Java Assignment

Name : Syedanwar Hanzahusen Mashalkar

==

Java Keywords

1. **abstract**: Specifies that a class or method is abstract, meaning it cannot be instantiated or must be overridden.
2. **assert**: Used for debugging purposes to make an assertion.
3. **boolean**: Defines a boolean variable with a true or false value.
4. **break**: Exits from a loop or switch statement.
5. **byte**: Defines a byte data type, which is an 8-bit signed integer.
6. **case**: Defines a block of code in a switch statement.
7. **catch**: Catches exceptions generated by try statements.
8. **char**: Defines a character data type, which is a 16-bit Unicode character.
9. **class**: Defines a class.
10. **const**: Not used; reserved for future use.
11. **continue**: Skips the current iteration of a loop and proceeds to the next iteration.
12. **default**: Specifies the default block of code in a switch statement.
13. **do**: Starts a do-while loop.
14. **double**: Defines a double-precision 64-bit floating-point number.
15. **else**: Specifies a block of code to be executed if a condition in an if statement is false.
16. **enum**: Defines a set of named constants.
17. **extends**: Indicates that a class is inheriting from a superclass.
18. **final**: Defines an entity that can only be assigned once.
19. **finally**: Creates a block of code that follows a try block; it will be executed regardless of whether an exception was thrown or not.
20. **float**: Defines a single-precision 32-bit floating-point number.
21. **for**: Starts a for loop.
22. **goto**: Not used; reserved for future use.
23. **if**: Tests a condition and executes a block of code if the condition is true.
24. **implements**: Indicates that a class implements an interface.
25. **import**: Imports other Java packages or classes.
26. **instanceof**: Tests whether an object is an instance of a specific class or implements an interface.
27. **int**: Defines a 32-bit signed integer.
28. **interface**: Defines an abstract type used to specify a behavior that classes must implement.
29. **long**: Defines a 64-bit signed integer.
30. **native**: Specifies that a method is implemented in native code using JNI (Java Native Interface).

31. **new**: Creates new objects.
32. **null**: Represents a null reference.
33. **package**: Defines a package (namespace) for classes.
34. **private**: Specifies that a member is accessible only within its own class.
35. **protected**: Specifies that a member is accessible within its package and by subclasses.
36. **public**: Specifies that a member is accessible by any other class.
37. **return**: Exits from a method, optionally returning a value.
38. **short**: Defines a 16-bit signed integer.
39. **static**: Indicates that a member belongs to the class itself rather than instances of the class.
40. **strictfp**: Restricts floating-point calculations to ensure portability.
41. **super**: Refers to the superclass of the current object.
42. **switch**: Starts a switch statement.
43. **synchronized**: Specifies that a method or block of code is synchronized; only one thread can access it at a time.
44. **this**: Refers to the current object.
45. **throw**: Throws an exception.
46. **throws**: Indicates which exceptions a method can throw.
47. **transient**: Specifies that a member is not part of the serialized form of an object.
48. **try**: Starts a block of code that will be tested for exceptions.
49. **void**: Specifies that a method does not return a value.
50. **volatile**: Indicates that a variable may be changed unexpectedly, used in multi-threading.
51. **while**: Starts a while loop.

Java Data Types

1. **byte**:
   - Size: 8-bit
   - Range: -128 to 127
   - Description: Used to save space in large arrays.
2. **short**:
   - Size: 16-bit
   - Range: -32,768 to 32,767
   - Description: A short integer data type.
3. **int**:
   - Size: 32-bit
   - Range: -2^31 to 2^31-1
   - Description: The default integer data type.
4. **long**:
   - Size: 64-bit
   - Range: -2^63 to 2^63-1
   - Description: Used when a wider range than int is needed.
5. **float**:
   - Size: 32-bit
   - Range: Approximately ±3.40282347E+38F (6-7 significant decimal digits)
   - Description: Single-precision floating-point.

6. **double**:
    - Size: 64-bit
    - Range: Approximately ±1.79769313486231570E+308 (15 significant decimal digits)
    - Description: Double-precision floating-point.
7. **boolean**:
    - Size: 1-bit
    - Values: true or false
    - Description: Represents one bit of information.
8. **char**:
    - Size: 16-bit
    - Range: 0 to 65,535 (Unsigned)
    - Description: A single 16-bit Unicode character.

OOPS Concepts in Java

Java Inheritance

Inheritance allows one class (subclass) to inherit the fields and methods of another class (superclass), enabling code reuse and the creation of hierarchical relationships.

Polymorphism

Polymorphism enables objects to be treated as instances of their parent class rather than their actual class, allowing methods to perform different tasks based on the object they are acting upon.

Method Overloading in Java

Method Overloading is when multiple methods in the same class have the same name but different parameters (different type, number, or both), allowing different ways to call a method.

Method Overriding in Java

Method Overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass, allowing runtime polymorphism.

Java Abstraction

Abstraction involves hiding the implementation details of a class and exposing only the essential features. It is achieved using abstract classes and interfaces.

Java Encapsulation

Encapsulation is the practice of wrapping data (variables) and methods that operate on the data into a single unit or class, restricting direct access to some of the object's components.

Rules for Java Method Overriding

1. The method must have the same name as in the parent class.
2. The method must have the same parameter list as in the parent class.
3. There must be an IS-A relationship (inheritance) between the classes.

Constructors in Java

Constructors are special methods invoked when an object is instantiated. They initialize the object's state.

Types of Java Constructors

1. **Default Constructor**: No parameters, initializes objects with default values.
2. **Parameterized Constructor**: Accepts parameters to initialize objects with specific values.

Constructor Overloading in Java

Constructor Overloading is defining multiple constructors in a class, each having a different parameter list, allowing different ways to initialize an object.

Interface in Java

An interface is a reference type in Java, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors, and they are used to achieve abstraction and multiple inheritance.

Java Inheritance

Allows a class to inherit fields and methods from another class.

Polymorphism

Enables methods to perform different tasks based on the object calling them.

Method Overloading in Java

Multiple methods with the same name but different parameters within the same class.

Method Overriding in Java

A subclass provides a specific implementation for a method already defined in its superclass.

Java Abstraction

Hides implementation details and exposes only essential features using abstract classes and interfaces.

Java Encapsulation

Combines data and methods into a single unit (class) and restricts direct access to some components.

Rules for Java Method Overriding

1. Same method name as in the parent class.
2. Same parameter list as in the parent class.
3. The subclass must inherit from the superclass.

Constructors in Java

Special methods to initialize objects when they are created.

Types of Java Constructors

1. **Default Constructor**: No parameters.
2. **Parameterized Constructor**: With parameters.

Constructor Overloading in Java

Multiple constructors with different parameter lists in the same class.

Interface in Java

A reference type that can contain constants, method signatures, default methods, and static methods. Used for abstraction and multiple inheritance.

Exception Handling in Java

Hierarchy of Exception Classes:

- **Throwable**: The superclass for all exceptions and errors.

Error: Serious issues typically not handled by applications (e.g., `OutOfMemoryError`).

Exception: Conditions applications might want to handle.

- **RuntimeException**: Unchecked exceptions, handled at runtime (e.g., `NullPointerException`).
- **Checked Exceptions**: Must be declared or caught (e.g., `IOException`).

Types of Exception:

1. **Checked Exceptions**: Checked at compile-time.
2. **Unchecked Exceptions**: Checked at runtime.

try: Block to write code that might throw an exception.

catch: Block to handle exceptions thrown by the try block.

finally: Block that executes code regardless of whether an exception was thrown.

throw: Used to explicitly throw an exception.

throws: Used in method signatures to declare exceptions that might be thrown.

Multiple catch blocks: Allows handling of different types of exceptions separately after a try block.

Exception Handling with Method Overriding: In overridden methods, exceptions thrown must be the same or subclasses of those declared in the superclass method.Top of Form

Java Collection Framework

Hierarchy of Collection Framework:

- **Collection Interface**: The root interface for most of the collection classes.
  - **Set Interface**: Does not allow duplicate elements.
  - **List Interface**: Ordered collection that allows duplicate elements.
  - **Queue Interface**: Typically orders elements in a FIFO manner.
  - **Map Interface**: Collection of key-value pairs.

Collection Interface: Defines basic methods for adding, removing, and checking elements within a collection.

Iterator Interface: Provides methods to traverse through elements in a collection.

Set Interface: A collection that cannot contain duplicate elements.

List Interface: An ordered collection (also known as a sequence) that allows duplicate elements.

Queue Interface: A collection used to hold multiple elements prior to processing, typically ordered in FIFO.

Map Interface: An object that maps keys to values, with no duplicate keys allowed.

Comparator Interface: Defines a method for comparing two objects to determine their ordering.

Comparable Interface: Allows an object to be compared with another object of the same type to define a natural ordering.

ArrayList Class: A resizable array implementation of the List interface.

Vector Class: A synchronized resizable array implementation of the List interface.

LinkedList Class: A doubly-linked list implementation of the List and Deque interfaces.

PriorityQueue Class: An unbounded priority queue based on a priority heap.

HashSet Class: A hash table-based implementation of the Set interface.

LinkedHashSet Class: A hash table and linked list implementation of the Set interface, with predictable iteration order.

TreeSet Class: A NavigableSet implementation based on a TreeMap, which is sorted.

HashMap Class: A hash table-based implementation of the Map interface, allowing null values and keys.

ConcurrentHashMap Class: A thread-safe implementation of the Map interface that allows concurrent access to its elements.

Multithreading in Java

Lifecycle of a Thread: A thread can be in one of the following states during its lifecycle:

1. **New**: Created but not yet started.
2. **Runnable**: Ready to run, waiting for CPU allocation.
3. **Blocked**: Waiting to acquire a lock.
4. **Waiting**: Waiting indefinitely for another thread to perform a particular action.

5. **Timed Waiting**: Waiting for another thread to perform a specific action for a specified period.
6. **Terminated**: Has finished execution.

Thread Priority in Multithreading: Threads can have priority, which helps the thread scheduler decide the order of thread execution. Priorities range from `MIN_PRIORITY` (1) to `MAX_PRIORITY` (10), with `NORM_PRIORITY` (5) as the default.

Runnable Interface in Java: Defines a single method, `run()`, which is intended to contain the code executed by a thread. Classes that implement this interface can be executed by a thread.

start() Function in Multithreading: The `start()` method is used to begin the execution of a thread. It causes the JVM to call the `run()` method of the thread.

Thread.sleep() Method in Java: Puts the currently executing thread to sleep for a specified number of milliseconds, allowing other threads to execute.

Thread.run() in Java: Contains the code that constitutes the new thread's task. This method is called when the thread is started but should not be called directly.

Deadlock in Java: Occurs when two or more threads are blocked forever, each waiting on the other to release a resource.

Synchronization in Java: Prevents concurrent access to shared resources, ensuring that only one thread can access a resource at a time. This can be achieved using synchronized methods or blocks.

Method Level Lock: Achieved by applying the `synchronized` keyword to a method. It locks the object for any synchronized method calls.

Block Level Lock: More granular synchronization by locking only a specific block of code within a method using the `synchronized` keyword.

Executor Framework in Java: A higher-level replacement for working with threads directly. It provides a pool of threads and API for assigning tasks to them.

Callable Interface in Java: Similar to `Runnable` but can return a result and throw a checked exception. It defines a single method, `call()`, which is executed by a thread.