverer` 구현체입니다:

- **AspectJAnnotationParameterNameDiscoverer**: `argNames` 속성으로 사용자가 명시적으로 지정한 파라미터 이름을 사용합니다.
- **KotlinReflectionParameterNameDiscoverer**: Kotlin 리플렉션 API를 사용하여 파라미터 이름을 결정합니다.
- **StandardReflectionParameterNameDiscoverer**: Java 8+의 `-parameters` 플래그로 컴파일된 코드를 통해 파라미터 이름을 결정합니다.
- **AspectJAdviceParameterNameDiscoverer**: 포인트컷 표현식, 반환, 및 throw 절에서 파라미터 이름을 유추합니다.

Explicit Argument Names

`@AspectJ` advice와 포인트컷 애노테이션에 `argNames` 속성을 사용하여 파라미터 이름을 명시할 수 있습니다:

```
@Before(
    value = "com.xyz.Pointcuts.publicMethod() && target(bean) &&
@annotation(auditable)",
    argNames = "bean,auditable")
public void audit(Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code and bean
}
```

첫 번째 파라미터가 `JoinPoint`, `ProceedingJoinPoint`, 또는 `JoinPoint.StaticPart` 타입일 경우, `argNames` 속성에서 해당 파라미터 이름을 생략할 수 있습니다:

```
@Before(
    value = "com.xyz.Pointcuts.publicMethod() && target(bean) &&
@annotation(auditable)",
    argNames = "bean,auditable")
public void audit(JoinPoint jp, Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code, bean, and jp
}
```

`JoinPoint` 타입의 파라미터가 첫 번째일 경우, `argNames` 속성에서 생략할 수 있습니다:

```
@Before("com.xyz.Pointcuts.publicMethod()")
public void audit(JoinPoint jp) {
    // ... use jp
}
```

3. Proceeding with Arguments

`ProceedingJoinPoint`를 사용할 때, 메서드의 파라미터를 순서대로 `proceed` 메서드에 전달할 수 있습니다:

```
@Around("execution(List<Account> find*(..)) && " +
        "com.xyz.CommonPointcuts.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp,
        String accountHolderNamePattern) throws Throwable {
    String newPattern = preProcess(accountHolderNamePattern);
    return pjp.proceed(new Object[] {newPattern});
}
```

여기서 `proceed` 메서드에 전달할 인자는 원래 메서드의 파라미터와 동일한 순서로 배열에 담아 전달해야 합니다.

4. Advice Ordering

여러 어드바이스가 동일한 조인 포인트에서 실행될 경우, Spring AOP는 AspectJ와 동일한 우선순위 규칙을 따릅니다:

- **Before advice**: 우선순위가 가장 높은 advice가 먼저 실행됩니다.
- **After advice**: 우선순위가 가장 높은 advice가 마지막에 실행됩니다.

서로 다른 애스펙트에서 정의된 advice는 별도로 우선순위를 지정할 수 있으며, `Ordered` 인터페이스를 구현하거나 `@Order` 애노테이션을 사용할 수 있습니다:

```
@Aspect
@Order(1)
public class FirstAspect {
    @Before("somePointcut()")
    public void advice() {
        // ... first advice
    }
}

@Aspect
@Order(2)
public class SecondAspect {
    @Before("somePointcut()")
    public void advice() {
        // ... second advice
    }
}
```

Spring Framework 5.2.7부터 동일한 @Aspect 클래스 내에서는 다음과 같은 순서로 advice가 실행됩니다:

- **@Around**
- **@Before**

- **@After**
- **@AfterReturning**
- **@AfterThrowing**

`@After` advice는 `@AfterReturning`이나 `@AfterThrowing` 이후에 호출됩니다.

동일한 @Aspect 클래스에 두 개의 동일한 유형의 advice가 정의된 경우, 순서는 정의되지 않습니다. 이러한 경우, advice를 하나로 통합하거나 별도의 @Aspect 클래스로 리팩터링하여 `Ordered`나 `@Order`를 통해 우선순위를 지정하는 것이 좋습니다.

5. Introductions

Introductions 또는 **inter-type declarations**는 특정 aspect가 advised된 객체가 새로운 인터페이스를 구현하도록 선언하고, 그 인터페이스의 구현을 제공하는 기능입니다.

`@DeclareParents` 애노테이션을 사용하여 Introduction을 생성할 수 있습니다:

```
@Aspect
public class UsageTracking {

    @DeclareParents(value="com.xyz.service.*+",
defaultImpl=DefaultUsageTracked.class)
    public static UsageTracked mixin;

    @Before("execution(* com.xyz..service.*.*(..)) && this(usageTracked)")
    public void recordUsage(UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
    }

}
```

이 aspect는 `com.xyz.service` 패키지의 모든 빈이 `UsageTracked` 인터페이스를 구현하도록 합니다. `DefaultUsageTracked` 클래스는 `UsageTracked` 인터페이스의 기본 구현을 제공합니다. 서비스 빈을 `UsageTracked` 타입으로 사용할 수 있습니다:

```
UsageTracked usageTracked = context.getBean("myService", UsageTracked.class);
```

이렇게 하면 모든 서비스 객체가 `UsageTracked` 인터페이스의 메서드를 사용할 수 있습니다.

[^포인트컷분석]:`com.intheeast.aspectj.declaringadvice.service.Sample+`라는 표현은 AspectJ에서 사용되는 타입 패턴입니다. 여기서 `+` 기호는 특정 타입뿐만 아니라 그 타입의 모든 서브타입(즉, 해당 타입을 상속하거나 구현한 모든 클래스들)을 포함시킨다는 의미입니다.
예를 들어, `Sample`이라는 인터페이스가 존재하고, `SampleService`라는 클래스가 이 인터페이스를 구현하고 있다고 가정해 보겠습니다. 이때 `Sample+` 타입 패턴은 `Sample` 인터페이스뿐만 아니라 `SampleService` 클래스와 같이 `Sample` 인터페이스를 구현하는 모든 클래스에도 동일하게 적용됩니다.
따라서, 이 타입 패턴을 사용하면 `Sample` 인터페이스와 그 하위 타입들에 속한 모든 메서드에 특정한 어드바이

스(Advice)를 적용할 수 있습니다. 이는 AspectJ에서 다양한 클래스 계층에 일관된 방식으로 어드바이스를 적용할 때 매우 유용합니다.