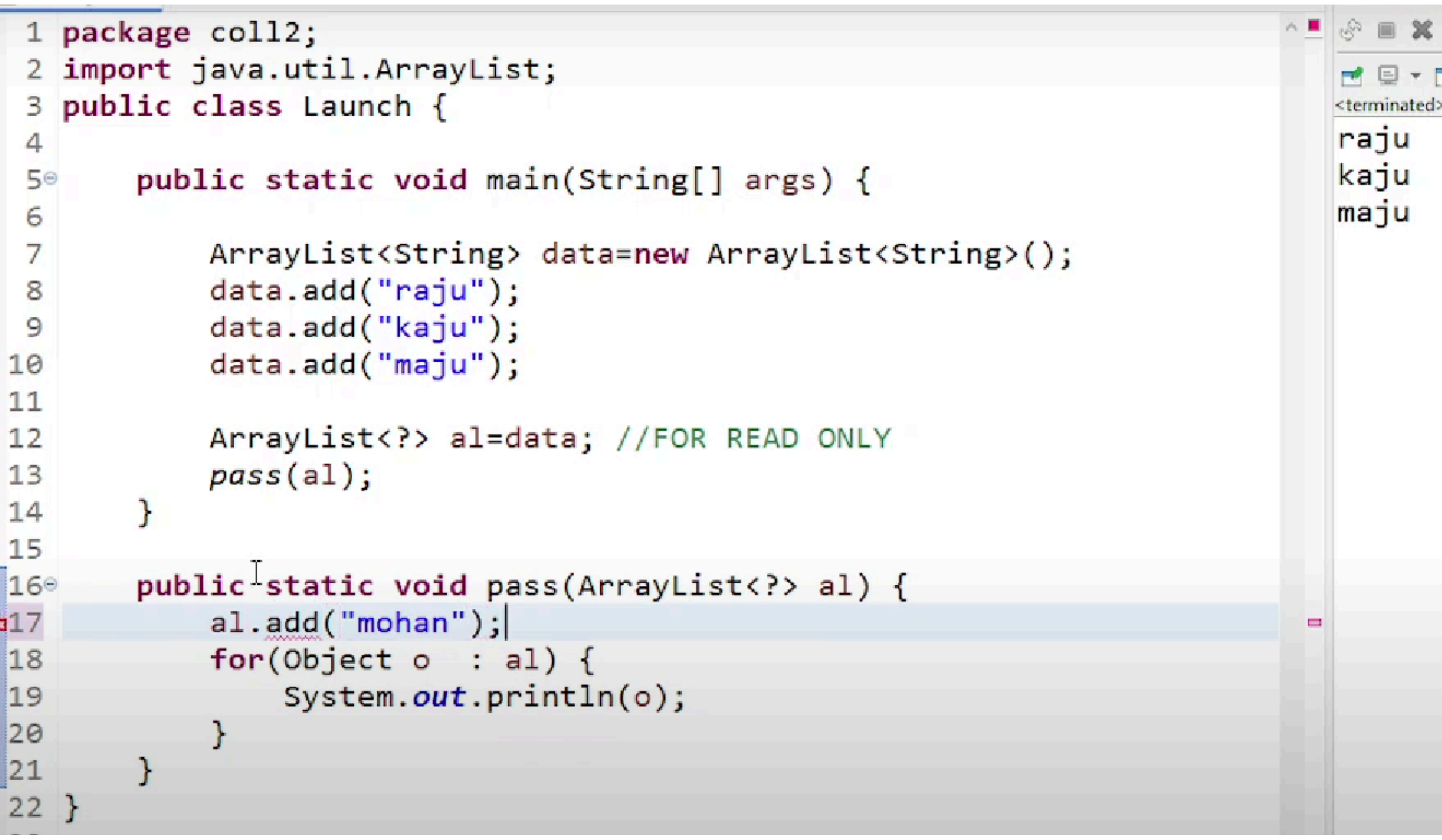
Lec 40 Collections P5:

This lesson explains advanced usage of Java collections, focusing on wildcard generics and the Map interface. It covers how wildcard generics provide type safety for read-only access, how to customize collection behavior, and how different Map implementations (HashMap, LinkedHashMap, TreeMap) work with key-value pairs, including how to nest collections for complex data storage.

Understanding Set and Wildcard Generics

Sets in Java do not store duplicate elements and allow only one null. Wildcard generics (using ?) in collections mean the type is unknown, so you cannot add elements except for null. This makes the collection effectively read-only, useful when you want to share data without allowing modifications.



```java
package coll2;
import java.util.ArrayList;
public class Launch {

    public static void main(String[] args) {

        ArrayList<String> data=new ArrayList<String>();
        data.add("raju");
        data.add("kaju");
        data.add("maju");

        ArrayList<?> al=data; //FOR READ ONLY
        pass(al);
    }

    public static void pass(ArrayList<?> al) {
        al.add("mohan");
        for(Object o  : al) {
            System.out.println(o);
        }
    }
}
```

<terminated>

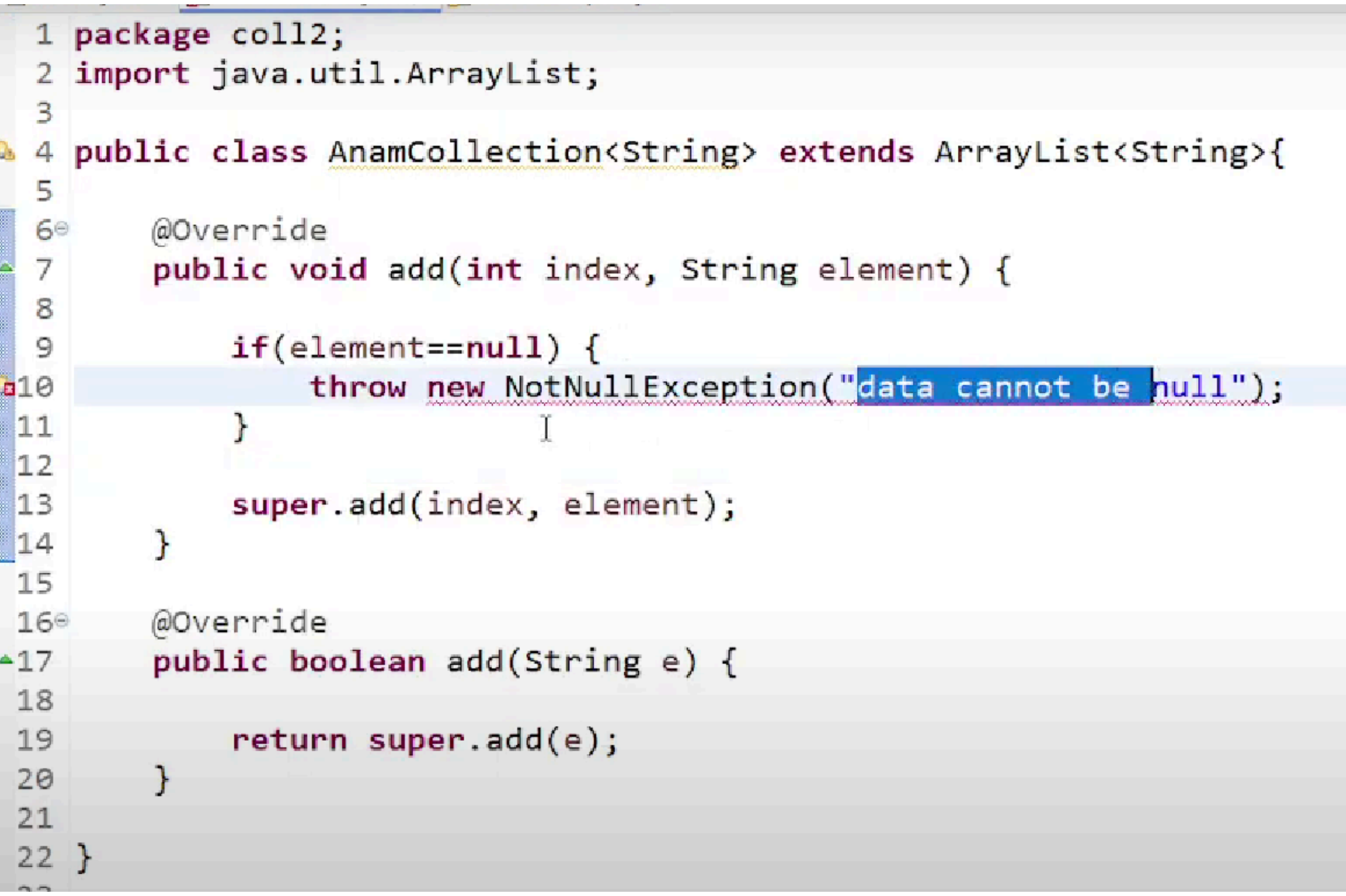
```
raju
kaju
maju
```

Passing Collections with Wildcards

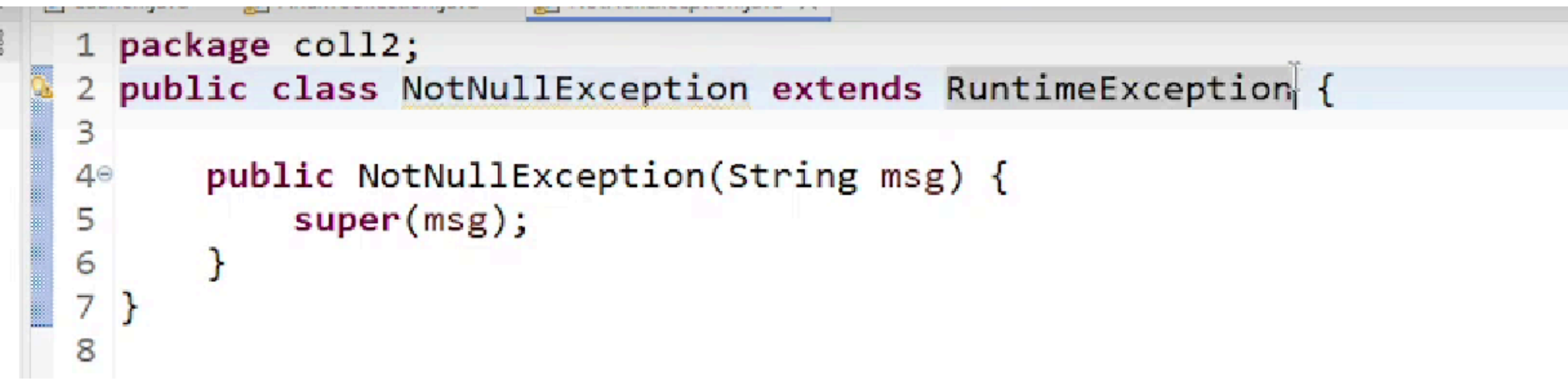
Wildcard generics are helpful when a method should accept any type of collection without knowing its specific type. You can pass a collection (like an ArrayList of Strings) to a method that only needs to read data, ensuring the method cannot add new elements, preserving data integrity.

Customizing Collection Behavior

You can extend existing collections (like ArrayList) to create custom behavior, such as preventing null values from being added. This is done by overriding methods (like add) and throwing custom exceptions if unwanted data (like null) is inserted.



```java
package coll2;
import java.util.ArrayList;

public class AnamCollection<String> extends ArrayList<String>{

    @Override
    public void add(int index, String element) {

        if(element==null) {
            throw new NotNullException("data cannot be null");
        }

        super.add(index, element);
    }

    @Override
    public boolean add(String e) {

        return super.add(e);
    }

}
```

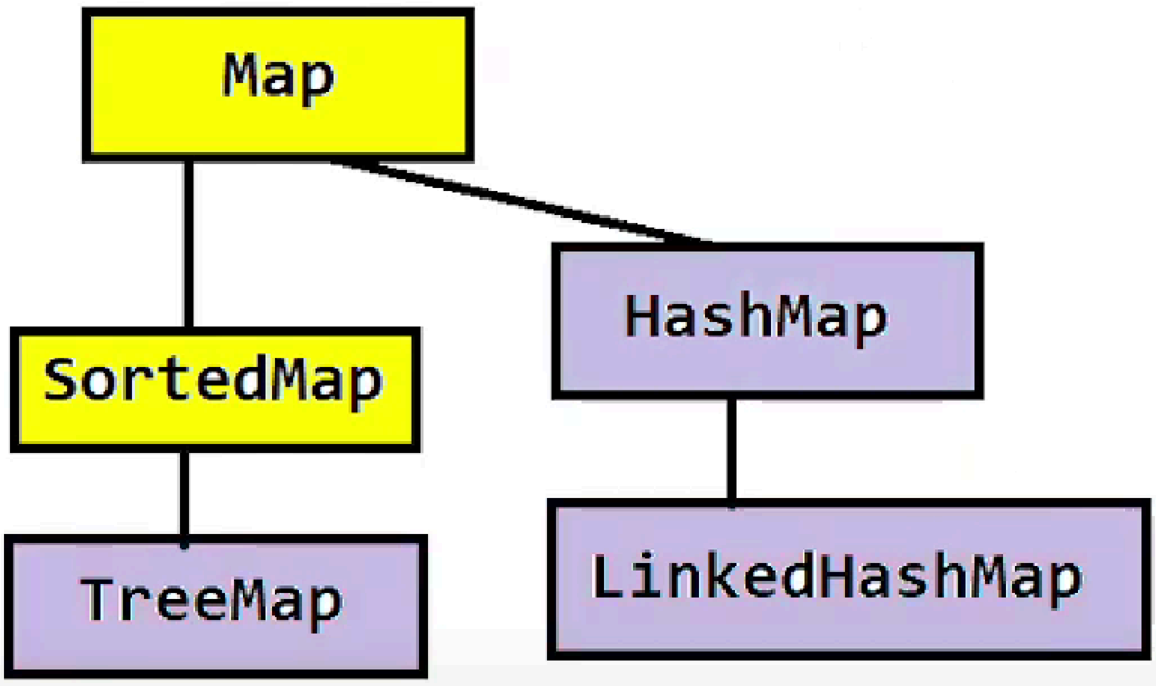


```java
package coll2;
public class NotNullException extends RuntimeException {

    public NotNullException(String msg) {
        super(msg);
    }
}
```

- Above code shows how to make custom collections and this code also revises custom exception creation process.

Introduction to Map Interface

The Map interface is used for storing data as key-value pairs, unlike other collections. Maps are not part of the main collection hierarchy. Key implementations include HashMap (random order), LinkedHashMap (insertion order), and TreeMap (sorted order). Maps require unique keys but allow duplicate values and support one null key and multiple null values.

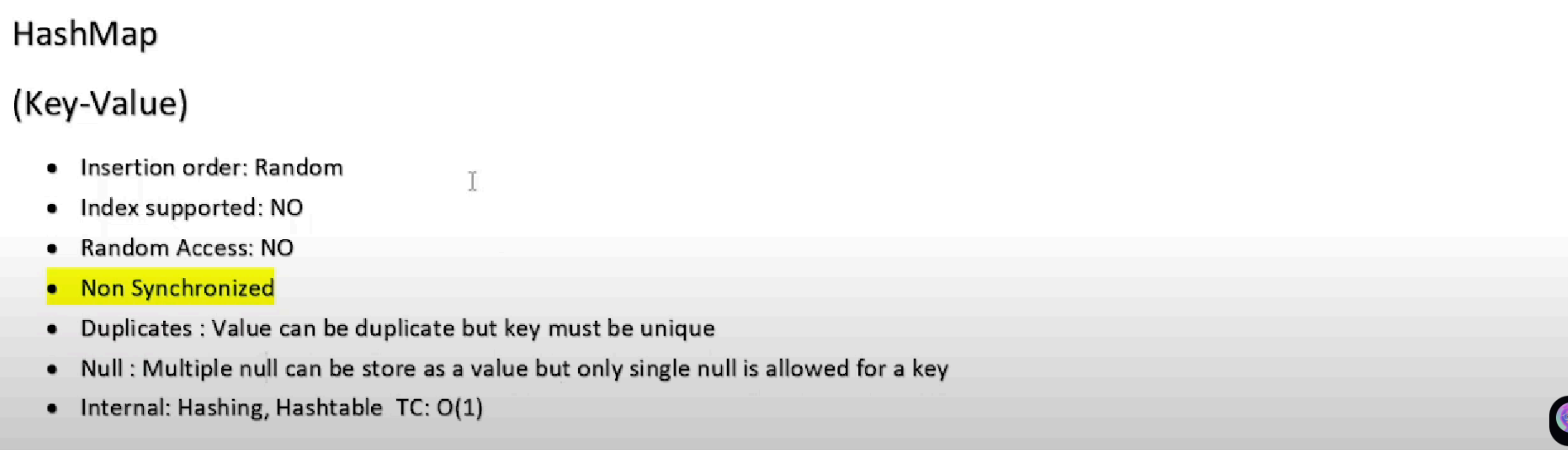


HashMap, Linked HashMap, and Tree Map Differences

HashMap: Stores key-value pairs with random order using hashing, provides fast access, allows one null key and many null values.

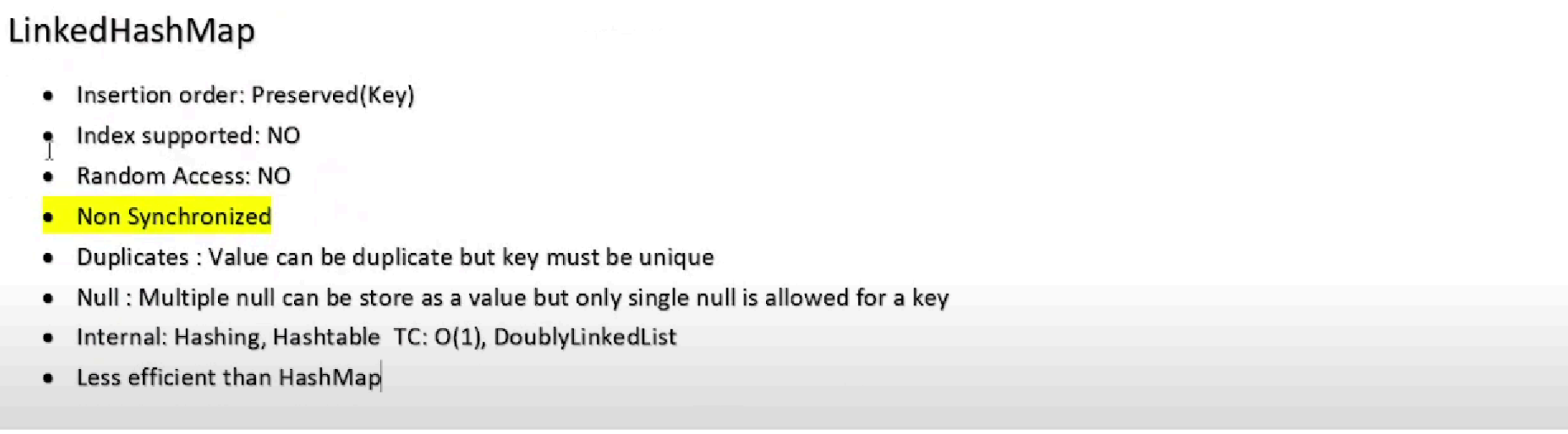
Linked HashMap: Maintains insertion order using a doubly linked list, less efficient than HashMap but useful when order matters.

Tree Map: Maintains keys in sorted (ascending) order using a Red-Black tree, does not allow null keys, and supports custom sorting with a comparator.

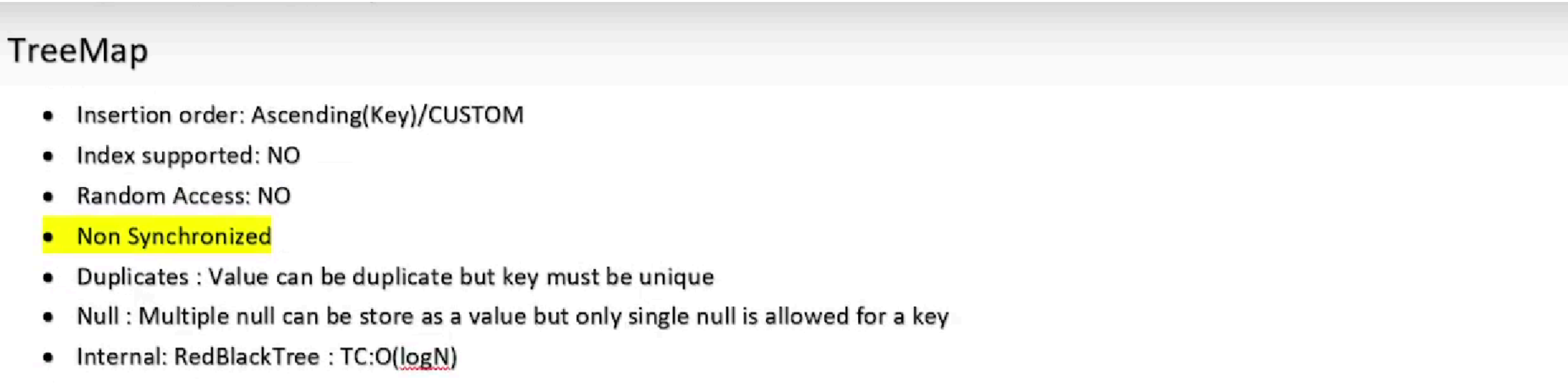

HashMap

(Key-Value)

- Insertion order: Random
- Index supported: NO
- Random Access: NO
- Non Synchronized
- Duplicates : Value can be duplicate but key must be unique
- Null : Multiple null can be store as a value but only single null is allowed for a key
- Internal: Hashing, Hashtable TC: O(1)


LinkedHashMap

- Insertion order: Preserved(Key)
- Index supported: NO
- Random Access: NO
- Non Synchronized
- Duplicates : Value can be duplicate but key must be unique
- Null : Multiple null can be store as a value but only single null is allowed for a key
- Internal: Hashing, Hashtable TC: O(1), DoublyLinkedList
- Less efficient than HashMap


TreeMap

- Insertion order: Ascending(Key)/CUSTOM
- Index supported: NO
- Random Access: NO
- Non Synchronized
- Duplicates : Value can be duplicate but key must be unique
- Null : Multiple null can be store as a value but only single null is allowed for a key
- Internal: RedBlackTree : TC:O(logN)

Creating and Using a HashMap

HashMap can store any object types as keys and values, but primitives are auto-boxed into objects. Data is added using the put method and can be updated by reusing the same key. Duplicate values are allowed, but keys must remain unique.

Internal Structure: Entry Objects in a Map

Each key-value pair in a Map is stored as an Entry object. When iterating over a Map, you work with a set of these Entry objects, allowing you to retrieve both the key and the value for each pair.

Iterating Over a Map

Maps do not support direct iteration like lists, so you use the entrySet() method to convert the Map into a set of entries, then use an iterator to loop through and access each key-value pair.

Nested Collections with Maps

Maps can store complex values, including collections like lists. For example, a Map can associate a pin code (key) with a list of car names (value), allowing for efficient organization and retrieval of grouped data.

Practical Uses and Deep Nesting

You can nest collections inside each other (e.g., a Map of Lists, or even a List inside a List) to model complex data structures. This is useful for real-world scenarios like mapping product IDs to lists of related items or categories.