FREE Live Classes

Java

Theory

Coding

RedSysTech
-Love to Learn...

Core Java

by Karthik Ponnusamy

Course Contents:-

- Core Java Basic Concepts
- OOP Concepts
- Fundamentals of Java Programming
- Java Control Flow Statements
- Deep dive into Collection Framework
- Deep dive into Multithreading
- Java 1.8 Features

Batch Starts from Jan - 18 - 2022

Mon to Thurs - 10 PM to 11 PM CST /
9:30 AM to 10:30 AM IST

Day 02: Agenda

Basics of Design patterns

Design patterns - What, Why?

Types of Design patterns - Creational, Structural and Behavioral

Use Cases to study each of these Design patterns

Design patterns - laymen terms

Design patterns are like reusable templates that can be applied to solve common problems in software development.

Just like a recipe provides a step-by-step guide to cooking a particular dish, a design pattern provides a guide to solving a specific problem in software design.

For example, imagine you want to build a house with several rooms. Instead of starting from scratch, you can use a pre-designed floor plan that includes a layout of the rooms, doors, and windows.

This is similar to using a design pattern in software development - instead of starting from scratch, you can use a pre-designed solution to a common problem.

Design patterns are useful because they have been tested and proven to work over time. They can save time and effort in the design process by providing a reliable way to solve a problem, and they can help to ensure that software systems are more efficient, maintainable, and scalable.

Design patterns - What it contains?

A design pattern diagram is a visual representation of the structure and relationships between the components of a design pattern.

It typically shows the classes, objects, interfaces, and their interactions in a design pattern.

The diagram may include the following components:

Classes: These represent the objects or entities involved in the design pattern.

Interfaces: These define the interactions and behaviors of the objects involved.

Relationships: These show the connections between the classes and interfaces in the design pattern.

Roles: These describe the responsibilities and behaviors of each object or entity in the design pattern.

UML notation: Design pattern diagrams are often represented using Unified Modeling Language (UML) notation, which provides a standardized way to represent object-oriented design concepts.

Design pattern diagrams can be useful for understanding the structure and behavior of a design pattern, as well as for communicating the design pattern to other developers. They can help to clarify the relationships between the components of a design pattern and highlight its key features and benefits.

Design patterns - What, Why?

A design pattern is a reusable solution to a common problem that occurs in software design. It is a general approach or a template that can be applied to solve a particular design problem in a consistent and efficient way. A design pattern can be seen as a proven solution to a recurring problem in software design, which has been tested and refined over time.

Design patterns can be applied at different levels of software development, including software architecture, design, and coding. They can be used to improve the quality, efficiency, maintainability, and scalability of software systems.

There are several types of design patterns, including creational patterns, structural patterns, and behavioral patterns.
Creational patterns are concerned with object creation mechanisms, trying to create objects in a manner suitable to the situation.
Structural patterns are about organizing objects and classes into larger structures, such as a class hierarchy, to solve larger problems.
Behavioral patterns are about communication between objects, how objects communicate and how they operate together to achieve a common goal.

Design patterns provide a common vocabulary for software developers to communicate and share knowledge about software design problems and solutions. They can help to simplify and streamline the design process, reduce the risk of errors and mistakes, and improve the overall quality of software systems.

Types of Design patterns - Creational, Structural and Behavioral

Creational:

These design patterns are all about class instantiation or object creation.

Creational design patterns are the

1. Factory Method,
2. Abstract Factory,
3. Builder,
4. Singleton,
5. Object Pool,
6. Prototype.

Structural

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are

1. Adapter,
2. Bridge,
3. Composite,
4. Decorator,
5. Facade,
6. Flyweight,
7. Private Class Data,
8. Proxy.

Behavioral

Behavioral patterns are about identifying common communication patterns between objects and realizing these patterns.

Behavioral patterns are

1. Chain of responsibility,
2. Command,
3. Interpreter,
4. Iterator,
5. Mediator,
6. Memento,
7. Null Object,
8. Observer,
9. State,
10. Strategy,
11. Template method,
12. Visitor

Creational - Use Cases

One common use case for creational design patterns in Java is when you need to create objects in a way that is more flexible or extensible than just using the new operator directly. Here are some examples:

Factory Method pattern: This pattern is used to create objects of a particular class, but the exact type of object is determined at runtime based on some conditions or parameters. For example, if you have a Shape interface with several implementations (e.g. Circle, Rectangle, Triangle), you can use a ShapeFactory class that takes a String parameter to decide which type of shape to create. This allows you to add new types of shapes without changing the client code that uses the ShapeFactory.

Abstract Factory pattern: This pattern is used when you need to create a family of related objects, but you want to hide the details of the concrete classes from the client code. For example, if you have a GUIFactory interface with several implementations (e.g. WindowsGUIFactory, MacGUIFactory), you can use an AbstractGUIFactory class that returns objects of different types (e.g. Button, Label, TextBox) depending on which factory is used. This allows you to create GUIs for different platforms without changing the client code.

Builder pattern: This pattern is used when you need to create objects that have many optional parameters, or when the process of constructing the object is complex and needs to be separated from the object itself. For example, if you have a Person class with many optional fields (e.g. middle name, address, phone number), you can use a PersonBuilder class that takes care of the construction process and returns a Person object. This allows you to create Person objects with only the required fields, or with all the optional fields in any order.

Singleton pattern: This pattern is used when you need to ensure that there is only one instance of a particular class in the entire application. For example, if you have a Logger class that is used throughout the application to write messages to a file or console, you can use a LoggerSingleton class that provides a single instance of the Logger object. This ensures that all the messages are written to the same file or console, and prevents multiple instances from interfering with each other.

Structural - Use Cases

Structural design patterns in Java are useful when you need to organize your code in a way that separates the interface from the implementation or when you need to compose objects into larger structures. Here are some examples:

Adapter pattern: This pattern is used when you need to adapt an existing class or interface to work with a different interface. For example, if you have a LegacyRectangle class with methods drawLegacy() and resizeLegacy(), you can use an Adapter class that implements a Shape interface and calls the drawLegacy() and resizeLegacy() methods. This allows you to use the LegacyRectangle class with other classes that expect a Shape interface.

Decorator pattern: This pattern is used when you need to add functionality to an existing object without changing its interface. For example, if you have a Window interface with a draw() method, you can use a BorderDecorator class that adds a border around the window by calling the draw() method of the Window interface and then drawing a border. This allows you to add new features to the window without changing its interface.

Facade pattern: This pattern is used when you need to provide a simplified interface to a complex subsystem. For example, if you have a complex system with many classes and methods, you can use a Facade class that provides a simplified interface to the most commonly used methods. This allows you to hide the complexity of the system from the client code.

Proxy pattern: This pattern is used when you need to control access to an object or when you need to defer the creation of an object until it is actually needed. For example, if you have a RealObject class that is expensive to create or that requires special permissions to access, you can use a Proxy class that checks permissions or delays the creation of the RealObject until it is actually needed. This allows you to control access to the RealObject and improve performance by deferring its creation until it is needed.

Behavioral - Use Cases

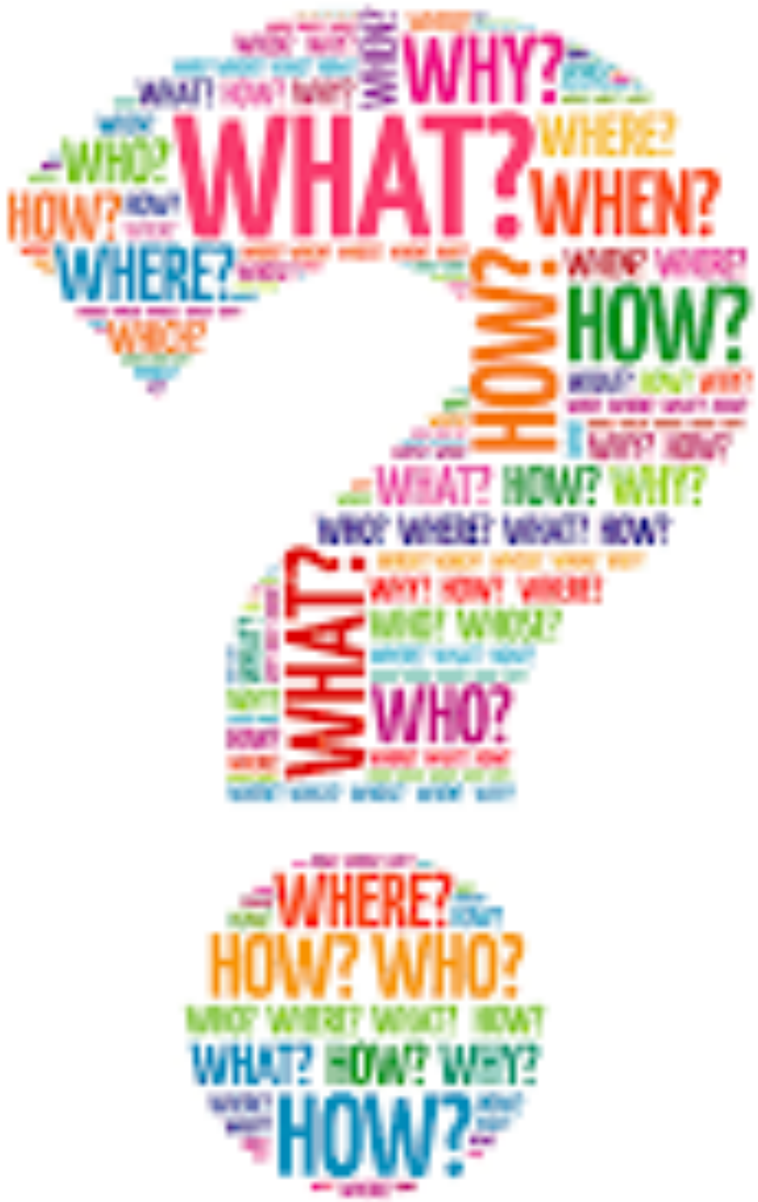
Behavioral design patterns in Java are useful when you need to define how objects interact with each other or when you need to define how objects behave in a particular situation. Here are some examples:

Observer pattern: This pattern is used when you need to notify a set of objects when a state change occurs in another object. For example, if you have a Subject class with a list of Observer objects, you can use the Observer interface to define a notify() method that is called by the Subject class when its state changes. This allows the Observer objects to be notified and take appropriate action.

Command pattern: This pattern is used when you need to encapsulate a request as an object, thereby allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations. For example, if you have a Receiver class with several methods (e.g. action1(), action2(), action3()), you can use a Command interface that defines a execute() method to encapsulate each method as a command object. This allows you to queue or log the command objects, undo or redo actions, and parameterize clients with different commands.

Template method pattern: This pattern is used when you need to define the skeleton of an algorithm in a superclass, but allow subclasses to override certain steps of the algorithm without changing its structure. For example, if you have an Algorithm class with a run() method that contains several steps (e.g. step1(), step2(), step3()), you can use a TemplateMethod class that defines the run() method as a template and allows subclasses to override the step2() method. This allows you to reuse the common steps of the algorithm in different subclasses and customize the behavior of certain steps.

Iterator pattern: This pattern is used when you need to traverse a collection of objects without exposing its underlying representation or changing its interface. For example, if you have a Collection interface with several implementations (e.g. List, Set, Map), you can use an Iterator interface that defines a next() and hasNext() method to traverse the collection. This allows you to iterate over the collection without exposing its internal structure or changing its interface.


WHY?
WHAT?
WHERE?
WHEN?
WHO?
HOW?
WHERE?
HOW?
HOW?
WHAT? HOW? WHY?
WHO? WHERE? WHAT? HOW?
WHAT?
WHO?
WHERE?
HOW? WHO?
WHAT? HOW? WHY?
HOW?