**Object-Oriented Programming (OOP) Principles**

**Encapsulation**

Groups attributes and methods into a single unit known as a class. To ensure data integrity and security, the attributes are made private. This restricts their direct access from outside the class. Public methods are used to retrieve or modify the attributes.

**Inheritance**

Inheritance allows one class to inherit the properties and methods of another.

**Polymorphism**

Polymorphism enables methods to have different behaviors, achieved through:

**Method Overloading:** Different methods in a class have the same name but different parameters. This is resolved at compile time.

**Method Overriding:** A subclass implements a method that already exists in its parent class, allowing it to execute specific behavior during runtime.

These features allow methods to perform different behaviors under the same name.

**Abstraction**

It uses abstract classes and interfaces to abstract common methods across multiple implementations.

**Encapsulation:** Groups attributes (private) and methods (public) into a single unit, a class, to ensure data integrity and security.

Inheritance: Allows one class to inherit the properties and methods of another.

**Polymorphism:** Enables methods to have different behaviors through method overloading (compile-time) and method overriding (run-time).

**Abstraction:** Utilizes abstract classes and interfaces to abstract common methods across multiple implementations.

**SOLID principles**

The SOLID principles are guidelines in object-oriented design that enhance code readability, scalability and maintainability. Each letter of “SOLID” represents a core principle:

**Single Responsibility Principle:** A class have only one reason to change.

**Open/Closed Principle:** Classes should be open for extension but closed for modification.

**Liskov Substitution Principle:** Subclasses should be substitutable for their base classes without affecting program correctness.

**Interface Segregation Principle:** Splitting large interfaces into smaller, more targeted ones to ensure that implementing classes only need to interact with the methods that are relevant to them. This approach prevents classes from being forced to implement methods they do not use

**Dependency Inversion Principle :** High-level modules should not depend on low-level modules but on abstractions. This principle encourages the use of interfaces to reduce dependencies on concrete implementations, enhancing flexibility and decoupling components.

**Single Responsibility Principle:** A class should have only one reason to change.

**Open/Closed Principle:** Classes should be open for extension but closed for modification.

**Liskov Substitution Principle:** Subclasses should be substitutable for their base classes without altering the correctness of the program.

**Interface Segregation Principle:** Interfaces should be divided into smaller, more relevant parts to avoid unnecessary methods in implementing classes.

**Dependency Inversion Principle:** High-level modules should rely on abstractions rather than on low-level modules, promoting flexibility and decoupling.

**High Cohesion**

High cohesion refers to the principle that a class should have a single responsibility. All methods and instance variables of a class should be directly related to its specific purpose.

**Low Coupling**

Low coupling implies that classes should be as independent from one another. This means that changes in one class should not necessitate changes in another, or at least, the impacts should be minimal.

**Interfaces:** Using interfaces to define contracts improves low coupling, as classes can communicate through abstractions rather than concrete implementations.

**Dependency Inversion Principle:** This principle, which is fundamental for achieving low coupling, states that high-level classes should not depend on low-level classes, but on abstractions. This translates to using interfaces or abstract classes as reference types instead of concrete classes.

**High Cohesion:**  
A class should focus on a single responsibility, with all methods and variables closely aligned to its core purpose.

**Low Coupling:**  
Classes should maintain independence, minimizing the impact of changes in one class on others.  
Using interfaces helps achieve low coupling by defining contracts for class interactions via abstractions rather than concrete implementations.

**Core Java Collections**

* List: Maintains insertion order and permits duplicates. Examples: ArrayList (fast random access), LinkedList (efficient insertions/deletions).
* Set: Ensures unique elements. Variants include HashSet (fastest access), LinkedHashSet (maintains insertion order), and TreeSet (orders elements).
* Queue: Manages a sequence of elements for processing, supporting various insertion and removal operations. Types include PriorityQueue and ArrayDeque.
* Map: Pairs unique keys to values. Examples: HashMap (hash table based implementation), LinkedHashMap (maintains key insertion order), TreeMap (sorted map).

**List:**  
Maintains insertion order and allows duplicates.  
**ArrayList:** Offers fast random access.  
LinkedList: Efficient for insertions and deletions.

**Set:**  
Ensures all elements are unique.  
HashSet: Provides the fastest access.  
**LinkedHashSet:** Maintains insertion order.  
**TreeSet:** Orders elements naturally or by a comparator.

**Queue:**  
Manages a sequence of elements for processing.  
**PriorityQueue:** Elements are ordered by natural ordering or by a comparator.  
**ArrayDeque:** Efficient array-based queue supporting both ends.

**Map:**  
Associates unique keys with values.  
HashMap: Hash table-based implementation.  
**LinkedHashMap:** Maintains key insertion order.  
**TreeMap:** Keeps keys in sorted order.

**Checked exceptions**

Checked exceptions must be either declared in the method signature with a throws clause or managed within the method using a try-catch block. These exceptions, checked at compile-time, require explicit handling to ensure robust application behavior. Derived from the Exception class, they include a variety of types, such as IOException and its subclasses.

**IOException**

Primarily thrown to signal that an input or output operation has failed or been interrupted. Subclasses like FileNotFoundException and SocketException specify errors related to file access or network operations, respectively.

**InterruptedException**

This exception is thrown when a thread is interrupted while it is waiting, sleeping.

**Unchecked Exceptions**

Unchecked exceptions, also known as runtime exceptions, derive from RuntimeException, which is a subclass of Exception. These exceptions are not checked at compile time, meaning the compiler does not require methods to catch or declare them.

**NullPointerException**

Thrown when an application attempts to use null in a case where an object is required, such as calling a method on a null object reference.

**IndexOutOfBoundsException**

Thrown to indicate that an index of some sort (e.g., array, string, or vector) is out of range.

**ArithmeticException**

Thrown when an exceptional arithmetic condition has occurred, such as dividing by zero.

**ClassCastException**

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.

**Errors**

Represent serious issues like OutOfMemoryError that are not meant to be caught.

Checked Exceptions  
Must be declared or handled.  
Checked at compile-time.  
FileNotFoundException, InterruptedException

Unchecked Exceptions  
Derived from RuntimeException.  
Not checked at compile time; catching or declaring them is not mandatory.  
NullPointerException, IndexOutOfBoundsException, ArithmeticException, ClassCastException

Errors  
Not be caught because they are not intended to be recoverable.  
OutOfMemoryError

**Singleton Pattern**

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.  
Private Constructor: Prevents instantiation of the class from other classes.  
Private Static Instance: Holds the single instance of the class.  
Public Static Method: Provides a global access point to the instance and controls its creation.  
The attribute that holds the instance of the class is declared as static. Static members belong to the class rather than to any specific instance of the class.

Ensures a class has only one instance.  
Provides a global point of access to that instance.  
Private Constructor prevents instantiation from other classes.  
Controls the instantiation of the class, typically checking if an instance already exists and creating one if not.  
Static Attribute  
The instance of the class is stored in a static attribute, meaning it belongs to the class itself rather than any particular object instance.

**Creational Patterns**

Simplify object creation.

**Singleton Pattern**

Ensures that a class has only one instance.

**Factory Method Pattern**

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

**Abstract Factory Pattern**

Offers an interface for creating families of related or dependent objects without specifying their concrete classes.

**Builder Pattern**

Separates the construction of a object, allowing the same construction process to create various representations.

**Prototype Pattern**

Creates new objects by cloning an existing object.

**Behavioral Patterns**

Behavioral patterns focus on improving the interactions and responsibilities among objects.

**Observer Pattern**

This pattern establishes a one-to-many relationship between a subject and multiple observers. When the subject’s state changes, all registered observers are notified.

**Strategy Pattern**

Use different algorithms, each implemented through a method with the same interface. This makes it easy to switch behaviors dynamically.

**Structural Patterns**

Simplify relationships between objects.

**Adapter Pattern**

Allows objects with incompatible interfaces to collaborate by converting the interface into another interface that clients expect. This pattern lets you integrate classes that couldn’t otherwise work together due to incompatible interfaces.

**Decorator Pattern**

Enables adding new functionalities to objects dynamically by wrapping them with new features.

**Facade Pattern**

Provides a simplified interface to a complex subsystem. By introducing a facade, you reduce the complexity of interactions between the client and the subsystem, making the subsystem easier to use.

**Creational Patterns**  
Focus on simplifying object creation.

**Singleton Pattern:** Ensures only one instance of a class exists, providing a single point of access to it.

**Factory Method Pattern:** Defines an interface for creating objects but allows subclasses to change the type of objects that will be created.

**Abstract Factory Pattern:** Provides an interface for creating families of related objects without specifying their concrete classes.

**Builder Pattern:** Separates object construction.

**Prototype Pattern:** Creates new objects by copying an existing object.

**Behavioral Patterns**  
Enhance communication among objects.

**Observer Pattern**  
Forms a one-to-many dependency between a subject and its observers that are notified on state changes.

**Strategy Pattern:** Enables a set of algorithms selected at runtime.

**Structural Patterns**  
Simplify relationships between objects.

**Adapter Pattern:** Allows classes with incompatible interfaces to work together by wrapping one interface around another.

**Decorator Pattern:** Adds new functionalities to objects dynamically.

**Facade Pattern:** Provides a simple interface to a complex system.

**Immutable Class**

Creates classes whose instances cannot be modified once constructed, ensuring thread safety.  
An immutable class is designed so that its instances cannot be modified after they are created. This means all of their fields must be final, and the class itself is often declared as final to prevent subclassing.  
Immutable classes do not provide “setter” methods.  
Initialization in Constructor: All fields of an immutable object are typically initialized in the constructor. External modification after construction is prohibited.  
Immutable objects are naturally thread-safe because their state cannot change after construction.

**String**

Represents immutable sequences of characters.

**Wrapper Classes**

All primitive wrapper classes such as Integer, Long, Double, Float, Short, Byte, Character, and Boolean are immutable.

BigDecimal

Involves numbers requiring precision.

LocalDate

Represents a date without time or timezone (e.g., 2019–12–31).

LocalTime

Represents a time without date or timezone (e.g., 10:15).

LocalDateTime

Combines date and time, but lacks timezone (e.g., 2019–12–31T10:15).

All fields are declared as final to prevent their values from being modified after initialization.  
The class itself is often declared as final to prevent it from being subclassed.  
Immutable classes do not have setter methods.  
All fields are initialized in the constructor.  
Immutable objects are thread-safe.

**Functional Interface**

An interface with exactly one abstract method. It can have multiple default or static methods.  
Designed to be used with lambda expressions in Java.  
Runnable: Used in threading contexts, with no arguments and no return value (void run()).  
Callable: Similar to Runnable but returns a value and can throw a checked exception (V call()).  
Comparator: Defines a comparison function for sorting that returns a negative integer, zero, or a positive integer (int compare(T o1, T o2)).  
Consumer: Performs an action on the given argument, returning no result (void accept(T t)).  
Supplier: Returns a result and does not take any input (T get()).  
Predicate: Returns a boolean value based on its input, used for evaluating conditions (boolean test(T t)).  
Function: Accepts one argument and produces a result, typical use case is for converting or transforming data (R apply(T t)).  
UnaryOperator: A specialization of Function where the input and output are of the same type (T apply(T t)).  
BinaryOperator: A specialization of Function for two inputs of the same type, producing a result of the same type (T apply(T t1, T t2)).

@FunctionalInterface Annotation: Optional but helps in making the intent clear and ensures the interface cannot have more than one abstract method.

**Lambda Expression**

Represent an anonymous function that can be passed as argument or stored in a variable.  
Parameters, an arrow ->, and a body. Example: (x, y) -> x + y.  
Provide implementation of functional interfaces. Use for functional programming concepts in Java.

**Streams**

A sequence of elements supporting sequential and parallel aggregate operations.  
Created from collections, lists, sets, ints, longs, doubles, arrays, or lines of a file.  
Divided into intermediate and terminal operations.

**Intermediate Operations**

Transform the stream into another stream. These operations are lazy.

**Terminal Operations**

Produce a result from the stream once the operations have been applied. They are eager, executing all the intermediate operations and processing the data.

* forEach: Performs an action for each element of the stream.
* collect: Gathers elements into a summary result, such as a list or a map.
* reduce: Combines elements sequentially and returns an optional reduced value.
* anyMatch, allMatch, noneMatch: Return a boolean based on predicates applied to stream elements.
* count: Returns the count of elements in the stream.

**Spring Boot**

Simplified Configuration

Simplify the setup and configuration.

Auto-Configuration

Automatically sets up application configuration based on included dependencies. For example, it configures Spring MVC automatically if the Spring Web Starter dependency is included in the project.

Embedded Servers

Spring Boot applications can run as standalone applications using an embedded web server (like Tomcat), eliminating the need for deploying to an external web server.

Self-contained

Enables the packaging of applications into single executable JARs (or WARs), which includes the embedded server and other necessary dependencies.

Simplified Configuration  
Reduces the complexity of setup and configuration.

Embedded Servers  
Allows Spring Boot applications to run independently using an embedded web server like Tomcat.

Self-contained  
Facilitates the creation of executable JARs or WARs that contain all necessary dependencies including the embedded server, enabling easy distribution and deployment.

Microservices Architecture with Java and Spring Boot

Microservices architecture breaks down a single application into a suite of small, independent services, each running in its own process.  
These services interact through RESTful APIs using lightweight HTTP methods.  
Each microservice is decoupled from others, which enhances the overall resilience and reliability of the application.  
Every microservice manages its own database, which helps in isolating the service data and improving the fault tolerance. This separation ensures that the failure of a single service does not affect the availability of the entire system.  
Services can be scaled independently based on their specific load and performance requirements, enabling more efficient resource use.

REST API and HTTP Verbs

Services use RESTful interfaces, which employ HTTP verbs such as GET, POST, PUT, and DELETE to manage resources. This approach allows clear, standard operations including retrieving data (GET), creating new entries (POST), updating existing entries (PUT), and deleting entries (DELETE).

JSON for Data Exchange

JSON is commonly used for request and response formats in REST APIs, providing a lightweight, human-readable, and easily parsable structure for data interchange between services.

Breaks down an application into a suite of small, independent services, each running in its own process.  
Each microservice manages its own database.  
Services are independently scalable based on their specific performance and load requirements, allowing for optimized resource usage.  
Utilizes RESTful interfaces employing HTTP verbs such as GET, POST, PUT, and DELETE for standard operations on resources, facilitating clear and standardized interactions.  
Employs JSON for sending and receiving data in REST APIs, benefiting from its easily parsable format for effective data interchange between services.

Resilience patterns

These patterns are essential in microservices architectures, where the failure of a single component can potentially affect the entire system.

Resilience patterns are implemented to ensure that a system remains responsive and available even under adverse conditions. This includes handling and recovering from errors, managing system overloads, and maintaining service during partial system failures.

Preventing the failure of a single component from impacting the entire system.  
Ensure that systems remain responsive and available, even under adverse conditions.  
Includes strategies for recovering from errors.

Circuit Breaker

Prevents cascading failures in a distributed system by managing calls to a service that is experiencing issues.

States

Closed: The default state where requests to the service are allowed. The circuit breaker monitors for failures and will trip if errors exceed a predetermined threshold.

Open: Once the failure threshold is reached, the circuit breaker trips to the open state, stopping all attempts to access the service for a set recovery period.

Half-Open: After the recovery period, the circuit breaker enters a half-open state, where a limited number of test requests are allowed to pass through. If these requests succeed, the circuit breaker resets to the closed state; if not, it returns to open.

Protection Against Slow Responses

The circuit breaker can detect and abort calls to a microservice that is responding slowly.

Handling Errors

A microservice that is being called might be returning errors such as internal server errors and the circuit breaker implement a fallback strategies to maintain functionality. This can include:  
Custom Error Messages: Returning predefined messages to users, informing them of the issue while maintaining a good user experience.  
Alternative Service Calls: Automatically routing requests to a backup service that can handle the request.

Bulkhead

This pattern limits the amount of concurrent calls to a microservices, protecting it from being exhausted.

Retry

Enables a service to retry a failed call.

Rate Limiter

Regulates the calls to service. This is crucial to prevent overwhelming a service with too many requests at once.

TimeLimiter

Allows to set a time limit on a service call. If the method execution exceeds the specified time limit, it is automatically aborted.

Cache

Store the result of a service call for a specified time. Caching reduce the number of calls to a service.

Circuit Breaker  
Prevents cascading failures by managing calls to services experiencing issues.  
Closed: Default state where requests are allowed; trips if errors exceed a threshold.  
Open: Stops all access to the service during a recovery period.  
Half-Open: Tests service availability with limited requests post-recovery; resets to Closed if successful, returns to Open if not.  
Protection Against Slow Responses: Detects and aborts calls to slow-responding microservices.  
Handling Errors: Implements fallback strategies for services returning errors, like internal server errors. Strategies include:  
Custom Error Messages: Sends predefined messages to users to maintain a good experience.  
Alternative Service Calls: Routes requests to a backup service capable of handling them.

Bulkhead  
Limits the number of concurrent calls to a service, protecting it from becoming overwhelmed.

Retry  
Allows services to automatically retry failed calls.

Rate Limiter  
Controls the rate of calls to a service to prevent overloading it.

TimeLimiter  
Sets a maximum time limit for a service call; aborts the call if this limit is exceeded, helping to manage resources efficiently.

Cache  
Stores results of service calls for a designated period, reducing the need for repeated calls and decreasing load on the service.

SAGA pattern

The SAGA pattern is used to manage data consistency across multiple microservices. It can be implemented in two main ways:

Event/Choreography-based SAGA

Each service performs its local transaction and publishes an event . Other services listen to these events and act accordingly, without a central coordinator. This method relies on events to trigger and handle the next steps in the business process, making it a decentralized decision-making approach.

Command/Orchestration-based SAGA

Uses a central coordinator, called an orchestrator, which is responsible for directing the saga’s execution across all involved services. The orchestrator sends commands to each participating service to perform local transactions and awaits their responses. Based on the success or failure of each transaction, the orchestrator decides the next steps, which might include triggering compensating transactions in the case of failures.

Used to manage data consistency across multiple microservices.  
Helps handle transactions in distributed systems where each service operates independently.

Two Main Implementations:

Event/Choreography-based SAGA:  
Each service performs its local transaction and publishes an event.  
Other services listen to these events and act accordingly, without a central coordinator.  
Relies on events to trigger and manage the next steps in the business process.  
Decentralized decision-making; each service independently handles part of the process.

Command/Orchestration-based SAGA:  
Uses a central coordinator, often referred to as an orchestrator.  
The orchestrator directs the saga’s execution across all involved services.  
Sends commands to services to perform transactions and waits for responses.  
Decides next steps based on the success or failure of transactions, including potentially triggering compensating transactions.  
Centralized decision-making; the orchestrator manages the overall process and error handling.

Integration of Docker and Kubernetes

Docker

Docker allows developers to package an application with all of its dependencies into a standardized unit for software development. Docker containers are:

* Portable: Containers can run on any system that supports Docker’s container runtime environment.
* Isolated: Docker ensures that applications use only the resources allocated to them.

Example Dockerfile

# Step 1: Use an official OpenJDK runtime as a parent image  
FROM openjdk:11-jdk-slim  
  
# Step 2: Set the working directory inside the container  
WORKDIR /app  
  
# Step 3: Copy the JAR file into the container at /app  
COPY target/my-spring-boot-app.jar /app/spring-app.jar  
  
# Step 4: Expose port 8080 for the application  
EXPOSE 8080  
  
# Step 5: Define the command to run the application  
CMD ["java", "-jar", "/app/spring-app.jar"]

Kubernetes

Kubernetes is an orchestration system for Docker containers that facilitates the deployment and scaling of applications. Key functionalities include:

* Load Balancing: Kubernetes can distribute network traffic so that the cluster does not get overloaded.
* Automated Rollouts and Rollbacks: Kubernetes allows you to describe the desired state for your deployed containers using declarative configuration files, and can change the actual state to the desired state at a controlled rate.