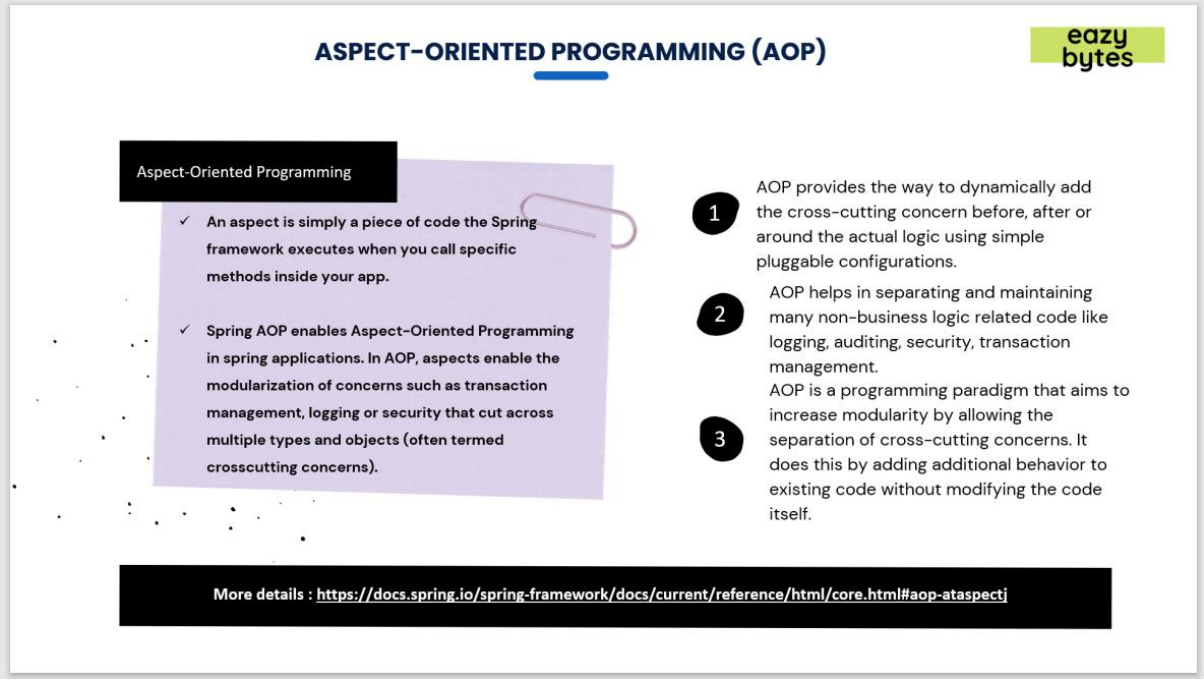
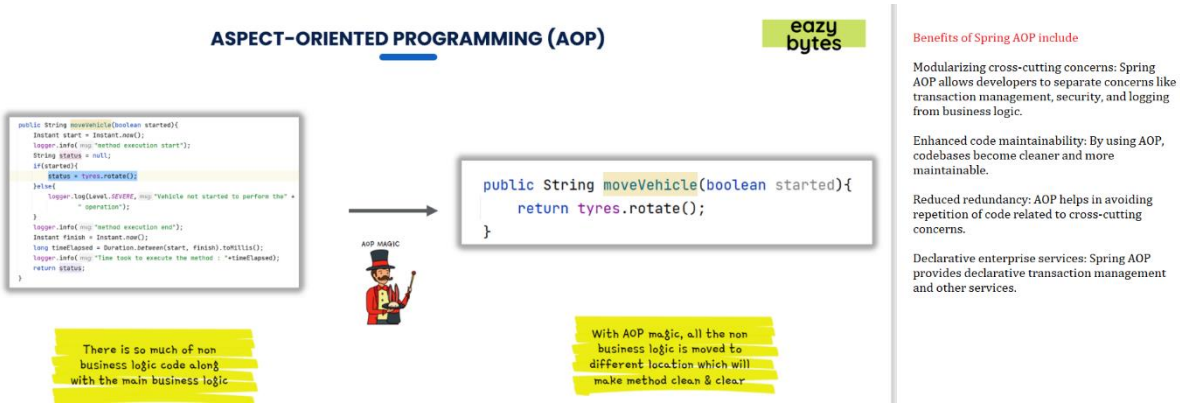
49.Introduction to Aspect Oriented Programming:



Basically like moving non-business logic to different module.

50: Understanding the Problems inside Web Application without AOP.



We will be moving non-business logic to different module.

We will not be creating beans for the class which does not have business logic.

51.Understanding and Running Application without AOP.

52.AOP Jargons

ASPECT-ORIENTED PROGRAMMING (AOP)

eazy bytes

AOP Jargons

✓ When we define an Aspect or doing configurations, we need to follow WWW (3 Ws)

- WHAT -> Aspect
- WHEN -> Advice
- WHICH -> Pointcut

Logic related to logging,Security checks and few non-bussiness logic.

1. WHAT code or logic we want the Spring to execute when you call a specific method. This is called as Aspect.
2. WHEN the Spring need to execute the given Aspect. For example is it before or after the method call. This is called as Advice.
3. WHICH method inside App that framework needs to intercept and execute the given Aspect. This is called as a Pointcut.

- Join point which defines the event that triggers the execution of an aspect. Inside Spring, this event is always a method call.
- Target object is the bean that declares the method/pointcut which is intercepted by an aspect.

we will decide which methods of your application needs to be intercepted.
And these methods stay inside a bean object.
So whatever bean that holds the actual methods which have business logic inside them, that bean we call it as target object.

In Java mostly the Join Point is method call.

ASPECT-ORIENTED PROGRAMMING (AOP)

eazy bytes

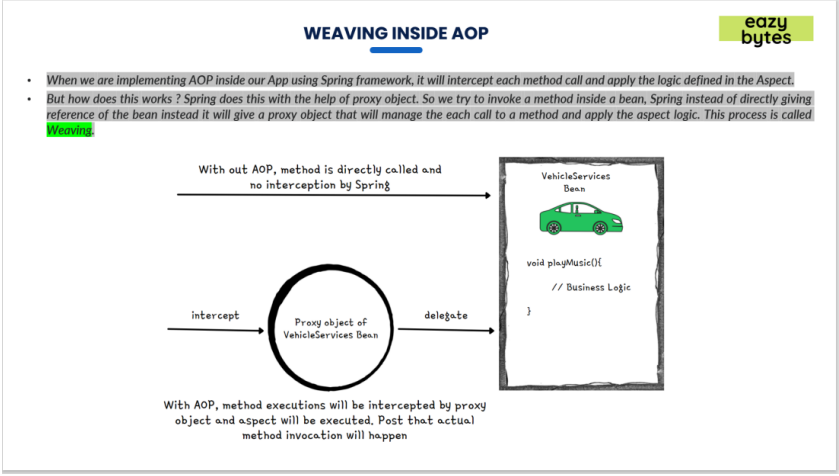
Typical Scenario of AOP implementation

Aspect Advice Joinpoint

Developer want some logic to be executed before each execution of method playMusic() present inside the bean VehicleServices.

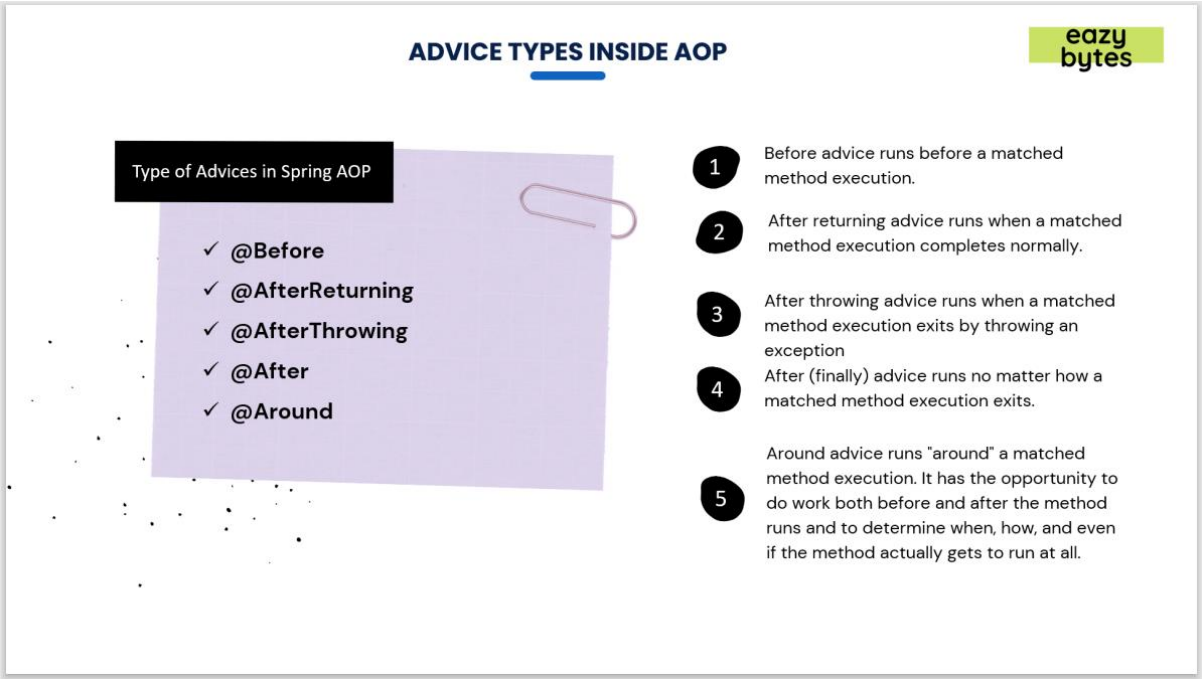
Point cut Target Object

53:Weaving inside AOP.



whenever I'm trying to get the bean of vehicle service bean from the spring context, spring context will never give the reference of vehicle service bean directly.
Instead, it will give a reference of proxy object which belongs to vehicle servicebean, So this proxy object is responsible to intercept the each and method call that is happening to the play music method inside the vehicle service bean.
Once it is intercepted based upon the advice configurations and aspect logic that we have defined, all the code related to aspect will be executed and
post that only based upon the conditions that we defined.
The delegation of the actual method call will go to the play music.
So that's why whenever you implement AOP there is a proxy object of vehicle service bean or of your target bean will be maintained by the spring context and the same proxy object bean will be injected as part of dependency injection and autowiring.

54:Types of Advices in AOP:



@Around -Runs before and after matched method runs.

55:Configuring Advices inside AOP-Theory :

CONFIGURING ADVICES INSIDE AOP

eazy bytes

- *We can use the AspectJ pointcut expression to provide details to Spring about what kind of methods it needs to intercept by mentioning details around modifier, return type, name pattern, package name pattern, params pattern, exceptions pattern etc.*
- *Below is the format of the same,*

Used to define method modifiers like public, private

Used to define the desired return type of the method

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?
 name-pattern(param-pattern) throws-pattern?)
```

Used to define any name pattern, package pattern, params pattern of the method

Used to define the specific exception pattern that can be thrown by the method

? --Indicates that it is not mandatory.

Execution expression approach

- *Like shown below, we can mention the pointcut expressions as an input after advise annotations that we use,*

```
@Configuration
@ComponentScan(basePackages = {"com.example.implementation",
            "com.example.services", "com.example.aspects"})
@EnableAspectJAutoProxy
public class ProjectConfig {

}
```

Used to enable AOP inside the Application

@Aspect Annoatation to differentaite with other beans.

```
@Aspect
@Component
public class LoggerAspect {

    @Around("execution(* com.example.services.*.*(..))")
    public void log(ProceedingJoinPoint joinPoint) throws Throwable {
        // Aspect Logic
    }
}
```

First * -- Any Return Type
com.example.services -- Package Name
com.example.services.* -- Every thing that package with any name
com.example.services.*.* -- Every thing that package with any method
com.example.services.*.*(..) -- Every thing that package with any classname and method with any number of parameters.

@EnabledAspectJAutoproxy.

So this annotation will enable the AOP inside your application. So this is an indication to your spring IOC container that my application has AOP implemented and it needs to create the proxy objects also for all the target beans where we have implemented aspect j programming.

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression follows:

```
execution(modifiers-pattern?
                ret-type-pattern
                declaring-type-pattern?name-pattern(param-pattern)
                throws-pattern?)
```

All parts except the returning type pattern (ret-type-pattern in the preceding snippet), the name pattern, and the parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. * is most frequently used as the returning type pattern. It matches any return type. A fully-qualified type name matches only when the method returns the given type. The name pattern matches the method name. You can use the * wildcard as all or part of a name pattern. If you specify a declaring type pattern, include a trailing . to join it to the name pattern component. The parameters pattern is slightly more complex: () matches a method that takes no parameters, whereas (..) matches any number (zero or more) of parameters. The (*) pattern matches a method that takes one parameter of any type. (*,String) matches a method that takes two parameters. The first can be of any type, while the second must be a String. Consult the Language Semantics section of the AspectJ Programming Guide for more information.

The following examples show some common pointcut expressions:

The execution of any public method:
execution(public * *(..))
The execution of any method with a name that begins with set:
execution(* set*(..))
The execution of any method defined by the AccountService interface:
execution(* com.xyz.service.AccountService.*(..))
The execution of any method defined in the service package:
execution(* com.xyz.service.*.*(..))
The execution of any method defined in the service package or one of its sub-packages:
execution(* com.xyz.service..*.*(..))
Any join point (method execution only in Spring AOP) within the service package:
within(com.xyz.service.*)
Any join point (method execution only in Spring AOP) within the service package or one of its sub-packages:
within(com.xyz.service..*)
Any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:
this(com.xyz.service.AccountService)

56.Configuring @Around Advice:

Group id will be similar for all the Spring framework libraries.

We need add the Spring AOP dependence to the existing dependencies.

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.2.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>6.2.1</version>
    </dependency>
```

```
@Aspect  1 usage
@Component
public class LoggerAspect {

    //private static final Log log = LogFactory.getLog(LoggerAspect.class);
    private Logger logger=Logger.getLogger(LoggerAspect.class.getName());  3 usages

    @Around("execution(* com.example.services.*.*(..))")  no usages
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable
    {
        logger.info( msg: joinPoint.getSignature().toString()+"Method Excecution Start");
        Instant start=Instant.now();
        Object result =joinPoint.proceed();
        Instant finish=Instant.now();
        long timeElapsed= Duration.between(start,finish).toMillis();
        logger.info( msg: "Time took for the completion of the method : "+timeElapsed);
        logger.info( msg: joinPoint.getSignature().toString()+"Method Excecution Ends");
        return result;
    }
```

Here we have used ProceedingJoinPoint so that it will be aware from Where to proceed.

Ex_20_Around_Advice_Example

57: Configuring @Before Advice

```java
public class VehicleService {

    private Logger logger = Logger.getLogger(VehicleService.class.getName());  no usages


    public String playMusic(boolean vehicleStarted){  no usages

        if(vehicleStarted)
        {
            return "Playing";
        }
        else {
            return "Vehicle not started";
        }
    }

    public String moveVehicle(boolean vehicleStarted){  no usages

        if(vehicleStarted)
        {
            return "Vehicle is Moving";
        }
        else {
            return "Vehicle not started";
        }
```

Here this vehicleStarted check is also common and it is like security check.we can move this different aspect.

```java
package com.example.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

import java.util.logging.Logger;

@Aspect
@Component
@Order(1)
public class VehicleStartCheckAspect {

    private Logger logger = Logger.getLogger(VehicleStartCheckAspect.class.getName());

    @Before("execution(* com.example.services.*.*(..)) && args(vehicleStarted,..)")
    public void checkVehicleStarted(JoinPoint joinPoint, boolean vehicleStarted) throws Throwable {
        if(!vehicleStarted){
            throw new RuntimeException("Vehicle not started");
        }
    }
}
```

JoinPoint as the method invocation starts only after this completion.

```java
package com.example.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

import java.util.logging.Logger;

@Aspect
@Component
@Order(1)
public class VehicleStartCheckAspect {

    private Logger logger = Logger.getLogger(VehicleStartCheckAspect.class.getName());

    @Before("execution(* com.example.services.*.*(..)) && args(vehicleStarted,..)")
    public void checkVehicleStarted(JoinPoint joinPoint, boolean vehicleStarted) throws Throwable {
        if(!vehicleStarted){
            throw new RuntimeException("Vehicle not started");
        }
    }
}
```

```java
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.context.annotation.Bean;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

import java.time.Duration;
import java.time.Instant;
import java.util.logging.Level;
import java.util.logging.Logger;

@Aspect
@Component
@Order(2)
public class LoggerAspect {

    private Logger logger = Logger.getLogger(LoggerAspect.class.getName());

    @Around("execution(* com.example.services.*.*(..))")
    public void log(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info(joinPoint.getSignature().toString() + " method execution start");
        Instant start = Instant.now();
        joinPoint.proceed();
        Instant finish = Instant.now();
        long timeElapsed = Duration.between(start, finish).toMillis();
        logger.info("Time took to execute the method : "+timeElapsed);
        logger.info(joinPoint.getSignature().toString() + " method execution end");
    }
```

when we have mutiple Aspects we can provide ordering to them so that they will get excecuted in the same order.
If we dont provide any ordering then they will be excecuted in random order.

Ex_21_Before_Advice

58.Configuring @AfterThrowing and @AfterReturning Advice

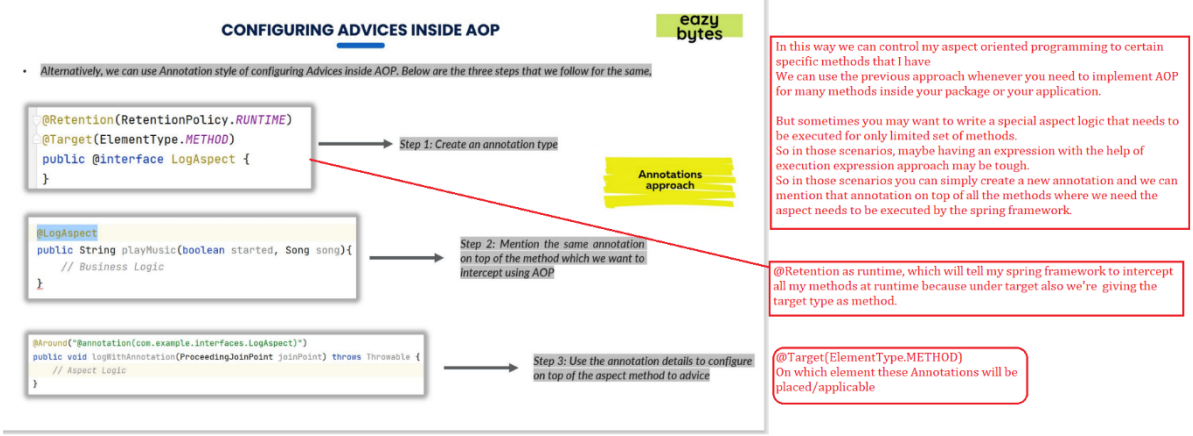
```java
    @Around("@annotation(com.example.interfaces.LogAspect)")
    public void logWithAnnotation(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info(joinPoint.toString() + " method execution start");
        Instant start = Instant.now();
        joinPoint.proceed();
        Instant finish = Instant.now();
        long timeElapsed = Duration.between(start, finish).toMillis();
        logger.info("Time took to execute the method : "+timeElapsed);
        logger.info(joinPoint.getSignature().toString() + " method execution end");
    }

    @AfterThrowing(value = "execution(* com.example.services.*.*(..))",throwing = "ex")
    public void logException(JoinPoint joinPoint, Exception ex) {
        logger.log(Level.SEVERE,joinPoint.getSignature()+ " An exception thrown with the help of" +
                " @AfterThrowing which happened due to : "+ex.getMessage());
    }

    @AfterReturning(value = "execution(* com.example.services.*.*(..))",returning = "retVal")
    public void logStatus(JoinPoint joinPoint,Object retVal) {
        logger.log(Level.INFO,joinPoint.getSignature()+ " Method successfully processed with the status " +
                retVal.toString());
    }
}
```

Will be excecuted when the method returns any Exception.
ex catches the Exception and takes the message and prits the same.

Will be excecuted when the completes successfully without any Exception.The retval value will be captured into retVal

Ex_22_After_Throwing_Returning

59:Configuring Advices inside AOP with Annotation Approach



60:Demo of Configuring advices inside AOP with Annotation Approach

Annotation Approach is used when we want to apply AOP to specific methods.

Execution expression approach is used when all the methods need to considered for Advice in the package.

Ex_23/Ex_17 of Eazy Bytes Code