

SET B 1. Write a Java program to perform a sequential search on an array of integers. The program defines an array `numbers` containing the values {10, 20, 30, 40, 50} and a target value of 30. It then calls a method `sequentialSearch` that iterates through the array to find the target value. If the target value is found, the program outputs the index at which it is located; otherwise, it outputs that the element is not found. Analyze the provided program and explain how the sequential search algorithm works. Additionally, modify the program to handle a larger array of integers and search for a different target value.

Here's a Java program that performs a sequential search on an array of integers. The program defines an array `numbers` containing the values {10, 20, 30, 40, 50} and a target value of 30. It then calls a method `sequentialSearch` that iterates through the array to find the target value. If the target value is found, the program outputs the index at which it is located; otherwise, it outputs that the element is not found.

```
java
Copy code
public class SequentialSearch {

    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int target = 30;
        int index = sequentialSearch(numbers, target);

        if (index != -1) {
            System.out.println("Element " + target + " is found at index: " +
index);
        } else {
            System.out.println("Element " + target + " is not found.");
        }
    }

    public static int sequentialSearch(int[] array, int target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == target) {
                return i;
            }
        }
        return -1;
    }
}
```

## Analysis of the Sequential Search Algorithm

Sequential search, also known as linear search, is a simple search algorithm that checks every element in an array sequentially until the target value is found or the end of the array is reached.

Here's a step-by-step explanation of how it works:

1. **Initialization:** The method `sequentialSearch` takes an array and a target value as input parameters.
2. **Iteration:** A `for` loop iterates through each element of the array.
3. **Comparison:** For each element in the array, the algorithm compares the current element with the target value.

4. **Match Found:** If a match is found (i.e., `array[i] == target`), the method returns the index `i`.
5. **End of Array:** If the loop completes without finding the target value, the method returns `-1`, indicating that the target value is not in the array.

## Modified Program for a Larger Array and Different Target Value

Here's the modified program that handles a larger array of integers and searches for a different target value:

java

Copy code

```
public class SequentialSearch {

    public static void main(String[] args) {
        int[] numbers = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70,
75, 80, 85, 90, 95, 100};
        int target = 75;
        int index = sequentialSearch(numbers, target);

        if (index != -1) {
            System.out.println("Element " + target + " is found at index: " +
index);
        } else {
            System.out.println("Element " + target + " is not found.");
        }
    }

    public static int sequentialSearch(int[] array, int target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == target) {
                return i;
            }
        }
        return -1;
    }
}
```

In this modified program:

- The `numbers` array contains 20 integers ranging from 5 to 100.
- The `target` value is changed to 75.
- The `sequentialSearch` method is used in the same way to find the target value in the larger array.

40

2. Write a Java program that implements the binary search algorithm to find a specific number in a sorted array of integers. The program should define a sorted array and a target value to search for. Ensure the program handles cases where the target value is not present in the array by returning an appropriate message.

Here's a Java program that implements the binary search algorithm to find a specific number in a sorted array of integers. The program defines a sorted array and a target value to search for. It also handles cases where the target value is not present in the array by returning an appropriate message.

java

Copy code

```

public class BinarySearch {

    public static void main(String[] args) {
        int[] numbers = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70,
75, 80, 85, 90, 95, 100};
        int target = 75;
        int index = binarySearch(numbers, target);

        if (index != -1) {
            System.out.println("Element " + target + " is found at index: " +
index);
        } else {
            System.out.println("Element " + target + " is not found.");
        }
    }

    public static int binarySearch(int[] array, int target) {
        int left = 0;
        int right = array.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Check if the target is present at mid
            if (array[mid] == target) {
                return mid;
            }

            // If the target is greater, ignore the left half
            if (array[mid] < target) {
                left = mid + 1;
            } else {
                // If the target is smaller, ignore the right half
                right = mid - 1;
            }
        }

        // Target is not present in the array
        return -1;
    }
}

```

## Explanation of the Binary Search Algorithm

Binary search is an efficient algorithm for finding an item from a sorted array of integers. It works by repeatedly dividing in half the portion of the array that could contain the target value, until the possible locations are reduced to zero. Here is a step-by-step explanation:

1. **Initialization:** The method `binarySearch` takes a sorted array and a target value as input parameters. It initializes two pointers, `left` and `right`, to the start and end of the array, respectively.
2. **Midpoint Calculation:** In each iteration, the midpoint (`mid`) is calculated as `left + (right - left) / 2` to prevent potential overflow.
3. **Comparison:** The algorithm compares the value at the midpoint with the target value.
  - If `array[mid]` equals the target, the midpoint index is returned.

- If `array[mid]` is less than the target, the left pointer is updated to `mid + 1`, ignoring the left half.
  - If `array[mid]` is greater than the target, the right pointer is updated to `mid - 1`, ignoring the right half.
4. **Termination:** The loop continues until the left pointer exceeds the right pointer. If the target value is not found, the method returns `-1`.

This algorithm efficiently reduces the search space by half in each step, making it much faster than sequential search for large arrays.

40

3. Write a Java program that sorts a list of numbers using the Selection Sort algorithm, a quadratic sorting technique. The program should define an array of unsorted integers and include a method that performs the selection sort. This method should iterate through the array to find the minimum element in the unsorted portion and swap it with the first unsorted element, continuing this process until the entire array is sorted. The program should also include a utility method to print the array before and after sorting.

Here's a Java program that sorts a list of numbers using the Selection Sort algorithm. The program defines an array of unsorted integers and includes a method that performs the selection sort. It also includes a utility method to print the array before and after sorting.

java

Copy code

```
public class SelectionSort {

    public static void main(String[] args) {
        int[] numbers = {64, 25, 12, 22, 11};

        System.out.println("Unsorted array:");
        printArray(numbers);

        selectionSort(numbers);

        System.out.println("Sorted array:");
        printArray(numbers);
    }

    public static void selectionSort(int[] array) {
        int n = array.length;

        for (int i = 0; i < n - 1; i++) {
            // Find the minimum element in the unsorted portion
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first element
            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }

    public static void printArray(int[] array) {
```

```

        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
        }
        System.out.println();
    }
}

```

## Explanation of the Selection Sort Algorithm

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly finding the minimum element from the unsorted portion of the array and swapping it with the first unsorted element. Here's a step-by-step explanation:

1. **Initialization:** The method `selectionSort` takes an array as an input parameter. The length of the array is stored in the variable `n`.
2. **Outer Loop:** The outer loop iterates over each element of the array except the last one (`i` ranges from 0 to `n - 2`), as the last element will be in place after the second last iteration.
3. **Finding the Minimum:** For each position `i`, the algorithm initializes `minIndex` to `i`. Then, it uses an inner loop to find the minimum element in the remaining unsorted portion of the array (from `i + 1` to `n - 1`). If a smaller element is found, `minIndex` is updated.
4. **Swapping:** After finding the minimum element in the unsorted portion, it swaps this element with the element at position `i`. This places the minimum element in its correct position in the sorted portion of the array.
5. **Utility Method:** The `printArray` method iterates through the array and prints each element followed by a space. This method is used to print the array before and after sorting.

This process continues until the entire array is sorted. The Selection Sort algorithm has a time complexity of  $O(n^2)$ , making it less efficient for large arrays compared to more advanced sorting algorithms like Quick Sort or Merge Sort. However, it is easy to implement and understand.

4o

4. Write a Java program that sorts a list of numbers using the insertion Sort algorithm, a quadratic sorting technique. The program should define an array of unsorted integers and include a method that performs the selection sort. This method should iterate through the array to find the minimum element in the unsorted portion and swap it with the first unsorted element, continuing this process until the entire array is sorted. The program should also include a utility method to print the array before and after sorting.

It looks like the prompt mentioned "insertion sort" but then describes the process of "selection sort" again. I'll provide the correct implementation and explanation for insertion sort as described initially.

Here's a Java program that sorts a list of numbers using the Insertion Sort algorithm. The program defines an array of unsorted integers and includes a method that performs the insertion sort. It also includes a utility method to print the array before and after sorting.

```

java
Copy code
public class InsertionSort {

```

```

public static void main(String[] args) {
    int[] numbers = {64, 25, 12, 22, 11};

    System.out.println("Unsorted array:");
    printArray(numbers);

    insertionSort(numbers);

    System.out.println("Sorted array:");
    printArray(numbers);
}

public static void insertionSort(int[] array) {
    int n = array.length;
    for (int i = 1; i < n; ++i) {
        int key = array[i];
        int j = i - 1;

        // Move elements of array[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j = j - 1;
        }
        array[j + 1] = key;
    }
}

public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
    System.out.println();
}
}

```

## Explanation of the Insertion Sort Algorithm

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. Here's a step-by-step explanation:

1. **Initialization:** The method `insertionSort` takes an array as an input parameter. The length of the array is stored in the variable `n`.
2. **Outer Loop:** The outer loop iterates over the array from the second element to the last element (`i` ranges from `1` to `n-1`), treating the first element as a sorted sublist.
3. **Key Element:** For each position `i`, the element at `array[i]` is stored in the variable `key`. The variable `j` is initialized to `i-1`.
4. **Inner Loop:** The inner loop shifts elements of the sorted sublist (from `array[0]` to `array[i-1]`) that are greater than `key` to one position ahead of their current position. This creates a gap for the `key` to be inserted at its correct position.
5. **Insertion:** Once the correct position is found (when `array[j]` is no longer greater than `key`), the `key` is placed at `array[j + 1]`.

6. **Utility Method:** The `printArray` method iterates through the array and prints each element followed by a space. This method is used to print the array before and after sorting.

This process continues until the entire array is sorted. The Insertion Sort algorithm has a time complexity of  $O(n^2)$ , making it less efficient for large arrays compared to more advanced sorting algorithms, but it is simple to implement and efficient for small data sets or nearly sorted arrays.

40

5. Write a Java program that sorts a list of numbers using the Insertion Sort algorithm, a simple and intuitive sorting technique. The program should define an array of unsorted integers and include a method that performs the insertion sort. This method should iterate through the array, taking each element and inserting it into its correct position in the sorted portion of the array, thereby growing the sorted portion one element at a time until the entire array is sorted. The program should also include a utility method to print the array before and after sorting.

Here's a Java program that sorts a list of numbers using the Insertion Sort algorithm. The program defines an array of unsorted integers and includes a method that performs the insertion sort. It also includes a utility method to print the array before and after sorting.

```
java
Copy code
public class InsertionSort {

    public static void main(String[] args) {
        int[] numbers = {64, 25, 12, 22, 11};

        System.out.println("Unsorted array:");
        printArray(numbers);

        insertionSort(numbers);

        System.out.println("Sorted array:");
        printArray(numbers);
    }

    public static void insertionSort(int[] array) {
        int n = array.length;
        for (int i = 1; i < n; ++i) {
            int key = array[i];
            int j = i - 1;

            // Move elements of array[0..i-1], that are greater than key,
            // to one position ahead of their current position
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j = j - 1;
            }
            array[j + 1] = key;
        }
    }

    public static void printArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
        }
        System.out.println();
    }
}
```

## Explanation of the Insertion Sort Algorithm

Insertion Sort is a simple and intuitive sorting algorithm that works by building a sorted array one element at a time. Here's how it works:

1. **Initialization:** The method `insertionSort` takes an array of integers as an input parameter. The length of the array is stored in the variable `n`.
2. **Outer Loop:** The outer loop iterates over the array starting from the second element (`i` ranges from 1 to `n - 1`), treating the first element as a sorted sublist.
3. **Key Element:** For each position `i`, the element at `array[i]` is stored in the variable `key`. The variable `j` is initialized to `i - 1`.
4. **Inner Loop:** The inner loop shifts elements of the sorted sublist (from `array[0]` to `array[i - 1]`) that are greater than `key` to one position ahead of their current position. This creates a gap for the `key` to be inserted at its correct position.
5. **Insertion:** Once the correct position is found (when `array[j]` is no longer greater than `key` or `j` becomes less than 0), the `key` is placed at `array[j + 1]`.
6. **Utility Method:** The `printArray` method iterates through the array and prints each element followed by a space. This method is used to print the array before and after sorting.

This process continues until the entire array is sorted. The Insertion Sort algorithm has a time complexity of  $O(n^2)$ , which makes it less efficient for large arrays compared to more advanced sorting algorithms, but it is simple to implement and works well for small or nearly sorted arrays.

40

6. Write a Java program to demonstrate the Stack data structure using classes and objects. The program should define a Stack class that includes methods for standard stack operations: pushing an element onto the stack, popping an element from the stack, checking if the stack is empty, and checking if the stack is full. Do the operations:

`Push(50), Push(27), Push(40), Pop(), Push(90), Push(50), Push(50), Pop()` Print the elements in the

stack. 7. Write a Java program to demonstrate the Queue data structure using classes and objects. The program should define a Queue class that includes methods for standard queue operations: enqueueing an element (adding it to the end of the queue), dequeueing an element (removing it from the front of the queue), checking if the queue is empty, and checking if the queue is full. Performs various queue operations, and print the results.

## Stack Implementation in Java

Here's a Java program to demonstrate the Stack data structure using classes and objects. The program defines a `Stack` class with methods for standard stack operations: pushing an element onto the stack, popping an element from the stack, checking if the stack is empty, and checking if the stack is full.

```
java
Copy code
class Stack {
    private int maxSize;
    private int[] stackArray;
    private int top;
```



```

public Stack(int size) {
    maxSize = size;
    stackArray = new int[maxSize];
    top = -1;
}

public void push(int value) {
    if (!isFull()) {
        stackArray[++top] = value;
    } else {
        System.out.println("Stack is full. Cannot push " + value);
    }
}

public int pop() {
    if (!isEmpty()) {
        return stackArray[top--];
    } else {
        System.out.println("Stack is empty. Cannot pop.");
        return -1;
    }
}

public boolean isEmpty() {
    return (top == -1);
}

public boolean isFull() {
    return (top == maxSize - 1);
}

public void printStack() {
    if (isEmpty()) {
        System.out.println("Stack is empty.");
    } else {
        System.out.print("Stack elements: ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stackArray[i] + " ");
        }
        System.out.println();
    }
}

public static void main(String[] args) {
    Stack stack = new Stack(10);

    stack.push(50);
    stack.push(27);
    stack.push(40);
    stack.pop();
    stack.push(90);
    stack.push(50);
    stack.push(50);
    stack.pop();

    stack.printStack();
}
}

```

## Queue Implementation in Java

Here's a Java program to demonstrate the Queue data structure using classes and objects. The program defines a `Queue` class with methods for standard queue operations: enqueueing an element (adding it to the end of the queue), dequeueing an element (removing it from the front of the queue), checking if the queue is empty, and checking if the queue is full.

java

Copy code

```
class Queue {
    private int maxSize;
    private int[] queueArray;
    private int front;
    private int rear;
    private int nItems;

    public Queue(int size) {
        maxSize = size;
        queueArray = new int[maxSize];
        front = 0;
        rear = -1;
        nItems = 0;
    }

    public void enqueue(int value) {
        if (!isFull()) {
            if (rear == maxSize - 1) {
                rear = -1;
            }
            queueArray[++rear] = value;
            nItems++;
        } else {
            System.out.println("Queue is full. Cannot enqueue " + value);
        }
    }

    public int dequeue() {
        if (!isEmpty()) {
            int temp = queueArray[front++];
            if (front == maxSize) {
                front = 0;
            }
            nItems--;
            return temp;
        } else {
            System.out.println("Queue is empty. Cannot dequeue.");
            return -1;
        }
    }

    public boolean isEmpty() {
        return (nItems == 0);
    }

    public boolean isFull() {
        return (nItems == maxSize);
    }

    public void printQueue() {
        if (isEmpty()) {
            System.out.println("Queue is empty.");
        } else {
            // Implementation for printing the queue contents
        }
    }
}
```

```

        System.out.print("Queue elements: ");
        int index = front;
        for (int i = 0; i < nItems; i++) {
            System.out.print(queueArray[index] + " ");
            index = (index + 1) % maxSize;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Queue queue = new Queue(10);

        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);
        queue.enqueue(40);
        queue.dequeue();
        queue.enqueue(50);
        queue.dequeue();

        queue.printQueue();
    }
}

```

## Explanation

### Stack:

- **Push Operation:** Adds an element to the top of the stack. Checks if the stack is full before pushing.
- **Pop Operation:** Removes the top element from the stack. Checks if the stack is empty before popping.
- **isEmpty:** Checks if the stack is empty.
- **isFull:** Checks if the stack is full.
- **printStack:** Prints all elements in the stack.

### Queue:

- **Enqueue Operation:** Adds an element to the rear of the queue. Checks if the queue is full before enqueueing.
- **Dequeue Operation:** Removes an element from the front of the queue. Checks if the queue is empty before dequeuing.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full.
- **printQueue:** Prints all elements in the queue.

The provided code includes examples of pushing and popping elements for the stack, and enqueueing and dequeuing elements for the queue. The `printStack` and `printQueue` methods are used to print the elements in the stack and queue, respectively.

4o

8. Write a Java program that prompts the user to input 10 numbers, each between 100 and 1000 inclusive. As each number is read, the program should check if it's already been entered (a duplicate) and only display it if it's unique. After each new input, the program should display the

complete set of unique numbers entered so far. Utilize a Set data structure to store and manage the unique numbers efficiently. Ensure the program validates each input to ensure it falls within the specified range and isn't a duplicate of any previously entered number. Additionally, include appropriate messages to guide the user through the input process and handle edge cases gracefully, such as attempting to input a number outside the specified range or exceeding the allowed number of inputs.

Here's a Java program that prompts the user to input 10 numbers, each between 100 and 1000 inclusive. The program ensures that only unique numbers within the specified range are accepted and displays the set of unique numbers after each input.

```
java
Copy code
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class UniqueNumbers {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Set<Integer> uniqueNumbers = new HashSet<>();
        int count = 0;

        System.out.println("Please enter 10 unique numbers between 100 and 1000
(inclusive):");

        while (count < 10) {
            System.out.print("Enter number " + (count + 1) + ": ");
            int number = scanner.nextInt();

            if (number < 100 || number > 1000) {
                System.out.println("Invalid number. Please enter a number
between 100 and 1000.");
            } else if (uniqueNumbers.contains(number)) {
                System.out.println("Duplicate number. Please enter a unique
number.");
            } else {
                uniqueNumbers.add(number);
                count++;
                System.out.println("Unique numbers so far: " + uniqueNumbers);
            }
        }

        System.out.println("All unique numbers entered: " + uniqueNumbers);
        scanner.close();
    }
}
```

## Explanation

1. **Import Statements:** The program uses `HashSet` for storing unique numbers and `Scanner` for reading user input.
2. **Main Method:**
  - A `Scanner` object is created to read user input.
  - A `HashSet` named `uniqueNumbers` is created to store unique numbers.

- A counter `count` is initialized to keep track of the number of valid inputs.
3. **User Prompt:** The program prompts the user to enter 10 unique numbers between 100 and 1000 inclusive.
  4. **Input Loop:**
    - The loop runs until 10 unique numbers are entered.
    - The user is prompted to enter a number.
    - The program checks if the number is within the valid range (100 to 1000).
      - If the number is outside the range, an error message is displayed.
    - The program checks if the number is a duplicate.
      - If the number is a duplicate, an error message is displayed.
    - If the number is valid and unique, it is added to the `HashSet`, the counter is incremented, and the set of unique numbers entered so far is displayed.
  5. **Completion:** After 10 unique numbers are entered, the program displays all unique numbers entered and closes the scanner.

## Edge Cases

- **Invalid Input Range:** If the user enters a number outside the range of 100 to 1000, the program prompts the user to enter a valid number.
- **Duplicate Numbers:** If the user enters a duplicate number, the program prompts the user to enter a unique number.
- **Valid and Unique Inputs:** The program continues to accept valid and unique inputs until 10 numbers are entered.

This program effectively demonstrates the use of a `Set` data structure to manage and store unique numbers efficiently while providing a user-friendly interface for input validation and guidance.

40

9. Write a Java program to illustrate how constructors in subclasses interact with the constructors in their superclasses, emphasizing the concept of constructor chaining. 10. Consider a simple scenario where you have a base class `Vehicle` with attributes `make` and `model`. Create a subclass `Car` that inherits from `Vehicle` and adds a new attribute `numDoors`. Write a Java program to demonstrate how the concept of inheritance is applied in this scenario.

## 9. Constructors in Subclasses and Constructor Chaining

Here's a Java program that illustrates how constructors in subclasses interact with constructors in their superclasses, emphasizing the concept of constructor chaining:

```
java
Copy code
class SuperClass {
    public SuperClass() {
        System.out.println("SuperClass no-argument constructor called");
    }

    public SuperClass(String message) {
        System.out.println("SuperClass parameterized constructor called with
message: " + message);
    }
}
```

```

class SubClass extends SuperClass {
    public SubClass() {
        // Calls the no-argument constructor of SuperClass
        super();
        System.out.println("SubClass no-argument constructor called");
    }

    public SubClass(String message) {
        // Calls the parameterized constructor of SuperClass
        super(message);
        System.out.println("SubClass parameterized constructor called with
message: " + message);
    }
}

public class ConstructorChainingDemo {
    public static void main(String[] args) {
        System.out.println("Creating SubClass object with no-argument
constructor:");
        SubClass subClass1 = new SubClass();

        System.out.println("\nCreating SubClass object with parameterized
constructor:");
        SubClass subClass2 = new SubClass("Hello from SubClass");
    }
}

```

## Explanation

1. **SuperClass:** This class has two constructors - a no-argument constructor and a parameterized constructor.
  - The no-argument constructor prints a message.
  - The parameterized constructor prints a message with the provided string.
2. **SubClass:** This class extends **SuperClass** and also has two constructors.
  - The no-argument constructor of **SubClass** calls the no-argument constructor of **SuperClass** using `super()`, and then prints a message.
  - The parameterized constructor of **SubClass** calls the parameterized constructor of **SuperClass** using `super(message)`, and then prints a message.
3. **ConstructorChainingDemo:** The `main` method demonstrates the creation of **SubClass** objects using both constructors, showing how constructor chaining works.

## Output

```

sql
Copy code
Creating SubClass object with no-argument constructor:
SuperClass no-argument constructor called
SubClass no-argument constructor called

Creating SubClass object with parameterized constructor:
SuperClass parameterized constructor called with message: Hello from SubClass
SubClass parameterized constructor called with message: Hello from SubClass

```

## 10. Inheritance Example with Vehicle and Car

Here's a Java program to demonstrate inheritance with a base class `Vehicle` and a subclass `Car`:

```
java
Copy code
class Vehicle {
    private String make;
    private String model;

    public Vehicle(String make, String model) {
        this.make = make;
        this.model = model;
    }

    public String getMake() {
        return make;
    }

    public String getModel() {
        return model;
    }

    public void displayInfo() {
        System.out.println("Make: " + make + ", Model: " + model);
    }
}

class Car extends Vehicle {
    private int numDoors;

    public Car(String make, String model, int numDoors) {
        // Call the constructor of the superclass (Vehicle)
        super(make, model);
        this.numDoors = numDoors;
    }

    public int getNumDoors() {
        return numDoors;
    }

    @Override
    public void displayInfo() {
        // Call the displayInfo method of the superclass (Vehicle)
        super.displayInfo();
        System.out.println("Number of doors: " + numDoors);
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle("Toyota", "Camry");
        System.out.println("Vehicle Info:");
        vehicle.displayInfo();

        System.out.println("\nCar Info:");
        Car car = new Car("Honda", "Civic", 4);
        car.displayInfo();
    }
}
```

## Explanation

1. **Vehicle Class:** This is the base class with attributes `make` and `model`, a constructor to initialize them, and a method `displayInfo` to print the vehicle's information.
2. **Car Class:** This is a subclass that extends `Vehicle` and adds a new attribute `numDoors`.
  - The constructor of `Car` calls the constructor of `Vehicle` using `super(make, model)` to initialize the inherited attributes.
  - The `displayInfo` method in `Car` overrides the method in `Vehicle` and also prints the number of doors.
3. **InheritanceDemo:** The `main` method demonstrates the creation of `Vehicle` and `Car` objects, showing how inheritance is applied and how the `displayInfo` method is overridden in the subclass.

## Output

```
yaml
Copy code
Vehicle Info:
Make: Toyota, Model: Camry

Car Info:
Make: Honda, Model: Civic
Number of doors: 4
```

This program shows how inheritance allows the `Car` class to reuse and extend the functionality of the `Vehicle` class, demonstrating the core principles of object-oriented programming.

40

11. In a basic library system, you have a base class `LibraryItem` representing common attributes like title and itemID. Create two subclasses, `Book` and `DVD`, that inherit from `LibraryItem` and add attributes specific to each, such as author for books and duration for DVDs. Write a Java program to implement a sample scenario where you instantiate objects of both subclasses and demonstrate the use of inherited and subclass-specific attributes. 12. Imagine a scenario where you have a base class `Shape` with attributes color and area. Create a subclass `Circle` that inherits from `Shape` and adds a new attribute radius. Write a Java program to demonstrate the concept of inheritance. 13. Write a scenario-based Java program that demonstrates the use of inner classes. In this scenario, create a bank application where customers have accounts, and each account can have transactions associated with it. Use inner classes to represent accounts and transactions within the bank. create a bank instance, add accounts, perform transactions, and display account information using the bank's methods. 14. Write a scenario-based Java program that demonstrates the use of final methods and classes. In this scenario, create a simple application for managing employees in a company. Use final methods and classes to ensure that certain behaviors or classes cannot be overridden or extended. The `EmployeeApp` class demonstrates creating an employee object and displaying its information using the final method `displayInfo()`. 15. Write a Java program that models a simple inheritance hierarchy for employees in a company. You have a base class called `Employee` and derived classes `Manager` and `Developer`. Each class will have properties and methods specific to their role in the company. Demonstrate inheritance by creating objects of both derived classes and invoking their methods to display information specific to each role in the company. 16. Write a



scenario-based Java program that models a simple inheritance hierarchy for bank accounts. You have a base class called `Account` and derived classes `SavingsAccount` and `CheckingAccount`. Each class will have properties and methods specific to the type of account. Demonstrate inheritance by creating objects of both derived classes and invoking their methods to perform account operations.

17. Write a scenario-based Java program that models a simple inheritance hierarchy for electronic devices. You have a base class called `ElectronicDevice` and derived classes `Television` and `Laptop`. Each class will have properties and methods specific to the type of electronic device. Demonstrate inheritance by creating objects of both derived classes and invoking their methods to display device information.

18. Write a scenario-based Java program where you simulate a ticket booking system for a ride. Define a custom exception called `InvalidAgeException` to handle cases where the user's age is below the minimum age requirement for certain rides. Handle exceptions such as invalid age and invalid input using try-catch blocks, and display appropriate error messages to the user.

19. Write a java program to read Bank details from a file, sort them based on Balance, and write the sorted data back to the file

20. Write a java program that implements a multi-threaded application that has five threads. First thread generates a random integer every 3 second and if the value is odd, the second thread computes the area of a circle using the odd value as radius and prints. If the value is even, the third thread will compute the area of a Square using the odd value and print the result.

21. Write a Java program that utilizes either `ArrayList`, `LinkedList`, `HashSet`, `LinkedHashSet`, and `TreeSet` to manage a collection of integers. The program should allow the user to perform operations such as adding integers to the collection, removing integers, and displaying the current contents of the collection. Ensure that the program handles duplicates appropriately based on the chosen data structure's characteristics (e.g., allowing duplicates in `ArrayList` and `LinkedList` but not in `HashSet`, `LinkedHashSet`, and `TreeSet`). Include functionality to demonstrate unique features of the chosen data structure, such as maintaining insertion order (`ArrayList`, `LinkedList`, `LinkedHashSet`) or sorting elements (`TreeSet`). Provide clear instructions and example scenarios to illustrate the behavior of the program when adding, removing, and retrieving elements.

## 11. Basic Library System

Here is a Java program that demonstrates a basic library system with a base class `LibraryItem` and two subclasses `Book` and `DVD`.

```
java
Copy code
class LibraryItem {
    private String title;
    private int itemID;

    public LibraryItem(String title, int itemID) {
        this.title = title;
        this.itemID = itemID;
    }

    public String getTitle() {
        return title;
    }

    public int getItemID() {
        return itemID;
    }

    public void displayInfo() {
        System.out.println("Title: " + title + ", Item ID: " + itemID);
    }
}
```

```

    }
}

class Book extends LibraryItem {
    private String author;

    public Book(String title, int itemID, String author) {
        super(title, itemID);
        this.author = author;
    }

    public String getAuthor() {
        return author;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Author: " + author);
    }
}

class DVD extends LibraryItem {
    private int duration;

    public DVD(String title, int itemID, int duration) {
        super(title, itemID);
        this.duration = duration;
    }

    public int getDuration() {
        return duration;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Duration: " + duration + " minutes");
    }
}

public class LibrarySystem {
    public static void main(String[] args) {
        Book book = new Book("The Great Gatsby", 1, "F. Scott Fitzgerald");
        DVD dvd = new DVD("Inception", 2, 148);

        System.out.println("Book Information:");
        book.displayInfo();

        System.out.println("\nDVD Information:");
        dvd.displayInfo();
    }
}

```

## 12. Shape and Circle Inheritance Example

Here is a Java program that demonstrates inheritance with a base class **Shape** and a subclass **Circle**.

```

java
Copy code
class Shape {

```

```

    private String color;
    protected double area;

    public Shape(String color) {
        this.color = color;
        this.area = 0.0;
    }

    public String getColor() {
        return color;
    }

    public double getArea() {
        return area;
    }

    public void displayInfo() {
        System.out.println("Color: " + color + ", Area: " + area);
    }
}

class Circle extends Shape {
    private double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
        calculateArea();
    }

    private void calculateArea() {
        this.area = Math.PI * radius * radius;
    }

    public double getRadius() {
        return radius;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Radius: " + radius);
    }
}

public class ShapeDemo {
    public static void main(String[] args) {
        Circle circle = new Circle("Red", 5.0);
        System.out.println("Circle Information:");
        circle.displayInfo();
    }
}

```

### 13. Bank Application with Inner Classes

Here is a Java program that demonstrates the use of inner classes in a bank application.

```

java
Copy code
import java.util.ArrayList;

class Bank {
    private String name;

```

```

private ArrayList<Account> accounts;

public Bank(String name) {
    this.name = name;
    this.accounts = new ArrayList<>();
}

public void addAccount(String customerName, int accountNumber) {
    accounts.add(new Account(customerName, accountNumber));
}

public void performTransaction(int accountNumber, double amount, String
type) {
    for (Account account : accounts) {
        if (account.getAccountNumber() == accountNumber) {
            account.new Transaction(amount, type).execute();
            return;
        }
    }
    System.out.println("Account not found.");
}

public void displayAccounts() {
    for (Account account : accounts) {
        account.displayInfo();
    }
}

class Account {
    private String customerName;
    private int accountNumber;
    private double balance;
    private ArrayList<Transaction> transactions;

    public Account(String customerName, int accountNumber) {
        this.customerName = customerName;
        this.accountNumber = accountNumber;
        this.balance = 0.0;
        this.transactions = new ArrayList<>();
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public void displayInfo() {
        System.out.println("Customer: " + customerName + ", Account Number:
" + accountNumber + ", Balance: " + balance);
        System.out.println("Transactions:");
        for (Transaction transaction : transactions) {
            System.out.println(transaction);
        }
    }

    class Transaction {
        private double amount;
        private String type;

        public Transaction(double amount, String type) {
            this.amount = amount;
            this.type = type;
        }

        public void execute() {

```

```

        if (type.equalsIgnoreCase("deposit")) {
            balance += amount;
        } else if (type.equalsIgnoreCase("withdraw")) {
            if (amount <= balance) {
                balance -= amount;
            } else {
                System.out.println("Insufficient funds for
withdrawal.");
                return;
            }
        }
        transactions.add(this);
        System.out.println("Transaction successful: " + this);
    }

    @Override
    public String toString() {
        return "Type: " + type + ", Amount: " + amount;
    }
}

}

public class BankApplication {
    public static void main(String[] args) {
        Bank bank = new Bank("MyBank");

        bank.addAccount("Alice", 1001);
        bank.addAccount("Bob", 1002);

        bank.performTransaction(1001, 500, "deposit");
        bank.performTransaction(1001, 200, "withdraw");
        bank.performTransaction(1002, 1000, "deposit");
        bank.performTransaction(1002, 1500, "withdraw");

        System.out.println("\nBank Accounts Information:");
        bank.displayAccounts();
    }
}

```

## 14. Final Methods and Classes Example

Here is a Java program that demonstrates the use of final methods and classes.

```

java
Copy code
final class Employee {
    private String name;
    private int id;
    private String department;

    public Employee(String name, int id, String department) {
        this.name = name;
        this.id = id;
        this.department = department;
    }

    public final void displayInfo() {
        System.out.println("Employee ID: " + id + ", Name: " + name + ",
Department: " + department);
    }
}

```

```

public class EmployeeApp {
    public static void main(String[] args) {
        Employee emp = new Employee("John Doe", 101, "Engineering");
        emp.displayInfo();
    }
}

```

## 15. Employee Inheritance Hierarchy

Here is a Java program that demonstrates a simple inheritance hierarchy for employees.

```

java
Copy code
class Employee {
    private String name;
    private int id;

    public Employee(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public int getId() {
        return id;
    }

    public void displayInfo() {
        System.out.println("Employee ID: " + id + ", Name: " + name);
    }
}

class Manager extends Employee {
    private String department;

    public Manager(String name, int id, String department) {
        super(name, id);
        this.department = department;
    }

    public String getDepartment() {
        return department;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Department: " + department);
    }
}

class Developer extends Employee {
    private String programmingLanguage;

    public Developer(String name, int id, String programmingLanguage) {
        super(name, id);
        this.programmingLanguage = programmingLanguage;
    }

    public String getProgrammingLanguage() {

```

```

        return programmingLanguage;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Programming Language: " + programmingLanguage);
    }
}

public class CompanyHierarchy {
    public static void main(String[] args) {
        Manager manager = new Manager("Alice", 201, "HR");
        Developer developer = new Developer("Bob", 301, "Java");

        System.out.println("Manager Information:");
        manager.displayInfo();

        System.out.println("\nDeveloper Information:");
        developer.displayInfo();
    }
}

```

## 16. Bank Account Inheritance Hierarchy

Here is a Java program that models a simple inheritance hierarchy for bank accounts.

```

java
Copy code
class Account {
    private int accountNumber;
    private double balance;

    public Account(int accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0.0;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    public void displayInfo() {
        System.out.println("Account Number: " + accountNumber + ", Balance: " +
balance);
    }
}

```

```

class SavingsAccount extends Account {
    private double interestRate;

    public SavingsAccount(int accountNumber, double interestRate) {
        super(accountNumber);
        this.interestRate = interestRate;
    }

    public double getInterestRate() {
        return interestRate;
    }

    public void applyInterest() {
        double interest = getBalance() * interestRate / 100;
        deposit(interest);
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Interest Rate: " + interestRate + "%");
    }
}

class CheckingAccount extends Account {
    private double overdraftLimit;

    public CheckingAccount(int accountNumber, double overdraftLimit) {
        super(accountNumber);
        this.overdraftLimit = overdraftLimit;
    }

    @Override
    public void withdraw(double amount) {
        if (amount > 0 && amount <= getBalance() + overdraftLimit) {
            super.withdraw(amount);
        }
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Overdraft Limit: " + overdraftLimit);
    }
}

public class BankAccountDemo {
    public static void main(String[] args) {
        SavingsAccount savings = new SavingsAccount(101, 3.5);
        CheckingAccount checking = new CheckingAccount(102, 500);

        savings.deposit(1000);
        checking.deposit(500);

        savings.applyInterest();
        checking.withdraw(800);

        System.out.println("Savings Account Information:");
        savings.displayInfo();

        System.out.println("\nChecking Account Information:");
        checking.displayInfo();
    }
}

```



```
}
```

## 17. Electronic Device Inheritance Hierarchy

Here is a Java program that models a simple inheritance hierarchy for electronic devices.

java

Copy code

```
class ElectronicDevice {
    private String brand;
    private String model;

    public ElectronicDevice(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }

    public String getBrand() {
        return brand;
    }

    public String getModel() {
        return model;
    }

    public void displayInfo() {
        System.out.println("Brand: " + brand + ", Model: " + model);
    }
}

class Television extends ElectronicDevice {
    private int screenSize;

    public Television(String brand, String model, int screenSize) {
        super(brand, model);
        this.screenSize = screenSize;
    }

    public int getScreenSize() {
        return screenSize;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Screen Size: " + screenSize + " inches");
    }
}

class Laptop extends ElectronicDevice {
    private int ramSize;

    public Laptop(String brand, String model, int ramSize) {
        super(brand, model);
        this.ramSize = ramSize;
    }

    public int getRamSize() {
        return ramSize;
    }

    @Override
    public void displayInfo() {
```

```

        super.displayInfo();
        System.out.println("RAM Size: " + ramSize + " GB");
    }
}

public class ElectronicDeviceDemo {
    public static void main(String[] args) {
        Television tv = new Television("Samsung", "QLED", 55);
        Laptop laptop = new Laptop("Dell", "XPS", 16);

        System.out.println("Television Information:");
        tv.displayInfo();

        System.out.println("\nLaptop Information:");
        laptop.displayInfo();
    }
}

```

## 18. Ticket Booking System with Custom Exception

Here is a Java program that simulates a ticket booking system with a custom exception for invalid age.

```

java
Copy code
import java.util.Scanner;

class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class TicketBookingSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Enter your age: ");
            int age = scanner.nextInt();

            if (age < 18) {
                throw new InvalidAgeException("You must be at least 18 years old
to book a ticket for this ride.");
            }

            System.out.println("Ticket booked successfully.");
        } catch (InvalidAgeException e) {
            System.out.println("Booking failed: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Invalid input. Please enter a valid age.");
        } finally {
            scanner.close();
        }
    }
}

```

## 19. Read, Sort, and Write Bank Details from File

Here's a Java program that reads bank details from a file, sorts them based on balance, and writes the sorted data back to the file.

java

Copy code

```
import java.io.*;
import java.util.*;

class BankDetail implements Comparable<BankDetail> {
    private String name;
    private int accountNumber;
    private double balance;

    public BankDetail(String name, int accountNumber, double balance) {
        this.name = name;
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    @Override
    public String toString() {
        return name + "," + accountNumber + "," + balance;
    }

    @Override
    public int compareTo(BankDetail other) {
        return Double.compare(this.balance, other.balance);
    }

    public static BankDetail fromString(String line) {
        String[] parts = line.split(",");
        String name = parts[0];
        int accountNumber = Integer.parseInt(parts[1]);
        double balance = Double.parseDouble(parts[2]);
        return new BankDetail(name, accountNumber, balance);
    }
}

public class BankDetailsSorter {
    public static void main(String[] args) {
        List<BankDetail> bankDetails = new ArrayList<>();

        // Read bank details from file
        try (BufferedReader reader = new BufferedReader(new
FileReader("bank_details.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                bankDetails.add(BankDetail.fromString(line));
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }

        // Sort bank details based on balance
        Collections.sort(bankDetails);
    }
}
```

```

        // Write sorted bank details back to file
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter("sorted_bank_details.txt"))) {
            for (BankDetail detail : bankDetails) {
                writer.write(detail.toString());
                writer.newLine();
            }
            System.out.println("Bank details sorted and saved successfully.");
        } catch (IOException e) {
            System.out.println("Error writing file: " + e.getMessage());
        }
    }
}

```

## 20. Multi-threaded Application with Odd/Even Value Processing

Here's a Java program that implements a multi-threaded application with five threads. The first thread generates a random integer every 3 seconds, and based on whether the value is odd or even, different threads compute and print the area of a circle or square.

```

java
Copy code
import java.util.Random;

class RandomNumberGenerator extends Thread {
    @Override
    public void run() {
        Random random = new Random();
        while (true) {
            try {
                Thread.sleep(3000);
                int number = random.nextInt(100);
                System.out.println("Generated number: " + number);
                if (number % 2 == 0) {
                    new EvenThread(number).start();
                } else {
                    new OddThread(number).start();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class OddThread extends Thread {
    private int number;

    public OddThread(int number) {
        this.number = number;
    }

    @Override
    public void run() {
        double area = Math.PI * number * number;
        System.out.println("Area of Circle with radius " + number + " (Odd
number): " + area);
    }
}

class EvenThread extends Thread {
    private int number;

```

```

    public EvenThread(int number) {
        this.number = number;
    }

    @Override
    public void run() {
        double area = number * number;
        System.out.println("Area of Square with side " + number + " (Even
number): " + area);
    }
}

public class MultiThreadedApp {
    public static void main(String[] args) {
        new RandomNumberGenerator().start();
    }
}

```

## 21. Collection Management with Various Data Structures

Here's a Java program that manages a collection of integers using ArrayList, LinkedList, HashSet, LinkedHashSet, and TreeSet. It allows the user to add integers, remove integers, and display the current contents of the collection based on the chosen data structure.

```

java
Copy code
import java.util.*;

public class CollectionManagement {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();
        Set<Integer> hashSet = new HashSet<>();
        Set<Integer> linkedHashSet = new LinkedHashSet<>();
        Set<Integer> treeSet = new TreeSet<>();

        while (true) {
            System.out.println("\nChoose a data structure:");
            System.out.println("1. ArrayList");
            System.out.println("2. LinkedList");
            System.out.println("3. HashSet");
            System.out.println("4. LinkedHashSet");
            System.out.println("5. TreeSet");
            System.out.println("6. Exit");

            int choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    operateCollection(arrayList, "ArrayList");
                    break;
                case 2:
                    operateCollection(linkedList, "LinkedList");
                    break;
                case 3:
                    operateSet(hashSet, "HashSet");
                    break;
                case 4:
                    operateSet(linkedHashSet, "LinkedHashSet");
                    break;
                case 5:

```

```

        operateSet(treeSet, "TreeSet");
        break;
    case 6:
        System.out.println("Exiting program...");
        scanner.close();
        return;
    default:
        System.out.println("Invalid choice. Please enter a valid
option.");
    }
}

private static void operateCollection(List<Integer> collection, String
collectionType) {
    Scanner scanner = new Scanner(System.in);

    while (true) {
        System.out.println("\nOperations on " + collectionType + ":");
        System.out.println("1. Add an integer");
        System.out.println("2. Remove an integer");
        System.out.println("3. Display current collection");
        System.out.println("4. Back to data structure selection");

        int choice = scanner.nextInt();
        switch (choice) {
            case 1:
                System.out.print("Enter integer to add: ");
                int numToAdd = scanner.nextInt();
                collection.add(numToAdd);
                System.out.println(numToAdd + " added to " +
collectionType);
                break;
            case 2:
                System.out.print("Enter integer to remove: ");
                int numToRemove = scanner.nextInt();
                if (collection.remove(Integer.valueOf(numToRemove))) {
                    System.out.println(numToRemove + " removed from " +
collectionType);
                } else {
                    System.out.println(numToRemove + " not found in " +
collectionType);
                }
                break;
            case 3:
                System.out.println("Current " + collectionType + ": " +
collection);
                break;
            case 4:
                return;
            default:
                System.out.println("Invalid choice. Please enter a valid
option.");
        }
    }
}

private static void operateSet(Set<Integer> set, String setType) {
    Scanner scanner = new Scanner(System.in);

    while (true) {
        System.out.println("\nOperations on " + setType + ":");
        System.out.println("1. Add an integer");
        System.out.println("2. Remove an integer");

```

```

System.out.println("3. Display current set");
System.out.println("4. Back to data structure selection");

int choice = scanner.nextInt();
switch (choice) {
    case 1:
        System.out.print("Enter integer to add: ");
        int numToAdd = scanner.nextInt();
        if (set.add(numToAdd)) {
            System.out.println(numToAdd + " added to " + setType);
        } else {
            System.out.println(numToAdd + " is already present in "
+ setType);
        }
        break;
    case 2:
        System.out.print("Enter integer to remove: ");
        int numToRemove = scanner.nextInt();
        if (set.remove(numToRemove)) {
            System.out.println(numToRemove + " removed from " +
setType);
        } else {
            System.out.println(numToRemove + " not found in " +
setType);
        }
        break;
    case 3:
        System.out.println("Current " + setType + ": " + set);
        break;
    case 4:
        return;
    default:
        System.out.println("Invalid choice. Please enter a valid
option.");
}
}
}
}

```

These programs cover a range of functionalities and demonstrate key concepts such as file handling, multi-threading, and data structure usage in Java. Let me know if you have any further questions or if there's anything else you'd like to explore!