**Exercise 1: HashSet**

1. Create a **HashSet** of strings and add the following elements: "apple", "banana", "cherry", "date", "fig".
2. Write code to check if the set contains "banana" and "grape" and print the results.

**Answer 1: HashSet**

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

HashSet<String> fruitSet = new HashSet<>();

fruitSet.add("apple");

fruitSet.add("banana");

fruitSet.add("cherry");

fruitSet.add("date");

fruitSet.add("fig");

System.out.println("Contains 'banana': " + fruitSet.contains("banana"));

System.out.println("Contains 'grape': " + fruitSet.contains("grape"));

}

}

**Exercise 2: LinkedHashSet**

1. Create a **LinkedHashSet** of integers and add the elements 5, 2, 8, 3, 1 in this order.
2. Iterate through the set and print the elements in the order they were added.

**Answer 2: LinkedHashSet**

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

public static void main(String[] args) {

LinkedHashSet<Integer> numberSet = new LinkedHashSet<>();

numberSet.add(5);

numberSet.add(2);

numberSet.add(8);

numberSet.add(3);

numberSet.add(1);

System.out.println("Elements in the LinkedHashSet:");

for (int num : numberSet) {

System.out.println(num);

}

}

}

**Exercise 3: TreeSet**

1. Create a **TreeSet** of strings and add the elements "cat", "dog", "elephant", "bat", "ant" in random order.
2. Iterate through the set and print the elements in ascending order.

**Answer 3: TreeSet**

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

TreeSet<String> animalSet = new TreeSet<>();

animalSet.add("cat");

animalSet.add("dog");

animalSet.add("elephant");

animalSet.add("bat");

animalSet.add("ant");

System.out.println("Elements in the TreeSet (in ascending order):");

for (String animal : animalSet) {

System.out.println(animal);

}

}

}

**Exercise 4: HashSet Operations**

1. Create two **HashSet** objects, **set1** and **set2**, with integer elements.
2. Write code to find and print the intersection and union of these sets.

**Answer 4: HashSet Operations**

import java.util.HashSet;

public class HashSetOperations {

public static void main(String[] args) {

HashSet<Integer> set1 = new HashSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

HashSet<Integer> set2 = new HashSet<>();

set2.add(2);

set2.add(3);

set2.add(4);

// Intersection

set1.retainAll(set2);

System.out.println("Intersection: " + set1);

// Union

set1.addAll(set2);

System.out.println("Union: " + set1);

}

}

**Exercise 5: LinkedHashSet with Custom Objects**

1. Create a **LinkedHashSet** of custom objects representing books with attributes (title, author, ISBN).
2. Add a few books to the set and iterate through them, printing their details.

**Answer 5: LinkedHashSet with Custom Objects**

import java.util.LinkedHashSet;

class Book {

private String title;

private String author;

private String ISBN;

public Book(String title, String author, String ISBN) {

this.title = title;

this.author = author;

this.ISBN = ISBN;

}

@Override

public String toString() {

return "Title: " + title + ", Author: " + author + ", ISBN: " + ISBN;

}

}

public class CustomObjectLinkedHashSet {

public static void main(String[] args) {

LinkedHashSet<Book> bookSet = new LinkedHashSet<>();

bookSet.add(new Book("Book1", "Author1", "ISBN1"));

bookSet.add(new Book("Book2", "Author2", "ISBN2"));

System.out.println("Books in the LinkedHashSet:");

for (Book book : bookSet) {

System.out.println(book);

}

}

}

**Exercise 6: TreeSet with Comparator**

1. Create a **TreeSet** of strings and use a custom **Comparator** to sort the strings in descending order.
2. Add some strings to the set and iterate through them to print them in descending order.

**Answer 6: TreeSet with Comparator**

import java.util.TreeSet;

import java.util.Comparator;

public class TreeSetWithComparator {

public static void main(String[] args) {

TreeSet<String> stringSet = new TreeSet<>(Comparator.reverseOrder());

stringSet.add("apple");

stringSet.add("banana");

stringSet.add("cherry");

stringSet.add("date");

System.out.println("Elements in the TreeSet (in descending order):");

for (String str : stringSet) {

System.out.println(str);

}

}

}

**Exercise 7: HashSet vs. LinkedHashSet vs. TreeSet Performance**

1. Create large **HashSet**, **LinkedHashSet**, and **TreeSet** objects containing integers.
2. Measure and compare the time it takes to add, remove, and search for elements in each set.

**Answer 7: HashSet vs. LinkedHashSet vs. TreeSet Performance (Sample Code)**

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.TreeSet;

import java.util.Random;

public class SetPerformanceComparison {

public static void main(String[] args) {

int numElements = 100000;

Random random = new Random();

// HashSet

HashSet<Integer> hashSet = new HashSet<>();

long startTime = System.nanoTime();

for (int i = 0; i < numElements; i++) {

hashSet.add(random.nextInt(numElements));

}

long endTime = System.nanoTime();

System.out.println("HashSet - Add Time: " + (endTime - startTime) + " ns");

// LinkedHashSet

LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>();

startTime = System.nanoTime();

for (int i = 0; i < numElements; i++) {

linkedHashSet.add(random.nextInt(numElements));

}

endTime = System.nanoTime();

System.out.println("LinkedHashSet - Add Time: " + (endTime - startTime) + " ns");

// TreeSet

TreeSet<Integer> treeSet = new TreeSet<>();

startTime = System.nanoTime();

for (int i = 0; i < numElements; i++) {

treeSet.add(random.nextInt(numElements));

}

endTime = System.nanoTime();

System.out.println("TreeSet - Add Time: " + (endTime - startTime) + " ns");

}

}

**Exercise 8: Removing Duplicates from an ArrayList using HashSet**

1. Create an **ArrayList** of integers with duplicate values.
2. Use a **HashSet** to remove duplicates from the **ArrayList**.

**Answer 8: Removing Duplicates from an ArrayList using HashSet**

import java.util.ArrayList;

import java.util.HashSet;

public class RemoveDuplicatesUsingHashSet {

public static void main(String[] args) {

ArrayList<Integer> numbersList = new ArrayList<>();

numbersList.add(1);

numbersList.add(2);

numbersList.add(2);

numbersList.add(3);

numbersList.add(4);

numbersList.add(4);

System.out.println("Original ArrayList: " + numbersList);

HashSet<Integer> uniqueNumbers = new HashSet<>(numbersList);

numbersList.clear();

numbersList.addAll(uniqueNumbers);

System.out.println("ArrayList after removing duplicates: " + numbersList);

}

}

**Exercise 9: Using LinkedHashSet to Maintain Insertion Order**

1. Create a **LinkedHashSet** and add a series of names in a specific order.
2. Print the names to demonstrate that the insertion order is maintained.

**Answer 9: Using LinkedHashSet to Maintain Insertion Order**

import java.util.LinkedHashSet;

public class MaintainInsertionOrderWithLinkedHashSet {

public static void main(String[] args) {

LinkedHashSet<String> namesSet = new LinkedHashSet<>();

namesSet.add("Alice");

namesSet.add("Bob");

namesSet.add("Charlie");

namesSet.add("David");

System.out.println("Names in LinkedHashSet (Maintaining Insertion Order):");

for (String name : namesSet) {

System.out.println(name);

}

}

}

**Exercise 10: TreeSet and Custom Object Ordering**

1. Create a **TreeSet** of custom objects representing students with attributes (name, age).
2. Implement a custom **Comparator** to sort the students by age in ascending order.
3. Add some student objects to the set and iterate through them to print their details in ascending order of age.

**Answer 10: TreeSet and Custom Object Ordering**

import java.util.TreeSet;

import java.util.Comparator;

class Student {

private String name;

private int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

public int getAge() {

return age;

}

@Override

public String toString() {

return "Name: " + name + ", Age: " + age;

}

}

public class CustomObjectTreeSetWithComparator {

public static void main(String[] args) {

TreeSet<Student> studentSet = new TreeSet<>(Comparator.comparingInt(Student::getAge));

studentSet.add(new Student("Alice", 20));

studentSet.add(new Student("Bob", 22));

studentSet.add(new Student("Charlie", 19));

System.out.println("Students in the TreeSet (in ascending order of age):");

for (Student student : studentSet) {

System.out.println(student);

}

}

}

**Exercise 11: HashSet vs. LinkedHashSet vs. TreeSet for Performance**

1. Create large **HashSet**, **LinkedHashSet**, and **TreeSet** objects containing integers.
2. Measure and compare the time it takes to search for a specific element in each set.

**Answer 11: HashSet vs. LinkedHashSet vs. TreeSet for Performance (Sample Code)**

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.TreeSet;

import java.util.Random;

public class SetSearchPerformanceComparison {

public static void main(String[] args) {

int numElements = 100000;

Random random = new Random();

HashSet<Integer> hashSet = new HashSet<>();

LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>();

TreeSet<Integer> treeSet = new TreeSet<>();

for (int i = 0; i < numElements; i++) {

int element = random.nextInt(numElements);

hashSet.add(element);

linkedHashSet.add(element);

treeSet.add(element);

}

int searchElement = random.nextInt(numElements);

// Measure search time for HashSet

long startTime = System.nanoTime();

boolean foundInHashSet = hashSet.contains(searchElement);

long endTime = System.nanoTime();

System.out.println("HashSet - Search Time: " + (endTime - startTime) + " ns");

// Measure search time for LinkedHashSet

startTime = System.nanoTime();

boolean foundInLinkedHashSet = linkedHashSet.contains(searchElement);

endTime = System.nanoTime();

System.out.println("LinkedHashSet - Search Time: " + (endTime - startTime) + " ns");

// Measure search time for TreeSet

startTime = System.nanoTime();

boolean foundInTreeSet = treeSet.contains(searchElement);

endTime = System.nanoTime();

System.out.println("TreeSet - Search Time: " + (endTime - startTime) + " ns");

}

}

**Exercise 12: LinkedHashSet for LRU Cache**

1. Implement an LRU (Least Recently Used) cache using a **LinkedHashSet** with a maximum capacity.
2. Add methods for adding and getting items from the cache while maintaining the LRU policy.

**Answer 12: LinkedHashSet for LRU Cache (Sample Code)**

import java.util.LinkedHashSet;

class LRUCache<K> {

private int capacity;

private LinkedHashSet<K> cache;

public LRUCache(int capacity) {

this.capacity = capacity;

this.cache = new LinkedHashSet<>(capacity);

}

public void put(K key) {

if (cache.size() >= capacity) {

// Remove the least recently used item

K first = cache.iterator().next();

cache.remove(first);

}

cache.add(key);

}

public boolean contains(K key) {

return cache.contains(key);

}

public void printCache() {

System.out.println("LRU Cache: " + cache);

}

}

public class LRUCacheExample {

public static void main(String[] args) {

LRUCache<Integer> lruCache = new LRUCache<>(3);

lruCache.put(1);

lruCache.put(2);

lruCache.put(3);

lruCache.printCache(); // [1, 2, 3]

lruCache.put(4);

lruCache.printCache(); // [2, 3, 4]

System.out.println("Contains 2: " + lruCache.contains(2)); // true

System.out.println("Contains 5: " + lruCache.contains(5)); // false

}

}

**Exercise 13: TreeSet and Subsets**

1. Create a **TreeSet** of integers and add some random numbers.
2. Find and print a subset of numbers within a specific range, e.g., between 20 and 50.

**Answer 13: TreeSet and Subsets**

import java.util.TreeSet;

import java.util.NavigableSet;

public class TreeSetSubsets {

public static void main(String[] args) {

TreeSet<Integer> numberSet = new TreeSet<>();

numberSet.add(10);

numberSet.add(30);

numberSet.add(25);

numberSet.add(50);

numberSet.add(15);

System.out.println("Original TreeSet: " + numberSet);

NavigableSet<Integer> subset = numberSet.subSet(20, true, 50, false);

System.out.println("Subset (20 <= x < 50): " + subset);

}

}

**Exercise 14: Removing Elements from HashSet**

1. Create a **HashSet** of strings and add several elements.
2. Remove all elements starting with the letter "A".

**Answer 14: Removing Elements from HashSet**

import java.util.HashSet;

import java.util.Iterator;

public class RemoveElementsFromHashSet {

public static void main(String[] args) {

HashSet<String> stringSet = new HashSet<>();

stringSet.add("Apple");

stringSet.add("Banana");

stringSet.add("Avocado");

stringSet.add("Orange");

System.out.println("Original HashSet: " + stringSet);

Iterator<String> iterator = stringSet.iterator();

while (iterator.hasNext()) {

String element = iterator.next();

if (element.startsWith("A")) {

iterator.remove();

}

}

System.out.println("HashSet after removing elements starting with 'A': " + stringSet);

}

}

**Exercise 15: Checking If Sets are Equal**

1. Create two **HashSet** objects, **set1** and **set2**, with integer elements.
2. Write code to check if the two sets are equal (contain the same elements).

**Answer 15: Checking If Sets are Equal**

import java.util.HashSet;

public class SetEqualityCheck {

public static void main(String[] args) {

HashSet<Integer> set1 = new HashSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

HashSet<Integer> set2 = new HashSet<>();

set2.add(3);

set2.add(2);

set2.add(1);

boolean areEqual = set1.equals(set2);

System.out.println("Are set1 and set2 equal? " + areEqual);

}

}

**Exercise 16: LinkedHashSet for Maintaining Order of User Actions**

1. Create a **LinkedHashSet** to store user actions (e.g., "Login", "Browse", "Logout") in the order they occur.
2. Simulate user actions and add them to the set.
3. Print the user actions in the order they were performed.

**Answer 16: LinkedHashSet for Maintaining Order of User Actions**

import java.util.LinkedHashSet;

public class UserActionsLog {

public static void main(String[] args) {

LinkedHashSet<String> userActions = new LinkedHashSet<>();

// Simulate user actions

userActions.add("Login");

userActions.add("Browse");

userActions.add("Search");

userActions.add("Logout");

System.out.println("User Actions in Order:");

for (String action : userActions) {

System.out.println(action);

}

}

}

**Exercise 17: TreeSet and Subtraction of Sets**

1. Create two **TreeSet** objects, **set1** and **set2**, with integer elements.
2. Find and print the elements that are in **set1** but not in **set2**.

**Answer 17: TreeSet and Subtraction of Sets**

import java.util.TreeSet;

public class TreeSetSetSubtraction {

public static void main(String[] args) {

TreeSet<Integer> set1 = new TreeSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

set1.add(4);

TreeSet<Integer> set2 = new TreeSet<>();

set2.add(2);

set2.add(4);

set1.removeAll(set2);

System.out.println("Elements in set1 but not in set2: " + set1);

}

}

**Exercise 18: LinkedHashSet for Maintaining Recent Items**

1. Create a **LinkedHashSet** to maintain a list of recently viewed items.
2. Add items in a specific order, and ensure that only the most recent items are retained (limit the capacity).

**Answer 18: LinkedHashSet for Maintaining Recent Items**

import java.util.LinkedHashSet;

public class RecentItemsList {

public static void main(String[] args) {

LinkedHashSet<String> recentItems = new LinkedHashSet<>(5);

// Simulate viewing items

recentItems.add("Item1");

recentItems.add("Item2");

recentItems.add("Item3");

recentItems.add("Item4");

recentItems.add("Item5");

recentItems.add("Item6");

// Limit the capacity to 5 items

if (recentItems.size() > 5) {

recentItems.remove(recentItems.iterator().next());

}

System.out.println("Recent Items (Most recent first): " + recentItems);

}

}

**Exercise 19: HashSet Intersection with Common Elements**

1. Create two **HashSet** objects, **set1** and **set2**, with integer elements.
2. Write code to find and print the elements that are common to both sets.

**Answer 19: HashSet Intersection with Common Elements**

import java.util.HashSet;

public class HashSetIntersection {

public static void main(String[] args) {

HashSet<Integer> set1 = new HashSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

HashSet<Integer> set2 = new HashSet<>();

set2.add(3);

set2.add(4);

set2.add(5);

set1.retainAll(set2);

System.out.println("Common elements in set1 and set2: " + set1);

}

}

**Exercise 20: TreeSet and Higher Elements**

1. Create a **TreeSet** of integers and add some random numbers.
2. Find and print the elements that are higher (greater) than a specific number, e.g., greater than 30.

**Answer 20: TreeSet and Higher Elements**

import java.util.TreeSet;

public class TreeSetHigherElements {

public static void main(String[] args) {

TreeSet<Integer> numberSet = new TreeSet<>();

numberSet.add(10);

numberSet.add(30);

numberSet.add(25);

numberSet.add(50);

numberSet.add(15);

System.out.println("Original TreeSet: " + numberSet);

TreeSet<Integer> higherElements = new TreeSet<>(numberSet.tailSet(30, false));

System.out.println("Elements higher than 30: " + higherElements);

}

}

**Exercise 21: HashSet with User-Defined Class**

1. Create a **HashSet** to store a collection of user-defined objects (e.g., a **Person** class).
2. Add several **Person** objects to the set and demonstrate the ability to check for the existence of a specific person in the set.

**Answer 21: HashSet with User-Defined Class**

import java.util.HashSet;

class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int hashCode() {

return name.hashCode();

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Person person = (Person) obj;

return age == person.age && name.equals(person.name);

}

}

public class HashSetWithUserDefinedClass {

public static void main(String[] args) {

HashSet<Person> personSet = new HashSet<>();

personSet.add(new Person("Alice", 25));

personSet.add(new Person("Bob", 30));

personSet.add(new Person("Charlie", 22));

Person searchPerson = new Person("Alice", 25);

System.out.println("Does the set contain " + searchPerson.getName() + "? " + personSet.contains(searchPerson));

}

}

**Exercise 22: LinkedHashSet with User-Defined Class**

1. Create a **LinkedHashSet** to store a collection of user-defined objects (e.g., a **Product** class) to maintain insertion order.
2. Add several **Product** objects to the set and iterate through them to display product details in the order they were added.

**Answer 22: LinkedHashSet with User-Defined Class**

import java.util.LinkedHashSet;

class Product {

private String name;

private double price;

public Product(String name, double price) {

this.name = name;

this.price = price;

}

@Override

public String toString() {

return "Product{name='" + name + "', price=" + price + '}';

}

}

public class LinkedHashSetWithUserDefinedClass {

public static void main(String[] args) {

LinkedHashSet<Product> productSet = new LinkedHashSet<>();

productSet.add(new Product("Laptop", 999.99));

productSet.add(new Product("Smartphone", 599.99));

productSet.add(new Product("Tablet", 349.99));

System.out.println("Products in LinkedHashSet (Maintaining Insertion Order):");

for (Product product : productSet) {

System.out.println(product);

}

}

}

**Exercise 23: TreeSet with User-Defined Class and Custom Sorting**

1. Create a **TreeSet** to store a collection of user-defined objects (e.g., a **Book** class) and implement a custom **Comparator** to sort the books by title in alphabetical order.
2. Add several **Book** objects to the set and iterate through them to display the books in alphabetical order by title.

**Answer 23: TreeSet with User-Defined Class and Custom Sorting**

import java.util.TreeSet;

import java.util.Comparator;

class Book {

private String title;

private String author;

public Book(String title, String author) {

this.title = title;

this.author = author;

}

public String getTitle() {

return title;

}

@Override

public String toString() {

return "Book{title='" + title + "', author='" + author + "'}";

}

}

public class TreeSetWithUserDefinedClass {

public static void main(String[] args) {

TreeSet<Book> bookSet = new TreeSet<>(Comparator.comparing(Book::getTitle));

bookSet.add(new Book("The Great Gatsby", "F. Scott Fitzgerald"));

bookSet.add(new Book("To Kill a Mockingbird", "Harper Lee"));

bookSet.add(new Book("1984", "George Orwell"));

System.out.println("Books in TreeSet (Sorted by Title):");

for (Book book : bookSet) {

System.out.println(book);

}

}

}

**Exercise 24: HashSet with User-Defined Class and Custom Equality**

1. Create a **HashSet** to store a collection of user-defined objects (e.g., a **Song** class).
2. Implement custom equality for the **Song** class by comparing the song title and artist.
3. Add several **Song** objects to the set and demonstrate the ability to check for the existence of a specific song in the set.

**Answer 24: HashSet with User-Defined Class and Custom Equality**

import java.util.HashSet;

class Song {

private String title;

private String artist;

public Song(String title, String artist) {

this.title = title;

this.artist = artist;

}

@Override

public int hashCode() {

return title.hashCode() + artist.hashCode();

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Song song = (Song) obj;

return title.equals(song.title) && artist.equals(song.artist);

}

public String getTitle() {

return title;

}

public String getArtist() {

return artist;

}

}

public class HashSetWithUserDefinedClassCustomEquality {

public static void main(String[] args) {

HashSet<Song> songSet = new HashSet<>();

songSet.add(new Song("Imagine", "John Lennon"));

songSet.add(new Song("Bohemian Rhapsody", "Queen"));

songSet.add(new Song("Imagine", "John Lennon"));

Song searchSong = new Song("Bohemian Rhapsody", "Queen");

System.out.println("Does the set contain " + searchSong.getTitle() + " by " + searchSong.getArtist() + "? " + songSet.contains(searchSong));

}

}

**Exercise 25: LinkedHashSet with User-Defined Class and Maintain Order of Events**

1. Create a **LinkedHashSet** to store a collection of user-defined objects (e.g., an **Event** class) to maintain the order of events.
2. Add several **Event** objects to the set and demonstrate that the order in which the events were added is preserved.

**Answer 25: LinkedHashSet with User-Defined Class and Maintain Order of Events**

import java.util.LinkedHashSet;

class Event {

private String name;

private String date;

public Event(String name, String date) {

this.name = name;

this.date = date;

}

@Override

public String toString() {

return "Event{name='" + name + "', date='" + date + "'}";

}

}

public class LinkedHashSetWithUserDefinedClassOrderOfEvents {

public static void main(String[] args) {

LinkedHashSet<Event> eventSet = new LinkedHashSet<>();

eventSet.add(new Event("Conference", "2023-01-15"));

eventSet.add(new Event("Workshop", "2023-02-20"));

eventSet.add(new Event("Meeting", "2023-03-10"));

System.out.println("Events in LinkedHashSet (Maintaining Order):");

for (Event event : eventSet) {

System.out.println(event);

}

}

}

**Exercise 26: TreeSet with User-Defined Class and Custom Sorting**

1. Create a **TreeSet** to store a collection of user-defined objects (e.g., a **Employee** class) and implement a custom **Comparator** to sort employees by their ID in ascending order.
2. Add several **Employee** objects to the set and demonstrate that employees are sorted by their ID.

**Answer 26: TreeSet with User-Defined Class and Custom Sorting**

import java.util.TreeSet;

import java.util.Comparator;

class Employee {

private int employeeId;

private String name;

public Employee(int employeeId, String name) {

this.employeeId = employeeId;

this.name = name;

}

public int getEmployeeId() {

return employeeId;

}

@Override

public String toString() {

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

}

}

public class TreeSetWithUserDefinedClassCustomSorting {

public static void main(String[] args) {

TreeSet<Employee> employeeSet = new TreeSet<>(Comparator.comparingInt(Employee::getEmployeeId));

employeeSet.add(new Employee(101, "Alice"));

employeeSet.add(new Employee(103, "Bob"));

employeeSet.add(new Employee(102, "Charlie"));

System.out.println("Employees in TreeSet (Sorted by Employee ID):");

for (Employee employee : employeeSet) {

System.out.println(employee);

}

}

}

**Exercise 27: HashSet with User-Defined Class and Custom Equality**

1. Create a **HashSet** to store a collection of user-defined objects (e.g., a **Product** class).
2. Implement custom equality for the **Product** class by comparing the product name and category.
3. Add several **Product** objects to the set and demonstrate the ability to check for the existence of a specific product in the set.

**Answer 27: HashSet with User-Defined Class and Custom Equality**

import java.util.HashSet;

class Product {

private String name;

private String category;

public Product(String name, String category) {

this.name = name;

this.category = category;

}

@Override

public int hashCode() {

return name.hashCode() + category.hashCode();

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Product product = (Product) obj;

return name.equals(product.name) && category.equals(product.category);

}

public String getName() {

return name;

}

public String getCategory() {

return category;

}

}

public class HashSetWithUserDefinedClassCustomEqualityPart2 {

public static void main(String[] args) {

HashSet<Product> productSet = new HashSet<>();

productSet.add(new Product("Laptop", "Electronics"));

productSet.add(new Product("Smartphone", "Electronics"));

productSet.add(new Product("Laptop", "Electronics"));

Product searchProduct = new Product("Smartphone", "Electronics");

System.out.println("Does the set contain " + searchProduct.getName() + " in the " + searchProduct.getCategory() + " category? " + productSet.contains(searchProduct));

}

}

**Exercise 28: LinkedHashSet with User-Defined Class and Maintain Order of Transactions**

1. Create a **LinkedHashSet** to store a collection of user-defined objects (e.g., a **Transaction** class) to maintain the order of transactions.
2. Add several **Transaction** objects to the set and demonstrate that the order in which the transactions were added is preserved.

**Answer 28: LinkedHashSet with User-Defined Class and Maintain Order of Transactions**

import java.util.LinkedHashSet;

class Transaction {

private String transactionId;

private double amount;

public Transaction(String transactionId, double amount) {

this.transactionId = transactionId;

this.amount = amount;

}

@Override

public String toString() {

return "Transaction{transactionId='" + transactionId + "', amount=" + amount + '}';

}

}

public class LinkedHashSetWithUserDefinedClassOrderOfTransactions {

public static void main(String[] args) {

LinkedHashSet<Transaction> transactionSet = new LinkedHashSet<>();

transactionSet.add(new Transaction("T123", 500.0));

transactionSet.add(new Transaction("T124", 300.0));

transactionSet.add(new Transaction("T125", 200.0));

System.out.println("Transactions in LinkedHashSet (Maintaining Order):");

for (Transaction transaction : transactionSet) {

System.out.println(transaction);

}

}

}

**Exercise 29: TreeSet with User-Defined Class and Custom Sorting**

1. Create a **TreeSet** to store a collection of user-defined objects (e.g., a **Student** class) and implement a custom **Comparator** to sort students by their age in descending order.
2. Add several **Student** objects to the set and demonstrate that students are sorted by age in descending order.

**Answer 29: TreeSet with User-Defined Class and Custom Sorting**

import java.util.TreeSet;

import java.util.Comparator;

class Student {

private String name;

private int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

public int getAge() {

return age;

}

@Override

public String toString() {

return "Student{name='" + name + "', age=" + age + '}';

}

}

public class TreeSetWithUserDefinedClassCustomSortingPart2 {

public static void main(String[] args) {

TreeSet<Student> studentSet = new TreeSet<>(Comparator.comparingInt(Student::getAge).reversed());

studentSet.add(new Student("Alice", 25));

studentSet.add(new Student("Bob", 30));

studentSet.add(new Student("Charlie", 22));

System.out.println("Students in TreeSet (Sorted by Age in Descending Order):");

for (Student student : studentSet) {

System.out.println(student);

}

}

}

**Exercise 30: HashSet with User-Defined Class**

1. Create a **HashSet** to store a collection of user-defined objects (e.g., a **Student** class).
2. Implement custom equality for the **Student** class by comparing the student's ID and name.
3. Add several **Student** objects to the set.
4. Demonstrate the ability to check for the existence of a specific student in the set.

**Answer 30: HashSet with User-Defined Class and Custom Sorting**

import java.util.HashSet;

class Student {

private int studentId;

private String name;

public Student(int studentId, String name) {

this.studentId = studentId;

this.name = name;

}

@Override

public int hashCode() {

return studentId;

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Student student = (Student) obj;

return studentId == student.studentId && name.equals(student.name);

}

public int getStudentId() {

return studentId;

}

public String getName() {

return name;

}

}

public class HashSetWithUserDefinedClass {

public static void main(String[] args) {

HashSet<Student> studentSet = new HashSet<>();

studentSet.add(new Student(101, "Alice"));

studentSet.add(new Student(102, "Bob"));

studentSet.add(new Student(103, "Charlie"));

Student searchStudent = new Student(102, "Bob");

System.out.println("Does the set contain student " + searchStudent.getName() + " with ID " + searchStudent.getStudentId() + "? " + studentSet.contains(searchStudent));

}

}