Basic Java Interview Questions

---

1. What is ?

Java is a high-level, object-oriented, platform-independent programming language. Key features:

- **Platform-independent**: Code can run on different platforms without modification using JVM.
- **High performance**: JIT compiler converts bytecode to machine code for faster execution.
- **Multi-threaded**: Supports concurrent task handling.
- **No multiple inheritance through classes**: Achieved using interfaces.
- **Security**: No explicit pointers; strong memory management with garbage collection.

---

2. OOP Concepts

- **Inheritance**: Code reuse between superclasses and subclasses; private members are not inherited.
- **Encapsulation**: Protects code by making variables private and using getter/setter methods.
- **Polymorphism**: Objects behave differently based on their type (method overriding).
- **Abstraction**: Hides implementation details while exposing essential features.
- **Interface**: Provides multiple inheritance by defining method declarations.

---

3. Access Modifiers

- **Default**: Accessible within the same package.
- **Private**: Accessible only within the declaring class.
- **Protected**: Accessible in the same package or subclasses in other packages.
- **Public**: Accessible everywhere.

---

4. SOLID Principles

- **Single Responsibility**: Each class has one responsibility.
- **Open-Closed**: Classes should be open to extension but closed to modification.
- **Liskov Substitution**: Subtypes must be substitutable for their base types.

- **Interface Segregation**: Avoid forcing clients to implement unused methods.
- **Dependency Inversion**: High-level modules depend on abstractions, not concrete implementations.

---

5. Primitive Data Types

- **byte** (1 byte), **short** (2 bytes), **char** (2 bytes), **int** (4 bytes), **long** (8 bytes)
- **float** (4 bytes), **double** (8 bytes), **boolean** (1 byte)

---

6. Order of Initialization

1. **Static Blocks**: Run once when the class is loaded.
2. **Instance Initialization Blocks**: Execute before constructors.
3. **Constructors**: Initialize objects.

---

7. Lambda Expressions vs. Closures

- **Lambda Expressions**: Anonymous functions that treat actions as objects; can't access variables outside their scope unless final or effectively final.
- **Closures**: Not directly supported in ; allow referencing variables outside the parameter list.

---

8. Object Class Methods

- Examples: clone(), equals(), toString(), hashCode(), notify(), wait(), getClass()

---

9. Atomic, Volatile, Synchronized

- **Volatile**: Ensures visibility of variable updates across threads.
- **AtomicInteger**: Provides thread-safe operations without synchronization using CAS.
- **Synchronized**: Allows only one thread at a time to access a block or method.

---

10. Serialization

- Converts objects to byte streams for persistence or network transfer.

- Implement Serializable interface for default serialization.
- Customize using readObject and writeObject.
- Use transient or static for variables not to be serialized.

---

11. Cloning

- **Shallow Copy**: Only copies primitive fields; object references point to the same memory.
- **Deep Copy**: Creates new objects for all references, duplicating all data.

---

12. StringBuffer vs. StringBuilder

- **StringBuffer**: Thread-safe, slower.
- **StringBuilder**: Non-thread-safe, faster.

---

13. XML Parsers

- **DOM**: Tree-based, memory-intensive.
- **SAX**: Event-driven, forward-only, low memory.
- **StAX**: Pull-based, allows parsing and modification.

---

14. New Features in

- **8**: Lambda expressions, Stream API, Optional, Functional Interfaces, Date API.
- **11**: isBlank(), lines(), strip(), HTTP client updates, readString(), writeString().

---

15. Lambda Expressions

Used for inline implementation of functional interfaces, improving code readability and compactness.

---

16. Exceptions

Abnormal events disrupting program flow, managed using try-catch or throws.

If you need further explanation on any specific topic, let me know!

Collections Interview Questions

---

1. Collections Type Hierarchy

The Collections Framework is structured around several core interfaces:

- **Iterable**: The root interface enabling iteration over elements (e.g., for-each loop).
- **Collection**: Extends Iterable and defines methods like add, remove, size, and more.
    - **List**: Ordered, index-based collections (e.g., ArrayList, LinkedList).
    - **Set**: Unordered collections with unique elements (e.g., HashSet, TreeSet).
    - **Queue**: Supports element ordering for processing (e.g., PriorityQueue).
- **Map**: Key-value pairs, distinct from Collection, emphasizing mapping rather than collections.

Each interface serves a unique purpose, ensuring flexibility and scalability in designing applications.

---

2. Map Implementations and Use Cases

- **HashMap**: Best for general-purpose use where order is not a concern. O(1) time complexity for operations.
- **LinkedHashMap**: Retains insertion order, ideal for access-order-sensitive applications (e.g., LRU caches).
- **TreeMap**: Maintains keys in sorted order. Suitable for range queries or ordered data (logarithmic access time).
- **ConcurrentHashMap**: Optimized for concurrent access, avoiding global locks for better performance in multi-threaded environments.
- **Hashtable**: Synchronized but outdated; typically replaced by ConcurrentHashMap.

---

3. LinkedList vs. ArrayList

- **ArrayList**:
    - Backed by a resizable array.
    - Fast random access (O(1)).
    - Costly insertions/removals in the middle (O(n)) due to shifting elements.
- **LinkedList**:
    - Uses a doubly-linked list.
    - Efficient insertions/removals (O(1)).
    - Slower traversal (O(n)).
    - Higher memory overhead due to node references.

Use Case: Use ArrayList for frequent access; prefer LinkedList for heavy insertions/deletions.

---

4. HashSet vs. TreeSet

- **HashSet**:
  - Backed by HashMap.
  - No order guarantee.
  - Fast operations (O(1)).
- **TreeSet**:
  - Backed by TreeMap.
  - Maintains natural or custom order.
  - Slower operations (O(log n)).

Use Case: Use HashSet for performance; prefer TreeSet for ordered sets.

---

5. HashMap Implementation

A HashMap employs:

- **Hashing**: Calculates the bucket index from the hash code of keys.
- **Buckets**: Hold key-value pairs. Collisions are handled using linked lists or red-black trees (since 8).
- **Resizing**: Automatically doubles in size when the load factor exceeds the threshold.

Key operations like put and get are optimized for O(1), but excessive collisions can degrade performance to O(log n).

---

6. Initial Capacity and Load Factor

- **Initial Capacity**: Starting size of the internal array (rounded to the nearest power of 2).
- **Load Factor**: Threshold (default 0.75) for resizing when the ratio of elements to buckets is exceeded.

Optimizing these parameters minimizes resizing overhead.

---

7. Special Collections for Enums

- **EnumSet**: Extremely efficient Set for enums using bitwise operations.
- **EnumMap**: Uses enums as keys and arrays for storage, providing fast lookups without hashing.

---

8. Fail-Fast vs. Fail-Safe Iterators

- **Fail-Fast**:
  - Detects concurrent modifications and throws ConcurrentModificationException.
  - Found in collections like ArrayList, HashMap.
- **Fail-Safe**:
  - Operates on a copy of the collection.
  - Found in CopyOnWriteArrayList, ConcurrentHashMap.

---

9. Comparable vs. Comparator

- **Comparable**:
  - Natural ordering defined by compareTo method.
  - Example: Sorting integers in ascending order.
- **Comparator**:
  - Custom ordering via compare method.
  - Example: Sorting strings by length.

Both interfaces are pivotal for sorting collections using methods like Collections.sort or for maintaining order in TreeSet/TreeMap.

Concurrency Interview Questions

Creating and Running a Thread Instance:

Using `Runnable`:

```
Thread thread1 = new Thread(() -> System.out.println("Hello from
Runnable!"));
thread1.start();
```

1.

Using a Subclass of `Thread`:

```
Thread thread2 = new Thread() {
    @Override
```

```
    public void run() {
        System.out.println("Hello from subclass!");
    }
};
thread2.start();
```

2.

Thread States and Transitions:

1. **NEW**: Thread created but not started (`Thread t = new Thread()`).
2. **RUNNABLE**: After calling `start()`, thread is ready for execution.
3. **BLOCKED**: Thread waiting to enter a synchronized block.
4. **WAITING**: Thread waiting indefinitely for another thread's signal.
5. **TIMED_WAITING**: Thread waiting for a specified duration.
6. **TERMINATED**: Thread completes execution.

Runnable vs Callable:

- **Runnable**: Does not return a result or throw checked exceptions.

Callable: Returns a result and can throw checked exceptions. Example:

```
Callable<Integer> callable = () -> 42;
```

-

Daemon Threads:

- **Definition**: Threads supporting non-daemon threads. JVM exits once all non-daemon threads finish.

Creation:

```
Thread daemon = new Thread(() -> System.out.println("Daemon
thread!"));
daemon.setDaemon(true);
daemon.start();
```

-

Interrupt Flag and Exceptions:

- **Set Interrupt**: `thread.interrupt()`.
- **Check Interrupt**: `Thread.interrupted()` (clears flag) or `isInterrupted()` (doesn't clear flag).
- **Behavior**: Throws `InterruptedException` if in blocking operation like `sleep()`.

Executor vs ExecutorService:

- **Executor**: Simple interface for executing `Runnable` tasks.
- **ExecutorService**: Adds lifecycle management, task tracking, and advanced execution controls.

ExecutorService Implementations:

1. **ThreadPoolExecutor**: Manages a pool of threads.
2. **ScheduledThreadPoolExecutor**: Schedules tasks with delays or periodic execution.
3. **ForkJoinPool**: Efficient for divide-and-conquer tasks using work-stealing.

Memory Model (JMM):

- Defines how threads interact through shared memory.
- Ensures predictable memory visibility and ordering for concurrent threads.
- Key Concepts: Actions, Synchronization actions, Happens-before relationships, and Happens-before consistency.

Volatile Fields in JMM:

- **Guarantees**:
    - Visibility: Changes to `volatile` variables are immediately visible to other threads.
    - Atomicity: Single read/write of `long` or `double` variables.

Atomicity of Operations:

- **Atomic**: Writes to `int` or `volatile long`.
- **Non-atomic**: Incrementing a `volatile long`.

Final Field Guarantees:

- `final` ensures proper initialization visibility across threads.

Synchronized Keyword:

- Synchronizes access using:
  - **Instance methods**: Lock on the object.
  - **Static methods**: Lock on the class object.
  - **Block**: Lock on a specified object.

Static vs Instance Synchronized Methods:

- **Instance methods**: Threads accessing different instances don't block.
- **Static methods**: Threads share a single class-level lock.

`wait()`, `notify()`, `notifyAll()`:

- **Purpose**: For thread coordination in synchronized blocks.
- **Behavior**:
  - `wait()`: Releases lock and waits for notification.
  - `notify()`: Wakes a single waiting thread.
  - `notifyAll()`: Wakes all waiting threads.

Java Garbage Collection Questions

What is Garbage Collection?

Garbage collection (GC) is the process of automatically reclaiming memory by identifying and removing objects no longer referenced by the application, thus preventing memory leaks.

Which Part of the Memory is Involved in Garbage Collection?

Garbage collection occurs in the **Heap Memory**, specifically in the **Young Generation** and **Old Generation**.

Minor vs Major Garbage Collection

1. **Minor GC**:
    - Cleans the Young Generation (Eden and Survivor spaces).
    - Occurs more frequently and is faster.
    - Triggers when the Eden space is full.
2. **Major GC**:
    - Cleans the Old Generation.
    - Occurs less frequently and is more time-consuming.
    - Triggers when the Old Generation is full or at JVM-defined thresholds.

Algorithm for Garbage Collection in Java

1. **Mark-and-Sweep**:
    - **Mark Phase**: Identifies reachable objects.
    - **Sweep Phase**: Reclaims memory occupied by unreachable objects.
2. **Generational GC**:
    - Objects are categorized into generations (Young and Old).
    - Different strategies are applied to each generation.
3. **Stop-the-World**:
    - All application threads pause during garbage collection.

4. **Compact Phase (Optional)**:
    - Fragmented memory is consolidated for efficiency.

`finalize()` Method in Java

- **Purpose**: Provides a way to perform cleanup operations before an object is reclaimed by the garbage collector.
- **When Called**: The garbage collector invokes `finalize()` if it detects that the object is eligible for GC and overrides the method.
- **Issues**:
    - No guarantee that `finalize()` will be called.
    - Can cause performance issues and unpredictable behavior.
    - Deprecated as of Java 9.

Making an Object Eligible for Garbage Collection

Nullify References: Set object references to `null`.
obj = null;

1.

Reassign References: Point the reference to another object.
obj1 = obj2;

2.

Scope Limitation: Let the object go out of scope.
// Object declared inside a method is no longer accessible outside it.

3.

Ways to Call Garbage Collector

1. **Explicitly Suggest**:
    - Using `System.gc()` or `Runtime.getRuntime().gc()`.
    - No guarantee that GC will run immediately.
2. **JVM Triggered**:
    - Happens automatically based on JVM's memory thresholds and requirements.

Heap vs Stack Memory in Java

Aspect	Heap Memory	Stack Memory
Purpose	Stores objects and JRE classes.	Stores method calls, local variables, and references.
Allocation	Dynamic (at runtime).	Static (based on the method lifecycle).
Lifetime	Exists throughout the application lifecycle.	Exists only during method execution.
Thread Scope	Shared across threads.	Thread-local (each thread has its stack).
Speed	Slower due to global access and dynamic allocation.	Faster because of simpler memory access patterns.
Memory Management	Managed by Garbage Collector.	Managed automatically via method calls.

Parts of the Heap Memory

1. **Young Generation**:
    - Divided into **Eden Space** and **Survivor Spaces**.
    - New objects are allocated here, and minor garbage collection occurs to remove short-lived objects.
2. **Old Generation (Tenured Space)**:
    - Stores long-lived objects.
    - Major garbage collection happens here.
3. **Metaspace (Java 8 and later)**:
    - Stores metadata about classes, methods, and other reflective data.
4. **Permanent Generation (Java 7 and earlier)**:
    - Similar to Metaspace but had a fixed size and was part of the heap.

PermGen vs Metaspace

Aspect	PermGen (Java 7 and earlier)	Metaspace (Java 8 and later)
Purpose	Stores class metadata and reflective data.	Also stores class metadata, replacing PermGen.

Memory Limit	Fixed size, configurable via JVM options.	Dynamically resized; grows as needed.
Garbage Collection	Part of the heap, managed by GC.	Managed by GC, but part of native memory.

For More Click

Java Annotation Interview Questions

---

1. What are Meta-Annotations, and What is the Purpose of Each in the `java.lang.annotation` Package?

- **`@Retention`:** Defines the scope of the annotation.
  - **Policies:**
    - `SOURCE`: Annotation is discarded during compilation.
    - `CLASS`: Annotation is in the class file but not available at runtime.
    - `RUNTIME`: Annotation is retained in the JVM and available via reflection.
- **`@Target`:** Specifies the kinds of elements an annotation can apply to (e.g., `METHOD`, `FIELD`).
- **`@Inherited`:** Allows subclasses to inherit annotations from their parent classes.
- **`@Documented`:** Indicates that the annotation should appear in the generated Javadoc.
- **`@Repeatable`:** Allows multiple instances of the same annotation on an element.

---

2. How Do You Create a Custom Annotation?

Example:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
   String description() default "No description";
   int priority() default 0;
   Class<? extends Throwable> expectedExceptions() default None.class;
}
```

---

3. Explain the Difference Between Runtime and Compile-Time Annotations. How Are They Processed?

- **Compile-Time Annotations:**
  - Examples: `@Override`, `@Deprecated`.
  - Processed during compilation.
  - Ensures correctness and compliance with Java conventions.
- **Runtime Annotations:**
  - Examples: Custom annotations with `@Retention(RUNTIME)`.
  - Accessed using reflection during runtime.
  - Commonly used in frameworks like Spring and Hibernate.

---

4. How Can You Implement a Custom Annotation Processor?

Example Implementation:

```
@SupportedAnnotationTypes("com.example.MyCustomAnnotation")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class MyAnnotationProcessor extends AbstractProcessor {
    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
        // Custom processing logic
        return true;
    }
}
```

---

5. Describe Advanced Use Cases of Java Reflection with Annotations.

- **Dynamic Method Invocation:** Execute methods based on annotations.
- **Runtime Configurations:** Modify behavior dynamically using annotations.
- **Dependency Injection:** Frameworks like Spring use annotations for DI.
- **Validation Frameworks:** Build field-level validators with annotations.
- **Test Frameworks:** Custom test runners and assertions.

---

6. What Are the Limitations of Annotations, and How Can They Be Overcome?

- **Limitations:**
    - Cannot contain method implementations.
    - Limited to specific element types.
    - May introduce runtime overhead.
- **Strategies:**
    - Combine annotations with reflective or bytecode-based tools.
    - Design minimalistic annotations to reduce complexity.

---

7. Compare and Contrast Annotations with Marker Interfaces.

- **Annotations:**
    - Flexible metadata, no hierarchy constraints.
    - Better suited for declarative programming.
- **Marker Interfaces:**
    - Used for type-checking and enforceable contracts.
    - Limited flexibility compared to annotations.

---

8. How Would You Implement a Custom Validation Framework Using Annotations?

Example:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Validate {
    boolean notNull() default false;
    int minLength() default 0;
    int maxLength() default Integer.MAX_VALUE;
}

public class ValidationProcessor {
    public boolean validate(Object object) {
        // Implement reflection-based validation logic
        return true;
    }
}
```

---

9. How Do Frameworks Like Spring Utilize Annotations for Dependency Injection and Configuration?

- **Annotations:**
    - `@Autowired`: For injecting dependencies.
    - `@Component`: Marks a class as a Spring bean.
    - `@Configuration`: Indicates a configuration class.
    - `@Bean`: Defines beans in configuration classes.
- **Mechanism:**
    - Frameworks scan annotations at runtime or compile-time to perform injections and configurations.

---

10. Demonstrate a Complex Use Case of Meta-Programming Using Annotations.

- **Scenario: Create a Custom ORM (Object-Relational Mapping). Example:**

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Column {
    String name();
    boolean primaryKey() default false;
}

public class ORMProcessor {
```

```
    public void mapObjectToDB(Object entity) {
        // Use reflection to map fields with @Column to database columns
    }
}
```

Java Dependency Injection (DI) Interview Questions

---

1. What is Dependency Injection (DI), and Why is it Important?

- **Definition:** Dependency Injection is a design pattern in which an object's dependencies are provided externally rather than created internally by the object.
- **Purpose:**
  - Promotes **loose coupling** between objects.
  - Makes code easier to maintain, test, and extend.
- **Problems Solved:**
  - Avoids hard-coding dependencies.
  - Facilitates unit testing by allowing the use of mock dependencies.
- **Comparison:** Traditional object creation couples a class with its dependencies, whereas DI separates their lifecycle.

---

2. Explain the Different Types of Dependency Injection

Constructor Injection:

- Dependencies are injected through the constructor.
- Ensures dependencies are immutable and required for the object to function.

```
public class UserService {
    private final UserRepository repository;

    public UserService(UserRepository repository) {
        this.repository = repository;
    }
}
```

Setter Injection:

- Dependencies are set through setter methods.
- Allows optional or late dependency injection.

```
public class UserService {
    private UserRepository repository;

    public void setUserRepository(UserRepository repository) {
        this.repository = repository;
    }
}
```

Interface Injection:

- Dependencies are provided through a custom method defined in an interface.
- Rarely used compared to constructor or setter injection.

```
public interface RepositoryInjector {
   void injectRepository(UserService service);
}
```

---

3. Advanced DI Concepts

- **Inversion of Control (IoC):**
  - Delegates the responsibility of dependency management to a container.
- **Dependency Inversion Principle:**
  - High-level modules should not depend on low-level modules; both should depend on abstractions.
- **Service Locator Pattern vs Dependency Injection:**
  - **Service Locator:** Centralized access to dependencies.
  - **DI:** Dependencies are pushed directly into objects.
- **Pros/Cons of Injection Methods:**
  - Constructor: Mandatory dependencies, immutability.
  - Setter: Optional dependencies, flexibility.

---

4. Spring Framework DI Questions

Annotations:

```
@Component
public class UserService {
   @Autowired
   private UserRepository repository;
}
```

- **`@Autowired`:** Injects dependencies automatically.
- **`@Component`, `@Service`, `@Repository`:** Define Spring-managed beans.
- **`@Scope`:** Defines bean scope (`singleton`, `prototype`, etc.).
- **`@Qualifier`:** Resolves ambiguity when multiple beans of the same type exist.

---

5. Implementing a Custom Dependency Injection Container

Example Implementation:

```
public class DIContainer {
    private Map<Class<?>, Object> instances = new HashMap<>();

    public <T> void register(Class<T> type, T instance) {
        instances.put(type, instance);
    }

    public <T> T resolve(Class<T> type) {
        return (T) instances.get(type);
    }
}
```

---

6. Complex Dependency Injection Scenarios

- **Circular Dependencies:**
  - A depends on B, and B depends on A.
  - Solved using `@Lazy` initialization or refactoring.
- **Handling Optional Dependencies:**
  - Use `Optional<T>` or setter injection.
- **Lazy Initialization:**
  - Delays the creation of dependencies until needed.
- **Prototype vs Singleton Scopes:**
  - Prototype: New instance per injection.
  - Singleton: Single shared instance.

---

7. Performance and Best Practices

- **Constructor Injection Preferred:**
  - Ensures all dependencies are available at object creation.

```
@Service
public class UserService {
    private final UserRepository repository;

    @Autowired
    public UserService(UserRepository repository) {
        this.repository = repository;
    }
}
```

- **Best Practices:**
  - Avoid too many dependencies.
  - Use dependency graphs judiciously.

- Properly scope beans to avoid memory leaks.

---

8. Advanced DI with Generics

Example:

```
public interface GenericRepository<T> {
    void save(T entity);
}

@Service
public class GenericService<T> {
    private final GenericRepository<T> repository;

    @Autowired
    public GenericService(GenericRepository<T> repository) {
        this.repository = repository;
    }
}
```

---

9. Testing with Dependency Injection

- Use mocking frameworks like Mockito for dependency injection during tests.

```
public class UserServiceTest {
    @Mock
    private UserRepository mockRepository;

    @InjectMocks
    private UserService userService;

    @Test
    public void testUserCreation() {
        // Test logic here
    }
}
```

- **Strategies:**
  - Mock dependencies for isolated unit testing.
  - Test integration with actual dependencies for end-to-end testing.

---

10. Java EE and Jakarta EE DI Annotations

Annotations:

```
@Named
@ApplicationScoped
public class UserService {
    @Inject
    private UserRepository repository;
}
```

- **CDI (Context and Dependency Injection):**
    - `@Inject`: Standard for injecting dependencies.
    - `@Named`: Binds a name to a bean.
    - Scope annotations like `@ApplicationScoped`, `@RequestScoped`.

Patterns Interview Questions

Creational Patterns

Creational Design Patterns are focused on the process of object creation. They aim to reduce complexity and instability by creating objects in a controlled manner.

Singleton Design Pattern

The **Singleton Design Pattern** ensures that only one instance of a class exists throughout the Virtual Machine (JVM) and provides a global access point to that instance.

- **Lazy Initialization with Static Inner Class:**

```
public class Singleton {
    private Singleton() {}

    private static class SingletonHolder {
        public static final Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }
}
```

- **Eager Initialization (Not Lazy):**

```
public class Singleton {

    public static final Singleton INSTANCE = new Singleton();


    private Singleton() {}

}
```

Problems with Serialization and Deserialization

If you serialize a Singleton class, it can create new instances during deserialization, violating the Singleton property. Implementing the `Serializable` interface alone won't solve this.

Problems with Reflection

Advanced users can break the Singleton by using reflection to change the access modifier of the constructor at runtime, allowing for multiple instances of the Singleton.

Singleton with Enum

Enums provide a much more robust solution. Since enums are inherently serializable, they avoid the serialization and reflection issues. Here's how you can implement a Singleton with an enum:

```
public enum SingletonEnum {

    INSTANCE;


    private int value;
```

```
    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Usage:

```
public class EnumDemo {
    public static void main(String[] args) {
        SingletonEnum singleton = SingletonEnum.INSTANCE;
        System.out.println(singleton.getValue());
        singleton.setValue(2);
        System.out.println(singleton.getValue());
    }
}
```

When to Use Singleton Design Pattern

- For expensive resources like database connections.

- To optimize performance (e.g., loggers).
- For configuration settings in an application.
- For resources accessed in shared mode.

Factory Design Pattern

The **Factory Design Pattern** defines an interface for creating objects, but lets subclasses decide which class to instantiate. This pattern is often used to create objects based on input, such as the number of sides of a polygon.

- **Example:**

```
public interface Polygon {
    String getType();
}

public class PolygonFactory {
    public Polygon getPolygon(int numberOfSides) {
        switch (numberOfSides) {
            case 3: return new Triangle();
            case 4: return new Square();
            case 5: return new Pentagon();
            case 7: return new Heptagon();
            case 8: return new Octagon();
            default: return null;
        }
    }
```

```
}
```

When to Use Factory Design Pattern

- When implementations of an interface or abstract class change frequently.
- When the current implementation cannot easily accommodate changes.
- When the initialization process is simple and requires only a few parameters.

Abstract Factory Design Pattern

The **Abstract Factory Design Pattern** is used to create families of related or dependent objects. It's essentially a factory of factories. This is useful when dealing with complex objects that belong to a specific family and need to be created together.

Builder Design Pattern

The **Builder Design Pattern** is used when the creation of an object becomes too complex. It separates the construction of an object from its representation.

- **Example:**

```
public class BankAccount {
    private String name;
    private String accountNumber;
    private String email;
    private boolean newsletter;

    public static class BankAccountBuilder {
        private String name;
```

```
        private String accountNumber;
        private String email;
        private boolean newsletter;

        public BankAccountBuilder(String name, String accountNumber)
{
            this.name = name;
            this.accountNumber = accountNumber;
        }

        public BankAccountBuilder withEmail(String email) {
            this.email = email;
            return this;
        }

        public BankAccountBuilder wantNewsletter(boolean newsletter)
{
            this.newsletter = newsletter;
            return this;
        }

        public BankAccount build() {
            return new BankAccount(this);
        }
    }
```

```
}
```

Usage:

```
BankAccount newAccount = new BankAccount
    .BankAccountBuilder("Jon", "22738022275")
    .withEmail("jon@example.com")
    .wantNewsletter(true)
    .build();
```

When to Use Builder Pattern

- When the object creation process is complex and has many mandatory and optional parameters.
- When constructors with many parameters lead to a large number of constructors.
- When clients expect different representations of the object.

Structural Patterns

Structural Design Patterns deal with the composition of classes and objects, focusing on how objects are composed to form larger structures.

Proxy Pattern

The **Proxy Pattern** creates an intermediary object that acts as an interface to another resource, such as a file or database connection. It allows for controlled access and protects the original object from complexity.

- **Example:**

```
public interface ExpensiveObject {
    void process();
}
```

```
public class ExpensiveObjectImpl implements ExpensiveObject {
    public ExpensiveObjectImpl() {
        heavyInitialConfiguration();
    }
    @Override
    public void process() {
        System.out.println("Processing complete.");
    }
    private void heavyInitialConfiguration() {
        System.out.println("Loading initial configuration...");
    }
}

public class ExpensiveObjectProxy implements ExpensiveObject {
    private static ExpensiveObject object;
    @Override
    public void process() {
        if (object == null) {
            object = new ExpensiveObjectImpl();
        }
        object.process();
    }
}
```

When to Use Proxy Pattern

- When you want to simplify a complex or heavy object.
- When the original object resides in a different address space and needs to be accessed locally.
- When you need to add a layer of security to control access.

Decorator Pattern

The **Decorator Pattern** enhances the behavior of an object dynamically by adding additional responsibilities without modifying the original class.

- **Example:**

```
public interface ChristmasTree {

    String decorate();

}

public class ChristmasTreeImpl implements ChristmasTree {

    @Override

    public String decorate() {

        return "Christmas tree";

    }

}

public abstract class TreeDecorator implements ChristmasTree {

    private ChristmasTree tree;


    public TreeDecorator(ChristmasTree tree) {

        this.tree = tree;

    }

    @Override

    public String decorate() {
```

```
        return tree.decorate();
    }
}
public class BubbleLights extends TreeDecorator {
    public BubbleLights(ChristmasTree tree) {
        super(tree);
    }
    @Override
    public String decorate() {
        return super.decorate() + " with Bubble Lights";
    }
}
```

When to Use Decorator Pattern

- When you want to add, enhance, or remove the behavior of an object dynamically.
- When you need to modify the functionality of a single object without changing others.

For more questions and answers, you can follow these additional links:

- Devinterview-io Java Interview Questions
- Aatul Java Interview Questions and Answers
