**JAVA ASSIGNMENT – OOPS**

**1. Java Inheritance**

In Java, inheritance enables a class to utilize fields and methods from another class, fostering code reusability and establishing a hierarchical relationship among classes. This is achieved using the keyword "extends".

**2.** **Polymorphism**

Polymorphism in Java allows objects to be treated as instances of their parent class, facilitating uniform handling of diverse objects, notwithstanding their underlying differences. This is accomplished through method overriding and overloading.

**3. Method Overloading in Java**

Method overloading occurs when a class contains multiple methods with the same name but different parameter lists, enabling similar tasks to be performed in varying ways based on specific requirements.

**4. Method Overriding in Java**

Method overriding in Java refers to a subclass altering the behavior of a method inherited from its superclass, while maintaining the method's signature, return type, and parameter list.

**5. Java Abstraction**

Abstraction in Java involves concealing the complexity of an entity and exposing only its essential features. This is realized through abstract classes and interfaces.

**6. Java Encapsulation**

Encapsulation in Java involves bundling data and its associated methods within a single unit, thereby restricting access to the data and enforcing controlled usage through designated methods.

**7. Rules for Java Method Overriding**

- The overriding method in the subclass must have the same name, return type, and parameter list as the overridden method in the superclass.

- The overriding method cannot impose stricter accessibility constraints than the overridden method and must only throw exceptions compatible with those thrown by the overridden method.

**8. Constructors in Java**

Constructors in Java serve as specialized mechanisms for initializing objects, preparing them for use. They can be tailored to accommodate various initialization scenarios.

**9. Types of Java Constructors**

- Default Constructor: Automatically provided by Java if a class does not define its own constructor.

- Parameterized Constructor: Accepts specific instructions for customizing the initialization of objects.

**10. Constructor Overloading in Java**

Constructor overloading entails defining multiple constructors within a class, each catering to different initialization requirements, thereby offering flexibility in object creation.

**11. Interface in Java**

An interface in Java acts as a blueprint specifying the contract that implementing classes must adhere to, without dictating the implementation details. Implementing classes comply with these rules while retaining the freedom to define their unique behavior

**JAVA ASSIGNMENT-2**

**Access Modifiers**

- private: Restricts access to class members within the same class only.

- protected: Grants access to class members within the same package and to subclasses, including those outside the package.

- public: Allows unrestricted access to class members from any other class.

- default: Permits access to package-private data members.

**Class, Interface, and Object Keywords**

- class: Identifies a class definition.

- interface: Defines an interface.

- enum: Declares an enumerated type.

- extends: Indicates inheritance from a superclass.

- implements: Specifies implementation of an interface.

- new: Instantiates new objects.

**Control Flow Statements**

- if: Evaluates a condition and executes associated code if true.

- else: Executes alternative code if the preceding if condition is false.

- switch: Selects and executes one of multiple code blocks.

- case: Defines a specific path in a switch statement.

- default: Specifies the default code block in a switch statement if no case matches.

- while: Repeats a block of statements while a condition remains true.

- do: Executes a block of statements once, then repeats while a condition holds true.

- for: Iterates over a range or collection.

- break: Exits the current loop or switch statement.

- continue: Skips the current iteration of a loop and proceeds to the next.

- return: Exits the current method optionally returning a value.

- try: Initiates a block of code to be tested for exceptions.

- catch: Catches exceptions thrown within the try block.

- finally: Executes a block of code after the try and catch blocks, regardless of errors.

- throw: Raises an exception.

- throws: Lists exceptions a method might throw.

**Primitive Data Types**

- byte: Represents 8-bit integer data.

- short: Represents 16-bit integer data.

- int: Represents 32-bit integer data.

- long: Represents 64-bit integer data.

- float: Represents single-precision 32-bit floating-point data.

- double: Represents double-precision 64-bit floating-point data.

- char: Represents a single 16-bit Unicode character.

- boolean: Represents a boolean value with only two possible states: true and false.

**Non-Access Modifiers**

- abstract: Indicates that a class or method is abstract and must be implemented by subclasses.

- final: Prevents further inheritance or modification of a class, method, or variable.

- static: Indicates that a member belongs to the class itself, rather than to instances of the class.

- synchronized: Ensures that a method or block of code is accessed by only one thread at a time.

- volatile: Specifies that a variable's value may be changed by different threads.

- transient: Prevents a field from being serialized.

**Miscellaneous**

- this: Refers to the current instance of a class.

- super: Refers to the superclass of the current object.

- void: Specifies that a method does not return any value.

- const: Reserved but unused.

- goto: Reserved but unused.

- instanceof: Checks whether an object is an instance of a specific class or interface.

- package: Declares a package.

- import: Brings in other Java packages or classes.

- native: Indicates that a method is implemented in native code using JNI.

- strictfp: Constrains floating-point arithmetic to ensure consistency across platforms.

- null: Represents no reference.

- assert: Used for debugging purposes to assert a statement.

**JAVA Assignment 3**

Exception Handling

Hierarchy of Exception classes

In Java, all exception classes inherit from the base class Throwable. The two main subclasses are Exception (checked exceptions) and RuntimeException (unchecked exceptions). Error represents serious problems that a reasonable application should not try to catch.

Types of Exception

1. Checked Exceptions: These are checked at compiletime, and the programmer is required to handle them using trycatch or declare them using throws.

2. Unchecked Exceptions (RuntimeExceptions): These exceptions are not checked at compiletime and typically occur due to programming errors or unexpected conditions (e.g., NullPointerException, ArrayIndexOutOfBoundsException).

Try catch finally

try: Encloses the code that might throw exceptions.

catch: Handles specific exceptions caught by try.

finally: Optional block that executes after try block regardless of whether an exception is thrown or not. Useful for cleanup actions (closing resources, etc.).

throw

throw is used to explicitly throw an exception. It is followed by an instance of an exception class.

throws

throws is used in method signature to declare that the method might throw one or more exceptions. It lists the exception classes that the method might throw.

Multiple catch block

You can have multiple catch blocks to handle different types of exceptions thrown by the try block.

Exception Handling with method Overriding

Subclasses can override methods from their superclass to throw exceptions that are subclasses of the exceptions thrown by the superclass method. This allows for more specific exception handling in subclasses.

Java Collection Framework

Hierarchy of Collection Framework

At the top level, the Java Collection Framework includes the Collection interface, which is extended by List, Set, and Queue interfaces. The Map interface stands alone but is also part of the framework.

Collection interface

Defines the basic methods that all collections will have.

Iterator interface

Provides methods to iterate over elements in a collection.

Set, List, Queue, Map interfaces

Set: Represents a collection that contains no duplicate elements.

List: An ordered collection (sometimes called a sequence). Allows duplicate elements.

Queue: A collection designed for holding elements prior to processing (FIFO order).

Map: An object that maps keys to values. Each key can map to at most one value.

ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet, HashMap, ConcurrentHashMap classes

ArrayList: Resizable array implementation of List.

Vector: Synchronized version of ArrayList.

LinkedList: Doublylinked list implementation of List.

PriorityQueue: Priority queue based on a priority heap.

HashSet: Implements Set using a hash table.

LinkedHashSet: Hash table and linked list implementation of Set.

TreeSet: Redblack tree implementation of Set.

HashMap: Hash table based implementation of Map.

ConcurrentHashMap: A highperformance concurrent version of HashMap.

**JAVA Assignment 4**

Multithreading

Multithreading in Java allows concurrent execution of multiple threads within a single process, enabling efficient utilization of CPU resources and better responsiveness in applications.

Lifecycle of a Thread

1. New: When a thread is created but not yet started.

2. Runnable: When the thread is executing in the JVM.

3. Blocked/Waiting: When the thread is waiting for a monitor lock to enter or reenter a synchronized block/method or waiting indefinitely for another thread to perform a certain action.

4. Timed Waiting: When the thread is waiting for another thread to perform an action for up to a specified waiting time.

5. Terminated: When the thread has completed execution or terminated due to an exception.

Thread Priority in Multithreading

Threads in Java can have priorities ranging from 1 (lowest) to 10 (highest). The default priority is 5. Higher priority threads are scheduled before lower priority threads.

Runnable interface in Java

The Runnable interface in Java is used to define a task that can be executed by a thread. It requires implementing a single method run(), where the code for the task is defined.

start() function in multithreading

The start() method in Java is used to start the execution of a thread. When start() is called, the JVM calls the run() method on the thread instance, which executes the thread's task.

Thread.sleep() Method in Java

Thread.sleep() is a static method that causes the currently executing thread to sleep (temporarily pause execution) for a specified number of milliseconds. It's used for delaying execution or for timingbased operations.

Thread.run() in Java

Thread.run() is the entry point for the thread's task. It's where the code specified in the run() method of the Runnable interface or overridden in a Thread subclass is executed.

Deadlock in Java

Deadlock occurs when two or more threads are blocked forever, waiting for each other to release resources. This happens when two or more threads have circular dependencies on a pair of synchronized objects.

Synchronization in Java

Synchronization in Java is used to control access to shared resources or critical sections by multiple threads. It ensures that only one thread can execute a synchronized block or method at a time, preventing data inconsistency and thread interference.

Method Level Lock: When a synchronized method is used, it acquires an implicit lock on the object's monitor (this is also known as objectlevel locking).

Block Level Lock: Explicit synchronization can also be achieved by using synchronized blocks, where a specific block of code is synchronized on a given object.

Executor Framework in Java

The Executor Framework provides a higherlevel abstraction for managing and executing threads. It decouples task submission from the mechanics of how each task will be run, which can improve the efficiency and scalability of thread use in applications.

Callable Interface in Java

The Callable interface in Java is similar to Runnable, but it can return a result and throw a checked exception. It's used with the ExecutorService to submit tasks that need to return results or propagate exceptions.

**SPRINGBOOT ANNOTATIONS**

Spring Boot utilizes annotations extensively to simplify the development of Springbased applications.

1. @SpringBootApplication:

This annotation is used to mark the main class of a Spring Boot application. It combines @Configuration, @EnableAutoConfiguration, and @ComponentScan annotations, enabling component scanning and autoconfiguration.

2. @RestController:

Used to define a Spring MVC controller that handles RESTful requests. Methods within such controllers annotated with @RequestMapping, @GetMapping, @PostMapping, etc., handle HTTP requests and return appropriate responses.

3. @RequestMapping:

Maps HTTP requests to handler methods of @Controller classes. It is often used at the class level in combination with methodlevel annotations like @GetMapping, @PostMapping, etc., to specify the request path.

4. @Autowired:

This annotation can be used to automatically wire dependencies into a Spring Bean. It supports both constructorbased and fieldbased dependency injection.

5. @Component:

Indicates that a class is a Spring component. Spring Boot will automatically detect and register components annotated with @Component during component scanning.

6. @Service:

Indicates that a class is a service layer component in the application. It is typically used to define business logic and is a specialization of @Component.

7. @Repository:

Indicates that a class is a repository component in the application. It is used to indicate a data access component that interacts with a database or external data source.

8. @Configuration:

Indicates that a class provides configuration to the Spring application context. It is used in conjunction with @Bean methods to define beans explicitly.

9. @EnableAutoConfiguration:

Enables Spring Boot's autoconfiguration mechanism, which automatically configures the Spring application based on dependencies and classpath settings.

10. @Transactional:

Used to mark a method or class as transactional. Spring Boot manages transactions for methods annotated with @Transactional, providing rollback capabilities for runtime exceptions.