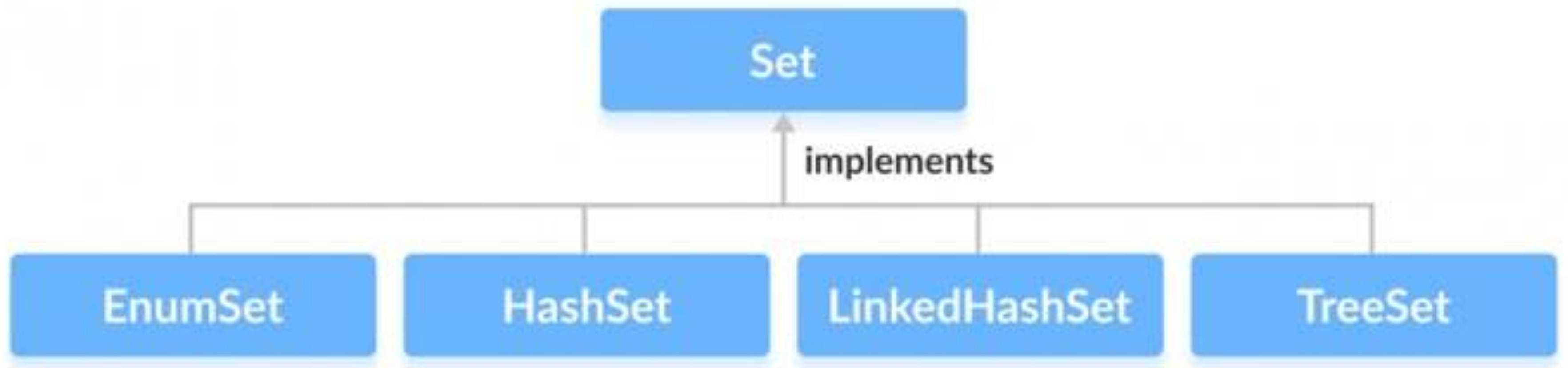
Java Set Interface

The Set interface of the Java Collections framework provides the features of the mathematical set in Java. It extends the Collection interface.

Unlike the List interface, sets cannot contain duplicate elements.



How to use Set?

In Java, we must import java.util.Set package in order to use Set.

```
// Set implementation using HashSet
Set<String> animals = new HashSet<>();
```

Here, we have created a Set called animals. We have used the HashSet class to implement the Set interface.

Set Operations

The Java Set interface allows us to perform basic mathematical set operations like union, intersection, and subset.

Union - to get the union of two sets x and y, we can use x.addAll(y)

Intersection - to get the intersection of two sets x and y, we can use x.retainAll(y)

Subset - to check if x is a subset of y, we can use y.containsAll(x)

1. Implementing HashSet Class

```
import java.util.Set;
import java.util.HashSet;

class Main {
  public static void main(String[] args) {
    // Creating a set using the HashSet class
    Set<Integer> set1 = new HashSet<>();
    // Add elements to the set1
    set1.add(2);
    set1.add(3);
    System.out.println("Set1: " + set1);
    // Creating another set using the HashSet class
    Set<Integer> set2 = new HashSet<>();
    // Add elements
    set2.add(1);
    set2.add(2);
    System.out.println("Set2: " + set2);
    // Union of two sets
    set2.addAll(set1);
    System.out.println("Union is: " + set2);
  }
}
```

2. Implementing TreeSet Class

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Iterator;

class Main {
  public static void main(String[] args) {
    // Creating a set using the TreeSet class
    Set<Integer> numbers = new TreeSet<>();
    // Add elements to the set
    numbers.add(2);
    numbers.add(3);
    numbers.add(1);
    System.out.println("Set using TreeSet: " + numbers);
    // Access Elements using iterator()
    System.out.print("Accessing elements using iterator(): ");
    Iterator<Integer> iterate = numbers.iterator();
    while(iterate.hasNext()) {
      System.out.print(iterate.next());
      System.out.print(", ");
    }
  }
}
```

Java HashSet Class

Creating a HashSet

In order to create a hash set, we must import the java.util.HashSet package first.

Once we import the package, here is how we can create hash sets in Java.

```
// HashSet with 8 capacity and 0.75 load factor
HashSet<Integer> numbers = new HashSet<>(8, 0.75);
```

Here, we have created a hash set named numbers.

```
import java.util.HashSet;

class Main {
  public static void main(String[] args) {
    HashSet<Integer> evenNumber = new HashSet<>();

    // Using add() method
    evenNumber.add(2);
    evenNumber.add(4);
    evenNumber.add(6);
    System.out.println("HashSet: " + evenNumber);

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

    // Using addAll() method
    numbers.addAll(evenNumber);
    numbers.add(5);
    System.out.println("New HashSet: " + numbers);
  }
}
```

Union of Sets

```
import java.util.HashSet;

class Main {
  public static void main(String[] args) {
    HashSet<Integer> evenNumbers = new HashSet<>();
    evenNumbers.add(2);
    evenNumbers.add(4);
    System.out.println("HashSet1: " + evenNumbers);

    HashSet<Integer> numbers = new HashSet<>();
    numbers.add(1);
    numbers.add(3);
    System.out.println("HashSet2: " + numbers);

    // Union of two set
    numbers.addAll(evenNumbers);
    System.out.println("Union is: " + numbers);
  }
}
```

Intersection of Sets

```
import java.util.HashSet;

class Main {
  public static void main(String[] args) {
    HashSet<Integer> primeNumbers = new HashSet<>();
    primeNumbers.add(2);
    primeNumbers.add(3);
    System.out.println("HashSet1: " + primeNumbers);

    HashSet<Integer> evenNumbers = new HashSet<>();
    evenNumbers.add(2);
    evenNumbers.add(4);
    System.out.println("HashSet2: " + evenNumbers);

    // Intersection of two sets
    evenNumbers.retainAll(primeNumbers);
    System.out.println("Intersection is: " + evenNumbers);
  }
}
```

Difference of Sets

```
import java.util.HashSet;

class Main {
  public static void main(String[] args) {
    HashSet<Integer> primeNumbers = new HashSet<>();
    primeNumbers.add(2);
    primeNumbers.add(3);
    primeNumbers.add(5);
    System.out.println("HashSet1: " + primeNumbers);
    HashSet<Integer> oddNumbers = new HashSet<>();
    oddNumbers.add(1);
    oddNumbers.add(3);
    oddNumbers.add(5);
    System.out.println("HashSet2: " + oddNumbers);

    // Difference between HashSet1 and HashSet2
    primeNumbers.removeAll(oddNumbers);
    System.out.println("Difference : " + primeNumbers);
  }
}
```

Subset

```java
import java.util.HashSet;

class Main {
  public static void main(String[] args) {
    HashSet<Integer> numbers = new HashSet<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    numbers.add(4);
    System.out.println("HashSet1: " + numbers);
    HashSet<Integer> primeNumbers = new HashSet<>();
    primeNumbers.add(2);
    primeNumbers.add(3);
    System.out.println("HashSet2: " + primeNumbers);

    // Check if primeNumbers is a subset of numbers
    boolean result = numbers.containsAll(primeNumbers);
    System.out.println("Is HashSet2 is subset of HashSet1? " + result);
  }
}
```

Why HashSet?

In Java, HashSet is commonly used if we have to access elements randomly. It is because elements in a hash table are accessed using hash codes.

The hashcode of an element is a unique identity that helps to identify the element in a hash table.

HashSet cannot contain duplicate elements. Hence, each hash set element has a unique hashcode.

Java EnumSet

The EnumSet class of the Java collections framework provides a set implementation of elements of a single enum.

Creating EnumSet

In order to create an enum set, we must import the java.util.EnumSet package first.

Unlike other set implementations, the enum set does not have public constructors. We must use the predefined methods to create an enum set.

1. Using allOf(Size)

The allof() method creates an enum set that contains all the values of the specified enum type Size. For example,

```
import java.util.EnumSet;
class Main {
  // an enum named Size
  enum Size {
    SMALL, MEDIUM, LARGE, EXTRALARGE
  }
  public static void main(String[] args) {
    // Creating an EnumSet using allOf()
    EnumSet<Size> sizes = EnumSet.allOf(Size.class);
    System.out.println("EnumSet: " + sizes);
  }
}
```

2. Using noneOf(Size)

The noneOf() method creates an empty enum set. For example,

```
import java.util.EnumSet;
class Main {
   // an enum type Size
   enum Size {
      SMALL, MEDIUM, LARGE, EXTRALARGE
   }
   public static void main(String[] args) {
      // Creating an EnumSet using noneOf()
      EnumSet<Size> sizes = EnumSet.noneOf(Size.class);
      System.out.println("Empty EnumSet: " + sizes);
   }
}
```

3. Using range(e1, e2) Method

The range() method creates an enum set containing all the values of an enum between e1 and e2 including both values. For example,

```
import java.util.EnumSet;
class Main {
    enum Size {
        SMALL, MEDIUM, LARGE, EXTRALARGE
    }
    public static void main(String[] args) {
        // Creating an EnumSet using range()
        EnumSet<Size> sizes = EnumSet.range(Size.MEDIUM, Size.EXTRALARGE);
        System.out.println("EnumSet: " + sizes);
    }
}
```

4. Using of() Method

The of() method creates an enum set containing the specified elements. For example,

```
import java.util.EnumSet;
class Main {
   enum Size {
      SMALL, MEDIUM, LARGE, EXTRALARGE
   }
   public static void main(String[] args) {
      // Using of() with a single parameter
      EnumSet<Size> sizes1 = EnumSet.of(Size.MEDIUM);
      System.out.println("EnumSet1: " + sizes1);
     EnumSet<Size> sizes2 = EnumSet.of(Size.SMALL, Size.LARGE);
      System.out.println("EnumSet2: " + sizes2);
   }
}
```

Why EnumSet?

The EnumSet provides an efficient way to store enum values than other set implementations (like HashSet, TreeSet).

An enum set only stores enum values of a specific enum. Hence, the JVM already knows all the possible values of the set.

This is the reason why enum sets are internally implemented as a sequence of bits. Bits specifies whether elements are present in the enum set or not.

The bit of a corresponding element is turned on if that element is present in the set.

Java LinkedHashSet

The LinkedHashSet class of the Java collections framework provides functionalities of both the hashtable and the linked list data structure.

Create a LinkedHashSet

In order to create a linked hash set, we must import the java.util.LinkedHashSet package first.

Once we import the package, here is how we can create linked hash sets in Java.

```
// LinkedHashSet with 8 capacity and 0.75 load factor
LinkedHashSet<Integer> numbers = new LinkedHashSet<>(8, 0.75);
```

Here, we have created a linked hash set named numbers.

LinkedHashSet Vs. HashSet

Both LinkedHashSet and HashSet implements the Set interface. However, there exist some differences between them.

LinkedHashSet maintains a linked list internally. Due to this, it maintains the insertion order of its elements.

The LinkedHashSet class requires more storage than HashSet. This is because LinkedHashSet maintains linked lists internally.

The performance of LinkedHashSet is slower than HashSet. It is because of linked lists present in LinkedHashSet.

LinkedHashSet Vs. TreeSet

Here are the major differences between LinkedHashSet and TreeSet:

The TreeSet class implements the SortedSet interface. That's why elements in a tree set are sorted. However, the LinkedHashSet class only maintains the insertion order of its elements.

A TreeSet is usually slower than a LinkedHashSet. It is because whenever an element is added to a TreeSet, it has to perform the sorting operation.

LinkedHashSet allows the insertion of null values. However, we cannot insert a null value to TreeSet.

Java SortedSet Interface

The SortedSet interface of the Java Collections framework is used to store elements with some order in a set.

How to use SortedSet?

To use SortedSet, we must import the java.util.SortedSet package first.

```
// SortedSet implementation by TreeSet class
SortedSet<String> animals = new TreeSet<>();
```

We have created a sorted set called animals using the TreeSet class.

Implementation of SortedSet in TreeSet Class

```
import java.util.SortedSet;
import java.util.TreeSet;

class Main {
  public static void main(String[] args) {
    // Creating SortedSet using the TreeSet
    SortedSet<Integer> numbers = new TreeSet<>();
    // Insert elements to the set
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    numbers.add(4);
    System.out.println("SortedSet: " + numbers);
    // Access the element
    int firstNumber = numbers.first();
    System.out.println("First Number: " + firstNumber);
    int lastNumber = numbers.last();
    System.out.println("Last Number: " + lastNumber);
    // Remove elements
    boolean result = numbers.remove(2);
    System.out.println("Is the number 2 removed? " + result);
  }
}
```

Java NavigableSet Interface

The NavigableSet interface of the Java Collections framework provides the features to navigate among the set elements.

How to use NavigableSet?

In Java, we must import the java.util.NavigableSet package to use NavigableSet. Once we import the package, here's how we can create navigable sets.

```
// SortedSet implementation by TreeSet class
NavigableSet<String> numbers = new TreeSet<>();
```

Here, we have created a navigable set named numbers of the TreeSet class.

Implementation of NavigableSet in TreeSet Class

```
import java.util.NavigableSet;
import java.util.TreeSet;
class Main {
  public static void main(String[] args) {
    // Creating NavigableSet using the TreeSet
    NavigableSet<Integer> numbers = new TreeSet<>();
    // Insert elements to the set
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    System.out.println("NavigableSet: " + numbers);
    // Access the first element
    int firstElement = numbers.first();
    System.out.println("First Number: " + firstElement);
    // Access the last element
    int lastElement = numbers.last();
    System.out.println("Last Element: " + lastElement);
    // Remove the first element
    int number1 = numbers.pollFirst();
    System.out.println("Removed First Element: " + number1);
    // Remove the last element
    int number2 = numbers.pollLast();
    System.out.println("Removed Last Element: " + number2);

  }
}
```

Java TreeSet

The TreeSet class of the Java collections framework provides the functionality of a tree data structure.

Creating a TreeSet

In order to create a tree set, we must import the java.util.TreeSet package first.

Once we import the package, here is how we can create a TreeSet in Java.

```
TreeSet<Integer> numbers = new TreeSet<>();
```

Here, we have created a TreeSet without any arguments. In this case, the elements in TreeSet are sorted naturally (ascending order).

TreeSet Vs. HashSet

Both the TreeSet as well as the HashSet implements the Set interface. However, there exist some differences between them.

Unlike HashSet, elements in TreeSet are stored in some order. It is because TreeSet implements the SortedSet interface as well.

TreeSet provides some methods for easy navigation. For example, first(), last(), headSet(), tailSet(), etc. It is because TreeSet also implements the NavigableSet interface.

HashSet is faster than the TreeSet for basic operations like add, remove, contains and size.

TreeSet Comparator

```java
import java.util.TreeSet;
import java.util.Comparator;
class Main {
  public static void main(String[] args) {
    // Creating a tree set with customized comparator
    TreeSet<String> animals = new TreeSet<>(new CustomComparator());
    animals.add("Dog");
    animals.add("Zebra");
    animals.add("Cat");
    animals.add("Horse");
    System.out.println("TreeSet: " + animals);
  }
  // Creating a comparator class
  public static class CustomComparator implements Comparator<String> {
    @Override
    public int compare(String animal1, String animal2) {
      int value =  animal1.compareTo(animal2);
      // elements are sorted in reverse order
      if (value > 0) {
        return -1;
      }
      else if (value < 0) {
        return 1;
      }
      else {
        return 0;
      }
    }
  }
}
```

Java Algorithms

The Java collections framework provides various algorithms that can be used to manipulate elements stored in data structures.

Algorithms in Java are static methods that can be used to perform various operations on collections.

Since algorithms can be used on various collections, these are also known as **generic algorithms**.

1. Sorting Using sort()

The sort() method provided by the collections framework is used to sort elements. For example,

```
import java.util.ArrayList;
import java.util.Collections;
class Main {
   public static void main(String[] args) {
      // Creating an array list
      ArrayList<Integer> numbers = new ArrayList<>();
      // Add elements
      numbers.add(4);
      numbers.add(2);
      numbers.add(3);
      System.out.println("Unsorted ArrayList: " + numbers);
      // Using the sort() method
      Collections.sort(numbers);
      System.out.println("Sorted ArrayList: " + numbers);
   }
}
```

2. Shuffling Using shuffle()

The shuffle() method of the Java collections framework is used to destroy any kind of order present in the data structure. It does just the opposite of the sorting. For example,

```
import java.util.ArrayList;
import java.util.Collections;
class Main {
   public static void main(String[] args) {
     // Creating an array list
      ArrayList<Integer> numbers = new ArrayList<>();
      // Add elements
      numbers.add(1);
      numbers.add(2);
      numbers.add(3);
      System.out.println("Sorted ArrayList: " + numbers);
      // Using the shuffle() method
      Collections.shuffle(numbers);
      System.out.println("ArrayList using shuffle: " + numbers);
   }
}
```

3. Routine Data Manipulation

In Java, the collections framework provides different methods that can be used to manipulate data.

- reverse() - reverses the order of elements
- fill() - replace every element in a collection with the specified value
- copy() - creates a copy of elements from the specified source to destination
- swap() - swaps the position of two elements in a collection
- addAll() - adds all the elements of a collection to other collection

Note: While performing the copy() method both the lists should be of the same size.

```
import java.util.Collections;
import java.util.ArrayList;

class Main {
  public static void main(String[] args) {
    // Creating an ArrayList
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    System.out.println("ArrayList1: " + numbers);

    // Using reverse()
    Collections.reverse(numbers);
    System.out.println("Reversed ArrayList1: " + numbers);

    // Using swap()
    Collections.swap(numbers, 0, 1);
    System.out.println("ArrayList1 using swap(): " + numbers);

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

    // Using addAll
    newNumbers.addAll(numbers);
    System.out.println("ArrayList2 using addAll(): " + newNumbers);

    // Using fill()
    Collections.fill(numbers, 0);
    System.out.println("ArrayList1 using fill(): " + numbers);

    // Using copy()
    Collections.copy(newNumbers, numbers);
    System.out.println("ArrayList2 using copy(): " + newNumbers);
  }
}
```

4. Searching Using binarySearch()

The binarySearch() method of the Java collections framework searches for the specified element. It returns the position of the element in the specified collections. For example,

```
import java.util.Collections;
import java.util.ArrayList;
class Main {
  public static void main(String[] args) {
    // Creating an ArrayList
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    // Using binarySearch()
    int pos = Collections.binarySearch(numbers, 3);
    System.out.println("The position of 3 is " + pos);
  }
}
```

Note: The collection should be sorted before performing the binarySearch() method.

5. Composition

frequency() - returns the count of the number of times an element is present in the collection

disjoint() - checks if two collections contain some common element

```
import java.util.Collections;
import java.util.ArrayList;

class Main {
  public static void main(String[] args) {
    // Creating an ArrayList
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    numbers.add(2);
    System.out.println("ArrayList1: " + numbers);
    int count = Collections.frequency(numbers, 2);
    System.out.println("Count of 2: " + count);
    ArrayList<Integer> newNumbers = new ArrayList<>();
    newNumbers.add(5);
    newNumbers.add(6);
    System.out.println("ArrayList2: " + newNumbers);
    boolean value = Collections.disjoint(numbers, newNumbers);
    System.out.println("Two lists are disjoint: " + value);
  }
}
```

6. Finding Extreme Values

The min() and max() methods of the Java collections framework are used to find the minimum and the maximum elements, respectively. For example,

```
import java.util.Collections;
import java.util.ArrayList;
class Main {
  public static void main(String[] args) {
    // Creating an ArrayList
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);
    // Using min()
    int min = Collections.min(numbers);
    System.out.println("Minimum Element: " + min);
    // Using max()
    int max = Collections.max(numbers);
    System.out.println("Maximum Element: " + max);
  }
}
```