Thread Task

ERAASOFT
Abdulrahman Hussien Ahmed
Presented to: Eng. Eslam khder

1) List vs Vector in Java:

Java provides multiple implementations of the List interface. ArrayList and Vector are two of them. Here's a breakdown of how List (specifically ArrayList) differs from Vector.

1. Thread Safety

- Vector is thread-safe. All its methods are synchronized.
- ArrayList is **not** thread-safe. You need to handle synchronization manually if using it in multithreaded environments.

2. Synchronization

- Vector synchronizes each method (like add(), remove(), get()).
- ArrayList does **not** synchronize any methods by default.
- You can make ArrayList synchronized by wrapping it:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
```

3. Performance

- ArrayList is faster in single-threaded environments because it avoids the overhead of synchronization.
- Vector is slower due to constant synchronization, even when thread safety is not needed.
- In modern Java apps, developers prefer ArrayList with external synchronization if needed.

2) HashSet vs LinkedHashSet in Java

Both are implementations of the Set interface. Both store unique elements. But they behave differently.

1. Order of Elements

- HashSet does **not** maintain insertion order

  Example: Add 10, 5, 20 → You may get [20, 10, 5]
- LinkedHashSet **maintains insertion order**

  Example: Add 10, 5, 20 → You always get [10, 5, 20]

2. Performance

- HashSet is slightly faster

  It uses a simple hash table with no order tracking

  Good for fast lookup, insert, delete
- LinkedHashSet is slower
  It adds a linked list on top of the hash table to preserve order
  Slight memory and CPU overhead

3. Use Cases

Use **HashSet** when:

- You don't care about order
- You want high performance
- Example: Storing unique IDs, tags, or hashed values

Use **LinkedHashSet** when:

- You care about the order of insertion
- You want predictable iteration
- Example: Caching items, preserving user input order

3) String vs StringBuilder vs StringBuffer in Java

1. Mutability

- String is **immutable**
  You can't change its content once created
  Every change creates a new object
  Example:

  ```
  String a = "hi";
  a = a + " there";   // creates new object
  ```

- StringBuilder is **mutable**
  You can modify the object directly
  Example:

  ```
  StringBuilder sb = new StringBuilder("hi");
  sb.append(" there");  // same object
  ```

- StringBuffer is **also mutable**
  Same as StringBuilder but synchronized

2. Thread Safety

- String is thread-safe
  It’s immutable, so it's always safe to share
- StringBuilder is **not thread-safe**
  Multiple threads can break it if used without care
- StringBuffer is **thread-safe**
  Methods are synchronized, safe in multithreaded code

3. Performance

- String is **slow** for heavy modification
  Each change creates a new object
- StringBuilder is **fastest**
  Best for building strings in loops or large operations
- StringBuffer is **slower** than StringBuilder
  Synchronization adds overhead

4. When to Use

Use **String** when:

- You don't need to modify the text
- Example: keys in maps, constants, messages

Use **StringBuilder** when:

- You build strings often and work in a single thread
- Example: creating SQL queries, JSON strings, log messages

Use **StringBuffer** when:

- You build strings in a multithreaded context
- Example: shared log writer, multi-threaded parsers