COLLECTIONS

MIGUEL ANGEL RAMIREZ JUAREZ

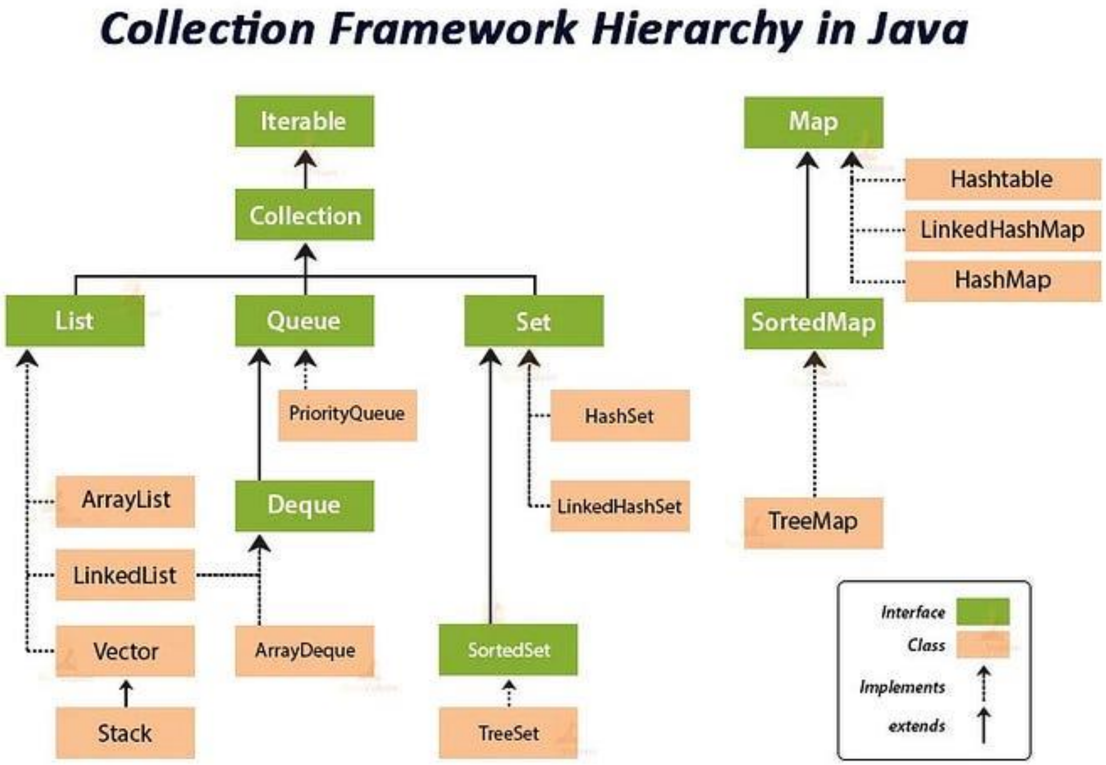
JAVA ACADEMY

AUGUST/ 22/ 2024

What is a Collection in Java?

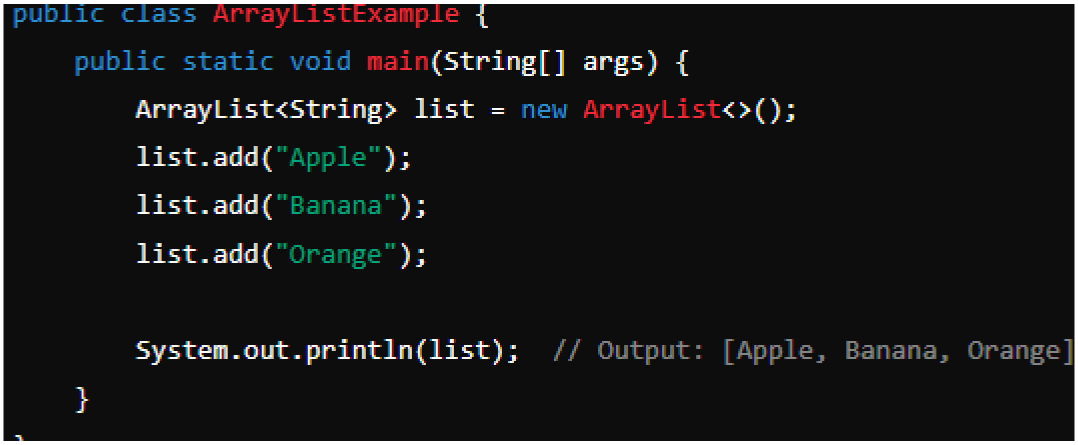
A **collection** in Java is an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. They represent data structures such as lists, sets, and maps, and are a fundamental part of the Java programming language. Collections enable developers to manage groups of objects more efficiently by providing various interfaces and classes for handling data structures with built-in functionality for tasks like sorting, searching, and iteration.

The Java Collections Framework (JCF) provides a set of standard interfaces and classes that allow developers to work with groups of objects in a consistent and flexible manner. This framework includes both the core interfaces like List, Set, Queue, and Map, and concrete implementations like ArrayList, HashSet, LinkedList, and HashMap.



LIST

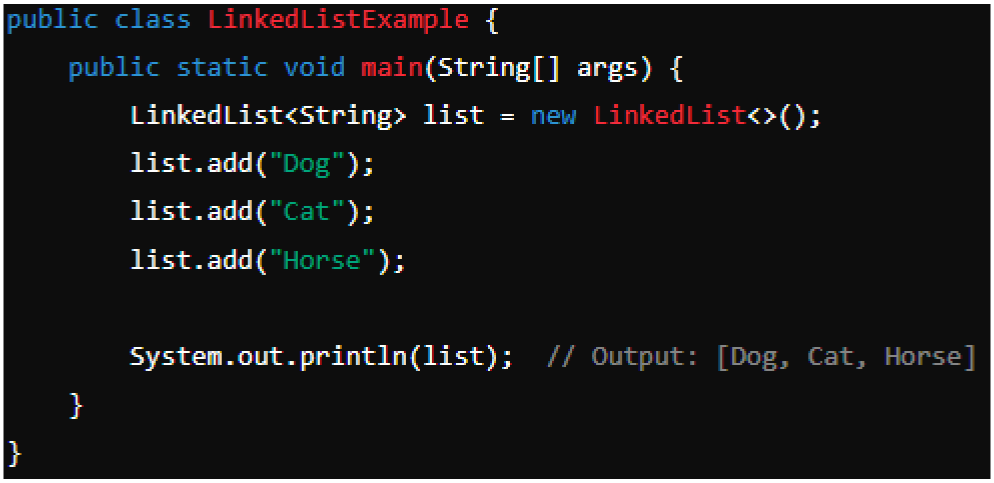
ARRAYLIST: ArrayList is an array-based implementation that allows quick access to elements, but is slower for inserting or removing elements in the middle of the list.



```
public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Orange");

        System.out.println(list);  // Output: [Apple, Banana, Orange]
    }
}
```

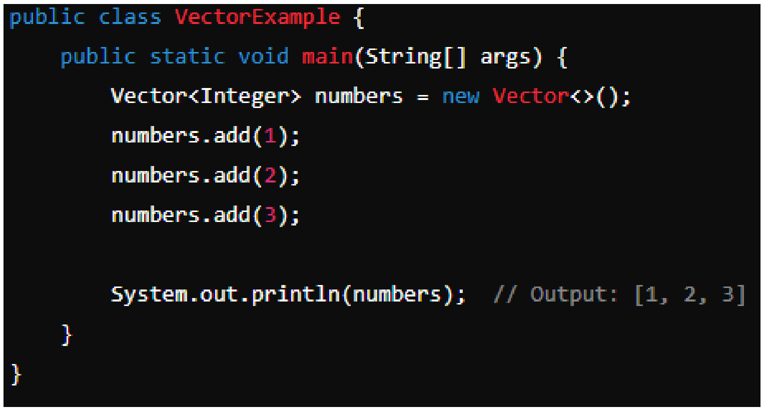
LINKEDLIST: LinkedList is a doubly-linked list implementation that is more efficient for insertions and removals at any position in the list but slower for accessing elements.



```
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.add("Dog");
        list.add("Cat");
        list.add("Horse");

        System.out.println(list);  // Output: [Dog, Cat, Horse]
    }
}
```

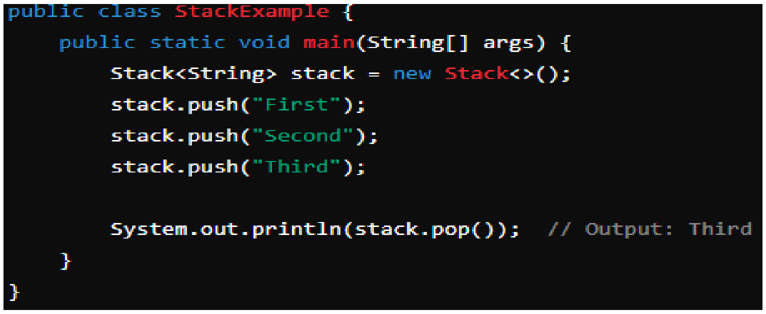
VECTOR: Vector is similar to ArrayList but synchronized, making it thread-safe but with a performance cost.



```java
public class VectorExample {
    public static void main(String[] args) {
        Vector<Integer> numbers = new Vector<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);

        System.out.println(numbers);  // Output: [1, 2, 3]
    }
}
```

STACK: Stack is a subclass of Vector that represents a stack (LIFO - Last In, First Out). It is ideal for scenarios where you need a stack data structure.

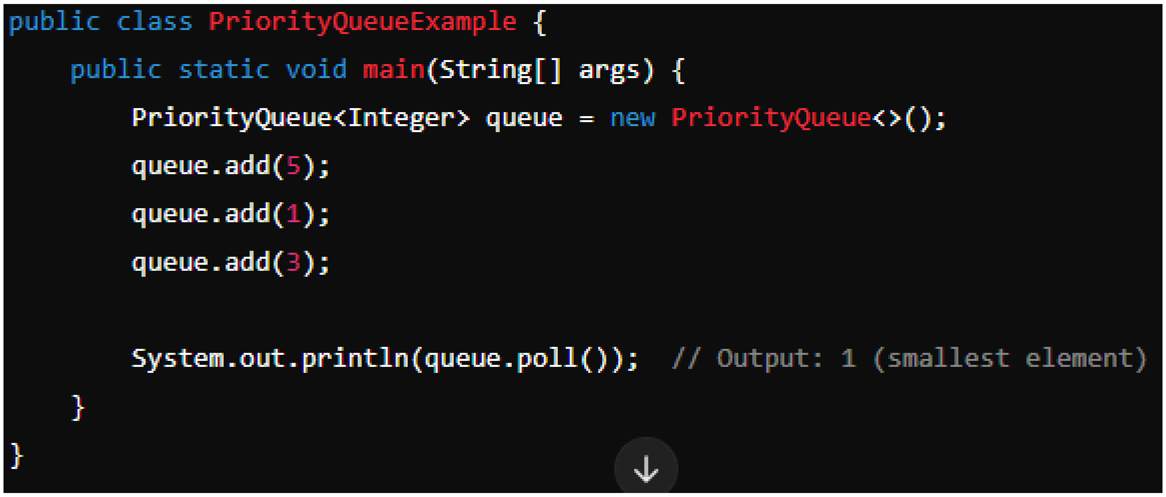


```java
public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("First");
        stack.push("Second");
        stack.push("Third");

        System.out.println(stack.pop());  // Output: Third
    }
}
```

QUEUE INTERFACE

PRIORITYQUEUE: PriorityQueue is a queue that automatically orders its elements based on their natural order or a provided comparator.

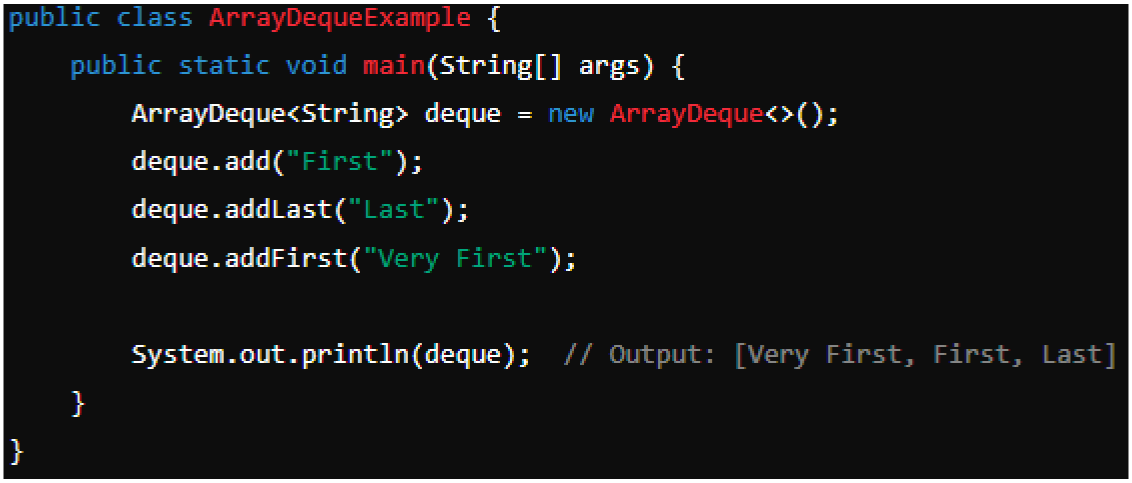


```java
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> queue = new PriorityQueue<>();
        queue.add(5);
        queue.add(1);
        queue.add(3);

        System.out.println(queue.poll());  // Output: 1 (smallest element)
    }
}
```

DEQUE INTERFACE

ARRAYDEQUE: is an array-based implementation of the Deque interface that allows insertion and removal at both ends of the queue. It is faster than Stack for stack operations.

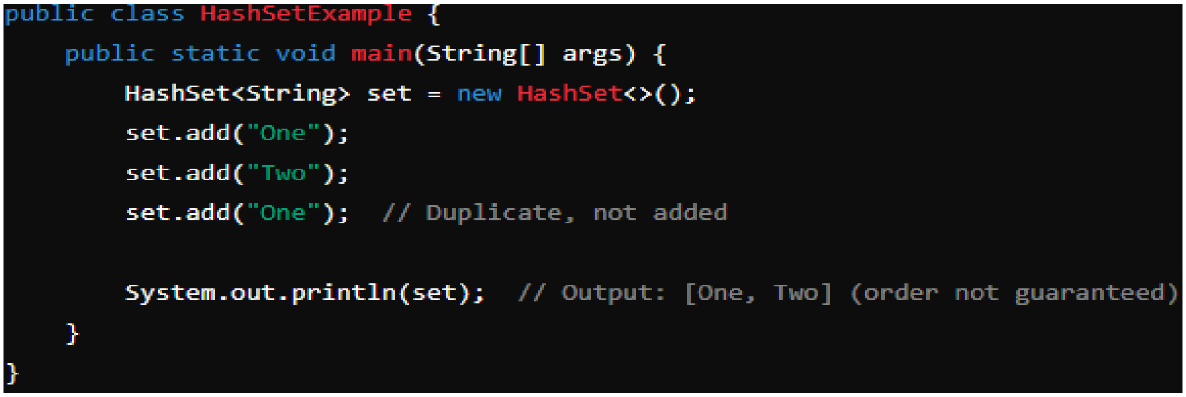


```java
public class ArrayDequeExample {
    public static void main(String[] args) {
        ArrayDeque<String> deque = new ArrayDeque<>();
        deque.add("First");
        deque.addLast("Last");
        deque.addFirst("Very First");

        System.out.println(deque);  // Output: [Very First, First, Last]
    }
}
```

SETINTERFACE

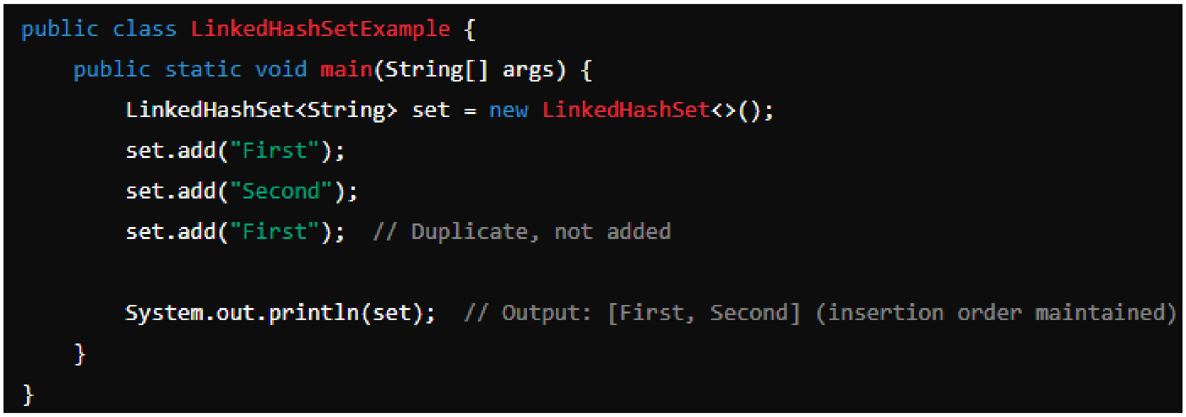
HASHSET: is an implementation of the Set interface based on a hash table. It does not guarantee the order of elements and does not allow duplicates.



```java
public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("One");
        set.add("Two");
        set.add("One");  // Duplicate, not added

        System.out.println(set);  // Output: [One, Two] (order not guaranteed)
    }
}
```

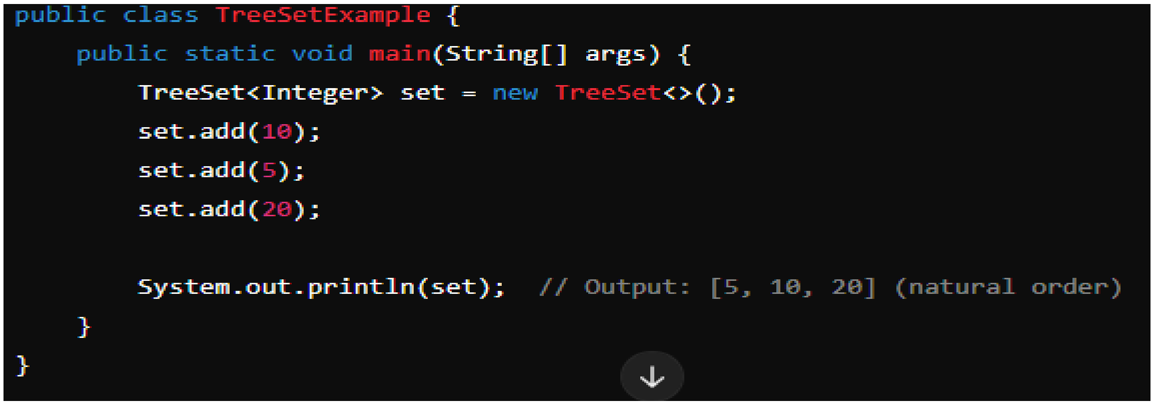
LINKEDHASHSET: is similar to HashSet but maintains the order of insertion of elements.



```java
public class LinkedHashSetExample {
    public static void main(String[] args) {
        LinkedHashSet<String> set = new LinkedHashSet<>();
        set.add("First");
        set.add("Second");
        set.add("First");  // Duplicate, not added

        System.out.println(set);  // Output: [First, Second] (insertion order maintained)
    }
}
```

TREESET: is an implementation of SortedSet that keeps elements sorted in natural order (or by a comparator). It is useful when you need a sorted set.

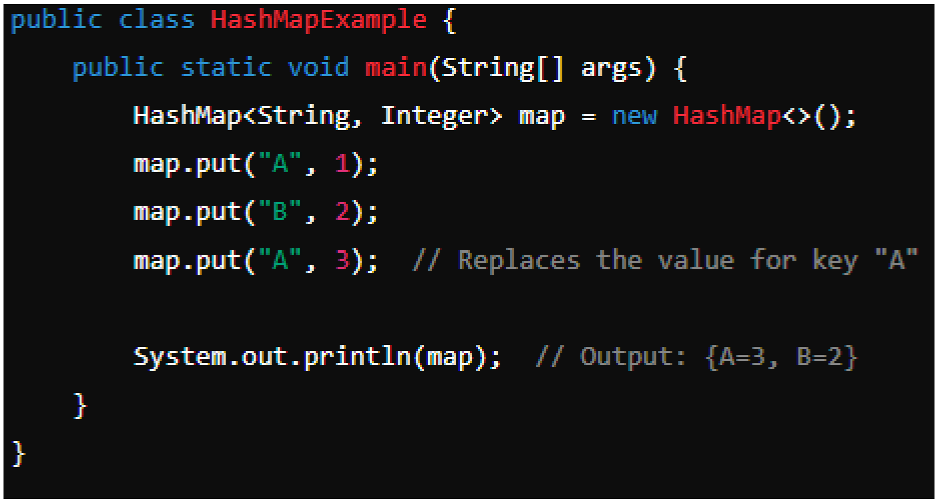


```
public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(10);
        set.add(5);
        set.add(20);

        System.out.println(set);  // Output: [5, 10, 20] (natural order)
    }
}
```

MAPINTERFACE

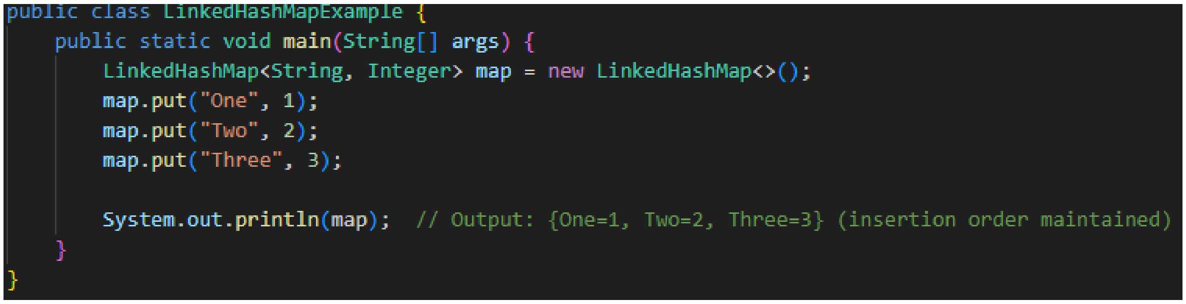
HASHMAP: is an implementation of Map based on hash tables. It allows null keys and values and does not guarantee the order of elements.



```
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("A", 1);
        map.put("B", 2);
        map.put("A", 3);  // Replaces the value for key "A"

        System.out.println(map);  // Output: {A=3, B=2}
    }
}
```

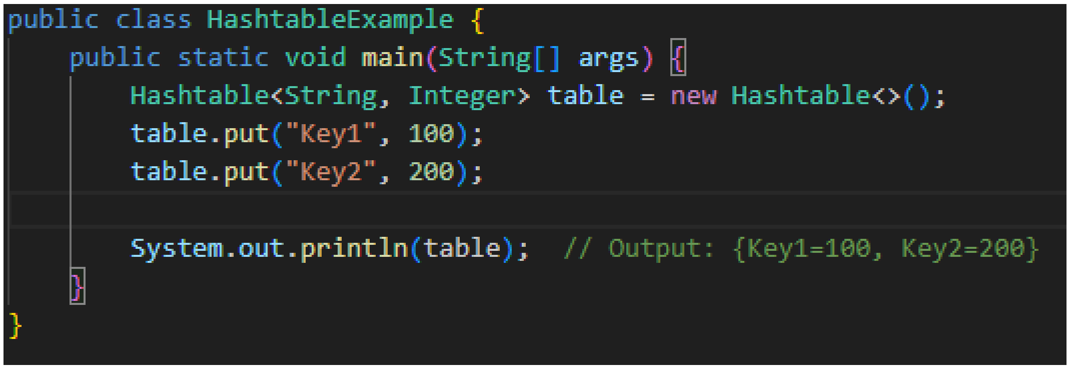
LINKEDHASMAP: similar to HashMap but maintains the order of insertion of elements, which can be useful when the order of elements is important.



```
public class LinkedHashMapExample {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
        map.put("One", 1);
        map.put("Two", 2);
        map.put("Three", 3);

        System.out.println(map);  // Output: {One=1, Two=2, Three=3} (insertion order maintained)
    }
}
```

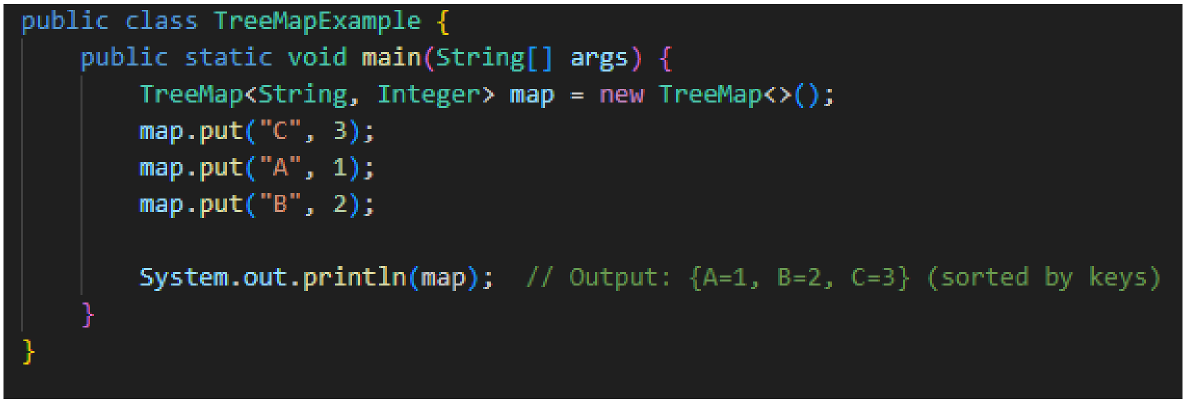
HASHTABLE: is a synchronized implementation of Map and does not allow null keys or values. It is an older class but still useful in multi-threaded applications.



```java
public class HashtableExample {
    public static void main(String[] args) {
        Hashtable<String, Integer> table = new Hashtable<>();
        table.put("Key1", 100);
        table.put("Key2", 200);

        System.out.println(table);  // Output: {Key1=100, Key2=200}
    }
}
```

TREEMAP: is an implementation of SortedMap that automatically orders keys in natural order or using a custom comparator.



```java
public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> map = new TreeMap<>();
        map.put("C", 3);
        map.put("A", 1);
        map.put("B", 2);

        System.out.println(map);  // Output: {A=1, B=2, C=3} (sorted by keys)
    }
}
```