import java.util.\*;

class Employee implements Comparable<Employee> {

private String name;

private double salary;

public Employee(String name, double salary) {

this.name = name;

this.salary = salary;

}

public String getName() {

return name;

}

public double getSalary() {

return salary;

}

@Override

public String toString() {

return "Employee{name='" + name + "', salary=" + salary + '}';

}

@Override

public int compareTo(Employee other) {

// Compare employees based on salary

return Double.compare(this.salary, other.salary);

}

}

public class collections {

public static void main(String args[]) {

System.out.println("\n\n ArrayList");

ArrayList<Employee> employeeList = new ArrayList<Employee>();

employeeList.add(new Employee("Sham", 50000));

employeeList.add(new Employee("Dev", 60000));

employeeList.add(new Employee("Joel", 55000));

Iterator itr = employeeList.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

System.out.println("\n\nLinkedLink");

LinkedList<Employee> al = new LinkedList<Employee>();

al.add(new Employee("John", 70000));

al.add(new Employee("Jane", 65000));

itr = al.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

System.out.println("\n\n Vector");

Vector<Employee> v = new Vector<Employee>();

v.add(new Employee("Ayush", 75000));

v.add(new Employee("Amit", 72000));

v.add(new Employee("Ashish", 78000));

itr = v.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

System.out.println("\n\n Stack");

Stack<Employee> stack = new Stack<Employee>();

stack.add(new Employee("John", 70000));

stack.add(new Employee("Jane", 65000));

stack.add(new Employee("Ayush", 75000));

stack.add(new Employee("Amit", 72000));

stack.add(new Employee("Ashish", 78000));

itr = stack.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

System.out.println("\n\n HashSet");

HashSet<Employee> set = new HashSet<Employee>();

set.add(new Employee("Jane", 65000));

set.add(new Employee("Ayush", 75000));

set.add(new Employee("Amit", 72000));

itr = set.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

System.out.println("\n\n LinkedHashSet");

LinkedHashSet<Employee> lset = new LinkedHashSet<Employee>();

lset.add(new Employee("John", 70000));

lset.add(new Employee("Jane", 65000));

lset.add(new Employee("Ayush", 75000));

itr = lset.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

System.out.println("\n\n TreeSet");

TreeSet<Employee> tset = new TreeSet<Employee>();

tset.add(new Employee("Sham", 50000));

tset.add(new Employee("Dev", 60000));

tset.add(new Employee("Joel", 55000));

itr = tset.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

System.out.println("\n\n QueueInterface");

PriorityQueue<Employee> queue = new PriorityQueue<Employee>();

queue.add(new Employee("Sham", 50000));

queue.add(new Employee("Dev", 60000));

queue.add(new Employee("Joel", 55000));

queue.add(new Employee("John", 70000));

queue.add(new Employee("Jane", 65000));

queue.add(new Employee("Ayush", 75000));

System.out.println("Head:" + queue.element());

System.out.println("head:" + queue.peek());

System.out.println("\nIterating the queue elements:");

itr = queue.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

queue.remove();

queue.poll();

System.out.println("\nAfter removing two elements:");

Iterator itr2 = queue.iterator();

while (itr2.hasNext()) {

System.out.println(itr2.next());

}

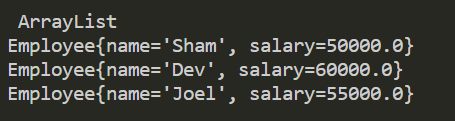
}

}

1. **ArrayList**

ArrayList is relevant in the employee salary management domain as it provides dynamic resizing and fast random access. Major advantages include:

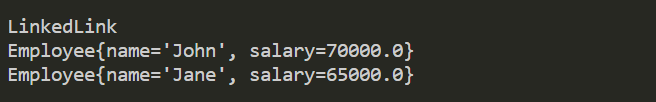
* Efficient for frequent access and traversal.
* Automatically resizes itself when the number of elements increases.
* Supports a variety of operations like add, remove, and get.



1. **LinkedList**

LinkedList is relevant for scenarios where frequent insertions and deletions are required. Major advantages include:

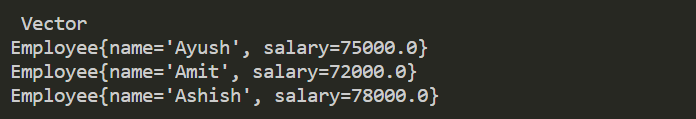
* Efficient for frequent insertions and deletions.
* Supports sequential access.
* Dynamic resizing.



1. **Vector**

Vector is relevant in the employee salary management domain for situations requiring synchronized operations and dynamic resizing. Major advantages include:

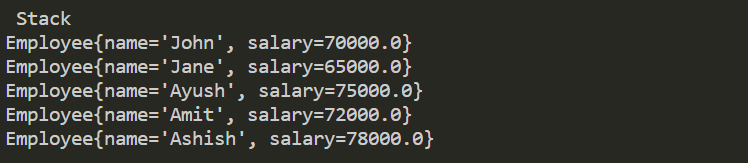
* Supports dynamic resizing, making it efficient for handling a variable number of elements.
* Provides synchronized methods, making it thread-safe.



1. **Stack**

Stack is relevant when you need to follow the Last In, First Out (LIFO) principle. Major advantages include:

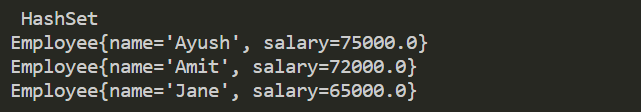
* Implements the Stack data structure, allowing operations like push (add) and pop (remove) in a last-in, first-out order.
* Provides methods for checking the top element without removal.



1. **HashSet**

HashSet is relevant in the employee salary management domain for efficient storage and retrieval of unique elements. Major advantages include:

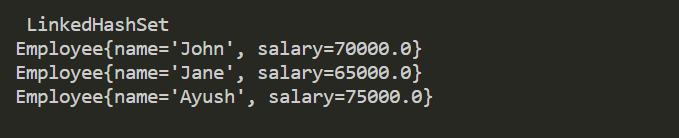
* Stores unique elements, eliminating duplicates.
* Offers constant-time performance for basic operations like add, remove, and contains.



1. **LinkedHashSet**

LinkedHashSet is relevant when you need to maintain the order of insertion in addition to the benefits of HashSet. Major advantages include:

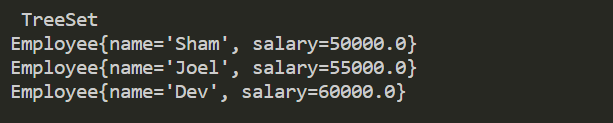
* Maintains the order of elements as they were inserted.
* Provides constant-time performance for basic operations.



1. **TreeSet**

TreeSet is relevant when you need elements sorted in a natural or custom order. Major advantages include:

* Automatically sorts elements based on their natural order or a specified comparator.
* Supports efficient retrieval, addition, and removal operations.



1. **PriorityQueue**

PriorityQueue is relevant when you need elements in a priority order. Major advantages include:

* Retrieves elements in priority order (based on natural order or a custom comparator).
* Provides efficient insertion and removal of the highest-priority element.

