About me

- Sudhin Varghese
  - 2002 Passout
  - Technology, Movie and Music Buff
  - Currently Living in Dallas, TX
  - Working for Mr. Cooper
    - This presentation is purely educational and does not represent or reference any real-world operations, practices, or affiliations of the company I work for.

Talking about me

Thanks to Cinema

Java - The Greatest of All Time

Decoding The Language And Object-Oriented Programming Through Cinema

The Movie Premise

Characters as Classes

Classes represent entities in the movie:

Class Gandhi (attributes: name, profession, skills; methods: fight(), interrogate()).

Class Jeevan (attributes: alias, loyalty; methods: planRevenge(), detonateBomb()).

Class Menon (attributes: past, motivations; methods: manipulate(), kidnap()).

Objects as Classes – Abstracting the Characters

In object-oriented programming, classes represent entities, and objects are specific instances of those classes.

To promote abstraction and reusability, we define abstract classes or interfaces for shared behaviors and let specific classes implement or extend them.

Code Demo:

com/goat/objectClass/Main.java

Key Takeaways

Abstract Class: **Operative** provides shared structure and behavior.

Inheritance: **Strategist**, **Saboteur**, and **Manipulator** extend **Operative**, overriding **executeTask()** with their specific implementations.

Polymorphism: **executeTask()** is invoked on **Operative** references, while type-specific methods (e.g., **planRevenge()**) are invoked through casting.

Encapsulation

Definition:

Encapsulation is the bundling of data (fields) and methods into a single unit (class) while restricting direct access to them.

Access to data is controlled using modifiers like private, protected, and public, ensuring sensitive information is not exposed unnecessarily.

Encapsulation

Movie Analogy:

SATS team members like Gandhi and Sunil have privileged access to mission details due to their trusted roles.

Antagonists like Jeevan and Menon cannot directly access mission data but may attempt manipulation or sabotage through indirect means.

Code Demo:

com/goat/encapsulation/SatsTeam.java

Key Takeaways

Encapsulation ensures sensitive data is not directly accessible.

Private fields: Protect mission details from external access.

Controlled access through methods: Only trusted roles like SATS members can retrieve data.

Enhancing Security: Use role checks or class hierarchies to enforce stricter access controls.

Composition Vs Inheritance

Definition:

Inheritance: A mechanism where one class derives attributes and behavior from another class (is-a relationship).

Composition: A mechanism where a class is composed of one or more objects from other classes (has-a relationship).

Key Difference:

Inheritance creates a tight coupling between classes, while composition promotes loose coupling.

Movie Analogy (GOAT)

Inheritance Analogy:

If every SATS operative inherits a single "**Operative**" class, it forces them to share unnecessary behaviors (e.g., a strategist like Gandhi inheriting sabotage skills from Jeevan).

Composition Analogy:

Using composition, SATS members can be built with specific skill sets like **Fighter**, **Strategist**, or **Saboteur**, avoiding unnecessary behaviors and enabling flexibility.

Code Demo:

com/goat/composition/Main.java

Key Takeaways

Loose Coupling:

Gandhi and Jeevan are independent of shared, irrelevant behaviors.

Flexibility:

Behavior is added based on the object’s role without enforcing inheritance.

Reusability:

Fighter, Planner, and Saboteur behaviors can be reused across multiple classes.

Functional Programming in Java

Java introduced Streams and Functional Interfaces in Java 8 to support functional programming.

These concepts help simplify code, make it more readable, and handle large datasets efficiently.

In GOAT:

Gandhi manages missions as a series of operations (filter, map, collect).

Jeevan’s cloning and Menon’s manipulation reflect functional processing.

Lambda Expressions

Explanation:

A Lambda Expression provides a concise way to represent a method implementation.

Removes boilerplate code, enabling more readable and functional programming.

Movie Analogy:

In GOAT, Gandhi uses streamlined strategies to tackle missions.

Lambda expressions are like these strategies: concise, direct, and powerful.

Code Demo:

com/goat/lambdaexpression/LambdaExample.java

Functional Interface

Explanation:

A Functional Interface is an interface with a single abstract method (SAM).

Used as the target for lambda expressions or method references.

Movie Analogy:

Gandhi’s role as a leader requires one core action: delegate tasks (SAM).

Other team members (Sunil, Kalyan) implement this "action" in their unique ways.

Code Demo:

com/goat/lambdaexpression/functionalinterface/FunctionalInterfaceExample.java

Streams

Explanation:

Streams process collections and sequences of data with operations like filter, map, and reduce.

Can handle large datasets in a declarative way.

Movie Analogy:

Filtering and analyzing data, like Jeevan’s planning against Gandhi, mirrors how Streams work.

Example: Filter out "targets" from a list of potential threats.

Code Demo:

com/goat/lambdaexpression/streams/StreamExample.java

Combining Streams and Lambdas

Explanation:

Combining Streams and Lambda Expressions creates powerful and concise operations on data.

Movie Analogy:

Gandhi processes mission tasks step by step:

Filter for high-priority missions.

Map tasks to appropriate teammates.

Collect results of operations.

Code Demo:

com/goat/lambdaexpression/streams/MissionProcessor.java

Generics in Java

Definition:

Generics allow you to write reusable and type-safe code by defining classes, interfaces, and methods with type parameters.

Key Benefits:

Type Safety: Ensures compile-time type checking.

Reusability: Code can work with different types without duplication.

Eliminates Type Casting: No need for explicit casting while retrieving objects.

Code Demo:

com/goat/generics/Mission.java

Advanced Concepts in Generics

Bounded Generics

Definition: Bounded generics restrict the types that can be passed to a generic class or method.

Syntax: <T extends ClassName> restricts T to types that are subclasses of ClassName.

Movie Analogy (GOAT):

Gandhi might assign tasks only to members with specific skills (e.g., fighters or strategists).

Code Demo:

com/goat/generics/bounded/Mission.java

Wildcards in Generics

Definition: Wildcards allow flexibility in working with unknown types in generics.

Syntax:

<?>: Represents an unknown type.

<? extends Type>: Upper-bounded wildcard (subclasses of Type).

<? super Type>: Lower-bounded wildcard (superclasses of Type).

Movie Analogy (GOAT):

The SATS squad might need to operate with members of various roles without knowing their exact roles ahead of time.

Code Demo:

com/goat/generics/bounded/wildcard/WildcardExample.java

Key Takeaways

Bounded Generics: Restrict generics to specific types for more control.

Wildcards: Add flexibility when dealing with collections of unknown or related types.

Practical Use: Combine streams and generics to process lists of SATS team members or mission objectives.

Mastering Design Patterns in Java

What are Design Patterns?

Definition:

- Design patterns are proven solutions to common software design problems.
- They are templates for writing reusable, maintainable, and scalable code.

Purpose:

- To standardize software development.
- To improve communication among developers through a shared vocabulary.
- To avoid reinventing the wheel for recurring design challenges.

Why Use Design Patterns?

Structured Solutions: Provide a clear roadmap for addressing complex design problems.

Reusability: Encourage the reuse of designs and solutions, reducing development time.

Scalability: Help create flexible systems that can grow without major rewrites.

Maintainability: Ensure that the codebase is easier to understand and modify.

Types of Design Patterns

- **Creational Patterns** – Focus on object creation.
  - Examples: Singleton, Factory, Builder.
- **Structural Patterns** – Deal with object composition and relationships.
  - Examples: Adapter, Composite, Proxy.
- **Behavioral Patterns** – Focus on communication and responsibility between objects.
  - Examples: Strategy, Observer, Chain of Responsibility.

Singleton Pattern

The Singleton Pattern ensures a class has only one instance and provides global access to it.

Movie Analogy:

SATS team’s control center is a single entity coordinating all missions.

Code Demo:

com/goat/designpatterns/singleton/Main.java

Key takeaways

- Lazy Initialization with Static Inner Class:
  - The Holder class is loaded only when getInstance() is called for the first time.
- Ensures thread safety without requiring synchronization.
- Efficient Memory Usage:
  - The Singleton instance is not created until needed, reducing memory overhead.
- Thread Safety:
  - The JVM guarantees that class loading is thread-safe, so no explicit locking or synchronization is required.

Chain of Responsibility Pattern

What is Chain of Responsibility?

Definition:

A behavioral design pattern that allows a request to pass through a chain of handlers until one handles it.

Purpose:

Decouples sender and receiver of a request.

Allows dynamic addition or modification of handlers in the chain.

Movie Analogy (GOAT):

In GOAT, Gandhi’s SATS team processes mission tasks like intelligence gathering, combat operations, and sabotage.

Requests (e.g., "Stealth Mission" or "Bomb Defusal") are passed through handlers like intelligence experts, tactical operatives, or combat specialists.

Each handler processes the request if it matches their responsibility, or passes it along the chain.

Code Demo:

com/goat/designpatterns/chainofresponsibility/Main.java

Key takeaways

Benefits of Chain of Responsibility

Decoupling: Sender doesn’t need to know which handler will process the request.

Flexibility: Handlers can be added or changed dynamically.

Single Responsibility: Each handler focuses on a specific task.

Strategy Pattern

The Strategy Pattern allows a family of algorithms to be defined and interchangeable at runtime.

Movie Analogy:

Gandhi employs different tactics against Menon and his team, adapting strategies based on the situation.

Code Demo:

com/goat/designpatterns/strategy/Main.java

Introduction to SOLID Principles

SOLID Principles ensure maintainable, scalable, and robust software design.

Acronym for:

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

What Next?