Chapter 8 : AOP & OOP

Object-Oriented Programming (OOP)

Definition:

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of “objects,” which can contain data (fields or attributes) and code (methods or functions). OOP models real-world entities as software objects that interact with each other.

Core Concepts of OOP:

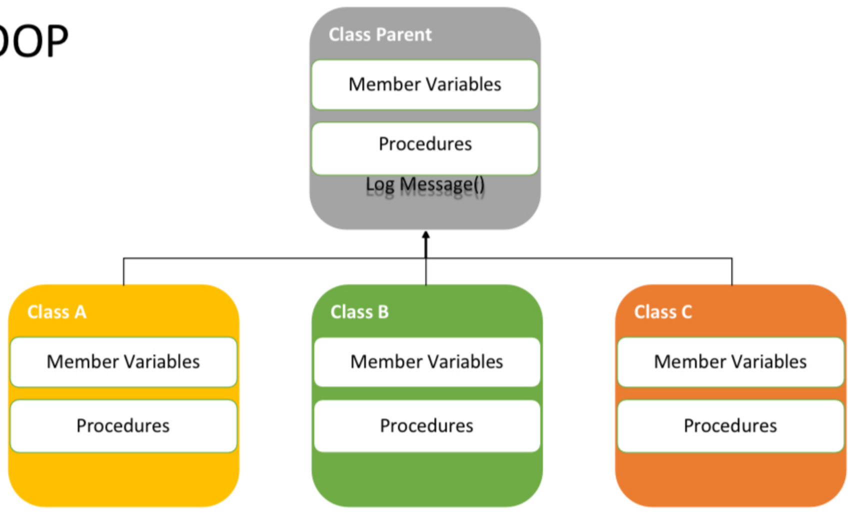
1. Encapsulation – Hides internal object details and exposes only what is necessary.
2. Abstraction – Simplifies complex reality by modeling classes appropriate to the problem.
3. Inheritance – Allows a class to inherit behavior and state from another class.
4. Polymorphism – Allows methods to behave differently based on the object calling them.

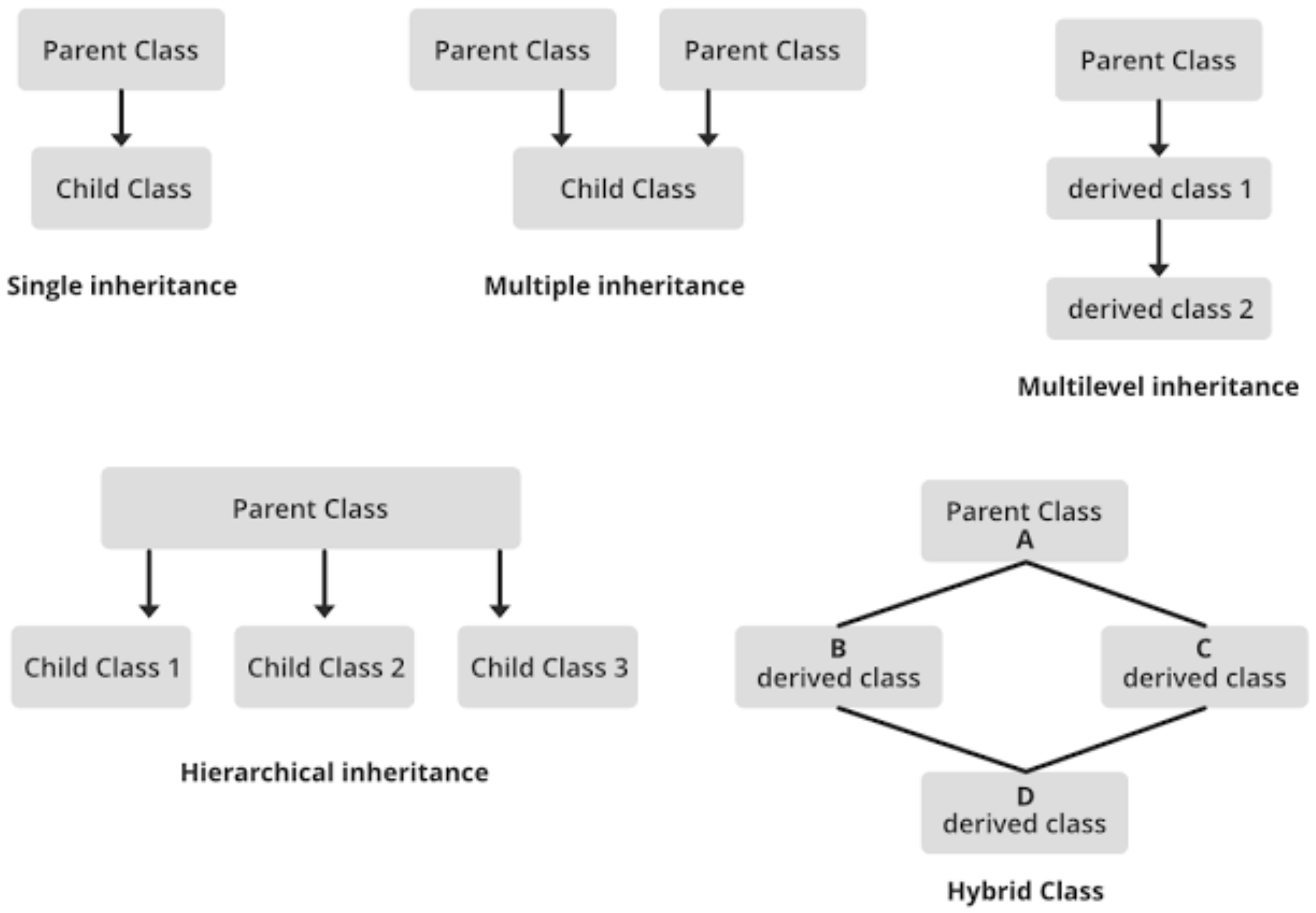
Advantages of OOP:

- Code reusability through inheritance.
- Modularity, as each object forms a separate entity.
- Easier to maintain and scale.
- Promotes real-world modeling.

- Encourages design clarity and team collaboration.

OOP





Example Use Case:

In a banking system, different classes like Account, Customer, and Transaction can be modeled using OOP principles, where each class has its own behavior and properties.

```
// Base class (Parent)
public class Animal {
   public void makeSound() {
      System.out.println("Some generic sound...");
   }
}

// Derived class (Child)
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark!");
    }
}

// Main class to demonstrate polymorphism
public class Zoo {
    public static void main(String[] args) {
        Animal a = new Dog();  // Polymorphism
        a.makeSound();         // Output: Bark!
    }
}
```

🔍 **Explanation:**

- Animal is a base class.
- Dog is a child class that overrides the makeSound() method.
- In Zoo, we create a Dog but reference it as an Animal. This is polymorphism.

```
# Base class
```

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        print("Some generic animal sound.")

# Derived class (inherits from Animal)
class Dog(Animal):
    def make_sound(self):
        print(f"{self.name} says: Woof!")

# Another derived class
class Cat(Animal):
    def make_sound(self):
        print(f"{self.name} says: Meow!")

# Main code
def main():
    animals = [Dog("Buddy"), Cat("Whiskers")]

    for animal in animals:
        animal.make_sound()

main()
```

•	Class Animal is the base class. It has a constructor (__init__) and a method make_sound.

- Classes Dog and Cat are derived classes that override the make_sound() method.
- In the main() function, we create a list of animals and call make_sound() on each. This demonstrates polymorphism: the same method name behaves differently depending on the object's class.

✅ Concerns in Software Engineering

In software engineering, concerns refer to the various interests, priorities, and requirements expressed by stakeholders regarding a system. These are not bugs or issues, but rather aspects of the system that must be addressed during its development and maintenance.

🔶 What Are Concerns?

- Concerns are reflections of system requirements and stakeholder priorities.
- They may include functionality, performance, security, maintainability, reliability, and more.

- They help drive design decisions and influence system architecture.
- Separation of concerns (SoC) is a design principle that promotes dividing a system into distinct features with minimal overlap to simplify development and maintenance.

🔶 Types of Concerns

1. Core Concerns
- These are functional concerns that relate to the primary purpose of the system.
- Example: For an e-commerce application, processing payments and managing inventory are core concerns.

2. Secondary Concerns
- These are non-functional or cross-cutting concerns that are important but not the main goal.
- Example: Logging, security, error handling, performance monitoring.

Type of Concern	Description	Example
Functional Concerns	Specific features and functionalities the	User authentication, order placement

	system must provide	
Quality of Service Concerns	Related to non-functional behavior such as performance, availability	Response time under 2 seconds
Policy Concerns	High-level rules or regulations governing system behavior	GDPR compliance, access control rules
System Concerns	Attributes of the system as a whole, including architecture	Scalability, configurability
Organisational Concerns	Related to business goals, budget, reuse, or company reputation	Minimize development cost, meet deadlines

🔶 **Importance of Concerns**

- **Traceability: Reflecting concerns in the system design improves traceability from stakeholder requirements to code.**
- **Clarity: Helps different teams understand their roles and what aspects of the system they are responsible for.**
- **Modularity: Encourages clean separation, making code easier to maintain and scale.**
- **Better Decision-Making: Enables more informed trade-offs between performance, cost, complexity, etc.**

🔶 **Example**

Imagine a hospital management system:

- Functional Concern: Register a new patient.
- Quality of Service Concern: Ensure system uptime is above 99.9%.
- Policy Concern: Only authorized personnel can access patient records.
- System Concern: The system should be easy to configure for different hospital sizes.
- Organisational Concern: Deliver the product within 6 months using existing software tools.

Aspect-Oriented Programming (AOP)

Definition:

Aspect-Oriented Programming (AOP) is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. These are aspects of a program that affect multiple modules, such as logging, security, or transaction management.

Core Concepts of AOP:

1. Aspect – A module that encapsulates a cross-cutting concern.

2. Join Point – A point in the program (e.g., method call) where an aspect can be applied.
3. Advice – The code to be executed at a join point (before, after, or around).
4. Pointcut – An expression that selects one or more join points.
5. Weaving – The process of applying aspects to a target object, usually done at compile time, load time, or runtime.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* OrderService.placeOrder(..))")
    public void logBefore() {
        System.out.println("Logging BEFORE placing the order.");
    }

    @After("execution(* OrderService.placeOrder(..))")
    public void logAfter() {
        System.out.println("Logging AFTER placing the order.");
    }
}
```

Advantages of AOP:

- Improves separation of concerns.
- Reduces code duplication by centralizing cross-cutting logic.
- Enhances maintainability and readability.
- Allows focus on core business logic, abstracting repetitive tasks.

AOP Advice Types

In Aspect-Oriented Programming (AOP), *advice* **is the action taken by an aspect at a particular** ***join point*** **in the program (a specific point in the execution of the program, such as method execution).**

There are five main types of AOP advice:

---

1. Before Advice
   - Executed before the target method is invoked.

- Useful for logging, security checks, validation, etc.

```
@Before("execution(*
com.example.service.OrderService.placeOrder(..))")
public void logBefore() {
   System.out.println("Before placing an order");
}
```

----

2. After (Finally) Advice

- Executed after the target method finishes, regardless of its outcome (success or exception).
- Commonly used for cleanup tasks or logging.

```
@After("execution(*
com.example.service.OrderService.placeOrder(..))")
public void logAfter() {
   System.out.println("After placing an order (finally)");
}
```

----

3. After Returning Advice

- Executed after the method successfully returns (no exception thrown).

- Can access the return value and perform post-processing.

```
@AfterReturning(
   pointcut = "execution(*
com.example.service.OrderService.getTotal(..))",
   returning = "total"
)
public void logAfterReturning(double total) {
   System.out.println("Order total returned: " + total);
}
```

----

4. After Throwing Advice

- Executed only if the method throws an exception.
- Useful for logging errors or performing rollback actions.

```
@AfterThrowing(
   pointcut = "execution(*
com.example.service.PaymentService.process(..))",
   throwing = "ex"
)
public void logException(Exception ex) {
    System.out.println("Exception occurred: " + ex.getMessage());
}
```

----

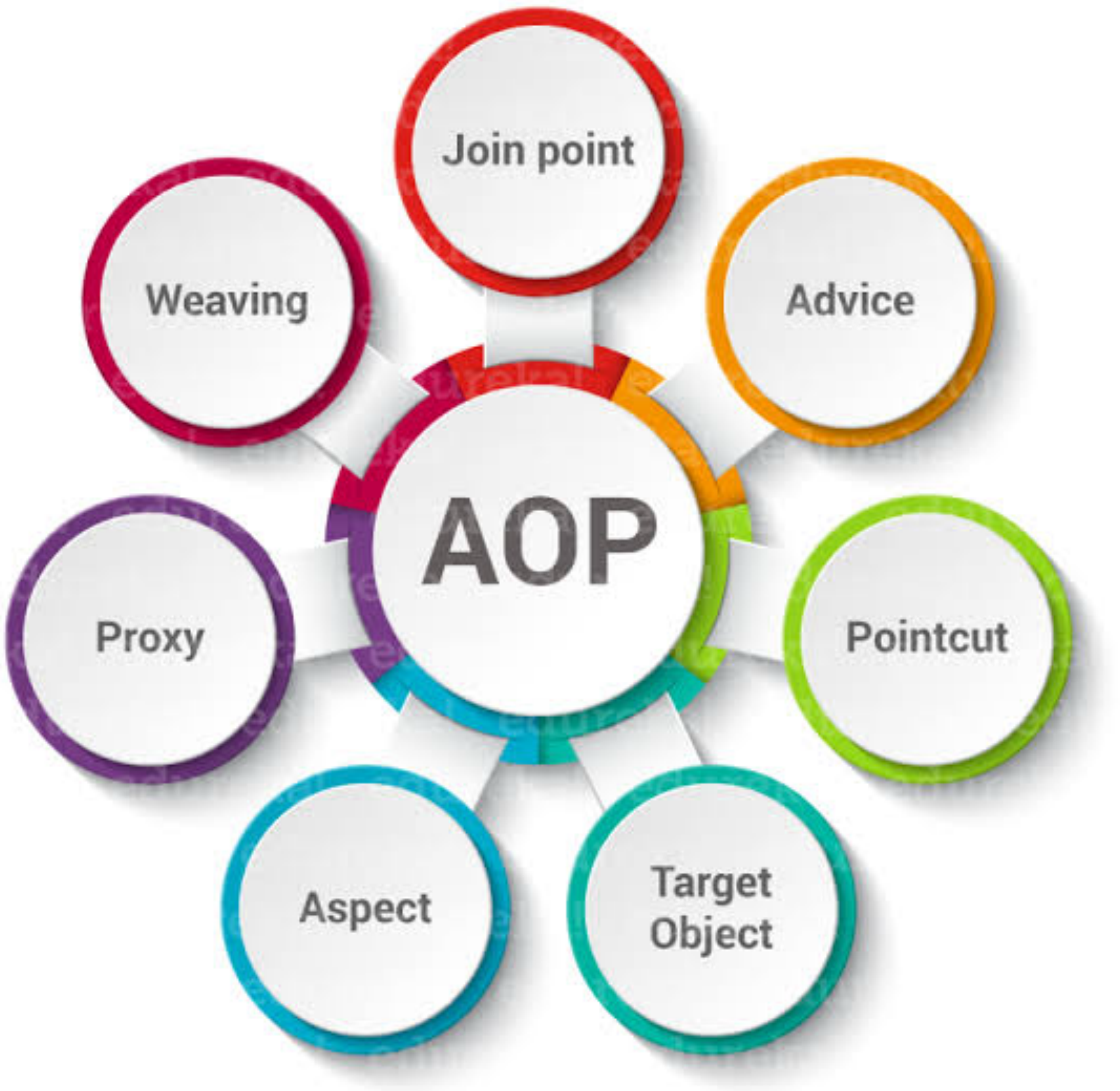
5. Around Advice

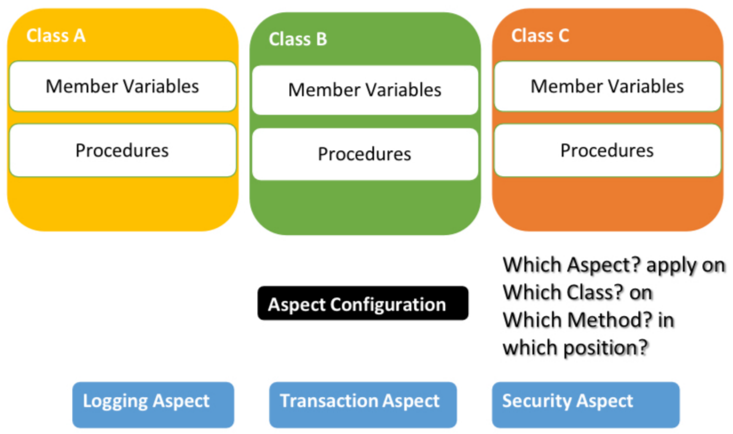
- Wraps the target method; can control whether the method executes or not.
- The most powerful type of advice — can modify input, return value, or even suppress execution.

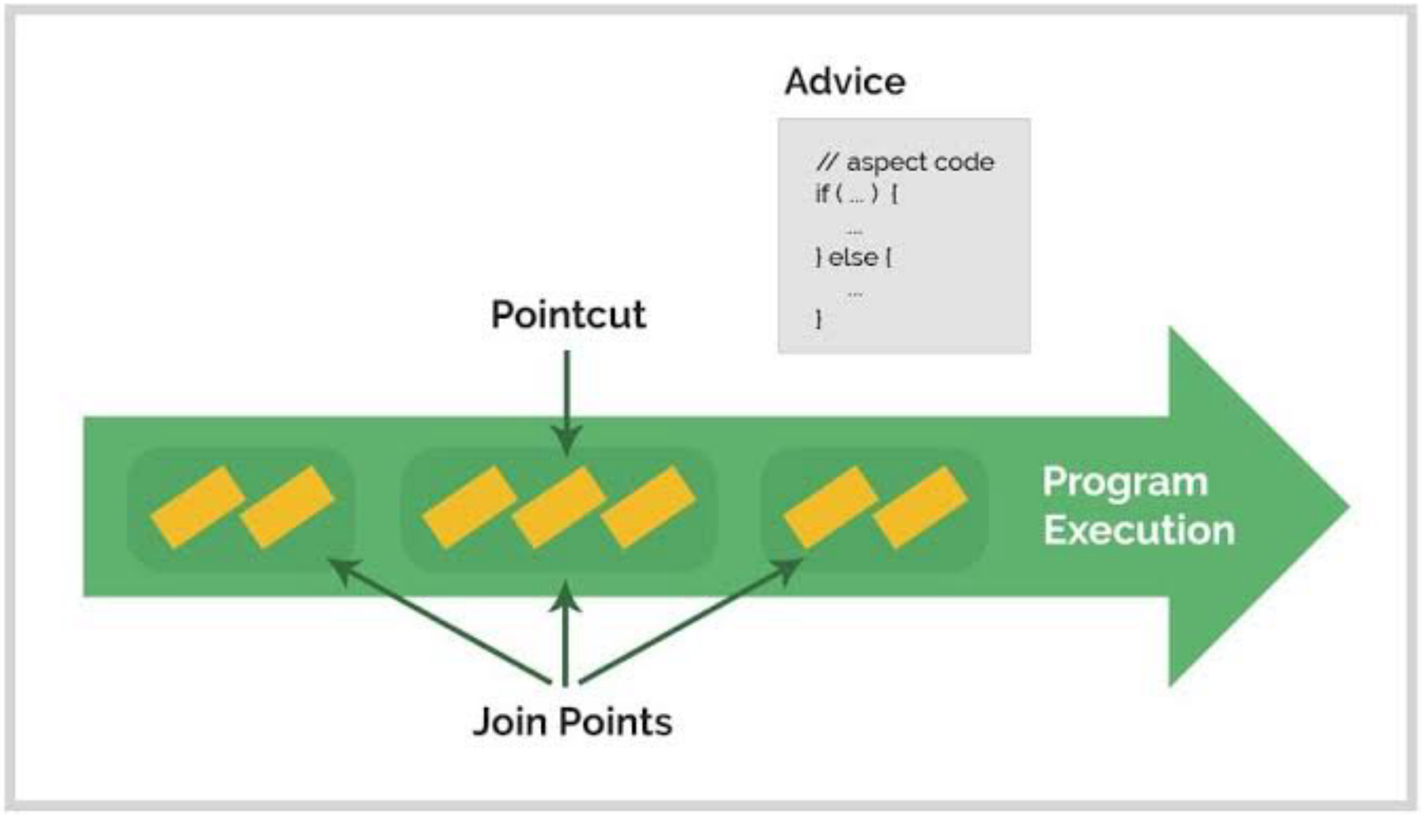
```
@Around("execution(* com.example.service.OrderService.placeOrder(..))")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("Before around advice");
    Object result = joinPoint.proceed();  // proceed to the actual method
    System.out.println("After around advice");
    return result;
}
```

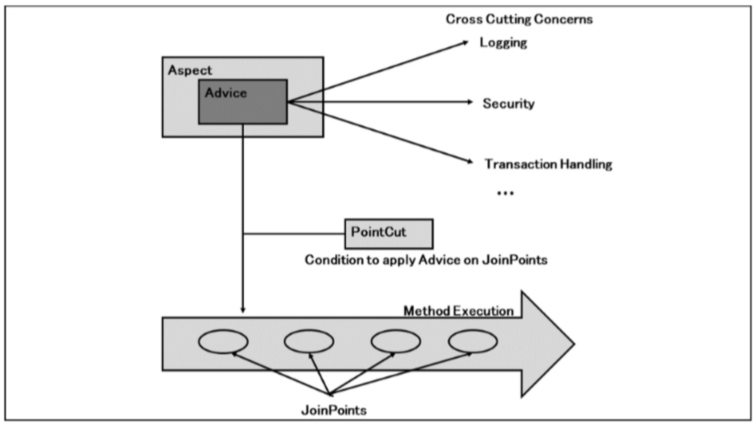
Example Use Case:

In an enterprise application, you can use AOP to log every method execution or enforce security checks without modifying the actual business logic of the application classes.




Class A
Member Variables
Procedures
Class B
Member Variables
Procedures
Class C
Member Variables
Procedures
Aspect Configuration
Which Aspect? apply on
Which Class? on
Which Method? in
which position?
Logging Aspect
Transaction Aspect
Security Aspect


Advice
// aspect code
if (...) {
...
} else {
...
}
Pointcut
Program Execution
Join Points



Feature	OOP	AOP
Focus	Data and behavior modeling	Cross-cutting concerns separation
Modular Unit	Class	Aspect
Code Reuse	Achieved through inheritance and composition	Achieved by applying aspects to multiple points
Typical Use Cases	Business logic, entity modeling	Logging, security, transactions, auditing
Cross-cutting Concerns	Scattered across classes	Centralized into reusable aspects
Coupling	Can lead to tight coupling	Promotes loose coupling for cross-cutting concerns
Language Support	Java, C++, Python, etc.	Requires frameworks/ tools (e.g., Spring AOP in Java)

When to Use AOP

- When you have repetitive code scattered across multiple classes.
- When you want to enforce consistent behavior (e.g., logging, authentication).
- When working in enterprise-level applications with complex infrastructure needs.

When to Use OOP

- When modeling real-world systems using entities and behaviors.
- When building modular, reusable components.
- When starting with clean, maintainable architectures for scalable systems.

Conclusion:

- OOP is ideal for organizing and structuring the core logic of applications.
- AOP complements OOP by modularizing behaviors that span multiple objects, enhancing code modularity and maintainability.

Both paradigms are not mutually exclusive. In fact, they are often used together—for example, in Spring Framework (Java), where you define your core logic using OOP and manage cross-cutting concerns using AOP.