Java Collections Framework

This project demonstrates the **usage, hierarchy, and performance characteristics** of core Java Collection Framework data structures through simple example methods.

The `main` method acts as a **driver** that invokes methods corresponding to different collection categories.

Java Collection Framework – Class Hierarchy

Collection Hierarchy

```
java.lang.Iterable
    |
    v
java.util.Collection
    |
    +-- java.util.List
    |       |
    |       +-- ArrayList
    |       |
    |       +-- LinkedList
    |               |
    |               +-- Deque
    |
    +-- java.util.Set
            |
            +-- HashSet
            |       |
            |       +-- (internally uses HashMap)
            |
            +-- TreeSet
                    |
                    +-- (internally uses TreeMap)
```

Queue and Deque Hierarchy

```
java.util.Collection
    |
    v
java.util.Queue
    |
    +-- PriorityQueue
    |
    +-- Deque
            |
            +-- ArrayDeque
            |
            +-- LinkedList
```

Stack Hierarchy (Legacy)

```
java.util.Vector
    |
    +-- Stack
```

`Stack` is a legacy synchronized class.
Modern Java prefers `Deque` for stack behavior.

Map Hierarchy (IMPORTANT)

```
java.util.Map
    |
    +-- HashMap
    |
    +-- TreeMap
    |       |
    |       +-- (Red-Black Tree based)
    |
    +-- HashTable   (Legacy, synchronized)
```

Internal Working Summary

ArrayList

- Backed by a dynamic array
- Fast random access
- Costly middle insertions/removals

LinkedList

- Doubly linked list
- Fast insert/remove when node is known
- Slow random access

HashSet

- Internally uses `HashMap`
- No ordering
- No duplicates

TreeSet

- Internally uses `TreeMap`
- Maintains sorted order
- No duplicates

HashMap

- Hash table with buckets
- Allows one null key
- Not thread-safe

HashTable

- Synchronized version of HashMap (legacy)
- No null key or value

- Slower due to locking

TreeMap

- Red-Black Tree based
- Sorted by key
- No hashing, no buckets, no collisions

Time Complexity Comparison (Average Case)

Legend

- `add` → insert element
- `remove` → delete element
- `contains` → search element / key

Data Structure	add	remove	contains
ArrayList	O(1)*	O(n)	O(n)
LinkedList	O(1)**	O(1)**	O(n)
HashSet	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)
HashMap	O(1)	O(1)	O(1)
Hashtable	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)
Stack	O(1)	O(1)	O(n)
Queue (LinkedList)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(n)
Deque (ArrayDeque)	O(1)	O(1)	O(n)

* Amortized time (resizing may cost O(n))
** Only when position/node is known

Worst case for hash-based structures can degrade to **O(n)** due to collisions.
Tree-based structures guarantee **O(log n)**.

HashMap vs HashTable vs TreeMap

Feature	HashMap	Hashtable	TreeMap
Ordering	No	No	Sorted
Thread-safe	No	Yes	No
Null key	One	No	No
Internal DS	Hash table	Hash table	Red-Black Tree
Performance	Fast	Slower	Moderate

Concurrent Collections (Covered in Project)

Examples:

- `ConcurrentHashMap`
- `CopyOnWriteArrayList`
- `CopyOnWriteArraySet`
- `BlockingQueue` variants

Purpose:

- Thread safety without global synchronization
- Better scalability than `HashTable`