**Module\_2\_Assignment:**

1. **Differentiate comparable and comparator interfaces.**

ANSWER:

**Comparable Interface**

The **Comparable** interface is used to define a natural ordering for objects of a class. It requires the implementation of a single method: **compareTo(T o)**, where **T** is the type of the object being compared. This method returns an integer value indicating the order of the objects:

* Negative integer: the current object is less than the compared object
* Zero: the current object is equal to the compared object
* Positive integer: the current object is greater than the compared object

When a class implements **Comparable**, it means that instances of that class can be compared to each other using the **compareTo** method.

**Comparator Interface**

The **Comparator** interface is used to define a custom ordering for objects of a class. It requires the implementation of two methods: **compare(T o1, T o2)** and **equals(Object obj)**. The **compare** method returns an integer value indicating the order of the objects, similar to the **compareTo** method in **Comparable**.

The key difference between **Comparable** and **Comparator** is that **Comparator** is an external comparison strategy, whereas **Comparable** is an internal comparison strategy. In other words, **Comparator** allows you to decouple the comparison logic from the class itself, making it more flexible and reusable.

**Example Use Cases**

* Use **Comparable** when you want to define a natural ordering for a class, such as sorting a list of strings alphabetically.
* Use **Comparator** when you want to define a custom ordering for a class, such as sorting a list of employees by their salary or department.

// Comparable example

public class Person implements Comparable<Person> {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int compareTo(Person other) {

return this.age - other.age;

}

}

// Comparator example

public class EmployeeComparator implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getSalary() - e2.getSalary();

}

}

public class Employee {

private String name;

private int salary;

public Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

public int getSalary() {

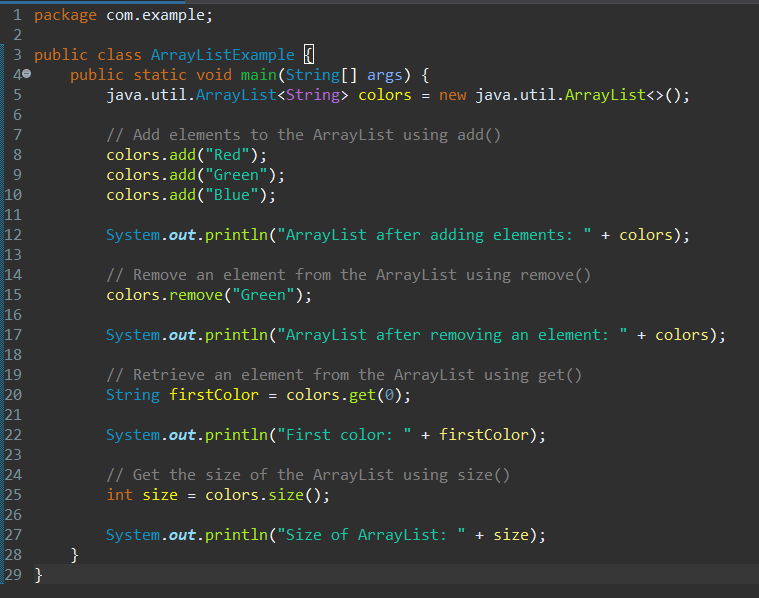
return salary;

}

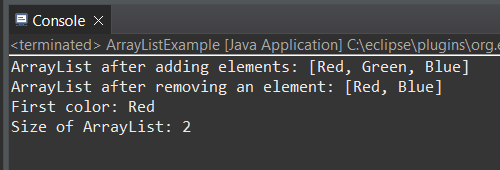
}

1. **Illustrate a java program that should implement any four ArrayList methods**

CODE:



OUTPUT:



1. **Analyze the Data class code smell, characterized by the declaration and use of only instance variables without any accompanying methods interacting with other classes. Provide a concise Java example illustrating this code smell.**

ANSWER:

A Data Class is a class that contains only instance variables and no methods, essentially making it a dumb data holder. This code smell is also known as "Anemic Domain Model" or "DTO (Data Transfer Object)".

public class Person {

public String firstName;

public String lastName;

public int age;

public String address;

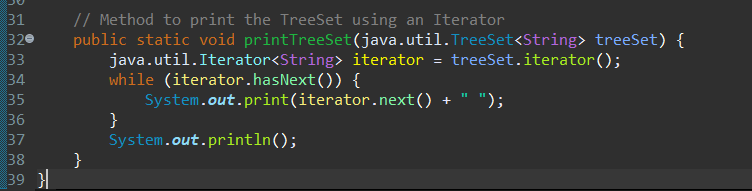
}

This code smell can lead to issues such as:

* Tight coupling between classes, as other classes need to access and manipulate the instance variables directly.
* Lack of encapsulation, making the data vulnerable to external modifications.
* Difficulty in maintaining and evolving the system, as changes to the data structure require updates in multiple places.

1. **Examine the collection class TreeSet ? Given a TreeSet containing five String objects, conduct experiments using the add() and remove() methods. Utilize the Iterator() to print the final data. Provide a detailed analysis of the outcomes, considering the behavior of the TreeSet during addition and removal operations.**

CODE:



OUTPUT:

