1. 集合（Set）

Set 接口继承自 Collection 接口，这意味着 Set 接口继承了 Collection 接口的所有方法和行为。
尽管 Set 接口继承自 Collection，但它并没有添加任何新的方法或常量。它主要是对集合的行为进行了额外的约束。
Set 接口的主要特征是它不允许集合中有重复的元素。也就是说，集合中的任意两个元素 e1 和 e2 不能同时存在，使得 e1.equals(e2) 为 true。
我们可以使用 HashSet、LinkedHashSet 或 TreeSet 之一来创建一个 Set 集合。这些类都是 Set 接口的具体实现。
实现 Set 接口的具体类必须确保不能向集合中添加重复的元素。
HashSet 和 LinkedHashSet 都是基于哈希（hash）表实现的。它们使用对象的 hashCode() 方法来计算哈希值，以及使用 equals() 方法来比较对象是否相等。
TreeSet 是基于红黑树（一种自平衡的二叉搜索树）实现的。它使用元素的自然顺序（natural ordering）来维护元素。这意味着元素必须实现 Comparable 接口，或者在创建 TreeSet 时提供一个 Comparator 来定义元素的排序规则。

1.1 HashSet 类

HashSet 是 Set 接口的一个具体实现，这意味着它提供了 Set 接口定义的所有方法的实现，并且确保集合中的元素是唯一的（没有重复元素）。

1.1.1 构造函数

我们可以使用 HashSet 类的无参构造函数来创建一个空的 HashSet 实例。
代码如下。

```
Set<Srting> set = new HashSet<>();
```

第一个菱形操作符（<>）被称为类型参数或泛型类型。它指定了HashSet将存储的元素类型。
在这个例子中，HashSet被指定为存储String类型的对象。
第二个菱形操作符（<>）中，编译器会根据上下文推断泛型类型，这通常与第一个菱形操作符中指定的类型相同（仅在简单情况下）。
括号()用于调用HashSet类的构造函数。在这个例子中，调用的是无参构造函数，它创建了一个空的集合。

我们也可以使用将一个List创建为一个 HashSet。
代码如下。

```
List<String> list = Arrays.asList("Apple");
HashSet<String> hashSet = new HashSet<>(list);
```

这里第一个菱形操作符（diamond operation）内是 String， 因为列表的类型是 List<String>。而这里括号()接受列表的参数用于 HashSet 的构造函数。

下面再介绍一种构造方式，这个方法可以指定 HashSet 的初始容量，以及它还可以接受一个负载因子。负载因子的范围是0.0到1.0，它衡量了在增加容量之前，集合可以被填充的程度。当集合的大小达到初始容量乘以负载因子时，容量会翻倍。
示例代码如下。

```
// 创建一个具有指定初始容量的HashSet
int initialCapacity = 16;
HashSet<String> hashSet1 = new HashSet<>(initialCapacity);

// 创建一个具有指定初始容量和负载因子的HashSet
float loadFactor = 0.75f;
HashSet<String> hashSet2 = new HashSet<>(initialCapacity, loadFactor);
```

因此这里初始容量是16，负载因子是0.75，当集合的大小达到12（16*0.75=12）时，容量会翻倍到32（*16*2）。

1.1.2 相关方法

HashSet，LinkedHashSet，TreeSet 都实现了 Collection 接口和抽象类 AbstractSet，所以他们都可以使用add(),remove()等方法。

1.1.2.1 add() 方法

这个方法可以向集合里添加元素。
示例如下。

```
public class TestMethodsInCollection {
    public static void main(String[] args) {
        // 创建 set1
        java.util.Set<String> set1 = new java.util.HashSet<>();

        // 向 set1 添加字符串
        set1.add("London");
        set1.add("Paris");
        set1.add("New York");
        set1.add("San Francisco");
        set1.add("Beijing");

        // 打印 set1 及其大小
        System.out.println("set1 is " + set1);
        System.out.println(set1.size() + " elements in set1");
    }
}
```

如果我们尝试在代码里多添加一个重复元素，示例如下。

```
public class TestMethodsInCollection {
    public static void main(String[] args) {
        // 创建 set1
        java.util.Set<String> set1 = new java.util.HashSet<>();

        // 向 set1 添加字符串
        set1.add("London");
        set1.add("Paris");
        set1.add("New York");
        set1.add("San Francisco");
        set1.add("Beijing");
        set1.add("New York");

        // 打印 set1 及其大小
        System.out.println("set1 is " + set1);
        System.out.println(set1.size() + " elements in set1");
    }
}
```

我们会发现最后的结果不变，因为 HashSet 是非重复的，因为它通过哈希表来存储元素，哈希表的键是唯一的。当尝试添加一个已经存在的元素时，HashSet会检查该元素是否已经存在于集合中。如果存在，HashSet不会再次添加该元素，从而确保集合中的元素是非重复的。

我们可以点击 String 类名，然后按F4键，来查看 String 类的源代码。
我们可以按Ctrl + F12快速找到相关方法。
我们可以发现 HashSet 利用 String 类的 equals() 和 hashCode() 方法来检查重复元素。

因此我们可以尝试下列代码。

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

public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + '}';
    }
}
public class PersonTest {
    public static void main(String[] args) {
        // 在另一个类中使用 Person 类
        Set<Person> set1 = new HashSet<>();
        set1.add(new Person("John", 19));
        set1.add(new Person("Mary", 20));
        set1.add(new Person("John", 19));
        System.out.println(set1);
    }
}
```

由于 HashSet 是基于哈希表实现的，它使用对象的 hashCode() 和 equals() 方法来确定对象的唯一性。在Java中，如果没有重写这些方法，hashCode() 默认返回对象的内存地址的哈希码，而 equals() 默认只比较对象的引用。
在这个例子中，尽管有两个 Person 对象具有相同的 name 和 age，但由于没有重写 hashCode() 和 equals() 方法，这两个对象被视为不同的对象。因此，HashSet 会将它们都添加到集合中。
下面的代码就指挥打印出两个结果。

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

public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
```

```
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + '}';
    }
}
public class PersonTest {
    public static void main(String[] args) {
        // 在另一个类中使用 Person 类
        Set<Person> set1 = new HashSet<>();
        set1.add(new Person("John", 19));
        set1.add(new Person("Mary", 20));
        set1.add(new Person("John", 19));
        System.out.println(set1);
    }
}
```

1.1.2.2 增强for循环（for-each循环）

Collection 接口扩展了 Iterable 接口，这意味着实现了 Collection 接口的类（如 HashSet）也实现了 Iterable 接口。
因此我们可以使用增强for循环遍历集合中的每个元素。
示例如下。

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

public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");

        // 使用增强型for循环遍历HashSet
        for (String fruit: set) {
            System.out.println(fruit.toLowerCase());
        }
    }
}
```

HashSet是无序的，这意味着它不保证元素的存储顺序，并且没有像数组那样的索引（即你不能通过下标[i]来访问元素）。增强型for循环允许你方便地遍历HashSet中的所有元素，而不需要关心元素的存储顺序或使用索引。

1.1.2.3 forEach() 方法

我们也可以使用 forEach() 方法循环遍历集合中的每个元素。
forEach()是Iterable接口中的一个默认方法，用于对集合中的每个元素执行指定的操作。
语法为Set.forEach(Consumer<? super E> action)。
常用方法为set.forEach(e -> System.out.print())。
这里e是传递给lambda表达式的参数，表示当前集合中的一个元素。
->是lambda表达式中的箭头，用于分隔参数和函数体。
因此代码如下。

```
set.forEach(e -> System.out.print(e.toLowerCase() + " "));
```

1.1.2.4 其他常用方法

下面的代码展示了其他常用方法的使用。

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

public class HashSetDemo {
    public static void main(String[] args) {
        Set<String> set1 = new HashSet<>();
        set1.add("London");
        set1.add("Paris");
        set1.add("New York");
        set1.add("San Francisco");
        set1.add("Beijing");

        Set<String> set2 = new HashSet<>();
        set2.add("New York");
        set2.add("London");
        set2.add("Taipei");

        // remove(): Delete a string from set1
        set1.remove("London");
        System.out.println("\nset1 is " + set1);

        // size(): The size of the set
        System.out.println(set1.size() + " elements in set1");

        // contains(): If the set contains a certain element, return T/F
        System.out.println("\nIs Taipei in set2? " + set2.contains("Taipei"));

        // addAll(): add the elements in set1 and set2 together. NO DUPLICATION!
        // hashCode() and equals() are called
        set1.addAll(set2);
        System.out.println("\nAfter adding set2 to set1, set1 is " + set1);

        // removeAll(): removing the elements in set 2 from set1
        set1.removeAll(set2);
        System.out.println("\nAfter removing set2 from set1, set1 is " + set1);

        // retainAll(): What is the printed result in this case?
        set1.retainAll(set2);
```

```
        System.out.println("\nAfter retaining common elements in set1 and set2,
set1 is " + set1);
    }
}
```

这里只稍微说明一下这里的retainAll()方法是保留set1中与set2共有的元素，但是前一步set1移除了所有在set2中存在的元素，所以最后一行运行的结果没有任何元素。

1.2 LinkedHastSet 类

LinkedHashSet 是 HashSet 的一个子类，它继承了 HashSet 的特性，并添加了一个链表实现以支持元素的顺序。
LinkedHashSet 维护元素的插入顺序，即元素可以按照它们被添加到集合中的顺序被检索和迭代。
LinkedHashSet 在很多方面与 HashSet 非常相似。之前提到的 HashSet 的创建方式和方法（如添加元素、删除元素等）同样适用于 LinkedHashSet。
LinkedHashSet 和 HashSet 的一个显著区别是，LinkedHashSet 中的元素可以按照它们被插入的顺序被检索。这意味着如果你需要保持元素的添加顺序，LinkedHashSet 是一个比 HashSet 更好的选择。
示例代码如下。

```
import java.util.*;

public class TestLinkedHashSet {
    public static void main(String[] args) {
        // Create a linked hash set
        Set<String> set = new LinkedHashSet<>();

        // Add strings to the set
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Beijing");
        set.add("New York");

        System.out.println(set);

        // Display the elements in the hash set
        for (String element : set) {
            System.out.print(element.toLowerCase() + " ");
        }
    }
}
```

这里打印出来的结果与添加的顺序一致。

1.3 TreeSet 类

TreeSet 是一个具体的类，它实现了 SortedSet 和 NavigableSet 接口。
SortedSet 是 Set 接口的一个子接口，它保证了集合中的元素是有序的。
这意味着当你遍历 SortedSet 时，元素会按照某种排序顺序（通常是自然排序或构造时指定的比较器）返回。
NavigableSet 接口扩展了 SortedSet 接口，提供了导航方法，用于在有序集合中查找特定元素或一组元素。

这些导航方法包括：
lower(E e)：返回集合中小于给定元素的最大元素。
floor(E e)：返回集合中小于或等于给定元素的最大元素。
higher(E e)：返回集合中大于给定元素的最小元素。
ceiling(E e)：返回集合中大于或等于给定元素的最小元素。

1.3.1 构造方法

我们可以创建一个空的 TreeSet，它将根据其元素的自然顺序（natural ordering）进行升序排序。
示例代码如下。

```
TreeSet<String> treeSet = new TreeSet<>();
```

由于 TreeSet 实现了 SortedSet 接口，它会自动对元素进行排序。
我们也可以从其他集合（如 List）创建一个 TreeSet，新集合将根据元素的自然顺序进行排序。

```
TreeSet<String> treeSet = new TreeSet<>(list);
```

我们也可以使用自定义比较器创建一个 TreeSet，以定义元素的排序顺序。

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

如果我们已经有一个根据特定规则排序的 SortedSet，你可以创建一个新的 TreeSet，它包含相同的元素并保持相同的排序顺序。

相关的UML图如下。

1.3.2 相关方法

1.3.2.1 add() 方法

类似于 HashSet，TreeSet 也不会添加重复的元素。但TreeSet 不使用 hashCode() 和 equals() 方法来检查重复元素，而是使用 Java 标准库中 String 和 Integer 等包装类提供的内置 compareTo() 方法。
HashSet 使用哈希表（Hashing）作为底层数据结构，而 TreeSet 使用树（Tree）结构，具体来说是红黑树。
这种底层数据结构的不同导致了 HashSet 和 TreeSet 在处理元素时的不同行为，尤其是在元素排序和查找方面。
示例代码如下。

```
import java.util.TreeSet;

public class TreeSetTest {
    public static void main(String[] args) {
        // 创建一个树形集合
        TreeSet<String> treeSet = new TreeSet<>();

        // 添加元素
        treeSet.add("Apple");
        treeSet.add("Banana");
        treeSet.add("Cherry");
        treeSet.add("Apple");  // 重复元素，不会被添加
        treeSet.add("Date");
```

```
        treeSet.add("Banana");  // 重复元素，不会被添加

        // 打印集合内容
        System.out.println("TreeSet contents: " + treeSet);
    }
}
```

1.3.2.2 其他常用方法

作为 SortedSet ， 其的常用方法如下：

1. first() 方法：
   返回集合中的第一个元素，即按照排序顺序最小的元素。
2. last() 方法：
   返回集合中的最后一个元素，即按照排序顺序最大的元素。
3. headSet() 方法：
   返回一个视图（原始集合的一个子集），包含所有小于指定元素的集合元素。
4. tailSet() 方法：
   返回一个视图（原始集合的一个子集），包含所有大于（或等于）指定元素的集合元素。

示例代码如下。

```
import java.util.*;

public class TestTreeSet {
    public static void main(String[] args) {
        // 创建一个树形集合
        TreeSet<String> set = new TreeSet<>();
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Beijing");

        // 打印排序后的树形集合
        System.out.println("Sorted tree set: " + set);

        // 使用 SortedSet 接口中的方法
        System.out.println("first(): " + set.first());
        System.out.println("last(): " + set.last());
        System.out.println("headSet(\"New York\"): " + set.headSet("New York"));
        System.out.println("tailSet(\"New York\"): " + set.tailSet("New York"));
    }
}
```

这里给出一个更简单的例子。

```
TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9));
SortedSet<Integer> headSet = numbers.headSet(5);
```

在这个例子中，headSet 将包含所有小于 5 的元素，即 [1, 2, 3, 4]。

作为 NavigableSet ， 其的常用方法如下：

1. lower():
   返回集合中小于给定元素的最大元素。
2. higher():
   返回集合中大于给定元素的最小元素。
3. floor():
   返回集合中小于或等于给定元素的最大元素。
4. ceiling():
   返回集合中大于或等于给定元素的最小元素。
5. pollFirst():
   检索并移除集合中的第一个（最小）元素。
6. pollLast():
   检索并移除集合中的最后一个（最大）元素。

示例代码如下。

```
TreeSet<Integer> treeSet1 = new TreeSet<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9));

// 使用 NavigableSet 接口中的方法
System.out.println("lower(5): " + treeSet1.lower(5));
System.out.println("higher(5): " + treeSet1.higher(5));
System.out.println("floor(5): " + treeSet1.floor(5));
System.out.println("ceiling(5): " + treeSet1.ceiling(5));
System.out.println("pollFirst(): " + treeSet1.pollFirst());
System.out.println("pollLast(): " + treeSet1.pollLast());
System.out.println("New tree set: " + treeSet1);
```

1.4 Set 和 List 的性能对比

```
import java.util.*;

public class SetListPerformanceTest {
    static final int N = 50000;

    public static long getTestTime(Collection<Integer> c) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < N; i++)
            c.contains((int)(Math.random() * 2 * N));
        return System.currentTimeMillis() - startTime;
    }

    public static long getRemoveTime(Collection<Integer> c) {
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < N; i++)
            c.remove(i);
        return System.currentTimeMillis() - startTime;
    }

    public static void main(String[] args) {
        // Add numbers 0, 1, 2, ..., N - 1 to an array list
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < N; i++)
            list.add(i);
```

```
        Collections.shuffle(list); // Shuffle the array list

        // Create a hash set, and test its performance
        Collection<Integer> set1 = new HashSet<>(list);
        System.out.println("Member test time for hash set is " +
getTestTime(set1) + " milliseconds");
        System.out.println("Remove element time for hash set is " +
getRemoveTime(set1) + " milliseconds");

        // Create a linked hash set, and test its performance
        Collection<Integer> set2 = new LinkedHashSet<>(list);
        System.out.println("Member test time for linked hash set is " +
getTestTime(set2) + " milliseconds");
        System.out.println("Remove element time for linked hash set is " +
getRemoveTime(set2) + " milliseconds");

        // Create a tree set, and test its performance
        Collection<Integer> set3 = new TreeSet<>(list);
        System.out.println("Member test time for tree set is " +
getTestTime(set3) + " milliseconds");
        System.out.println("Remove element time for tree set is " +
getRemoveTime(set3) + " milliseconds");

        // Create an array list, and test its performance
        Collection<Integer> list1 = new ArrayList<>(list);
        System.out.println("Member test time for array list is " +
getTestTime(list1) + " milliseconds");
        System.out.println("Remove element time for array list is " +
getRemoveTime(list1) + " milliseconds");

        // Create a linked list, and test its performance
        Collection<Integer> list2 = new LinkedList<>(list);
        System.out.println("Member test time for linked list is " +
getTestTime(list2) + " milliseconds");
        System.out.println("Remove element time for linked list is " +
getRemoveTime(list2) + " milliseconds");
    }
}
```

在我的ide上运行的结果如下。

HashSet 和 LinkedHashSet 在查找和删除操作上比 TreeSet 更快，因为它们使用哈希表实现，而 TreeSet 使用红黑树实现，这需要额外的时间开销。同样，ArrayList 和 LinkedList 在查找操作上也有所不同，ArrayList 使用数组实现，查找时间复杂度为 O(n)，而 LinkedList 使用链表实现，查找时间复杂度也为 O(n)，但删除操作的时间复杂度不同。

因此综合来看，Set 比 List 更适合存储不重复的元素。List 允许通过索引访问元素。Set 不支持索引访问，因为 Set 中的元素是无序的。要遍历 Set 中的所有元素，可以使用增强型for循环（for-each loop）或迭代器（iterator）。

1.5 总结

HashSet、LinkedHashSet 和 TreeSet 都是Java中 Set 接口的实现，这意味着它们都不允许存储重复元素。

1. HashSet：
   排序：它不保证迭代的顺序。
   内部结构：由哈希表支持。
2. LinkedHashSet：
   排序：维护一个贯穿所有条目的双向链表，这定义了迭代的顺序，通常是元素被插入到集合中的顺序（插入顺序）。
   内部结构：由哈希表支持，并通过一个链表贯穿。
3. TreeSet：
   排序：保证元素将按照元素的自然顺序或在集合创建时提供的比较器进行升序排序。
   内部结构：由树（通常是红黑树）支持。

2. 映射（Map）

与 Set 类似， Java提供了三种主要的 Map 实现：HashMap、LinkedHashMap 和 TreeMap。
这些 Map 实现需要确保键值对（key-value pairs）的唯一性。
HashMap 和 LinkedHashMap 使用 hashCode() 和 equals() 方法来检查键的唯一性。
TreeMap 使用 compareTo() 方法或在创建时提供的 Comparator 来检查键的唯一性。
TreeMap 除了实现 Map 接口外，还实现了 SortedMap 和 NavigableMap 接口。
下图展示了相关的UML图。

2.1 构造方法

可以使用泛型参数创建新的 HashMap、LinkedHashMap 或 TreeMap 实例。
构造函数的形式为Map<? extends K, ? extends V> 的参数，其中 K 是键的类型，V 是值的类型。
示例代码如下。

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // 创建一个包含整数键和字符串值的 Map
        Map<Integer, String> smallMap = new HashMap<>();
        smallMap.put(10, "Ten");
        smallMap.put(20, "Twenty");

        // 使用接受另一个 Map 的构造函数创建一个新的 HashMap
        Map<Number, Object> largerMap = new HashMap<>(smallMap);

        System.out.println("Contents of largerMap: " + largerMap);
    }
}
```

这里接受的参数中的键值对类型是Integer, String，它们分别是Number, Object的子类，因此可以。
当然也可以就是Integer, String。
如下。

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
```

```
        // 创建一个包含整数键和字符串值的 Map
        Map<Integer, String> smallMap = new HashMap<>();
        smallMap.put(10, "Ten");
        smallMap.put(20, "Twenty");

        // 使用接受另一个 Map 的构造函数创建一个新的 HashMap
        Map<Integer, String> largerMap = new HashMap<>(smallMap);

        System.out.println("Contents of largerMap: " + largerMap);
    }
}
```

LinkedHashMap 是 HashMap 的一个扩展，它使用链表实现，支持按插入顺序检索元素。
LinkedHashMap 的构造函数有三个参数：
initialCapacity：初始容量，整数类型。
loadFactor：负载因子，浮点数类型。
accessOrder：访问顺序，布尔类型。如果设置为 true，则 LinkedHashMap 会按照元素最后被访问的顺序排列元素，这意味着在每次访问（读或写）元素后，该元素会被移动到链表的末尾，从而使得最近访问的元素位于链表的尾部。如果设置为 false（默认值），LinkedHashMap 会按照元素的插入顺序来排列元素。即元素会按照它们被添加到映射中的顺序进行排序。

```
import java.util.*;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        // 创建一个 LinkedHashMap
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>(16,
0.75f,true);
        linkedHashMap.put("Smith", 30);
        linkedHashMap.put("Anderson", 31);
        linkedHashMap.put("Lewis", 29);
        linkedHashMap.put("Cook", 29);

        // 在访问任何元素之前显示地图
        System.out.println("\nDisplay before any access");
        System.out.println(linkedHashMap);

        // 访问 Lewis 以获取他的年龄
        System.out.println("\nThe age for Lewis is " +
linkedHashMap.get("Lewis"));

        // 在访问一个元素之后显示地图
        System.out.println("After an element is accessed the entries in
LinkedHashMap are\n");
        System.out.println(linkedHashMap);
    }
}
```

比如这里本来的顺序是插入的顺序，但是我们中途访问了 Lewis，所以导致现在 Lewis 出现在了最后。

2.2 相关方法

HashMap 是无序的，类似于 HashSet。
TreeMap 是有序的，按照键的自然顺序或指定的比较器进行排序。
LinkedHashMap 可以按插入顺序排序，也可以按访问顺序（accessOrder: true）排序。
常用方法如下：

1. get()：返回与指定键关联的值。
2. forEach()：对 Map 中的每个条目执行给定的操作，直到所有条目都被处理或操作抛出异常。用法和前面的Set的类似。

```
import java.util.*;

public class TestMap {
    public static void main(String[] args) {
        // 创建一个 HashMap
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("Smith", 30);
        hashMap.put("Anderson", 31);
        hashMap.put("Lewis", 29);
        hashMap.put("Cook", 29);
        System.out.println("Display entries in HashMap: " + hashMap);

        // 从 HashMap 创建一个 TreeMap
        Map<String, Integer> treeMap = new TreeMap<>(hashMap);
        System.out.println("Display entries in ascending order of key:");
        System.out.println(treeMap);

        // 创建一个 LinkedHashMap
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>(10, 0.75f,
true);
        linkedHashMap.put("Smith", 30);
        linkedHashMap.put("Anderson", 31);
        linkedHashMap.put("Lewis", 29);
        linkedHashMap.put("Cook", 29);

        // 在访问任何元素之前显示映射
        System.out.println("\nDisplay before any access");
        System.out.println(linkedHashMap);

        // 访问 Lewis 以获取他的年龄
        System.out.println("\nThe age for Lewis is " +
linkedHashMap.get("Lewis"));

        // 在访问一个元素之后显示映射
        System.out.println("After an element is accessed the entries in
LinkedHashMap are\n");
        System.out.println(linkedHashMap);

        // 使用 forEach 显示每个条目，带有名称和年龄
        System.out.println("\nNames and ages are ");
        treeMap.forEach((name, age) -> System.out.println(name + ": " + age + "
"));
    }
}
```

下面的代码展示了HashMap里的一些方法。

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.Collection;

public class HashMaps4Fun {
    public static void main(String[] args) {
        // 创建一个新的 HashMap
        Map<String, Integer> hashMap = new HashMap<>();

        // 向 map 中放入一些键值对
        hashMap.put("One", 1);
        hashMap.put("Two", 2);
        hashMap.put("Three", 3);

        // 测试 containsKey 方法
        System.out.println("Does hashMap contain 'Two'? " +
hashMap.containsKey("Two"));

        // 测试 containsValue 方法
        System.out.println("Does hashMap contain value '3'? " +
hashMap.containsValue(3));

        // 测试 entrySet 方法
        Set<Map.Entry<String, Integer>> entries = hashMap.entrySet();
        System.out.println("Entry set: " + entries);
    }
}
```

下面的代码展示了TreeMap里的一些方法。

```
import java.util.TreeMap;

public class TreeMaps4Fun {
    public static void main(String[] args) {
        // 创建一个 TreeMap 并添加一些条目
        TreeMap<String, Integer> treeMap = new TreeMap<>();
        treeMap.put("Apple", 3);
        treeMap.put("Banana", 5);
        treeMap.put("Date", 4);
        treeMap.put("Elderberry", 1);

        // 测试 SortedMap 方法
        System.out.println("First key: " + treeMap.firstKey());
        System.out.println("Last key: " + treeMap.lastKey());
        System.out.println("HeadMap (keys less than 'Date'): " +
treeMap.headMap("Date"));
        System.out.println("TailMap (keys greater than or equal to 'Date'): " +
treeMap.tailMap("Date"));

        // 测试 NavigableMap 方法
        System.out.println("Lower key than 'Cherry': " +
treeMap.lowerEntry("Cherry"));
```

```
        System.out.println("Floor key of 'Cherry': " +
treeMap.floorEntry("Cherry"));
        System.out.println("Ceiling key of 'Cherry': " +
treeMap.ceilingEntry("Cherry"));
        System.out.println("Higher key than 'Cherry': " +
treeMap.higherEntry("Cherry"));

        // 轮询条目
        System.out.println("Poll first entry: " + treeMap.pollFirstEntry());
        System.out.println("Poll last entry: " + treeMap.pollLastEntry());
        System.out.println("TreeMap after polling: " + treeMap);
    }
}
```

3. 练习

3.1 基础练习

假设我们有两个集合 s1 = [1, 2, 5], s2 = [2, 3, 6]。
运行s1.addAll(s2) 之后，s1变为[1, 2, 5, 3, 6]，而s2不变，仍为[2, 3, 6]。
运行s1.removeAll(s2) 之后，s1变为[1, 5]。
运行s1.retainAll(s2) 之后，s1变为[]。

3.2 进阶练习

3.2.1 获取TreeSet的第一个元素和最后一个元素

示例代码如下。

```
import java.util.*;

public class Test {
    public static void main(String[] args) throws Exception {
        TreeSet<String> set = new TreeSet<>();
        set.add("Red");
        set.add("Yellow");
        set.add("Green");
        set.add("Blue");
        System.out.println(set.first());
        System.out.println(set.last());
    }
}
```

3.2.2 深拷贝版 MyStack 类

MyStack类代码如下。

```
import java.util.ArrayList;

public class MyStack {
    private ArrayList list = new ArrayList();

    public void push(Object o) {
```

```
        list.add(o);
    }

    public Object pop() {
        Object o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public Object peek() {
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }
}
```

修改后的示例如下。

```
import java.util.ArrayList;
import java.util.List;

public class MyStack implements Cloneable {
    private List<Object> list = new ArrayList<>();

    public void push(Object o) {
        list.add(o);
    }

    public Object pop() {
        Object o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public Object peek() {
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }
```

```
    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }

    // 实现 Cloneable 接口的 clone 方法，进行深拷贝
    @Override
    public MyStack clone() throws CloneNotSupportedException {
        MyStack clonedStack = (MyStack) super.clone();
        clonedStack.list = new ArrayList<>(list); // 创建 list 的一个新副本
        return clonedStack;
    }
}
```

测试代码如下。

```
public class DeepCopyTest {
    public static void main(String[] args) {
        MyStack originalStack = new MyStack();
        originalStack.push("Apple");
        originalStack.push("Banana");
        originalStack.push("Cherry");

        try {
            MyStack clonedStack = originalStack.clone();
            System.out.println("Original stack: " + originalStack);
            System.out.println("Cloned stack: " + clonedStack);

            // 修改原始栈
            originalStack.push("Date");

            // 显示修改后的结果
            System.out.println("Modified original stack: " + originalStack);
            System.out.println("Cloned stack remains unchanged: " +
clonedStack);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

3.2.3 深拷贝版 MyStack 类

原来的MyStack代码如下。

```
在这里插入代码片
```

深拷贝版如下。

```
import java.util.ArrayList;

public class MyStack {
    private ArrayList list = new ArrayList();

    public void push(Object o) {
       list.add(o);
   }

   public Object pop() {
       Object o = list.get(getSize() - 1);
       list.remove(getSize() - 1);
       return o;
   }

   public Object peek() {
       return list.get(getSize() - 1);
   }

   public int search(Object o) {
       return list.lastIndexOf(o);
   }

   public boolean isEmpty() {
       return list.isEmpty();
   }

   public int getSize() {
       return list.size();
   }

   public String toString() {
       return "stack: " + list.toString();
   }
}
```

3.2.4 找出出现次数最多的整数

编写一个程序来读取一系列整数，并找出出现次数最多的整数。输入以0结束。如果多个整数出现次数最多，则应该报告所有这些整数。
这个代码与之前大二的Java练习类似，使用一个 Map 来跟踪每个整数的出现次数。
示例代码如下。

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

```java
public class MostFrequentNumber {
    public static void main(String[] args) {
        Map<Integer, Integer> countMap = new HashMap<>();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            int num = scanner.nextInt();
            if (num == 0) {
                break;
            }
            if (countMap.containsKey(num)) {
                countMap.put(num, countMap.get(num) + 1);
            } else {
                countMap.put(num, 1);
            }
        }

        int maxCount = 0;
        for (int value : countMap.values()) {
            if (value > maxCount) {
                maxCount = value;
            }
        }

        System.out.println("Numbers with most occurrences:");
        for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {
            if (entry.getValue() == maxCount) {
                System.out.println(entry.getKey());
            }
        }
    }
}
```